ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y SISTEMAS DE TELECOMUNICACIÓN

PROYECTO FIN DE GRADO

TÍTULO: Desarrollo de aplicaciones basadas en *Linux Embedded* en una arquitectura basada en Cyclone V SoC (*System on Chip*) de Altera

AUTOR: D. Marcos Rodríguez Díaz-Delgado

TITULACIÓN: Grado en Ingeniería Electrónica de Comunicaciones

TUTOR: D. Mariano Ruiz González

DEPARTAMENTO: Sistemas Electrónicos y de Control

VºBº

Miembros del Tribunal Calificador:

PRESIDENTE: D. Neftalí Núñez Mendoza

VOCAL: D. Mariano Ruiz González

SECRETARIO: D. Antonio Carpeño Ruiz

Fecha de lectura: 31/03/2014

Calificación:

El Secretario,

ACKNOWLEDGEMENTS

Special appreciation to all my college friends for being there these years, I had a great time in this career thanks to you, I made the best friends I have. I expect to have a lot more great moments.

Special appreciation to my dear family, who always strongly supported me and helped me to do my best, for encouraging me to carry on.

This project would not have been possible without the generous assistance of Mr. Antonio Carpeño Ruiz, who made a great effort to make this project succeed and helped me to overcome all the obstacles I found.

Special appreciation to my tutor Mr. Mariano Ruiz González, and also to professor Mr. Matías Garrido González, for helping with his revision of the first draft of the project report.

Lastly, special appreciation to that special person, for making me so happy.

"Education is an admirable thing, but it is well to remember from time to time that nothing that is worth knowing can be taught."

Oscar Wilde

ABSTRACT

We attempt to integrate and start up the set of necessary tools to deploy the design cycle of embedded systems based on Embedded Linux on a "Cyclone V SoC" made by Altera.

First, we will analyze the available tools for designing the hardware system of the SoCkit development kit, made by Arrow, which has a "Cyclone V SoC" system (based on a "ARM Cortex-A9 MP Core" architecture). When designing the SoCkit board hardware, we will create a new peripheral to integrate it into the hardware system, so it can be used as any other existent resource of the SoCkit board previously configured.

Next, we will analyze the tools to generate an Embedded Linux distribution adapted to the SoCkit board. In order to generate the Linux distribution we will use, on the one hand, a software package from Yocto recommended by Altera; on the other hand, the programs and tools of Altera, Embedded Development Suite. We will integrate all the components needed to build the Embedded Linux distribution, creating a complete and functional system which can be used for developing software applications.

Finally, we will study the programs for developing and debugging applications in C or C++ language that will be executed in this hardware platform, then we will program a Linux application as an example to illustrate the use of SoCkit board resources.

RESUMEN

Se pretende integrar y poner en funcionamiento el conjunto de herramientas necesarias para desplegar el ciclo de diseño de sistemas embebidos basados en "Embedded Linux" sobre una "Cyclone V SoC" de Altera.

En primer lugar, se analizarán las diversas herramientas disponibles para diseñar el sistema hardware de la tarjeta de desarrollo SoCkit, fabricada por Arrow, que dispone de un sistema "Cyclone V SoC" (basado en una arquitectura "ARM Cortex A9 MP Core"). En el diseño hardware de la SoCkit se creará un periférico propio y se integrará en el sistema, pudiendo ser utilizado como cualquier otro recurso de la tarjeta ya existente y configurado.

A continuación, también se analizarán las herramientas para generar una distribución de "Embedded Linux" adaptado a la placa SoCkit. Para generar la distribución de Linux se utilizará, por una parte, un paquete software de Yocto recomendado por Altera y, por otra parte, las propias herramientas y programas de Altera. Se integrarán todos los componentes necesarios para construir la distribución Linux, creando un sistema completo y funcional que se pueda utilizar para el desarrollo de aplicaciones software.

Por último, se estudiarán las herramientas para el diseño y depuración de aplicaciones en lenguaje C ó C++ que se ejecutarán en esta plataforma hardware. Se pretende desarrollar una aplicación de ejemplo para ilustrar el uso de los recursos más utilizados de la SoCkit.

Contents

1. Acronyms

OS	Operative System
PC	Personal Computer
IBM	International Business Machines
GNU	GNU is Not Unix
CPU	Central Processing Unit
ARM	Advanced RISC Machines
RISC	Reduced Instruction Set Computer
QNX	Quantum Software Systems
CE	Embedded Compact
NT	New Technology
PDA	Personal Digital Assistant
GPS	Global Positioning System
SDRAM	Synchronous Dynamic Random Access Memory
PLL	Phase Locked Loop
SD	Secure Digital
MMC	MultiMedia Card
DTB	Device Tree BLOB
BLOB	Binary Large OBject
FDT	Flattened Device Tree
RAM	Random Access Memory
ROM	Read Only Memory
SoC	System on Chip
FPGA	Field Programmable Gate Array
HPS	Hard Processor System
USB	Universial Serial Bus
UART	Universal Asynchronous Receiver-Transmitter
LED	Light-Emitting Diode
EDS	Environment Development System
YSP	Yocto Source Package
DTC	Device Tree Compiler
GHRD	Golden Hardware Reference Design
JTAG	Joint Test Action Group

MPU	Multiple Process Unit
MM	Multi Master
PIO	Peripheral Input/Output
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
TCL	Tool Command Language
BSP	Board Support Package
DTS	Device Tree Source
XML	eXtensible Markup Language
DS	Development Studio
RSE	Remote System Explorer
SSH	Secure SHell
IP	Internet Protocol
TCP	Transmission Control Protocol
SFTP	SSH File Transfer Protocol
HSMC	High Speed Mezzanine Card

2. Figures and tables list

3. Download links

[d1] VMware Player Plus for Windows:
https://my.vmware.com/web/vmware/free#desktop_end_user_computing/vmware_player/6_0

[d2] Ubuntu Desktop: http://www.ubuntu.com/download/desktop

[d3] Quartus II Web Edition: http://dl.altera.com/?edition=web

[d4] SoCkit GHRD: http://rocketboards.org/foswiki/view/Projects/SoCKITGHRD

[d5] "Sopc2dts" development repository: http://git.rocketboards.org/?p=sopc-tools.git;a=tree;f=sopc2dts;h=733dcf5a1a397a40c768bf592a0b2eadc46ab671;hb=HEAD

[d6] How to Use the PIO Button On the SoC Devkit Board (to download XML files):
http://rocketboards.org/foswiki/Documentation/HowToUseThePIOButtonOnTheSocDevkitBoard

4. Introduction

4.1. Antecedents and background

4.1.1. Embedded Systems

The word Embedded Linux refers to the use of Linux kernel-based OS (Operative System) on an embedded system. The first question to answer is: what is an embedded system?

An embedded system has a specific function within an electric or mechanic larger one. It is embedded as part of a whole device. On contrast, a general purpose system like a PC (Personal Computer) is designed to be flexible, with an extremely wide range of applications. Embedded systems comprise a large amount of common use devices nowadays; from portable devices (such as digital watches or mp3 players) to bigger fixed installations and highly complex systems (semaphores, magnetic resonance equipment, systems embedded into vehicles, etc). Control or industrial automation equipment, general electronic devices or microprocessor applications are embedded systems as well.

The key idea is a system dedicated to handle a specific task. Since embedded systems do specific tasks, designers can focus their efforts to optimize them in order to reduce their size and cost; as a result, they can increase their reliability and efficiency.

4.1.2. Linux

Now we will step into Linux world. Linux is a Unix-based OS, free and open source. It made its appearance in 1991, when Linus Torvalds, computing student at the University of Helsinki, started to write first Linux code lines, as a hobby and without imagining what this project would become. Linux quickly evolved from an individual project to a worldwide development project involving thousands of developers.

Linux is composed of the system kernel with a great number of programs and libraries, making its use feasible. We can observe in Figure 1 the essential structure of the OS. We will not delve into its architecture; the detailed performance of the OS is beyond the scope of this project. We will just say that we have, on the one hand, a kernel space, with the system kernel and a system call interface; on the other hand, the user space, with user applications and libraries. Should you want to know more about Linux, you can visit an article on IBM's (International Business Machines) webpage "Anatomy of the Linux kernel" (go to bibliography).

Many times the Linux OS is called GNU[1]/Linux, because part of the interaction between user and hardware is handled usually with tools from the GNU project.

[1] GNU is a recursive acronym for "GNU is Not Unix!", because GNU's design is Unix-like, but differs from Unix by being free software containing no Unix code.

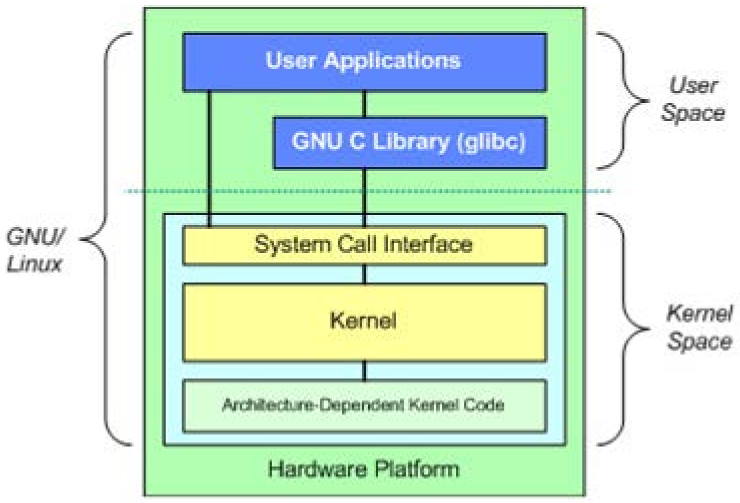


Fig. 1: GNU/Linux architecture

4.1.3. Embedded Linux

Linux has been adapted to a wide variety of CPUs (Central Processing Unit), not only used as processor of a desktop computer or a server, but also processors for embedded systems such as ARM[2]. Linux is used as an alternative way to proprietary systems; in the past, embedded systems development was mostly implemented with proprietary code written in assembly language. Developers had to write all drivers for hardware devices and interfaces starting from scratch. Linux kernel, combined with a set of some open software utilities, can be adjusted within the limited hardware space of embedded systems.

There are more embedded OS like QNX (Quantum Software Systems), LynxOS, Windows CE (Embedded Compact) or Windows NT (New Technology). Embedded Linux has some advantages over the rest of embedded OS: it has a smaller size, it is easily customizable, mature and stable (more than 20 years old and used in many devices).

Due to all its advantages, Linux has been inserted into a lot of consumer devices like PDAs (Personal Digital Assistant), GPS devices (Global Positioning System), residential gateways or smart phones. Today, the OS that is dominating the mobile phones market is Android, based on a modified Linux kernel along with a custom user space.

With the availability of consumer embedded devices, user and developer communities were founded around these devices. Replacements or improvements of Linux distribution have been often possible thanks to the availability of source code and the communities. Because of the large number of devices, standardized systems of automated Linux generation were created, as OpenEmbedded, OpenWrt or BuildRoot.

[2] ARM stands for Advanced RISC Machines, where RISC is the acronym for Reduced Instruction Set Computer

4.2. Linux components to generate. Boot flow

In order to build our own Linux distribution, we must generate a set of files:

- **Preloader:** It is responsible for performing some configurations before fetching the subsequent bootloader image. Some of its functions are: initializes the SDRAM (Synchronous Dynamic Random Access Memory) interface, including calibration and configuration of PLLs (Phase Locked Loop) to set up clocks, extracts the bootloader image from the required flash controller (in our case, SD/MMC (Secure Digital/MultiMedia Card)) and passes boot flow control over to the bootloader.

- **Bootloader:** A simple program designed to set up everything that the OS needs to work. On this project we will use U-Boot, an open source bootloader used for embedded systems. Its main functions are: configures the OS environment, extracts the OS image from the corresponding flash device (as previously mentioned, in our case is the SD/MMC), stores the kernel and the DTB[3] into memory, boots the kernel and passes the content of the DTB to it. It also provides a console that can be used for user operations, like modifying boot parameters.

- **DTB:** It is a binary file which represents the FDT (Flattened Device Tree), a data structure that describes the hardware of a system and "translates" it for the OS, so that the OS finds and registers the system devices. Some design aspects of the board it describes are: number and type of CPUs, size and base addresses of RAM (Random Access Memory) memory, bridges and buses, peripheral connections, interrupt managers, etc. A FDT image can be linked statically to the kernel or transferred during boot time.

- **Kernel.**

- **File system:** System that controls how information is stored (files and folders).

The steps to run an application on Linux are the following:

1. **Reset:** The boot process begins when a CPU exits a reset state, and starts running code in the memory address where the Boot ROM (Read Only Memory) is mapped.

2. **Boot ROM:** It contains code that, as we just said, runs after performing a reset. It reads the clocks and boot configuration from the configuration pins of the chip. It also reads the preloader from the corresponding flash device and loads it.

3. **Preloader:** Is copied into the RAM memory and executed. Its main purpose is to set up the appropriate clocks configuration that the Boot ROM previously read and set up the required pin multiplexing in order to route peripherals to the pins. When it ends, it will load the bootloader to start its execution.

[3] DTB is the acronym for Device Tree Blob, where BLOB is the acronym of Binary Large OBject

4. **Bootloader:** Just as we explained in the previous page, it loads the OS and passes control software (DTB) to it.

5. **Kernel:** At last, the kernel is responsible for the rest of the system configuration: interruptions, devices initialization, drivers and controllers, memory management, etc. After all this, the applications can be executed.

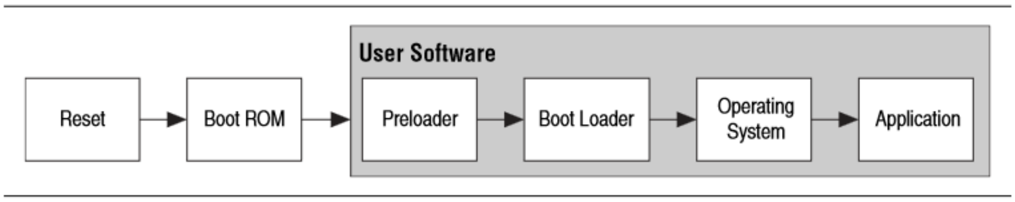


Fig. 2: Linux boot flow

4.3. SoC solutions

4.3.1 What is a SoC solution?

Before taking a look to our development kit, which has a SoC (System on Chip) solution, we will briefly explain what SoC consists in. Broadly speaking, a SoC solution is a microchip which contains all required components to power and provide energy to a complete system, such as a computer. A complete system needs several components to work: microprocessor, memory, peripheral interfaces, I/O logic control, data converters, etc. The components in a PC motherboard, for example, lay on separated chip. In the case of a SoC, all the components are in a single chip instead.

A computer needs several components, and it has enough space to contain them all, but this would be impossible to do in a device such as the SoCkit board. Thanks to SoC solutions, we have a good computing power in smaller space. In addition, these devices do not need too much power to work, thanks to the high scaled components integration, reducing wires. Therefore, efficiency is increased.

4.3.2. Altera's Cyclone V SoC

Next, we talk about our development kit. Altera develops a set of FPGAs (Field Programmable Gate Array), classified into several families according to their features. The SoC solution that contains the SoCkit board belongs to the Cyclone series, low-cost FPGAs. Particularly, it belongs to the Cyclone V family.

The Altera SoC solution integrates one FPGA Cyclone V fabric with a HPS (Hard Processor System) designed by ARM (Figure 3). This HPS is composed by a double core processor ARM Cortex-A9, a group of peripherals and a set of high performance buses that interfaces to the FPGA; all this can be viewed in Figure 4.

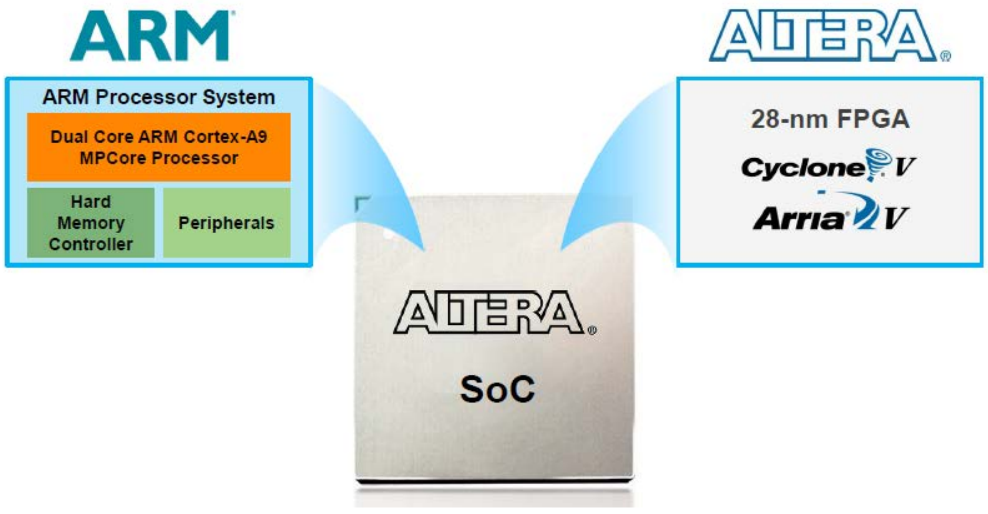


Fig. 3: Altera's SoC solution

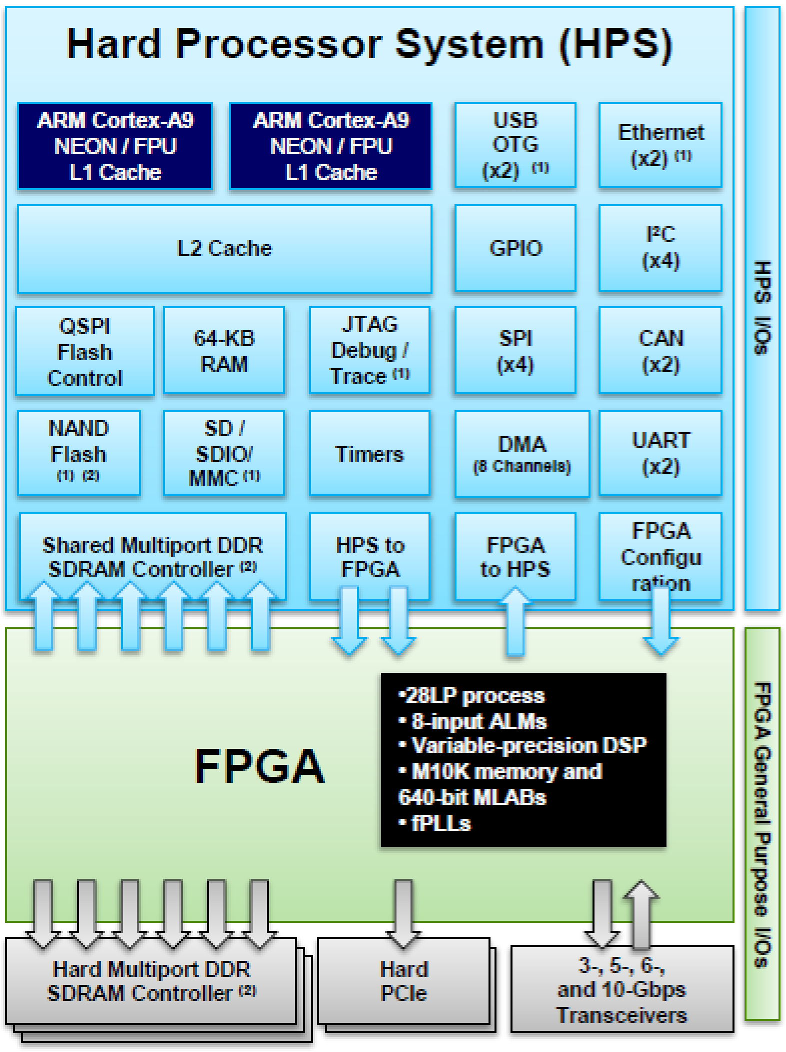


Fig. 4: Architecture of Altera's SoC system

So far, we have seen a general overview of the systems whose hardware is based on; this hardware will be integrated with the OS that we attempt to generate. This is the main objective of this project; next, we will see the objectives we aim to in more detail.

5. Objectives

We attempt to integrate and start up the set of needed tools to deploy the design cycle of Embedded Linux based systems on Altera's Cyclone V SoC.

Firstly, we analyze the available tools for designing the hardware system of the SoCkit development kit which includes a "Cyclone V SoC" system (based on a "ARM Cortex-A9 MPCore" architecture). When designing the SoCkit board hardware, we will intent to check if it is possible not only to use the pre-configured resources of the board, but also add some *ad hoc* peripherals to the system configuration, new and totally customizable. We will check the correct operation of this new component and connect it to the hardware system, so it can be used as any other existent resource of the SoCkit board.

Next, we will analyze the open source tools to generate an Embedded Linux distribution adapted to the SoCkit board. In order to generate the Embedded Linux distribution configured according to our board, we will use a software package from Yocto, recommended by Altera. This tool will be used together with Altera's software, to generate the necessary components of the Linux OS (the ones we specified in the previous section "4.2. Linux components to generate. Boot flow") that will implement the custom configuration we want for our SoCkit board. After creating and compiling all needed files, we will combine them creating a complete and functional system to be used for the development of applications.

Finally, we will also analyze the programs for developing and debugging applications in C or C++ language that will be executed in this hardware platform. Then, we will make a Linux application program as an example to illustrate the use of SoCkit board resources (including our new peripheral).

The main purpose is, summarizing, to document the whole process of generating and creating an own customizable Embedded Linux distribution for this hardware platform, without depending on "pre-cooked" content provided by the manufacturer and the developer communities, and to set up everything to open the way for software applications development over this board.

5. Objectives

6. General description of software and hardware tools

6.1. SoCkit development kit

The north-american company Arrow provides an easy way to learn and develop SoC solutions: the SoCkit board. It was developed with the purpose of helping engineers to quickly evaluate the operation and flexibility of the Cyclone V SoC system for their designs, to be convinced of how simple is to work with these devices. Figure 5 and Figure 6 show its resources; some of them will be briefly described since they are important for this project. We will describe what we use them for; but actually, they can be used in other ways.

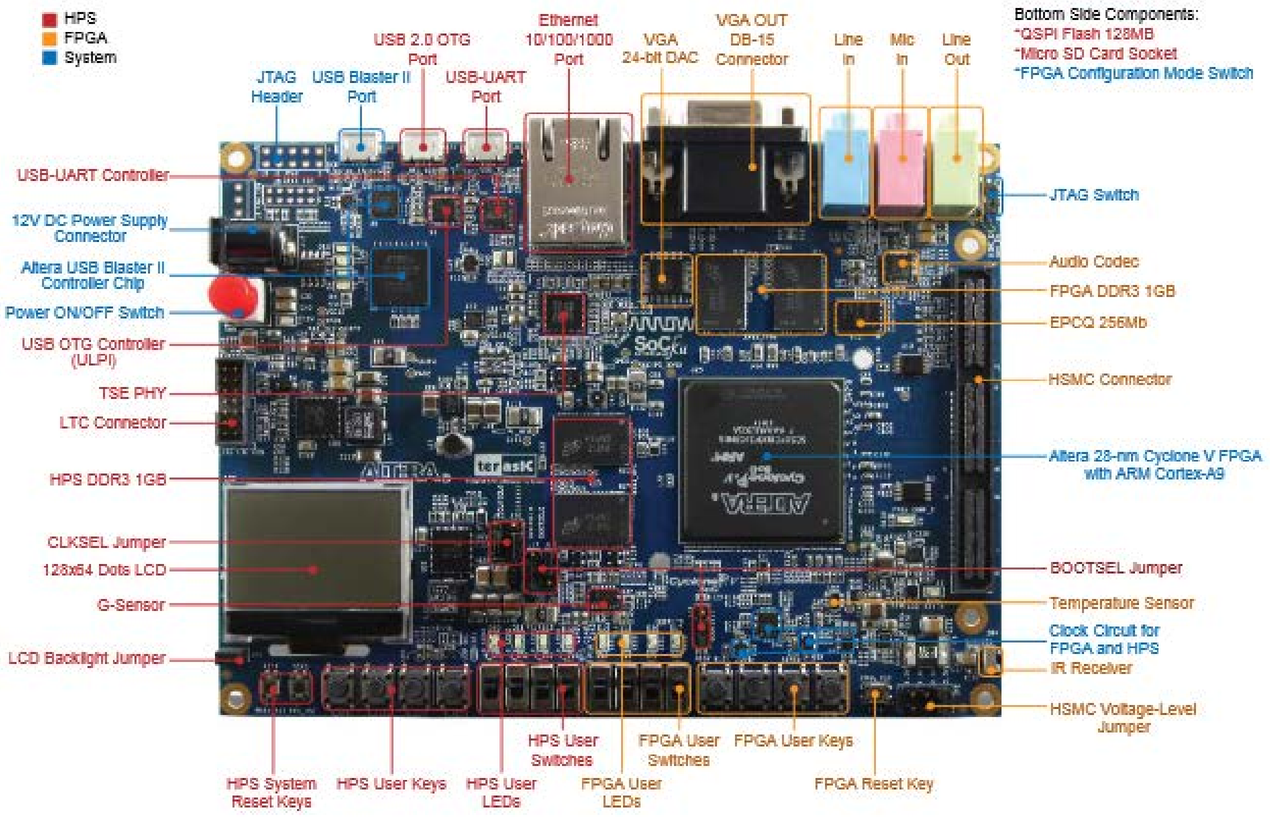


Fig. 5: SoCkit development board

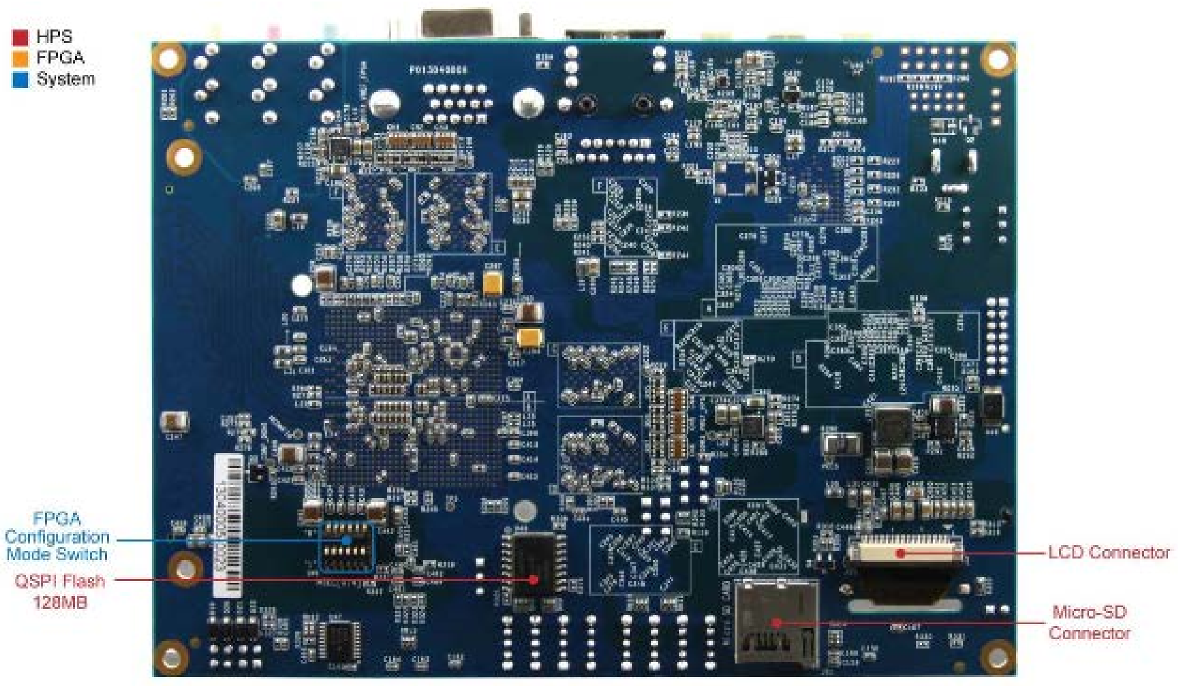


Fig. 6: Back of the SoCkit development board

- **Cyclone V SoC:** A system that combines a HPS, designed by the British company ARM, and an FPGA by Altera, in a single chip.

- **Micro-SD connector:** This connector will contain the Micro-SD card where we must write the image file containing all the elements of the Linux distribution. The Linux OS boots from this device.

- **USB-UART[4] connection:** Used for serial communication between our board and the computer where all software tools and programs are installed.

- **Ethernet connection:** Used for debugging C/C++ applications that will be executed on the board.

- **Warm reset:** Used to reboot the Linux system.

- **Buttons, switches and LEDs (Light-Emitting Diode):** Some of them used to test that the built configuration works properly.

- **Jumpers CLKSEL & BOOTSEL:** To select clock rates and system boot source, respectively. They must be configured in a specific way, as we will see further.

- **FPGA configuration switches (MSEL):** They select the FPGA configuration mode.

[4] USB-UART is the union of two acronyms, Universial Serial Bus and Universal Asynchronous Receiver-Transmitter

6.2. Altera software

Altera provides a set of tools to design and create our system:

- **Quartus II a nd Qsys:** These tools are very useful because they allow the configuration of the HPS and the FPGA of the SoCkit board. This hardware configuration includes the selection of peripherals will be used within our design. They generate some files to give design information to other tools that take control of the application debugging and create some blocks of the Linux distribution.

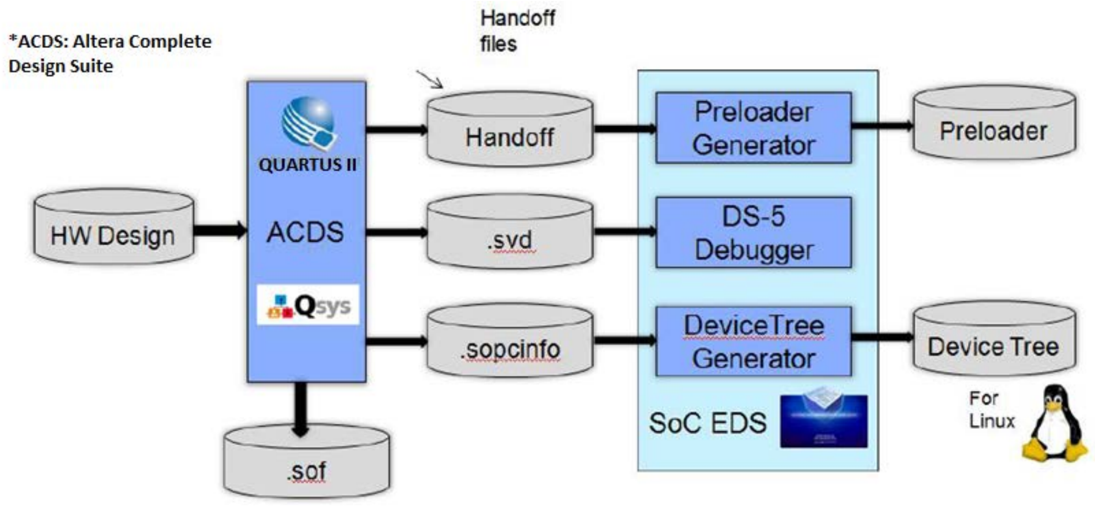


Fig. 7: Design flow of the Altera's tools

- **SoC EDS (Environment Development System):** A set of tools that help us to generate a preloader, a bootloader and a DTB for Linux, from the configuration files created in Quartus II and Qsys. They also help us to debug applications.

- **Modelsim and System Console:** Tools to validate the correct operation of the FPGA peripherals previously configured in Qsys.

6.3. Yocto Source Package

The YSP (Yocto Source Package) is an installer recommended by Altera that contains the Yocto's generating system and all the dependencies to compile the essential elements needed to build a full Linux distribution.

Yocto is a community project that aims to generation of Linux distributions for several architectures. Yocto Project counts on Open Embedded, a suite of tools (we will use some of them) that take charge of generating source code. Yocto is becoming the default standard to generate Linux source code.

6.4. More tools

6.4.1. Desktop computer

On this computer we will install and execute all software tools to configure our board resources, generate required configuration files, create image files to build the Linux distribution and write them to the SD card. Furthermore, we will perform the serial communication with our board through a PuTTY terminal in order to boot the OS and interact with it. The computer used has a 4 GB RAM memory, an i5 processor and a 64-bit Windows 7 Professional OS.

6.4.2. Virtual machine

We need a computer with a Linux-based OS in order to use some of the tools to generate the Linux distribution. The computer we will use has a Windows OS, as we just mentioned. Hence, we will use a virtual machine with a Linux-based OS to perform all the tasks.

We will use Ubuntu Linux, based on Debian distribution. It is the most popular Linux distribution. We have to be careful; some Ubuntu versions are not compatible with the Yocto software. This project was performed using an Ubuntu 12.04 (LTS) version. Currently, the Yocto Project is supported on the following Ubuntu distributions: Ubuntu 10.04, Ubuntu 11.10, Ubuntu 12.04 (LTS), Ubuntu 12.10.

In order to view a detailed list of supported distributions, not only Ubuntu Linux, please refer to "Yocto Project Reference Manual", chapter "1.3.1. Supported Linux Distributions" in the bibliography (consulted in January 2014, last revision released in April 2013).

6.4.3. SoCkit Lab Materials / GHRD

GHRD (Golden Hardware Reference Design) is a pre-built hardware reference design for our board. In a subdirectory of the folder where all the Altera's software is installed there is the GHRD for the "Altera SoC Development Board", another development kit for Cyclone V, created by Altera (we will mention this board a couple of times along this project). To obtain this design for the SoCkit board, we have to download it from the community webpage projects Rocketboards.org. We will not get deep into the components of this design, we will do it later as we move forward in this project.

Moreover, Arrow provides on their website a link with all the required files to perform all the hardware design process, including useful manuals and source files. Those are the "SoCkit Lab Materials".

6.4.4. PuTTY

PuTTY is a free and open source terminal emulator, that supports various network protocols. It will be used to connect the computer with our board, so that we can monitor the Linux system boot flow and interact with it.

7. Hardware development

We will follow some steps grouped into two categories: hardware and software development (separated by a grey vertical line in Figure 8). Figure 8 shows all these steps in a diagram, we will show it along the project as a guide that helps the reader not to get lost and to know which part of the project we are in at every moment.

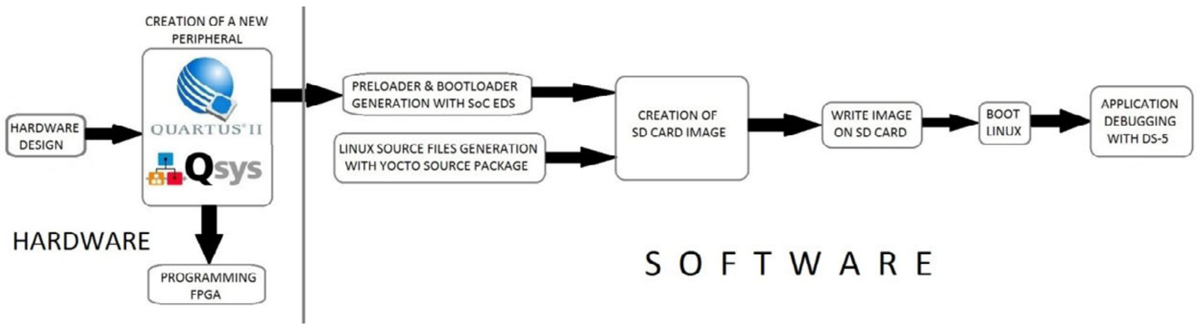

CREATION OF A NEW PERIPHERAL
HARDWARE DESIGN
QUARTUS II
Qsys
HARDWARE
PROGRAMMING FPGA
PRELOADER & BOOTLOADER GENERATION WITH SoC EDS
LINUX SOURCE FILES GENERATION WITH YOCTO SOURCE PACKAGE
CREATION OF SD CARD IMAGE
WRITE IMAGE ON SD CARD
BOOT LINUX
APPLICATION DEBUGGING WITH DS-5
SOFTWARE

Fig. 8: Steps to follow in this project

In this section we will perform all the steps that lead to a complete and functional hardware configuration using Quartus II and Qsys. We will also create a new custom peripheral, add it to the design and check whether it works correctly or not.

7.1. Installing the virtual machine

First of all, we must install the software of virtualization with all the necessary tools and programs. We are using software provided by VMware Inc.; particularly, the free tool VMware Player[d1]. **Download the latest version of VMware Player and install it with the desired options (the installation progress is very easy and intuitive).**

We have to download an image file (.iso format) containing a Linux-based OS for the virtual machine. As mentioned before, we will use Ubuntu Linux[d2] (12.04 LTS version), the most popular distribution and easy to use. To install Ubuntu Linux in the virtual machine, **go to VMware Player menu, select the "Create a New Virtual Machine" option. Select the option "Installer disc image file (iso)" and browse the Ubuntu image file (the OS will be detected automatically), choose a name, a p assword and a location for the virtual machine and select the space on hard disk for the virtual machine.**

The virtual machine will need a lot of memory space, since Yocto and Altera tools take up a large amount of it. We will assign 100 GB of memory space to our virtual machine; this is more than the space we need, but doing so we make sure that we have enough space and we will not have to worry about it later.

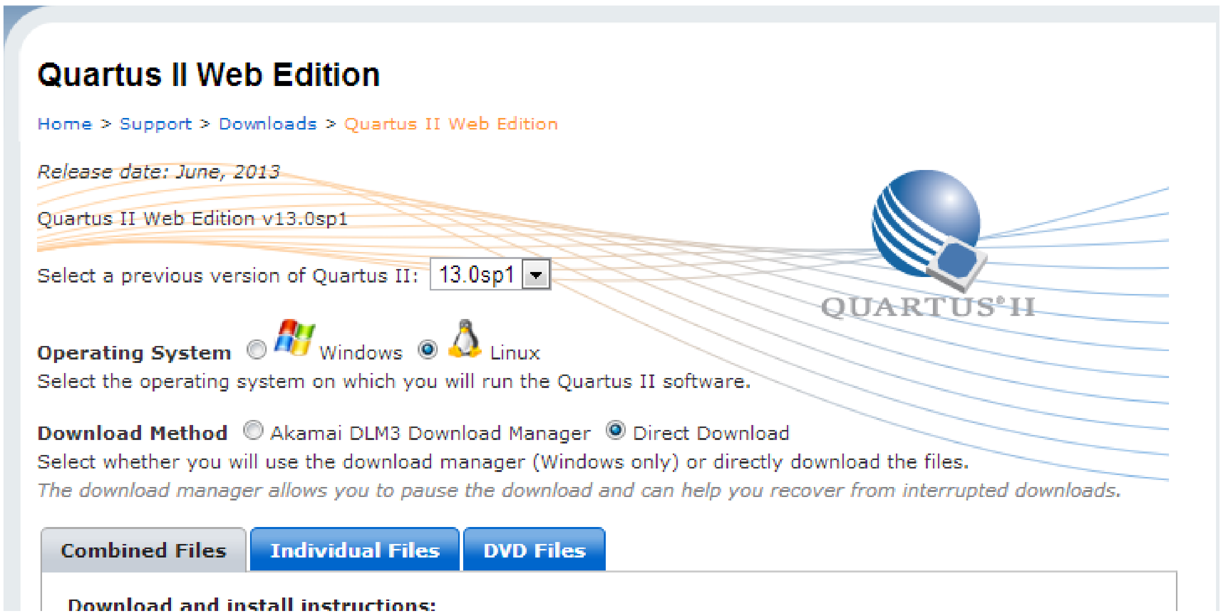
Before finish, click on "Customize Hardware..." to view or modify the hardware features of the virtual machine (we only modified the RAM memory in order to have a more fluent operation). When everything is ready, click on "Finish" and the virtual machine will start installing (this will take a while, be patient). If a command prompt is shown instead of a desktop, type the command "startx".

7.2. Download and install Quartus II

Among all Altera tools, **we must install Quartus II. Use the Web Edition**, a free version valid for our purpose because it includes support for low and medium cost FPGAs, including the Cyclone V. **On the download webpage[d3], select the following options (see Figure 9):**

- Software version: 13.0sp1
- OS: Linux (also available for Windows).
- Download method: direct download (a download manager is available for Windows hosts; it is not an option for Linux hosts).

It is necessary to be registered on Altera's website to download software; it is free.



Quartus II Web Edition

Home > Support > Downloads > Quartus II Web Edition

Release date: June, 2013

Quartus II Web Edition v13.0sp1

Select a previous version of Quartus II: 13.0sp1

Operating System Windows [✓ Linux]

Select the operating system on which you will run the Quartus II software.

Download Method Akamai DLM3 Download Manager [✓ Direct Download]

Select whether you will use the download manager (Windows only) or directly download the files.

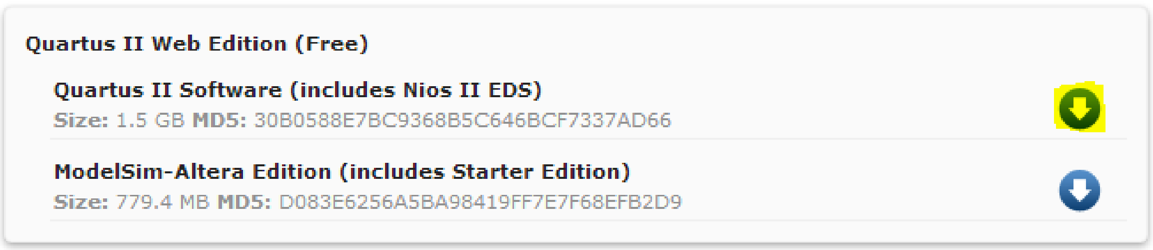
The download manager allows you to pause the download and can help you recover from interrupted downloads.

Combined Files | Individual Files | DVD Files

Download and install instructions:

Fig. 9: Quartus II download page

On the tab "Individual Files", scrolling down the page we can find all the different software packages. Select "Quartus II Software (includes Nios II EDS)", as seen in Figure 10.



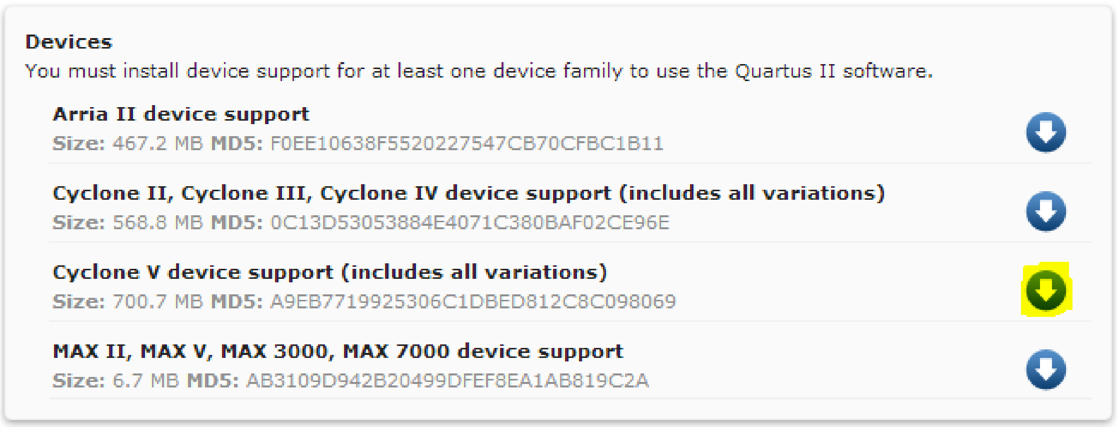
Quartus II Web Edition (Free)

Quartus II Software (includes Nios II EDS)
Size: 1.5 GB MD5: 30B0588E7BC9368B5C646BCF7337AD66

ModelSim-Altera Edition (includes Starter Edition)
Size: 779.4 MB MD5: D083E6256A5BA98419FF7E7F68EFB2D9

Fig. 10: Quartus II download options

Additionally, we have to download the device support file for the Cyclone V family, the one our FPGA belongs to. Doing so, when installing Quartus II we must see an option to include Cyclone V devices support (see Figure 11).



Devices
You must install device support for at least one device family to use the Quartus II software.

Arria II device support
Size: 467.2 MB MD5: F0EE10638F5520227547CB70CFBC1B11

Cyclone II, Cyclone III, Cyclone IV device support (includes all variations)
Size: 568.8 MB MD5: 0C13D53053884E4071C380BAF02CE96E

Cyclone V device support (includes all variations)
Size: 700.7 MB MD5: A9EB7719925306C1DBED812C8C098069

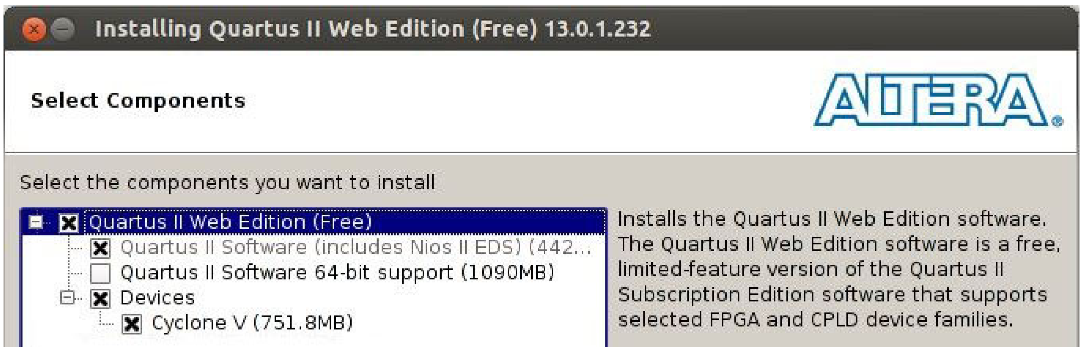
MAX II, MAX V, MAX 3000, MAX 7000 device support
Size: 6.7 MB MD5: AB3109D942B20499DFEF8EA1AB819C2A

Fig. 11: Downloading support for Cyclone V

Before installing it, **locate the downloaded Quartus II file. Right click on it and access to Properties. On the Permissions section, select the box "Allow executing file as a program".** If we do not select it, Ubuntu OS will try to open this file instead of executing it, so it will not find any application to open it. After that, to begin installation we have to execute the file with root user privileges (*sudo* command); otherwise, we will see a message indicating that there were troubles with the installation (for example, the uninstaller file will not be created correctly). Thus, despite these errors may not have a negative impact on the tools operation, we will execute it as root to make sure that everything is installed correctly so we will not have unexpected issues afterwards. **Open a terminal in Ubuntu, navigate to the directory where the file is (it will be stored on "/home/<user>/Downloads" by default) and execute the following command:**

```
$ sudo ./QuartusSetupWeb-13.0.1.232.run
```

When typing this command, the prompt asks for the user password. **Type the password you set in the installation. Install Quartus II on the default directory (*home/<user>/altera/13.0sp1*). When selecting the components we want to install, unselect the box "Quartus II Software 64-bit support"** (Figure 12), because in this project we are using a 32-bit Ubuntu OS on the virtual machine so we do not need the 64-bit support. After that, the program starts installation.



Installing Quartus II Web Edition (Free) 13.0.1.232

Select Components

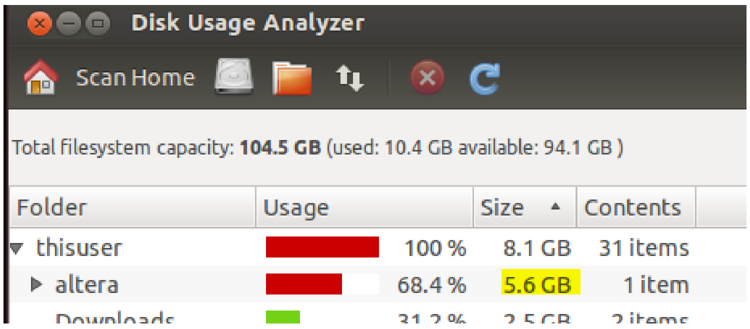
Select the components you want to install

- [x] Quartus II Web Edition (Free)
  - [x] Quartus II Software (includes Nios II EDS) (442...
  - [ ] Quartus II Software 64-bit support (1090MB)
  - [x] Devices
    - [x] Cyclone V (751.8MB)

Installs the Quartus II Web Edition software. The Quartus II Web Edition software is a free, limited-feature version of the Quartus II Subscription Edition software that supports selected FPGA and CPLD device families.

Fig. 12: Installing Quartus II

The installation takes up 5.6 GB so far (see Figure 13).



Total filesystem capacity: **104.5 GB** (used: 10.4 GB available: 94.1 GB)

Folder	Usage		Size	Contents
thisuser	🟥	100 %	8.1 GB	31 items
altera	🟥	68.4 %	5.6 GB	1 item
Downloads	🟩	31.2 %	2.5 GB	2 items

Fig. 13: Memory space usage

We can see some folders in the installation directory. All the Quartus II files are in the "quartus" folder (See Figure 14).

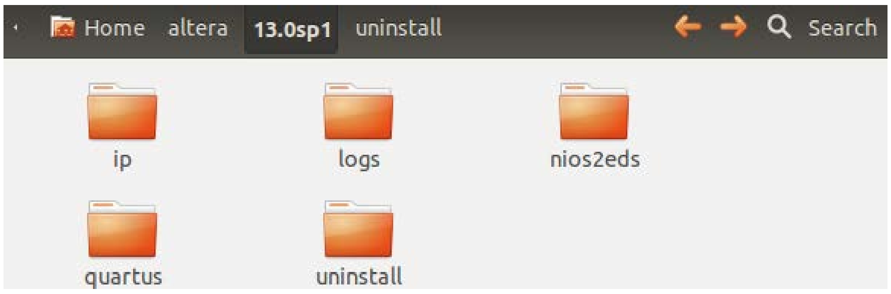


Fig. 14: Installation directory

This step is the same if we use Windows, even easier since there are no root privileges issues. To execute the installation file, just double-click on it. The rest of the process is the same.

7.3. Setting up the SoCkit

We must set up the SoCkit board in order to use it. Some jumpers need to be configured on a specific way before proceeding.

The processor in the HPS can be boot from many sources such as the SD card or the FPGA. The boot source can be set using the BOOTSEL jumpers (shown in Figure 15 as J17, J18 and J19) and CLKSEL jumpers (J15 and J16 in Figure 15). Table 1 lists BOOTSEL and CLKSEL settings. Table 2 lists the settings for selecting a suitable boot source.

Table 1: BOOTSEL and CLKSEL settings

Board Reference	Signal Name	Setting
J17	BOOTSEL0	Short Pin 1 and 2: Logic 1 Short Pin 2 and 3: Logic 0
J19	BOOTSEL1	Short Pin 1 and 2: Logic 1 Short Pin 2 and 3: Logic 0
J18	BOOTSEL2	Short Pin 1 and 2: Logic 1 Short Pin 2 and 3: Logic 0
J15	CLKSEL0	Short Pin 1 and 2: Logic 1 Short Pin 2 and 3: Logic 0
J16	CLKSEL1	Short Pin 1 and 2: Logic 1 Short Pin 2 and 3: Logic 0

Table 2: BOOTSEL[2:0] Setting Values and Flash Device Selection

BOOTSEL[2:0] Setting Value	Flash Device
000	Reserved
001	FPGA (HPS-to-FPGA bridge)
010	1.8V NAMD Flash memory*
011	3.0 V NAMD Flash memory*
100	1.8 V SD/MMC Flash memory*
101	3.0 V SD/MMC Flash memory
110	1.8 V SPI or quad SPI Flash memory*
111	3.0 V SPI or quad SPI Flash memory

*Not supported on SoCkit board.

Make sure the jumpers are correctly configured as seen on Figure 15. We choose the 3.0V SD/MMC Flash memory as the boot source (BOOTSEL[2:0] = 101). The CLKSEL jumpers should be configured as "00" for the slowest HPS peripheral clock speed option.

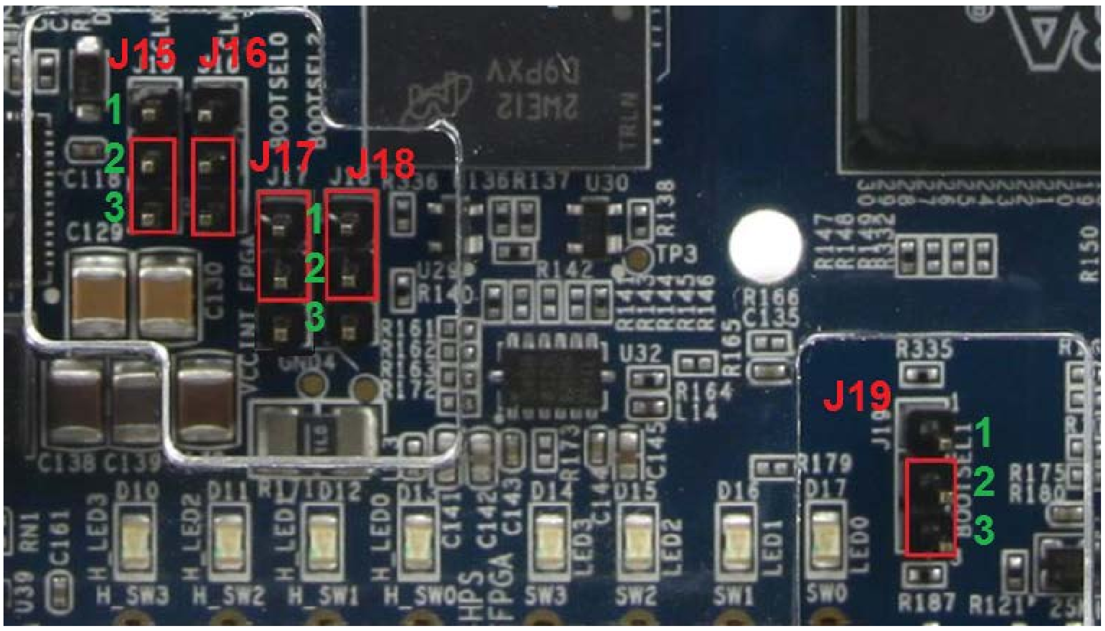


Fig. 15: Jumpers configuration on SoCkit

On the bottom side of the board there is the Dipswitch SW6 that need to be configured too (see Figure 16). It sets the MSEL pins to decide the FPGA configuration modes. Table 3 gives the MSEL pins setting for each configuration scheme of Cyclone V devices.

Table 3: MSEL pin Settings for each Scheme of Cyclone V Device

Configuration Scheme	Compression feature	Design security feature	POR Delay	Valid MSEL[4:0]
FPPx8	Disabled	Disabled	Fast	10100
			Standard	11000
	Disabled	Enabled	Fast	10101
			Standard	11001
	Enabled	Disabled	Fast	10110
			Standard	11010
FPPx16	Disabled	Enabled	Fast	00000
			Standard	00100
	Disabled	Disabled	Fast	00001
			Standard	00101
	Enabled	Enabled	Fast	00010
			Standard	00110
PS	Enabled/ Disabled	Disabled	Fast	10000
			Standard	10001
AS(X1 and X4)	Enabled/ Disabled	Enabled	Fast	10010
			Standard	10011

The FPGA default works in ASX4 mode. However, in this mode the SoCkit will be unable to boot Linux from the SD card or other devices. **Switch MSEL** to another mode **as shown on**

Figure 16 (MSEL[4:0] = 00001; the sixth switch has no function) to enable normal operations of Linux.

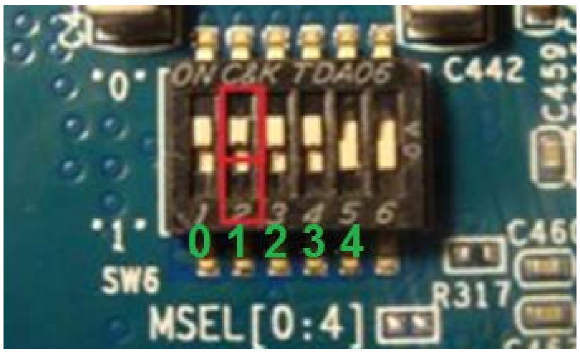


Fig. 16: Micro-switches from SoCkit back side

We need to verify the JTAG CHAIN micro-switch (SW4), also placed on the bottom side of the SoCkit. The SoCkit allow users to access the FPGA, HPS debug, or other JTAG chain devices via the on-board USB Blaster II. Users can control whether the HPS or the HSMC connector is included in the JTAG chain via SW4. **Configure the switch as shown in Figure 17** (it is placed on the right of the audio connectors) to include the HPS connector in the JTAG chain.

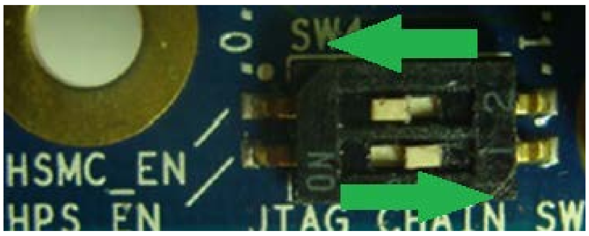


Fig. 17: JTAG CHAIN micro-switch

It is necessary to plug the SoCkit power cable, the USB-UART connection (for serial communication with the computer), **Ethernet connection** (for application debugging) **and USB Blaster II connection**. Observe in Figure 18 the location of all these connectors.

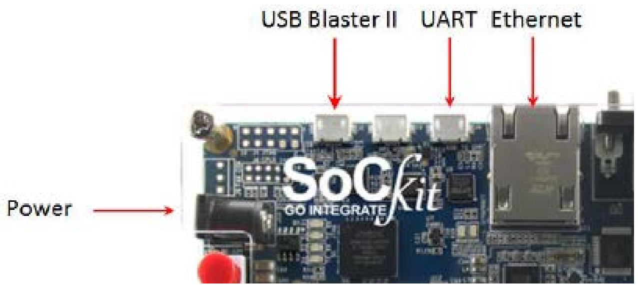


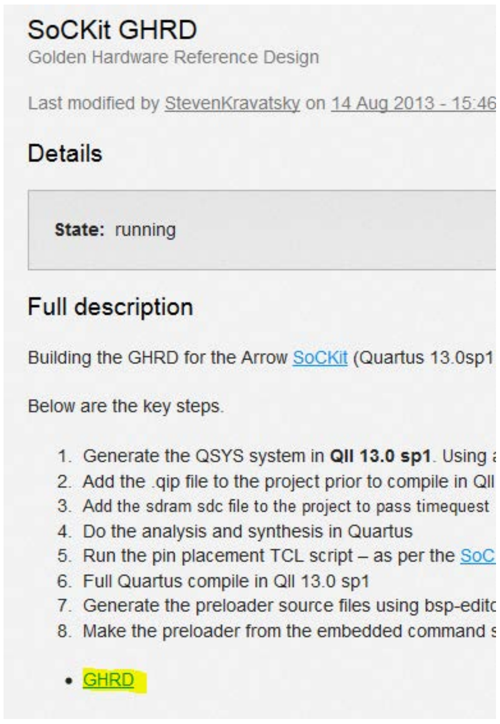
Fig. 18: SoCkit connectors

On Appendix A there is an additional section required to follow in case of using the Altera software in a Windows OS.

7.4. Downloading GHRD project

Now we are going to download from the Rocketboards webpage a file named "GHRD_soc_system.qar", which contains the archived GHRD project that we can open with Quartus II.

Access to the Rocketboards webpage to download the GHRD file (http://rocketboards.org/foswiki/view/Projects/SoCKITGHRD) (see Figure 19).


SoCKit GHRD
Golden Hardware Reference Design

Last modified by StevenKravatsky on 14 Aug 2013 - 15:46

Details

State: running

Full description

Building the GHRD for the Arrow SoCKit (Quartus 13.0sp1

Below are the key steps.

1. Generate the QSYS system in **QII 13.0 sp1**. Using
2. Add the .qip file to the project prior to compile in QII
3. Add the sdram sdc file to the project to pass timequest
4. Do the analysis and synthesis in Quartus
5. Run the pin placement TCL script – as per the SoC
6. Full Quartus compile in QII 13.0 sp1
7. Generate the preloader source files using bsp-editc
8. Make the preloader from the embedded command s

- GHRD

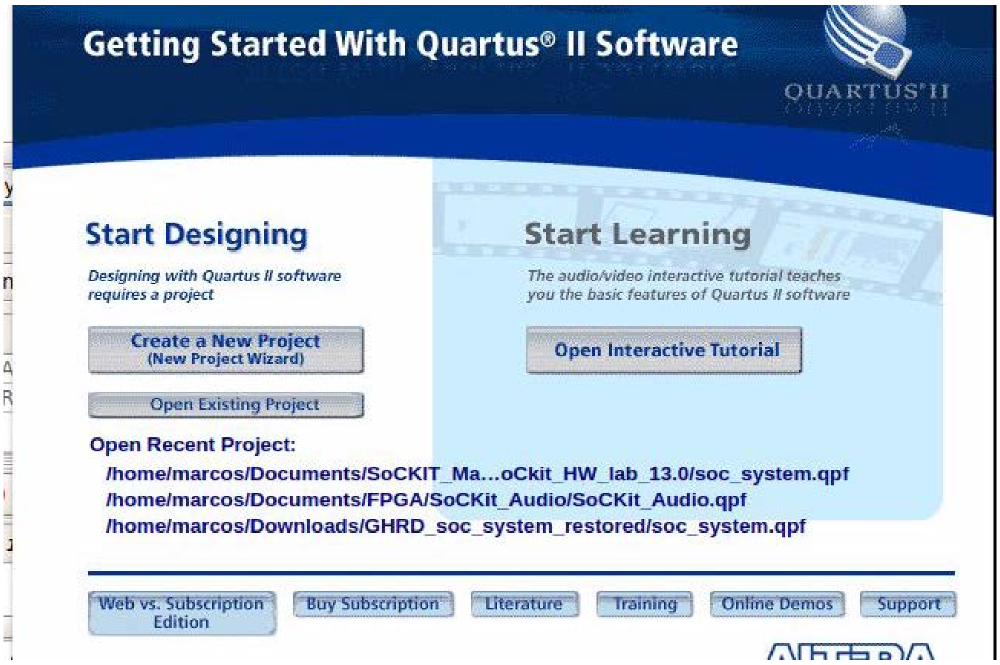
Fig. 19: Downloading GHRD

It is possible to encounter some troubles to download this file. **If you click on the link shown in Figure 19 and you just see a webpage full of strange symbols, try to right click on the GHRD link and select "Save Link As..."; you should be able to download the 'GHRD_soc_system.qar' file successfully**. **Once the file is downloaded, open Quartus II**. If Quartus II was installed with all options by default, a shortcut to the program should have been created in the Ubuntu desktop, then just double click on it. If not, open a terminal in Ubuntu and navigate to the "/home/<user>/altera/13.0sp1/quartus/bin" directory, then execute the following command:

```
$ sudo ./quartus
```

This command will execute the script that loads the program. It is better to use always this option, because we do it with root user privileges, so we will not have permission troubles or unexpected behavior.

When the program is open, the menu in Figure 20 will be shown. **Click on "Open existing project" and find the directory where the "GHRD_soc_system.qar" file was downloaded** (in our case, "/home/<user>/Downloads"). **Select this file to import the project from it**. We will see a window to choose the directory where all files will be extracted (we will extract it to "/home/<user>/Documents/GHRD_soc_system").


Getting Started With Quartus® II Software
QUARTUS® II
Start Designing
Designing with Quartus II software requires a project
Create a New Project (New Project Wizard)
Open Existing Project
Start Learning
The audio/video interactive tutorial teaches you the basic features of Quartus II software
Open Interactive Tutorial
Web vs. Subscription Edition
Buy Subscription
Literature
Training
Online Demos
Support

Open Recent Project:
/home/marcos/Documents/SoCKIT_Ma...oCkit_HW_lab_13.0/soc_system.qpf
/home/marcos/Documents/FPGA/SoCKit_Audio/SoCKit_Audio.qpf
/home/marcos/Downloads/GHRD_soc_system_restored/soc_system.qpf

Fig. 20: Initial Quartus II window

We have downloaded the hardware design for our system. Once the hardware design is completely and correctly configured, we will step into the software development (generating Linux preloader, bootloader, etc); but so far, we are working with the hardware design of the system.

This project contains a Verilog code file ("c5sx_soc") and a Qsys project ("soc_system.qsys") that contains the HPS hardware configuration. The "c5sx_soc" file has all of the I/O for the HPS instance and all I/O of the FPGA. If you want to visualize its content, just double click on it.

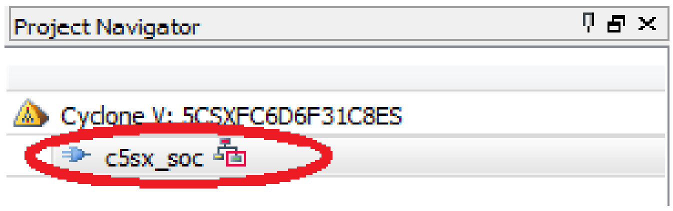
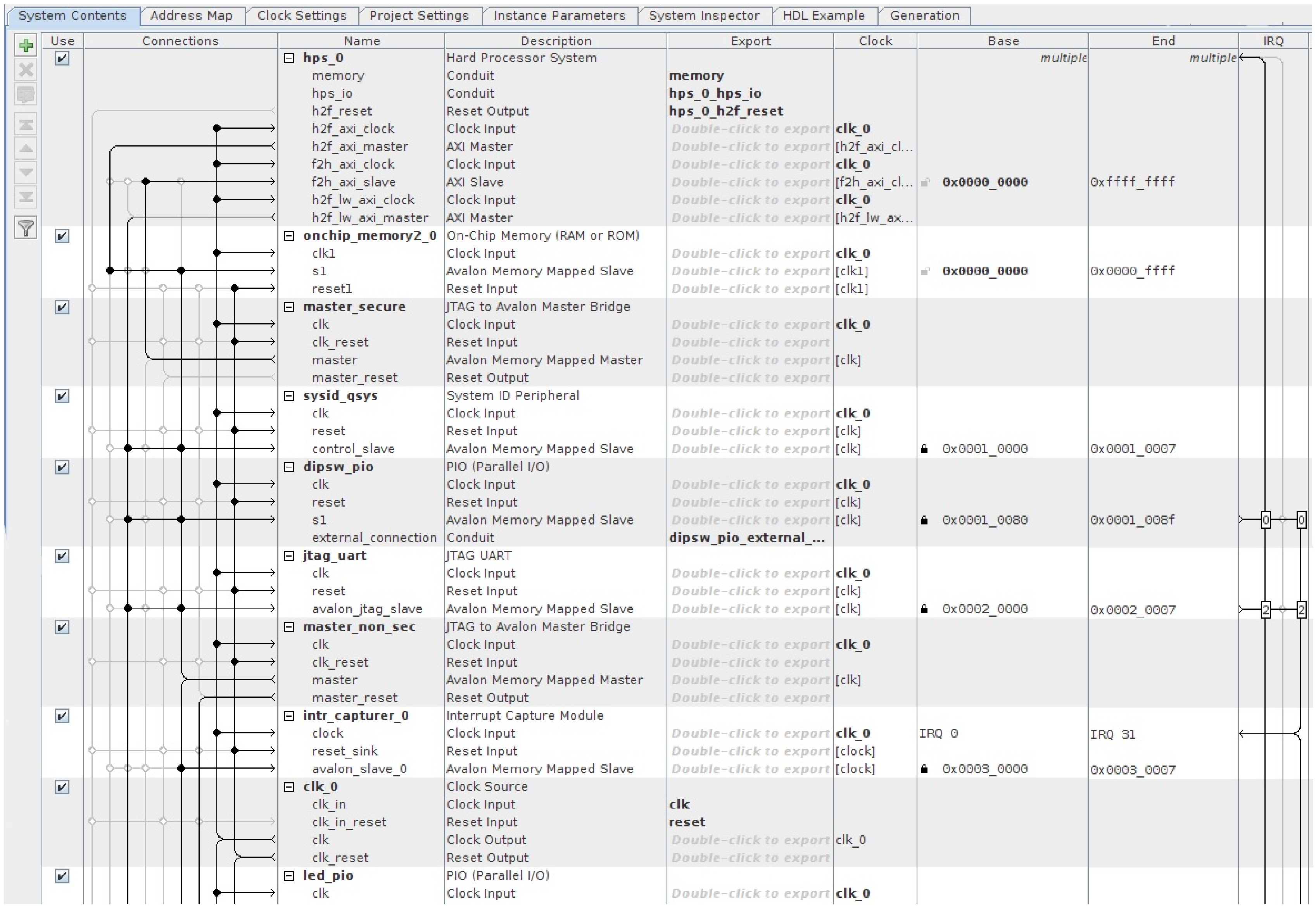


Fig. 21: c5sx_soc file

To launch Qsys, access to the Tools menu in Quartus II and select Qsys. While program is launching, it will ask for a Qsys project file, located in the project directory and named "soc_system.qsys". After that (it should take less than one minute), Qsys will display all the components of the Qsys design (see Figure 22 in next page).



System Contents | Address Map | Clock Settings | Project Settings | Instance Parameters | System Inspector | HDL Example | Generation

Use	Name	Description	Export	Clock	Base	End	IRQ
✓	⊟ hps_0	Hard Processor System			multiple	multiple	
	memory	Conduit	**memory**				
	hps_io	Conduit	**hps_0_hps_io**				
	h2f_reset	Reset Output	**hps_0_h2f_reset**				
	h2f_axi_clock	Clock Input	Double-click to export	**clk_0**			
	h2f_axi_master	AXI Master	Double-click to export	[h2f_axi_cl...			
	f2h_axi_clock	Clock Input	Double-click to export	**clk_0**			
	f2h_axi_slave	AXI Slave	Double-click to export	[f2h_axi_cl...	**0x0000_0000**	0xffff_ffff	
	h2f_lw_axi_clock	Clock Input	Double-click to export	**clk_0**			
	h2f_lw_axi_master	AXI Master	Double-click to export	[h2f_lw_ax...			
✓	⊟ onchip_memory2_0	On-Chip Memory (RAM or ROM)					
	clk1	Clock Input	Double-click to export	**clk_0**			
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk1]	**0x0000_0000**	0x0000_ffff	
	reset1	Reset Input	Double-click to export	[clk1]			
✓	⊟ master_secure	JTAG to Avalon Master Bridge					
	clk	Clock Input	Double-click to export	**clk_0**			
	clk_reset	Reset Input	Double-click to export				
	master	Avalon Memory Mapped Master	Double-click to export	[clk]			
	master_reset	Reset Output	Double-click to export				
✓	⊟ sysid_qsys	System ID Peripheral					
	clk	Clock Input	Double-click to export	**clk_0**			
	reset	Reset Input	Double-click to export	[clk]			
	control_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_0000	0x0001_0007	
✓	⊟ dipsw_pio	PIO (Parallel I/O)					
	clk	Clock Input	Double-click to export	**clk_0**			
	reset	Reset Input	Double-click to export	[clk]			
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_0080	0x0001_008f	0
	external_connection	Conduit	**dipsw_pio_external_...**				
✓	⊟ jtag_uart	JTAG UART					
	clk	Clock Input	Double-click to export	**clk_0**			
	reset	Reset Input	Double-click to export	[clk]			
	avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0002_0000	0x0002_0007	2
✓	⊟ master_non_sec	JTAG to Avalon Master Bridge					
	clk	Clock Input	Double-click to export	**clk_0**			
	clk_reset	Reset Input	Double-click to export				
	master	Avalon Memory Mapped Master	Double-click to export	[clk]			
	master_reset	Reset Output	Double-click to export				
✓	⊟ intr_capturer_0	Interrupt Capture Module					
	clock	Clock Input	Double-click to export	**clk_0**	IRQ 0	IRQ 31	
	reset_sink	Reset Input	Double-click to export	[clock]			
	avalon_slave_0	Avalon Memory Mapped Slave	Double-click to export	[clock]	0x0003_0000	0x0003_0007	
✓	⊟ clk_0	Clock Source					
	clk_in	Clock Input	**clk**				
	clk_in_reset	Reset Input	**reset**				
	clk	Clock Output	Double-click to export	clk_0			
	clk_reset	Reset Output	Double-click to export				
✓	⊟ led_pio	PIO (Parallel I/O)					
	clk	Clock Input	Double-click to export	**clk_0**			

Fig. 22: Qsys design

Before creating and configuring a new peripheral, we will end this section with a summary of all the components configured on Qsys design. We are in the first step of the diagram we mentioned before (Figure 23).

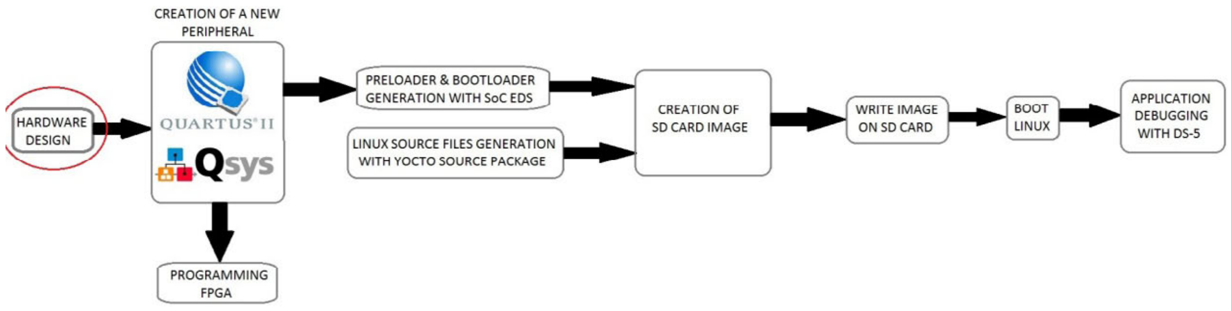

CREATION OF A NEW PERIPHERAL
HARDWARE DESIGN
QUARTUS II
Qsys
PRELOADER & BOOTLOADER GENERATION WITH SoC EDS
LINUX SOURCE FILES GENERATION WITH YOCTO SOURCE PACKAGE
CREATION OF SD CARD IMAGE
WRITE IMAGE ON SD CARD
BOOT LINUX
APPLICATION DEBUGGING WITH DS-5
PROGRAMMING FPGA

Fig. 23: First step. Setting up hardware design

In Figure 24 we can see an overview of the GHRD.

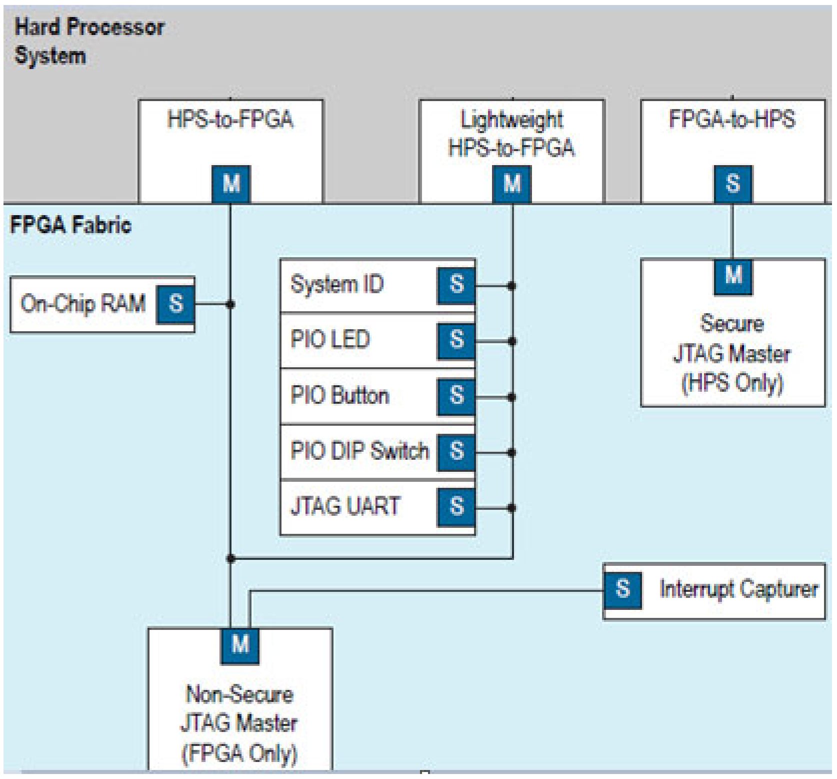
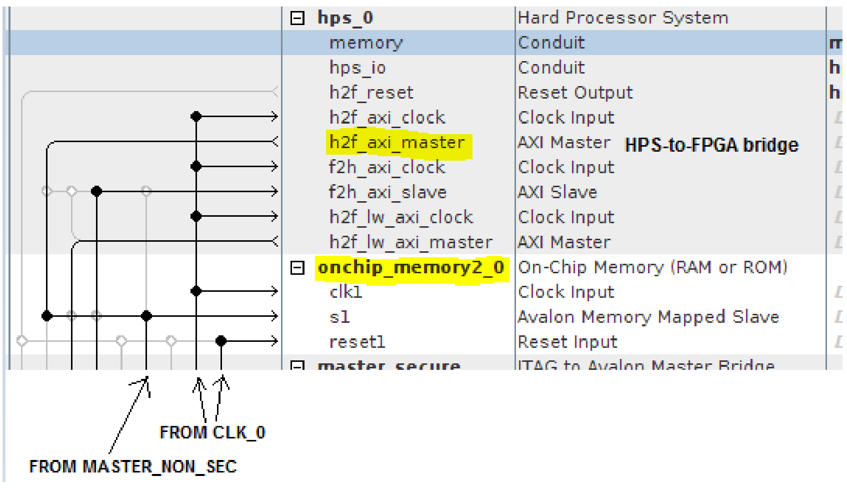


Fig. 24: GHRD overview

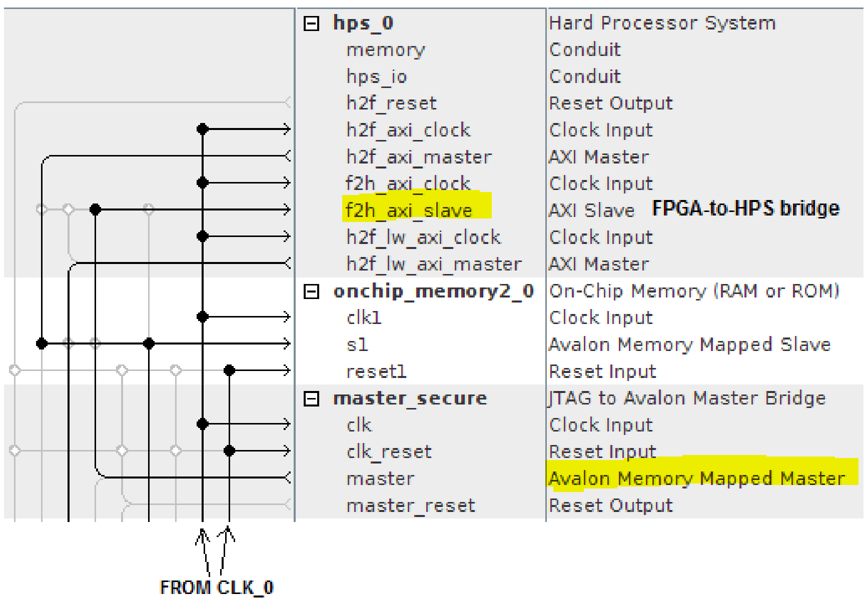
- **On-Chip RAM ("onchip_memory2_0" in Qsys):** This component provides the HPS Cortex-A9 MPU (Multiple Process Unit) access to memory, high speed and low latency. It is connected as a memory mapped slave to the Non-Secure JTAG Master and the HPS-to-FPGA bridge, as you can see in Figure 24 above and Figure 25 below (next page). It has a size of 64K bytes.



⊟ **hps_0**	Hard Processor System
memory	Conduit
hps_io	Conduit
h2f_reset	Reset Output
h2f_axi_clock	Clock Input
h2f_axi_master	AXI Master
f2h_axi_clock	Clock Input
f2h_axi_slave	AXI Slave
h2f_lw_axi_clock	Clock Input
h2f_lw_axi_master	AXI Master
⊟ **onchip_memory2_0**	On-Chip Memory (RAM or ROM)
clk1	Clock Input
s1	Avalon Memory Mapped Slave
reset1	Reset Input
⊟ **master_secure**	JTAG to Avalon Master Bridge

Fig. 25: On-Chip RAM

- There are two components described as JTAG to Avalon Master Bridge: **Secure JTAG Master** (**master_secure** in Qsys[5]) and **Non-Secure JTAG Master** (**master_non_sec** in Qsys). These are master components that accept encoded streams of bytes of transaction data on the JTAG interface and initiate Avalon-MM (Multi Master) transactions on the Avalon-MM interface. The JTAG to Avalon Master is also used for debugging, with tools such as System Console and SignalTap.
  Secure JTAG Master is for accessing secure peripherals in the HPS through the FPGA-to-HPS interface; it is connected to the high bandwidth FPGA-to-HPS bridge (see Figure 24 and Figure 26).

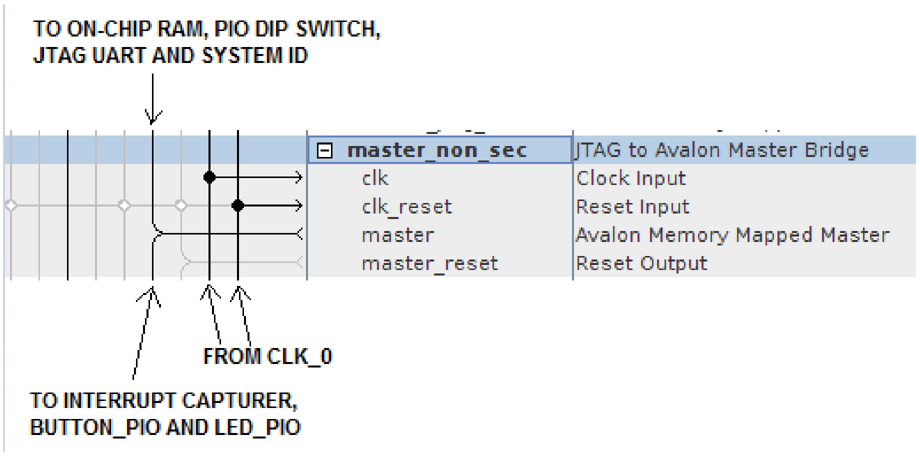


⊟ **hps_0**	Hard Processor System
memory	Conduit
hps_io	Conduit
h2f_reset	Reset Output
h2f_axi_clock	Clock Input
h2f_axi_master	AXI Master
f2h_axi_clock	Clock Input
f2h_axi_slave	AXI Slave
h2f_lw_axi_clock	Clock Input
h2f_lw_axi_master	AXI Master
⊟ **onchip_memory2_0**	On-Chip Memory (RAM or ROM)
clk1	Clock Input
s1	Avalon Memory Mapped Slave
reset1	Reset Input
⊟ **master_secure**	JTAG to Avalon Master Bridge
clk	Clock Input
clk_reset	Reset Input
master	Avalon Memory Mapped Master
master_reset	Reset Output

Fig. 26: Secure JTAG Master

Non-Secure JTAG Master is for accessing non-secure peripherals in the FPGA fabric; it is connected to the lightweight HPS-to-FPGA bridge, the On-Chip RAM and the Interrupt Capturer (see Figure 24 and Figure 27).

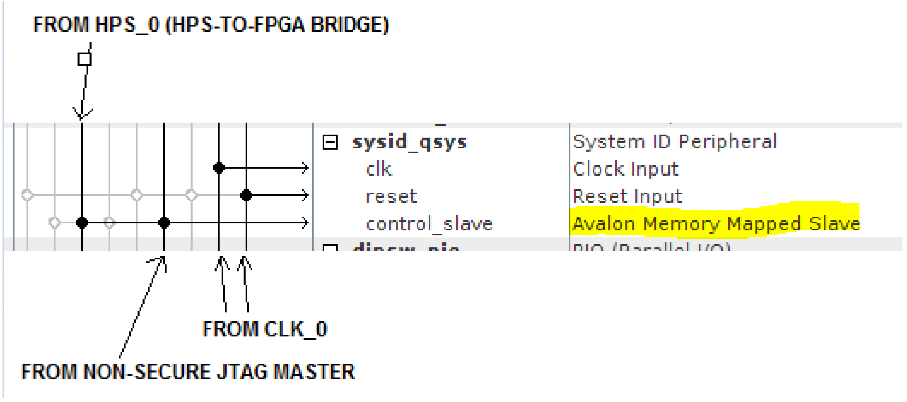
[5] We use the 13.0sp1 version of the Altera software. In newer versions, the Secure JTAG Master is named **hps_only_master**, and the Non-Secure JTAG Master is named **fpga_only_master**


TO ON-CHIP RAM, PIO DIP SWITCH, JTAG UART AND SYSTEM ID
FROM CLK_0
TO INTERRUPT CAPTURER, BUTTON_PIO AND LED_PIO

master_non_sec	JTAG to Avalon Master Bridge
clk	Clock Input
clk_reset	Reset Input
master	Avalon Memory Mapped Master
master_reset	Reset Output

Fig. 27: Non-Secure JTAG Master

- **System ID** (**sysid_qsys** in Qsys): This is a very important peripheral to include on the system. Allows the software development tools to validate an application is being developed for the appropriate hardware. Basically, it will not permit to execute software in an incompatible hardware configuration. It is connected as a memory mapped slave to the low bandwidth HPS-to-FPGA bridge and the Non-Secure JTAG Master (see Figure 24 and Figure 28).



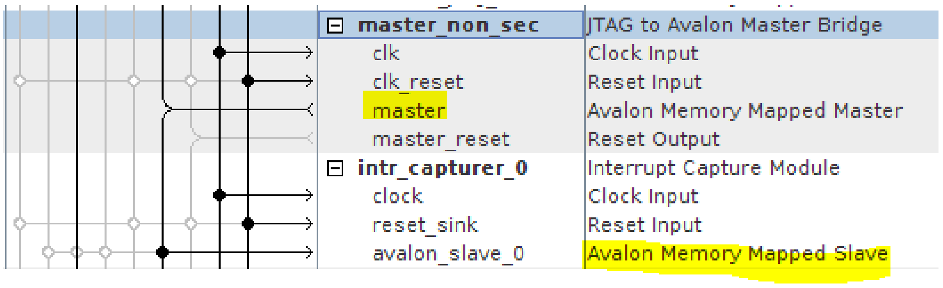
sysid_qsys	System ID Peripheral
clk	Clock Input
reset	Reset Input
control_slave	Avalon Memory Mapped Slave

Fig. 28: System ID

- **PIO DIP Switch** (**dipsw_pio** in Qsys): SoCkit board has four switches connected to FPGA. This component is a PIO (Peripheral Input/Output) input peripheral, used to read the state of these switches. It has the same connections than the System ID component, with an additional conduit signal since it is connected to an external peripheral, the switches (they do not belong to the HPS system).

- **JTAG UART** (**jtag_uart** in Qsys): Software developers need to have access to a debug serial port from the target for debugging, input control commands, log state information, etc. This component connects to the debug console and provides an interface for it. It has the same connections than the System ID component.

- **Interrupt capturer** (**interrupt_capturer** in Qsys): This component is a memory mapped Avalon module (written in Verilog language) used to capture system interrupts and pass them to the HPS Cortex-A9 MPU. It is connected to the Non-Secure JTAG Master (see Figure 24 and Figure 29).



⊟ master_non_sec	JTAG to Avalon Master Bridge
clk	Clock Input
clk_reset	Reset Input
master	Avalon Memory Mapped Master
master_reset	Reset Output
⊟ intr_capturer_0	Interrupt Capture Module
clock	Clock Input
reset_sink	Reset Input
avalon_slave_0	Avalon Memory Mapped Slave

Fig. 29: Interrupt capturer

- **PIO Button** (**button_pio** in Qsys): SoCkit board has four buttons connected to FPGA. Just as "dipsw_pio", this is a PIO input peripheral used for reading the status of these buttons. It has the same connections than the PIO DIP Switch component.

- **PIO LED** (**led_pio** in Qsys): SoCkit board has four LEDs connected to FPGA. This component is an output PIO peripheral to set LEDs configuration. It has the same connections than the PIO DIP Switch component and the PIO Button component.

- **Clock** (**clk_0** in Qsys): There is not much to say about this component; it is the system clock.

You can see the configuration of every component by double clicking on its name.

To this point, we have the hardware design of the SoC system, configured and ready to be compiled so that we can generate files for the Linux distribution. Before that, in the next section we are going to create and add a new custom component to our hardware design.

7.5. Creation and configuration of a new custom component. Adding the new component to the system and checking its correct operation.

We go to the next step in this project, as shown in Figure 30.

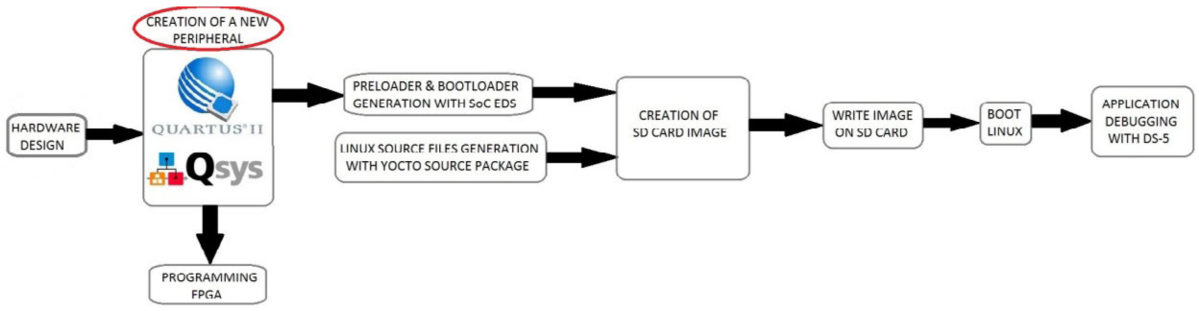

CREATION OF A NEW PERIPHERAL
HARDWARE DESIGN
QUARTUS II
Qsys
PROGRAMMING FPGA
PRELOADER & BOOTLOADER GENERATION WITH SoC EDS
LINUX SOURCE FILES GENERATION WITH YOCTO SOURCE PACKAGE
CREATION OF SD CARD IMAGE
WRITE IMAGE ON SD CARD
BOOT LINUX
APPLICATION DEBUGGING WITH DS-5

Fig. 30: Second step. Creating a new peripheral

7.5.1. Implementing/coding the component

In the Qsys tool, there are library components available for using them. These components available as libraries are actually hardware subcircuits, divided into two parts: the internal hardware modules, that implement the functionality of the component; and the external Avalon interfaces, used by the component to communicate with the external hardware modules.

In this section, we are going to create a new component from scratch. It is a really simple component: a 16-bit register. It just store data written to its memory address by us and it modifies its value; it modifies the data so that we have a better insurance that our component works correctly because, when we read the data again from it, we will not read the value we wrote, but that value with the corresponding changes applied (we will see how it works exactly in the next pages).

There are several Avalon interfaces. These interfaces help to design a system easier by allowing to connect components in an Altera FPGA; they are designed into the components available in Qsys. They can be used also in custom components; in fact, we will use one of them in our new component. We will not explain all the different Avalon interfaces; if you want to go further, please refer to Altera's "Avalon Interface Specifications" in bibliography.

We will use the Avalon-Multi Master interface, used for master and slave components in a memory-mapped system that have master and slave interfaces connected by an interconnect fabric (see Figure 31).

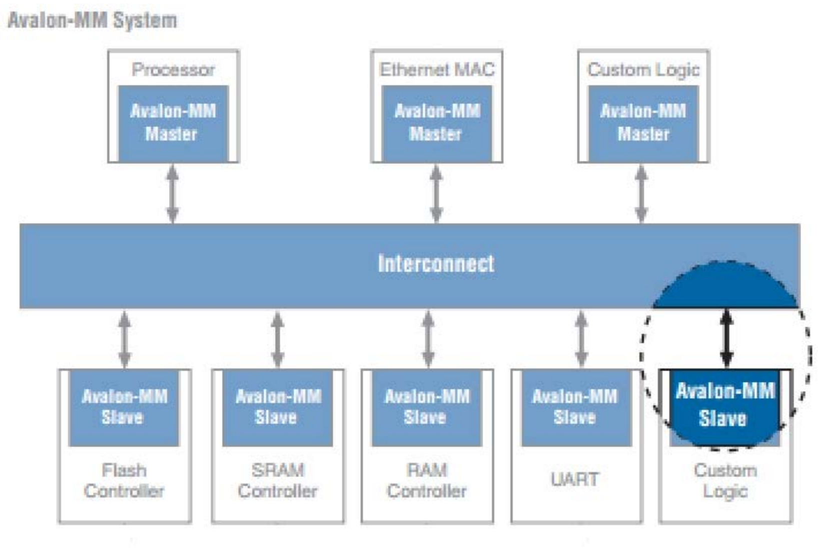


Fig. 31: Avalon-MM interface

Avalon-MM components normally include only the required for the component logic. In Figure 32 is shown a 16-bit general purpose I/O peripheral that only responds to write requests, so it has only the slave signals required for write operations.

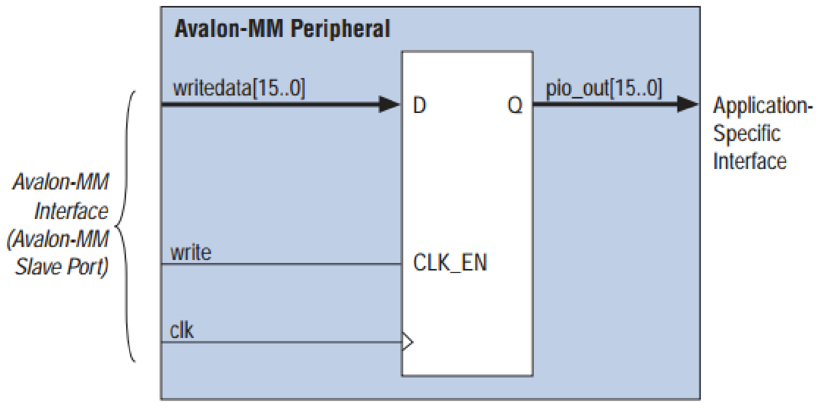

Avalon-MM Peripheral
writedata[15..0]
D
Q
pio_out[15..0]
Application-Specific Interface
Avalon-MM Interface (Avalon-MM Slave Port)
write
CLK_EN
clk

Fig. 32: Example slave component

To see all signals that constitute the Avalon-MM interface, please refer to Altera's "Avalon Interface Specifications" in bibliography. The signals we use for our component are:

- *clock*
- *resetn*
- *readdata*: 16-bit data read from the register.
- *writedata*: 16-bit data written on the register.
- *read*: A bit active when a reading is occurring.
- *write*: A bit active when a writing is occurring.

We will also use the Avalon Conduit Interface, an interface that accommodates signals that do not fit into any other interface types. A conduit signal can be any type of signal. A conduit signal can be used to connect to external devices and to FPGA logic defined outside the Qsys system.

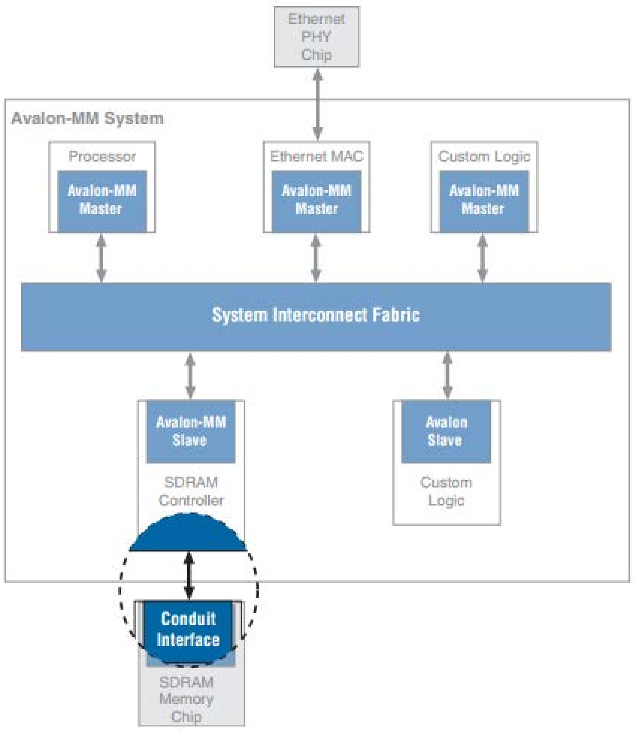


Fig. 33: Conduit interface

Our component has two parts, the internal modules and the Avalon interface, so we need two different files to implement the functionality of each part. VHDL or Verilog can be used as source code. We will use VHDL given that it is more familiar to us, but we will illustrate the Verilog source code of our component. Firstly, the file implementing the register contains the following code:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY reg16 IS
        PORT(   clock, resetn: IN STD_LOGIC;
                D: IN STD_LOGIC_VECTOR(15 DOWNTO 0);
                Q: OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END reg16;
ARCHITECTURE Behaviour OF reg16 IS
BEGIN
        PROCESS
        BEGIN
                WAIT UNTIL clock'EVENT AND clock = '1';
                IF resetn = '0' THEN
                        Q <= "0000000000000000";
                ELSE
                        Q(15 DOWNTO 0) <= D(15 DOWNTO 0) OR "1000000000000001";
                END IF;
        END PROCESS;
END Behaviour;
```

This file is named "reg16". Let D be the 16-bit input data and Q the 16-bit output data of the register.

The modification of the data can be viewed in the code above. The read data is modified with an OR operation with value 0x8001, so that if a 0x0002 is written, then it would be read the value 0x8003. Hence, we would know our component is responding and correctly working.

In the Avalon-MM interface, one of the components that exchange data implements a master interface that allows it to request and send data to slave components. A slave component can only receive and process petitions, either getting data from the master or sending requested data. Each slave component includes at least one register accessible for reading or writing by a master component. All transactions are synchronized with the rising edge of the Avalon clock signal.

There is a signal named "Q_export". This signal is used by the Avalon Conduit Interface, so it is exported outside of the Qsys system. The purpose of using this conduit is to be capable of displaying the data values of the register on external components such as LEDs.

The VHDL code for the Avalon interface is the following (the file name is "reg16_avalon_interface"):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY reg16_avalon_interface IS
	PORT(	clock, resetn: IN STD_LOGIC;
		read,write: IN STD_LOGIC;
		writedata: IN STD_LOGIC_VECTOR(15 DOWNTO 0);
		readdata: OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
		Q_export: OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END reg16_avalon_interface;
ARCHITECTURE Structure OF reg16_avalon_interface IS
	SIGNAL to_reg, from_reg: STD_LOGIC_VECTOR(15 DOWNTO 0);
	COMPONENT reg16
		PORT(	clock, resetn: IN STD_LOGIC;
			D: IN STD_LOGIC_VECTOR(15 DOWNTO 0);
			Q: OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
	END COMPONENT;
BEGIN
	to_reg <= writedata;
	reg_instance: reg16 PORT MAP(clock, resetn, to_reg, from_reg);
	readdata <= from_reg;
	Q_export <= from_reg;
END Structure;
```

In the Appendix B there is information about the VHDL code to explain why the component is implemented in this way.

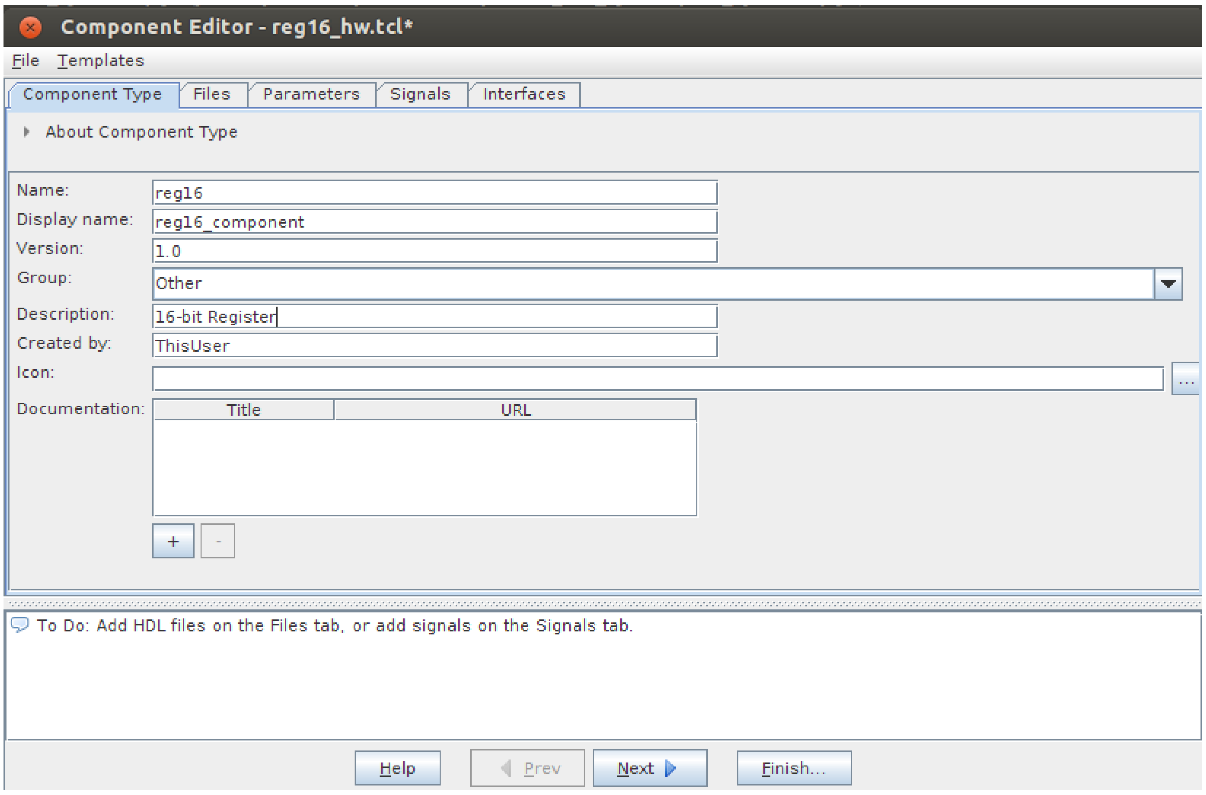
On the next page is the Verilog code for the register and the Avalon interface, respectively. We will not explain the code since, as we said, we have chose VHDL as the preferred HDL.

```
module reg16(clock, resetn, D, Q);
	input clock, resetn;
	input [15:0] D;
	output reg [15:0] Q;
	always@(posedge clock)
		if(!resetn)
			Q <= 16'b0;
		else
		begin
			Q [15:0] <= D [15:0];
		end
endmodule
```

```
module reg16_avalon_interface(clock, resetn, writedata, readdata, write,
read, Q_export);
        input clock, resetn, read, write;
        input [15:0] writedata;
        output [15:0] readdata;
        output [15:0] Q_export;
        wire [15:0] to_reg, from_reg;
        assign to_reg = writedata;
        reg16 U1(.clock(clock), .resetn(resetn), .D(to_reg), .Q(from_reg));
        assign readdata = from_reg;
        assign Q_export = from_reg;
endmodule
```

7.5.2. Adding the new component in Qsys

In this subsection we will learn how to create a new Qsys component for our new 16-bit register. **In Qsys, click on "New component..." button in the component library area. Fill in the fields of the window that appears just as in Figure 34. Click Next.**

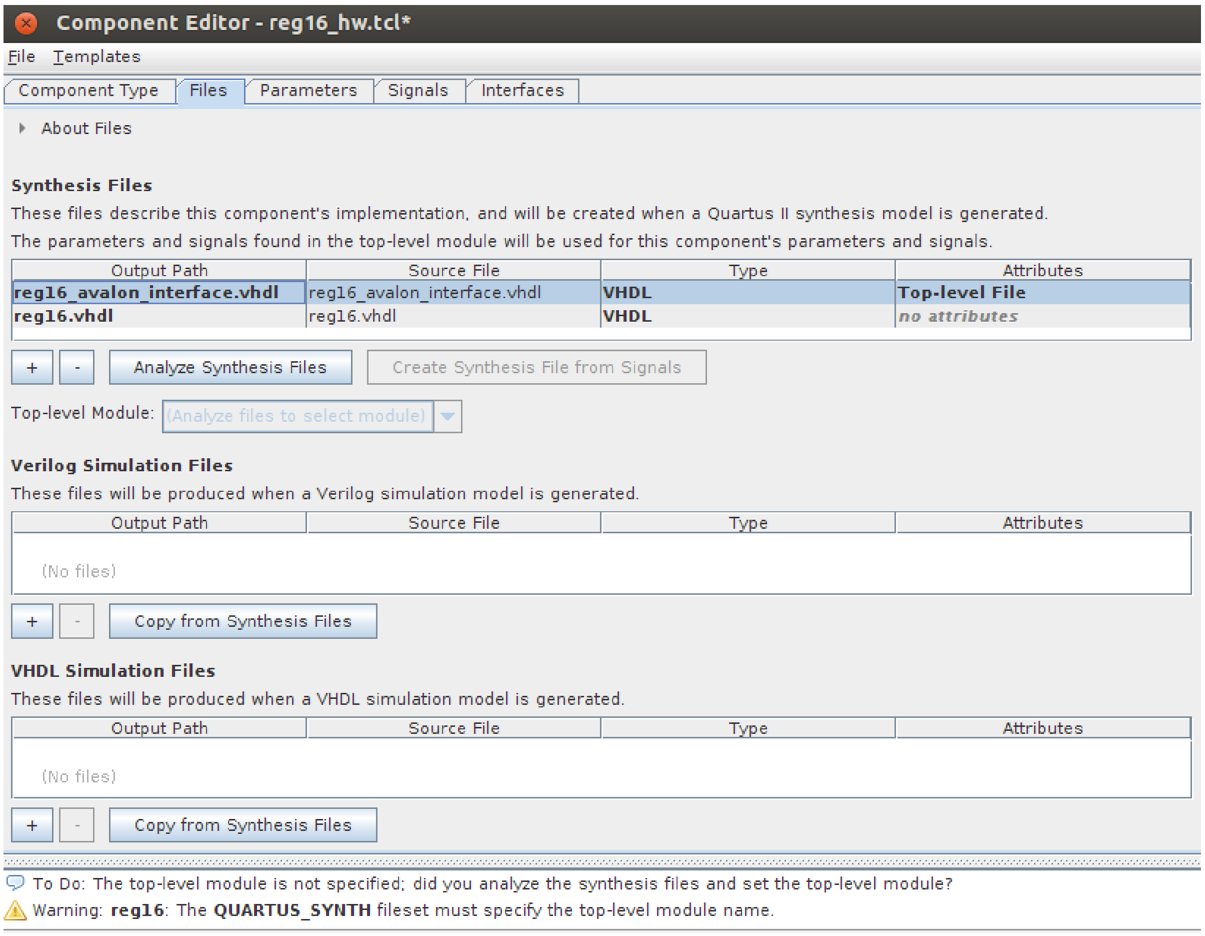


Name: reg16
Display name: reg16_component
Version: 1.0
Group: Other
Description: 16-bit Register
Created by: ThisUser
Icon:
Documentation:

To Do: Add HDL files on the Files tab, or add signals on the Signals tab.

Fig. 34: Configuring the new component in Qsys

The next step is to add the files that describe the component. **In the Synthesis Files area (see Figure 35), click in the + button and add the "reg16_avalon_interface.vhdl" file. Add also the "reg16.vhdl" file.** If we were going to make a simulation to check if our components works well, we would copy these two files to the Verilog Simulation Files or VHDL Simulation Files area (according to the source code used). We are not going to check our component by simulation, instead we will do a real test so we don't need to copy the files there.



Component Editor - reg16_hw.tcl*

File Templates

Component Type | Files | Parameters | Signals | Interfaces

About Files

Synthesis Files

These files describe this component's implementation, and will be created when a Quartus II synthesis model is generated.
The parameters and signals found in the top-level module will be used for this component's parameters and signals.

Output Path	Source File	Type	Attributes
reg16_avalon_interface.vhdl	reg16_avalon_interface.vhdl	VHDL	Top-level File
reg16.vhdl	reg16.vhdl	VHDL	no attributes

+ | - | Analyze Synthesis Files | Create Synthesis File from Signals

Top-level Module: (Analyze files to select module)

Verilog Simulation Files

These files will be produced when a Verilog simulation model is generated.

Output Path	Source File	Type	Attributes
(No files)			

+ | - | Copy from Synthesis Files

VHDL Simulation Files

These files will be produced when a VHDL simulation model is generated.

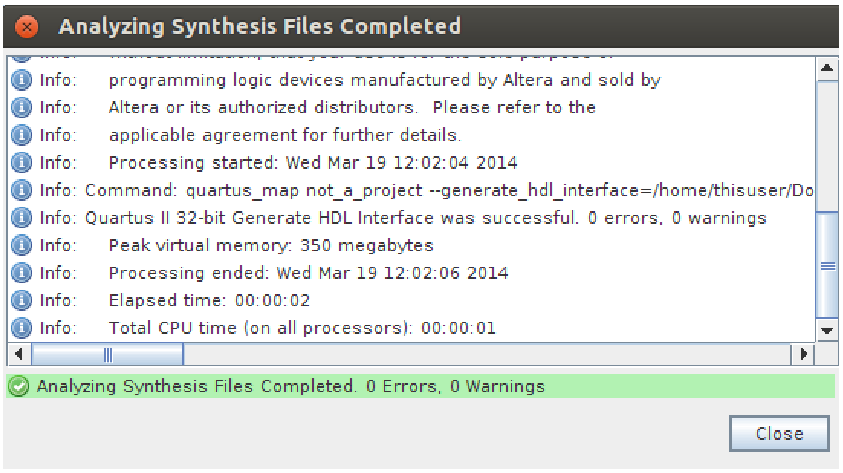
Output Path	Source File	Type	Attributes
(No files)			

+ | - | Copy from Synthesis Files

To Do: The top-level module is not specified; did you analyze the synthesis files and set the top-level module?
Warning: **reg16**: The **QUARTUS_SYNTH** fileset must specify the top-level module name.

Fig. 35: Synthesis Files

Click on the button Analyze Synthesis Files. Qsys starts analyzing the code of the top level entity, "reg16_avalon_interface"; if everything goes well, all the signals read from this file are added after several seconds (see Figure 36).



Analyzing Synthesis Files Completed

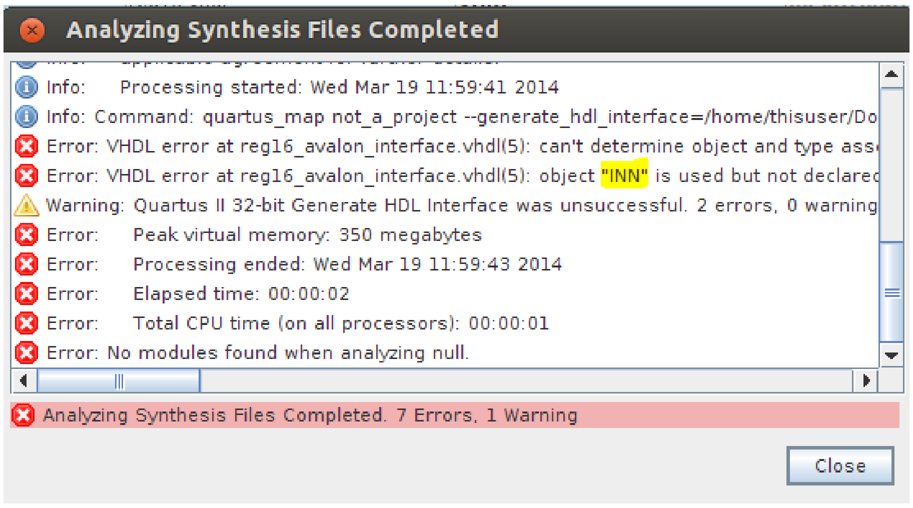
```
Info:     programming logic devices manufactured by Altera and sold by
Info:     Altera or its authorized distributors.  Please refer to the
Info:     applicable agreement for further details.
Info:     Processing started: Wed Mar 19 12:02:04 2014
Info: Command: quartus_map not_a_project --generate_hdl_interface=/home/thisuser/Do
Info: Quartus II 32-bit Generate HDL Interface was successful. 0 errors, 0 warnings
Info:     Peak virtual memory: 350 megabytes
Info:     Processing ended: Wed Mar 19 12:02:06 2014
Info:     Elapsed time: 00:00:02
Info:     Total CPU time (on all processors): 00:00:01
Analyzing Synthesis Files Completed. 0 Errors, 0 Warnings
```

Fig. 36: Analyzing synthesis files completed without errors

If we try to type a wrong code line in the VHDL file, the analysis will not end successfully. For example, if we modify the entity of "reg16_avalon_interface" just as below

```
read,write: INN STD_LOGIC; (instead of "IN")
```

We will see the following error messages in the analysis window, as shown in Figure 37.



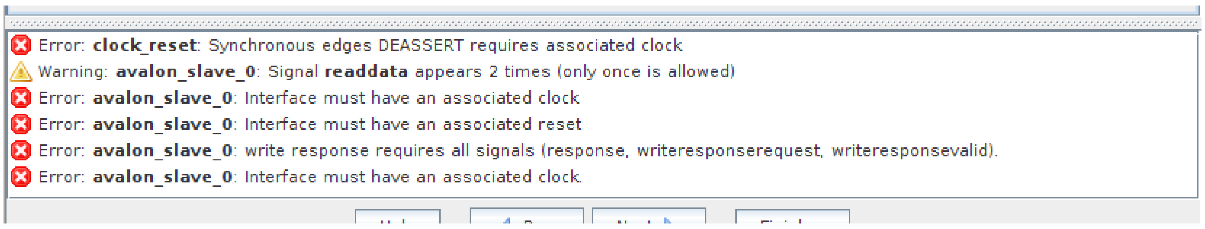
Analyzing Synthesis Files Completed

```
Info:     Processing started: Wed Mar 19 11:59:41 2014
Info: Command: quartus_map not_a_project --generate_hdl_interface=/home/thisuser/Do
Error: VHDL error at reg16_avalon_interface.vhdl(5): can't determine object and type ass
Error: VHDL error at reg16_avalon_interface.vhdl(5): object "INN" is used but not declarec
Warning: Quartus II 32-bit Generate HDL Interface was unsuccessful. 2 errors, 0 warning
Error:     Peak virtual memory: 350 megabytes
Error:     Processing ended: Wed Mar 19 11:59:43 2014
Error:     Elapsed time: 00:00:02
Error:     Total CPU time (on all processors): 00:00:01
Error: No modules found when analyzing null.
Analyzing Synthesis Files Completed. 7 Errors, 1 Warning
```

Fig. 37: Errors in the synthesis files

If the VHDL files are modified after adding them and analyzing them, it is necessary to remove and add them again to re-analyze them.

You can see in Figure 35 an info message and a warning message telling us that the module is not specified, and asking if we analyzed the synthesis files. These messages disappear when we analyze the synthesis files. Some errors will appear after that (see Figure 38), but they will be removed as we keep configuring our component.

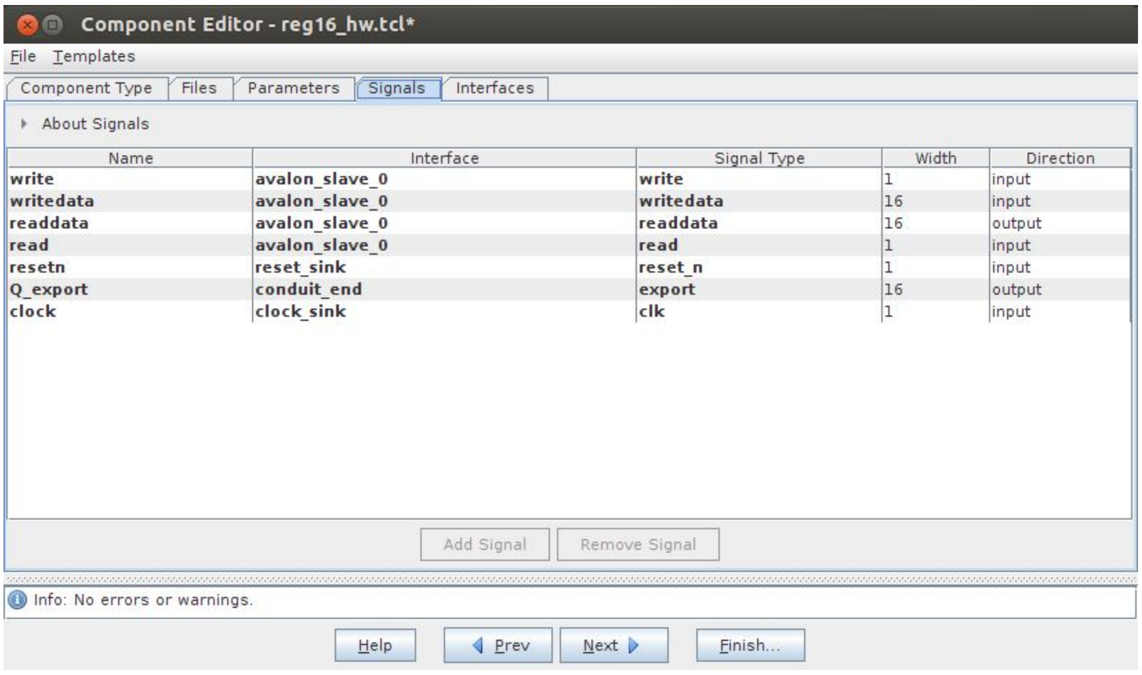


```
Error: clock_reset: Synchronous edges DEASSERT requires associated clock
Warning: avalon_slave_0: Signal readdata appears 2 times (only once is allowed)
Error: avalon_slave_0: Interface must have an associated clock
Error: avalon_slave_0: Interface must have an associated reset
Error: avalon_slave_0: write response requires all signals (response, writeresponserequest, writeresponsevalid).
Error: avalon_slave_0: Interface must have an associated clock.
```

Fig. 38: Some temporal errors in the configuration of the component

Click Next twice; we will ignore the Parameters tab because our component has no configurable elements, its flexibility is not important (parameters would be useful in the case of, for example, a component with a variable memory size or data length).

Now we will step into the Signals tab, where we will specify the purpose of each interface port in the top level entity ("reg16_avalon_interface"). The interface type by default of all signals is "avalon_slave_0". We have to change the interface for the clock signal, the reset signal and the conduit. To do that, click on the interface of the corresponding signal. In the drop-down list, select "new Clock Input..." for the clock signal, "new Reset Input..." for the reset signal and "new Conduit..." for the conduit. **Check if the final configuration of the signals matches the configuration in Figure 39. Click Next.**


Component Editor - reg16_hw.tcl*
File Templates
Component Type Files Parameters Signals Interfaces
About Signals
Add Signal Remove Signal
Help Prev Next Finish...

Name	Interface	Signal Type	Width	Direction
write	avalon_slave_0	write	1	input
writedata	avalon_slave_0	writedata	16	input
readdata	avalon_slave_0	readdata	16	output
read	avalon_slave_0	read	1	input
resetn	reset_sink	reset_n	1	input
Q_export	conduit_end	export	16	output
clock	clock_sink	clk	1	input

Info: No errors or warnings.

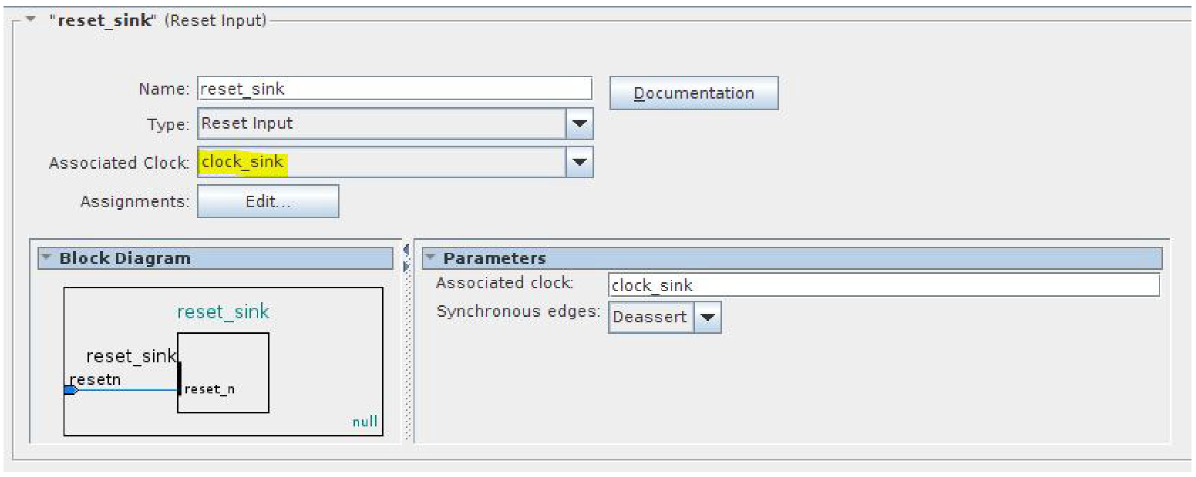
Fig. 39: Signals of the new component

Lastly, in the Interfaces tab, we can see all the interfaces of the component. Qsys components can include various types of interfaces:

- Clocks
- Resets
- Interrupts
- Conduits (for signals that do not fit into any other type. Conduit signals can be exported from the Qsys system)
- Streaming: For unidirectional traffic.
- Memory-Mapped: For traffic between master and slaves.

Our component contains four interfaces: clock, reset, conduit and memory mapped interface (the latter because it is a slave component).

We have to associate the clock signal to the reset (see Figure 40). We also have to make sure that both signals, "clock_sink" and "reset_sink", are associated to our slave component, "avalon_slave_0", and the external signal "conduit_end".


"reset_sink" (Reset Input)

Name: reset_sink

Type: Reset Input

Associated Clock: clock_sink

Assignments: Edit...

Parameters

Associated clock: clock_sink

Synchronous edges: Deassert

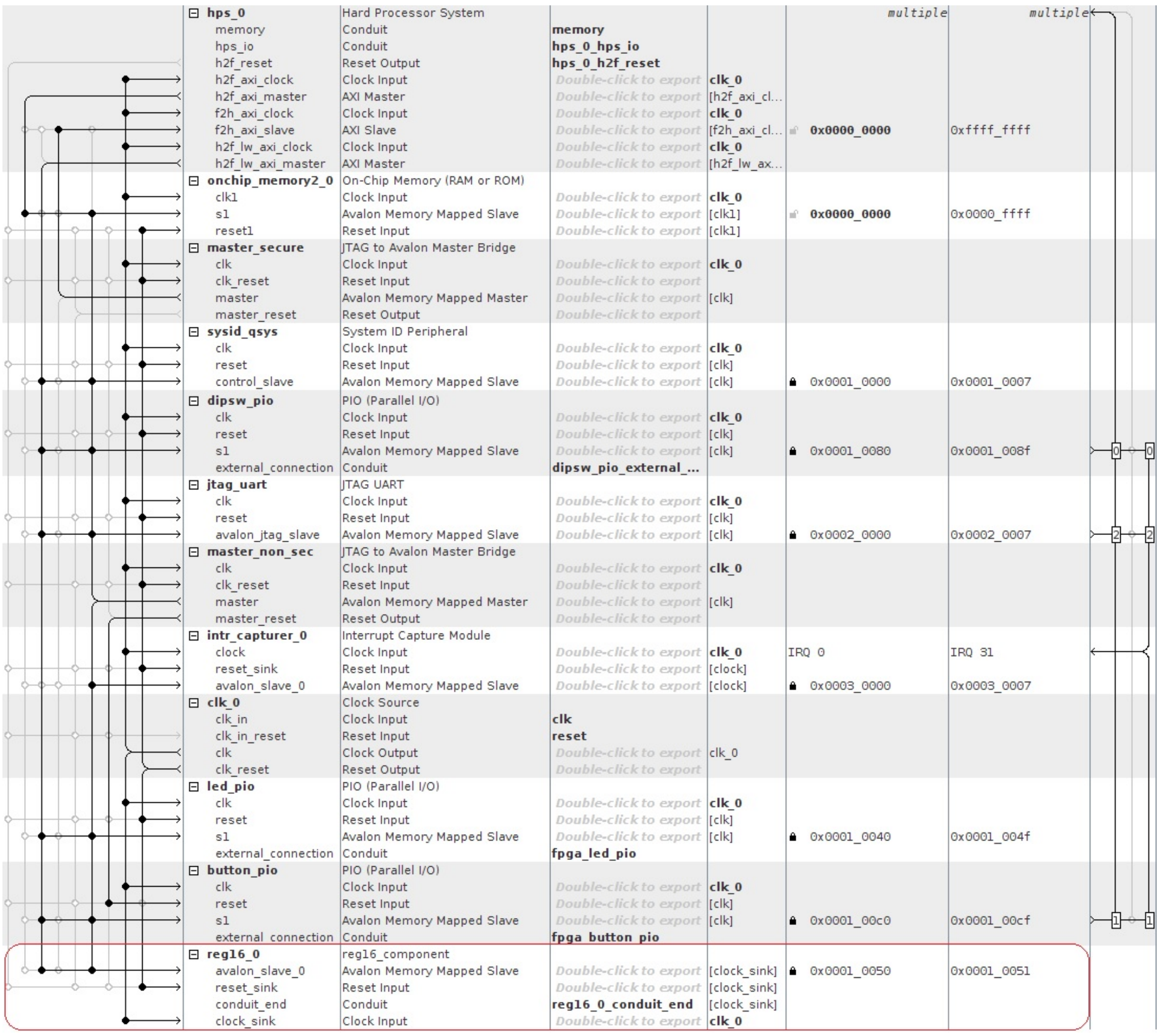
Fig. 40: Clock associated to the reset of the component

Note that an error message still remains because Qsys assumes that there must be some interface named "clock_reset"; hence, Qsys creates it. Even though, we chose a different interface for the clock, getting as a result a new interface named "clock_sink". **Click on the "Remove Interfaces With No Signals" button, so "clock_reset" will be removed. Click "Finish" to save the configuration of our new component.**

As we said before, **if the VHDL source code of the new component is modified after adding it to Qsys,** it is necessary to update it in Qsys so that any change can be reflected in the system. To do that, **expand the "Other" item in the components library area**. Our component "reg16_component" will be displayed. **Right click on it and select Edit. In the Files tab, click on the Analyze Synthesis Files button** as we did when we created the component, and the Qsys component will be modified according to the VHDL files.

7.5.3. Instantiating the new component

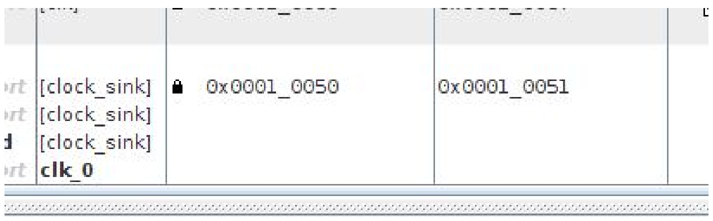
Expand the "Other" item in the components library area. Double click on the "reg16_component" and a w indow will appear. Click on "Finish" to add the component. A new component will be displayed in the main Qsys window. As seen in Figure 41, **make all the required connections to the component**: clock, reset and connection to the master component that communicates with our component using the Avalon-MM interface. **Finally, click on "Click to export..." to specify the name of the external conduit signal.**



⊟ **hps_0**	Hard Processor System			*multiple*	*multiple*
memory	Conduit	**memory**			
hps_io	Conduit	**hps_0_hps_io**			
h2f_reset	Reset Output	**hps_0_h2f_reset**			
h2f_axi_clock	Clock Input	*Double-click to export*	**clk_0**		
h2f_axi_master	AXI Master	*Double-click to export*	[h2f_axi_cl...		
f2h_axi_clock	Clock Input	*Double-click to export*	**clk_0**		
f2h_axi_slave	AXI Slave	*Double-click to export*	[f2h_axi_cl...	🔓 **0x0000_0000**	0xffff_ffff
h2f_lw_axi_clock	Clock Input	*Double-click to export*	**clk_0**		
h2f_lw_axi_master	AXI Master	*Double-click to export*	[h2f_lw_ax...		
⊟ **onchip_memory2_0**	On-Chip Memory (RAM or ROM)				
clk1	Clock Input	*Double-click to export*	**clk_0**		
s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk1]	🔓 **0x0000_0000**	0x0000_ffff
reset1	Reset Input	*Double-click to export*	[clk1]		
⊟ **master_secure**	JTAG to Avalon Master Bridge				
clk	Clock Input	*Double-click to export*	**clk_0**		
clk_reset	Reset Input	*Double-click to export*			
master	Avalon Memory Mapped Master	*Double-click to export*	[clk]		
master_reset	Reset Output	*Double-click to export*			
⊟ **sysid_qsys**	System ID Peripheral				
clk	Clock Input	*Double-click to export*	**clk_0**		
reset	Reset Input	*Double-click to export*	[clk]		
control_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	🔒 0x0001_0000	0x0001_0007
⊟ **dipsw_pio**	PIO (Parallel I/O)				
clk	Clock Input	*Double-click to export*	**clk_0**		
reset	Reset Input	*Double-click to export*	[clk]		
s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	🔒 0x0001_0080	0x0001_008f
external_connection	Conduit	**dipsw_pio_external_...**			
⊟ **jtag_uart**	JTAG UART				
clk	Clock Input	*Double-click to export*	**clk_0**		
reset	Reset Input	*Double-click to export*	[clk]		
avalon_jtag_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	🔒 0x0002_0000	0x0002_0007
⊟ **master_non_sec**	JTAG to Avalon Master Bridge				
clk	Clock Input	*Double-click to export*	**clk_0**		
clk_reset	Reset Input	*Double-click to export*			
master	Avalon Memory Mapped Master	*Double-click to export*	[clk]		
master_reset	Reset Output	*Double-click to export*			
⊟ **intr_capturer_0**	Interrupt Capture Module				
clock	Clock Input	*Double-click to export*	**clk_0**	IRQ 0	IRQ 31
reset_sink	Reset Input	*Double-click to export*	[clock]		
avalon_slave_0	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	🔒 0x0003_0000	0x0003_0007
⊟ **clk_0**	Clock Source				
clk_in	Clock Input	**clk**			
clk_in_reset	Reset Input	**reset**			
clk	Clock Output	*Double-click to export*	clk_0		
clk_reset	Reset Output	*Double-click to export*			
⊟ **led_pio**	PIO (Parallel I/O)				
clk	Clock Input	*Double-click to export*	**clk_0**		
reset	Reset Input	*Double-click to export*	[clk]		
s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	🔒 0x0001_0040	0x0001_004f
external_connection	Conduit	**fpga_led_pio**			
⊟ **button_pio**	PIO (Parallel I/O)				
clk	Clock Input	*Double-click to export*	**clk_0**		
reset	Reset Input	*Double-click to export*	[clk]		
s1	Avalon Memory Mapped Slave	*Double-click to export*	[clk]	🔒 0x0001_00c0	0x0001_00cf
external_connection	Conduit	**fpga_button_pio**			
⊟ **reg16_0**	reg16_component				
avalon_slave_0	Avalon Memory Mapped Slave	*Double-click to export*	[clock_sink]	🔒 0x0001_0050	0x0001_0051
reset_sink	Reset Input	*Double-click to export*	[clock_sink]		
conduit_end	Conduit	**reg16_0_conduit_end**	[clock_sink]		
clock_sink	Clock Input	*Double-click to export*	**clk_0**		

Fig. 41: Connections of the new component in Qsys

Next, we have to **modify the base address of our new component**. In our case, we will **assign the address 0x10050 to it**, just above the memory address range of the LEDs (0x10040-0x1004f). Figure 42 shows the memory address of our component modified.



```
[clock_sink]  🔒 0x0001_0050    0x0001_0051
[clock_sink]
[clock_sink]
clk_0
```

Fig. 42: Memory address of the new component

The complete address will be 0xff210050.What is the reason?

There are three bridges connecting the HPS and FPGA portion of the SoC. Two of them are meant for high bandwidth data transactions, HPS2FPGA and FPGA2HPS. The third bridge, LWHPS2FPGA, is used as a control/status path for the HPS into the FPGA. The HPS can separately control low bandwidth FPGA peripherals and the high bandwidth flow of data. Some peripherals ("dipsw_pio", "sysid_qsys", "led_pio", "button_pio" and our new component) are connected to this low bandwidth bridge. This bridge is mapped within the HPS peripherals span starting at 0xff200000 (see Figure 43). **In the "SoCkit lab materials" documents there is more info about the SoC memory management**.

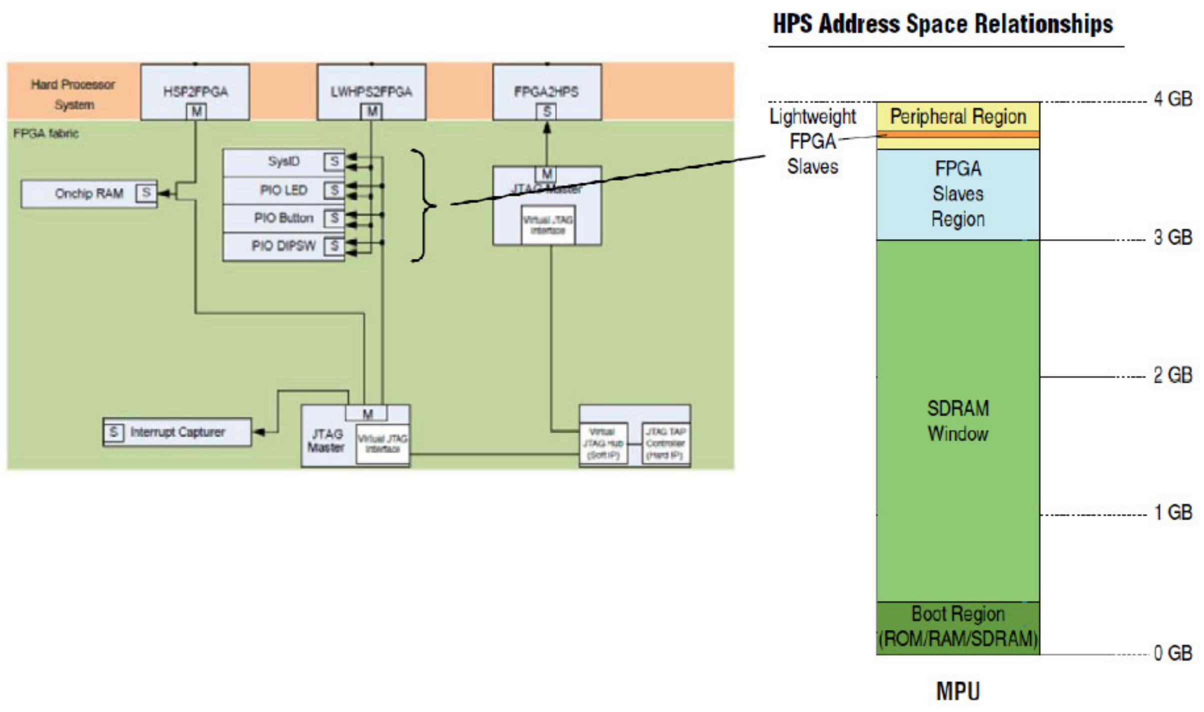
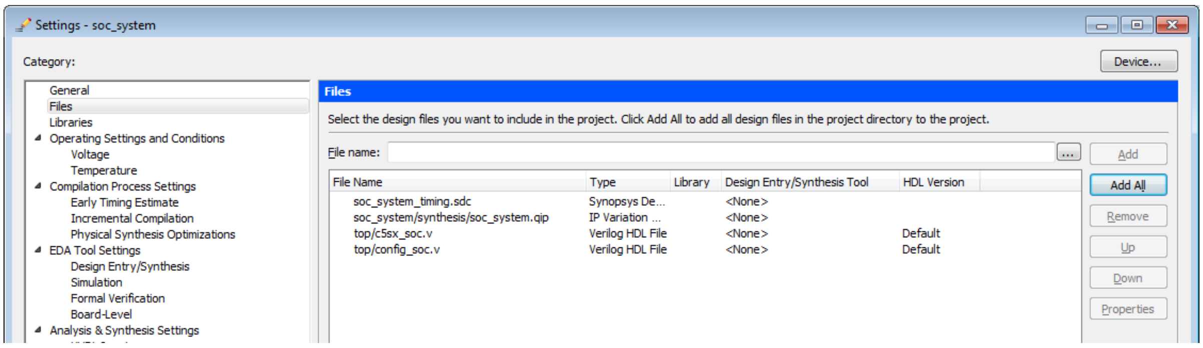


Fig. 43: The LWHPS2FPGA bridge

Since 0xff200000 is the base address of the LWHPS2FPGA bridge, our component has 0xff200000 as offset address. When we modify its address in Qsys to 0x0001_0050, its actual address is 0xff21_0050.

Finally, we have to complete the Quartus II project. **Save the Qsys project. In the Generate tab, click on the Generate button**. This will make Qsys generating HDL code that describes the system contents, including our new component and the Avalon interconnections. These files are used by the synthesis tool in Quartus II. They need to be added to the Quartus II project in order to be compiled.

Close the Qsys window. Go to "Assignments → Settings → Files" in Quartus II and click on the "..." browse button (see Figure 44). Search for a file named "soc_system.qip" included in the ".../GHRD_soc_system /soc_system/synthesis" folder. Select "Open", then add it to the project clicking on "Add". Follow the same process to add the "soc_system_timing.dsc" file included in the GHRD Quartus II project folder (if you cannot see the file when browsing, change "Files of type:" to "All files").

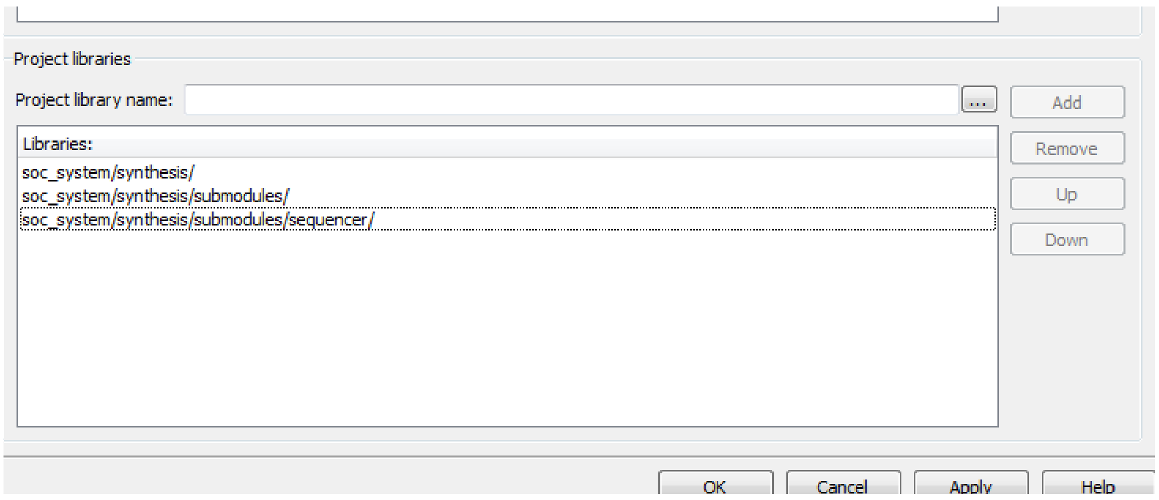


Select the design files you want to include in the project. Click Add All to add all design files in the project directory to the project.

File Name	Type	Library	Design Entry/Synthesis Tool	HDL Version
soc_system_timing.sdc	Synopsys De...		<None>	
soc_system/synthesis/soc_system.qip	IP Variation ...		<None>	
top/c5sx_soc.v	Verilog HDL File		<None>	Default
top/config_soc.v	Verilog HDL File		<None>	Default

Fig. 44: Adding files to the Quartus II project

Next, in the same Settings window, go to the "Libraries" category. We are going to add the synthesis directories to the project, used by Quartus II to compile the design. **In the "Project Libraries" section, click on the "..." button, select the ".../GHRD_soc_system /soc_system/synthesis/" folder and add it to the project. Do the same with the ".../soc_system/synthesis/submodules" and the ".../soc_system/synthesis/submodules/ sequencer" directories. Select Apply and OK to finish.**



Libraries:
soc_system/synthesis/
soc_system/synthesis/submodules/
soc_system/synthesis/submodules/sequencer/

Fig. 45: Adding libraries to the Quartus II project

The next step is to perform the analysis and synthesis in the Quartus II project. This stage analyzes and synthesizes the design files and creates a net-list within a project database. These nets can be assigned to actual device pins. **Select the icon of the purple arrow with the blue checkmark (Figure 46) in the Quartus II window to start analysis and synthesis**. This step might take a couple of minutes, depending on the computer. **Once it is completed, select OK.**

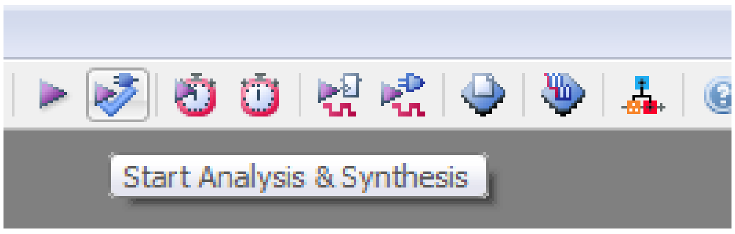
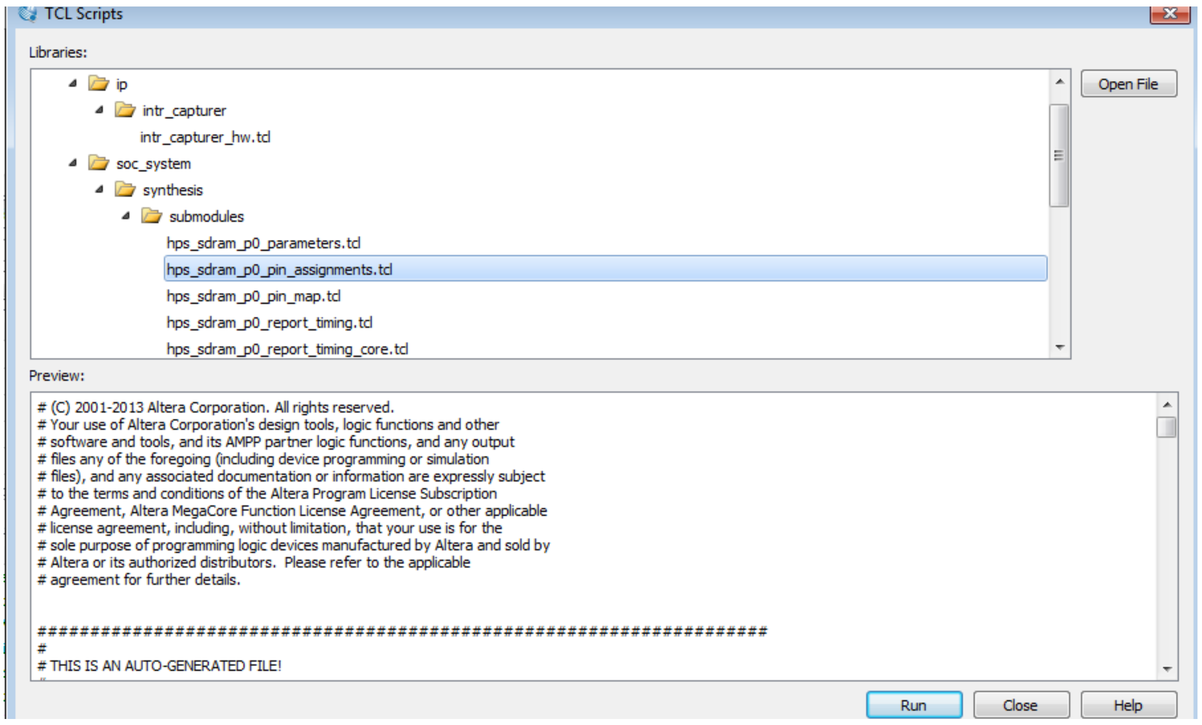


Fig. 46: Start Analysis & Synthesis

The HPS pin assignments are automatically specified when the HPS was instantiated in Qsys, but the HPS memory pins need to be assigned, since there are External Memory Interface variations that can occur. To assign them, there is a TCL (Tool Command Language) script created by Qsys for this purpose. To run it, **select "TCL script" in the Tools menu. Select the file "hps_sdram_p0_pin_assignments.tcl" and click on Run**. This step will take a couple of minutes as well.



```
# (C) 2001-2013 Altera Corporation. All rights reserved.
# Your use of Altera Corporation's design tools, logic functions and other
# software and tools, and its AMPP partner logic functions, and any output
# files any of the foregoing (including device programming or simulation
# files), and any associated documentation or information are expressly subject
# to the terms and conditions of the Altera Program License Subscription
# Agreement, Altera MegaCore Function License Agreement, or other applicable
# license agreement, including, without limitation, that your use is for the
# sole purpose of programming logic devices manufactured by Altera and sold by
# Altera or its authorized distributors.  Please refer to the applicable
# agreement for further details.


#####################################################################
#
# THIS IS AN AUTO-GENERATED FILE!
```

Fig. 47: Running the TCL script for pin assignment

Finally, to perform the complete compilation, **select "Start Compilation" in the Processing menu**. Be patient this time, this step will take longer than the previous ones (it may take about twenty or thirty minutes, it depends on the system). There should be no errors in the compile[6]; when it is finished, a dialog window will inform about the successful compilation. There will be some warnings, but we do not care about them because they do not affect the functionality of our system.

7.5.3. Checking the new component operation

In the previous section we have implemented our new component and integrated it into our project. In this section we are going to check that this component performs its function correctly.

To do this, we must use a tool included in Altera software: the System Console. This tool facilitates the visibility of our system for quick debugging. The hardware modules instantiated in the FPGA can be easily accessed. We will use one of its utilities that consist in reading and writing data on Avalon memory mapped slaves like our component through master components.

In order to test our component with System Console, we have to write a TCL script. There is a script named "test_one.tcl" in the Quartus "GHRD_soc_system " project directory; this script reads the value of the four FPGA buttons situated right to the eight switches of the board, and writes that value on the four FPGA LEDs (the SoCkit board has eight LEDs; the FPGA LEDs correspond to the right four ones).

We will write a new script. This script writes a value in the memory address where our component is mapped. It will read from the same address the written value modified by our component, and it will write it in the memory address where the FPGA LEDs are mapped.

Now we will explain the System Console operation flow and some commands we use to write our new TCL script. A high level flow comprises the following steps:

1. Add the required component to Qsys.
2. Connect the board and program the FPGA.
3. Launch System Console (to open it, access from Qsys to the Tools menu and select System Console).
4. Locate and open the service path.
5. Perform the desired operations with the service.
6. Close the service path.

When executing System Console, it automatically searches and detects all service instances based on JTAG and USB and recovers its services paths.

[6] If compilation is not successful and the following error appears: "out of memory in module quartus_fit", the system needs more RAM memory (in our case, the virtual machine had an initial RAM of 1 GB; we increased to 2 GB and the compilation was successful)

To open a service path, first we need to locate the available services. System Console uses a virtual file system to organize these services. There is the command "get_service_paths <service_type>" to obtain the service paths for a particular service. For example, with the following command:

```
set jtag_master [lindex [get_service_paths master] 0]
```

With "get_service_paths" we obtain all services of "master" type, that access to the mapped memory slaves. With "lindex" we get one element from the services list, and we choose the first element, with index 0. This element is assigned as a value to the variable "jtag_master" using the "set" command. This command is used in the "test_one.tcl" script. In System Console, the service paths might be different on every execution of the tool or on every version. Because of this, it is recommended to use this command to get them.

To use a particular service instance, there exists the "open_service" command. To open the master service we got before, the command would be the following:

```
open_service master $jtag_master
```

Next, do the desired operations. The only operations we will do are readings and writings on the components, and message sending to the console of System Console. We need three different commands.

To write data to a memory address, where 'X' can be 8, 16 or 32, depending on the number of bytes we want to access to, use the following command:

```
master_write_X <ruta-de-servicio>  <dirección> <valor>
```

To read data from a memory address. The last parameter consist of the number of bytes we want to read:

```
master_read_X <ruta-de-servicio>  <dirección> <tamaño-en-bytes>
```

To send messages to the console (we will use only "info" level messages):

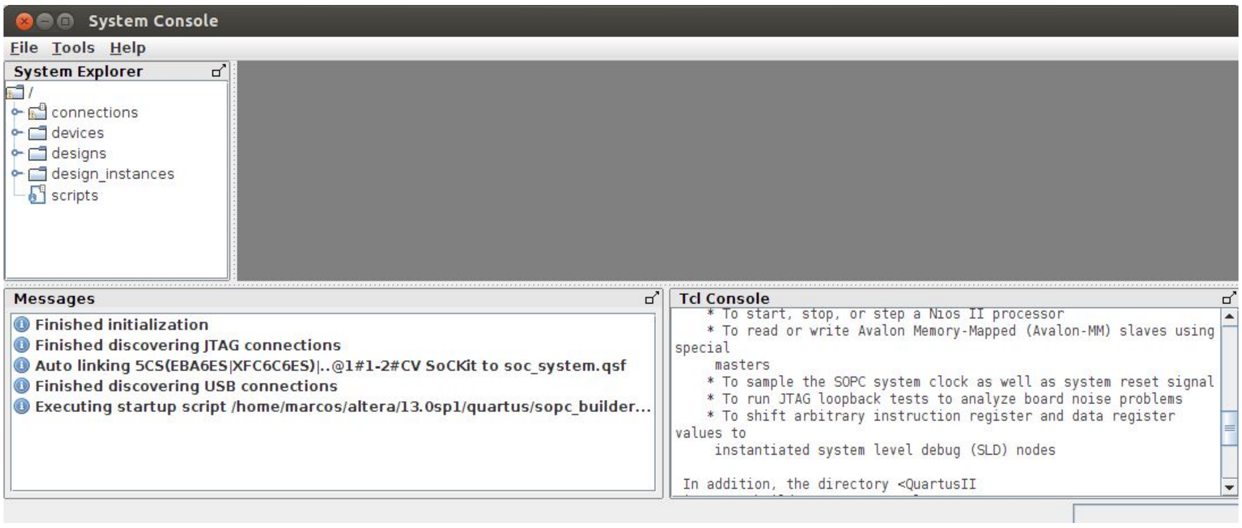
```
send_message <nivel> <mensaje>
```

After having clear the TCL scripts for System Console, observe how to implement the necessary operations for testing our component:

```
set jtag_master [lindex [get_service_paths master] 0] #Locate the service path

open_service master $jtag_master #Open the path

master_write_16 $jtag_master 0x1_0050 2 #Write '2' (0x0002, 16 bits) to the
                                      #memory address of our component
set valor [master_read_16 $jtag_master 0x1_0050 0x2] #Read the content from
                                        #the new component address (we read 2
                                        #bytes) and assign its value to the
                                        #variable 'valor'
send_message info $valor#Send an info message to the console
                        #with the value read from the component
master_write_8 $jtag_master 0x1_0040 $valor #Write to the LEDs memory address
                                        #(LEDs just take up 1 byte,
                                        #so the highest byte of the
                                        #component value will be truncated
close_service master $jtag_master #Close the path
```

Before executing System Console, we must ensure the SoCkit board is turned on, connected to the host and the FPGA has been programmed. As the peripheral to be used such as LEDs are peripherals of the FPGA, they should be configured. **To program the FPGA, follow the steps of "Appendix C: Programming the FPGA".**

Now, we get ready to run the System Console. **Select "System Console" from the Tools menu in Qsys.** In Figure 48 it is shown the initial screen of System Console. In the Messages window we should view the connection with the SoCkit established, a JTAG connection (message "Auto linking... SoCkit to soc_system.qsf").



```
Finished initialization
Finished discovering JTAG connections
Auto linking 5CS(EBA6ES|XFC6C6ES)|..@1#1-2#CV SoCKit to soc_system.qsf
Finished discovering USB connections
Executing startup script /home/marcos/altera/13.0sp1/quartus/sopc_builder...
```

```
    * To start, stop, or step a Nios II processor
    * To read or write Avalon Memory-Mapped (Avalon-MM) slaves using
special
     masters
    * To sample the SOPC system clock as well as system reset signal
    * To run JTAG loopback tests to analyze board noise problems
    * To shift arbitrary instruction register and data register
values to
     instantiated system level debug (SLD) nodes

 In addition, the directory <QuartusII
```

Fig. 48: Initial screen of System Console

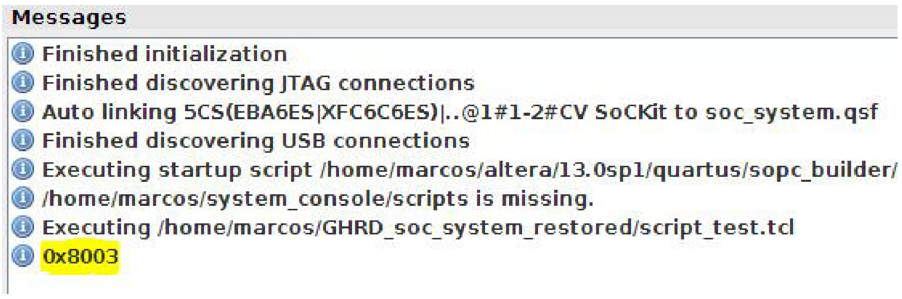
Once the program is ready, we are prepared to execute the script that we created. **In the File tab, select "Execute Script...". A window will appear to create a scripts folder. Select "Don't create".**

Another window will appear. **Search for our script**, which we named "script_test.tcl" and was saved in the "GHRD_soc_system " folder. **Select this script to execute it**.

Taking into account that we coded the script to write a '0x0002' to the memory address of our component, which value we should get returned when we read from the same address after writing? In the VHDL code of our component, we take the written value and apply the binary OR operation with 0x8001. Hence, we should read then:

```
0x0002 OR 0x8001 → 0x8003
```

If we take a look to the messages windows in System Console, where the read value from the address of our component will be shown, we see it after executing the script, as in Figure 49.



```
Messages
Finished initialization
Finished discovering JTAG connections
Auto linking 5CS(EBA6ES|XFC6C6ES)|..@1#1-2#CV SoCKit to soc_system.qsf
Finished discovering USB connections
Executing startup script /home/marcos/altera/13.0sp1/quartus/sopc_builder/
/home/marcos/system_console/scripts is missing.
Executing /home/marcos/GHRD_soc_system_restored/script_test.tcl
0x8003
```

Fig. 49: Read value from the address of the new component

Our new component works correctly! In fact, we will check twice if it works because, as we said, the read value from our component is written to the four FPGA LEDs address. The four LEDs must display in binary the number we view in the messages window in System Console. Anyway, as they are only four LEDs they will show just the four least significant bits of the written value (if we write 0x0F all LEDs will be lighted; if we write 0x10 no LEDs will be lighted).

Note that if we do a simple test, which consist of changing the memory address where we write data (for example, 0x1_0060, instead of our component address, 0x1_0050), we can observe when reading from the address that the value has not been modified.

We have just achieved one of the main objectives of this project: a new peripheral developed from scratch added to the system and prepared to be used.

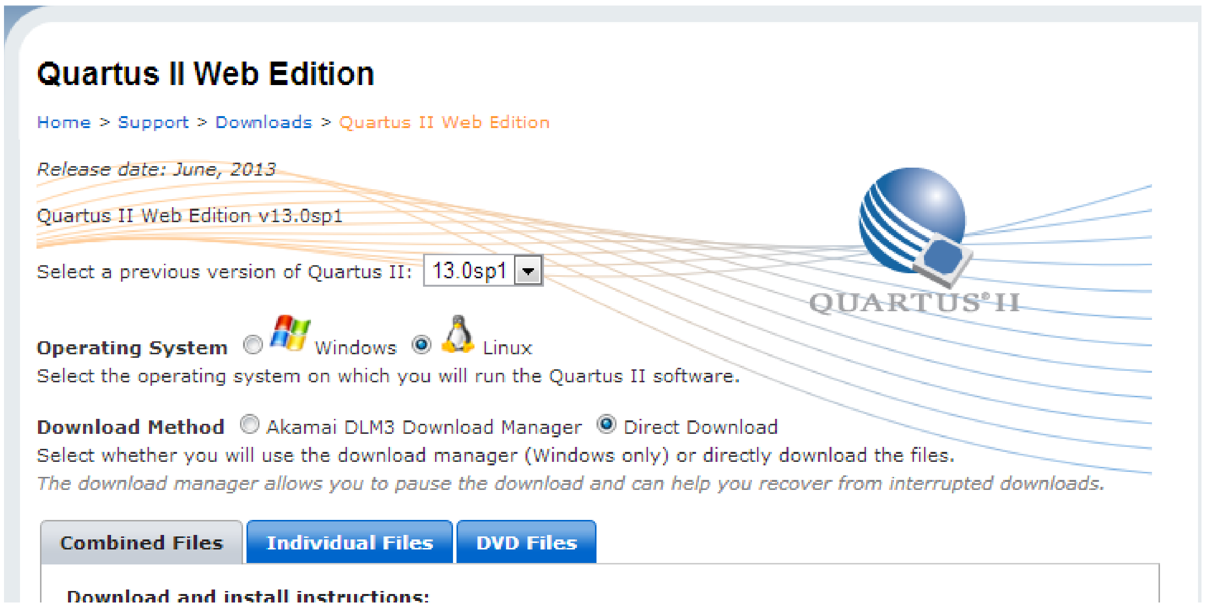
8. Software development

Now that we configured the hardware system of the SoCkit board, compiled the project and generated all the files needed to create the components of the Linux distribution, we are ready to confront our next aim: generate a Linux distribution for the SoCkit board.

8.1. Downloading and installing SoC EDS

First of all, we need to **install the Altera tool set SoC EDS**, as it provides some useful tools like "mkpimage", which we will use to generate the SD card image from the image files generated by Yocto Source Package.

Just as at the beginning of the section "7. Hardware Development" it was facilitated the link of the webpage for downloading Quartus II tool, now we have to access again for downloading SoC EDS. The webpage should look as in Figure 9; we show it here again for the reader to be comfortable.



Quartus II Web Edition

Home > Support > Downloads > Quartus II Web Edition

Release date: June, 2013

Quartus II Web Edition v13.0sp1

Select a previous version of Quartus II: 13.0sp1

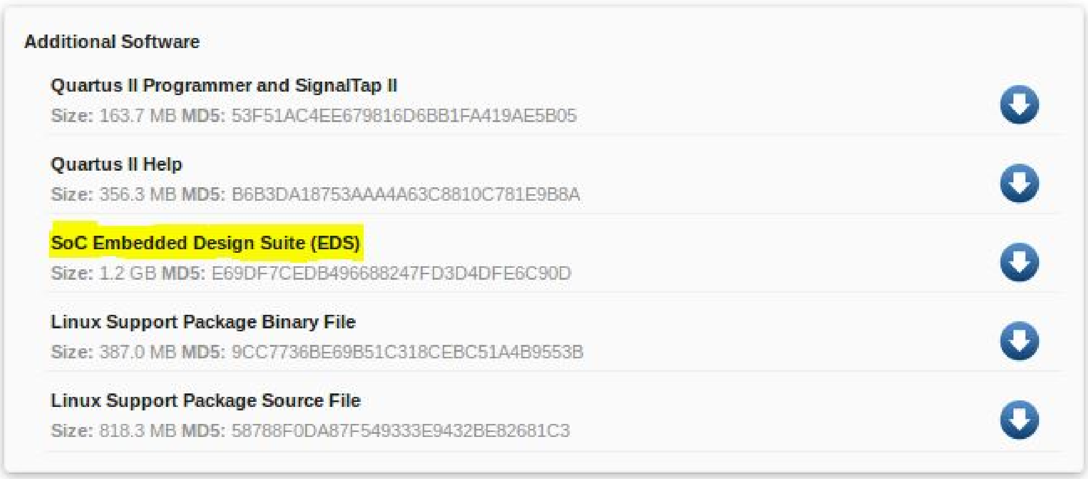
Operating System Windows [✓ Linux]
Select the operating system on which you will run the Quartus II software.

Download Method Akamai DLM3 Download Manager [✓ Direct Download]
Select whether you will use the download manager (Windows only) or directly download the files.
The download manager allows you to pause the download and can help you recover from interrupted downloads.

Download and install instructions:

Fig. 9: Quartus II download page

In the "Individual Files" tab, go to the "Additional Software" section, then download the file for "SoC Embedded Design Suite (EDS)", as seen in Figure 50.


Additional Software

Quartus II Programmer and SignalTap II
Size: 163.7 MB **MD5:** 53F51AC4EE679816D6BB1FA419AE5B05

Quartus II Help
Size: 356.3 MB **MD5:** B6B3DA18753AAA4A63C8810C781E9B8A

SoC Embedded Design Suite (EDS)
Size: 1.2 GB **MD5:** E69DF7CEDB496688247FD3D4DFE6C90D

Linux Support Package Binary File
Size: 387.0 MB **MD5:** 9CC7736BE69B51C318CEBC51A4B9553B

Linux Support Package Source File
Size: 818.3 MB **MD5:** 58788F0DA87F549333E9432BE82681C3

Fig. 50: Downloading SoC EDS

When the download completes, in the same way we did with Quartus II, **right click on the installation file and go to Properties. In Permissions, select the box "Allow executing file as a program".** Otherwise, the Ubuntu OS will try to open the file instead of executing it, so it will not find any application to open it. After that, to begin the installation, we have to run the file as root; if not, a message will appear indicating that there were troubles on installation (for example, the uninstaller file was not be correctly created). Thus, though these errors may not have a negative impact on all tools utilization, we will execute it as root to make sure that everything is installed correctly. **Open a terminal on Ubuntu, navigate to the directory where the file is (it will be on "/home/<user>/Downloads" by default) and execute the following command:**

```
$ sudo ./SoCEDSSetup-13.0.1.232.run
```

When the installation windows appears, **accept the license agreement and set as installation folder the one by default, "/home/<user>/altera/13.0sp1".** Software will be installed so we can now use its tools.

8.2. Generation of a preloader and a bootloader with the Altera software

The first step of the software development (and the third of this project) is to create, using the Altera software, the necessary files for the first phases of the system boot flow: the preloader and the bootloader.

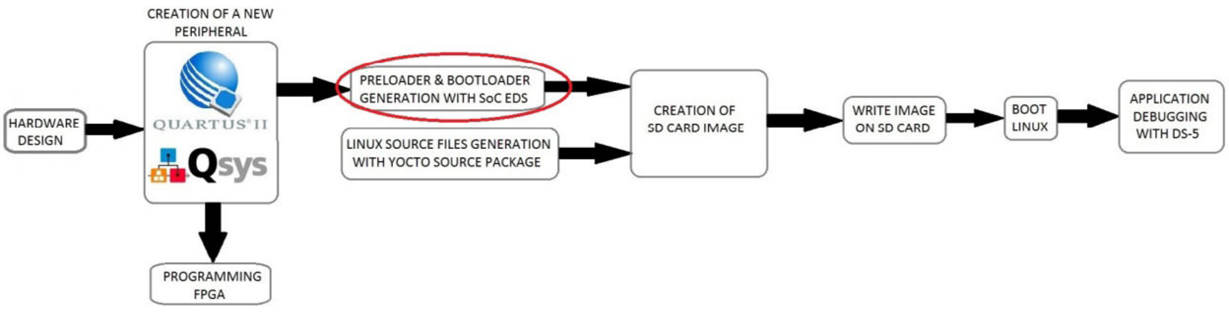
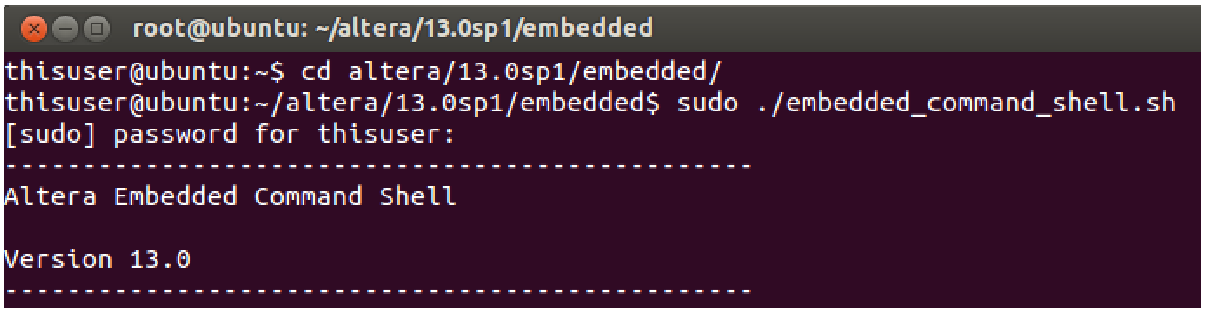

CREATION OF A NEW PERIPHERAL
HARDWARE DESIGN
QUARTUS II
Qsys
PROGRAMMING FPGA
PRELOADER & BOOTLOADER GENERATION WITH SoC EDS
LINUX SOURCE FILES GENERATION WITH YOCTO SOURCE PACKAGE
CREATION OF SD CARD IMAGE
WRITE IMAGE ON SD CARD
BOOT LINUX
APPLICATION DEBUGGING WITH DS-5

Fig. 51: Third step. Creating a preloader and a bootloader

We will use a tool named BSP[7] Editor, that sets up the configuration to create the preloader and the bootloader from the handoff files generated by the Altera tools responsible for configuring the system hardware (FPGA and HPS), Quartus II and Qsys. **To execute the program, we have to access from an Ubuntu terminal to the "Altera Embedded Command Shell", as shown in Figure 52.**

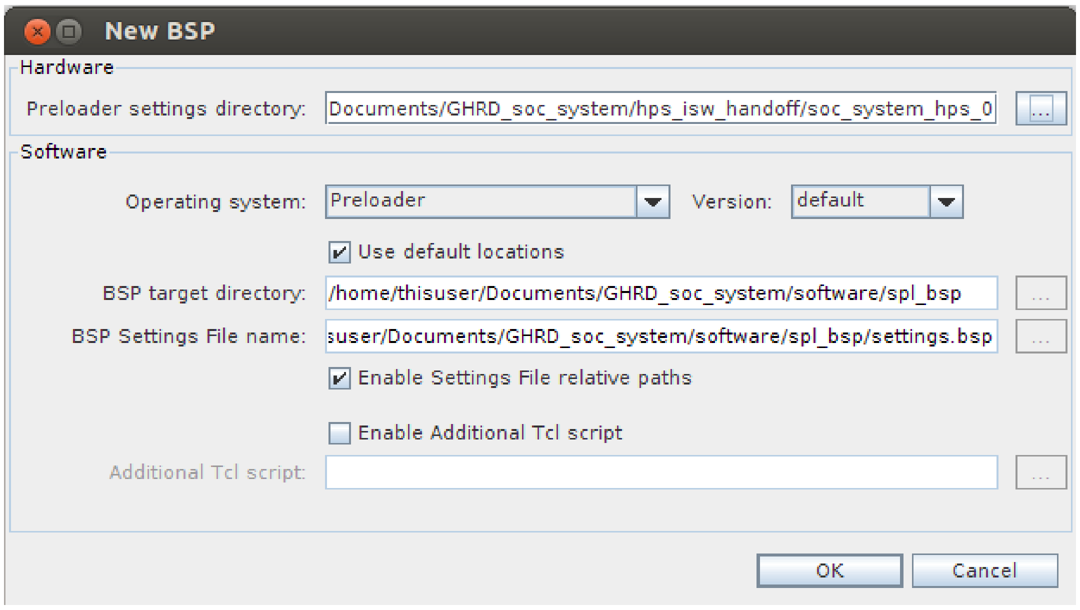


```
root@ubuntu: ~/altera/13.0sp1/embedded
thisuser@ubuntu:~$ cd altera/13.0sp1/embedded/
thisuser@ubuntu:~/altera/13.0sp1/embedded$ sudo ./embedded_command_shell.sh
[sudo] password for thisuser:
------------------------------------------------
Altera Embedded Command Shell

Version 13.0
------------------------------------------------
```

Fig. 52: Embedded Command Shell

Type the command "bsp-editor" on the prompt. The initial screen of BSP Editor will appear. **Select "New BSP..." in the File tab to create a new configuration. In a smaller window that will pop up, click the button to the right of "Preloader settings directory" to browse the file system. Select the "/home/<user>/GHRD_soc_system /hps_isw_handoff/soc_system_hps_0" folder.** The window should look as in Figure 53.

[7] BSP is the acronym for Binary Space Partitioning


New BSP

Hardware

Preloader settings directory: Documents/GHRD_soc_system/hps_isw_handoff/soc_system_hps_0

Software

Operating system: Preloader   Version: default

[✓] Use default locations

BSP target directory: /home/thisuser/Documents/GHRD_soc_system/software/spl_bsp

BSP Settings File name: suser/Documents/GHRD_soc_system/software/spl_bsp/settings.bsp

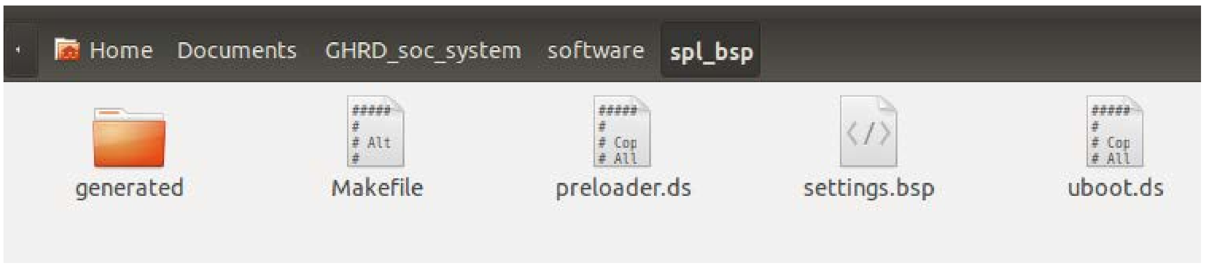
[✓] Enable Settings File relative paths

[ ] Enable Additional Tcl script

Additional Tcl script:

Fig. 53: BSP editor new configuration

Click OK, then the "settings.bsp" file will be created in the directory set in the New BSP window (Figure 42) as "BSP target directory". **Next, click on Generate** so it will create all files needed to compile and generate the preloader and bootloader (Figure 54).

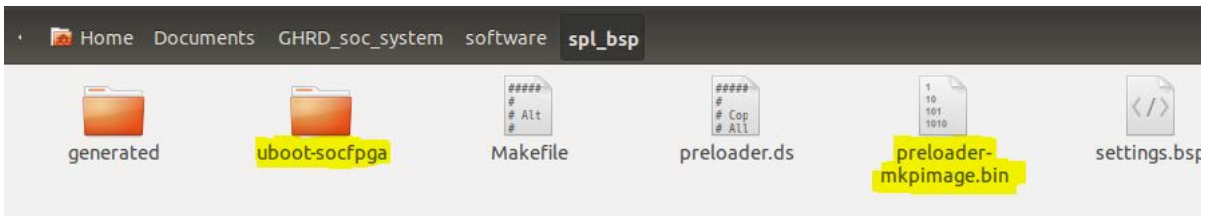

Home › Documents › GHRD_soc_system › software › spl_bsp

generated Makefile preloader.ds settings.bsp uboot.ds

Fig. 54: Files needed to generate preloader and bootloader

To this point, everything is ready to generate the preloader and the bootloader. **The last thing to do is typing "make" on the Embedded Command Shell prompt (being situated in the ".../GHRD_soc_system /software/spl_bsp" directory) to generate the preloader, and "make uboot" to generate the bootloader.** The compilation may take several minutes.

When compilation is finished, note that the binary file of preloader has been created in the "spl_bsp" directory, named as "preloader-mkpimage.bin". A new folder has also been created, "uboot-socfpga", which contains a lot of files including the image file of U-boot (Figure 55 and Figure 56).


Home › Documents › GHRD_soc_system › software › spl_bsp

generated uboot-socfpga Makefile preloader.ds preloader-mkpimage.bin settings.bsp

Fig. 55: Generated preloader file

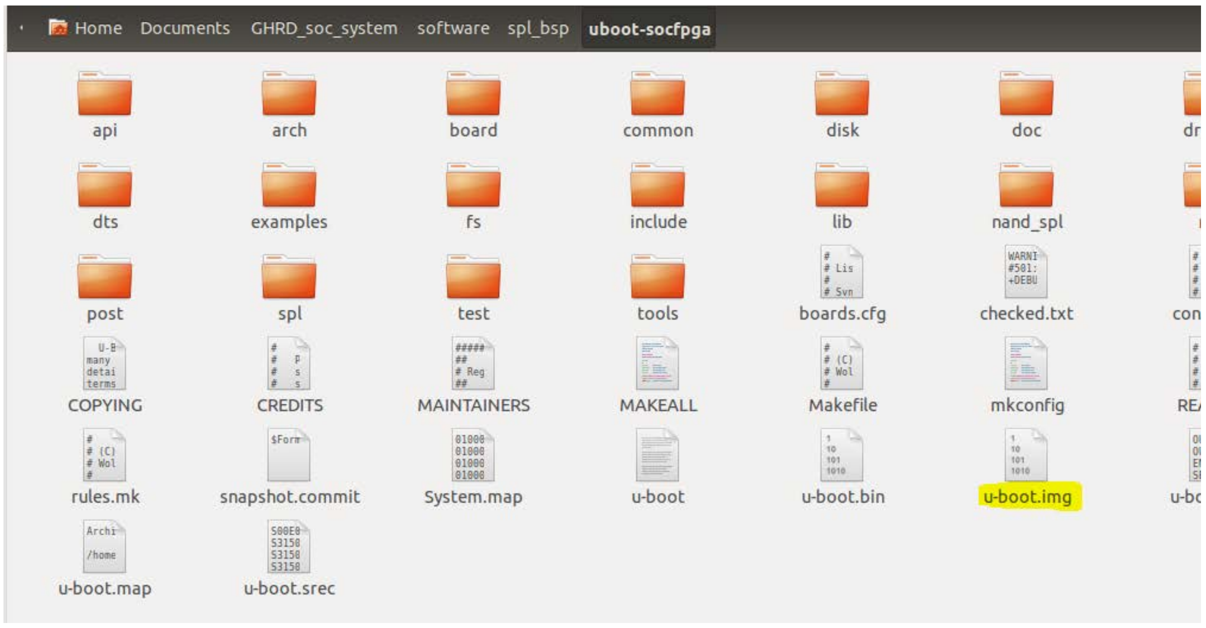
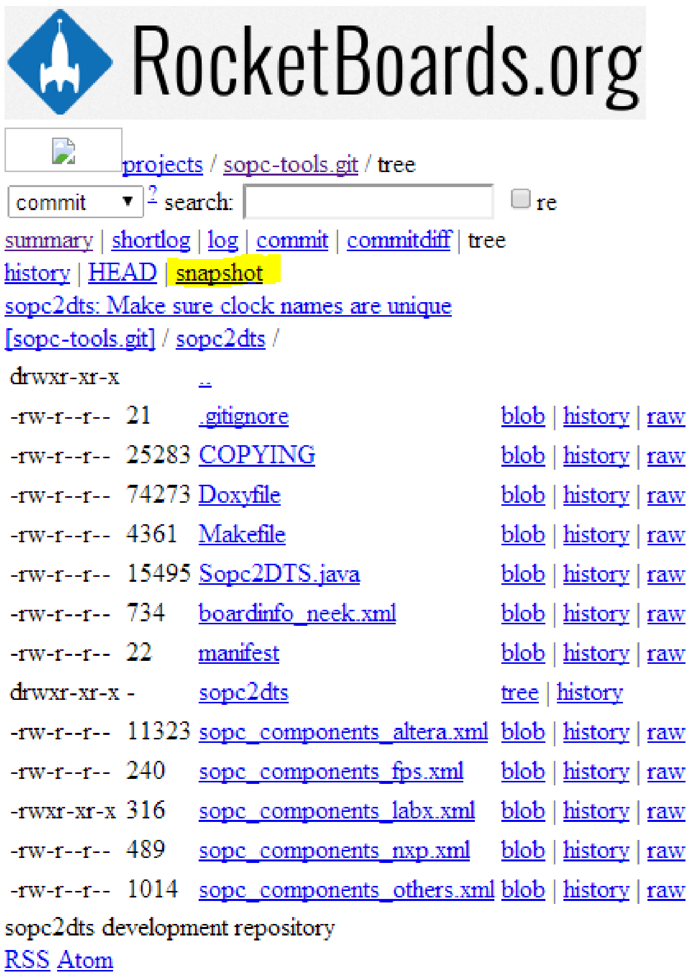


Fig. 56: Generated bootloader file

In order to generate the DTB **we need to download "sopc2dts",** a java tool programmed to generate the DTS (Device Tree Source) from the "soc_system.sopcinfo" file generated with Qsys. This tool also needs two XML (eXtensible Markup Language) files that contain information about the board and the clock tree.

To download this tool, **go to the developers web RocketBoards. In the download page[d5], click on "snapshot" as seen in Figure 57.** This will download all files in a compressed file.



RocketBoards.org

projects / sopc-tools.git / tree

commit ▾ ? search: re

summary | shortlog | log | commit | commitdiff | tree
history | HEAD | snapshot

sopc2dts: Make sure clock names are unique

[sopc-tools.git] / sopc2dts /

drwxr-xr-x		..	
-rw-r--r--	21	.gitignore	blob | history | raw
-rw-r--r--	25283	COPYING	blob | history | raw
-rw-r--r--	74273	Doxyfile	blob | history | raw
-rw-r--r--	4361	Makefile	blob | history | raw
-rw-r--r--	15495	Sopc2DTS.java	blob | history | raw
-rw-r--r--	734	boardinfo_neek.xml	blob | history | raw
-rw-r--r--	22	manifest	blob | history | raw
drwxr-xr-x	-	sopc2dts	tree | history
-rw-r--r--	11323	sopc_components_altera.xml	blob | history | raw
-rw-r--r--	240	sopc_components_fps.xml	blob | history | raw
-rwxr-xr-x	316	sopc_components_labx.xml	blob | history | raw
-rw-r--r--	489	sopc_components_nxp.xml	blob | history | raw
-rw-r--r--	1014	sopc_components_others.xml	blob | history | raw

sopc2dts development repository

RSS Atom

Fig. 57: Downloading sopc2dts tool

Extract the files from the downloaded compressed file. Within a terminal, navigate to the "sopc2dts" folder (in our case, "/home/<user>/Downloads/sopc-tools-733dcf5") and type "make"; this way, the java files contained in this directory will be compiled and a file named "sopc2dts.jar" will be generated. This file is used to generate the DTS.

If "make" command fails and the message "make: javac: command not found" is displayed in the command prompt, just download the Java Development Kit since Ubuntu does not include it by default. To download and install it, just type the following command:

```
~$ sudo apt-get install openjdk-6-jdk
```

Next, we have to **download two XML files needed to generate the DTS** together with the .sopcinfo file. **In the RocketBoards webpage[d6] scroll down to the bottom of the webpage. There are two links, "clock file" and "board info file".** If you click on any link, you will view the XML code of it. To download them, **right click on them and select "Save link as...". Save both files in the "sopc-tools-733dcf5" folder, named as "hps_clock_info.xml" and "soc_system_board_info.xml".**

Moreover, **copy the .sopcinfo file from the Quartus II project directory ("/home/<user>/GHRD_soc_system ") to the " sopc-tools-733dcf5" folder**. **To generate**

the DTS, navigate to the "sopc2dts" directory within a terminal and **use the following command:**

```
~$ java -jar sopc2dts.jar -i soc_system.sopcinfo -b
hps_clock_info.xml -b soc_system_board_info.xml -o socfpga.dts
```

Do not worry if a "component reg16_0 of class reg16 is unknown" warning message is displayed, it is just our new custom component, the program do not recognize it since we tagged its type as "Other" in Qsys.

Now that we have created the DTS file, we need to convert it to the DTB binary file. To do that, access to an Altera Embedded Command Shell from an Ubuntu terminal (as in Figure 52), then execute the following command:

```
~$ dtc -I dts -O dtb -o socfpga.dtb socfpga.dts
```

"dtc" is a command which uses the Altera Device Tree Compiler. The command options are:

- -I: Input format.
- -O: Output format.
- -o: Name of the output file (followed by the DTS input file).

So far, we have generated the preloader, the bootloader and the DTB. We will generate the rest of files in the next sections.

8.3. Installing Yocto software and creating the SD card image file

The Yocto Source Package is an installation file recommended by Altera that contains the necessary dependencies to compile the files that compose the Linux distribution. We will use it to get the rest of Linux parts that we could not generate with Altera tools.

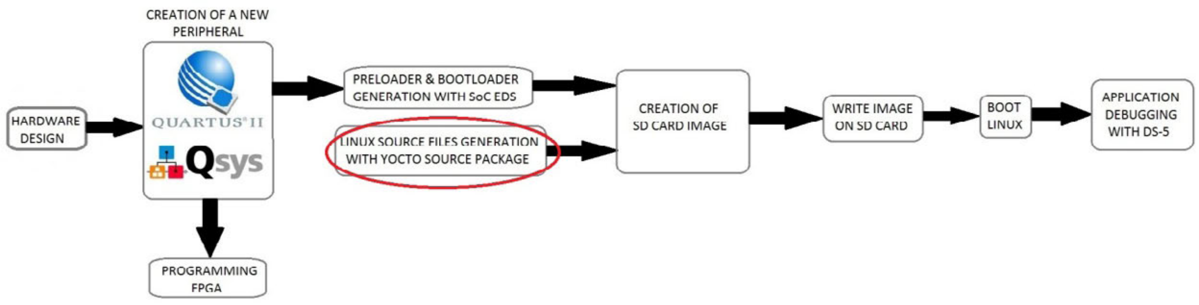

CREATION OF A NEW PERIPHERAL
HARDWARE DESIGN
QUARTUS II
Qsys
PROGRAMMING FPGA
PRELOADER & BOOTLOADER GENERATION WITH SoC EDS
LINUX SOURCE FILES GENERATION WITH YOCTO SOURCE PACKAGE
CREATION OF SD CARD IMAGE
WRITE IMAGE ON SD CARD
BOOT LINUX
APPLICATION DEBUGGING WITH DS-5

Fig. 58: Fourth step. Generating Linux files with YSP

We have to install some required packages before proceeding:

```
$ sudo apt-get update
$ sudo apt-get upgrade
$  sudo apt -get  install  sed  wget  cvs  subversion git -core  coreutils
unzip  texi2html  t exinfo  libsdl1.2-dev  docbook-utils  gawk python -
pysqlite2  diffstat  help2man make  gcc  build-essential  g++ desktop -
file-utils   chrpath libgl1  -mesa-dev   libglu1-mesa-dev   mercurial
autoconf automake groff libtool xterm
```

8.3.1. Downloading and installing YSP

The YSP can be downloaded from the Altera website. Type the following commands to download it (first command) and mark it as executable (second command):

```
$ wget
http://download.altera.com/akdlm/software/acdsinst/13.0/156/ib_inst
allers/linux-socfpga-13.02-RC10-src.bsx
$ chmod +x linux-socfpga-13.02-RC10-src.bsx
```

The installation progress will be made in "/opt/altera-linux" directory by default. We will install it there, but it can be installed on a different location. To use this directory, root access is needed (that is why the following command to install it is executed using "sudo").

```
$ sudo ./linux-socfpga-13.02-RC10-src.bsx
```

The next step is to **set a directory for all Yocto files. The default directory can be created within the "home" user directory. Use the following command to set the Yocto directory as "/home/<user>/yocto":**

```
$ /opt/altera-linux/bin/install_altera_socfpga_src.sh  ~/yocto
```

The last step is to **create a "build" directory**; in a subfolder of it there are the generated image files used to create the SD card image.

```
$ source ~/yocto/altera-init  ~/yocto/build
```

In order to see the entire process, please refer to the Rocketboards.org page "Linux Getting Started on Altera SoC Development Board - Using Yocto Source Package" in the bibliography.

8.3.2. Compiling u-boot, kernel and file system

Before compiling and generating the image file of U-boot, the Linux kernel and the file system from the build directory, we need to execute the following command:

```
$ sudo chmod -R 777 ~/yocto
```

To compile and generate the images, we need write permissions in all folders within the Yocto directory. We change the permissions with the command above (the -R is for changing permissions of all subfolders of Yocto directory).

To compile and generate the image file of U-boot, execute the following command:

```
$ bitbake virtual/bootloader
```

Or the equivalent:

```
$ bitbake u-boot
```

To generate the Linux kernel:

```
$ bitbake virtual/kernel
```

Or the equivalent:

```
$ bitbake linux-altera
```

To generate the file system:

```
$ bitbake altera-image
```

There is a command to generate a much smaller file system for reduced boot mediums. Anyway, in this project we will use the complete file system because we may use some resources that the minimal file system does not include.

The first time the YSP is generated it can take few hours, depending on the power of the computer used. Once finished, all image files should have been generated in the "/home/<user>/yocto/build/tmp/deploy/images" directory. Among all generated files, the following in Table 4 will be used to create the SD card image file.

Table 4: Files used to create the SD card image

File	Description
u-boot-spl-socfpga_cyclone5.bin	Preloader image
u-boot-socfpga_cyclone5.img	U-boot image
socfpga_cyclone5.dtb	DTB
uImage	Kernel image
altera-image-socfpga_cyclone5.tar.gz	Compressed file system

8.3.3. Generating SD card image. Writing image on SD card

We have everything that we need to install the Linux distribution for the SoCkit board. The first step from now on is to create the image file that we will write to the SD card.

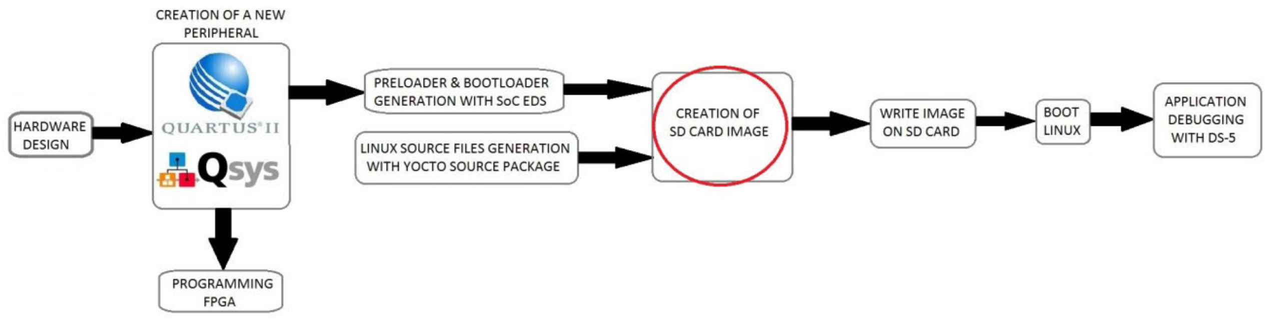

CREATION OF A NEW PERIPHERAL
HARDWARE DESIGN
QUARTUS II
Qsys
PROGRAMMING FPGA
PRELOADER & BOOTLOADER GENERATION WITH SoC EDS
LINUX SOURCE FILES GENERATION WITH YOCTO SOURCE PACKAGE
CREATION OF SD CARD IMAGE
WRITE IMAGE ON SD CARD
BOOT LINUX
APPLICATION DEBUGGING WITH DS-5

Fig. 59: Fifth step. Creating the SD card image file

We will use the Yocto files, including preloader, bootloader and DTB, to see what happen when we boot Linux. **The following commands are used to create the SD-card image :**

```
$ cp -f socfpga_cyclone5.dtb socfpga.dtb
$ mkdir rootfs
$ cd rootfs
$ sudo tar xzf ../altera-image-socfpga_cyclone5.tar.gz
$ cd ..
$ sudo /opt/altera-linux/bin/make_sdimage.sh \
  -k uImage,socfpga.dtb\
  -rp u-boot-spl-socfpga_cyclone5.bin \
  -t /home/<user>/altera/13.0sp1/embedded/host_tools/altera/mkpimage/mkpimage \
  -b u-boot-socfpga_cyclone5.img \
  -r rootfs/ \
  -o sd_image_yocto.bin
```

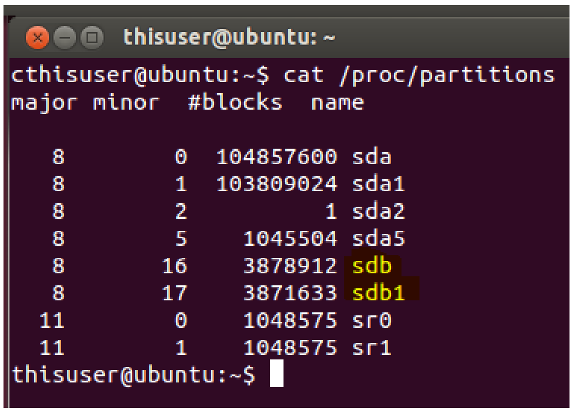
Of course, within the Ubuntu terminal we have to be situated in the "/home/<user>/yocto/build/tmp/deploy/images" directory, where all these files are. These commands:

- Copy the DTB "socfpga_cyclone5.dtb" and rename it as "socfpga.dtb" since that is the name U-boot knows.
- Extract the file system to the "rootfs" folder.
- Invokes the "make_sdimage.sh" script to create the SD card image, named as "sd_image_yocto.bin". "sudo" is used because some system utilities are being used.

Before writing the image file to the SD card, we have to **find out which device is assigned to the card on the host** (for example, "/dev/sdb"). This can be done **with the following command:**

```
$ cat /proc/partitions
```

A list will be displayed as in Figure 60. "sdb" is the name of the device assigned to the SD card. As we use a formatted SD card, we can view in the list just one partition of it, named as "sdb1" (you can see in Figure 60 that "sdb" and "sdb1" have the same capacity).



```
cthisuser@ubuntu:~$ cat /proc/partitions
major minor  #blocks  name

   8        0  104857600 sda
   8        1  103809024 sda1
   8        2          1 sda2
   8        5    1045504 sda5
   8       16    3878912 sdb
   8       17    3871633 sdb1
  11        0    1048575 sr0
  11        1    1048575 sr1
thisuser@ubuntu:~$ 
```

Fig. 60: Display of the SD partitions in the terminal

After writing the Linux image into the SD card, we should view three partitions of the "sdb" device: "sdb1", "sdb2" and "sdb3". The first partition contains the Linux kernel image file, the FPGA configuration file, the U-boot script for configuring the FPGA and the DTB file. The second partition contains the file system. The last partition contains the preloader and the u-boot images. See the layout of the SD card in Figure 61 (next page).

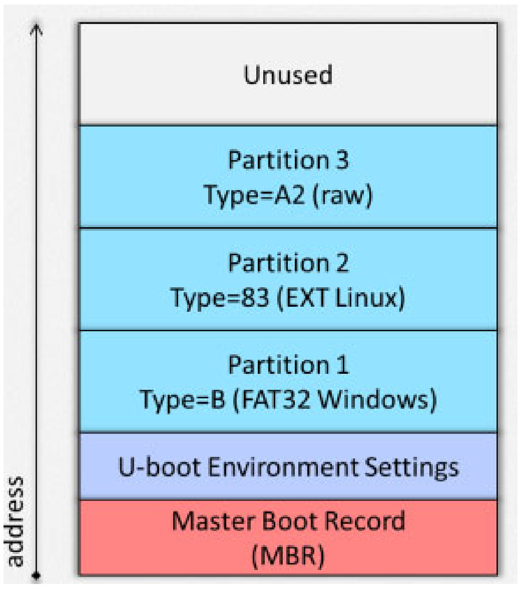


Fig. 61: Partitions of the SD card

Now that we have the image file created and we know the name of the device assigned to the SD card, we will write the image file to the SD card.

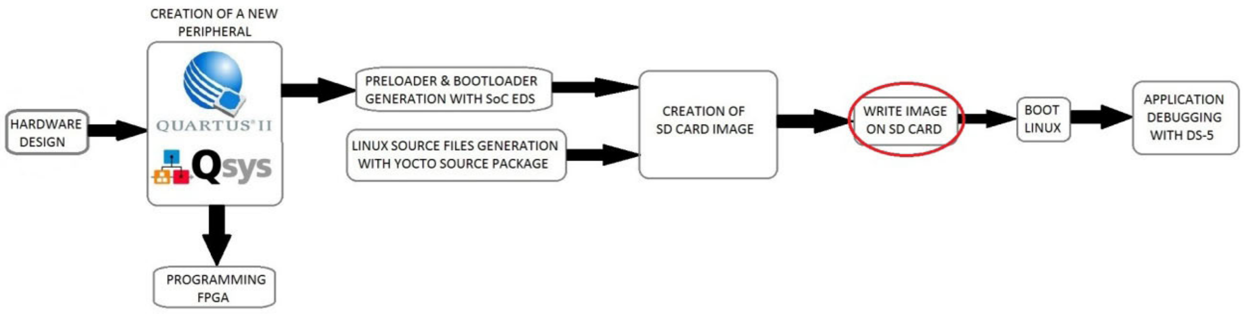

CREATION OF A NEW PERIPHERAL
HARDWARE DESIGN
QUARTUS II
Qsys
PROGRAMMING FPGA
PRELOADER & BOOTLOADER GENERATION WITH SoC EDS
LINUX SOURCE FILES GENERATION WITH YOCTO SOURCE PACKAGE
CREATION OF SD CARD IMAGE
WRITE IMAGE ON SD CARD
BOOT LINUX
APPLICATION DEBUGGING WITH DS-5

Fig. 62: Sixth step. Writing image file to SD card

There is the command "dd" for writing the image file to the SD card. Provided that the device assigned to the card is "/dev/sdb", the command would be:

```
$   sudo dd if=   ~/yocto/build/tmp/deploy/images/sd_image_yocto.bin
of=/dev/sdb bs=512
```

This operation will take a while depending on the performance of the computer where executed (in our case, about twenty minutes). Do not worry if nothing is displayed in the command prompt, it will do when operation is finished. **Finally, process the changes made in the card with the following command:**

```
$ sudo sync
```

After these operations, our image is written on the SD card, ready to be inserted into the SoCkit slot in order to boot the Linux OS. We will do it in the next section, "8.4. Booting Linux on the SoCkit board".

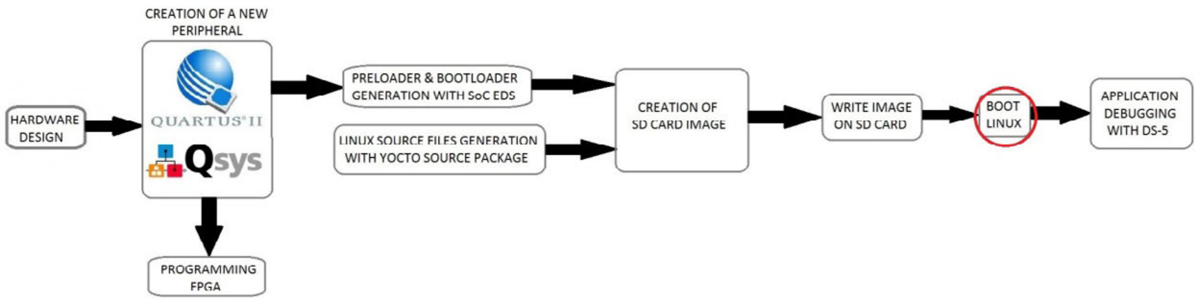

CREATION OF A NEW PERIPHERAL
HARDWARE DESIGN
QUARTUS II
Qsys
PROGRAMMING FPGA
PRELOADER & BOOTLOADER GENERATION WITH SoC EDS
LINUX SOURCE FILES GENERATION WITH YOCTO SOURCE PACKAGE
CREATION OF SD CARD IMAGE
WRITE IMAGE ON SD CARD
BOOT LINUX
APPLICATION DEBUGGING WITH DS-5

Fig. 63: Seventh step. Booting Linux on the SoCkit board

8.4. Booting Linux on the SoCkit board

8.4.1. Troubles with Yocto files and solution

After installing the software, generating necessary files to create the SD card image file, making an image out of these files and writing it to the SD card, we have to run Linux on the SoCkit board. To do that, **insert the SD card into the SoCkit slot and follow the steps of Appendix D.**

Press the "warm reset" to reboot the system and note that the screen of the PuTTY terminal remains black; the Linux OS does not boot. What is happening? Did we do some step incorrectly?

The explanation of this problem and its solution is simple; the cause is that the generated images are adapted by Yocto bitbake scripts for another development kit: the "SoC Development Board" made by Altera (remember that the SoCkit board was developed by Arrow).

The preloader generated by YSP needs to take into account the hardware features of the board, such as pins, to perform its configuration function; it is prepared and coded for the hardware configuration of this board. Therefore, when trying to boot Linux on the SoCkit, this preloader is not able to work because both boards are different in so many aspects.

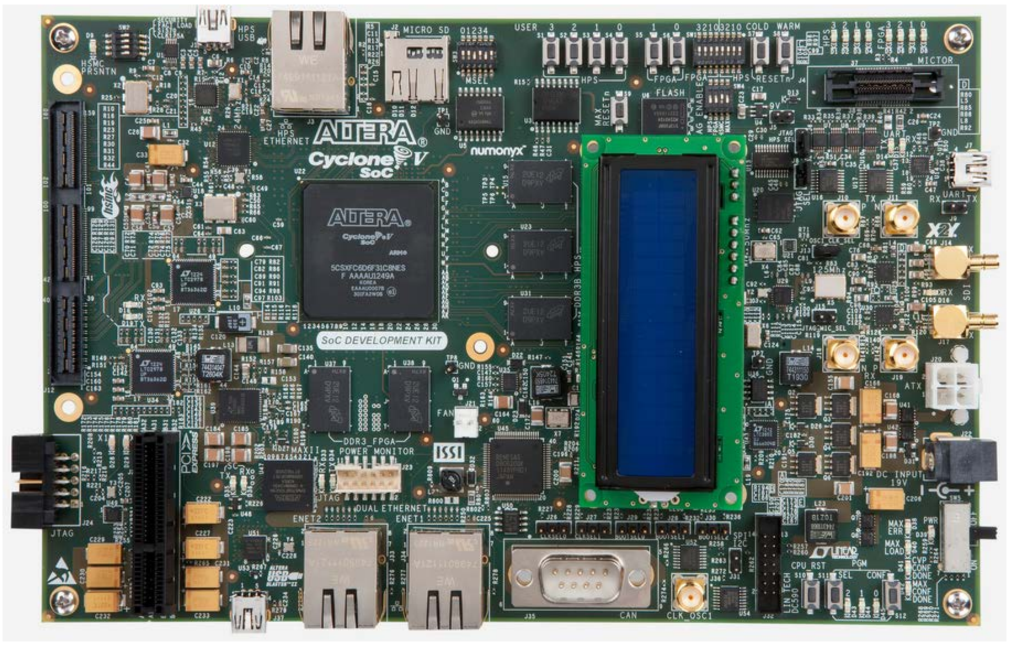


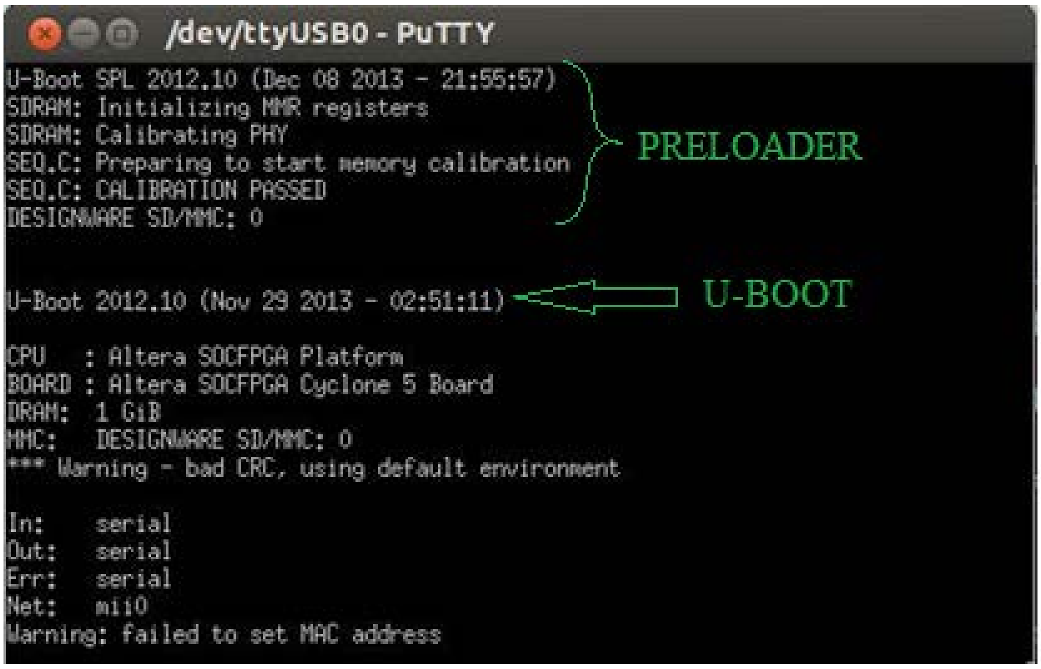
Fig. 64: Altera's SoC Development Board

The solution to this problem is easy: we should use a preloader generated specifically for the SoCkit board. Where can we get it? The answer is the section "7.6. Generation of a preloader and a bootloader with the Altera software for Linux"; we already got it. We have an own preloader generated with Altera tools.

We have to substitute the Yocto preloader written to the SD card with the preloader generated by Altera tools. To do that, it is not necessary to generate a whole SD card image and write it again, which would take a while. **There are some commands to update individual parts of the image in the card. Supposing the device assigned to the card is "/dev/sdb", the command to change the preloader would be the following:**

```
$ sudo dd if=preloader-mkpimage.bin of=/dev/sdb3 bs=64k seek=0
```

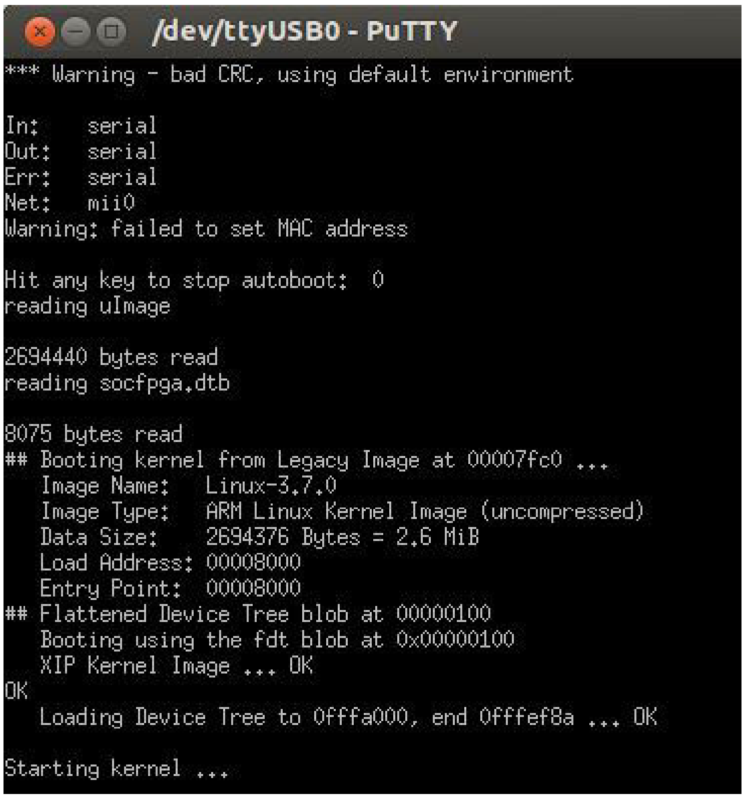
After doing this, insert again the card into the SD slot of the SoCkit, establish again a connection through a PuTTY terminal and press the warm reset button to reboot Linux. **Just after pressing the warm reset button, we can observe on the console the preloader and then U-boot loading, as seen in Figure 65.**



```
/dev/ttyUSB0 - PuTTY
U-Boot SPL 2012.10 (Dec 08 2013 - 21:55:57)
SDRAM: Initializing MMR registers
SDRAM: Calibrating PHY
SEQ.C: Preparing to start memory calibration
SEQ.C: CALIBRATION PASSED
DESIGNWARE SD/MMC: 0

U-Boot 2012.10 (Nov 29 2013 - 02:51:11)

CPU   : Altera SOCFPGA Platform
BOARD : Altera SOCFPGA Cyclone 5 Board
DRAM:  1 GiB
MMC:   DESIGNWARE SD/MMC: 0
*** Warning - bad CRC, using default environment

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

Fig. 65: Preloader and U-boot loading

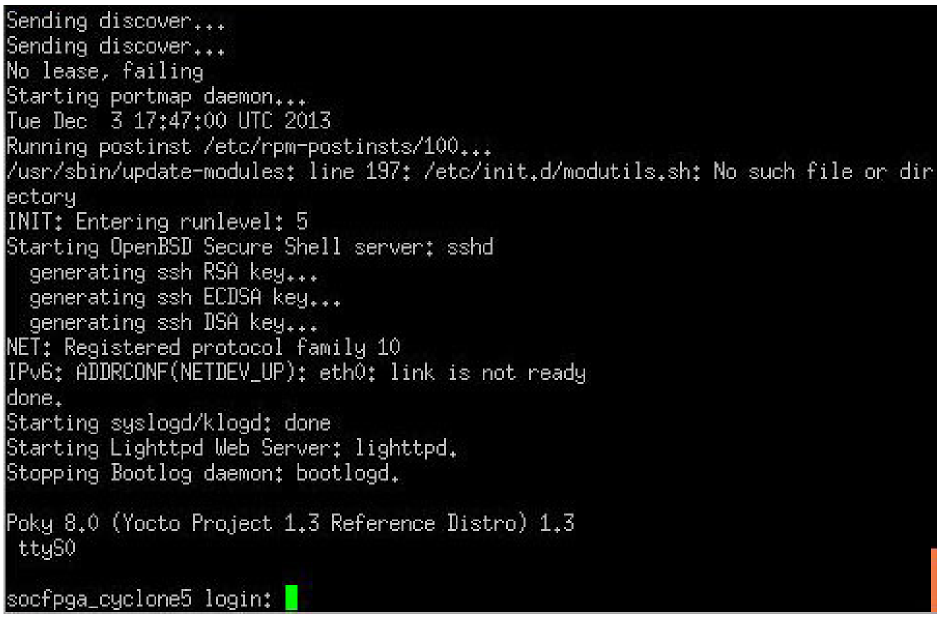
Next, the system loads the kernel image. The kernel is given the information of the DTB so that it can register the system devices. **Once DTB is loaded, the kernel begins to run the OS, just as in Figure 66.**



```
/dev/ttyUSB0 - PuTTY
*** Warning - bad CRC, using default environment

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

Hit any key to stop autoboot:  0
reading uImage

2694440 bytes read
reading socfpga.dtb

8075 bytes read
## Booting kernel from Legacy Image at 00007fc0 ...
   Image Name:   Linux-3.7.0
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2694376 Bytes = 2.6 MiB
   Load Address: 00008000
   Entry Point:  00008000
## Flattened Device Tree blob at 00000100
   Booting using the fdt blob at 0x00000100
   XIP Kernel Image ... OK
OK
   Loading Device Tree to 0fffa000, end 0fffef8a ... OK

Starting kernel ...
```

Fig. 66: Loading the kernel and DTB

After the message "Starting Kernel..." the console will display hundreds of lines as the whole system is being loaded. When it finishes, the system will be waiting us to type our user name. Type "root".



```
Sending discover...
Sending discover...
No lease, failing
Starting portmap daemon...
Tue Dec  3 17:47:00 UTC 2013
Running postinst /etc/rpm-postinsts/100...
/usr/sbin/update-modules: line 197: /etc/init.d/modutils.sh: No such file or dir
ectory
INIT: Entering runlevel: 5
Starting OpenBSD Secure Shell server: sshd
  generating ssh RSA key...
  generating ssh ECDSA key...
  generating ssh DSA key...
NET: Registered protocol family 10
IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
done.
Starting syslogd/klogd: done
Starting Lighttpd Web Server: lighttpd.
Stopping Bootlog daemon: bootlogd.

Poky 8.0 (Yocto Project 1.3 Reference Distro) 1.3
 ttyS0

socfpga_cyclone5 login: 
```

Fig. 67: Linux OS loaded

We made it; we achieved the main objective of this project: to generate an own Linux distribution able to load correctly.

We can change the bootloader in the SD, which is the one generated with the YSP, for the bootloader generated with the Altera software. **To do that[8], use the following command**:

```
$ sudo dd if=u-boot.img of=/dev/sdb3 bs=64k seek=4
```

After reinserting the SD into the SoCkit board and booting Linux, we checked that the system still boots correctly.

Then we may change the DTB in the SD for the DTB generated with the "sopc2dts" tool, **using the following commands**:

```
$ sudo mkdir sdcard
$ sudo mount /dev/sdb1 sdcard/
$ sudo cp socfpga.dtb sdcard/socfpga.dtb (being situated in the
folder where our DTB is contained)
$ sudo umount sdcard/
```

We tried to boot Linux one more time, but after the kernel is loaded into memory, the system is not able to go on with the boot flow; it hangs after the message "Starting kernel..." is displayed in the terminal. The possible cause of this problem is that the XML files we used are configured for the SoC Development Board, the same board which the YSP preloader was configured for. We searched the same files for the SoCkit board, with no successful results. We looked for a solution to this obstacle, but we could not make it on time.

[8] You have to shut down the SoCkit to make any changes to the SD card. To do it correctly, please follow steps in page 82.

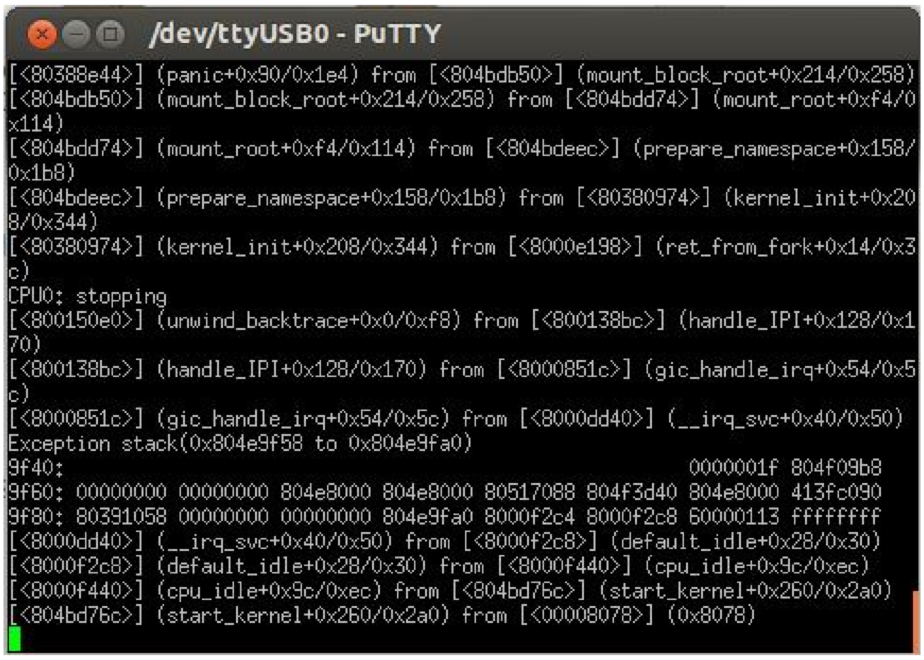
Accordingly, we decided to keep using the DTB generated with the YSP. As an alternative, **we can use an existent DTB contained in a subdirectory of the Altera folder, "/home/<user>/ altera/13.0sp1/embedded/embeddedsw/socfpga/sources/linux-altera-3.7/arch/arm/boot/dts". The DTS file is named "socfpga_cyclone5.dts". Use the following command to generate the DTB from this DTS (you have to be in an Altera Embedded Command Shell and situate in its folder):**

```
$ dtc -I dts -O dtb -o socfpga.dtb socfpga_cyclone5.dts
```

When we changed the DTB in the SD for this already configured DTB, system booted correctly.

8.4.2. Possible problem with the SD file system

After updating individual parts on some partition of the SD card or doing any changes, it is possible that when trying to boot Linux, it loads the kernel image but the system hangs, as seen in Figure 68.


/dev/ttyUSB0 - PuTTY

```
[<80388e44>] (panic+0x90/0x1e4) from [<804bdb50>] (mount_block_root+0x214/0x258)
[<804bdb50>] (mount_block_root+0x214/0x258) from [<804bdd74>] (mount_root+0xf4/0
x114)
[<804bdd74>] (mount_root+0xf4/0x114) from [<804bdeec>] (prepare_namespace+0x158/
0x1b8)
[<804bdeec>] (prepare_namespace+0x158/0x1b8) from [<80380974>] (kernel_init+0x20
8/0x344)
[<80380974>] (kernel_init+0x208/0x344) from [<8000e198>] (ret_from_fork+0x14/0x3
c)
CPU0: stopping
[<800150e0>] (unwind_backtrace+0x0/0xf8) from [<800138bc>] (handle_IPI+0x128/0x1
70)
[<800138bc>] (handle_IPI+0x128/0x170) from [<8000851c>] (gic_handle_irq+0x54/0x5
c)
[<8000851c>] (gic_handle_irq+0x54/0x5c) from [<8000dd40>] (__irq_svc+0x40/0x50)
Exception stack(0x804e9f58 to 0x804e9fa0)
9f40:                                                       0000001f 804f09b8
9f60: 00000000 00000000 804e8000 804e8000 80517088 804f3d40 804e8000 413fc090
9f80: 80391058 00000000 00000000 804e9fa0 8000f2c4 8000f2c8 60000113 ffffffff
[<8000dd40>] (__irq_svc+0x40/0x50) from [<8000f2c8>] (default_idle+0x28/0x30)
[<8000f2c8>] (default_idle+0x28/0x30) from [<8000f440>] (cpu_idle+0x9c/0xec)
[<8000f440>] (cpu_idle+0x9c/0xec) from [<804bd76c>] (start_kernel+0x260/0x2a0)
[<804bd76c>] (start_kernel+0x260/0x2a0) from [<00008078>] (0x8078)
```

Fig. 68: Linux does not load correctly

If that happens, move the SD card out from the SoCkit board and insert it again into the computer. Observe in Ubuntu a popup window like the one in Figure 69.

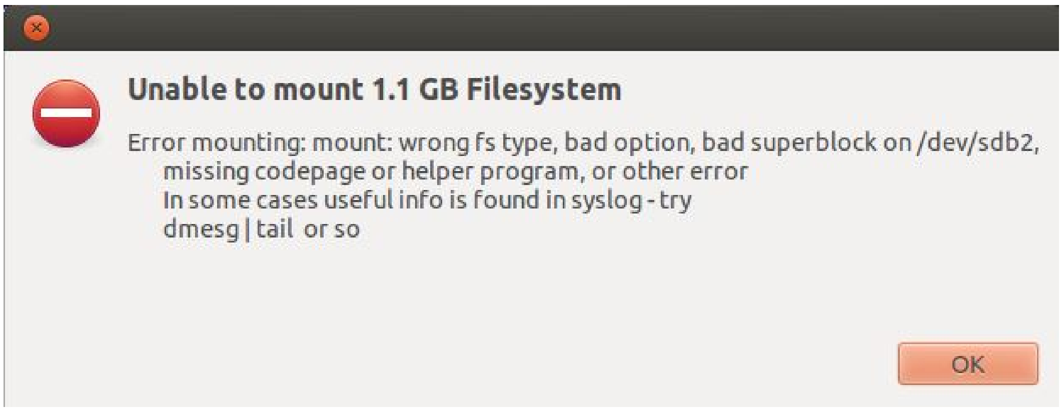

```
Unable to mount 1.1 GB Filesystem

Error mounting: mount: wrong fs type, bad option, bad superblock on /dev/sdb2,
    missing codepage or helper program, or other error
    In some cases useful info is found in syslog - try
    dmesg | tail or so
```

Fig. 69: Message of problems with file system

So it appears that there are some errors in the SD partitions where the Linux file system is contained. Just as in Figure 69, the problem is not specified but we can figure it out by typing the command "dmesg | tail" in an Ubuntu terminal. As it seems, the cause is that some file descriptors that are corrupted.

Trying to repair damaged memory blocks requires an advanced OS knowledge, so we chose a simple and quick solution: writing again the file system to the SD card. To do this, type the following command:

```
$ sudo dd if=altera-image-socfpga_cyclone5.ext3 of=/dev/sdb2
```

To execute this command it is required to be situated on the terminal in the directory where the compressed file system is, in our case "/home/<user>/yocto/build/tmp/deploy/images". Depending on how powerful is the computer where it is executed, it will take a while (in our case, 15-20 minutes). When reinserting the SD card into the SoCkit board slot and rebooting Linux, observe that it loads correctly this time, so the problem is solved.

8.5. ARM DS-5 Environment for developing and debugging applications. "Hello World" example

We are approaching to the end of this project. As mentioned before, the last objective was to develop a Linux application in order to use the SoCkit board resources and access to its peripherals. Unfortunately, we had not time enough to do so. We decided to give an overview of how to configure the environment offered by Altera for developing software applications (C, Java, C++...), ARM DS-5 (Development Studio). We will not get deep into it, but we will learn at least how to prepare it for debugging applications.

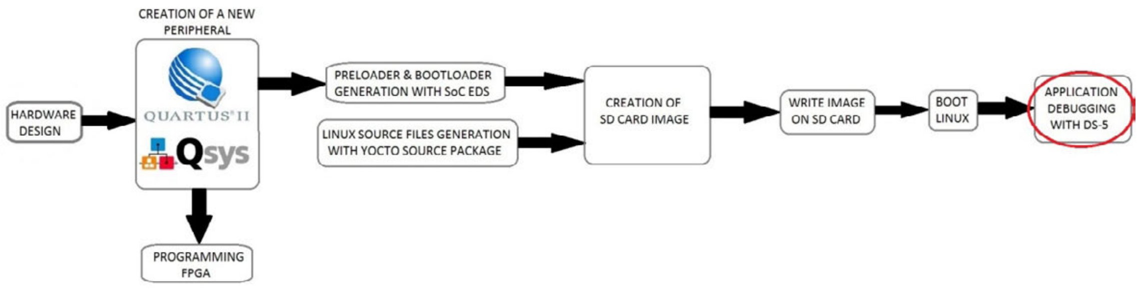

CREATION OF A NEW PERIPHERAL
HARDWARE DESIGN
QUARTUS II
Qsys
PROGRAMMING FPGA
PRELOADER & BOOTLOADER GENERATION WITH SoC EDS
LINUX SOURCE FILES GENERATION WITH YOCTO SOURCE PACKAGE
CREATION OF SD CARD IMAGE
WRITE IMAGE ON SD CARD
BOOT LINUX
APPLICATION DEBUGGING WITH DS-5

Fig. 70: Last step. Application debugging with DS-5

We will execute the most classic and simplest example in the programming field: the program Hello World. We will use an example project available in the Altera folder.

First, we have to **open the environment by executing the Embedded Command Shell from a terminal and typing "eclipse". Choose "/home/<user>/DS-5-Workspace" as the default workspace.**

Once DS-5 is loaded, close the welcome window and select "Import..." from the File tab. In the window that will appear, select "General → Existing projects into workspace". Click Next. In the next step choose the option "Select archive file", since what we have is a compressed file with the project (in case of having it in a non compressed folder, select the option "Select root directory"). **Click on the "Browse..." button and navigate to the directory where the project file is (in our case, in "/home/<user>/altera/13.0sp1/embedded/examples/software"), named "Altera-SoCFPGA-HelloWorld-Linux-GNU.tar.gz". The window should look as the one in Figure 71.**

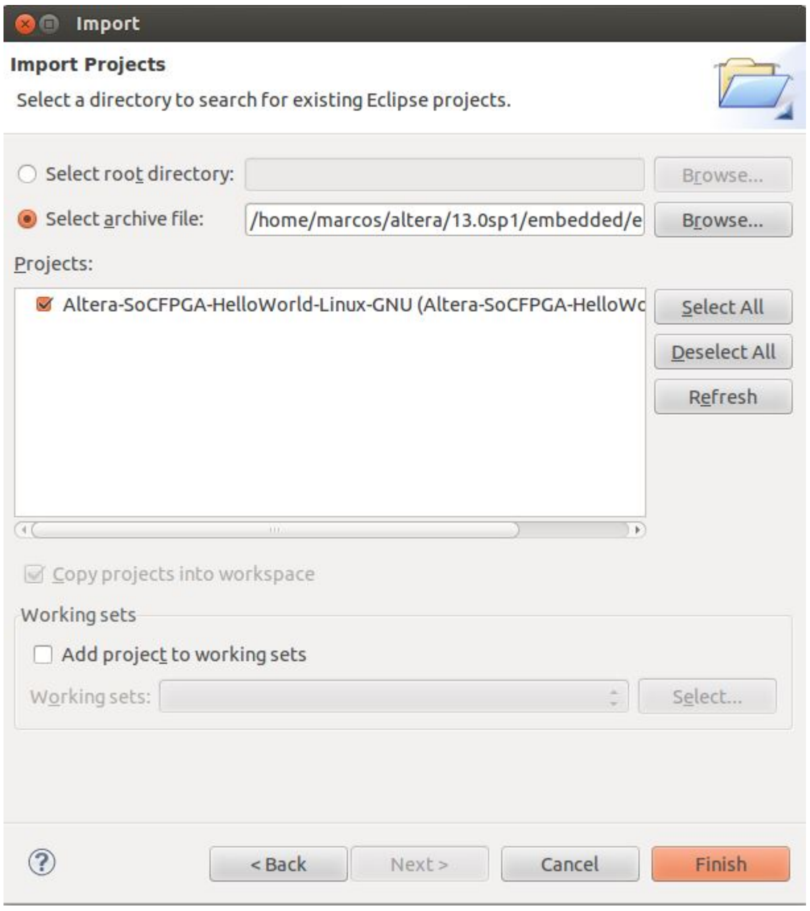

Import
Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory:
Browse...
Select archive file:
/home/marcos/altera/13.0sp1/embedded/e
Browse...
Projects:
Altera-SoCFPGA-HelloWorld-Linux-GNU (Altera-SoCFPGA-HelloWo
Select All
Deselect All
Refresh
Copy projects into workspace
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Cancel
Finish

Fig. 71: Import project in DS-5

Click Finish to import the project. Observe at the left of the DS-5 screen the new project imported including all its files (Figure 72).

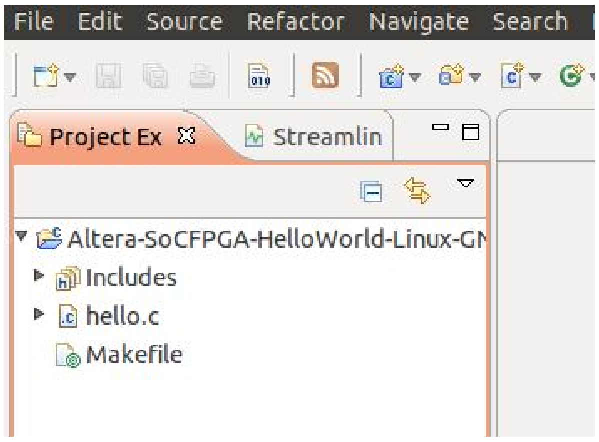
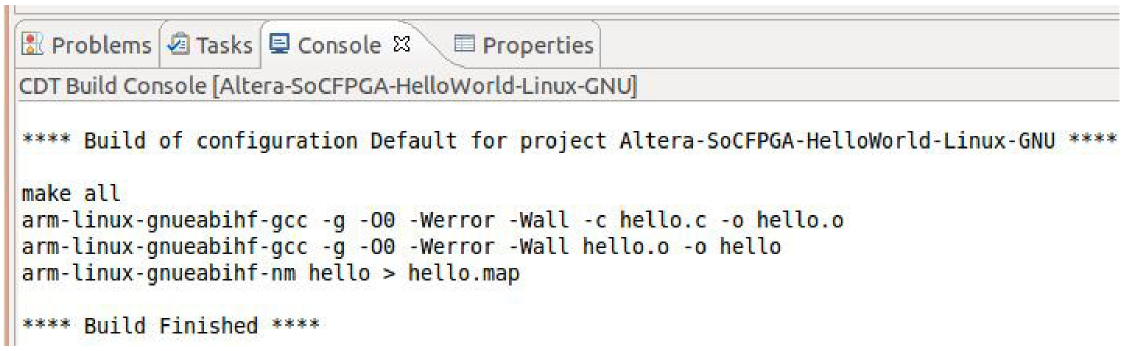

File Edit Source Refactor Navigate Search
Project Ex
Streamlin
Altera-SoCFPGA-HelloWorld-Linux-GN
Includes
hello.c
Makefile

Fig. 72: Project Hello World imported on DS-5

The next step is to compile the project. **Right click on the project and select "Build project".** A set of compilation messages are displayed in the Console screen at the bottom of the DS-5 window, as in Figure 73.



CDT Build Console [Altera-SoCFPGA-HelloWorld-Linux-GNU]

```
**** Build of configuration Default for project Altera-SoCFPGA-HelloWorld-Linux-GNU ****

make all
arm-linux-gnueabihf-gcc -g -O0 -Werror -Wall -c hello.c -o hello.o
arm-linux-gnueabihf-gcc -g -O0 -Werror -Wall hello.o -o hello
arm-linux-gnueabihf-nm hello > hello.map

**** Build Finished ****
```

Fig. 73: Compilating the project in DS-5

After finishing compilation, observe that a set of files that did not exist before compilation have been generated. Among all them, there is the executable file of the application (hello - [arm/le]). All project files are in Figure 74.

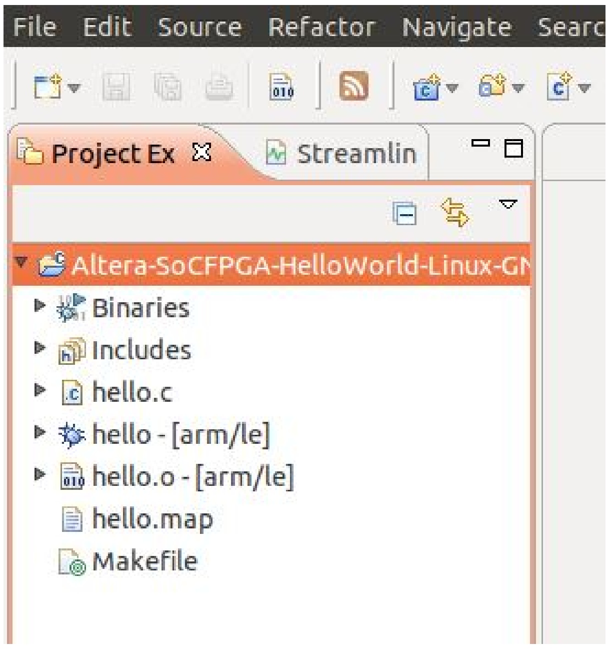
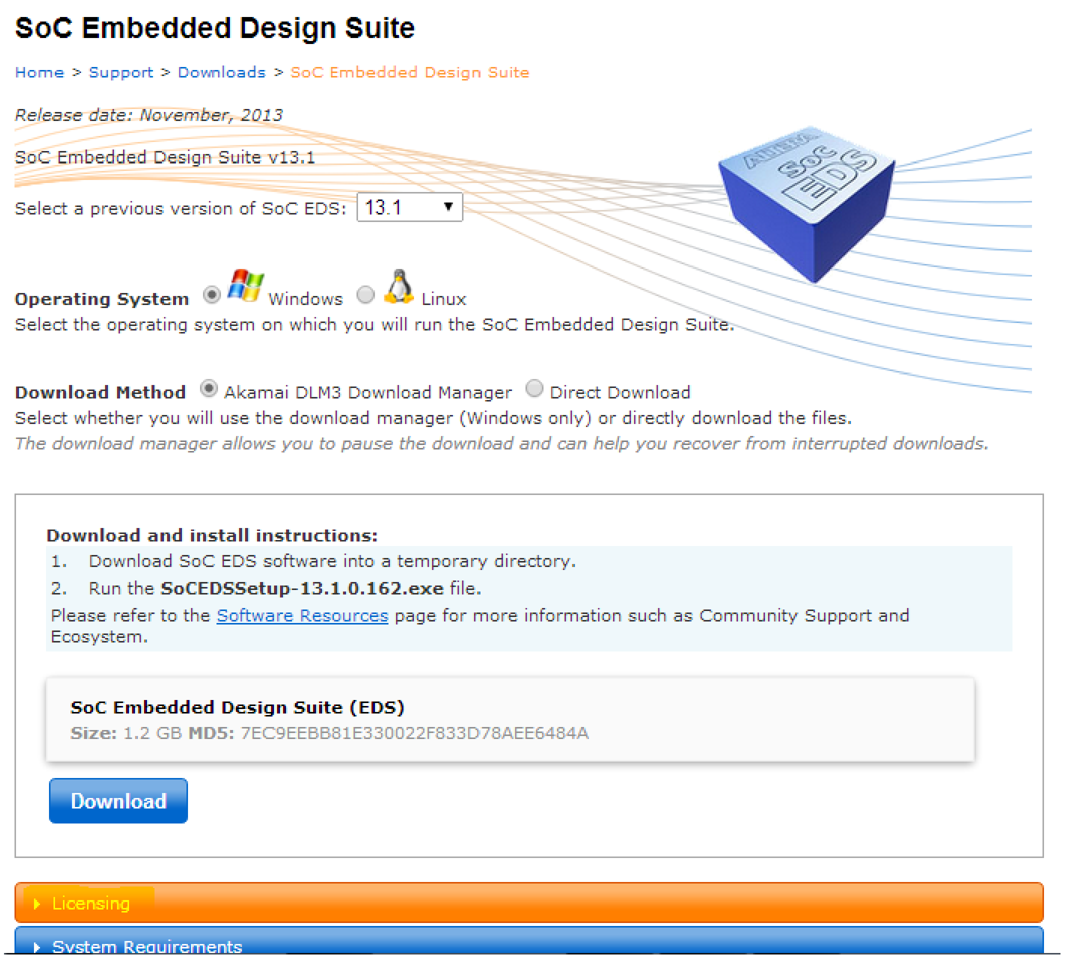


Fig. 74: Compiled project in DS-5

Before continuing, **we have to add a license for this software**, since we need it to debug applications. **Go to the SoC EDS download webpage**[9]**; scroll down and click on the "Licensing" tab (see Figure 75). We are using the free Web Edition of the Altera software, so click on the "activation code" link of the Web Edition, as seen in Figure 76.**

[9] http://dl.altera.com/soceds/?edition=subscription



SoC Embedded Design Suite

Home > Support > Downloads > SoC Embedded Design Suite

Release date: November, 2013

SoC Embedded Design Suite v13.1

Select a previous version of SoC EDS: 13.1

Operating System [✓ Windows] Linux
Select the operating system on which you will run the SoC Embedded Design Suite.

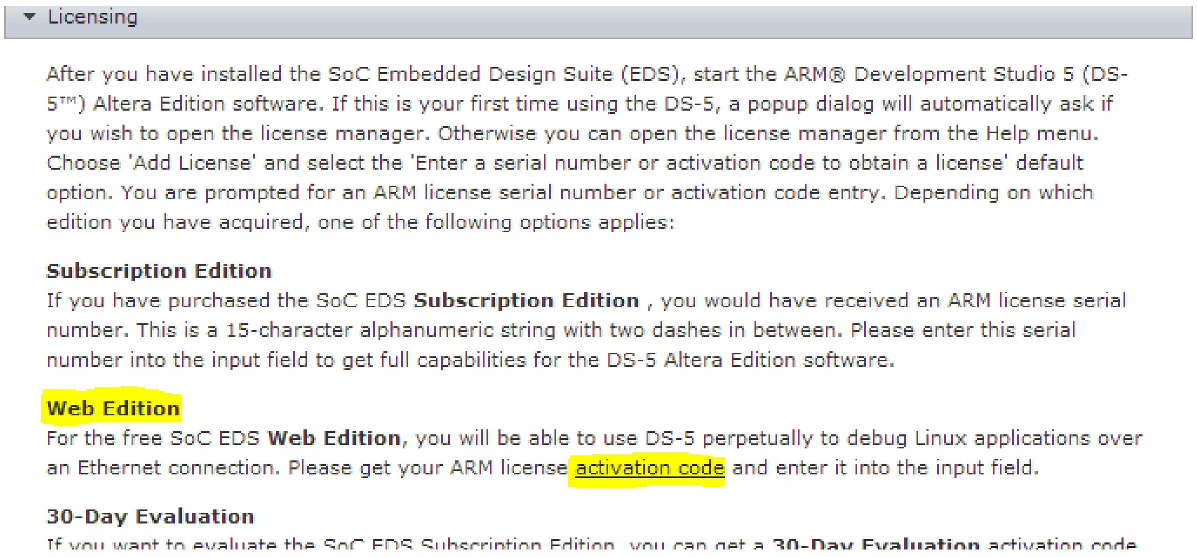
Download Method [✓ Akamai DLM3 Download Manager] Direct Download
Select whether you will use the download manager (Windows only) or directly download the files.
The download manager allows you to pause the download and can help you recover from interrupted downloads.

Download and install instructions:
1. Download SoC EDS software into a temporary directory.
2. Run the **SoCEDSSetup-13.1.0.162.exe** file.

Please refer to the Software Resources page for more information such as Community Support and Ecosystem.

SoC Embedded Design Suite (EDS)
Size: 1.2 GB **MD5:** 7EC9EEBB81E330022F833D78AEE6484A

Fig. 75: SoC EDS download page



Licensing

After you have installed the SoC Embedded Design Suite (EDS), start the ARM® Development Studio 5 (DS-5™) Altera Edition software. If this is your first time using the DS-5, a popup dialog will automatically ask if you wish to open the license manager. Otherwise you can open the license manager from the Help menu. Choose 'Add License' and select the 'Enter a serial number or activation code to obtain a license' default option. You are prompted for an ARM license serial number or activation code entry. Depending on which edition you have acquired, one of the following options applies:

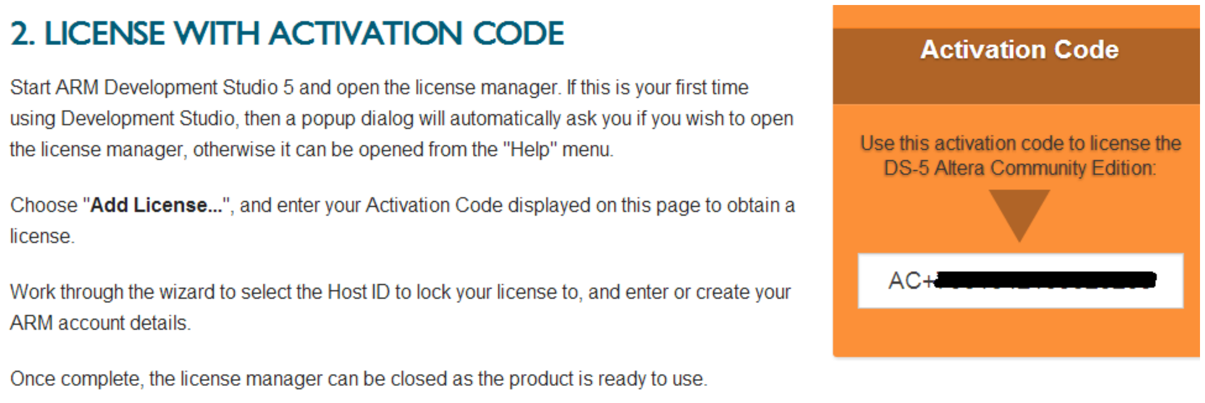
Subscription Edition
If you have purchased the SoC EDS **Subscription Edition** , you would have received an ARM license serial number. This is a 15-character alphanumeric string with two dashes in between. Please enter this serial number into the input field to get full capabilities for the DS-5 Altera Edition software.

Web Edition
For the free SoC EDS **Web Edition**, you will be able to use DS-5 perpetually to debug Linux applications over an Ethernet connection. Please get your ARM license activation code and enter it into the input field.

30-Day Evaluation
If you want to evaluate the SoC EDS Subscription Edition, you can get a **30-Day Evaluation** activation code

Fig. 76: Link to the DS-5 activation code

In the next webpage, scroll down and copy the activation code from the orange box (see Figure 77).



2. LICENSE WITH ACTIVATION CODE

Start ARM Development Studio 5 and open the license manager. If this is your first time using Development Studio, then a popup dialog will automatically ask you if you wish to open the license manager, otherwise it can be opened from the "Help" menu.

Choose "**Add License...**", and enter your Activation Code displayed on this page to obtain a license.

Work through the wizard to select the Host ID to lock your license to, and enter or create your ARM account details.

Once complete, the license manager can be closed as the product is ready to use.

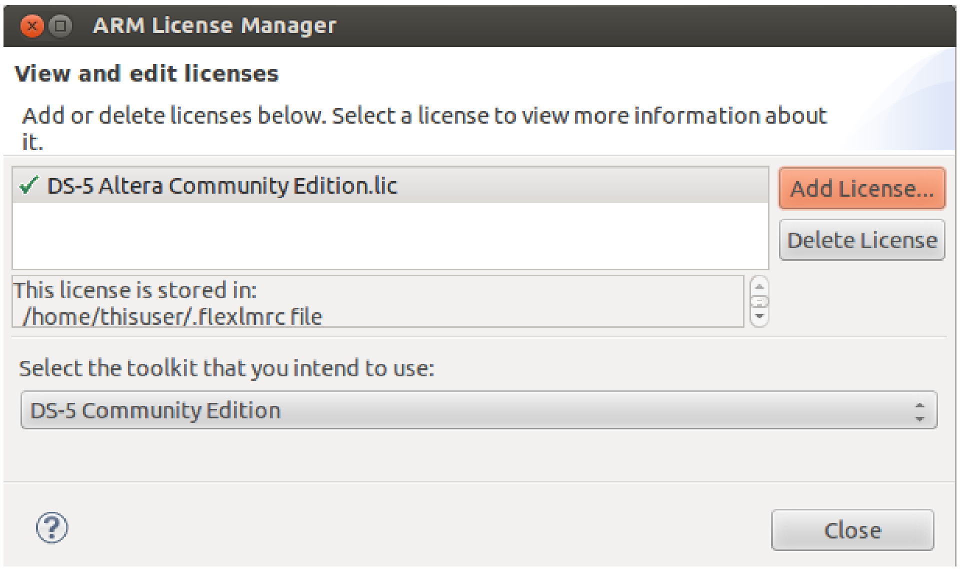
Activation Code

Use this activation code to license the DS-5 Altera Community Edition:

AC+[redacted]

Fig. 77: Activation code for DS-5

In DS-5, go to "ARM License Manager..." in the Help tab. Click on "Add License...". In the next window, paste the code copied from the webpage before. Click Next. You have to own an ARM account, register is free. Enter your account details and click Finish. The ARM License Manager window should look like in Figure 78. **Close the window.** You will be asked to restart Eclipse so the license changes will be processed. **Restart Eclipse.**



ARM License Manager

View and edit licenses

Add or delete licenses below. Select a license to view more information about it.

✓ DS-5 Altera Community Edition.lic

This license is stored in:
/home/thisuser/.flexlmrc file

Select the toolkit that you intend to use:

DS-5 Community Edition

Fig. 78: ARM License Manager

Now we have to **create a new connection between the host computer and the SoCkit board**. First, **open the Window tab in DS-5 and select "Open Perspective → Other..."**, whereupon a small window will appear. **Select the "Remote System Explorer" (RSE) option**.

Wait to the perspective to change to RSE, then right click on "Local" (left section of the DS-5 screen) and select "New → Connection..." as seen in Figure 79.

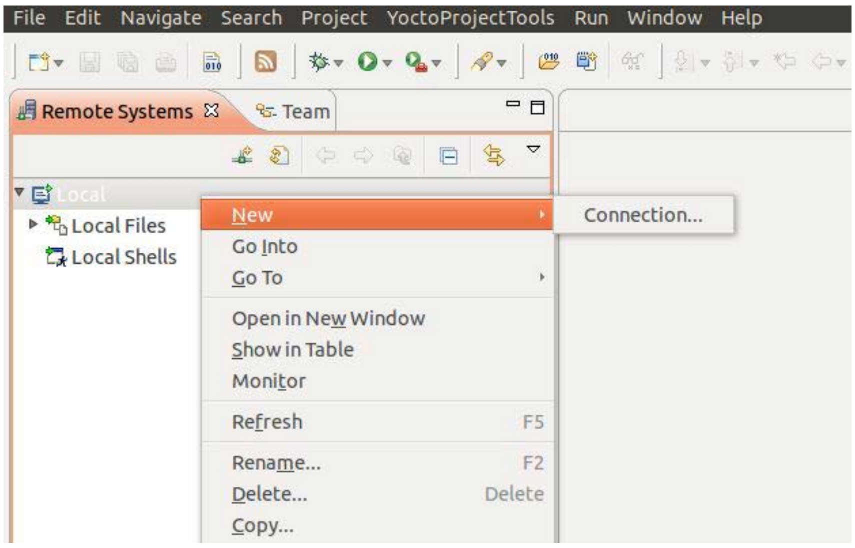
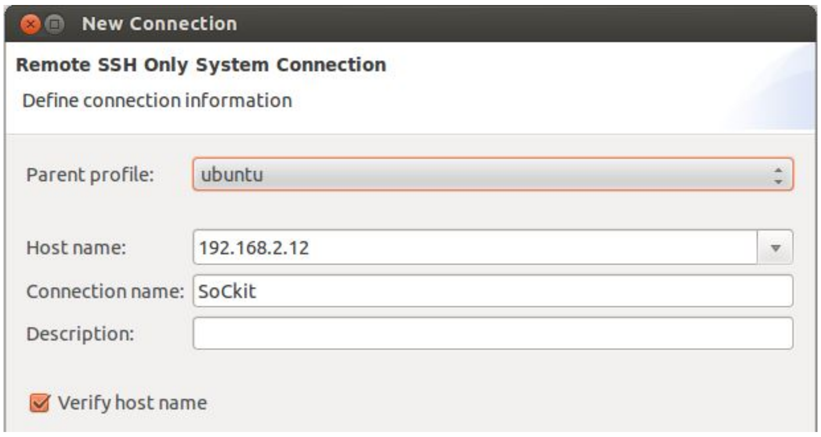


Fig. 79: New connection in DS-5

In the pop-up window select connection type as "SSH Only" (Secure SHell), then click Next. In the next step (Figure 80) fill the "Host name" field with the IP (Internet Protocol) address that we will assign to the SoCkit board, "192.168.2.12". Fill the "Connection name" field with the desired name (for example, "SoCkit"). Click Finish. We have created the connection to communicate with the SoCkit board. We can do this before assigning an IP to the SoCkit board; we are not connecting it to our computer yet.


New Connection

Remote SSH Only System Connection

Define connection information

Parent profile: ubuntu

Host name: 192.168.2.12

Connection name: SoCkit

Description:

☑ Verify host name

Fig. 80: Configuring a new connection in DS-5

We are creating an Ethernet network with two terminals: the host computer and the SoCkit board. We will assign two IP address within the private addresses range 192.168.0.0 - 192.168.255.255; we assigned the 192.168.2.12 address to the SoCkit board and the 192.168.2.13 address to the host computer. **Assign an IP address to the SoCkit board typing the following command in the PuTTY terminal:**

```
ifconfig eth0 192.168.2.12 up
```

To change the IP address of our computer go to "Start → Control Panel → Network and Internet → Network and Sharing Center". On the left side of the window click on "Change adapter settings".

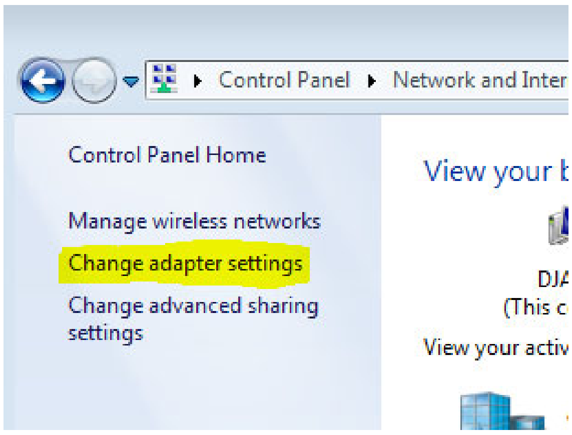
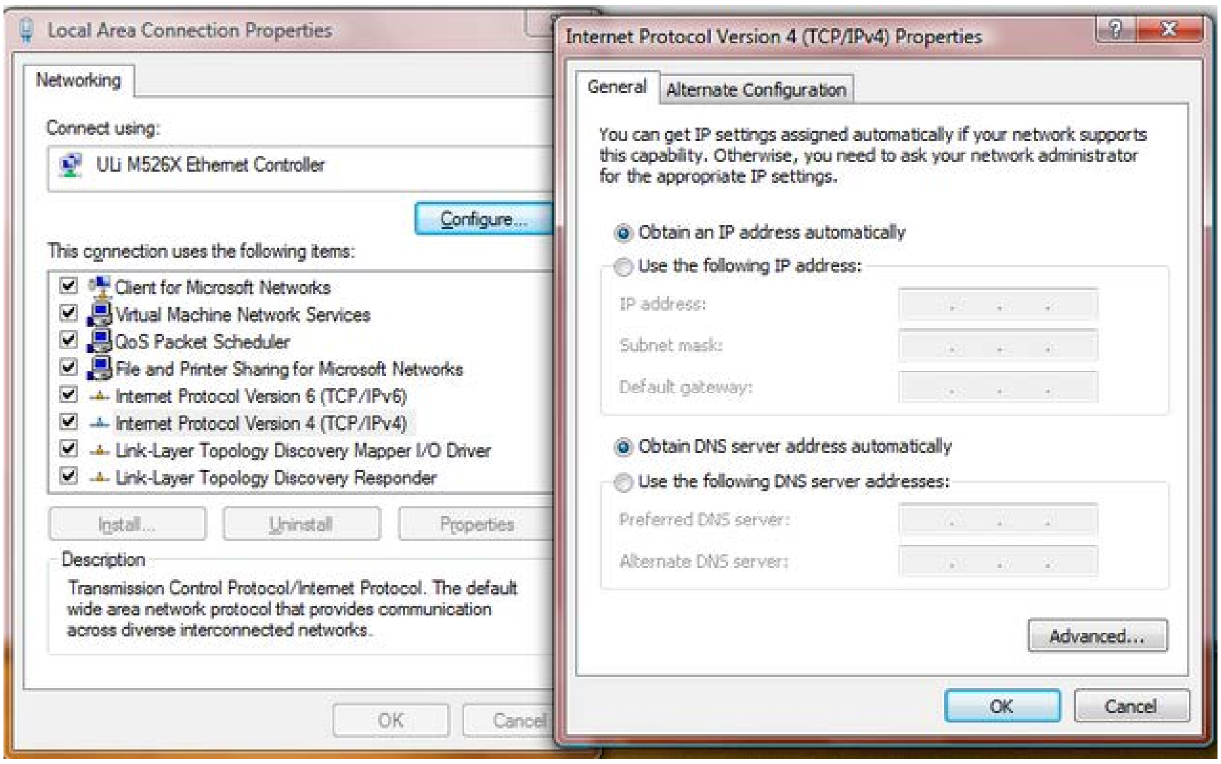


Fig. 81: Change adapter settings in Windows OS

In the new window, right click on "Local area connection" and select "Properties". The left window seen in Figure 82 will appear; **select "Internet Protocol Version 4 (TCP/IPv4)"** (Transmission Control Protocol) **and click on "Properties".** The right window seen in the same figure will appear now. **Select "Use the following IP address"; fill "IP address" as 192.168.2.13 and "Subnet mask" as 255.255.255.0. C lick OK in both windows.**


Local Area Connection Properties

Networking

Connect using:

ULi M526X Ethernet Controller

This connection uses the following items:

- ☑ Client for Microsoft Networks
- ☑ Virtual Machine Network Services
- ☑ QoS Packet Scheduler
- ☑ File and Printer Sharing for Microsoft Networks
- ☑ Internet Protocol Version 6 (TCP/IPv6)
- ☑ Internet Protocol Version 4 (TCP/IPv4)
- ☑ Link-Layer Topology Discovery Mapper I/O Driver
- ☑ Link-Layer Topology Discovery Responder

Description

Transmission Control Protocol/Internet Protocol. The default wide area network protocol that provides communication across diverse interconnected networks.

Internet Protocol Version 4 (TCP/IPv4) Properties

General | Alternate Configuration

You can get IP settings assigned automatically if your network supports this capability. Otherwise, you need to ask your network administrator for the appropriate IP settings.

● Obtain an IP address automatically

○ Use the following IP address:

IP address: . . .

Subnet mask: . . .

Default gateway: . . .

● Obtain DNS server address automatically

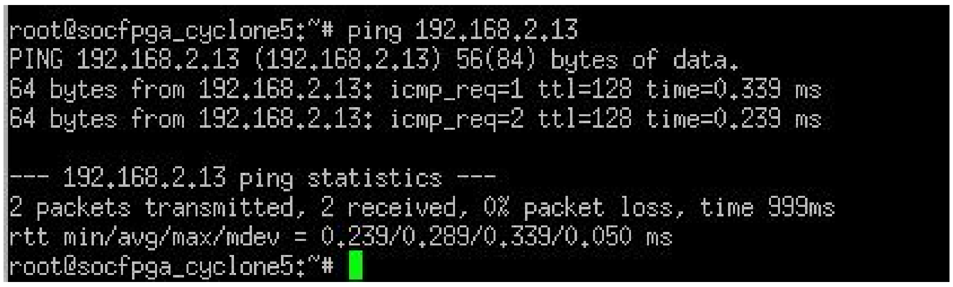
○ Use the following DNS server addresses:

Preferred DNS server: . . .

Alternate DNS server: . . .

Fig. 82: Setting IP address of host computer

To check if the network is connecting computer and board properly, use the "ping" command. Observe the execution of this command in Figure 83; both devices are correctly connected (of course, they have to be connected physically with an Ethernet cable). **If the ping fails, try to disable your Windows firewall for public networks** (we are not taking any risk since we are just connected to the SoCkit board, there is no danger).



```
root@socfpga_cyclone5:~# ping 192.168.2.13
PING 192.168.2.13 (192.168.2.13) 56(84) bytes of data.
64 bytes from 192.168.2.13: icmp_req=1 ttl=128 time=0.339 ms
64 bytes from 192.168.2.13: icmp_req=2 ttl=128 time=0.239 ms

--- 192.168.2.13 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.239/0.289/0.339/0.050 ms
root@socfpga_cyclone5:~#
```

Fig. 83: Checking connection between SoCkit and computer

Within the RSE perspective in DS-5, we can see now the new connection that we configured and named "SoCkit" (Figure 84). To establish connection we have to **deploy the "SFTP Files" (SSH File Transfer Protocol) instance and then "Root", after which DS-5 will try to connect with the SoCkit board.**

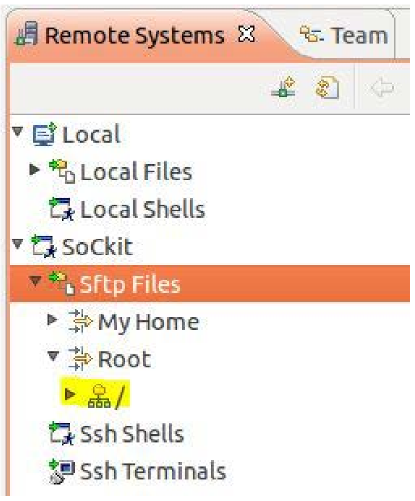
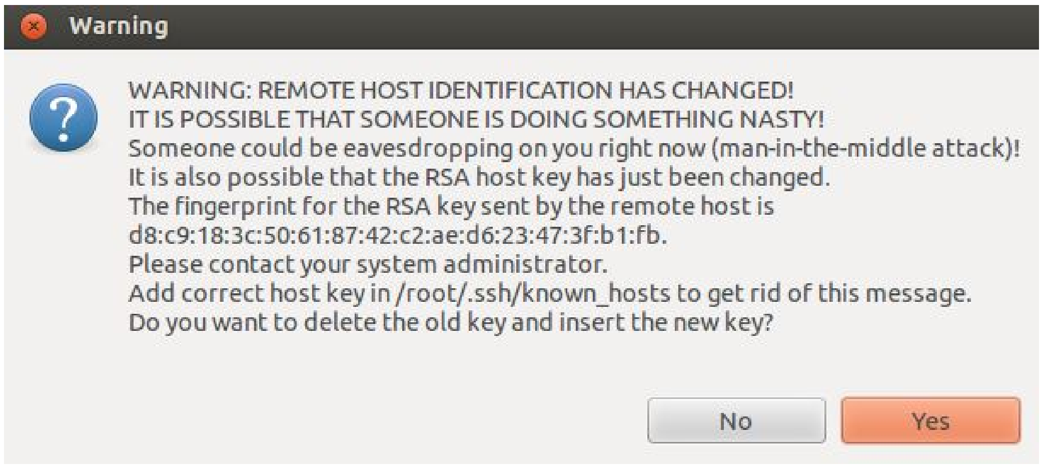


Fig. 84: New connection configured in DS-5

The first time we establish a connection, a pop-up window will ask us for a Linux user name and a password. **To add a password in the SoCkit Linux before entering it in the DS-5 window, just type "passwd" in the PuTTY terminal. It will ask for a new password twice, then the password is changed.**

If connection was established correctly, we will observe the Linux root directory in the left section of DS-5 screen (symbolized as a slash, /), as seen in Figure 84. When deploying it, all the file system and its folders will be displayed.

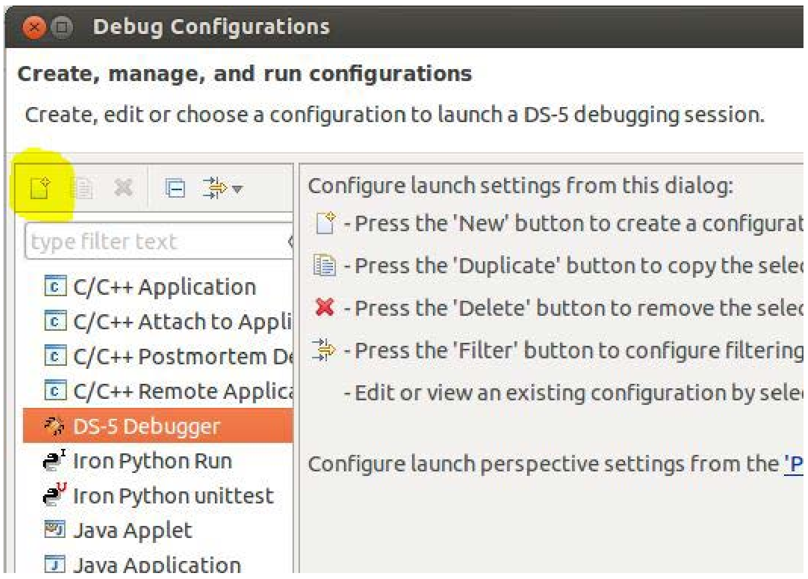
It is possible that, when connecting, a window as in Figure 85 emerges. This message seems to be caused because the RSA key of the host computer changed and does not match the one that the Linux OS of the SoCkit board knows. It could be, as in the message tells, a man-in-the-middle attack, but that is impossible since this network consist of our computer connected to the SoCkit, without internet. Click Yes to overwrite the key in the file where it is saved, so that we can establish the connection.



WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
d8:c9:18:3c:50:61:87:42:c2:ae:d6:23:47:3f:b1:fb.
Please contact your system administrator.
Add correct host key in /root/.ssh/known_hosts to get rid of this message.
Do you want to delete the old key and insert the new key?

Fig. 85: Warning message in DS-5 when connecting to the SoCkit board

The next step is to **configure the application debugging. Select "Debug Configurations..." in the Run tab; in the emerging window, s elect on the left menu "DS-5 Debugger" and click on the icon highlighted in Figure 86 to create a new configuration.**



Create, manage, and run configurations

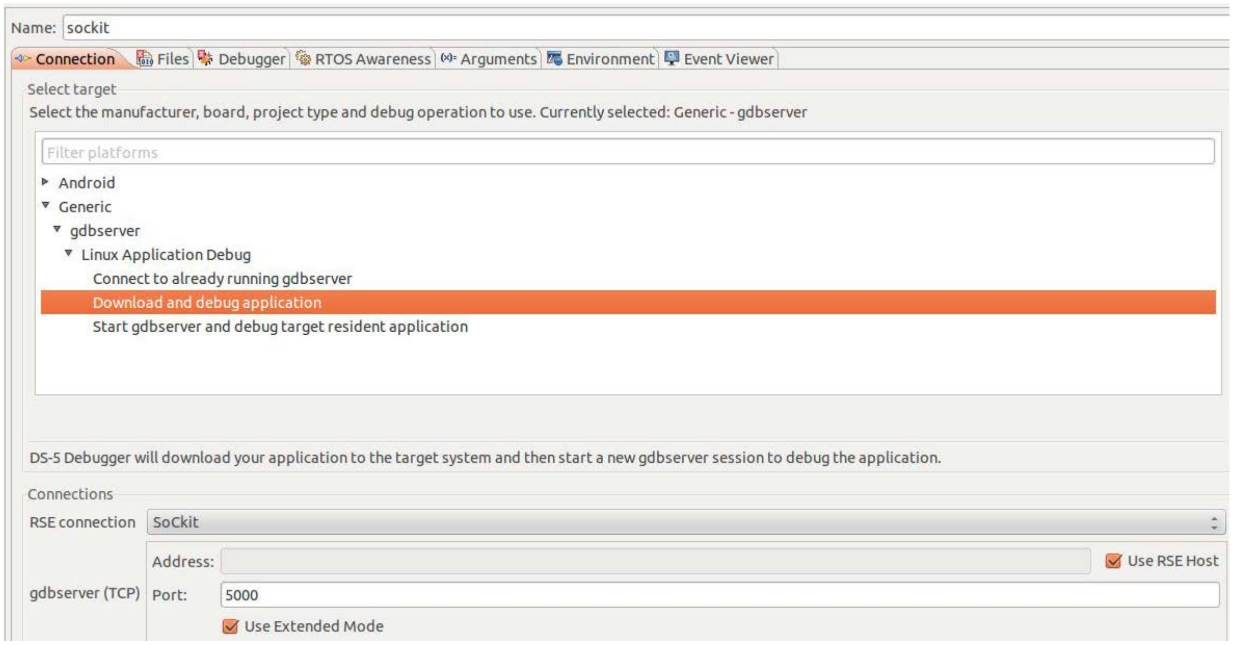
Create, edit or choose a configuration to launch a DS-5 debugging session.

Configure launch settings from this dialog:
- Press the 'New' button to create a configurat
- Press the 'Duplicate' button to copy the selec
- Press the 'Delete' button to remove the selec
- Press the 'Filter' button to configure filtering
- Edit or view an existing configuration by sele

Configure launch perspective settings from the 'P

Fig. 86: Debug configurations

In the new window, name the configuration as desired. Select the Connections tab. In the "Select target" section, choose the debugging type as "Generic → gdbserver → Linux Application Debug → Download and debug application". In the lower part of the window, we will see the connection we previously created in the RSE perspective, named "SoCkit". Observe all this in Figure 87.


Name: sockit

Connection | Files | Debugger | RTOS Awareness | Arguments | Environment | Event Viewer

Select target

Select the manufacturer, board, project type and debug operation to use. Currently selected: Generic - gdbserver

Filter platforms

- Android
- Generic
  - gdbserver
    - Linux Application Debug
      - Connect to already running gdbserver
      - Download and debug application
      - Start gdbserver and debug target resident application

DS-5 Debugger will download your application to the target system and then start a new gdbserver session to debug the application.

Connections

RSE connection: SoCkit

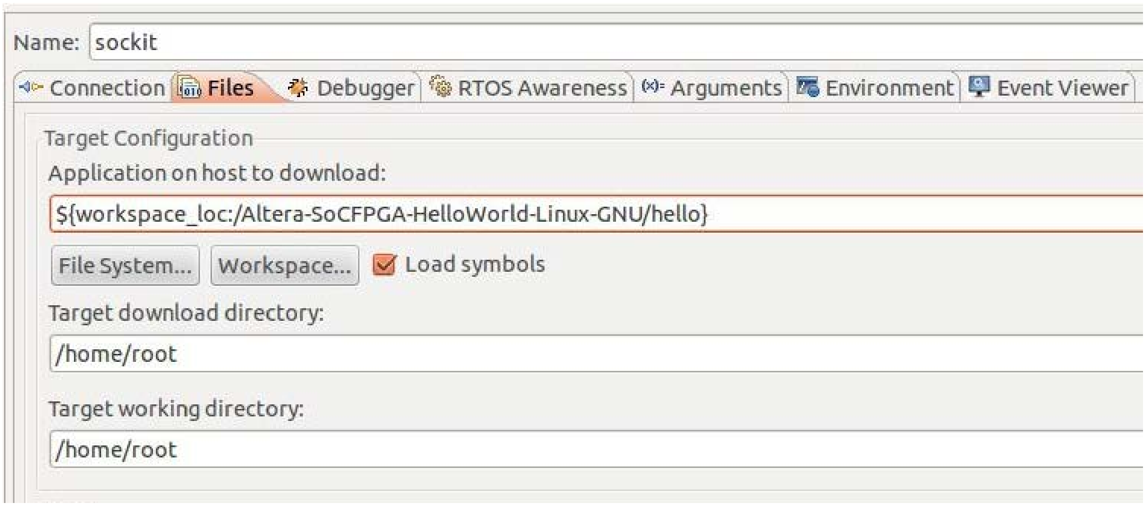
gdbserver (TCP): Address: ☑ Use RSE Host

Port: 5000

☑ Use Extended Mode

Fig. 87: Configuring the debugging. Connection

Now select the Files tab. To choose an "Application on host to download", click on "Workspace..." and search for the executable file of the Hello World application, named "hello" with no extension. Fill the "Target download directory" and "Target working directory" fields writing "/home/root" to copy the application files to that directory in the SoCkit Linux file system.


Name: sockit

Connection | Files | Debugger | RTOS Awareness | Arguments | Environment | Event Viewer

Target Configuration

Application on host to download:

${workspace_loc:/Altera-SoCFPGA-HelloWorld-Linux-GNU/hello}

File System... Workspace... ☑ Load symbols

Target download directory:

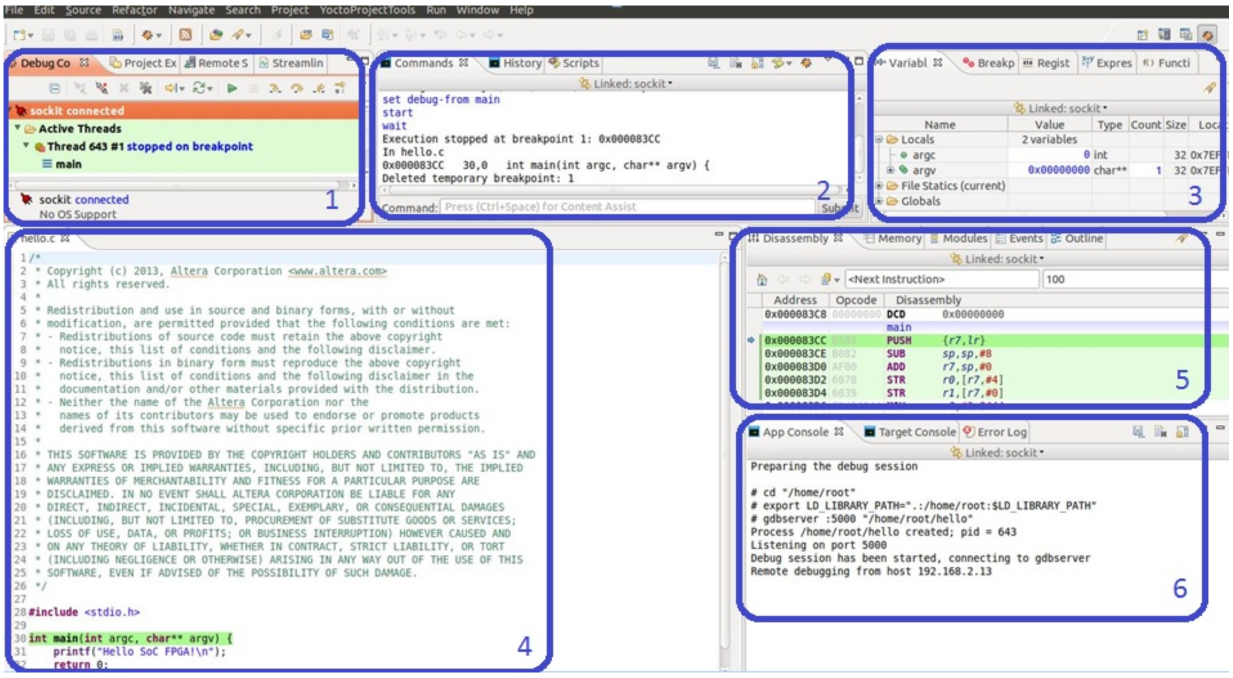
/home/root

Target working directory:

/home/root

Fig. 88: Configuring the debugging. Files

Click on Apply to save all the changes in the configuration (button situated in the lower right corner of the window). **Click on Debug** to launch the debugging. Once everything is ready to execute and debug the application, the DS-5 screen will look as the window in Figure 89 (next page).



```
sockit connected
Active Threads
  Thread 643 #1 stopped on breakpoint
    main
sockit connected
No OS Support
```

```
set debug-from main
start
wait
Execution stopped at breakpoint 1: 0x000083CC
In hello.c
0x000083CC   30,0   int main(int argc, char** argv) {
Deleted temporary breakpoint: 1
Command: Press (Ctrl+Space) for Content Assist
```

Name	Value	Type	Count	Size	Loca
Locals	2 variables				
argc	0	int		32	0x7EF
argv	0x00000000	char**	1	32	0x7EF
File Statics (current)					
Globals					

```
/*
 * Copyright (c) 2013, Altera Corporation <www.altera.com>
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 * - Redistributions of source code must retain the above copyright
 *   notice, this list of conditions and the following disclaimer.
 * - Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * - Neither the name of the Altera Corporation nor the
 *   names of its contributors may be used to endorse or promote products
 *   derived from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL ALTERA CORPORATION BE LIABLE FOR ANY
 * DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

#include <stdio.h>

int main(int argc, char** argv) {
    printf("Hello SoC FPGA!\n");
    return 0;
```

```
<Next Instruction>   100
Address      Opcode    Disassembly
0x000083C8   00000000  DCD     0x00000000
                       main
0x000083CC   B580      PUSH    {r7,lr}
0x000083CE   B082      SUB     sp,sp,#8
0x000083D0   AF00      ADD     r7,sp,#0
0x000083D2   6078      STR     r0,[r7,#4]
0x000083D4   6039      STR     r1,[r7,#0]
```

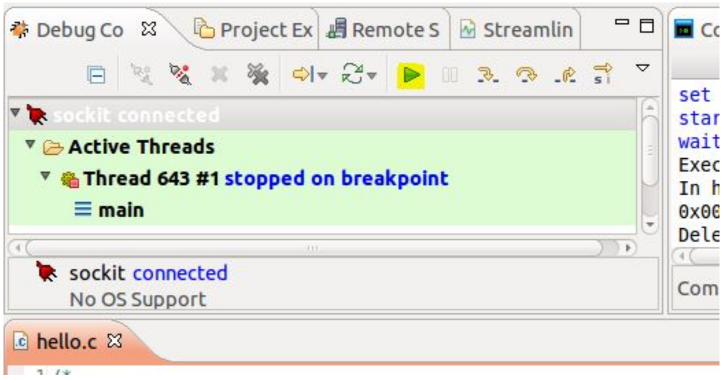
```
Preparing the debug session

# cd "/home/root"
# export LD_LIBRARY_PATH=".:/home/root:$LD_LIBRARY_PATH"
# gdbserver :5000 "/home/root/hello"
Process /home/root/hello created; pid = 643
Listening on port 5000
Debug session has been started, connecting to gdbserver
Remote debugging from host 192.168.2.13
```

Fig. 89: DS-5 environment ready for debugging

The DS-5 window divides now into six sections; we will briefly describe them:

1. In this section we can connect to and disconnect from the SoCkit board, execute the application or debug it going step by step through the code.
2. In this section we can add and execute scripts, type debugging commands or view output messages from the debugger.
3. Here we can enable, disable or delete breakpoints, view or modify the application variables or memory registers, etc.
4. Source code in C or C++ language of the application which is going to be executed or debugged.
5. In this section we can view the assembly code of the application.
6. Here we can observe errors of the application or the program and the SoCkit console.

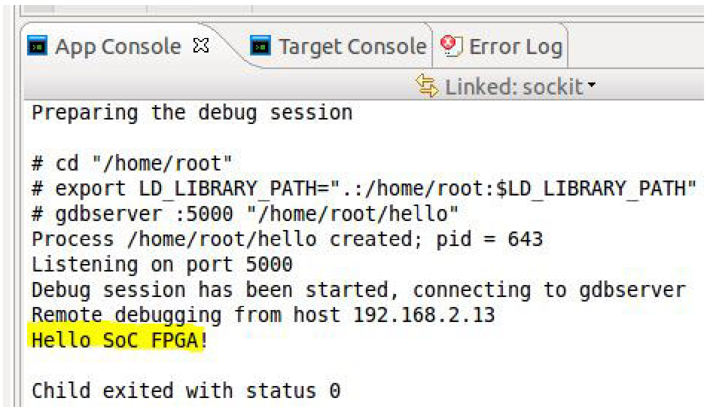
Observe in Figure 90 the button to execute the application highlighted. Click on it to check if our application runs.



```
sockit connected
Active Threads
  Thread 643 #1 stopped on breakpoint
    main
sockit connected
No OS Support
```

Fig. 90: Execute application

If everything goes well, we can see in the SoCkit console (section 6 in Figure 89) the message written by the application. This means our application was executed correctly and the DS-5 environment is properly configured and prepared for develop, debugging and executing applications in the SoCkit board.



```
Preparing the debug session

# cd "/home/root"
# export LD_LIBRARY_PATH=".:/home/root:$LD_LIBRARY_PATH"
# gdbserver :5000 "/home/root/hello"
Process /home/root/hello created; pid = 643
Listening on port 5000
Debug session has been started, connecting to gdbserver
Remote debugging from host 192.168.2.13
Hello SoC FPGA!

Child exited with status 0
```

Fig. 91: Hello World runs correctly

Lastly, **before turning off the SoCkit, we have to close the connection clicking on the button highlighted in Figure 92.**

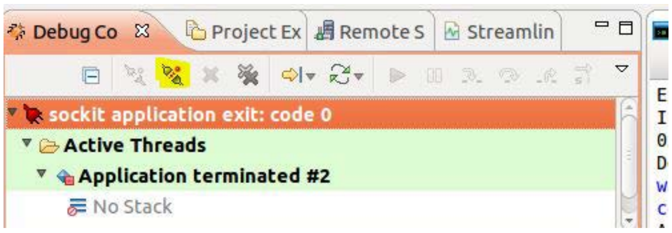
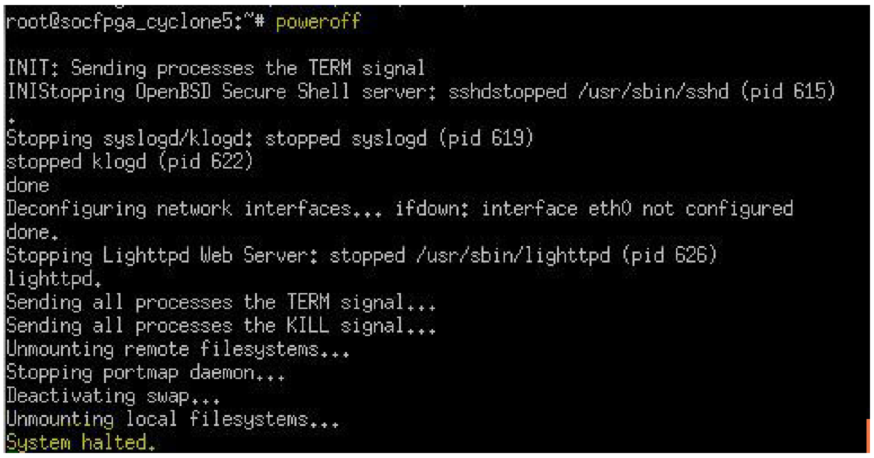


Fig. 92: Close connection with SoCkit board

Shut down the Linux OS in the SoCkit board by typing "poweroff", so all Linux processes will be made to stop its execution. After typing the command, a set of messages will be displayed in the console. **Do not turn off the SoCkit until the message "System Halted" is displayed. Now close the PuTTY terminal and turn off the SoCkit board.**



```
root@socfpga_cyclone5:~# poweroff

INIT: Sending processes the TERM signal
INIStopping OpenBSD Secure Shell server: sshdstopped /usr/sbin/sshd (pid 615)
.
Stopping syslogd/klogd: stopped syslogd (pid 619)
stopped klogd (pid 622)
done
Deconfiguring network interfaces... ifdown: interface eth0 not configured
done.
Stopping Lighttpd Web Server: stopped /usr/sbin/lighttpd (pid 626)
lighttpd.
Sending all processes the TERM signal...
Sending all processes the KILL signal...
Unmounting remote filesystems...
Stopping portmap daemon...
Deactivating swap...
Unmounting local filesystems...
System halted.
```

Fig. 93: Shutting down Linux

9. Summary and conclusions

To conclude this project, we will recapitulate all the important ideas: we have an embedded system where we have to install a Linux distribution, a free Unix based OS which offers some advantages over other embedded OSs. To build our Linux distribution we need to generate a set of files:

- Preloader: It performs some configurations and pass boot flow control to the bootloader.
- Bootloader: Prepares everything the OS needs to work; it loads the kernel and the DTB into memory, boots kernel and passes the DTB contents to it. We used an open source bootloader, U-boot.
- DTB: Binary file representing a data structure that describes the system hardware and "translates" it to the OS.
- Kernel
- File system

Our development kit, the SoCkit board, has a SoC system. A SoC system contains all the necessary components to power a computer, everything in one chip. The SoC solution within the SoCkit board unifies a FPGA made by Altera and a HPS made by ARM, composed by a double core Cortex-A9, a set of peripherals and a series of interfaces with the FPGA.

As main objective of this project we attempted to generate an own Linux distribution adapted to the SoCkit board, without depending on pre-made content facilitated by the manufacturer or developer communities. To do that, we used the Altera tools and a software package from Yocto, a custom Linux distributions project recommended by Altera. Thus, we have a complete and functional system to develop applications. Another objective consist of, as we already have a hardware design project configured, adding a totally new custom peripheral to it and checking whether it works correctly. The last proposed objective was to develop a Linux application in C language which use some of the resources in the SoCkit board.

We used a computer with Windows OS. However, some tools or programs need to be installed and executed in a Linux-based OS host. Therefore, we used a virtual machine and installed Ubuntu Linux 12.04 LTS on it, the most popular Linux distribution.

To configure the SoCkit hardware, we downloaded a project from the developers web RocketBoards.org. It is a reference design that contains all configurations and connections of the components needed to use the SoCkit resources. Nevertheless, it was necessary to change a couple of configuration details which may pass unperceived and may seem insignificant, but they must not be disregarded because the generated files for the Linux distribution do not work correctly if we do not change them.

Being the system hardware prepared, we got ready to create an own peripheral. We chose a very simple peripheral: a 16-bit register mapped in memory, that receives a value written to its memory address, it does some modification to that value so that we read the written value modified (we change it to make sure that we are really interacting with our peripheral). We used VHDL code to implement this peripheral and some interfaces (as a Qsys component is composed of two parts: internal hardware modules and external Avalon interfaces), though

Verilog can be used as well. We integrated the component in Qsys, adding it to the system, giving it a memory address and connecting it with the necessary components.

To check the correct operation of our component, we used an Altera tool that executes TCL scripts, System Console. We learnt how to develop a simple script which does some operations to check if our component works: we write a value to its memory address and we read the stored value. We verified twice if it works correctly: on the one hand, a message was sent to the console of System Console with the read value; on the other hand, the read value was written to the memory address where the FPGA LEDs are mapped so they light and show the value in binary. In both cases, the value should be the written value modified as coded in the VHDL file. After executing the script, we checked the value written to our component was modified just as expected. Thus, our component works correctly so we achieved one of the objectives: creating a new custom peripheral and adding it to the system, leaving it prepared to be used.

The next step was to generate the necessary files to build the Linux distribution for the SoCkit board. First of all, the Altera software that configures the hardware system generates a set of files that other tools use in order to create some of the components of the Linux distribution; particularly, a preloader, a bootloader and a DTB. The preloader and the bootloader were created with no troubles. To generate the DTB we encounter some difficulties: using the "sopc2dts" tool previously downloaded, we generated correctly the DTB, but when booting Linux OS in the SoCkit using this DTB, the system was not able to load the kernel, it hanged. Despite of looking for a solution to this problem, it could not be solved on time so we decided to use the DTB created with the YSP. There exists another DTB contained in a subdirectory of the Altera folder; using this DTB the system boots correctly as well.

After that, we downloaded and installed the YSP to generate the rest of the files needed for the Linux distribution (at least, we need the kernel and the file system). We compiled everything and all the files were created.

The last step before booting Linux in the SoCkit board is to build an image file containing all the Linux distribution and write it to the SD card. We used a tool from the Altera software with which we just have to type a set of commands. At the first time, we used all the files (preloader, bootloader, DTB, kernel and file system) from Yocto. The image was created without any problem.

Once we had the image file to write it to the SD, we only needed to type another command to write it. This took a while, but after several minutes, the image finished writing. A problem was encountered: Linux did not boot. The cause was looked into, and it turned out that the Yocto files were configured for another similar development kit made by Altera. The preloader needs to take into account some hardware aspects to perform its configurations, so this preloader was not able to run in the SoCkit. Hence, the solution was simple: changing the preloader in the SD for the one generated by the Altera software, which is for the SoCkit. After rebooting Linux with the new preloader, voilà! It works. We achieved the main objective: generating a custom Linux distribution for the SoCkit which boots correctly so in this way we can interact with the board through the Linux OS.

We also changed the bootloader in the SD for the one generated with the Altera software; the Linux OS booted correctly that time too. The DTB in the SD was also changed by the existing DTB from the Altera folder (not the one generated by the "sopc2dts" tool) and the system kept booting well.

The last objective, developing a Linux application in C which uses the resources of the SoCkit board, could not be achieved due to lack of time. Instead, we tried to see how to configure the development environment and to prepare it to execute and debugging applications. The typical "Hello World" application was executed as an example to check if the environment is configured properly.

As it has not been possible to achieve all the proposed objectives, we take advantage of it to propose future lines of work to continue with this project and complete all the initial objectives:

- Use the DTB generated with the "sopc2dts" tool and make the Linux OS booting in the SoCkit correctly using this DTB.
- Download and use a newer version of the GHRD project, released after this project was ended.
- Develop a software application in C or C++, simple or as complex as desired, which uses some resources of the SoCkit board through Linux.
- Instead of YSP, use some alternative way to generate the Linux distribution files, such as BuildRoot. This tool is easy to install, easy to use, flexible, with a lot of configuration options, simple structure and which supports hundreds of packages and different types of file systems.

In this project, saying it in one sentence, we installed a Linux distribution in a development kit which unifies a FPGA and a HPS in a SoC system. As possible uses of this integration of software (Linux) and hardware (SoCkit) we propose the didactic use, to learn how to configure and personalize the SoC system HPS, creating custom peripherals to integrate it into the system. In addition, it is conceived as a help for the software developers, since we prepared an environment to program Linux applications from which we can access to the peripherals of the SoCkit board. Thus, it could be used as an autonomous system, using the Linux OS advantages to use the SoCkit resources, with a wide range of possibilities: communication through Ethernet network, serial communication, audio or video coding and decoding, data collection (with the temperature sensor, infrared sensor...) for controlling or storage... It is a versatile and very flexible system with all the advantages of programmable logic, that we can use from now.

9. Summary and conclusions

10. Bibliography

WIKIPEDIA. *Embedded System* [online]. September 2006 [cited December 2013]. Available from Web: <http://en.wikipedia.org/wiki/Embedded_system>

SETIA, Priyansh. Linux - A Memorable Milestone. *Ocean of Webs* [online]. May 2013 [cited December 2013]. Available from Web: <http://oceanofwebs.com/linux-a-memorable-milestone/>

JONES, M. Tim. Anatomy of the Linux kernel: History and architectural decomposition. *IBM developerWorks* [online]. 06 June 2007 [cited December 2013]. Available from Web: <http://www.ibm.com/developerworks/library/l-linux-kernel/>

ROCKET BOARDS. *GSRD - Boot Flow* [online]. 10 October 2013, Updated 27 November 2013 [cited December 2013]. Available from Web: <http://rocketboards.org/foswiki/Documentation/GSRDBootFlow>

ALTERA. *Booting and Configuration Introduction* [online]. January 2012, updated 30 December 2013 [cited December 2013] . Available from Web: <http://www.altera.com/literature/hb/cyclone-v/cv_5400A.pdf>

SHARMA, Shashank. *System on a Chip: what you need to know about SoCs* [online]. 5 May 2013 [cited December 2013]. Available from Web: <http://www.techradar.com/news/computing/pc/system-on-a-chip-what-you-need-to-know-about-socs-1147235>

ALTERA. *Altera's User-Customizable ARM-based SoC* [online]. April 2013 [cited December 2013]. Available from: <http://www.altera.com/literature/br/br-soc-fpga.pdf>

ROCKET BOARDS. *Linux Getting Started on Altera SoC Development Board - Using Yocto Source Package* [online]. Updated 28 November 2013 [cited October 2013]. Available from:<http://rocketboards.org/foswiki/Documentation/AlteraSoCDevelopmentBoardYoctoGettingStarted>

ROCKET BOARDS. *GSRD - Generating the Device Tree* [online]. Updated 26 November 2013 [cited November 2013]. Available from Web: <http://rocketboards.org/foswiki/Documentation/GSRDDeviceTreeGenerator<

ALTERA. *Analyzing and Debugging Designs with the System Console* [online]. 04 November 2013 [cited December 2013]. Available from Web: <http://www.altera.com/literature/hb/qts/qts_qii53028.pdf>

ALTERA FORUM. *Problems with Yocto/Poky Linux on the Arrow SoCkit* [online]. 19 July 2013, updated 23 September 2013 [cited October 2013]. Available from Web: <http://www.alteraforum.com/forum/archive/index.php/t-41541.html>

10. Bibliography

ROCKET BOARDS. *Arrow SoCKIT Evaluation Board - How to Boot Linux* [online]. 01 May 2013, updated 31 January 2014 [cited October 2013]. Available from Web: <http://rocketboards.org/foswiki/Documentation/ArrowSoCKITEvaluationBoardLinuxGettingStarted>

ALTERA. *Altera SoC Embedded Design Suite: User Guide* [online]. 08 November 2013 [cited November 2013]. Available from Web: <http://www.altera.com/literature/ug/ug_soc_eds.pdf>

ALTERA. *Making Qsys components* [online]. Updated August 2012 [cited November 2013]. Available via FTP from: <ftp://ftp.altera.com/up/pub/Altera_Material/12.0/Tutorials/making_qsys_components.pdf >

ALTERA. *Creating Qsys components* [online]. December 2010, updated 04 November 2013 [cited November 2013]. Available from Web: <http://www.altera.com/literature/hb/qts/qsys_components.pdf>

ARM. *ARM DS-5 Getting Started with DS-5 (Version 5.2)* [online]. 2010 [cited November 2013]. Available from Web: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0478b/cacjhagg.html>

WIKIBOOKS. *Programmable Logic/VHDL* [online]. 3 November 2007, updated 2 February 2009 [cited December 2013]. Available from Web: <http://en.wikibooks.org/wiki/Programmable_Logic/VHDL>

ROCKET BOARDS. *GSRD - GHRD Overview* [online]. 09 October 2013, updated 27 November 2013 [cited November 2013]. Available from Web: <http://rocketboards.org/foswiki/Documentation/GSRDGhrd>

ARROW. *SoC Hardware Lab Instructions, version 13.0*. 16 May 2013 [cited September 2013]. Available for download from: <http://www.arrownac.com/solutions/sockit/files/SoCKIT_Materials.zip>

ARROW. *SoC Software Lab Instructions, version 13.0*. 16 May 2013 [cited September 2013]. Available for download from: <http://www.arrownac.com/solutions/sockit/files/SoCKIT_Materials.zip>

PURDIE, Richard. *Yocto Project Reference Manual* [online]. 24 November 2010, updated April 2013 [cited January 2014]. "1.3.1. Supported Linux Distros". Available from Web: <http://www.yoctoproject.org/docs/1.4/ref-manual/ref-manual.html#detailed-supported-distros>

ALTERA. *Avalon Interface Specifications* [online]. May 2013 [cited December 2013]. Available on Web: <http://www.altera.com/literature/manual/mnl_avalon_spec.pdf>

Appendix A: Installing USB Blaster II device driver in Windows OS

If we connect our SoCkit board to the Windows OS, it will not be recognized by the computer. For that reason, we have to install the device driver manually. In the case of Linux, this step is not necessary because Ubuntu incorporates drivers for the USB-serial converter chip embedded in the SoCkit board, letting the computer to see the board as a virtual serial port, usually called "/dev/ttyUSB0".

First, open the Windows device manager by going through "Start → Control Panel → System and Security → System" and clicking on "Device Manager" at the left column. Go to "Other devices" and right click on "Unknown device". Select "Update Driver Software...".

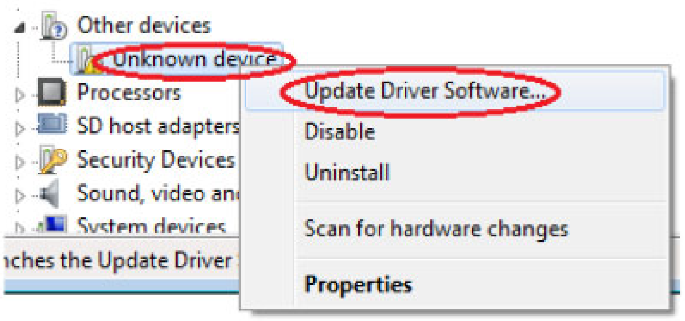
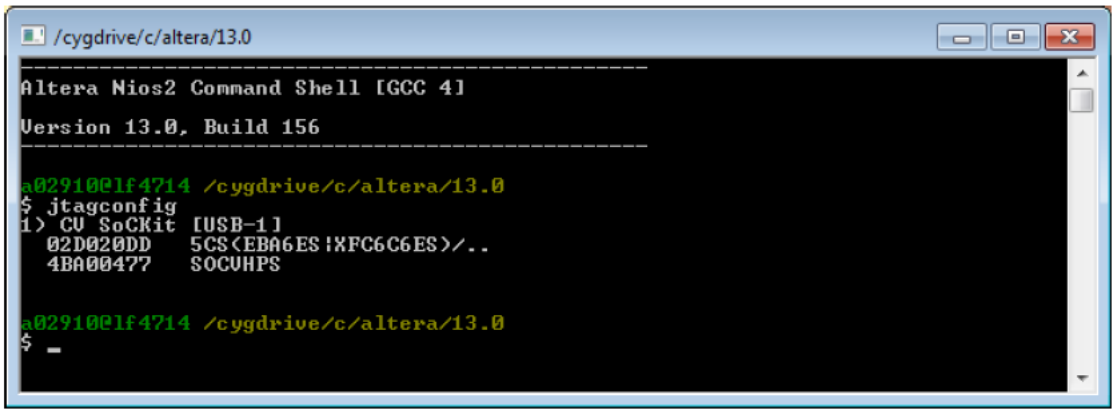


Fig. 94: Devices manager

Two options will be available: "Search automatically for updated driver software" and **"Browse my computer for driver software";** select the second option. **Specify the location of the software controller** ("<altera-folder>/13.0sp1/quartus/drivers/usb-blaster-ii" by default). **Click on Next** and proceed to the installation of the controller software.

Finally, we have to **configure the controller software. Go to the "C:/.../<altera-folder>/13.0sp1/embedded" folder and open a NIOS II Embedded Command Shell. Type "jtagconfig" on the prompt.** If configuration was performed correctly, the device will be displayed in the command shell as in Figure 95.



```
/cygdrive/c/altera/13.0
------------------------------------------------
Altera Nios2 Command Shell [GCC 4]

Version 13.0, Build 156
------------------------------------------------

a02910@lf4714 /cygdrive/c/altera/13.0
$ jtagconfig
1) CV SoCKit [USB-1]
  02D020DD   5CS(EBA6ES|XFC6C6ES)/..
  4BA00477   SOCVHPS


a02910@lf4714 /cygdrive/c/altera/13.0
$ _
```

Fig. 95: Embedded Command Shell in Windows OS

Appendix A: Installing USB Blaster II device driver in Windows OS

Appendix B: VHDL overview

In this appendix we will give a very brief overview of the VHDL code, just the necessary to understand the implementation of our component and its interfaces.

In VHDL code, we have two main unities: the entity and the architecture. The entity lists all the inputs and outputs of the circuit. Its syntax is simple, as seen below:

```
ENTITY ent_name IS
        PORT(   input1: IN STD_LOGIC;
                input2: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                output1, output2: OUT STD_LOGIC;
                output3: OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END ent_name;
```

In the code above, "input1" is described as an input (IN), one bit (STD_LOGIC) signal; "output3" is described as an output (OUT), 16 bits (STD_LOGIC_VECTOR(15 DOWNTO 0)) signal. If there are two or more signals with the same direction (input or output) and the same length (one bit, 8 bits, 16 bits...), they can be declared in one line just separated by a comma, as in the case of "output1" and "output2". Be careful with the last signal, because its line does not end with a semicolon.

The architecture is the unity that describes how the circuit behaves, its function. Its syntax is as seen below:

```
ARCHITECTURE arch_name OF ent_name IS
        (declarations)
BEGIN
        (concurrent sentences)
END arch_name;
```

There may be more than one task in one architecture. Each task has a set of operations that are executed sequentially, but every task are executed in a concurrent way. An architecture with one task is seen in the code example below:

```
ARCHITETURE arch_name OF ent_name IS
BEGIN
        PROCESS
                (declaration of variables)
        BEGIN
                (operations)
        END PROCESS;
END arch_name;
```

We can see the architecture of the "reg16.vhdl" file below, that implements the internal function of our component:

```
ARCHITETURE Behaviour OF reg16 IS
BEGIN
        PROCESS
        BEGIN
                WAIT UNTIL clock'EVENT AND clock = '1';
                IF resetn = '0' THEN
                        Q <= "0000000000000000";
                ELSE
                        Q(15 DOWNTO 0) <= D(15 DOWNTO 0) OR "1000000000000001";
                END IF;
        END PROCESS;
END Behaviour;
```

The structure is quite simple to learn for anyone that has a basic programming knowledge; the only difference is the syntax. The only line we will explain is the following one:

```
WAIT UNTIL clock'EVENT AND clock = '1';
```

The circuit waits until a clock event occurs; this means, when a change in the clock signal occurs (a falling or a rising edge). By adding "AND clock = '1';" we make the circuit to wait a rising edge to continue, because we take into account the value of the clock signal right after the event occurs. Then, when a clock event occurs, the output signal Q is assigned a value as seen in the code.

Let's see now the architecture of the "reg16_avalon_interface.vhdl" file, that implements the external interfaces of our component:

```
ARCHITECTURE Structure OF reg16_avalon_interface IS
        SIGNAL to_reg, from_reg: STD_LOGIC_VECTOR(15 DOWNTO 0);
        COMPONENT reg16
                PORT(   clock, resetn: IN STD_LOGIC;
                        D: IN STD_LOGIC_VECTOR(15 DOWNTO 0);
                        Q: OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
        END COMPONENT;
BEGIN
        to_reg <= writedata;
        reg_instance: reg16 PORT MAP(clock, resetn, to_reg, from_reg);
        readdata <= from_reg;
        Q_export <= from_reg;
END Structure;
```

In this architecture there are declared two SIGNALs. A SIGNAL is like a variable (as it is not an external port of the component, it is not declared as an input or an output) declared before the BEGIN line of an architecture. If a PROCESS modifies it, the SIGNAL becomes updated when the PROCESS executes all its operations and ends. There is also the type VARIABLE; the difference between a VARIABLE and a SIGNAL is that the first one becomes updated in the moment it is modified by a PROCESS, and a VARIABLE is declared within the PROCESS.

There is also a COMPONENT declared, corresponding to our register. A COMPONENT is another entity declared in a different module. A COMPONENT has a very similar syntax to the ENTITY of the module. The entity of the "reg16.vhdl" module is the following:

```
ENTITY reg16 IS
        PORT(   clock, resetn: IN STD_LOGIC;
                D: IN STD_LOGIC_VECTOR(15 DOWNTO 0);
                Q: OUT STD_LOGIC_VECTOR(15 DOWNTO 0));
END reg16;
```

In the architecture of "reg16_avalon_interface" we reference our reg16 component and we assign some signals to its ports: we assign "to_reg" to the "D" input, and "from_reg" to the "Q" output. The signal "to_reg" is used to pass to the register the data that we write to it, and "from_reg" is used to get from the register the data we read from it (they are like intermediates between the "reg16" ports and the "reg16_avalon_interface" ports).

The rest of the architecture code has no secrets to us, only some assignations to the signals and ports of the "reg16_avalon_interface" module. Just one last tip: in order to use the STD_LOGIC and STD_LOGIC_VECTOR type of signals, the following two lines must be added before the entity code to import the corresponding library:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

Appendix B: VHDL overview

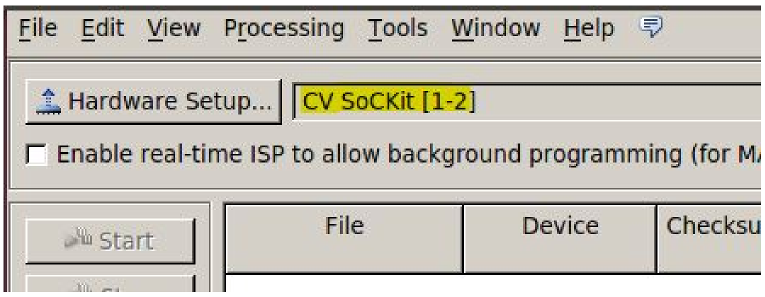
Appendix C: Programming the FPGA

To program the FPGA there is an Altera tool, Quartus Programmer. If not installed, access to the Quartus II Web Edition download page[d3], in the "Individual Files" tab choose the "Quartus II Programmer and SignalTap II" from the "Additional Software" section. Install it with the options by default. Execute the following commands from a terminal to execute this tool in Ubuntu:

```
$ cd ~/altera/13.0sp1/quartus/bin
$ sudo ./quartus_pgmw
```

By using these commands, we go to the folder that contains the script which loads Quartus Programmer, and we execute it. Another option to open it is from Quartus II, selecting "Programmer" from the Tools menu.

Once the program is open, if the SoCkit board is connected there must be selected "CV SoCkit [1-2]" as hardware in the higher part of the Quartus Programmer screen (Figure 96).

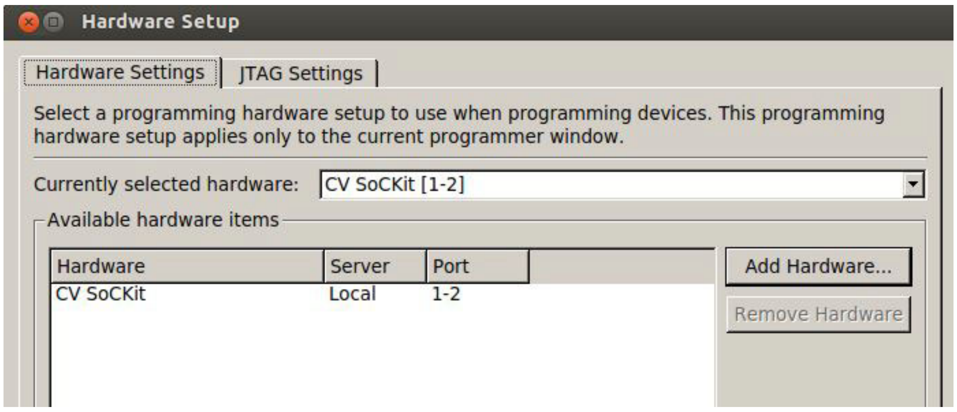


CV SoCKit [1-2]

Enable real-time ISP to allow background programming (for M

Fig. 96: Hardware connection from Quartus Programmer

On the contrary, click on "Hardware Setup...". In the new opened window, select "Currently selected hardware: CV SoCkit [1-2]" (Figure 97). Close the window.



Select a programming hardware setup to use when programming devices. This programming hardware setup applies only to the current programmer window.

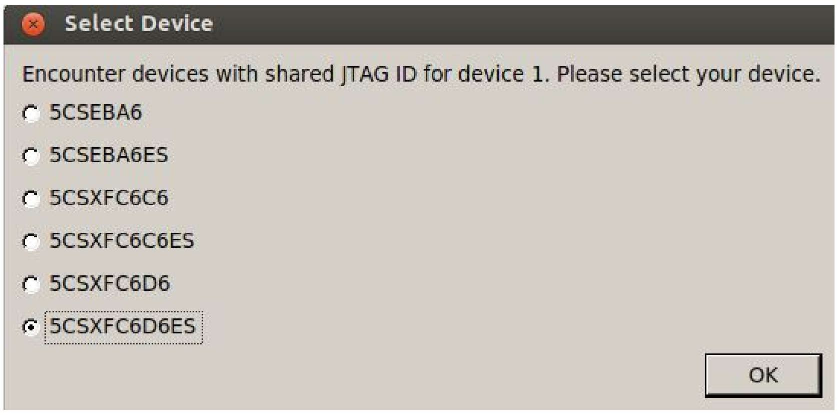
Currently selected hardware: CV SoCKit [1-2]

Available hardware items

Hardware	Server	Port
CV SoCKit	Local	1-2

Fig. 97: Hardware Setup in Quartus Programmer

Being the hardware selected, **click on the "Auto Detect" button in the left column of the Quartus Programmer screen. In the emerging window (Figure 98), select the option "5CSXFC6D6ES",** corresponding to the FPGA of the SoCkit board, "Cyclone V SX SoC FPGA". **Click OK** to choose it.

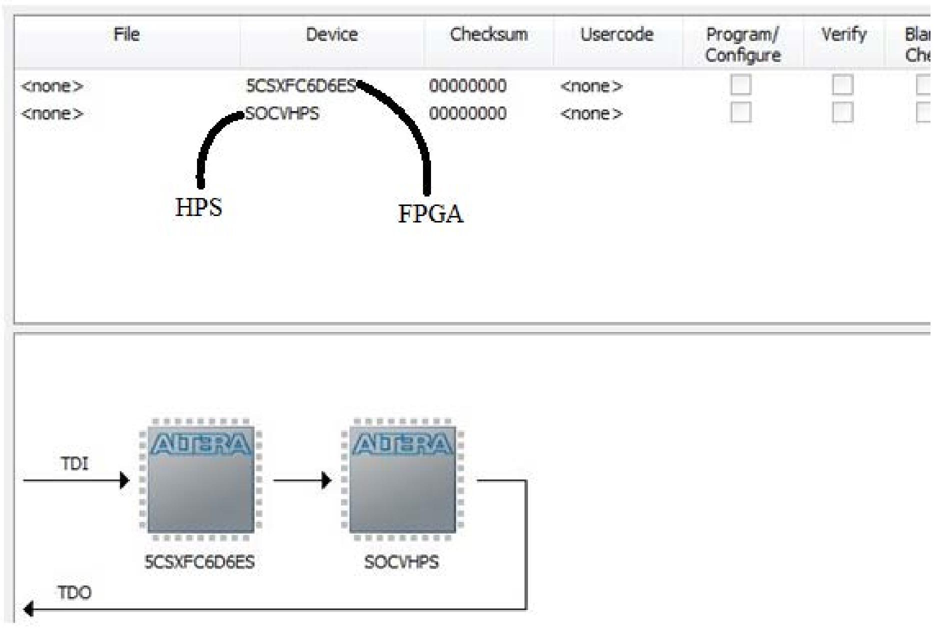

Select Device

Encounter devices with shared JTAG ID for device 1. Please select your device.

- 5CSEBA6
- 5CSEBA6ES
- 5CSXFC6C6
- 5CSXFC6C6ES
- 5CSXFC6D6
- [✓ 5CSXFC6D6ES]

Fig. 98: Select device in Quartus Programmer

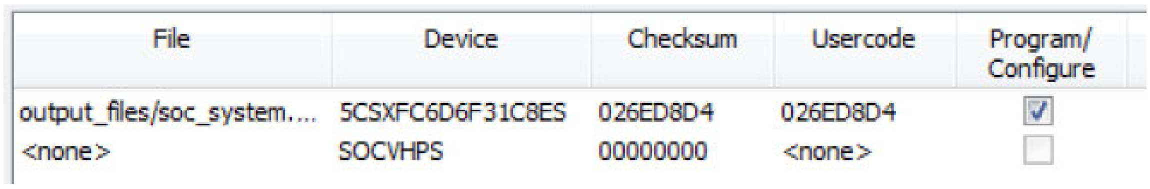
We will see the Quartus Programmer as in Figure 99. There are two different devices, related to the FPGA and the HPS of the Cyclone V SoC.



File	Device	Checksum	Usercode	Program/ Configure	Verify	Bla Che
<none>	5CSXFC6D6ES	00000000	<none>	☐	☐	☐
<none>	SOCVHPS	00000000	<none>	☐	☐	☐

Fig. 99: SoCkit devices in Quartus Programmer

Select the FPGA device (5CSXFC6D6ES). In the left column, click on "Add file..." and search for the "soc_system.sof" file, located in ".../GHRD_soc_system /output_files". This is the configuration file of Quartus II which programs the FPGA. After selecting this file, the list of devices should look as in Figure 100.



File	Device	Checksum	Usercode	Program/ Configure
output_files/soc_system....	5CSXFC6D6F31C8ES	026ED8D4	026ED8D4	☑
<none>	SOCVHPS	00000000	<none>	☐

Fig. 100: FPGA in Quartus Programmer ready to be programmed

As we can see, the first device in the list is the complete reference number of the Cyclone V SoC, 5CSXFC6D6F31C8ES. It is printed on the Cyclone V SoC package (Figure 101). If three devices are displayed, delete the extra 5CSXFC6D6ES device (click on the device to select it and then click Delete in the left column).

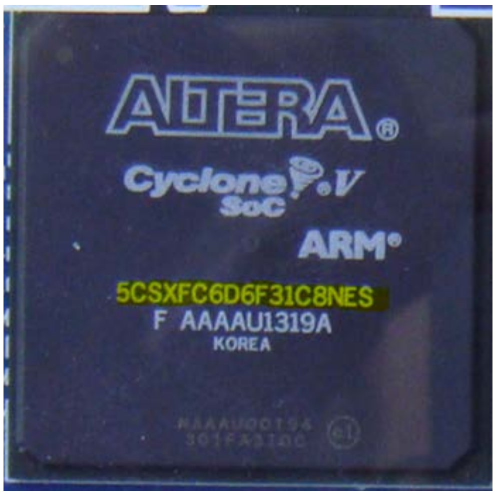
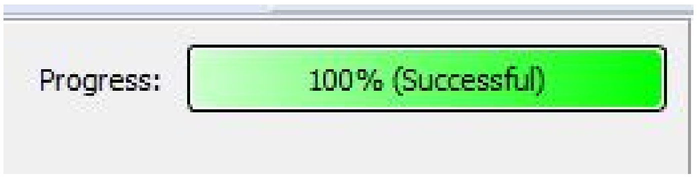


Fig. 101: Reference number of the Cyclone V SoC

Finally, make sure to mark the box "Program/Configure" of the 5CSXFC6D6F31C8ES device. Click on the "Start" button in the left column to program the FPGA. If everything goes well, we will observe at the higher right corner of the screen the programming progress and if it was successful or not.



Progress: 100% (Successful)

Fig. 102: The FPGA was successfully programmed

Appendix C: Programming the FPGA

Appendix D: Connection to the SoCkit and booting Linux

To establish communication with the SoCkit board, we have to use PuTTY. If not installed previously in the Ubuntu virtual machine, download it with the following command:

```
$ sudo apt-get install putty
```

Once installed, **use the following command to execute it** (always add 'sudo' at the beginning to avoid permission problems that may cause unexpected behavior):

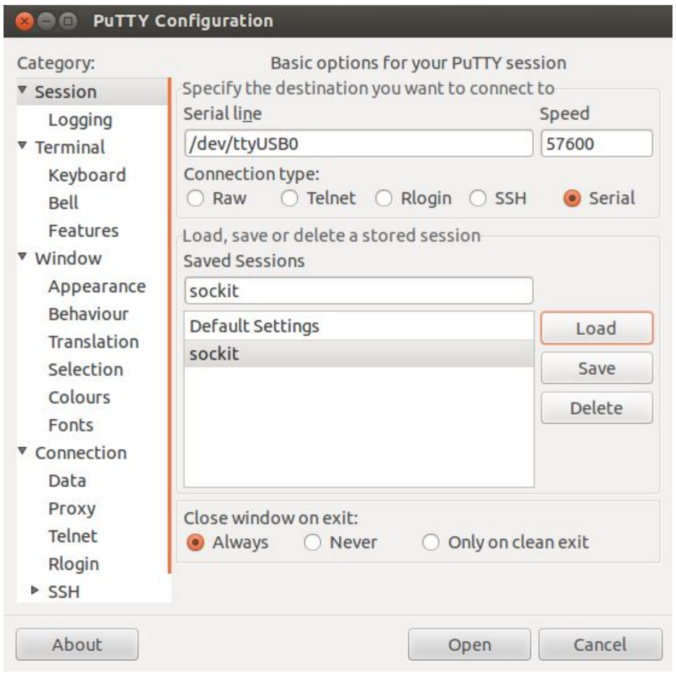
```
$ sudo putty
```

Ubuntu incorporates drivers for a USB-serial converter chip included in the SoCkit board, allowing the computer to see the board as a virtual serial port, usually named "/dev/ttyUSB*", being * a 0, 1, etc (in our case, the device is /dev/ttyUSB0). **To check which USB serial device is installed, use the following command:**

```
$ ls /dev/ttyUSB*
```

Next, configure the PuTTY terminal:

- Connection type: serial
- Speed: 57600
- Serial line: /dev/ttyUSB0


PuTTY Configuration

Basic options for your PuTTY session

Specify the destination you want to connect to

Serial line: /dev/ttyUSB0

Speed: 57600

Connection type: Raw Telnet Rlogin SSH [✓ Serial]

Load, save or delete a stored session

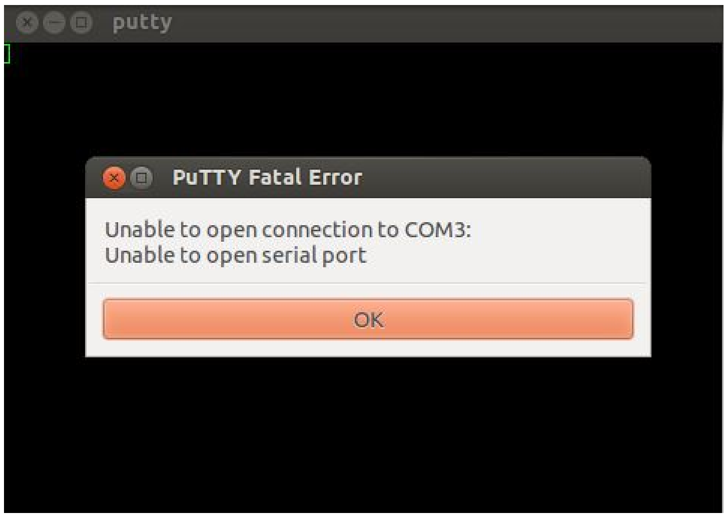
Saved Sessions: sockit

Default Settings
sockit

Close window on exit: [✓ Always] Never Only on clean exit

Fig. 103: Configuring PuTTY terminal to connect to the SoCkit board

A window like in Figure 104 may be shown; it tells that the serial port cannot be accessed.



Unable to open connection to COM3:
Unable to open serial port

Fig. 104: Failed connection to the SoCkit

The cause of this message is the lack of permissions the user has over the serial port. To change the permissions, use the following command:

```
$ sudo chmod 777 /dev/ttyUSB0
```

This way, the serial port can be accessed to view the console, so the problem is solved.

To reboot the Linux OS, press the warm reset button situated on the lower left corner of the SoCkit board, as in Figure 105.

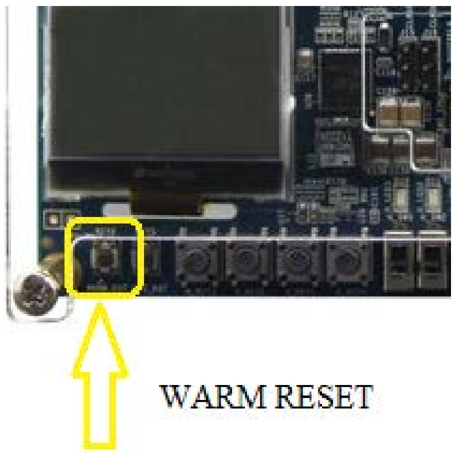


Fig. 105: Warm reset in SoCkit board