SoC Software Lab Instructions

Version 14.1	02/27/2015	Tutorial

Table of Contents

OVERVIEW

The **Altera SoC** combines a **Hard Processing System** (HPS) and an **FPGA** on a **single device**. The HPS has dual core **ARM Cortex-A9 MPU**s and a host of peripherals such as DDR3 controllers, Ethernet MACs, SPI controllers and many more. The FPGA portion of the device is tightly coupled through **high performance bridges** to the HPS. The designer can add peripherals they create or third party IP to the FPGA and map it into the HPS. **Thus you have a flexible and very powerful solution**.

This software lab aims to answer to following questions that a developer might have:

How do I build and debug software to boot my custom HPS configuration?

How do I map the FPGA peripherals into the HPS memory map?

How do I address the individual registers within these peripherals?

How does my host **OS** know which **peripherals** have been added and **which device drivers to load**?

The HPS is **configured** using **Qsys**, Altera's FPGA IP integration tool. Configuration includes **selecting DDR memory**, determining **clock frequencies** and selecting which **HPS peripherals** your design will use. As such Qsys inherently has most of the information to satisfy the questions asked above. Quartus is also used to define the **HPS peripheral pin outs**.

These two Altera FPGA development tools will generate the **files** needed for the **transfer of design information** from the **hardware** to the **software** domain. A significant portion of the software modules will use these handoff files to build a preloader, to examine the system register set (including FPGA registers) and lastly to follow the path of the **Device Tree** from the .**sopcinfo** file to the **Device Drivers in Linux**.

Module Summary:

The Software labs are based on the **Golden Hardware Reference Design** (GHRD) that is provided with the **SoCKit**. You will examine the **architecture** of the GHRD in **Module 2**.

In **Module 3** you will learn how to create, build and run a custom **preloader** that will be used to boot a high level operating system.

In **Module 4** you will see how to incrementally **validate** the **peripherals** created in the FPGA. First you will use the extended HPS **register** set (including those from FPGA peripherals) to read and write to those FPGA peripherals from the **DS-5 debugger**. Then you will see how to access them from a **Linux application** and finally how to address them from **Linux device drivers**.

Module 6 is a bonus lab that shows how to **cross trigger** during debug between the **CPU** and **FPGA** domains.

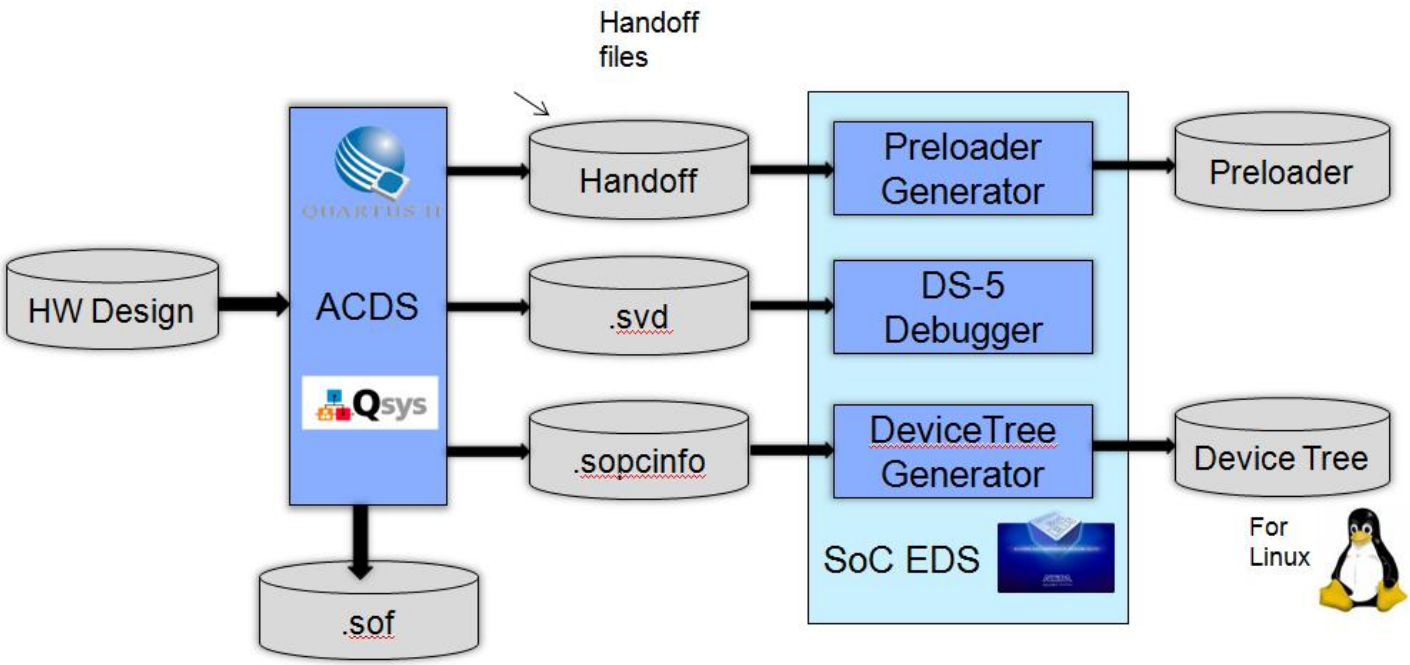
Hardware to software domain transfer:

The **diagram** below shows three main areas of **transfer** from the **hardware** to **software** domains.

1. The **files** necessary to create a custom **preloader**

2. The .**svd** file that describes the FPGA **peripherals** and is used by the DS-5 **register** function

3. The **sopcinfo** file that describes all of the HPS **devices** selected in Qsys and those custom peripherals added in the FPGA. These are used to build a **device tree**. The device tree is used by the Linux kernel to determine which device drivers to load at boot time.



MODULE 1: Getting Started

Your first objective is to ensure that you have all of the items needed and to install the tools so that you are ready to create and run your design.

List of Required Items:

- Arrow Electronics **SoCKit** evaluation board
- Quartus II v14.1 Stand-alone **Programmer**
- Altera **SoC EDS** v14.1
- **PuTTY** terminal emulator
- Computer with Windows 7, 4 GB RAM, minimum of I3 core and over 10 GB free hard disk space for the Quartus II install
- **Lab Design Files**

1.1 Acquiring the Arrow SoCKit

To order a SoCKit please click on the link below

Order an SoCKit from Arrow Electronics

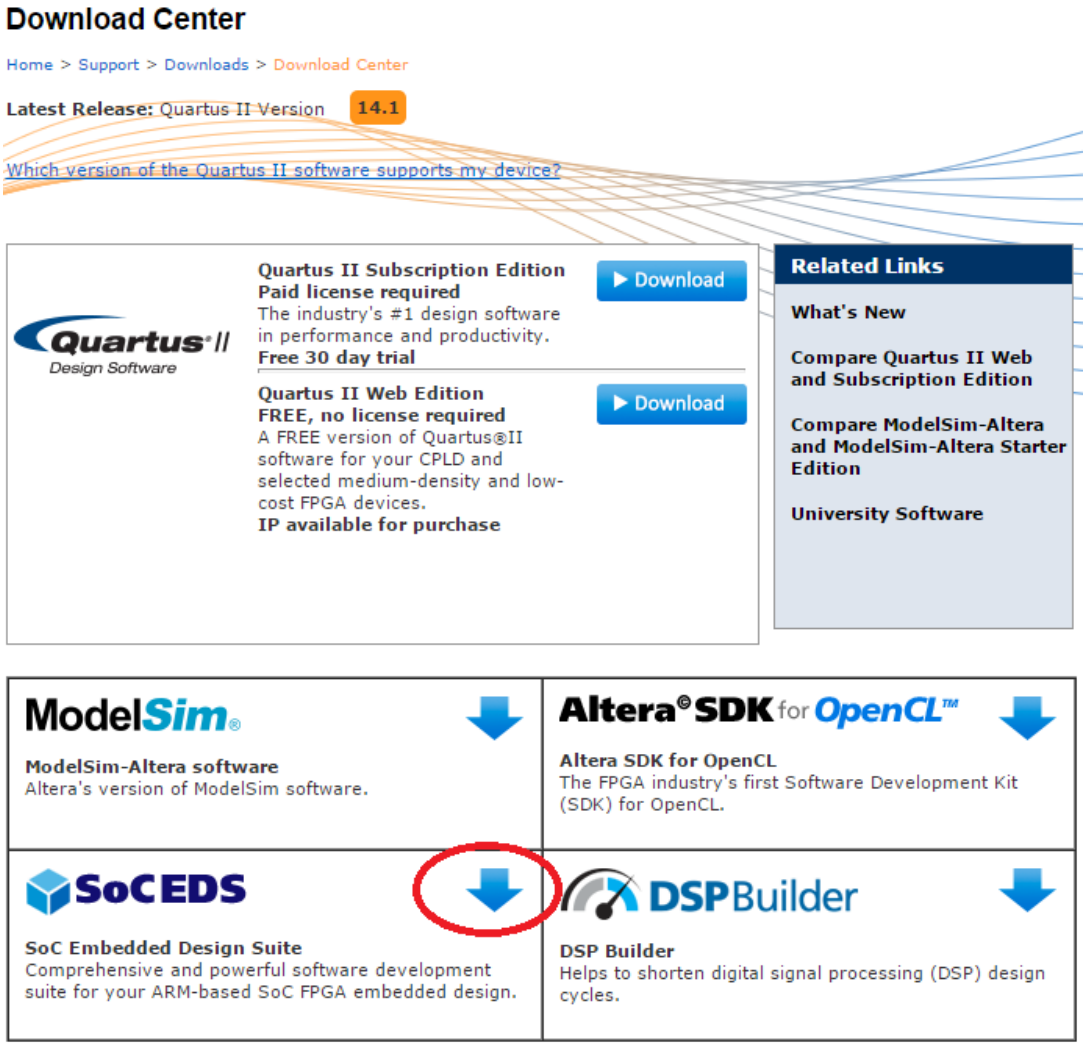
1.2 Download the Altera Design Software

You will need to install the following design software packages:

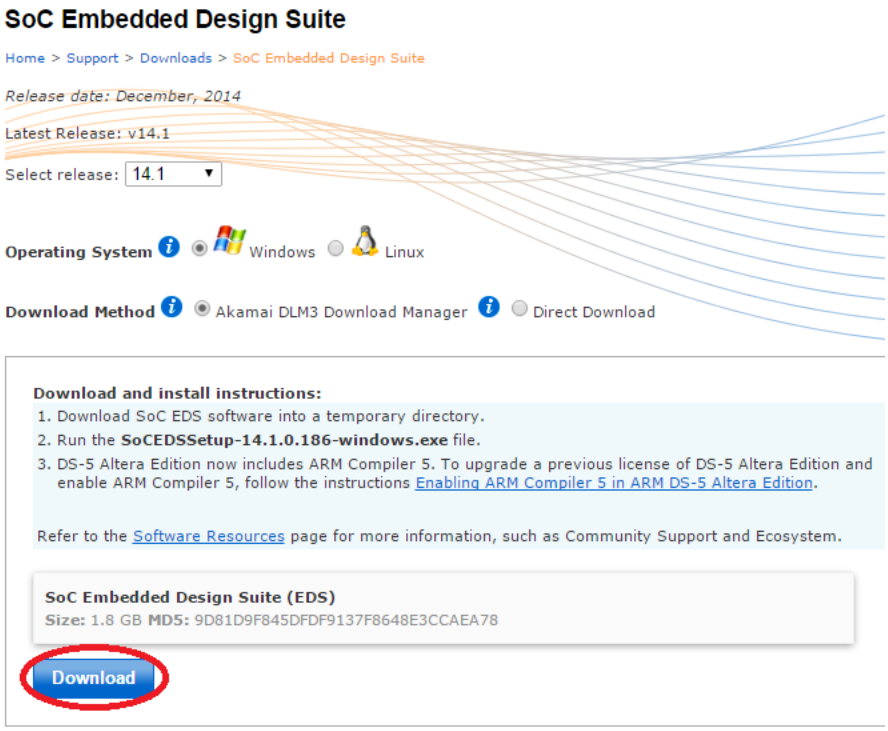
- SoC Embedded Design Suite (EDS) v14.1

The **Programming Software** can be **downloaded** from the Altera web site.

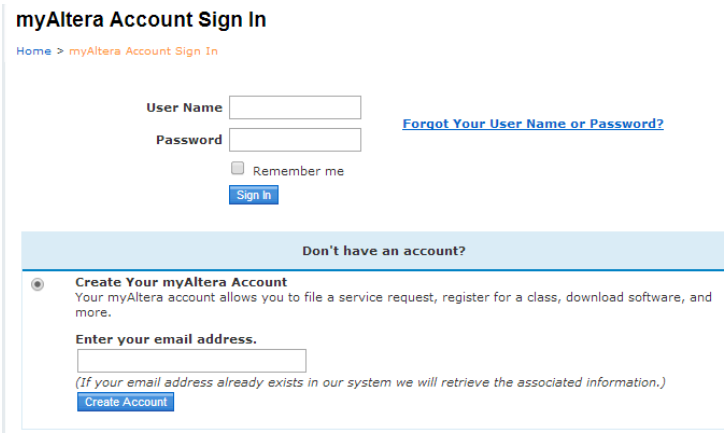
- Go to the **Altera Download web page** at **https://www.altera.com/download/dnl-index.jsp**
- Select the **Download** button next to the **SoCEDS**



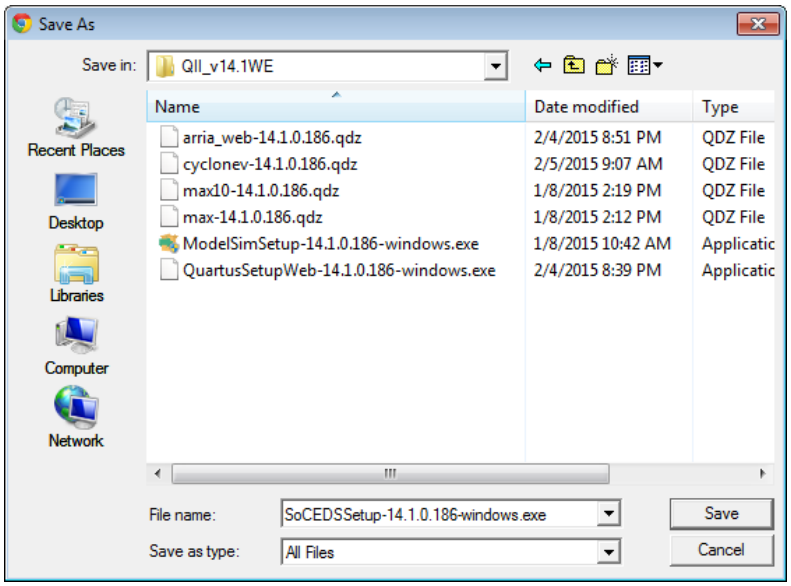
- Press the **Download** button.



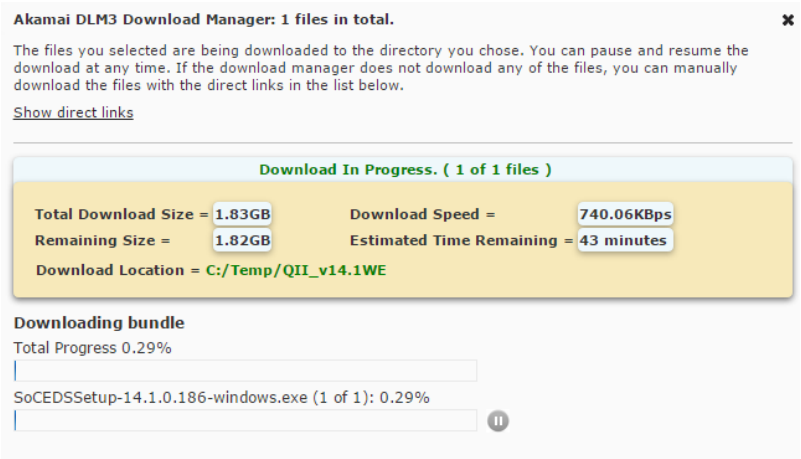
- **Login** to myAltera account at https://www.altera.com/myaltera/mal-index.jsp
- Use your **existing login**, or **Create Your myAltera account.**



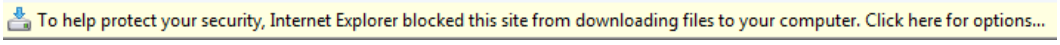
- Select a location.



- The **next page** of the installation will look like:
- **Verify** the **selections** shown below.
- The download of the selected files will begin once you have chosen a folder to save them in.



- If you are using Internet Explorer it may block the download. Click the options bar to allow the download

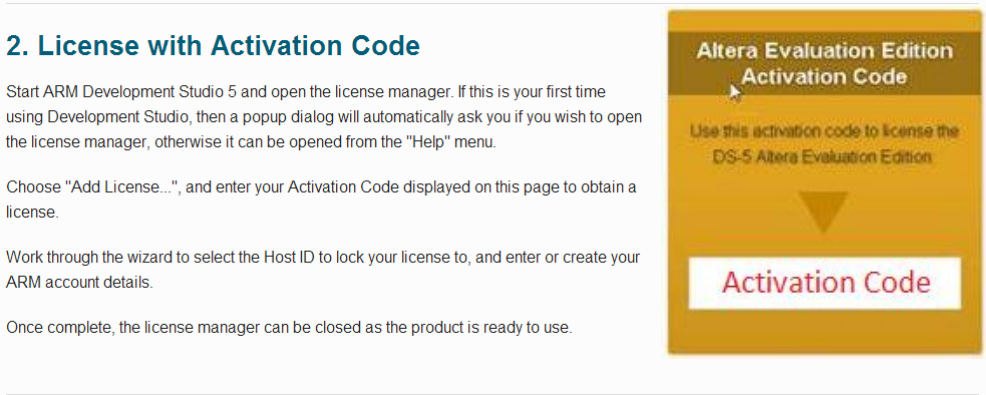


1.3 Install the Altera Design Software

- Obtain a **30-day evaluation license for SoC EDS** Subscription Edition by **clicking the activation code link below.**

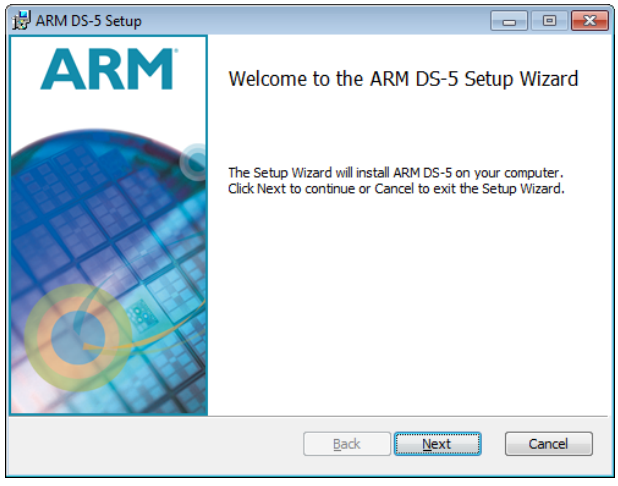
  http://ds.arm.com/altera/altera-eval-edition/

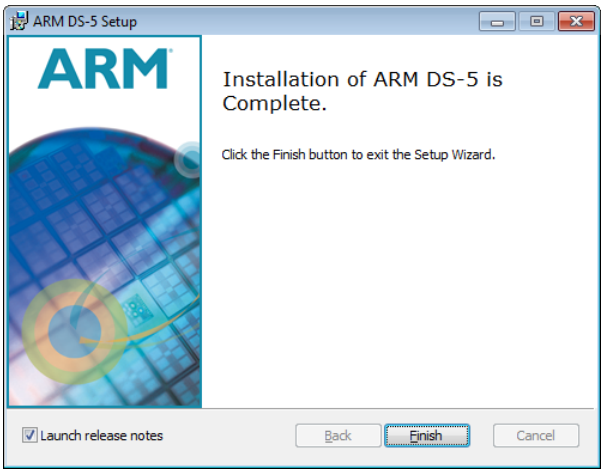
- You will be provided with an **activation** code. Use this code when prompted by the **ARM licensing manager**.



- **Start the SoC EDS Installation**. Double Click the **SoCEDSSetup-14.1.0.186-windows.exe** file that was downloaded.
- **Accept** the license agreement and use all the default **settings and locations** for installation

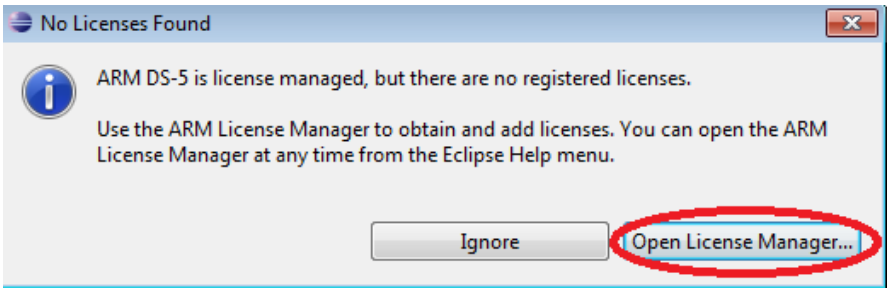
- **Install the drivers and DS-5**. Use all the defaults. **If you are notified that you should restart Windows, please ignore this and continue.**



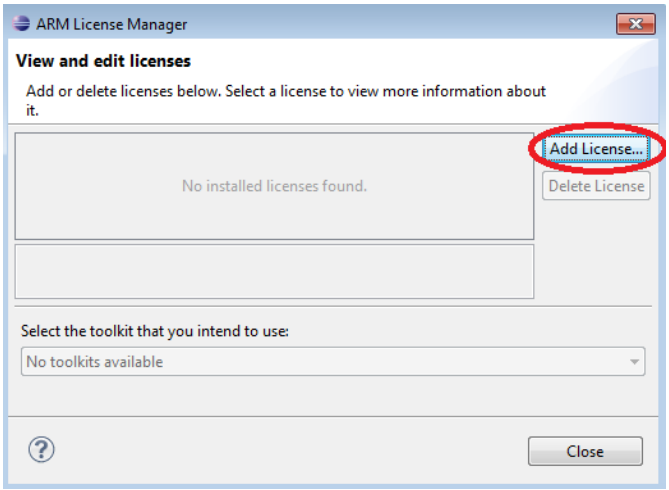


Install the 30 day DS-5 Altera Edition license

- **Launch DS-5**. **Start --> All Programs --> ARM DS-5 --> Eclipse for DS-5**
- A **Workspace Launcher** window will ask you to select a workspace.
- Press **OK** to select the **default**
- You will see a "**No Licenses Found**" Window. Select **Open License Manager**

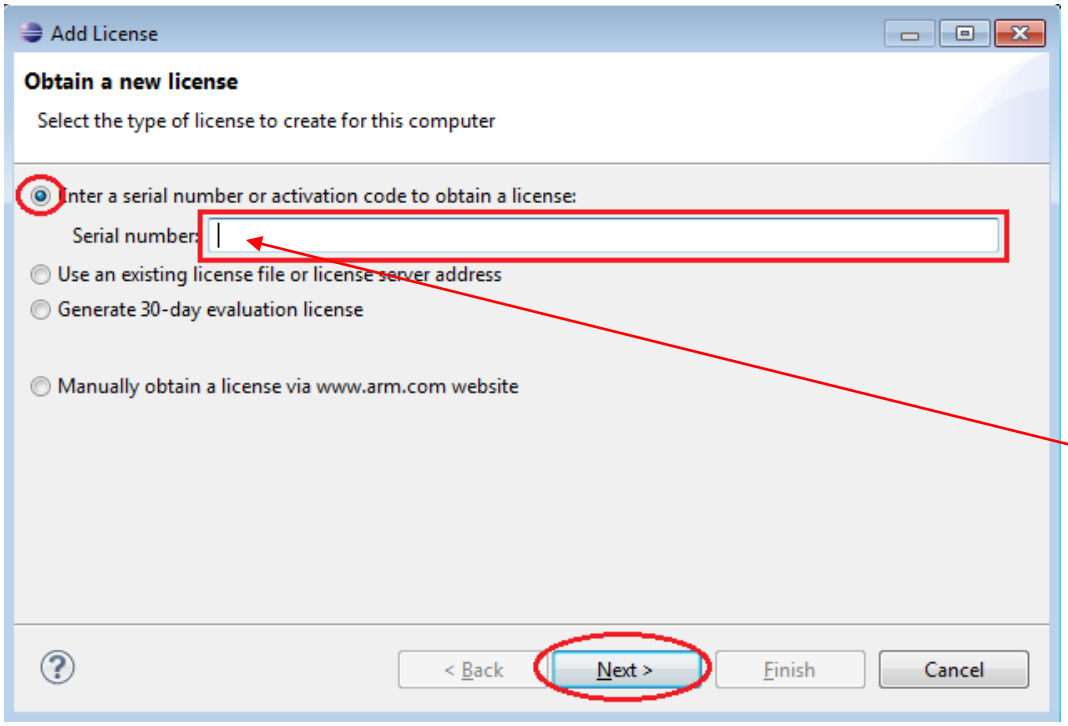


- **Press** the **Add License Button** in the ARM **License Manager**

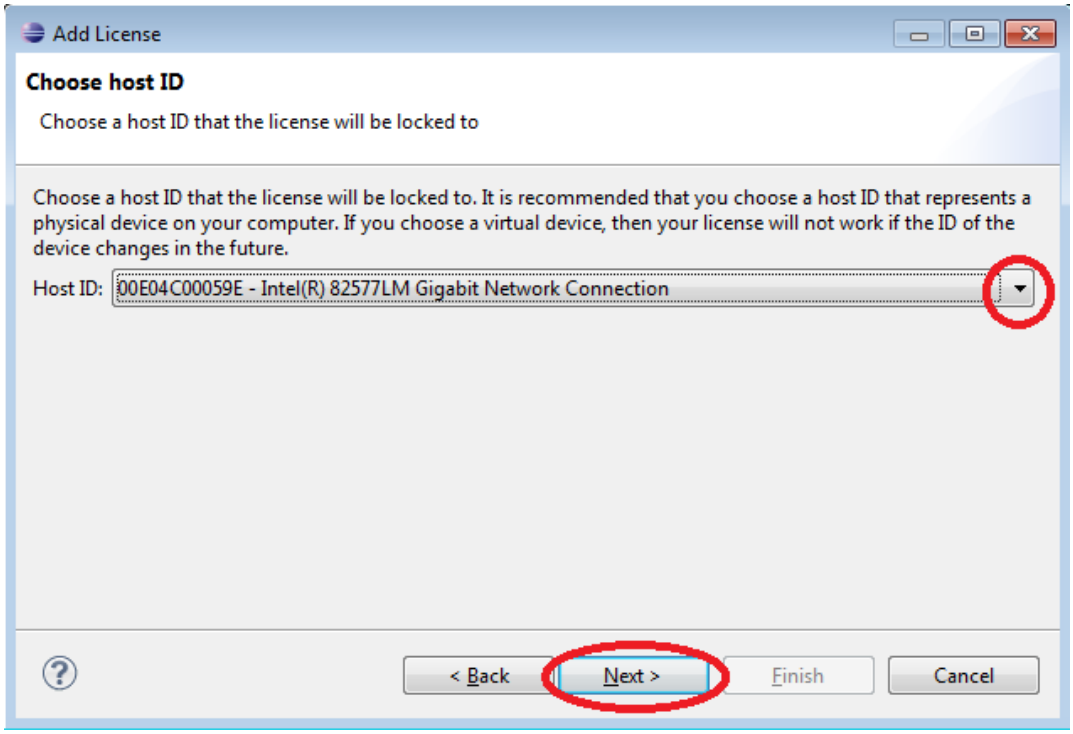


Please be aware that the **license will expire 30 days after** you perform the next step.

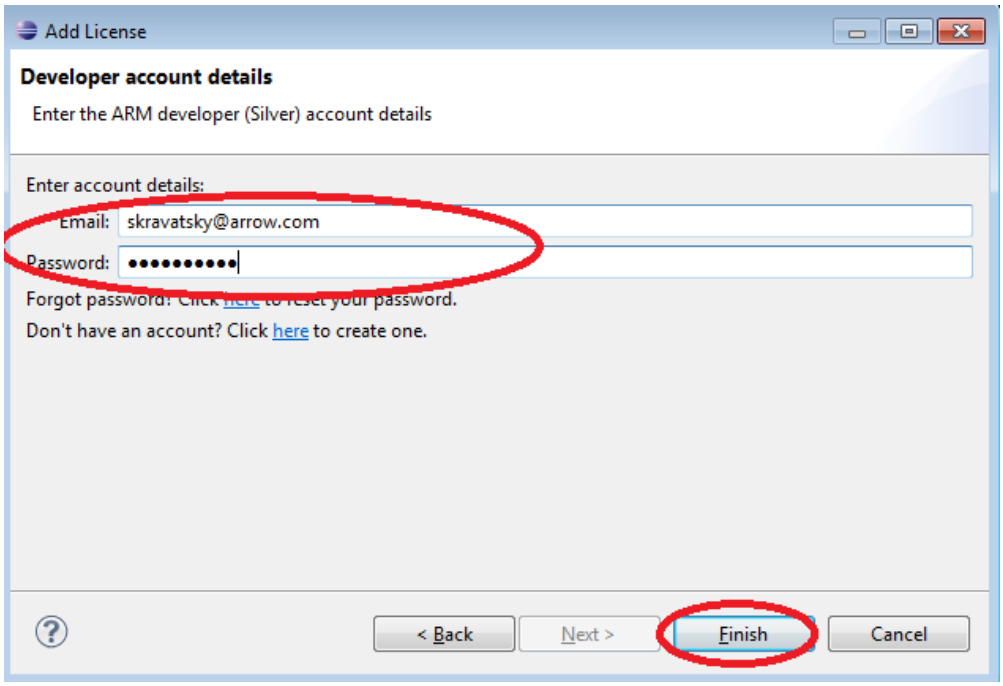
- Enter the **activation code** that you received **earlier.** Press the **Next** Button.

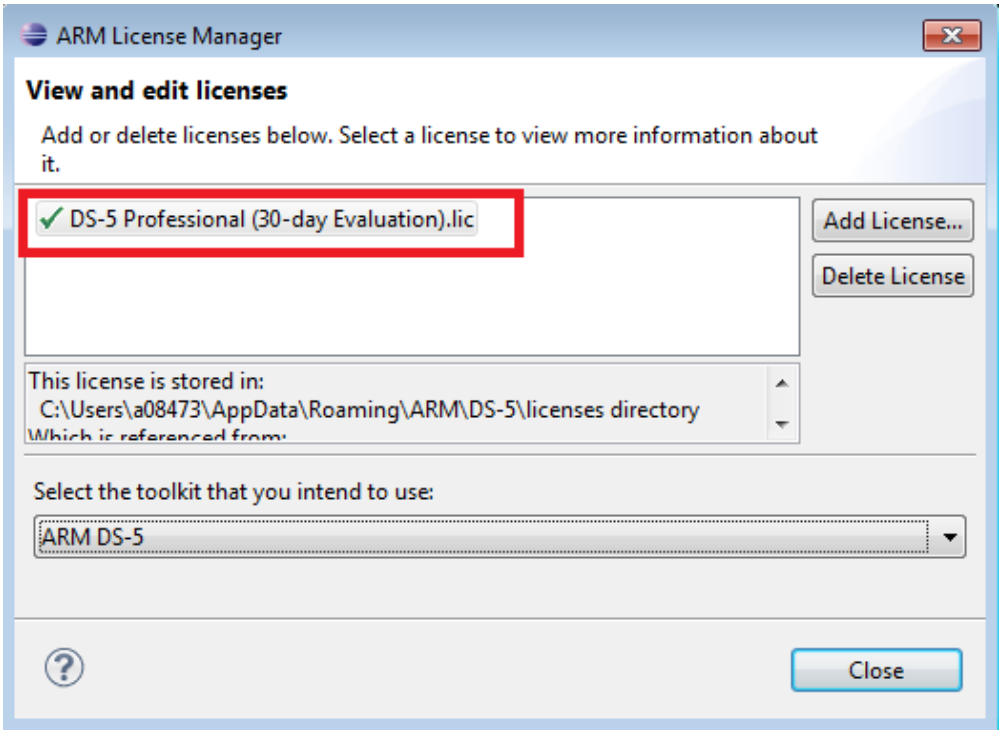


- Use **the pull down** menu to select a **host ID**. Press the **Next** button.



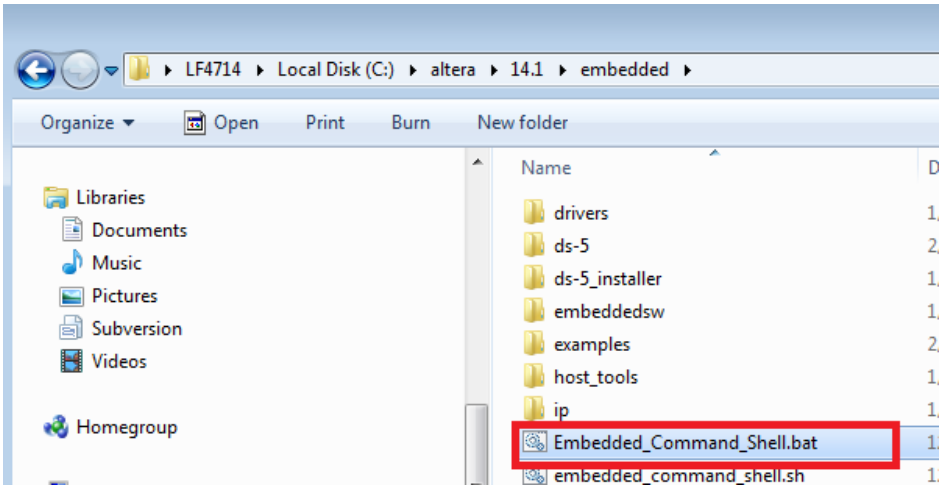
- **Enter** your ARM account **email address** and **password**.
- If you do not have an account then **click** on the **link** to **create** one.
- **Press** the **Finish** button.





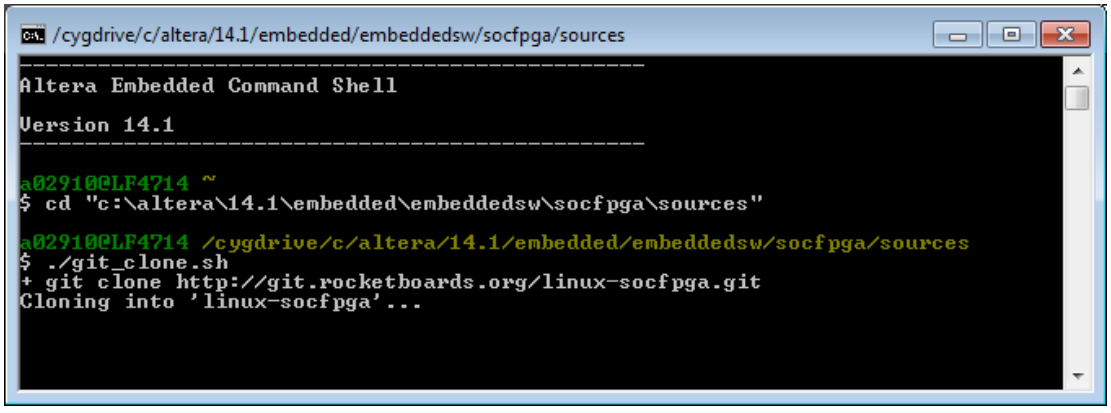
Use git to clone the **Linux source files. You must be connected to the internet to implement this step.** You will **need** these source files when you attempt **the optional cross triggering exercise in Module 6**.

- Open the **Embedded Command Shell**



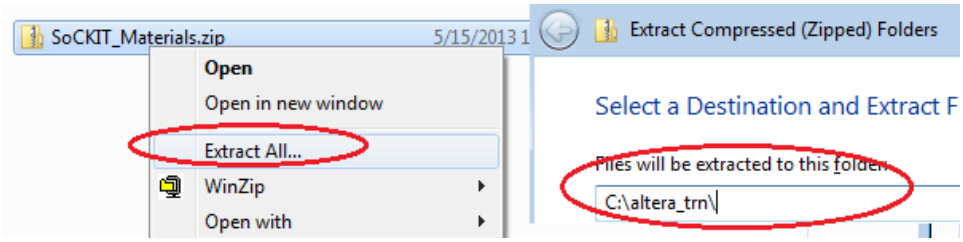
- Change directory to **c:\altera\14.1\embedded\embeddedsw\socfpga\sources**
- Type **source ./git_clone.sh**.Press **Enter.**

<u>Please note that this can take up to an hour to complete</u>

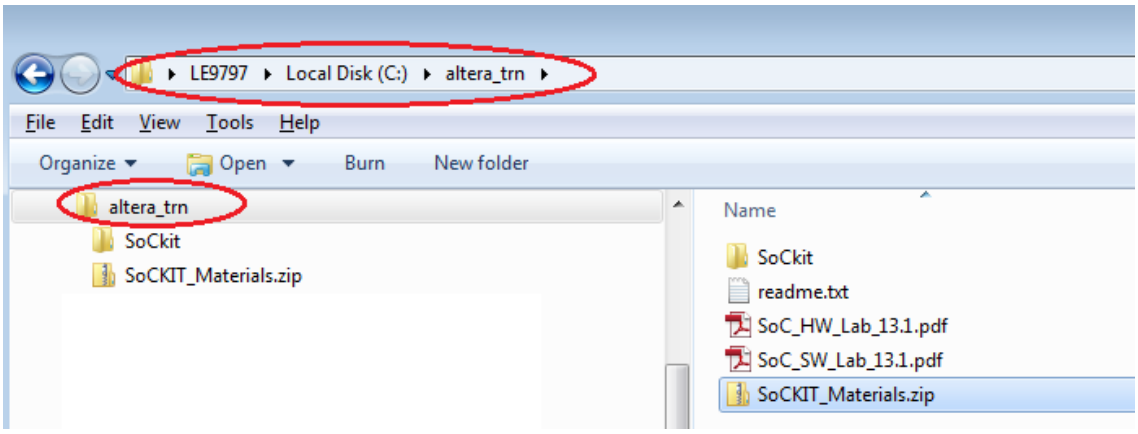


1.4 Extract the SoCKit Lab Files (Ignore if this has been done in the HW lab)

- Create a folder **c:\altera_trn** on your PC.
- Click on the following link to download **SoCKIT_Materials_14.1.zip**
- Save it to **c:\altera_trn** on your PC
- Extract the **SoCKIT_Materials_14.1.zip** file to this folder



- The **c:\altera_trn** directory should look like this



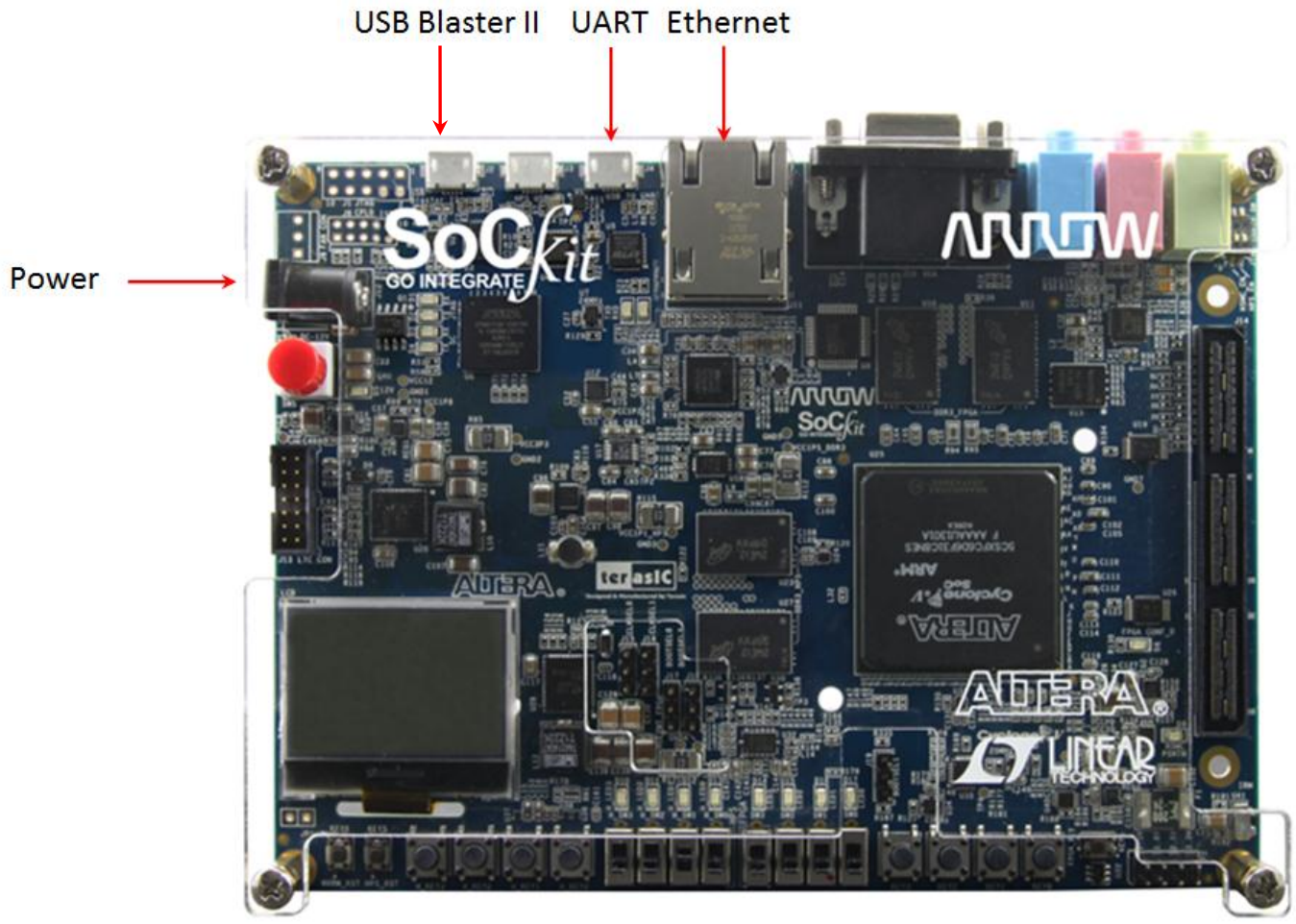
1.5 Download PuTTY

- Download **PuTTY** by clicking on this link: **Download PuTTY here**
- **No installation** is required. Move the .exe file to a convenient location that will be easily accessible during the lab.

1.6 Get the Cyclone V SoCKit ready for the Labs (Complete this at the Workshop)

Please connect cables to the connectors shown in the diagram below. All cables are provided in your SoCKit.

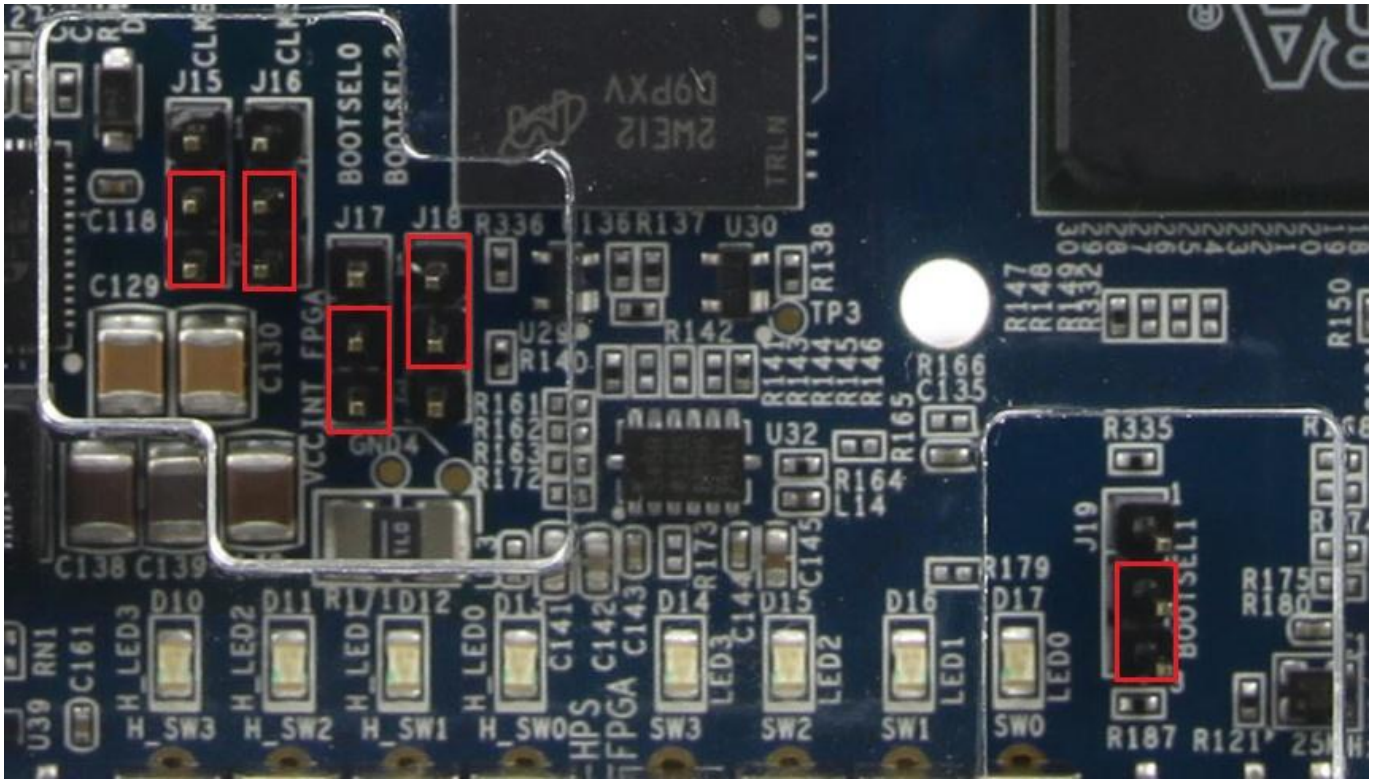
- Connect the micro USB cable to the USB host connector on your laptop and to the USB Blaster II connector on the SoCKit.
- Connect the second micro USB cable to the second USB host connector on your laptop and to the UART connector on the SoCKit.
- Connect the Ethernet cable to the Ethernet connector on your laptop and to the Ethernet connector on the SoCKit.
- Connect the Power Supply to the Power connector on the SoCKit.



There are a few jumpers that require configuring before proceeding with the labs.

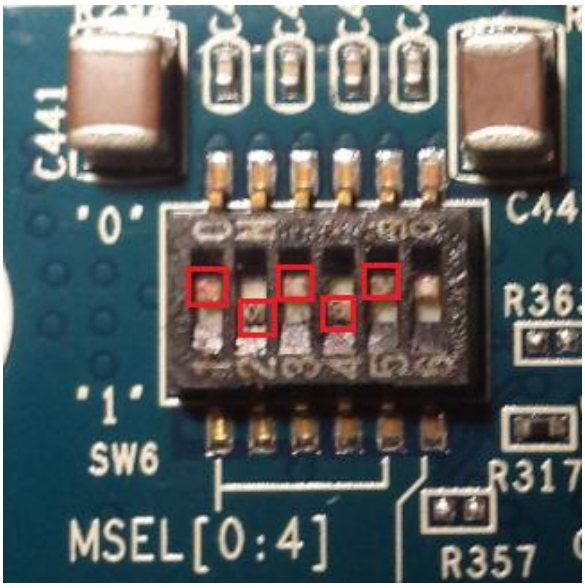
- **BOOTSEL[2..0]** jumpers. These should be configured as "100" to select boot from SD card 3.3V
- **CLKSEL[1..0]** jumpers. These should be configured as "00" for the slowest HPS peripheral clock speed option.

Please ensure that the jumpers are **configured** as **indicated** below.



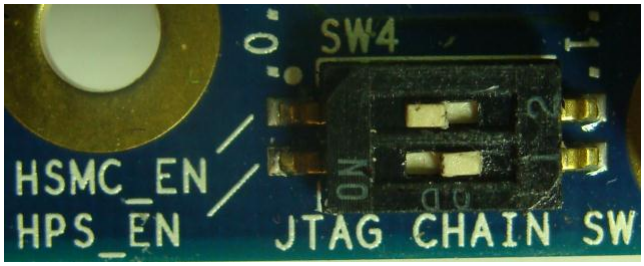
Modify the default **MSEL** bit settings. The board needs to be set to configure in the FPPx32, fast, compressed mode. This will allow u-boot to configure the FPGA.

- **SW6** is located on the **bottom** side of the SoCKit.
- Please change MSEL[0:4] to 01010.



Verify that the **JTAG chain is correctly configured**. The **JTAG chain switch** is located in to the right of the **green audio** connector.

- **HSMC_EN** should be **disabled** (left position) and the **HPS_EN** should be **enabled** (right position).

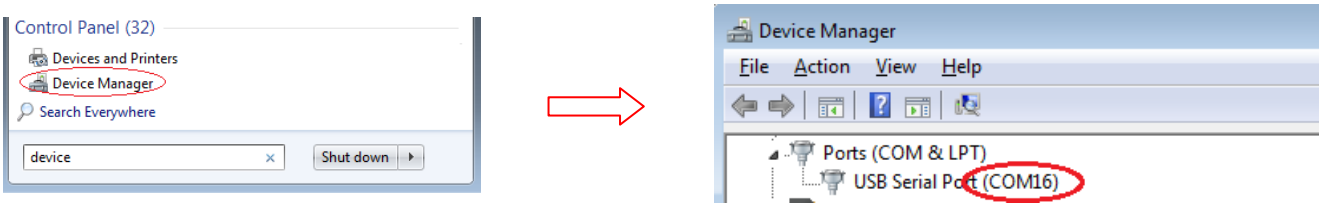


1.7 Configure the Serial Terminal for the Labs (Complete this at the Workshop)

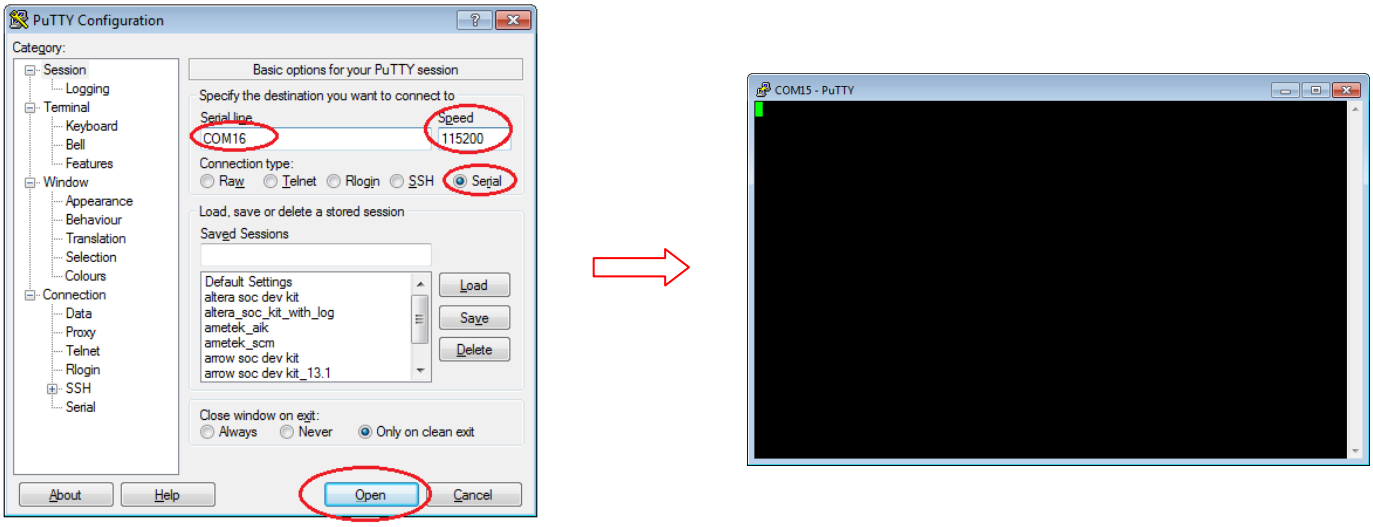
Caution:

Do not continue until you have done the following:

- **Eject the SD card before you power the board on.**
- **Turn your SoCKit on.**
- Verify the USB to UART COM Port. Open the Device Manager



- Open PuTTY and **configure** it for **Serial**, **115200** baud, COMxx. Press Open



You may Proceed

1.8 Preparing the SD Card

If you have purchased the SoCKit then your kit will most likely not contain an imaged SD card. The SD card is used to boot the Linux system and is used in a number of the Modules.

Please follow the links below to the Rocketboards.org web page that will provide step by step instructions on how to do so.

Creating an SD Card using a Windows Host

Creating an SD Card using a Linux Host

CONGRATULATIONS!!

You have just completed all the setup and installation requirements and are now ready to examine the system-level design.

MODULE 2: Examine the System Design

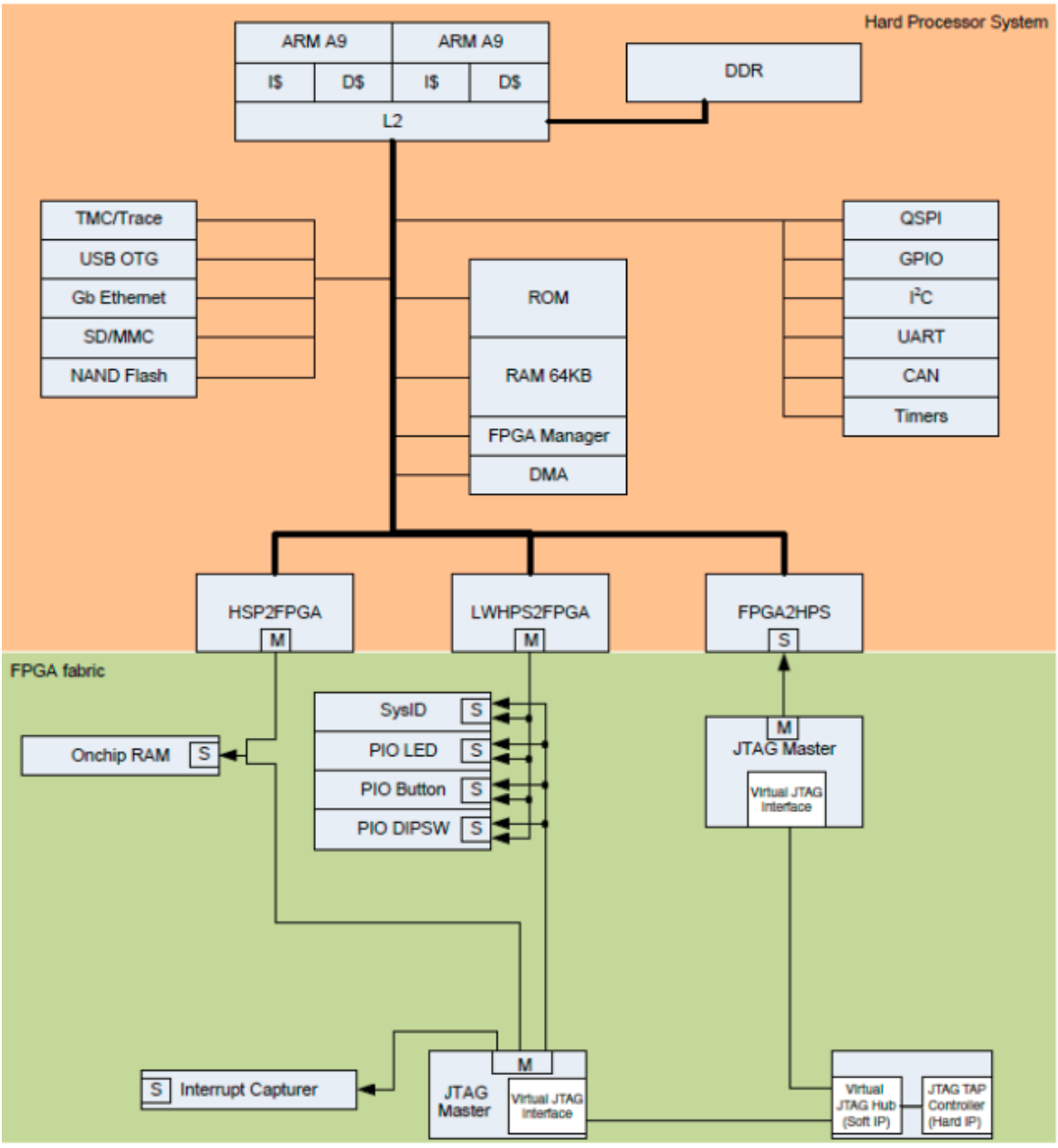
Module Objective

In this module you will **review** the **architecture** of the design that was created in **Qsys**. You will also **examine** the **layout** of the **SoCKit**.

2.1 System Architecture

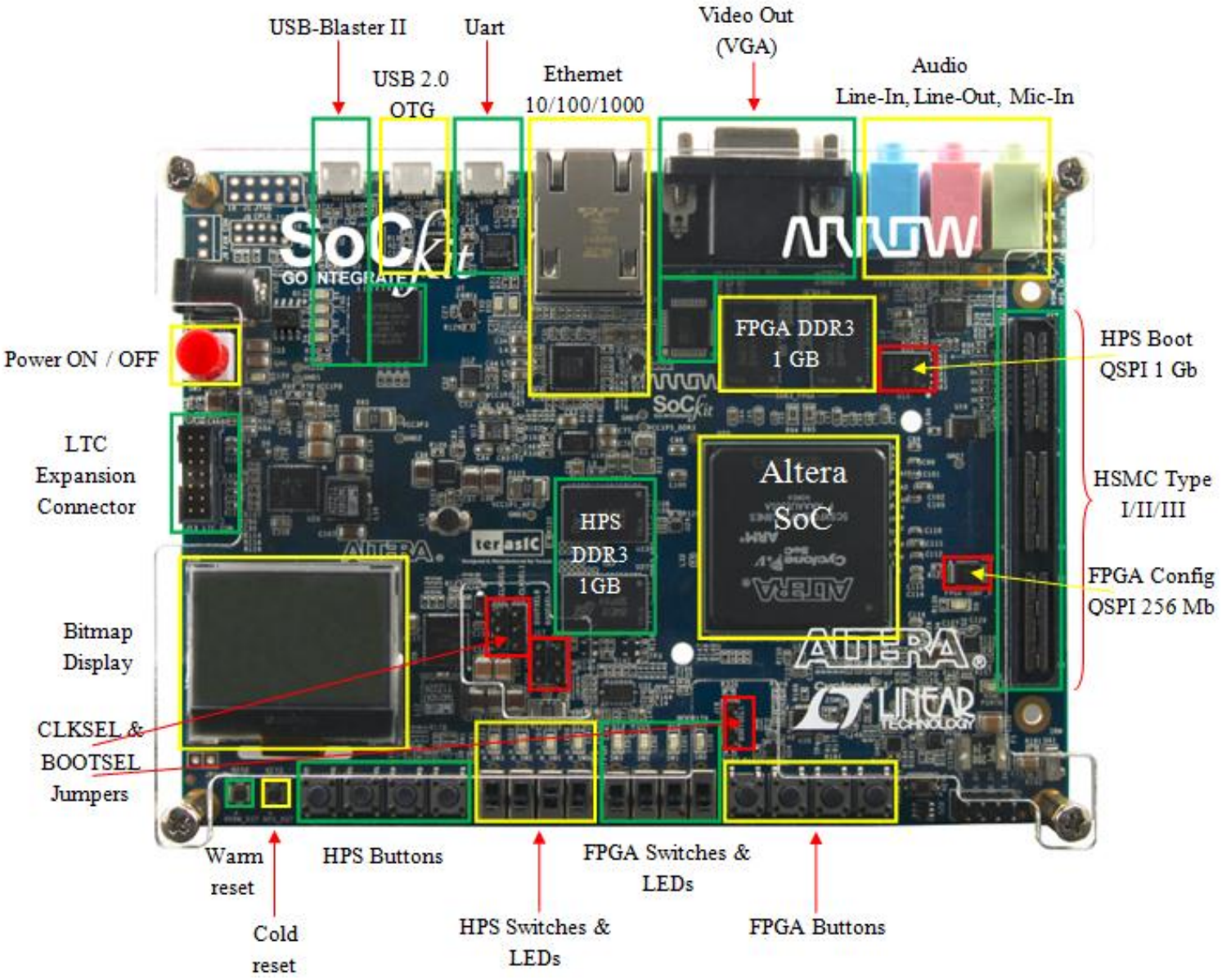
There are many components on the SoCKit that can be used, including the LCD, flash, Audio DACs, and IR.

The system was **created in QSys** using a **standard library** of **re-useable IP** blocks. The **orange section** of this diagram is the **HPS** section, while the **green** section is the **FPGA section**. The HPS section was configured in the **HPS component in Qsys**. There are **three bridges between the HPS and FPGA** sections. You will focus on peripherals connected to the **LWHPS2FPGA** bridge and for this lab, specifically, the LED PIO. They are mapped through the bridge into the HPS addressable map.



2.2 Examine the Cyclone V SoCKit

Examine the components on the Cyclone V SoCKit:



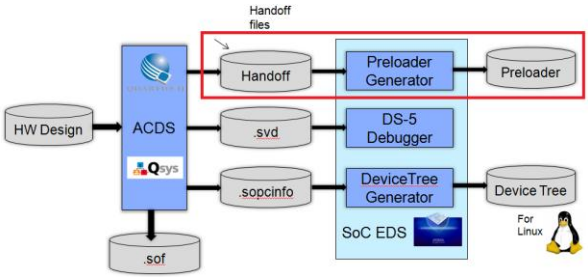
Note: The micro SD connector and the configuration DIP switch are located on the reverse side of the board.

CONGRATULATIONS!!

You have just completed the examination of the system-level design

MODULE 3: Generate, Build and Run the Preloader

In this section we will examine the path from the Handoff files through to the creation of the preloader as shown in the graphic on the right.

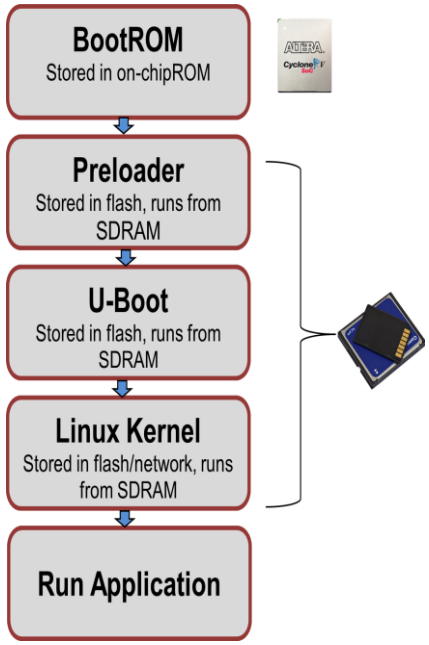


The preloader, also known as the spl or u-boot-spl (second program loader) is essential to being able to boot an operating system on an Applications class processor, such as a Cortex A-9.

The steps for booting an Application Class Processor include the following

1. The Boot ROM is run from power on reset or warm reset. It's only function is to read the BOOTSEL and CLKSEL settings and read the preloader from an appropriate source such as SD, QSPI or NAND flash.

2. The preloader is copied from the source to On Chip RAM (64K limit) and executed. Its main functions are to set the appropriate clocks for the processors and peripherals by manipulating the PLLs and setting up pin muxing required to route selected peripheral controllers to IO pins. It also sets the DDR memory controller parameters and calibrates the memory. When this is complete it will load the boot loader (in our case u-boot) from the external boot source to DDR and start its execution..



3. U-Boot will load the kernel and the device tree blob into memory from the boot source. It will launch the kernel and pass the dtb contents to it.

The Altera SoC is unique among Applications Class Processing solutions because the user can customize and add to the peripheral set attached to it by modifying the FPGA. All SoC customization is implemented by the user in the Qsys tool. This customization is passed to the software domain in the form of isw handoff files. These files are used by the BSP Editor to generate the preloader source files.

The first barrier to success that you will experience when you initially power up your own custom SoC based board will be to get the preloader to run. Being able to use the DS-5 Development suite and step through code will give you insight into what is functioning on your board and what might be causing a problem. It could be very helpful in uncovering any board level hardware issues.

In this module you will do the following

1. Generate the preloader using the BSP Editor
2. Build the preloader
3. Step thru the preloader using the ARM DS-5 development suite.

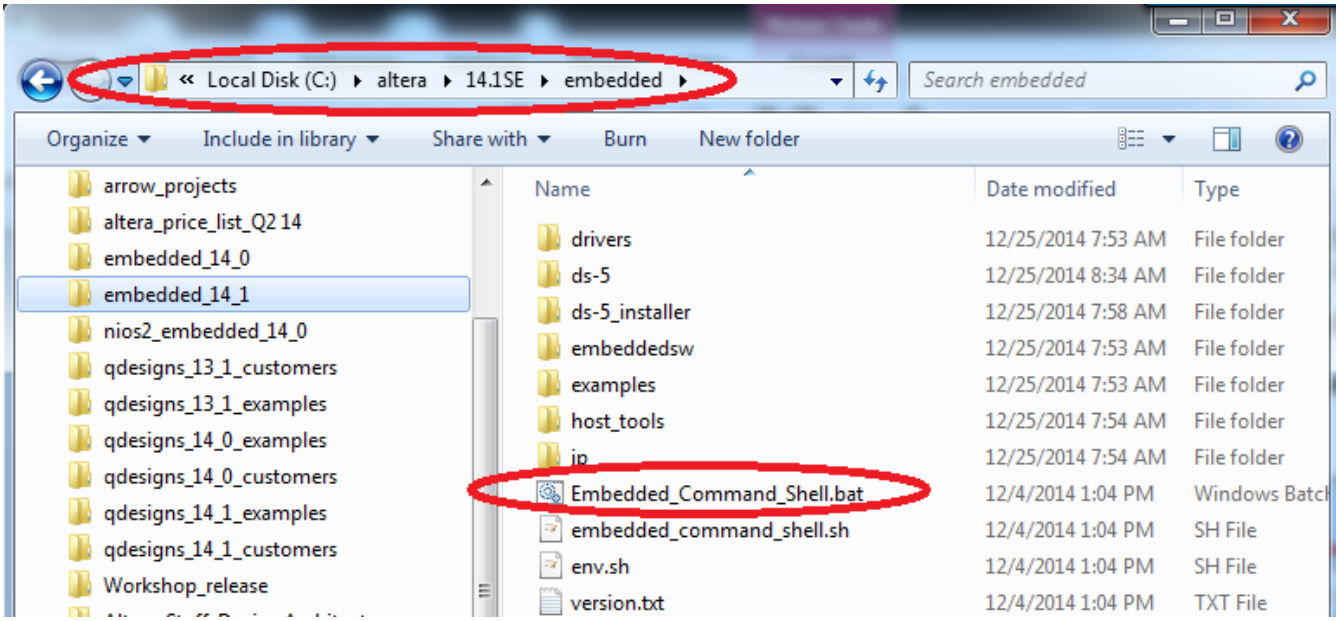
3.1 Generate the Preloader

Use the **ISW handoff** files and the **BSP Editor** to generate the **customized source code** for the preloader.

1. **Open the Embedded Command Shell**

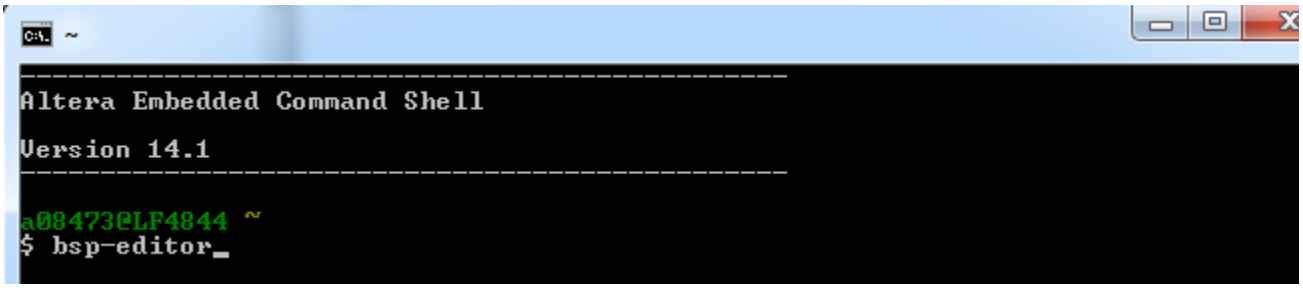
Navigate to the embedded install directory for the SoC EDS and launch the Embedded Command Shell

- **Browse** to **<Install Directory>\embedded** and select the Embedded_Command_Shell.bat file
- Double click the file to launch the shell



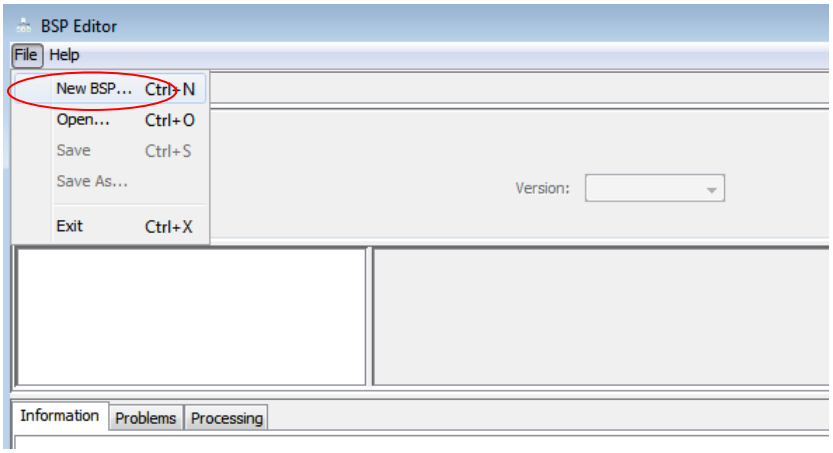
2. **Launch the BSP Editor**

- At the Command prompt type "**bsp-editor**" and press the enter key.



3. **Create a new BSP**

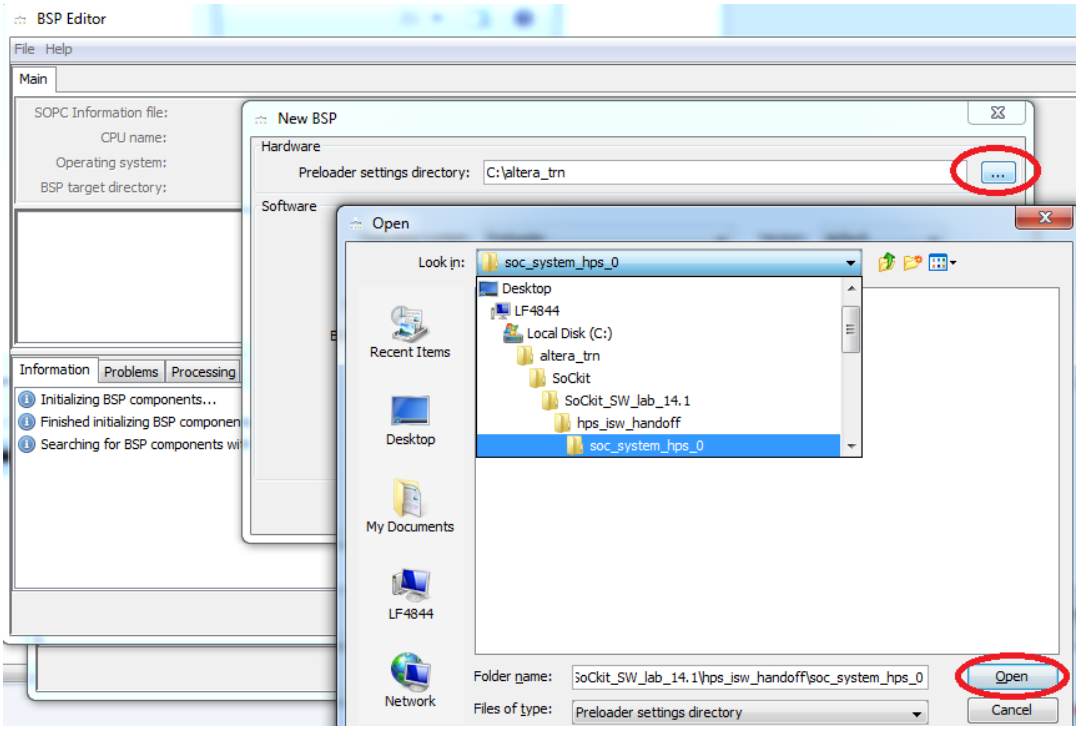
- Select **File --> New BSP** to create a new BSP



4. **Indicate the location of the Preloader Settings Directory**

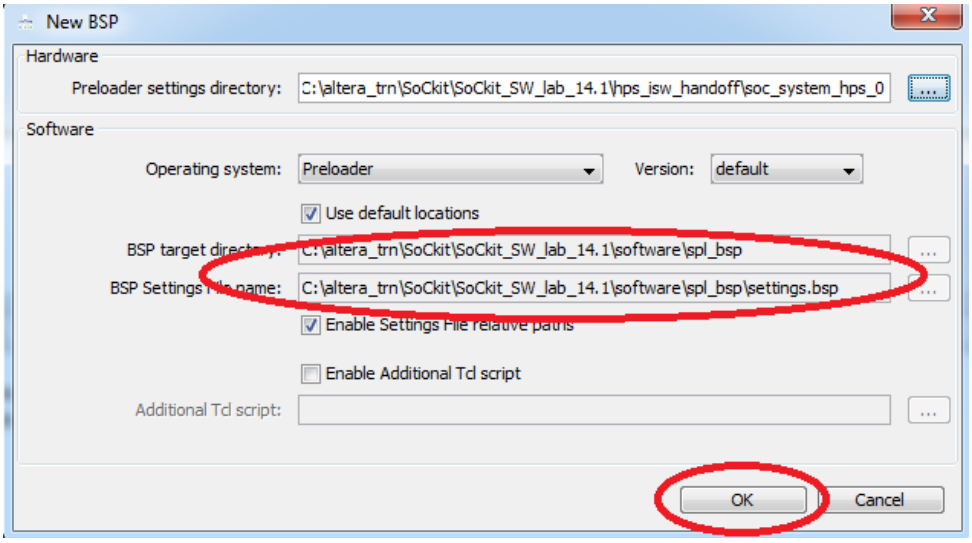
This directory contains the **xml files** that Quartus / Qsys has generated. They describe the customized peripheral and DDR **settings** for the **SoC**.

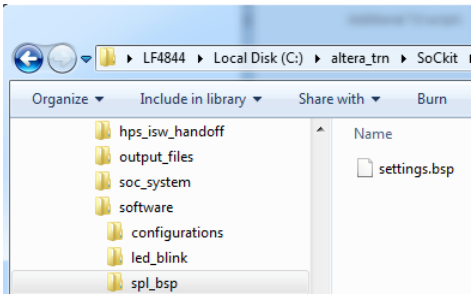
- Press the ... button to **navigate** to the directory, then press **Open**



5. Generate the preloader

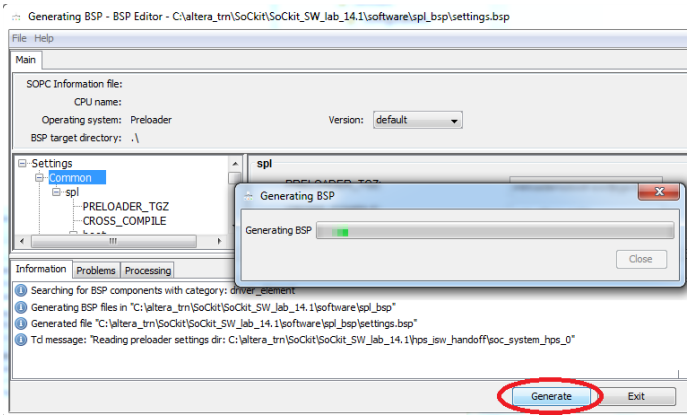
- Press OK to create the BSP settings file and directory

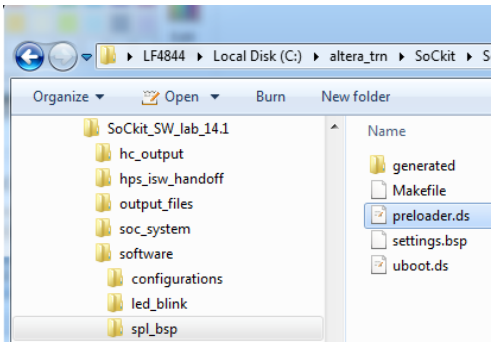




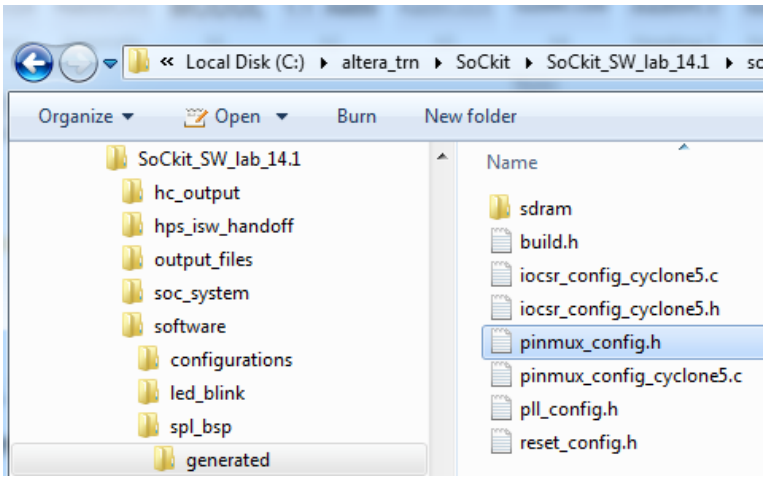
Note the **default location** of the created **preloader project directory** is **\software\spl_bsp**

- Press the **Generate button** to generate the preloader source and **makefile**
- Press **Exit** once generation is complete.





Take note of the **generated** sub-directory. The custom HPS information contained in the xml files have been converted into c header files that can be implemented when the preloader runs. A (1) next to a peripheral (in the pinmux_config.h file) indicates that its controllers output signals will be routed to the appropriate pins on the HPS portion of the SoC. The preloader will use this information when it runs the pinmux routine.



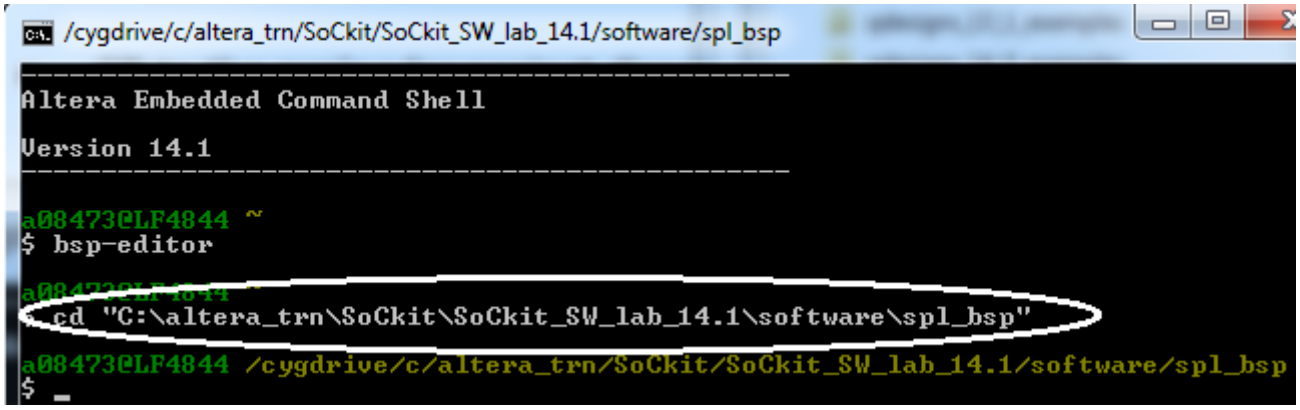

```
#ifndef _PRELOADER_PINMUX_CONFIG_H_
#define _PRELOADER_PINMUX_CONFIG_H_

#define CONFIG_HPS_EMAC0 (0)
#define CONFIG_HPS_EMAC1 (1)
#define CONFIG_HPS_USB0 (0)
#define CONFIG_HPS_USB1 (1)
#define CONFIG_HPS_NAND (0)
#define CONFIG_HPS_SDMMC (1)
#define CONFIG_HPS_QSPI (1)
#define CONFIG_HPS_UART0 (1)
#define CONFIG_HPS_UART1 (0)
#define CONFIG_HPS_TRACE (0)
#define CONFIG_HPS_I2C0 (0)
#define CONFIG_HPS_I2C1 (1)
#define CONFIG_HPS_I2C2 (0)
#define CONFIG_HPS_I2C3 (0)
#define CONFIG_HPS_SPIM0 (1)
#define CONFIG_HPS_SPIM1 (1)
#define CONFIG_HPS_SPIS0 (0)
#define CONFIG_HPS_SPIS1 (0)
#define CONFIG_HPS_CAN0 (0)
#define CONFIG_HPS_CAN1 (0)
```

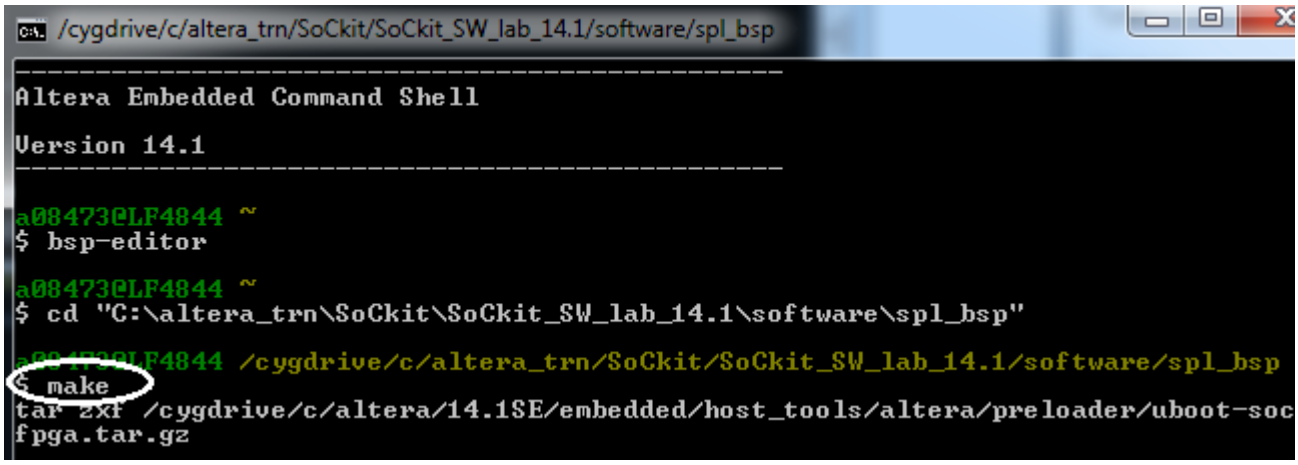
3.2 Build the Preloader

The preloader can be built from within the Embedded Command Shell

- CD to the preloader project directory within the shell

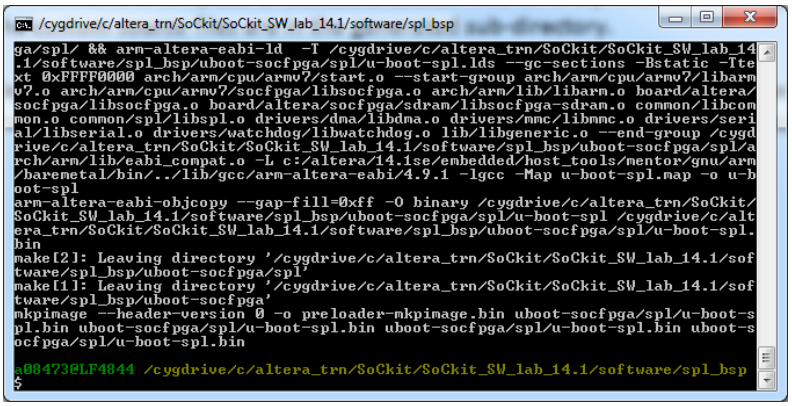


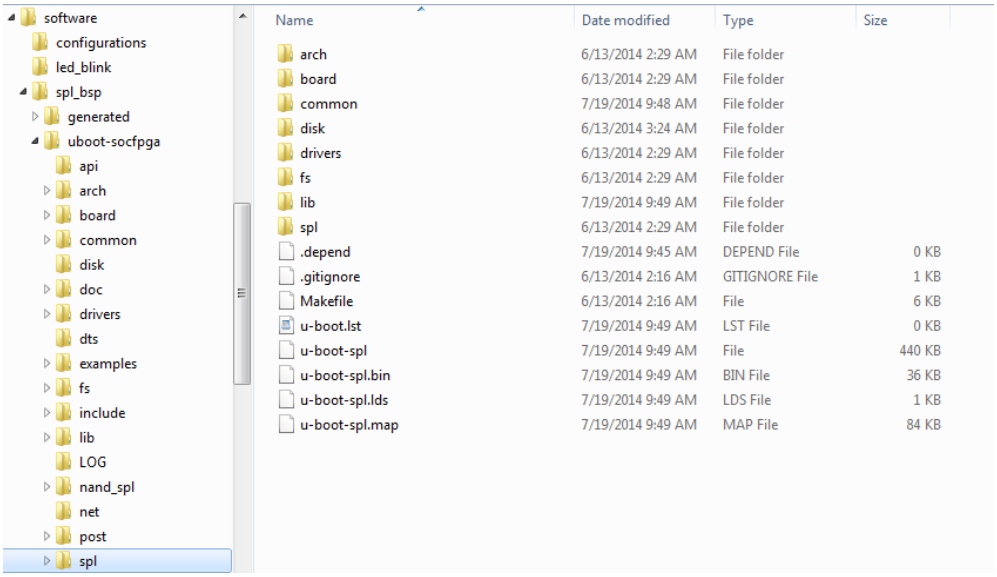
- Type "make" at the prompt and press enter



A tar file which contains a template of **standard source** files for the preloader is being copied from the SoC EDS install directory. The **custom source** files are in the generated sub-directory.

The preloader will take a few minutes to build. An examination of the preloader project directory after completion shows the project contents. The preloader ELF file resides in the **\software\spl_bsp\uboot-socfpga\spl** directory**.**



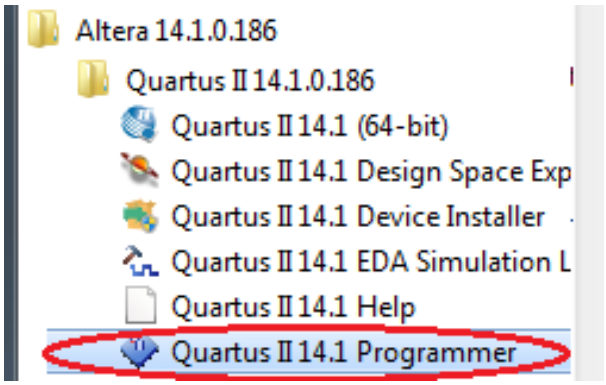


3.3 Download a hardware image to the FPGA

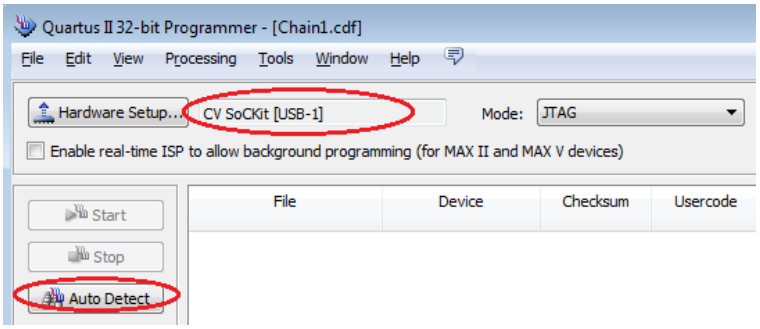
Before you continue please ensure the following:

- **The SD card is still in the ejected position**
- **The SoCKit is still powered on**

- **Launch the Quartus Programmer.**

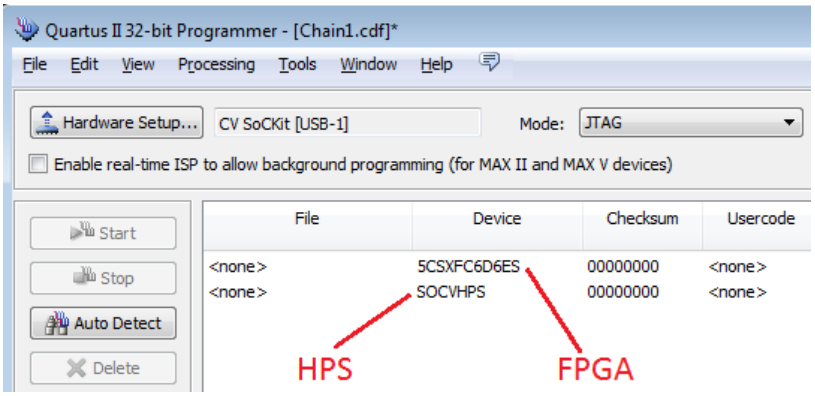


- Is the **USB-Blaster II** visible in the **Hardware Setup** window ? **If not**, press "**Hardware Setup**" and **select CV SoCKit** so that it populates the **currently selected hardware line**. Press **Close**
- Press the **Auto Detect** button to detect the **JTAG** chain.

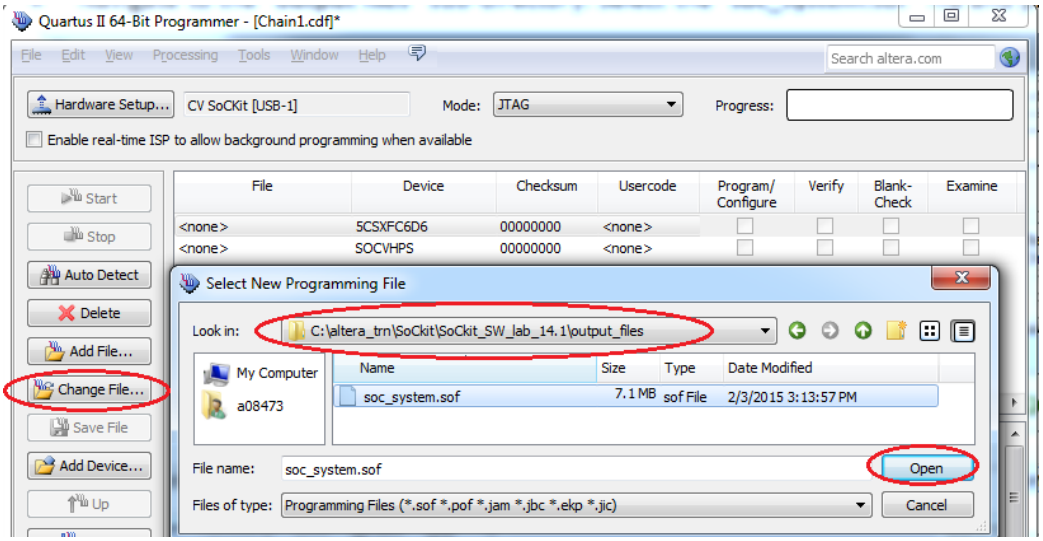


- Select the **5CSXFC6** device.

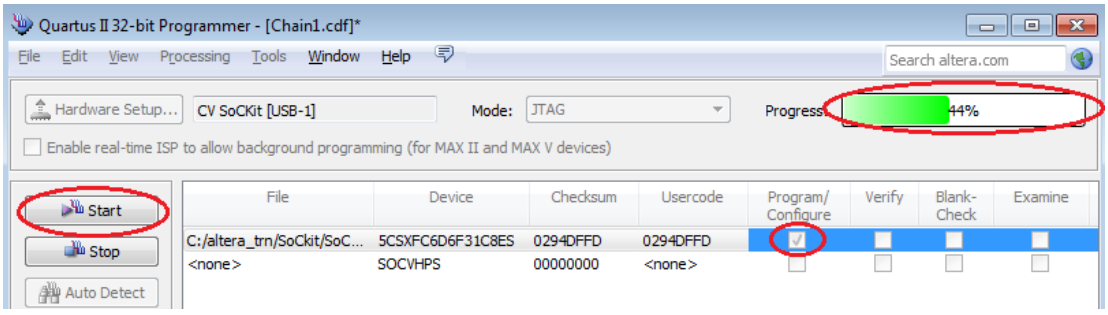
- Two devices are discovered. The first is the HPS section of the SoC. The second is the FPGA portion of the SoC



- Select the 5CSXFC6D6 for rev D kits or 5CSXFC6D6ES for earlier revisions. Press the Change File button.
- Navigate to the "**output files**" sub-directory. Select the "**soc_system.sof**" file and press the **Open** button.



- **Check** the **Program / Configure** box. Press the **Start** button. Wait till progress is at **100%.**

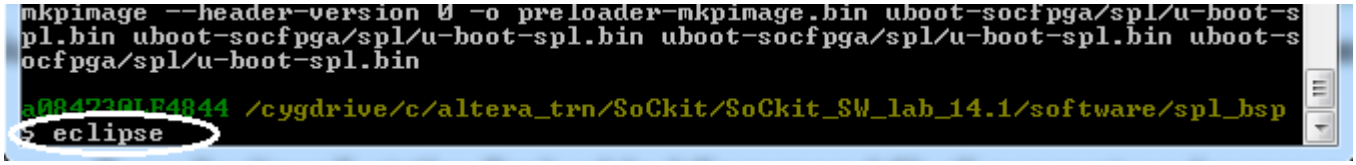


3.4 Launch DS-5 Embedded Development Suite & Import the Preloader project

1. **Launch DS-5 from the Embedded Command Shell**

Note: It is possible to launch DS-5 from the Windows Start button. **Do NOT** do this since the preloader project makefile requires that it be executed within a cygwin environment (the Embedded Command Shell).

- Type "eclipse" at the Embedded Command Shell prompt and press enter

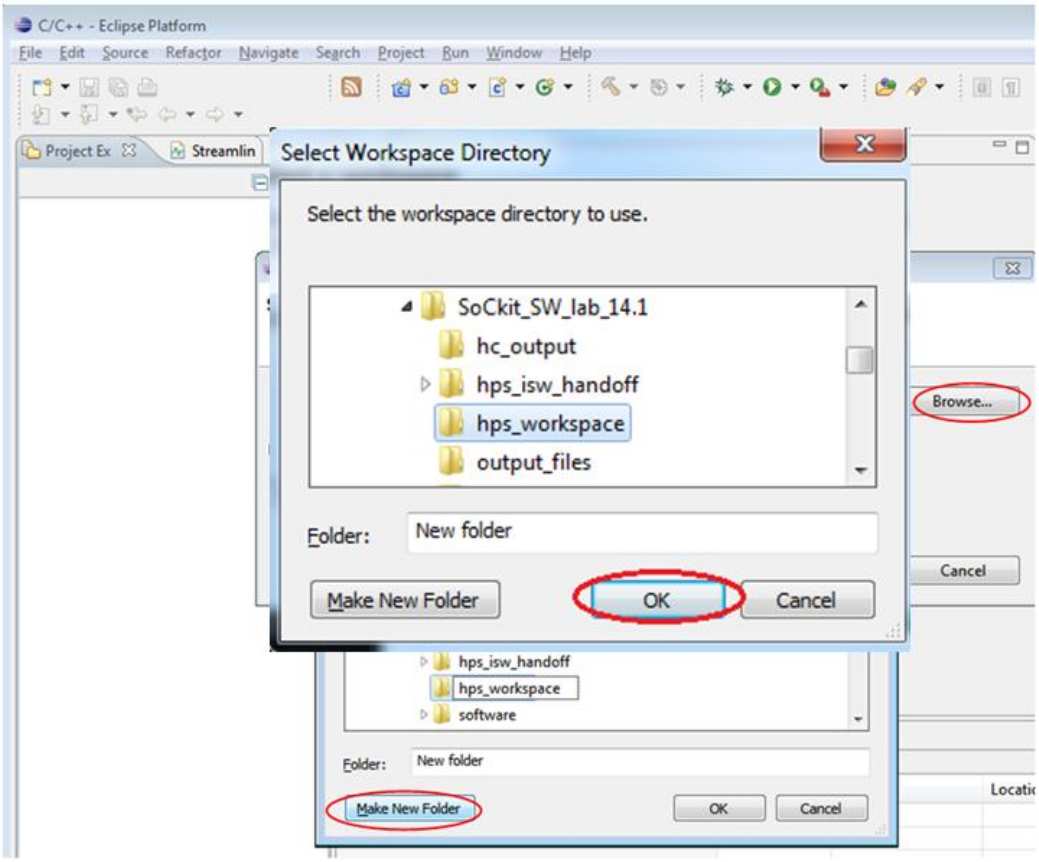


- Please wait for a few seconds while DS-5 starts up

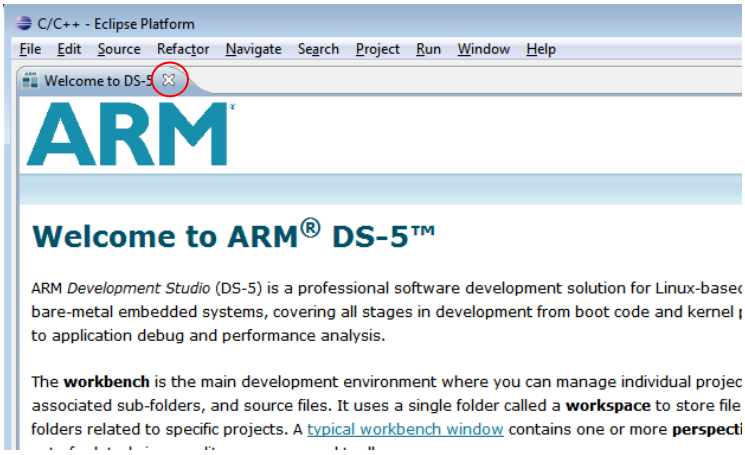
2. Initialize Eclipse workspace

When Eclipse first launches it is a good idea to select a specific workspace. It is useful to have a separate Eclipse workspace associated with each set of **hps_isw_handoff** files.

- Eclipse will request that you select a workspace
- Press the Browse... button to select a workspace directory.
- Navigate to the SoCKit_SW_lab_14.1 directory.
- Press the Make New Folder button and enter "hps_workspace". Press OK.
- Press OK. The DS-5 will shutdown and reload in the new workspace.
- Close the "Welcome to DS-5" tab



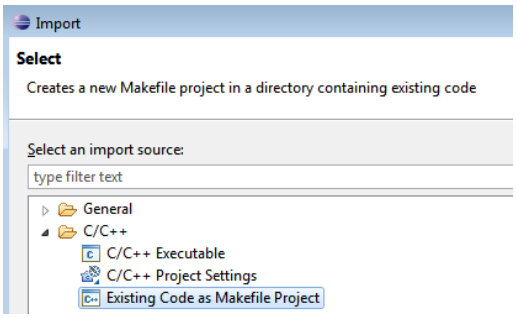
- Close the default "Welcome to DS-5" tab



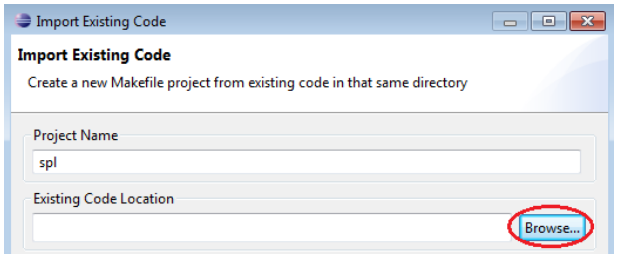
3. Import the Preloader project

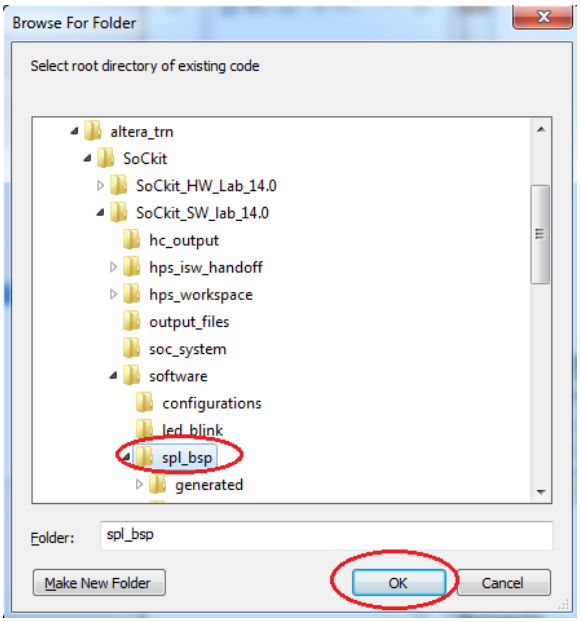
It is useful to import the preloader as a makefile project into the DS-5 environment. This allows the user to perform source level debugging.

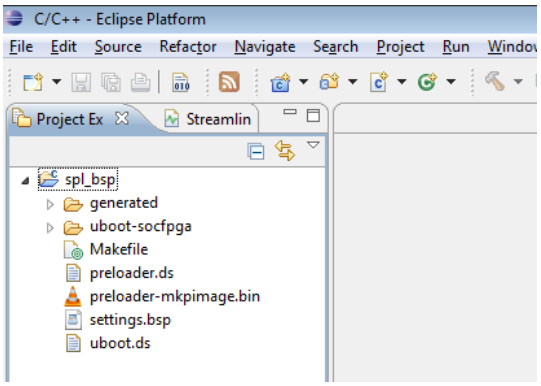
- Select **File --> Import**
- Navigate to **C/C++ --> Existing Code as Makefile Project.** Press Next



- Enter "spl" for the Project Name
- Press the Browse... button. Navigate to the Code location. Press OK. Press Finish





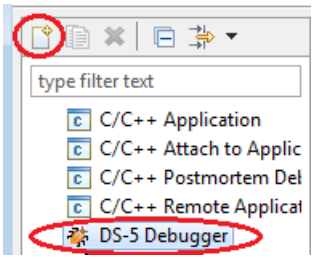


3.5 Create a Debug Configuration for the Preloader project

1. Create a new Debug Configuration

The debug configuration specifies the logistics required to debug the preloader software project. Connectivity to the SoCKit is selected here. DS-5 can be customized by using .ds scripts to perform initialization and setup functions before debugging begins. This is also where the specific ELF file that will be source level debugged is specified.

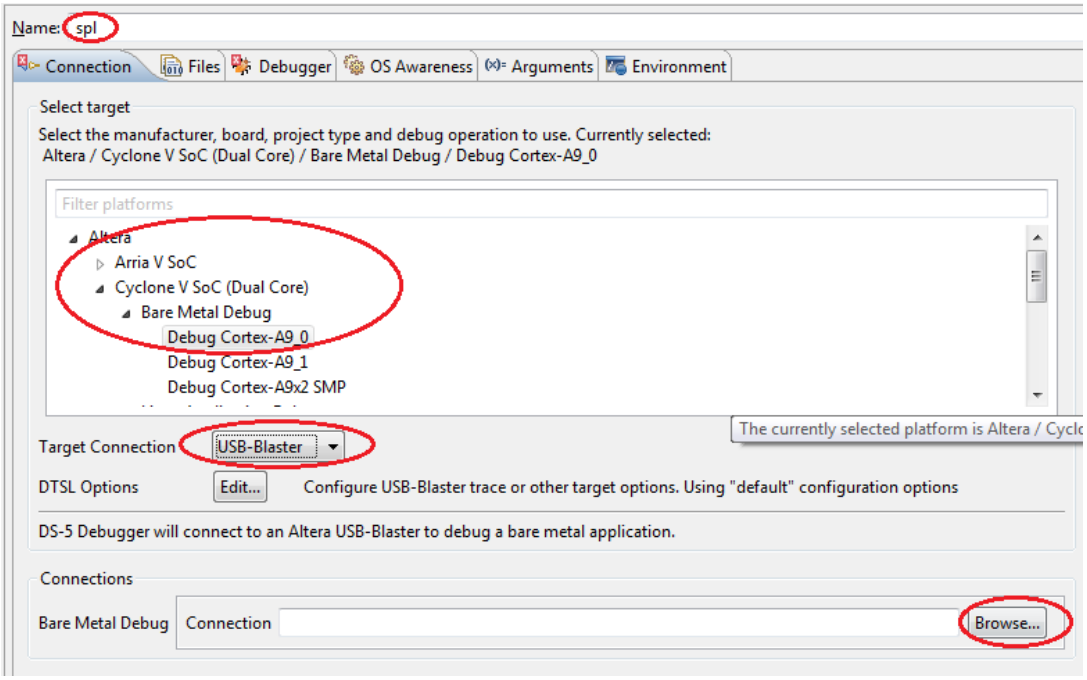
- Select **Run --> Debug Configurations**
- Select **DS-5 Debugger and press the "New Launch Configuration" button**



- Enter "**spl**" in the Name field

2. **Setup the Connection to the Target board**

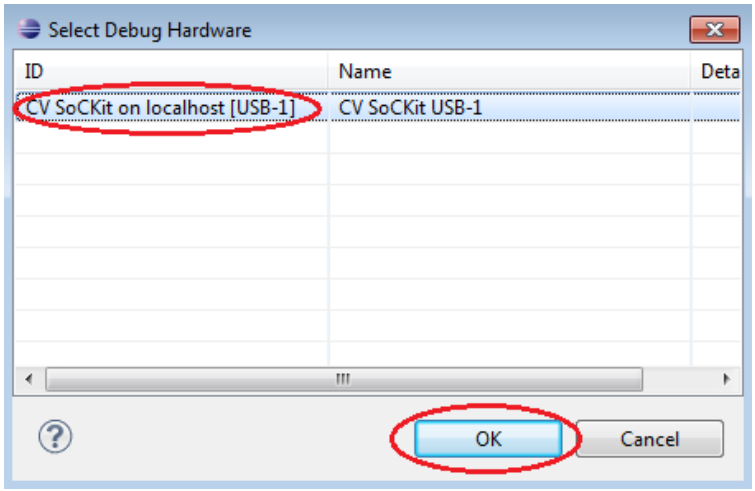
- Click on the **Connection** tab. Select **Cyclone V** --> **Bare Metal Debug --> Debug Cortex-A9_0** as the target.
- Click on the **Target Connection** pull down menu and select **USB-Blaster.**



Before you continue please ensure the following:

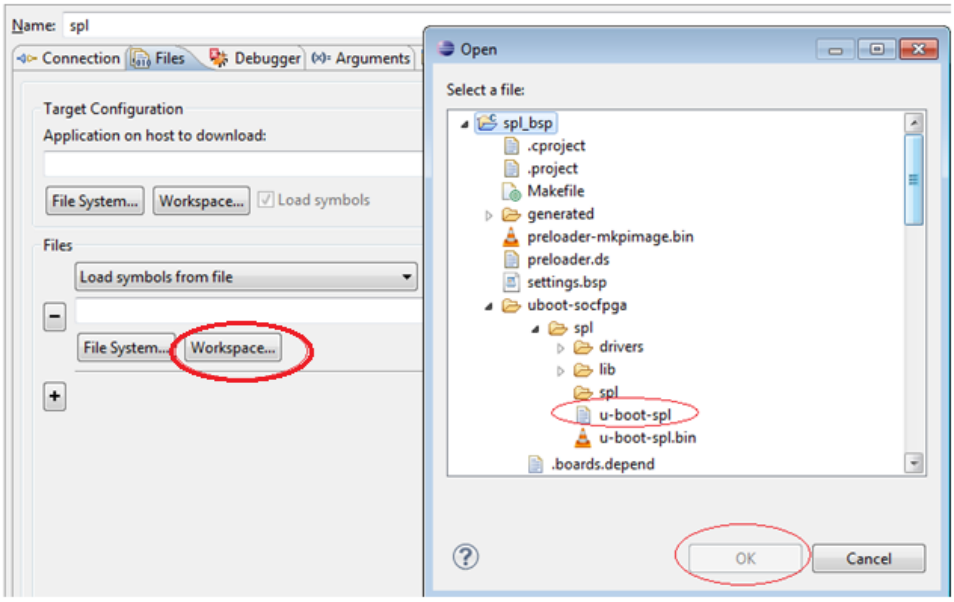
- **The SD card is still in the ejected position**
- **The SoCKit is still powered on**

- Click on the **Browse** button in the **Connections --> Bare Metal Debug** section.
- Wait a few seconds for the window to populate. Select the **CV SoCKit** and press the OK button.



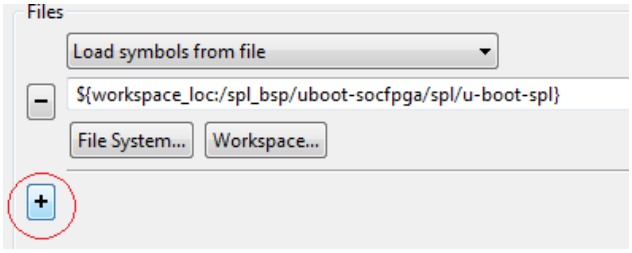
3. Select the files necessary for Target debug

- Click on the **Workspace** button in the **Files sub-section** of the **Files tab.**
- **Navigate** to the **spl_bsp --> uboot-socfpga --> spl directory** and select the **u-boot-spl** elf file. This file contains the **obj code** and **the symbol tables** for the preloader software project.

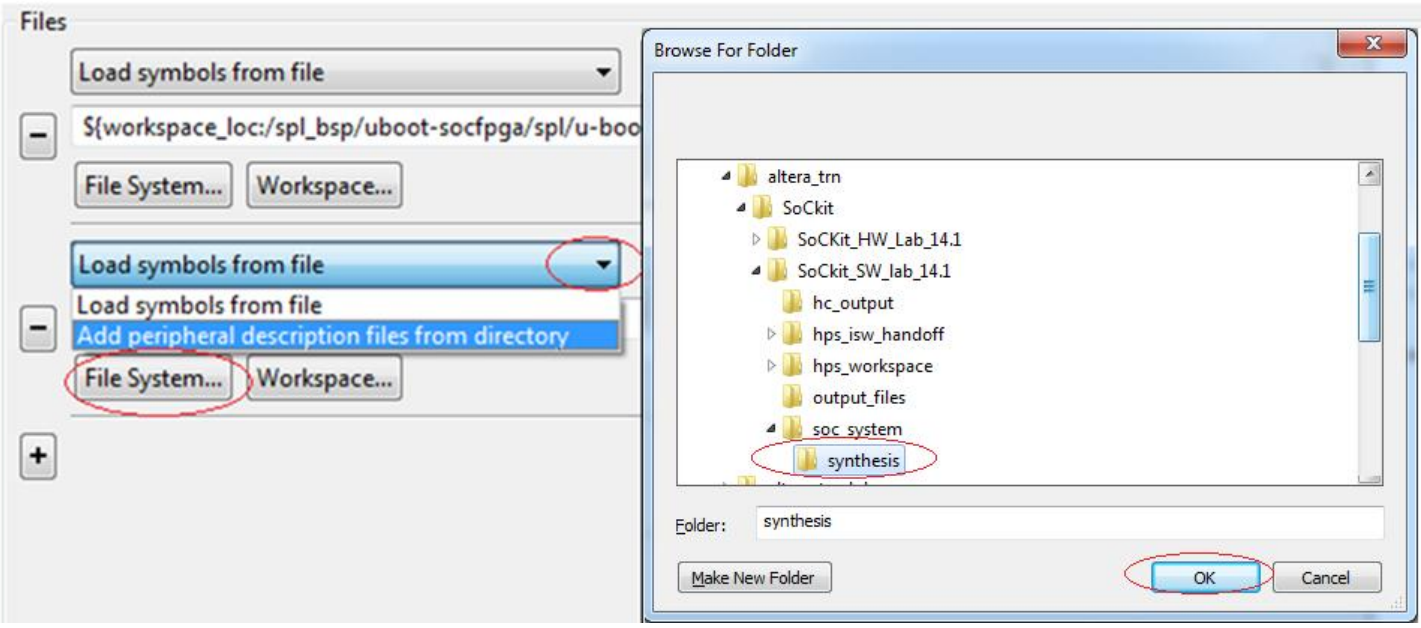


Note: Please verify that you have added "u-boot-spl" elf file to the Files section and NOT the Target Configuration section

- Press the + button to add another file



- Click on the pulldown arrow and select "Add peripheral description files from directory".
- Press the File System button. Navigate to the synthesis sub-directory. Select it and press OK

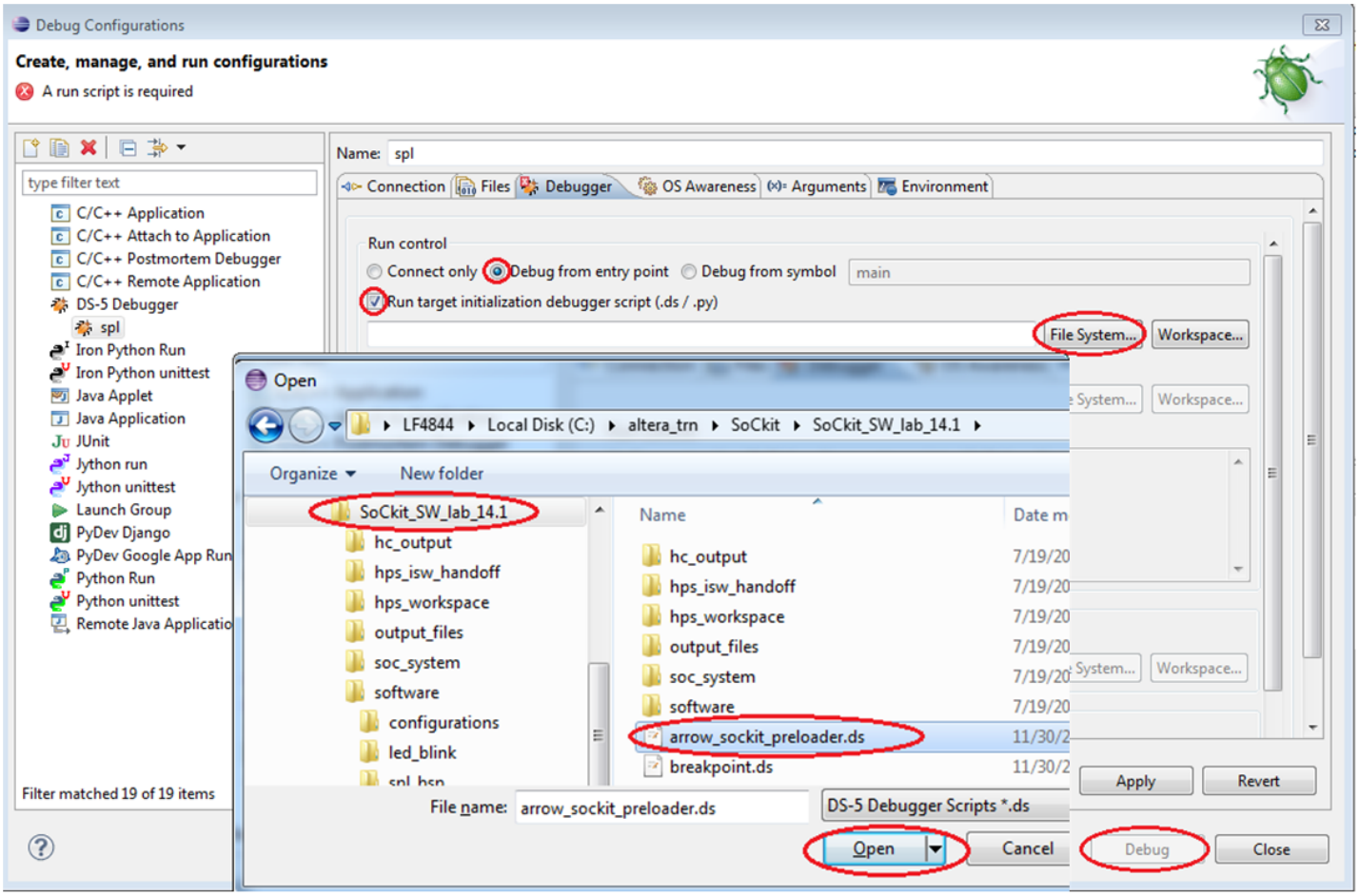


The **SVD (System View Description)** file is located in this directory. It was generated by Qsys and can be considered a handoff file for software debug. This file provides the DS-5 with information regarding the peripheral sub-system that was designed in the FPGA and connected to the HPS via the HPS2FPGA bridge. This will allow you to symbolically read or write to these peripherals and they will be seen as an extension to the HPS peripheral listing in the peripheral window in DS-5.

4. Configure the Debugger

- Click on the **Debugger** tab**.**
- Select the **"Debug from entry point"** pilot button.
- Check the "**Run target initialization debugger script**" box.

- Press the **File System** button and navigate to the "**arrow_sockit_preloader.ds**" script
- Press the **Open** button.
- Press the **Debug** button to start the **debug session**.



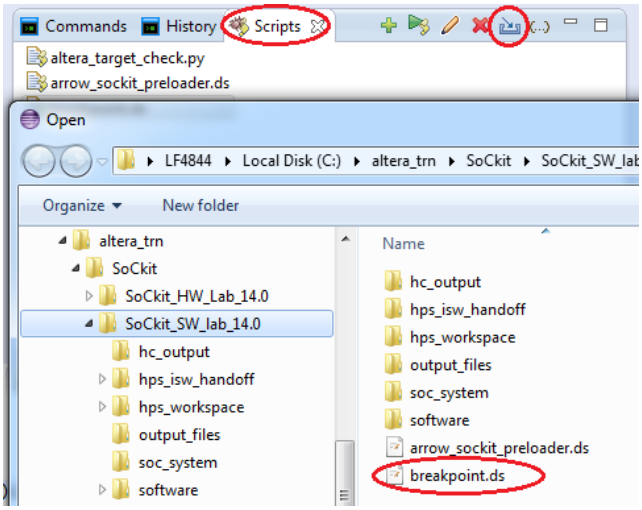
Note: For more information on DS-5 scripts please click on the following link. *Creating a debugger script file*

3.6 Step through and then Run the Preloader project

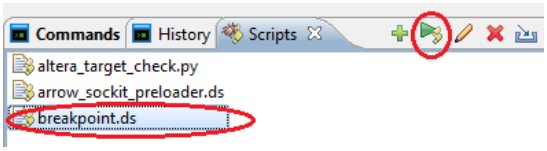
1. Add a ds breakpoint script

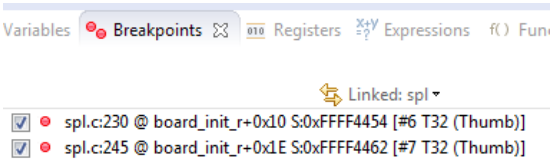
This script will conveniently add a few **breakpoints** that will assist in your **exploration** of the preloader code.

- Click on the **Scripts** tab
- Click on the **Import Scripts** icon
- Navigate to the **breakpoint.ds** script and press Open



- Select the **breakpoint.ds** script. Press **Open.**
- Press the **Execute Selected Scripts** button. Notice the **breakpoints** tab.





- Press the continue button to start the debugger. The debugger will stop at the first breakpoint

2. Explore the preloader code

As was discussed earlier the preloader is made up of standard code common to most system architecures and some generated code based on the customized system entry in Qsys. The section of code that you will explore is specific for the HPS, the DDR3 memory and peripherals that were specified in Qsys. Most of the board customization occurs in the **spl_board_init** function. This customization includes setting the PLLs, the HPS memory controller registers, the HPS I/O banks and implementing the necessary pin muxing.

```
230     spl_board_init();
231 #endif
232
233     boot_device = spl_boot_device();
234     debug("boot device - %d\n", boot_device);
235     switch (boot_device) {
236 #ifdef CONFIG_SPL_RAM_DEVICE
237     case BOOT_DEVICE_RAM:
238         spl_ram_load_image();
239         break;
240 #endif
241 #ifdef CONFIG_SPL_MMC_SUPPORT
242     case BOOT_DEVICE_MMC1:
243     case BOOT_DEVICE_MMC2:
244     case BOOT_DEVICE_MMC2_2:
245         spl_mmc_load_image();
246         break;
```

When the board initialization is complete the code will stop at the next breakpoint, spl_mmc_load_image. At this point it has examined the BOOTSEL jumper settings. It will attempt now to load the next loader from the SD card and run it out of DDR3 memory. At this point if the debugger becomes unstable and the next stage is unsuccessful, there is a good chance that the settings for the memory controller need to be fine tuned.

- **Press** the the **F5** key to **enter** the **spl_board_init** function
- **Examine the code**.

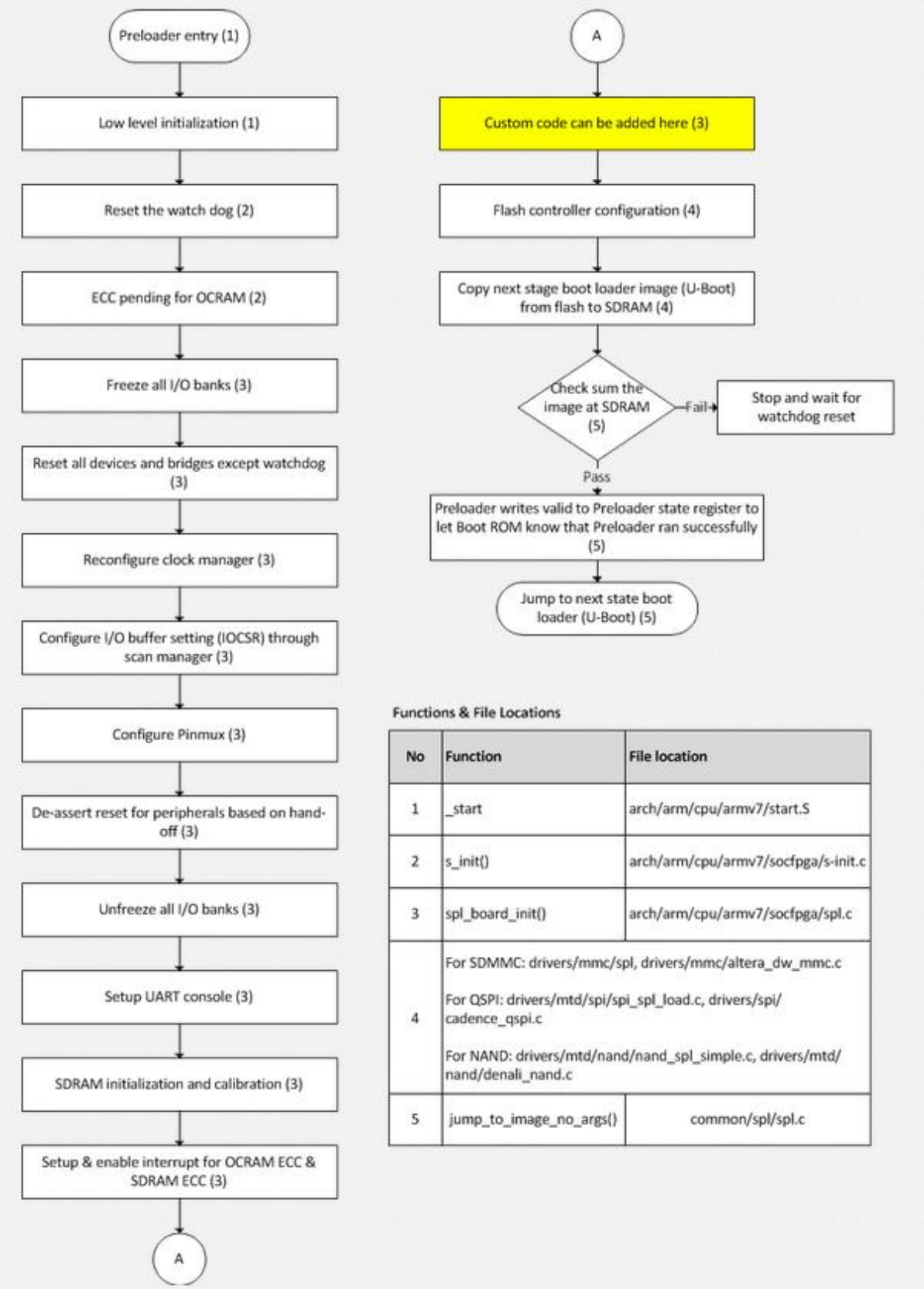
The flow diagram on the following page gives a good description of the order of operations taken to initialise the HPS. For more details please visit the preloader rocketboards page at

http://www.rocketboards.org/foswiki/Documentation/PreloaderUbootCustomization#Detailed_Preloader_Execution_Flow

Line 330 -409. Configure the main, peripheral and sdram **PLL** groups

Line 420 -429. IO Bank pins are configured via HPS I/O Scan chains. **Freeze the IO banks** before beginning the scan operation

Line 441 - 447. **Reset all peripherals and bridges** except for the L4 watchdog.



Functions & File Locations

No	Function	File location
1	_start	arch/arm/cpu/armv7/start.S
2	s_init()	arch/arm/cpu/armv7/socfpga/s-init.c
3	spl_board_init()	arch/arm/cpu/armv7/socfpga/spl.c
4	For SDMMC: drivers/mmc/spl, drivers/mmc/altera_dw_mmc.c For QSPI: drivers/mtd/spi/spi_spl_load.c, drivers/spi/cadence_qspi.c For NAND: drivers/mtd/nand/nand_spl_simple.c, drivers/mtd/nand/denali_nand.c	
5	jump_to_image_no_args()	common/spl/spl.c

Line 462 - 464. Timer used during PLL reconfig

Line 477 - 481. **Reconfigure the PLLs**. Any board level issues related to clock inputs **could result** in a problem here. On the SoCKit the **HPS CLK0** was double the specified frequency. **Executing** this step caused the system to **hang**. This provided a good clue and the problem was **resolved** soon after.

Line 499. Handshake the bootloader.

Line 504 - 520. The **Scan Manager** configures the **HPS I/O** via the scan chain.

Line 547. The **System Manager** sets the appropriate **pin muxing** for the HPS peripherals that were selected in Qsys. Stepping into this code will reveal that it uses the **pinmux_config**.h that was generated by the bsp-editor based on Qsys peripheral selections.

Line 586 - 595. **Unfreeze** the HPS I/O banks.

Line 605. **Enable UART** printing. The first line of code is printed to Putty from here.

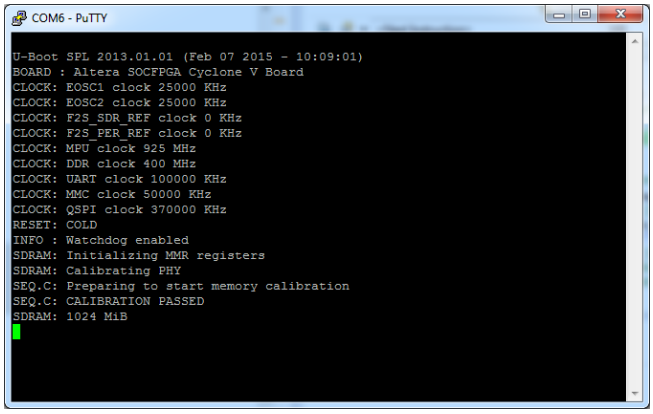
Line 635. **SDRAM Memory Manager** initialization.

Line 644. **SDRAM Calibration**.

Line 720 - 747. Setup and enable **exceptions**.

3. Run the preloader code

- **Press** the the **F7** key to **step out** of **the spl_board_init** function
- Examine the **PuTTY** console. You should see the **following**



- **Press the F8 (Continue key) to get to the breakpoint at line 245.**

Read the following paragraph but DO NOT implement

The next logical step would be to insert the SD Card and press F8. The preloader would attempt to **load U-Boot** from the **SD card**. It would first **transition** from **running** code out of **Onchip RAM** on the HPS to the **DDR3 memory**. If successful, you would see the system boot U-Boot and Linux. Any instability in this process would possibly point towards memory timing issues. **Tuning** of the memory **timing** in **Qsys** would be potentially required to **resolve** this.

However we will not do this since Module 4 requires DS-5 to still be connected to the target.

CONGRATULATIONS!!

You have generated, built and run the SoC preloader.

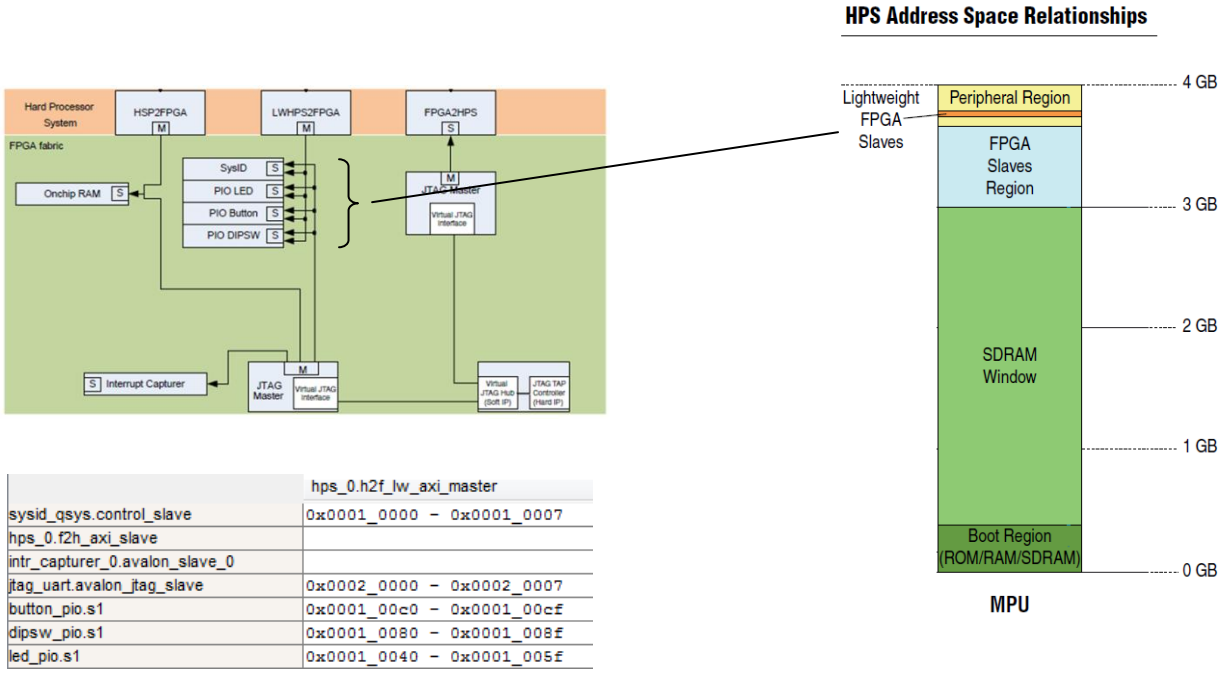
MODULE 4: Validating the FPGA Peripherals from the Hard Processor System (HPS)

It is **important** to understand how the **HPS** and FPGA **systems** are **combined** into a **common address map** as seen by the ARM **Cortex A-9** MPU.

First **examine** the **memory map** of the **SoC** as seen by the Cortex A-9 MPU. FPGA slaves connected to the high bandwidth HPS2FPGA bridge are mapped starting at 0xC000 0000 (3GB). The Onchip RAM is connected to this bridge. This bridge has a span of 960MB.

The **HPS peripherals** are mapped at **0xFC00 0000** with a 64MB address span.

The SysID, **PIO LED**, PIO Button and PIO DIPSW **FPGA slaves** are all connected to the **low bandwidth LWHP2FPGA** bridge. This bridge is mapped **within the HPS peripherals span** starting at **0xFF20 0000**. The span of this region is 2MB since it is only required for **control / status access**.



	hps_0.h2f_lw_axi_master
sysid_qsys.control_slave	0x0001_0000 - 0x0001_0007
hps_0.f2h_axi_slave	
intr_capturer_0.avalon_slave_0	
jtag_uart.avalon_jtag_slave	0x0002_0000 - 0x0002_0007
button_pio.s1	0x0001_00c0 - 0x0001_00cf
dipsw_pio.s1	0x0001_0080 - 0x0001_008f
led_pio.s1	0x0001_0040 - 0x0001_005f

The offset addresses of the FPGA slave peripherals relative to the base of the LWHPS2FPGA bridge are shown above.

So for example the LWHPS2FPGA bridge is mapped at 0xFF20 0000. The LED PIO will be offset from that base by 0x0001 0040. The resulting address for the LED PIO is 0xFF21 0040.

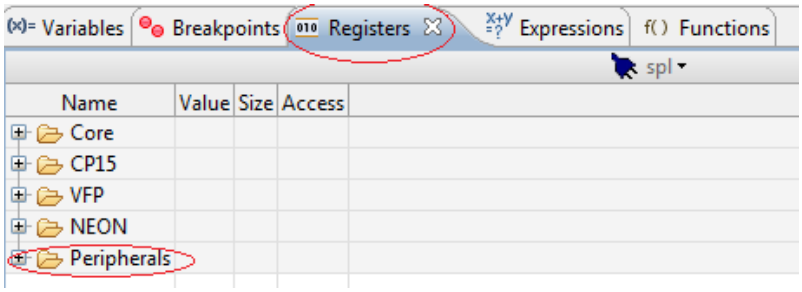
4.1 Validate the FPGA Peripherals from DS-5

Use the DS-5 **Debug** perspective **Register** tab to manually peek and poke the control, status and data registers of the FPGA peripherals that were defined in Qsys.

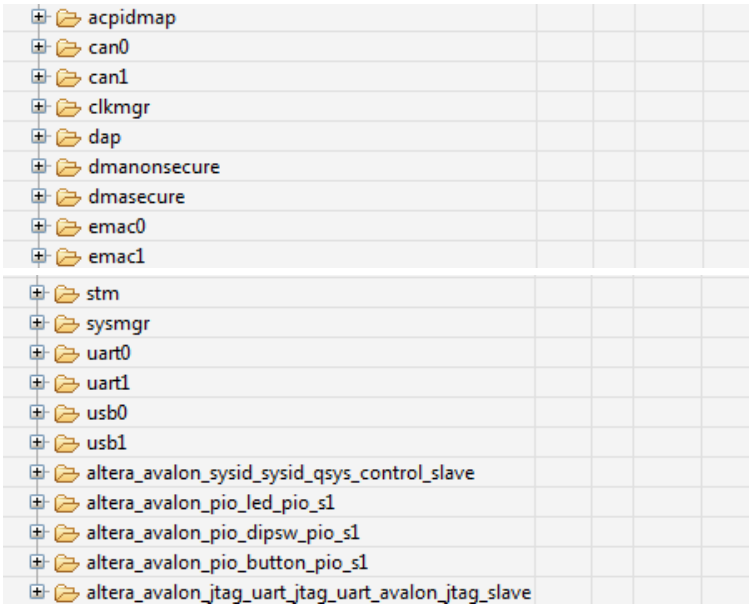
1. Use the Registers tab to access the FPGA peripherals.

The registers tab can be used to address all memory mapped entities within the HPS and the FPGA. It is a convenient way to validate newly created FPGA peripherals.

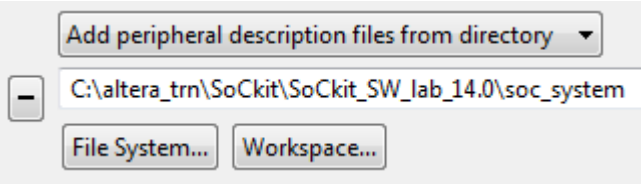
- Select the Registers tab. Press the ⊞ expander adjacent to the **Peripherals** field to see a complete list.

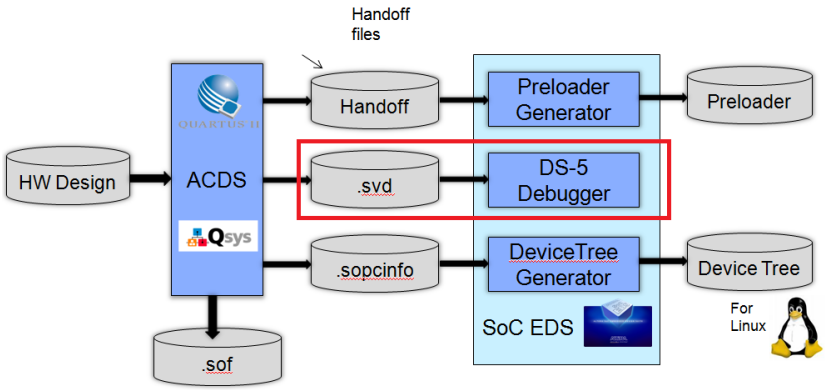


An incomplete list of peripherals is shown below. The peripherals that were added to the FPGA in the Qsys system are listed as **altera_avalon_<peripheral_name>.** All other listings are standard HPS peripherals.



The FPGA list of **peripherals** is dependent on what was added to the **Qsys system**. This information is passed to the DS-5 via the **SVD xml** file that Qsys generates. Recall that it was **referenced** in the Debug configuration setup in the Files section.





2. Exercise the FPGA led_pio peripheral.

There are **three bridges** that connect the **HPS** and **FPGA** portion of the **SoC**. Two of them are meant for high bandwidth data transactions (HPS2FPGA and FPGA2HPS). There is a third bridge (**LWHPS2FPGA**) that is intended as a control / status path for the HPS into the FPGA. This bridge allows the HPS to separately control low bandwidth FPGA peripherals without interrupting the flow of data on the high bandwidth paths.

These bridges are by default **left** in a **reset state** after power on and must be **removed** from this state.

View the Bridge Reset Status within the Reset Manager

- Navigate to the **rstmgr** peripheral and press the expander .
- Locate the **rstmgr_brgmodrst** register.
- Take note of its value

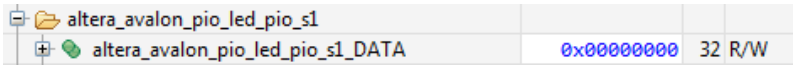
When the preloader **ran it detected** that the FPGA was **configured** and thus released all three **bridges from reset**. You are now **able** to access the FPGA peripherals.

Expand the LED_PIO peripheral.

The **programming model** for the **LED PIO** can be found in Chapter 12 of the *Embedded Peripherals Users Guide*.

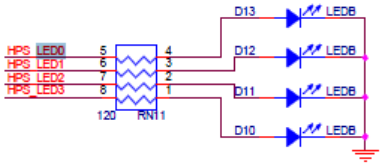
The **PIO is four bits**. Each **output bit is connected to an LED**. A bit value of **one will turn the LED on** and a value of **zero will turn it off**. The **FPGA LEDS** are located near the **Altera and Linear Technology logos**.

- Navigate to the **altera_avalon_pio_led_pio_s1** peripheral and press the ⊞ expander .
- Locate the **altera_avalon_pio_led_pio_s1_DATA** register.
- Type **F** in the data field to turn all the LEDs **on**.
- Type **0** in the data field to turn all the LEDs **off**.
- Type **1,2, 4 or 8** to turn individual LEDs **on**.



5. Use the HPS GPIO peripheral to turn on the HPS LEDs.

It also is possible to communicate with all HPS peripherals via the Registers tab. Four HPS LEDs are connected to GPIO pins [56..53] . These map to bits [27..24] in HPS register gpio1. The four HPS LEDs are located to the left of the four FPGA LEDs.



- Navigate to the **gpio1** peripheral and press the ⊞ expander .
- Locate the **gpio1_gpio_swporta_ddr** register. This is the data direction register. A gpio bit is an output if its corresponding ddr bit is set to a one. **Set** the **seventh** nibble to an **F**. All four **gpio** connected to the LEDs are **now outputs**.
- Locate the **gpio1_gpio_swporta_dr** register. This is the data register. Change the data in the **seventh** nibble of the data register to turn the LEDs on or off
- Type **0** in the data field to turn all the LEDs **off**.
- Type F in the data field to turn all the LEDs **on**.
- Type **1,2, 4 or 8** to turn individual LEDs **on**.

For more information on the GPIO , refer to the *General-Purpose I/O Interface*.
For more information on the HPS memory map refer to *Address Map information for the HPS*

- **Stop. Do NOT turn off the power. You MUST first disconnect the DS-5 from the target and then remove all connections for a clean session termination.**
- **Press the**  **"Disconnect from Target" button.**
- **Press the**  **"Remove Connection" button.**
- **Exit DS-5. File --> Exit**

4.2 Validate the FPGA Peripherals from a simple Linux Application

This section continues the philosophy of incrementally validating the FPGA peripherals that were added to the HPS in Qsys. The FPGA peripherals will now be validated from within the Linux operating system by way of a simple Linux application.

Linux has a virtual addressing scheme, so the application has to acquire a virtual address that represents the physical beginning of the HPS peripheral space. A simple application, " led_blink" was created as an example of how to validate FPGA peripherals from within a Linux application. An examination of the code below shows the mapping function implemented.

```
#include "socal/alt_gpio.h"
#include "hps_0.h"

#define HW_REGS_BASE ( ALT_STM_OFST )
#define HW_REGS_SPAN ( 0x04000000 )
#define HW_REGS_MASK ( HW_REGS_SPAN - 1 )

int main() {

   void *virtual_base;
   int fd;
   int loop_count;
   int led_direction;
   uint8_t led_state;

   // map the address space for the LED registers into user space so we can interact with them.
   // we'll actually map in the entire CSR span of the HPS since we want to access various registers within that span

   if( ( fd = open( "/dev/mem", ( O_RDWR | O_SYNC ) ) ) == -1 ) {
      printf( "ERROR: could not open \"/dev/mem\"...\n" );
      return( 1 );
   }

   virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, HW_REGS_BASE );

   if( virtual_base == MAP_FAILED ) {
      printf( "ERROR: mmap() failed...\n" );
      close( fd );
      return( 1 );
   }

   // initialize the LEDs

   // set the direction of the HPS GPIO1 bits attached to LEDs to output
   alt_setbits_word( ( virtual_base + ( ( uint32_t )( ALT_GPIO1_SWPORTA_DDR_ADDR ) & ( uint32_t )( HW_REGS_MASK ) ) ), 0x0000F000 );
   // set the value of the HPS GPIO1 bits attached to LEDs to ONE, turn OFF the LEDs
   alt_setbits_word( ( virtual_base + ( ( uint32_t )( ALT_GPIO1_SWPORTA_DR_ADDR ) & ( uint32_t )( HW_REGS_MASK ) ) ), 0x0000F000 );
   // set the value of the FPGA PIO bits attached to LEDs to ONE, turn OFF the LEDs
   alt_setbits_word( ( virtual_base + ( ( uint32_t )( ALT_LWFPGASLVS_OFST + LED_PIO_BASE ) & ( uint32_t )( HW_REGS_MASK ) ) ), 0x0000000F );
```

Provide the mmap function with HPS peripheral base and span and it returns a virtual mapping. Use this virtual base to address any peripherals with the HPS space including those mapped through the LWHPS2FPGA bridge.

Once the mapping function has been called the virtual base is used to manipulate HPS and FPGA LEDs via their respective PIOs. The memory map of the FPGA peripherals is provided in a header file (hps_0.h) that was generated by a utility called. sopc-create-header. The alt_setbits and alt_clr_bits functions are used to turn the LEDs on and off.

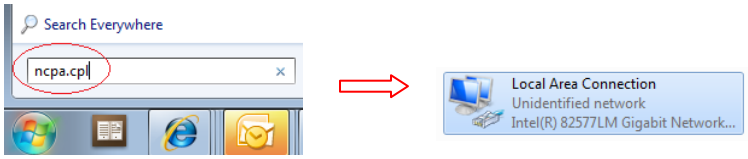
This application can be either built in a cygwin shell in Windows or on a Linux Host. In this section you will build this application within the cygwin shell, secure copy it to the target via ethernet and then execute it.

1. Connect the Linux target (SoCKit) to the laptop via Ethernet

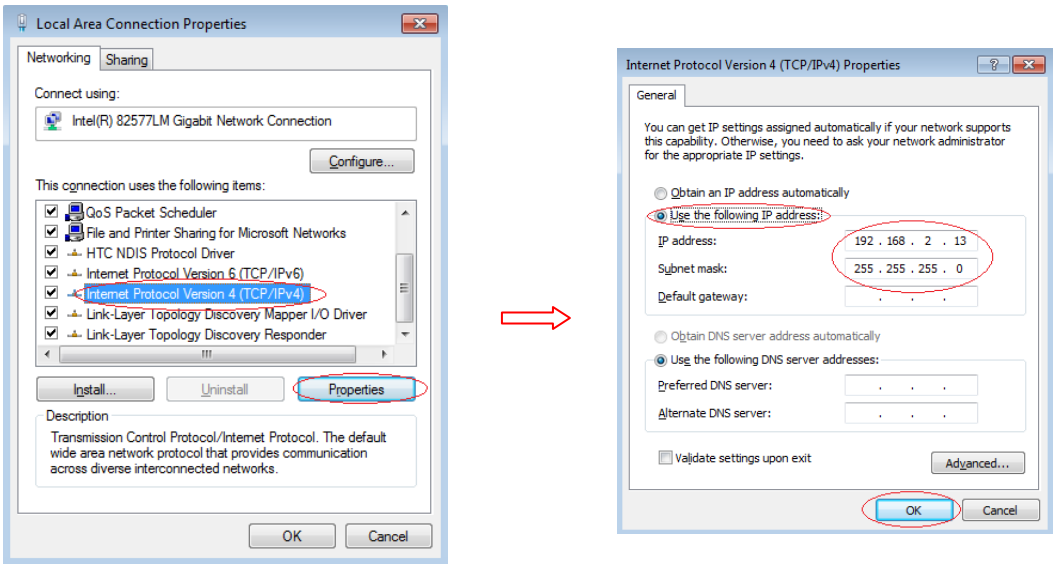
Since here is no router available, you will directly connect the laptop to the target using the provided Ethernet cable.
We will provide the laptop and the target with fixed IP addresses. There is no need for a (Rx/Tx) crossover adaptor since most modern Ethernet PHYs can perform the crossover internally.

Configure the **laptop network adaptor.**

- Type ncpa.cpl in the Windows search field. Press enter. Select the appropriate ethernet adaptor. **Right click** and select **Properties**.

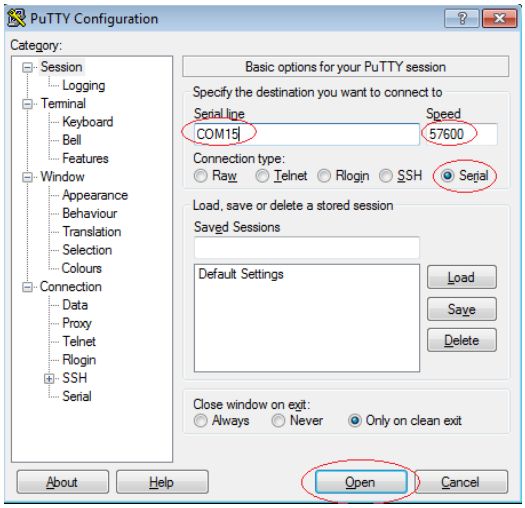


- Select **Internet Protocol Version 4**. Press **Properties**. Setup the IP address as shown below (192.168.2.13). Press **OK**.



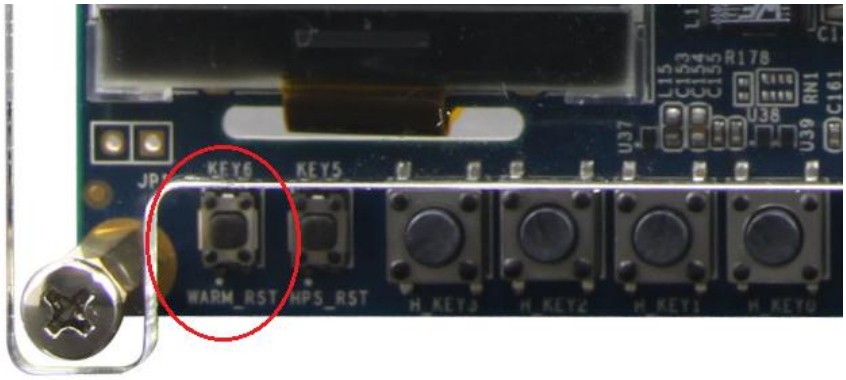
2. Connect to the Linux target (SoCKit).

- Open **PuTTY. Set it to Serial, 115200, COMxx**

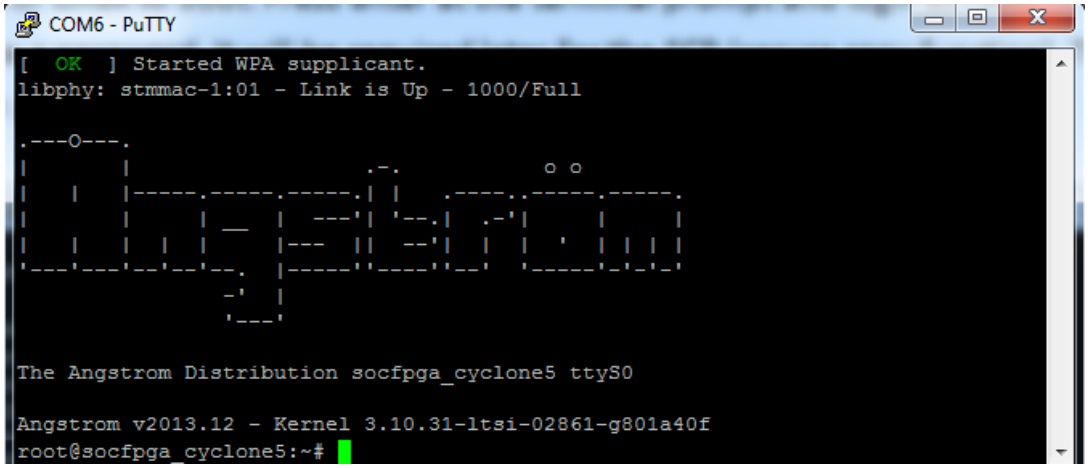


3. Warm reset and boot Linux

- **Insert the SD Card.**
- **Press the WARM_RST button. It is located** on the **bottom left corner** of the SoCKit. See the snapshot below.

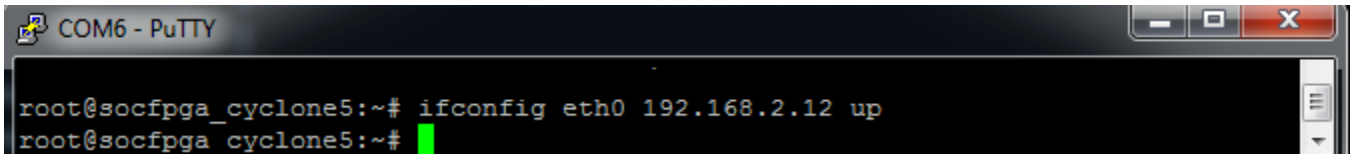


- Wait for Linux to boot. Press enter at the terminal prompt and login as root.
- Create a password. It will be required later for the SCP (secure copy function). Type "passwd" and enter root when prompted.

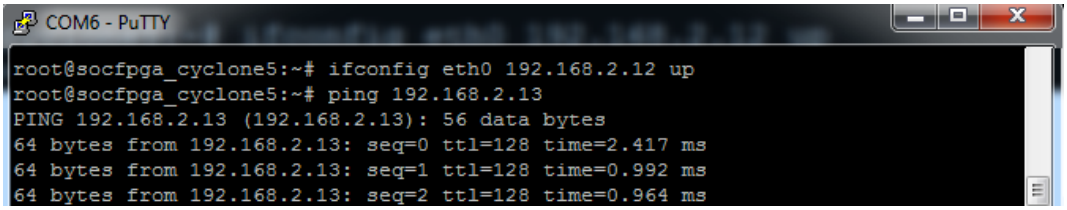


Assign the target board a fixed IP address

- At the prompt type **ifconfig eth0 192.168.2.12 up.** Press enter.



- Ping the host. Type **ping 192.168.2.13.** Press enter. Press Ctl C to abort ping.

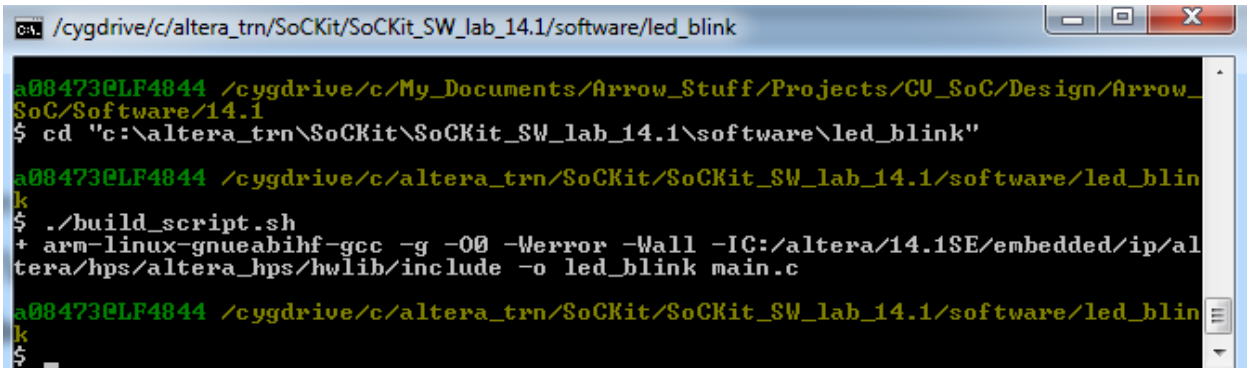


4. Halt the 'scroll_server' led process and clear the leds

- Type **./init_leds.sh**. Press enter

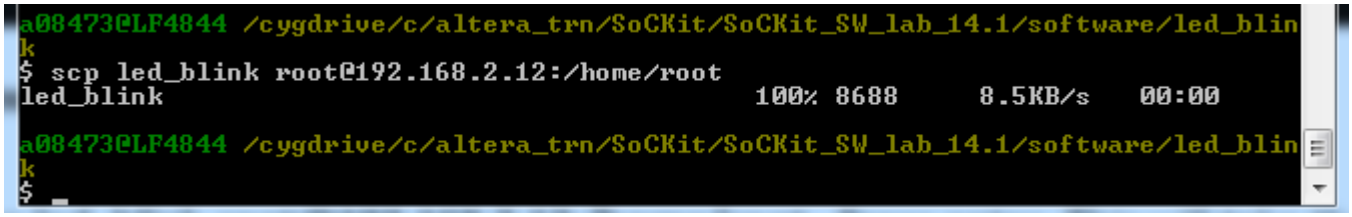
5. Build the "led_blink" example

- Open an **Embedded Shell**
- CD to **c:\altera_trn\SoCKit\SoCKit_SW_lab_14.1\software\led_blink**
- Type **./build_script.sh** and press enter.



6. Use **SCP** to copy the **executable** to the **target** via Ethernet.

- Type **scp led_blink root@192.168.2.12:/home/root**. Press enter. This will take the local file "led_blink" and securely copy it to the target at 192.168.2.12. It will place it in the **/home/root folder**.



- Navigate back to the target console.
- Type ls at the prompt.
- Change the permissions of led_blink. At the prompt type **chmod 555 led_blink**. Press enter.

7. Execute the led_blink application.

- Type **./led_blink** at the **target console** prompt. Press enter. The LEDs will blink for a few seconds.

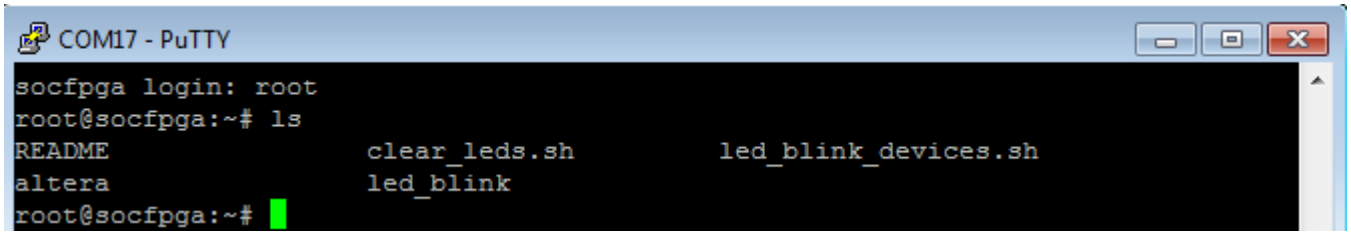
4.3 Validate the FPGA Peripherals using Linux Device Drivers (Modules)

In this module you will run a few **shell scripts**. These scripts will turn the **FPGA and HPS LEDs on and off**. The difference in this exercise is that there is **no explicit reference** to memory **map addresses or bit locations**. You will also install a **module** that **registers an interrupt** and prints a **message** when that interrupt occurs

1. Examine the installed devices

All the drivers associated with LEDs and gpios are loaded with the linux kernel and when the gsrd_init.sh script is loaded as part of the system initialisation at boot up.

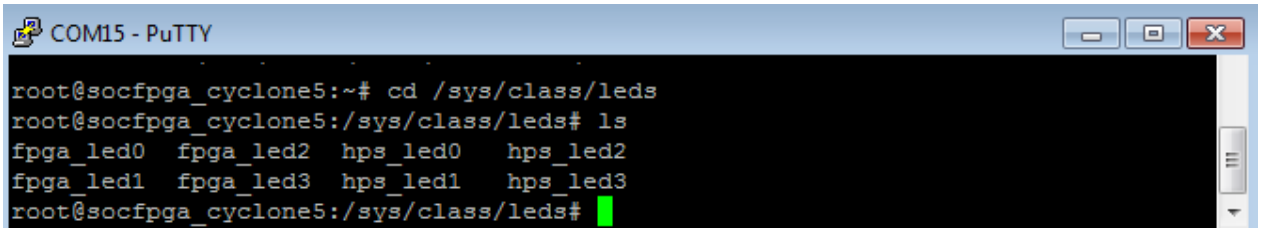
- Bring the PuTTY console to the foreground. Type **cd ~** . Press enter.
- Type **ls**. Press enter. Examine the directory contents.



- Type **cd /sys/class/leds**. Press enter. **Type ls**. Press enter.

Notice how each led (hps or fpga) now appears as an individual device. Take note of the naming syntax.

- Type **cd~** . Press enter.



2. Run the led_blink_devices script

This script will blink all the fpga and hps LEDs.

- Type **cat led_blink_devices.sh.** Press enter. Examine the **contents** of the script.

Notice that the echo command is being used to pipe data to each individual led (fpga & hps). There is **no knowledge** of the **custom FPGA hardware** that was created using **Qsys**. There is also no knowledge of the **custom GPIO assignments** that were made for the **HPS leds**. In the next section you will examine how the **driver** gets this information **from the Qsys system tool**.

- Type **source ./led_blink_devices.sh**. Press enter.

3. Detect the user pushbutton

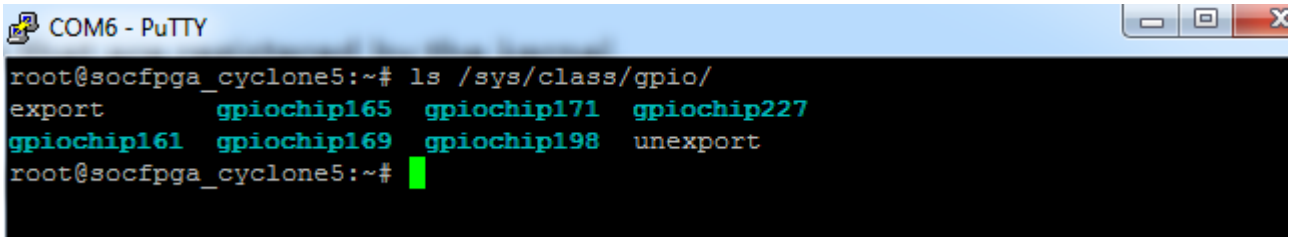
Install the **gpio_interrupt** module. The module is **installed** using the following **syntax**

```
modprobe gpio_interrupt gpio_number=<n>
```

GPIO numbers are automatically assigned by the **kernel** based on **device tree** entries. The **GPIO number** must be correlated with its **associated gpiochip** in order to determine which **interrupt** is being asserted.

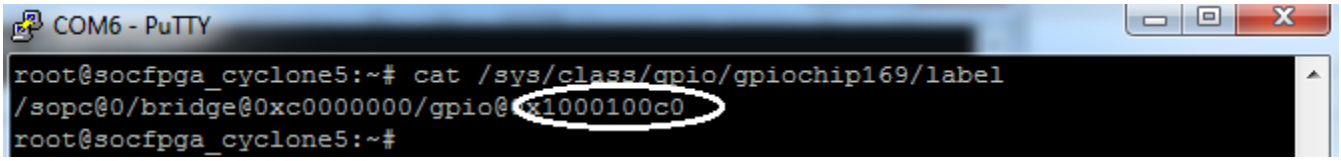
Examine all the available gpiochips's that are registered by the kernel.

- Type **'ls /sys/class/gpio'** at the prompt. Press **enter**.



Match the label of the GPIO chip to the address of push button and DIP switch in device tree

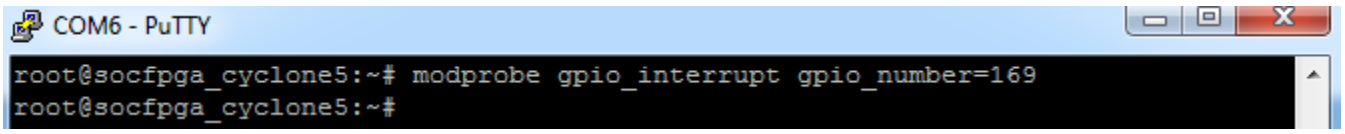
- Type **'cat /sys/class/gpio/gpiochip169/label'** at the prompt. Press **enter**



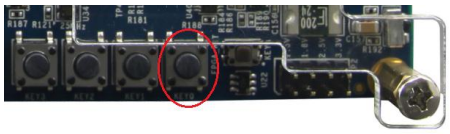
Note that the offset address of the pushbutton (button_pio) in the FPGA match those of gpiochip169. A match has been found.

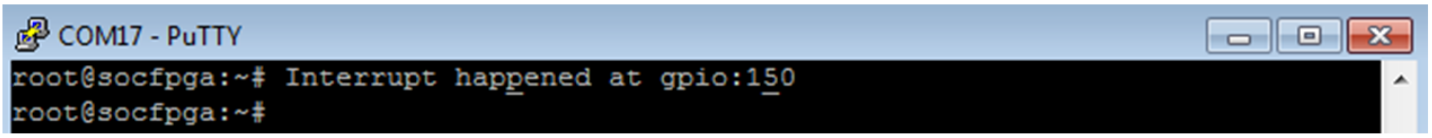
Register **gpiochip169** with the **gpio_interrupt module** in order to detect any pushbutton interrupts. Since there are **two pushbutton inputs** in the button_pio component, gpio_numbers **169 and 170** are allocated to **gpiochip169**.

- Type **'modprobe gpio_interrupt gpio_number=169'** at the prompt. Press **enter**.



- Press the the **pushbutton 0** on the SoCKit board to activate the interrupt



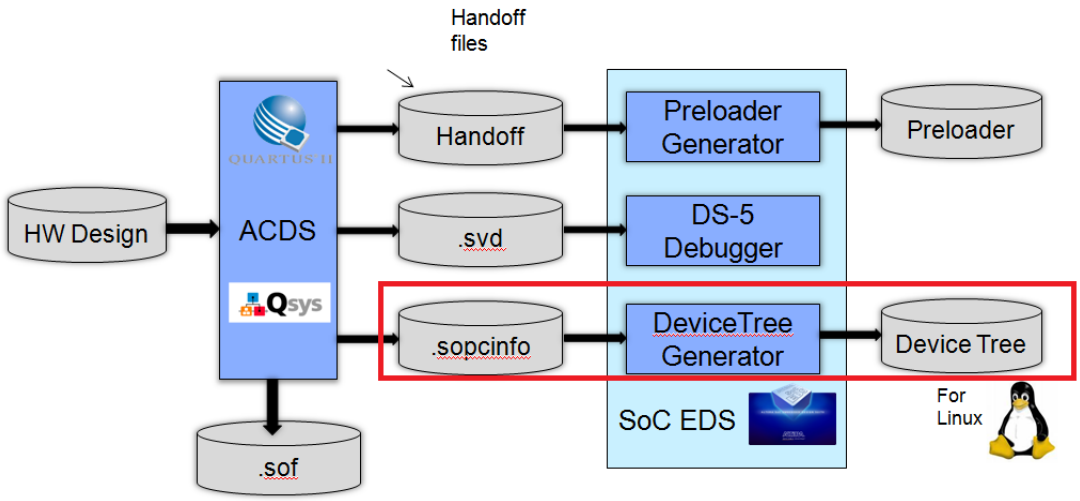


Remove the **gpio_interrupt**.

- Type **'rmmod gpio_interrupt'** at the prompt. Press **enter**

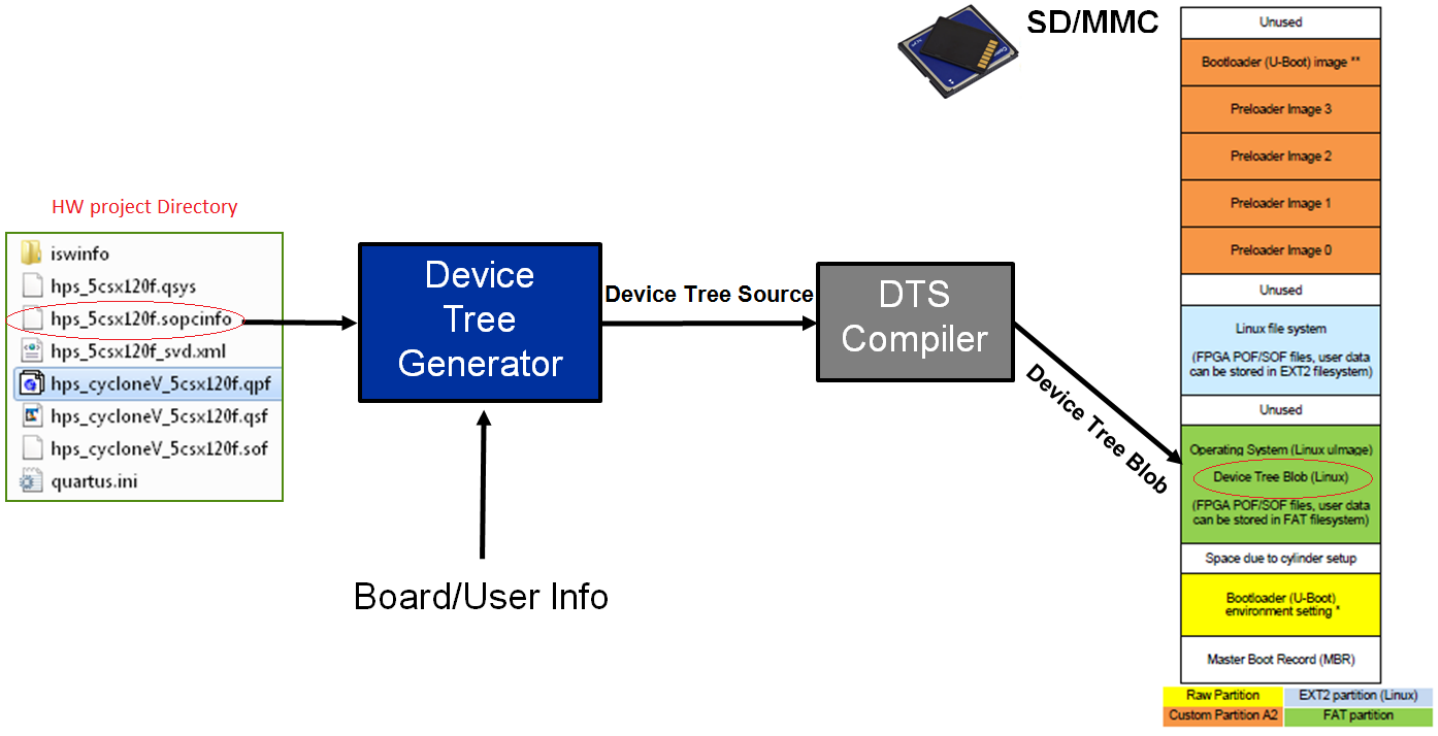
4.4 Examine the Device Tree Blob (DTB)

This section focuses on the flow of system information from the .sopcinfo file to the Device Tree.



The Device Tree standard specifies hardware connectivity so that Linux kernel can boot up correctly. For more on the device tree click on this link *Devicetree.org*

The diagram below shows the detailed connection from the Qsys system definition file (.sopcinfo) to the Device Tree Source (DTS) file, which is readable text, and finally to the Device Tree Blob (DTB) which is a binary format. The DTB is placed in the FAT partition of the SD card and is read by U-Boot and placed in DDR3 memory. It is read by the Linux kernel at boot time.



1. Examine the Device Tree Source (DTS)

Examine a section of the Device Tree Source file for the SoCKit. This section describes the LEDs connected to the FPGA and to the HPS.

As seen in Module 4.3 a high level device access requires no specific hardware knowledge of that device. That specific hardware knowledge is passed from the HW design via the sopcinfo file and placed in the DTS file. The kernel reads that information and passes it to the specific module (device driver).

Examine **fpga_led3** and **hps_led3**. The DTS entry for fpga_led3 specifies that it is connected the LED_PIO peripheral on bit 3. LED_PIO was added to the system using Qsys in the HW lab section. The base address offset for the LED_PIO is also specified in the DTS.

In the case of hps_led3 the DTS indicates that it is connected to the GPIO pin that is driven by GPIO register 1 on bit 24. The base address offset for GPIO register 1 is also specified in the DTS.

Automatic generation of the DTS is **now** supported in Quartus II 14.1.

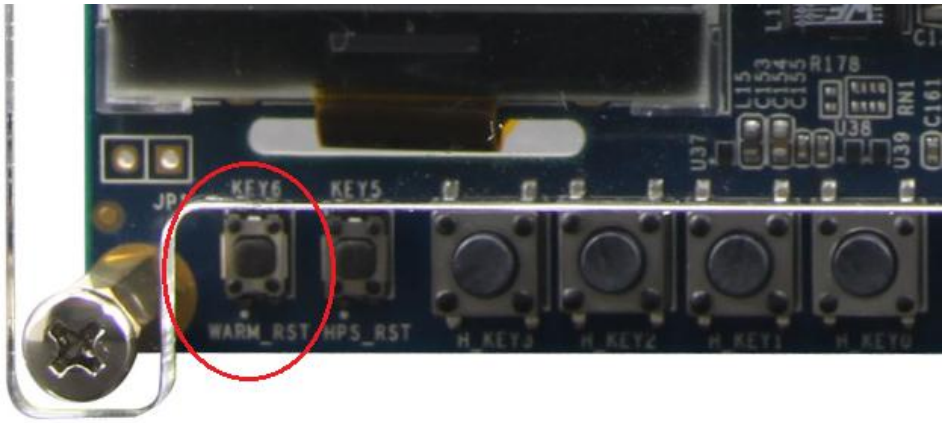
soc_system.dts

```
leds {
      compatible = "gpio-leds";
      fpga0 {
            gpios = <&led_pio 0 1>;
            label = "fpga_led0";
      };
      fpga1 {
            gpios = <&led_pio 1 1>;
            label = "fpga_led1";
      };
      fpga2 {
            gpios = <&led_pio 2 1>;
            label = "fpga_led2";
      };
      fpga3 {
            gpios = <&led_pio 3 1>;
            label = "fpga_led3";
      };
      hps0 {
            gpios = <&hps_0_gpio1 27 1>;
            label = "hps_led0";
      };
      hps1 {
            gpios = <&hps_0_gpio1 26 1>;
            label = "hps_led1";
      };
      hps2 {
            gpios = <&hps_0_gpio1 25 1>;
            label = "hps_led2";
      };
      hps3 {
            gpios = <&hps_0 gpio1 24 1>;
            label = "hps_led3";
      };
```

2. Examine the Device Tree Blob (DTB) in U-Boot

Reboot the SoCKit and **halt** U-Boot before it loads Linux

- Type **poweroff**. Press enter. Wait until you see **"System halted"**
- Press the **WARM_RST** button and then **press any key** (within 5 seconds) to **halt** U-Boot autoboot. The WARM_RST button is located on the **bottom left corner** of the SoCKit. See the snapshot below.




```
In:    serial
Out:   serial
Err:   serial
Net:   mii0
Warning: failed to set MAC address

Hit any key to stop autoboot:  0
SOCFPGA_CYCLONE5 #
```

Examine the contents of the SD card FAT partition.

- Type **fatls mmc 0:1**. Press enter. This displays the **contents** of the **fat partition** on the **SD card**.

```
Hit any key to stop autoboot:  0
ARROW_SOCKIT # fatls mmc 0:1
  3620800   zimage
      229   u-boot.scr
  2358007   soc_system.rbf
    21028   soc_system.dtb

4 file(s), 0 dir(s)

ARROW_SOCKIT #
```

Load the Device Tree Blob into Memory.

- Type **fatload mmc 0:1 0x100 soc_system.dtb**. Press enter. This **loads** the **DTB** from **the SD card** and places it in **DDR3 memory at 0x100**.

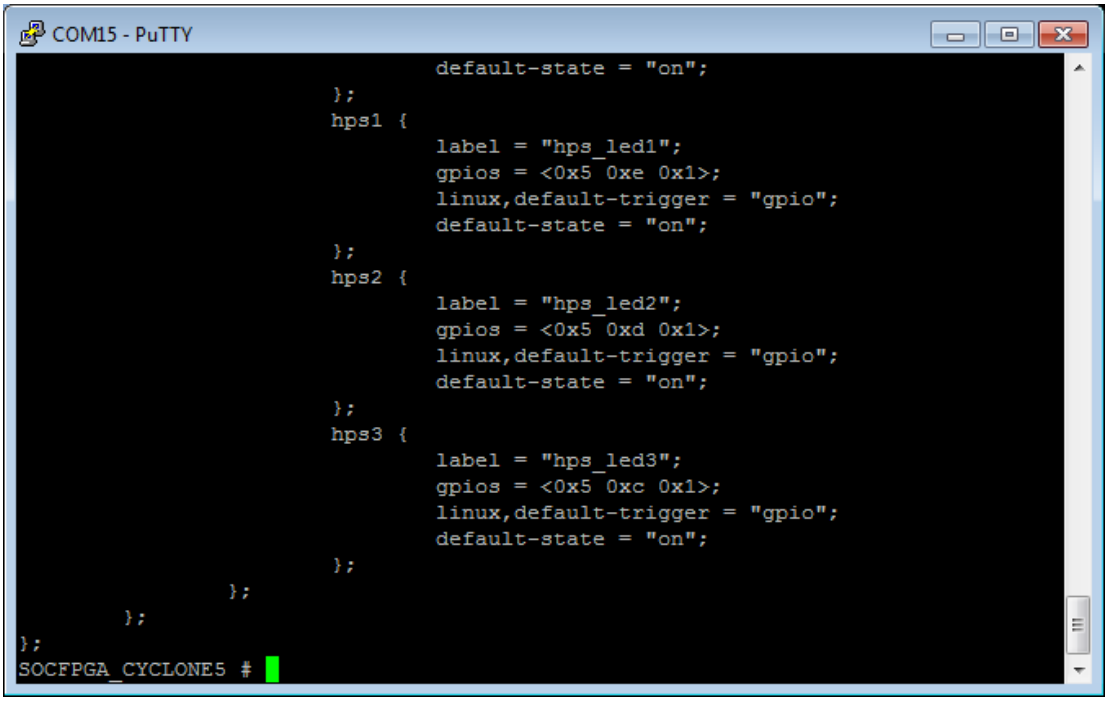
```
SOCFPGA_CYCLONE5 # fatload mmc 0:1 0x100 soc_system.dtb
reading soc_system.dtb
21028 bytes read in 9 ms (2.2 MiB/s)
SOCFPGA_CYCLONE5 #
```

Examine the contents of the Device Tree Blob

- Type **fdt addr 0x100.** Press enter. This assigns 0x100 to the **fdt** system **variable addr**.

```
8620 bytes read
SOCFPGA_CYCLONE5 # fdt addr 0x100
SOCFPGA_CYCLONE5 #
```

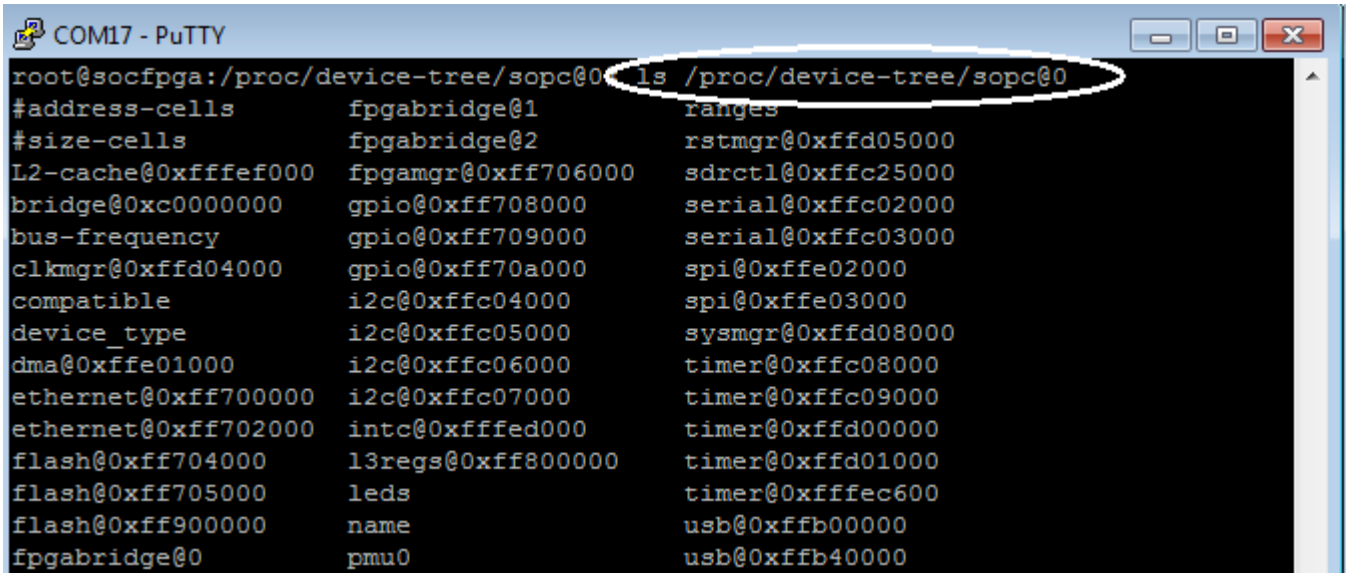
- Type **fdt print.** Press enter. This reads the binary DTB converts it to clear text and displays it. Wait for the text to stop scrolling. The content on the display will now look familiar. This is exactly what the kernel will see but in a binary format.



2. Examine the Device Tree in Linux

The device tree can also be viewed from within Linux.

- Type **bootd** at the **u-boot** prompt. This will boot linux from the SD card.
- login as **root**
- Type **ls /proc/device-tree/sopc@0.** Press **Enter.**



CONGRATULATIONS!!

You have validated the FPGA peripherals

For more detailed information on how to build u-boot and Linux for the SoCKit please visit the Golden System Reference Design (GSRD) page for the SoCKit on rocketboards.org

MODULE 5: Taking the Next Step

Altera has a number of resource available to assist you in further product development at www.altera.com/embedded

Some of the resources available are:

Visit the rocketboards.org community web site

http://www.rocketboards.org/foswiki

**** Start here ****
http://rocketboards.org/foswiki/Documentation/GSRDArrowSoCKitEdition

Arrow SoCKit Evaluation Board support site
http://www.rocketboards.org/foswiki/Documentation/ArrowSoCKitEvaluationBoard

Altera SoC Development Board support site
http://www.rocketboards.org/foswiki/Documentation/AlteraSoCDevelopmentBoard

Get more information about the SoC HPS

Hard Processor System Technical Reference Manual
http://www.altera.com/literature/hb/cyclone-v/cv_5v4.pdf

Get more information about the SoC Embedded Design Tools

Embedded Software for the Cortex-A9 MPCore Processor
http://www.altera.com/devices/processor/arm/cortex-a9/software/proc-a9-embedded-software.html

Get additional SoC training (discounted from $695 per course to $99 for workshop attendees)

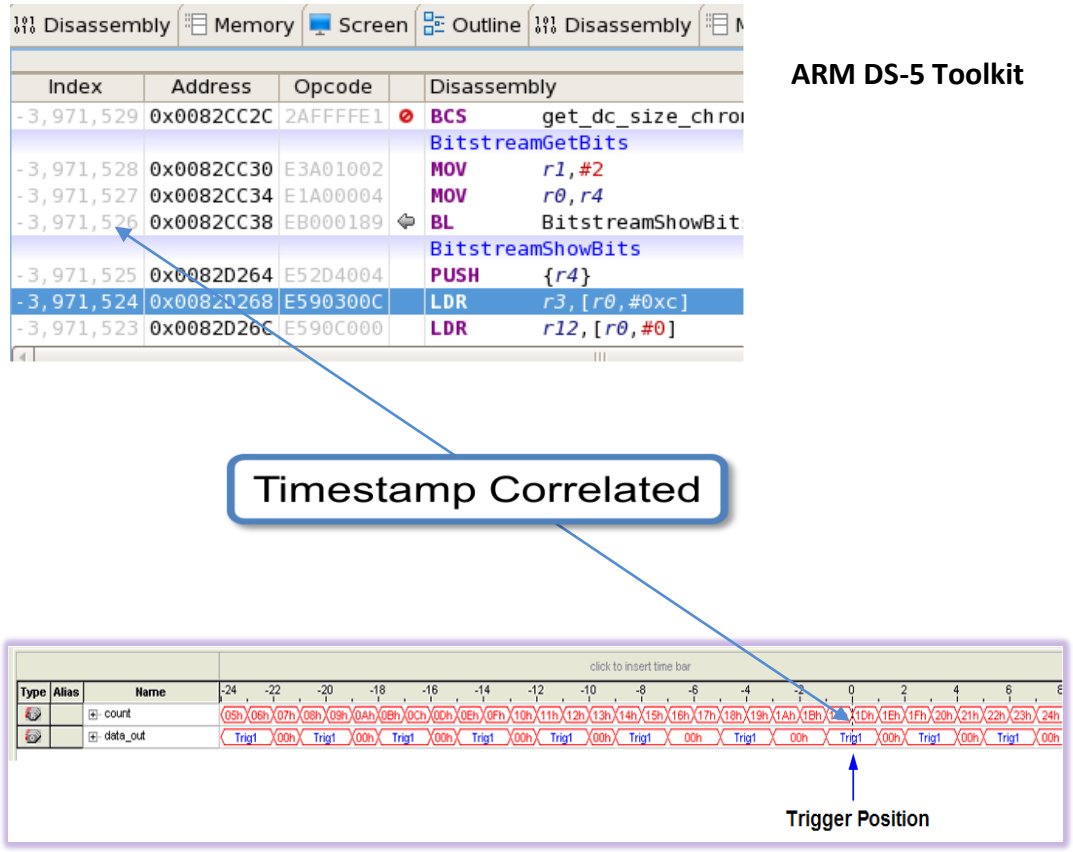
Designing with an ARM based SoC
http://www.altera.com/education/training/courses/ISOC101

Developing Software for an ARM based SoC
http://www.altera.com/education/training/courses/ISOC102

For all resources visit www.altera.com/embedded

MODULE 6: Cross Triggering (Do at home Exercise)

Working with the **Altera SignalTap™ II Logic Analyzer**, the **DS-5** toolkit provides advanced, **signal-level hardware cross triggering** between the **CPU** and **FPGA** domains. Using this capability, the **software** and **FPGA** designers can **analyze** the **captured trace** and co-debug across hardware-to-software bounds. In this Module you will first **learn** how to use **SignalTap II** to **trigger** the **DS-5** tool. You will then use a **breakpoint** or a manual trigger to **cross trigger SignalTap II.**



ARM DS-5 Toolkit

SignalTap II Logic Analyzer

Perform these steps **first**.

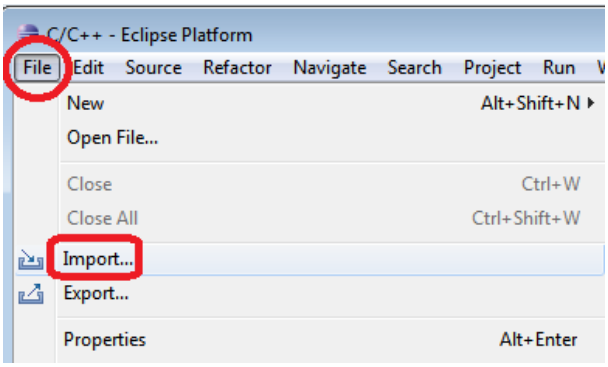
1. **Power** on the SoCKit
2. **Boot** to the Linux **prompt**.
3. Login as **root**
4. Kill the led scroll server. Type '**./init_leds.sh'** at the linux prompt.
4. Launch DS-5

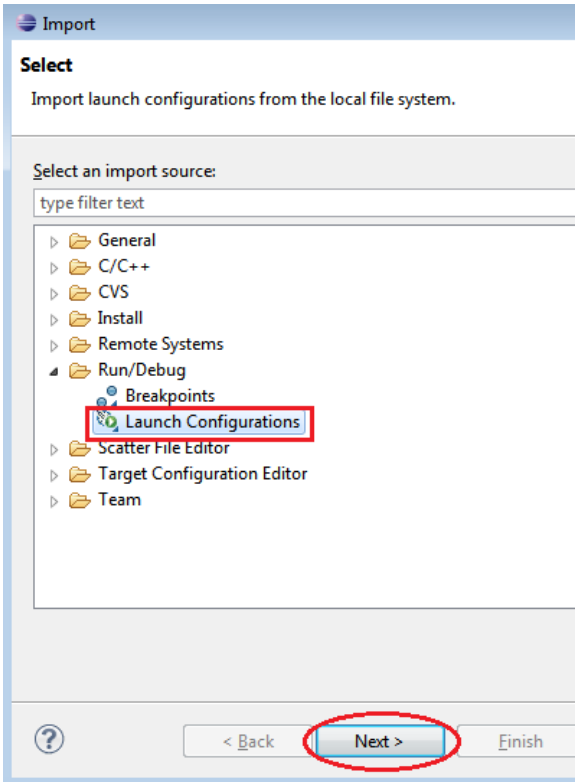
6.1 Configure Cross Triggering on the HPS

Configurations can be exported and imported within DS-5. This can be convenient when working as a team. In the context of this lab it assists since the user can re-use an existing configuration. In this instance you will be tracing and cross triggering the Linux kernel.

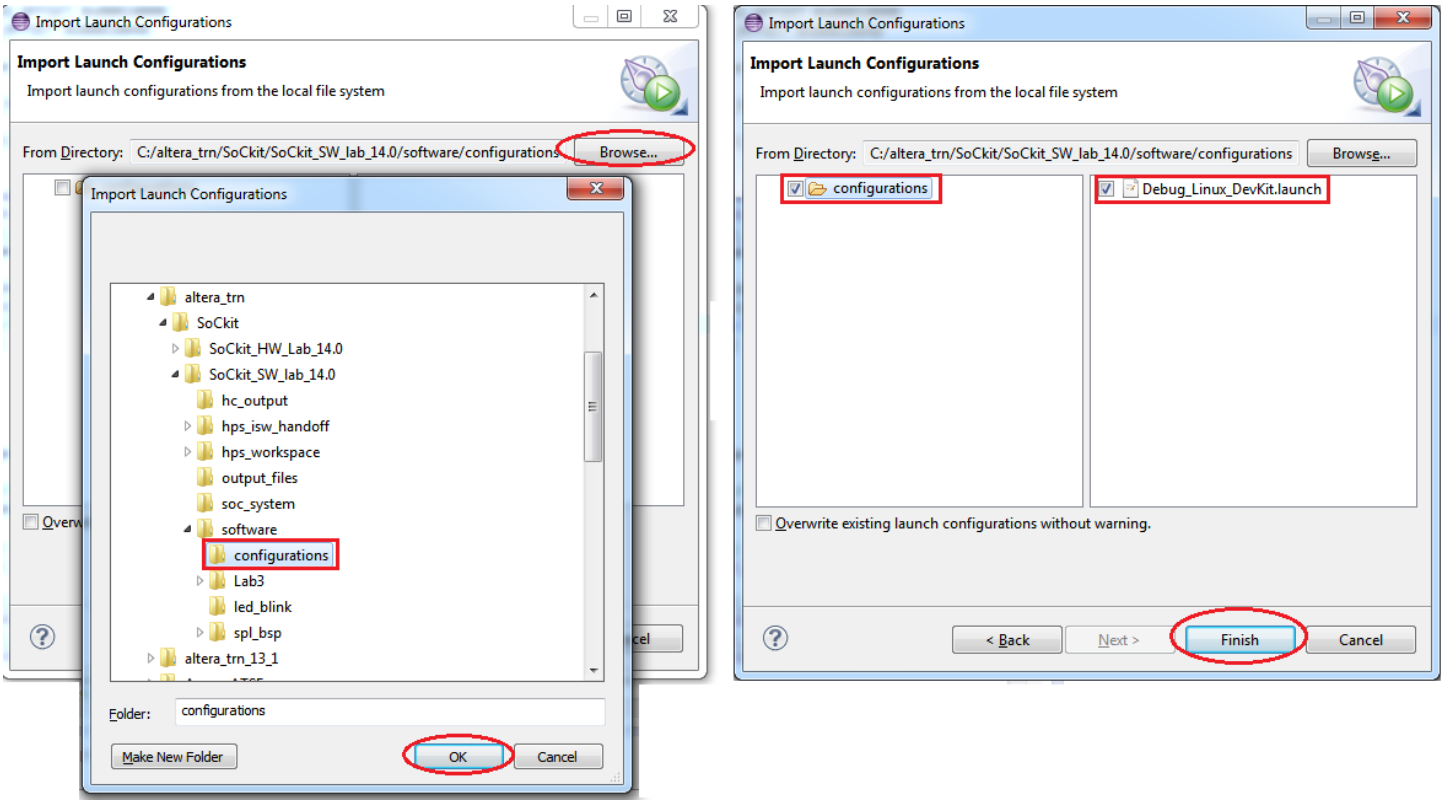
1. **Import** the desired **Debug Launch** Configuration.

- From the menu select **File --> Import**. Select **Launch Configurations**. Press **Next**.



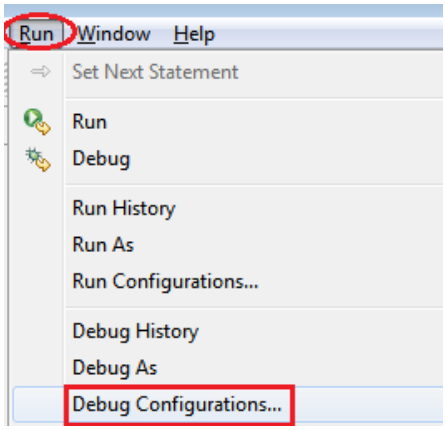


- Press the **Browse** button and **navigate** to the **configurations** sub **directory**. Press **OK**. Select both **check** boxes in the **Import Launch Configurations** window and press **Finish**.

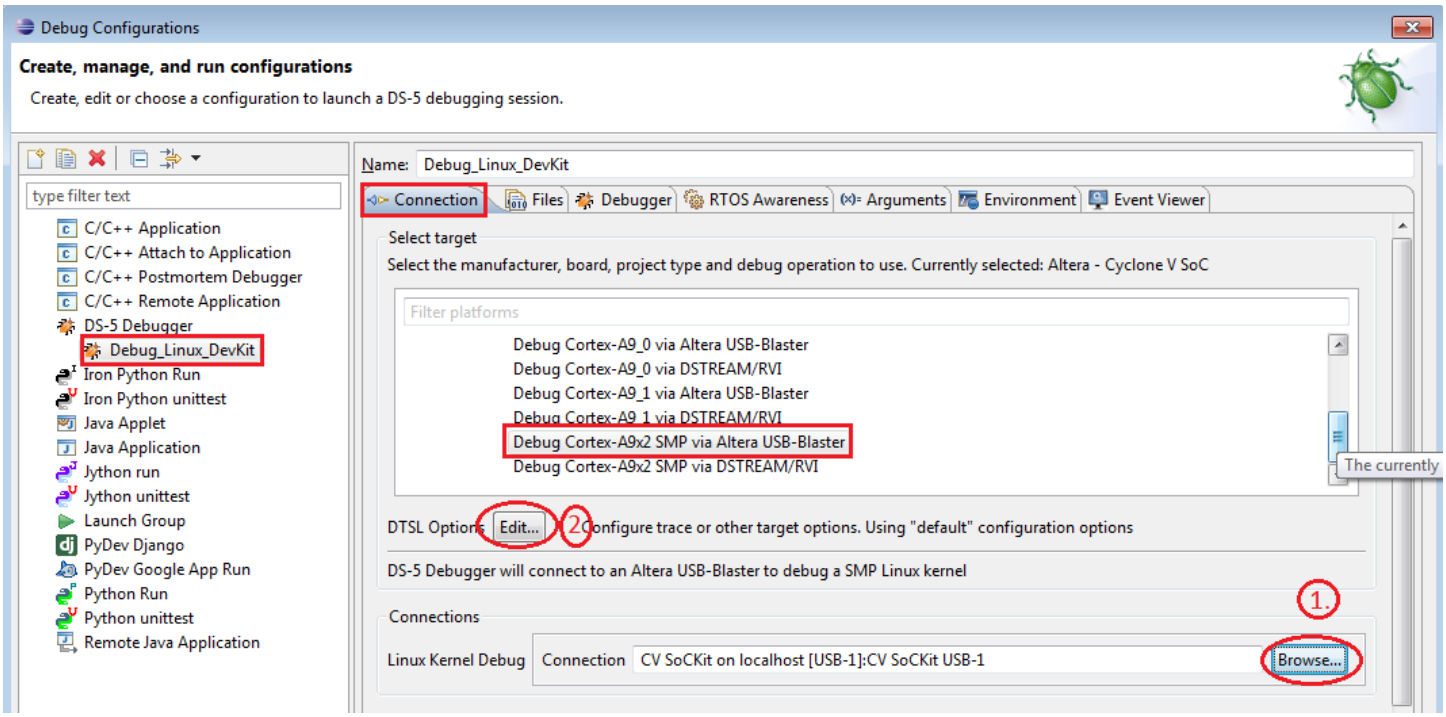


2. **Review** the **imported** debug launch **configuration**.

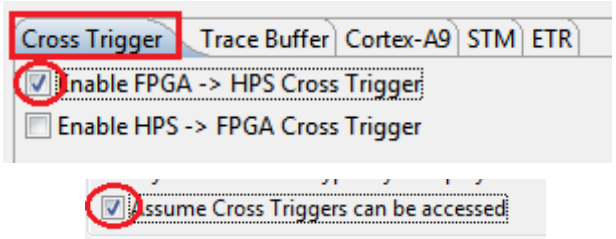
- From the menu select **Run --> Debug Configurations**



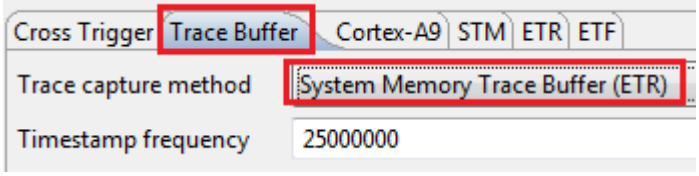
- From the menu select **Run --> Debug Configurations**
- Select the "**Debug_Linux_DevKit**" configuration
- Select the **Connection** tab
- **Refresh** the connection. Press the **Browse** button. Select the **Debug Harware**. Press **OK**.



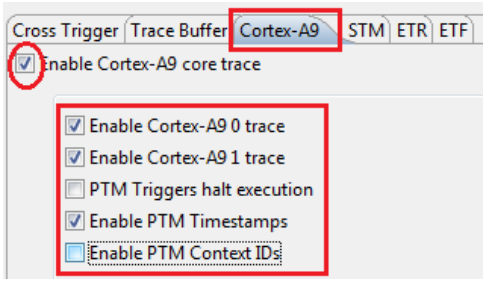
- Examine the **DSTL** options. Press the DSTL **Edit** button.
- Check the Enable **FPGA --> HPS Cross Trigger** for the first **example.**
- Check **Assume Cross Triggers can be accessed.**



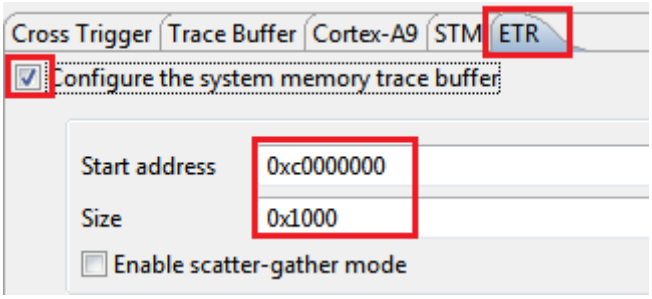
- Select the **Trace Buffer** tab. Select the **System Memory** Trace Buffer (ETR).



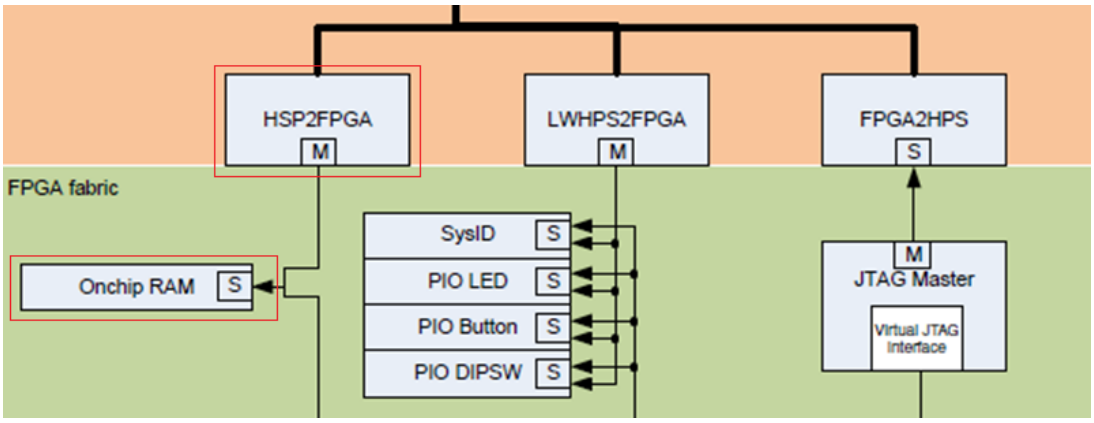
- Select the **Cortex-A9** tab. **Check** the **options** as shown **below**.

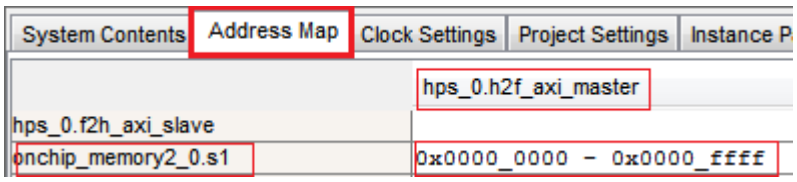


- Select the **ETR** tab. Press **OK**.



A 4KB **Embedded Trace Buffer** (ETR) has been selected at system address **0xC000 0000**. **Where** is this buffer **physically** located? Recall the **GHRD** block diagram. The **HPS2FPGA** bridge is located in the **HPS** memory **map** at address 0xC0000 0000. The **Onchip** RAM was added to the design in **Qsys** and is located at HPS2FPGA Bridge offset 0x0000 0000.





6.2 Configure Cross Triggering on the FPGA

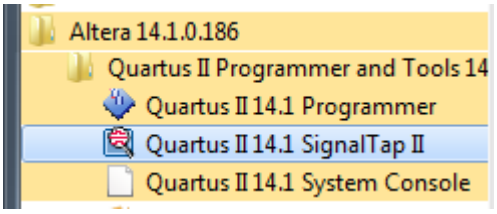
SignalTap II allows developers to embed a Logic Analyzer within the FPGA. It has the ability to monitor and capture FPGA signal activity at full data rates. Signals of interest are defined by the designer as are the trigger settings. SignalTap II can receive external triggers and also transmit triggers to other applications.

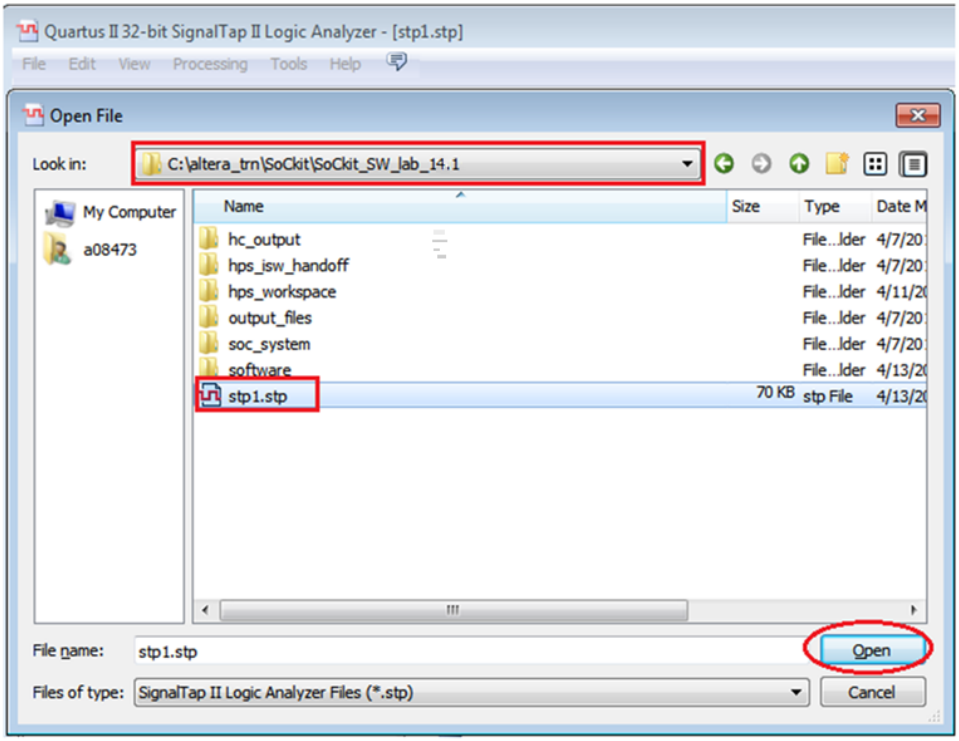
1. **Launch SignalTap II.**

- **Start --> All Programs --> Altera 14.1.0.186 --> Quartus II Programmer and SignalTap II 14.1.0.186 --> Quartus II 14.1 SignalTap**

2. **Load the SignalTap II definition (stp) file**

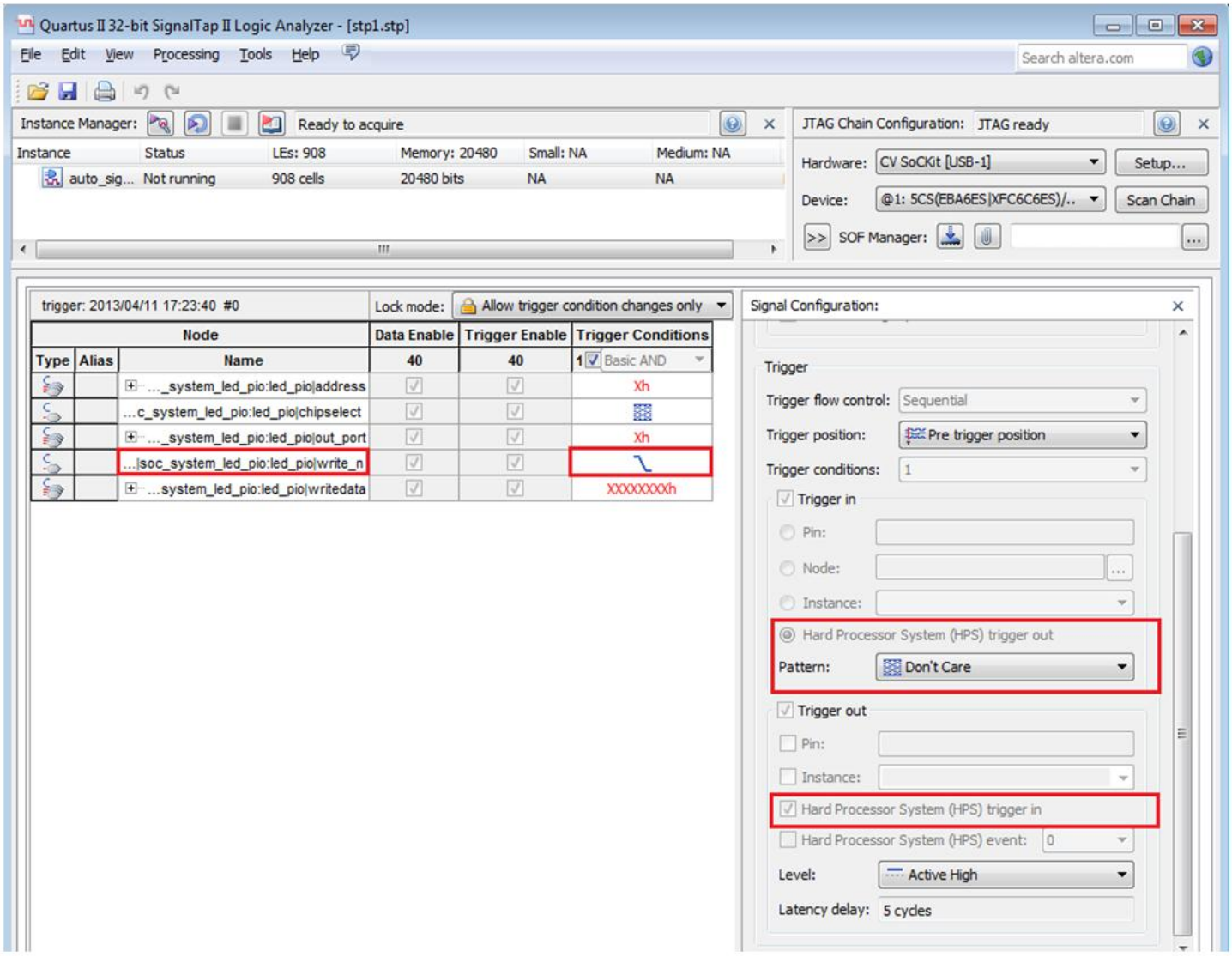
- **File --> Open.** Navigate to **c:\altera_trn\SoCKit\SoCKit_SW_lab_14.1**
- **Select** the **stp1.stp** file. Press the **Open** button





3. **Observe the SignalTap II setup**

- Five led_pio peripheral signals have been selected: address, chipselect, out_port, **write_n** and writedata.
- Notice that the **trigger** is set on the **falling edge** of **write_n.**
- Notice that both the **HPS trigger out** and **HPS trigger in** options have been **enabled.**
- Some of the options are grayed out because you are using the standalone version of SignalTap II.

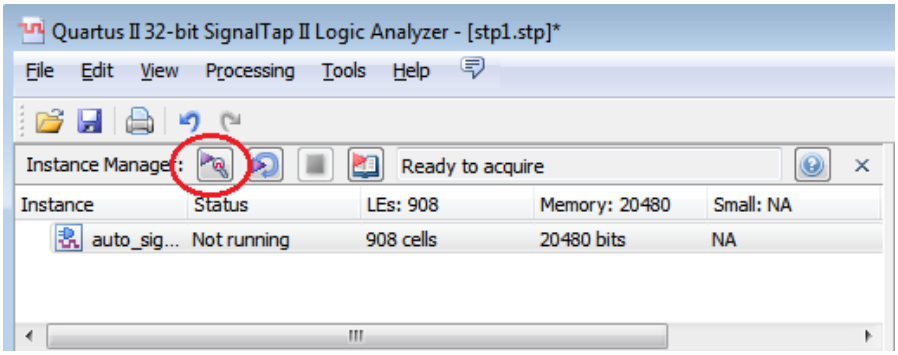


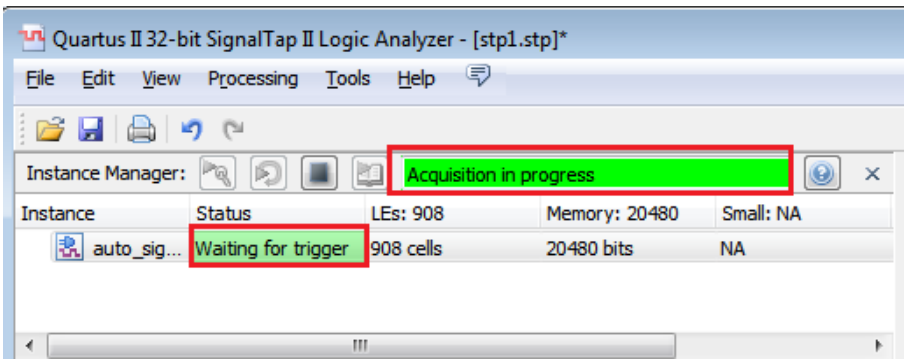
6.3 Cross Triggering Examples:

Example 1: FPGA --> HPS

1. **Arm** the SignalTap II **trigger**

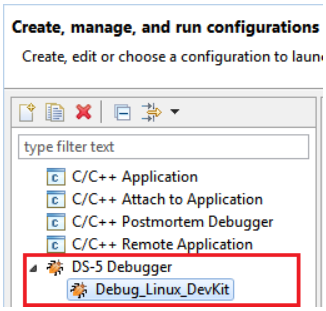
- Press the Run Analysis  button.

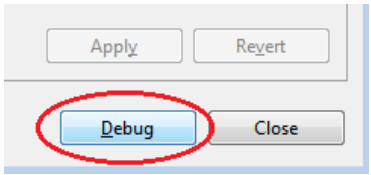




2. Start the DS-5 Debug configuration

- **Run** --> **Debug Configurations**. Select **Debug_Linux_DevKit.** Press **Debug.**

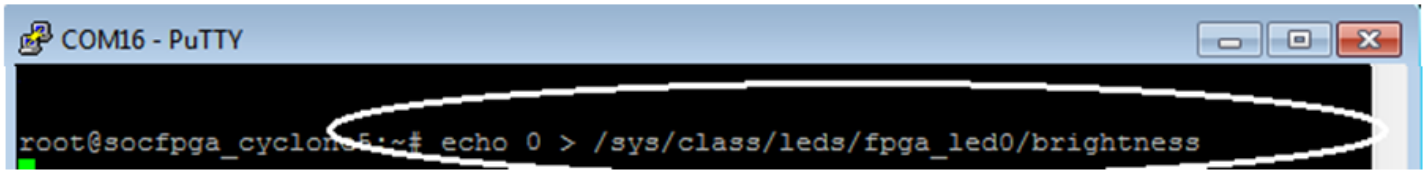




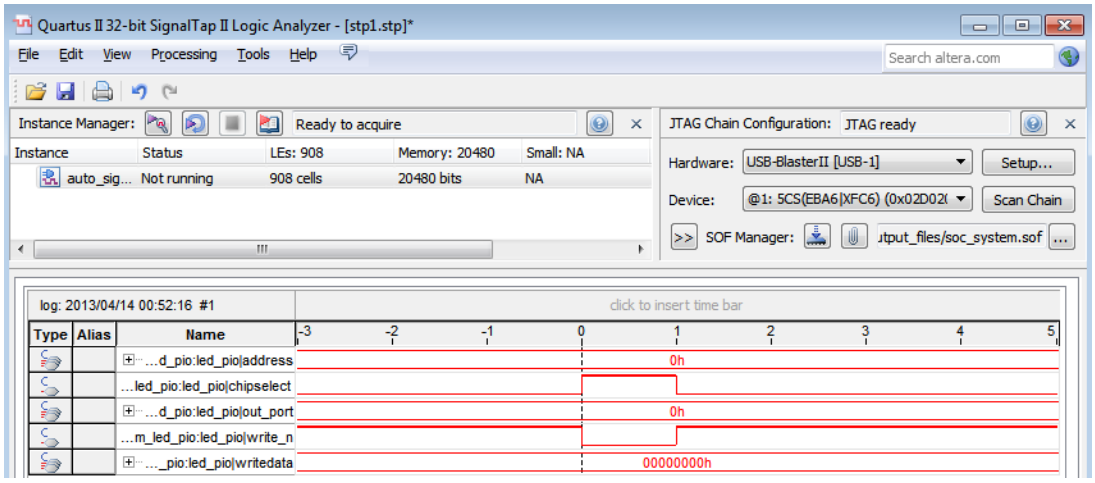
- Allow Linux to **run** by pressing **F8** or the green **Continue** button 

3. From within Linux turn the FPGA led on. This will cause SignalTap to trigger. In turn it will fire a trigger output to the HPS causing it to stop. The debugger will show the state of the two Cortex-A9 cores and will display the trace information.

- Type "**echo 0 > /sys/class/leds/fpga_led0/brightness**" at the Linux prompt. Press **enter**.



- SignalTap triggers on the falling edge of write_n.

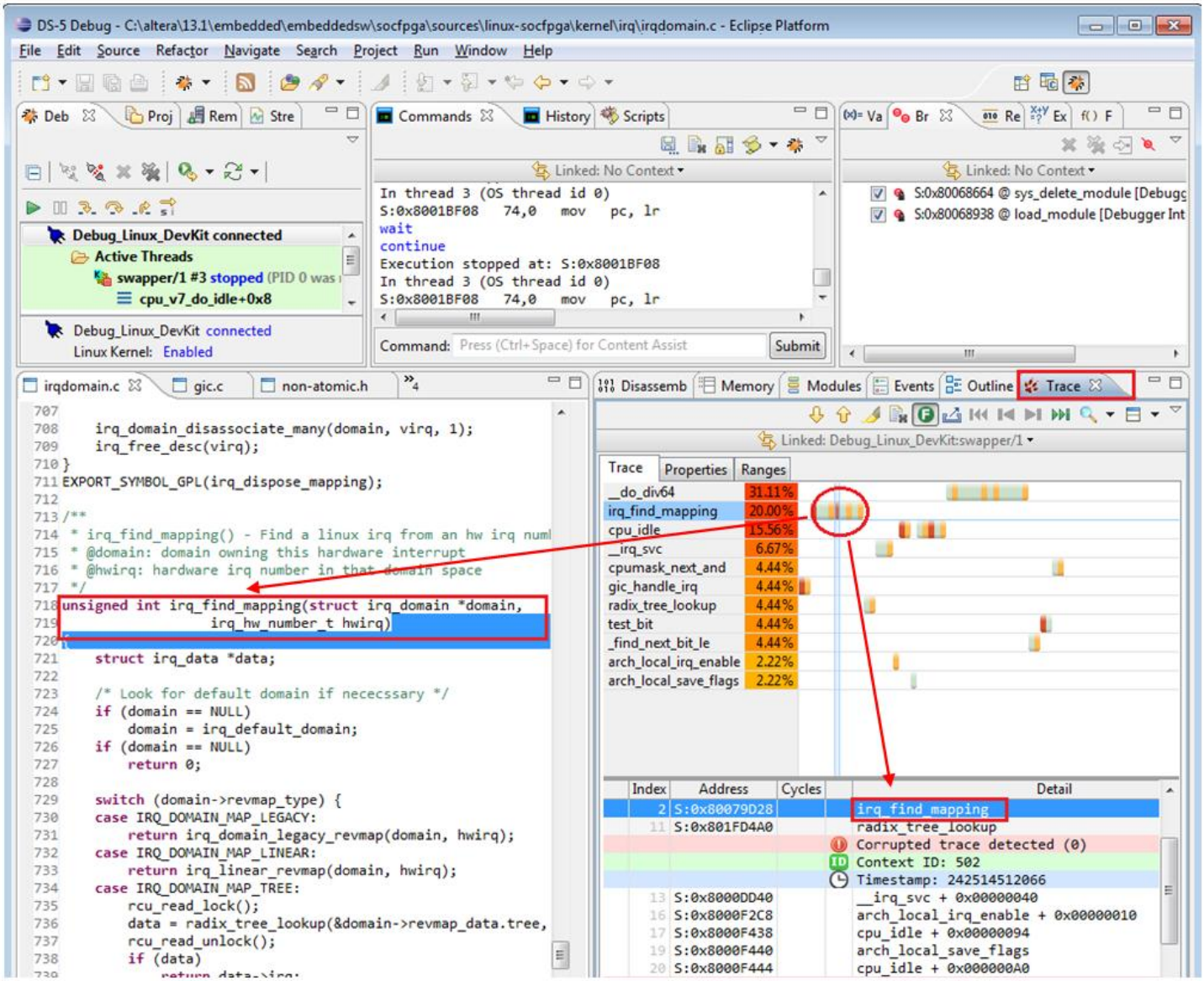


- **SignalTap** sends a **trigger** out signal to the **HPS** which causes a **halts exception** within **DS-5**.

At this point it is worth examining the sequence of events that occurred.

1. The echo command is entered at the Linux prompt
2. The command traverses through the Linux kernel.
3. The appropriate module issues the write led command.
4. SignalTap triggers on the falling edge of the write_n signal and immediately sends a trigger out signal to the HPS. This causes a halt exception within DS-5.
5. DS-5 then follows by capturing the trace and displaying it as seen below.

Note the each time you execute this sequence of events the trace captured by DS-5 will most likely be different because we are tracing the kernel and not an application.



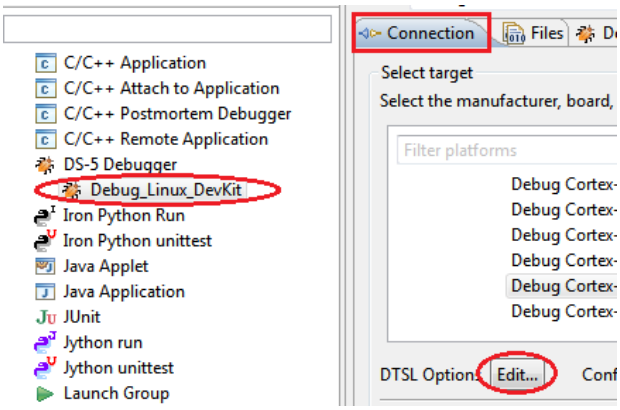
Example 2: HPS --> FPGA

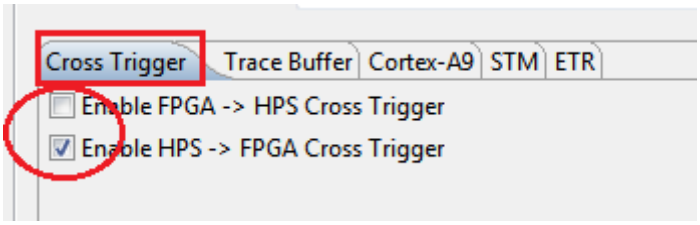
1. Disconnect the DS-5 from the target

- **Press the**  **"Disconnect from Target" button.**
- **Press the "Remove Connection" button.**

2. Modify the DS-5 **Debug Configuration** settings

- Run --> **Debug Configurations**.
- Select the **Debug_Linux_DevKit** configuration.
- Select the **Connection** tab. Press **Edit** to modify the **DSTL** options.
- Select the **Cross Trigger** tab.
- **Disable** the **Enable FPGA -> HPS** Cross Trigger
- **Enable the Enable HPS -> FPGA** Cross Trigger
- Press OK. Press **Debug** to **start** the debug session.
- Press the **F8** (Continue) button.

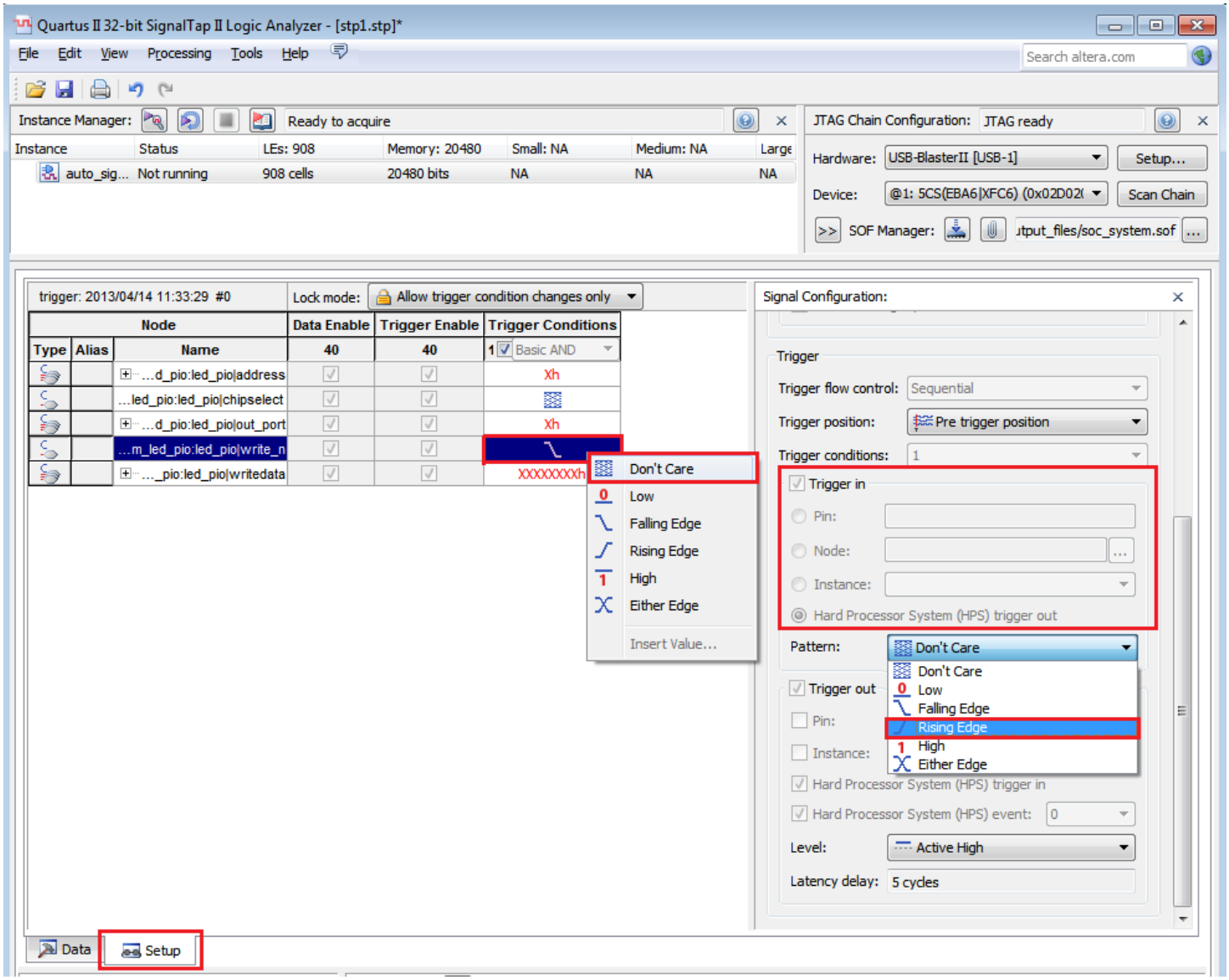




3. Modify the **SignalTap II** settings

Change the trigger **settings** in **SignalTap II**. **Remove** the **write_n falling edge** as a **trigger** and replace it with a **HPS rising edge trigger input**.

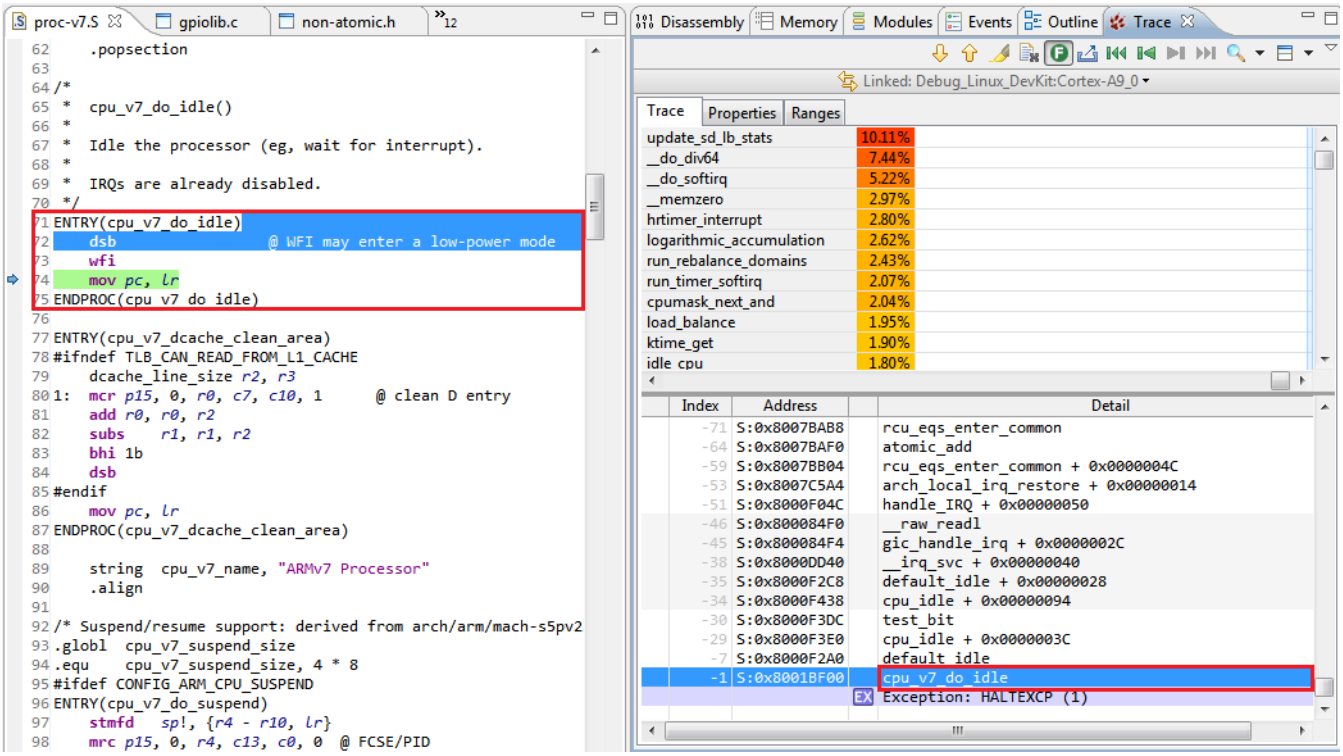
- Bring **SignalTap II** to the **foreground**
- Select the **Setup** tab.
- **Right click** on the cell adjacent to the **write_n node** as shown below. Click on **Don't care**.
- **Click** on the **HPS trigger out Pattern** select as shown below. Select **Rising Edge**.
- Press the **Run Analysis** button.

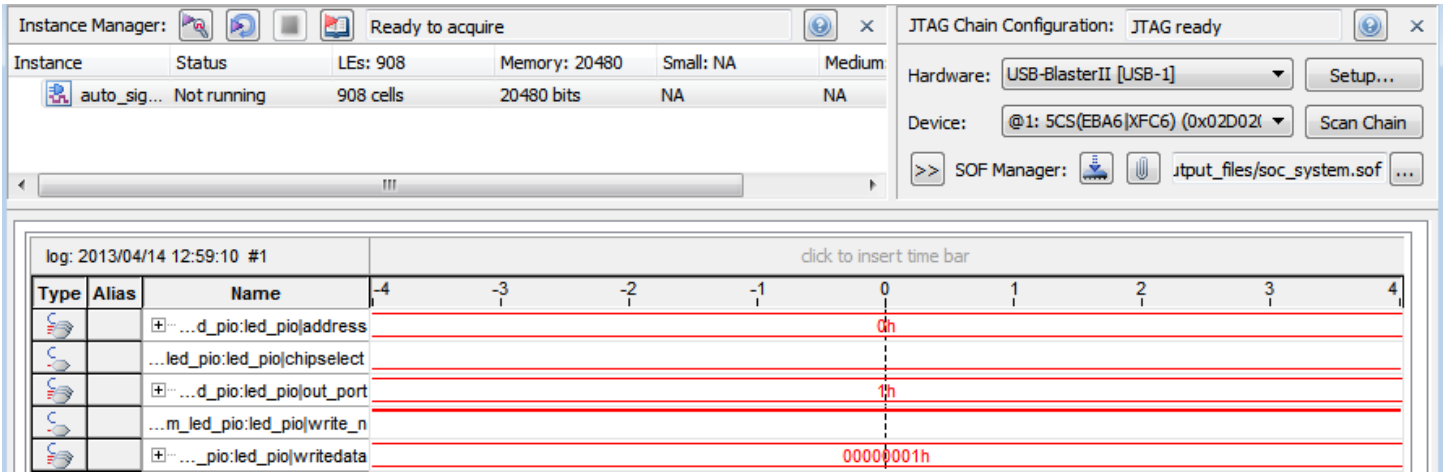


4. **SignalTap will NOT trigger until it receives the trigger** input from the **HPS**. The **HPS** will **transmit the trigger** signal if it hits a **breakpoint** or if it is manually **interrupted**.

In this example you are **arbitrarily** halting the **HPS**. It would be more meaningful if it halted on a breakpoint directly after the led was written to. This is not possible in this scenario since we are source level debugging the kernel and not the gpio-altera module.

- Bring DS-5 to the foreground.
- Break the HPS execution by pressing **F9 (Interrupt)**





CONGRATULATIONS!!

You have successfully used the cross triggering debug tools