I2A2
Universidad Politécnica de Madrid

Embedded Systems

Using Quartus and Buildroot for building Embedded Linux Systems (De1-SOC) V1.9

Mariano Ruiz
Antonio Carpeño

2017

Universidad Politécnica de Madrid

Special mentions and recognitions.

This document contains some paragraphs in chapter 2 and 3 copied and adapted from the document created by Sahand Kashani-Akhavan and René Beuchat from the École polytechnique fédérale de Lausanne (EPHL) available at https://github.com/sahandKashani/.

Table of contents

Table of figures

1 INTRODUCTION

1.1 Document Overview

This document describes the basic steps to setup and embedded Linux-based system using the Terasic DE1_SoC board (DE1). The document has been specifically written to use a DE1-SOC development system based on the Cyclone V SoC. All the software elements used to build the Linux distribution have a GPL license. You carefully have to read all the instructions before executing the practical part otherwise, you will find problems and probably unpredicted errors. In parallel, you need to review the slides available at Moodle site.

1.2 Acronyms

ACP	Accelerator Coherency Port
CAN	Controller Area Network
CPU	Central Processing Unit
DE1	DE1-SoC
EABI	Extended Application Binary Interface
EHCI	Enhanced Host Controller Interface
EMAC	Ethernet Media Access Controller
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input Output
HPS	Host Processing System
I2C	Inter-Integrated Circuit)
I/O	Input and Output
MMC	Multimedia card
MOSI	Master Output Slave Input
MPU	Memory Protection Unit
NAND	Flash memory type for fast sequential read and write
PCI	Peripheral Component Interconnect – computer bus standard
PCI Express	Peripheral Component Interconnect Express
OS	Operating system
SD/MMC	Secure Digital/Multimedia Card
SPI	Serial Peripheral Interface
TCL	Tool Command Language
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus

1.3 **Bibliography**

[1] Altera Corporation, “Cyclone V Device Handbook, Volume 3: Hard Processor System Technical Reference Manual,” 31 July 2014. [Online]. Available: http://www.altera.com/literature/hb/cyclone-v/cv_5v4.pdf.

[2] Terasic Technologies, “Terasic - DE Main Boards - Cyclone - DE1-SoC Board,” [Online]. Available: http://de1-soc.terasic.com.

[3] Altera, “HPS SoC Boot Guide - Cyclone V SoC Development Kit,” Altera, 27 1 2016. [Online]. [Accessed 12 2016].

[4] M. Ruiz and A. Carpeño, “https://github.com/mruizglz/UPM-ES-cyclonevbsp,” UPM, 12 1 2017. [Online].

[5] Terasic Technologies, [Online]. Available: https://github.com/sahandKashani/DE1-SoC/blob/master/Documentation/DE1-SoC%20Schematic.pdf.

[6] Integrated Silicon Solution, Inc, [Online]. Available: http://www.issiusa.com/pdf/43TR16256A-85120AL.pdf.

[7] P. Software, “Win32DiskImager sourceforge,” [Online]. Available: http://sourceforge.net/projects/win32diskimager/.

[8] Intel Altera, “Avalon MM Specification,” Altera, 2015. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf.

[9] CARL, “Altera USB-Blaster with Ubuntu 14.04,” [Online]. Available: http://www.fpga-dev.com/altera-usb-blaster-with-ubuntu/.

[10] GNU, “Getopt-long-options,” [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Getopt-Long-Options.html.

[11] S. Kashani-Akhavan, “pin_assignment_DE1_SoC.tcl,” [Online]. Available: https://github.com/sahandKashani/Altera-FPGA-top-level-files/blob/master/DE1-SoC/pin_assignment_DE1_SoC.tcl.

[12] “DE1_SoC_top_level.vhd,” [Online]. Available: https://github.com/sahandKashani/Altera-FPGA-top-level-files/blob/master/DE1-SoC/DE1_SoC_top_level.vhd.

[13] Free Electrons, “Embedded Linux system development.,” Free Electrons, 2016. [Online]. Available: http://free-electrons.com/training/embedded-linux/.

[14] C. Simmonds, Mastering Embedding Linux Programming, Birmingham: Packt Publishing, 2015.

[15] The Ubuntu Manual Team, [Online]. Available: http://files.ubuntu-manual.org/manuals/getting-started-with-ubuntu/12.04/en_US/screen/Getting Started with Ubuntu 12.04.pdf.

1.4 Prerequisites

1.4.1 Hardware.

This tutorial requires the use of the Terasic DE1-SoC board and a personal computer.

1.4.2 Software

To execute this lab properly, you need the following elements:

1. VMware WorkStation player version 12.0 or above. Available at www.wmware.com (free download and use). This software is already installed in the laboratory desktop computer.
2. A VMWare virtual machine with Ubuntu 14.04 and all the software packages installed is already available on the Desktop. This virtual machine is available for your personal use at the Department assistance office (preferred method). If you want to setup your virtual machine by yourself, follow the instructions of Annex I.
3. These software tools installed on the host computer or the Virtual Machine:
    - *Quartus Prime (16.0)*
    - *ModelSim-Altera*
    - *SoC Embedded Design Suite (SoC EDS)*

1.4.3 Other recommendations

1. You need a basic knowledge of Linux commands. A UNIX tutorial for beginners is available at in this URL http://www.ee.surrey.ac.uk/Teaching/Unix/.
2. ***SOME*** command-line instructions provided in this guide ***MUST*** be executed in an ***ALTERA EMBEDDED COMMAND SHELL***. The file for the *Altera Embedded Command Shell* can be found at "`<altera_install_directory>/<version>/embedded/embedded_command_shell.sh`"

1.4.3.1 Licenses

- No license are required for the execution of the different laboratories proposed in this tutorial.

2 CYCLONE V SOC OVERVIEW

This section describes some features of the Cyclone V family of devices. All information below, along with a complete documentation regarding this family can be found in the Cyclone V Device Handbook [1].

2.1 Introduction

The development of embedded systems based on chips containing one or more microprocessors and hardcore peripherals, as well as an FPGA part is becoming more and more important. The manufacturers are providing solutions known as Systems on Chip that includes these advanced digital solutions. SoC-based technology gives the designer a lot of freedom and powerful abilities. He can now design specific accelerators to significantly improve algorithms, or create specific programmable interfaces with the external world. Two main HDL (**H**ardware **D**esign **L**anguage) languages are available for the design of the FPGA part: **VHDL** and Verilog. Additionally, there also exist other tools that perform automatic translations from C to HDL. New emerging technologies like OpenCL allow compatibility between high-level software design and low-level hardware implementations such as:

- Compilation for single or multicore processors
- Compilation for GPUs (Graphical Processing Unit)
- Translation and compilation for FPGAs. The latest FPGAs-based devices use a PCIe interface or some other way of parameters passing between the main processor and the FPGA.

Altera and XILINX are the most important manufacturers of SoCs including processors and FPGA. Next paragraph present the key features of the Cyclone V SoC.

2.2 Introduction to the Cyclone V Hard Processor System

The Cyclone V device is a single-die system on a chip (SoC) that consists of two distinct parts – a hard processor system (HPS) portion and an FPGA portion (see Fig. 1).

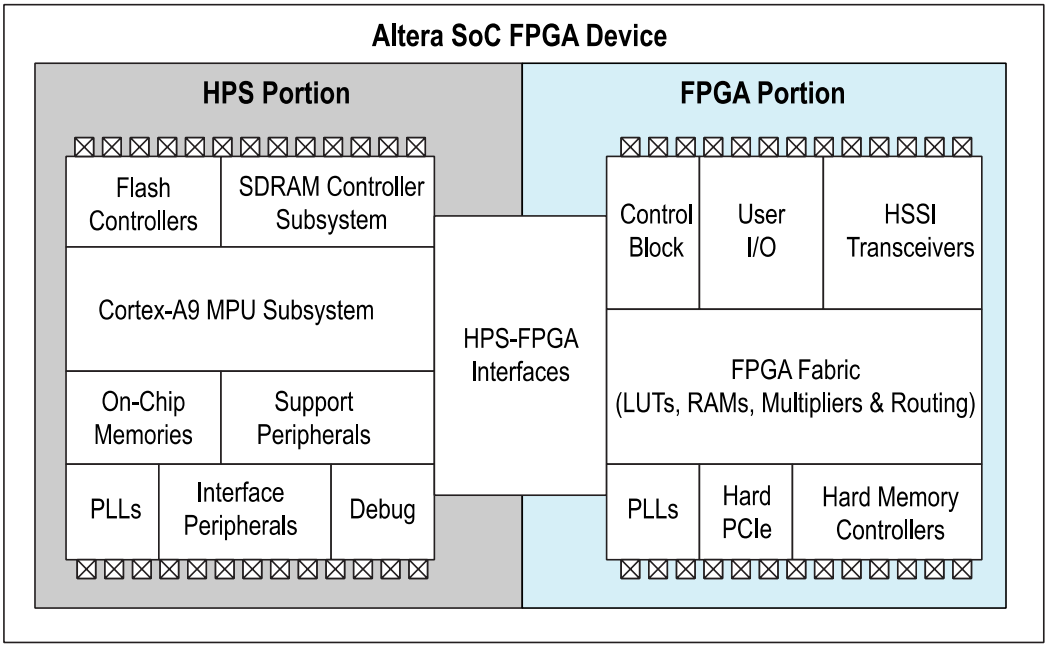


Fig. 1: Altera SoC FPGA Device Block Diagram [1, pp. 1-1]

The HPS contains a microprocessor unit (MPU) subsystem with single or dual ARM Cortex-A9 MPCore processors, flash memory controllers, SDRAM L3 Interconnect, on-chip memories, support peripherals, interface peripherals, debug capabilities, and phase-locked loops (PLLs). The dual-processor HPS supports symmetric (SMP) and asymmetric (AMP) multiprocessing.

The FPGA portion of the device contains the FPGA fabric, a control block (CB), phase-locked loops (PLLs), and depending on the device variant, high-speed serial interface (HSSI) transceivers, hard PCI Express (PCIe) controllers, and hard memory controllers. The HPS and FPGA portions of the device are distinctly different. The HPS can boot the software from the FPGA fabric; the external flash, or JTAG. In contrast, the FPGA must be configured either through the HPS, or an externally supported device such as the *Quartus Prime* programmer. The MPU subsystem can boot from flash devices connected to the HPS pins, or from memory available on the FPGA portion of the device (when the FPGA portion is previously configured by an external source). The HPS and FPGA portions of the device each have their own pins. Pins are not freely shared between the HPS and the FPGA fabric. The FPGA I/O pins are configured by an FPGA configuration image through the HPS or any external source supported by the device. The HPS I/O pins are configured by software executing in the HPS. A specific software running on the HPS accesses control registers in the Cyclone V system manager to assign HPS I/O pins to the available HPS modules. The software that configures the HPS I/O pins is called the preloader. The HPS and FPGA portions of the device have separate external power supplies and are power on independently. You can power on the HPS without powering on the FPGA side of the device. However, to power on the FPGA portion, the HPS must already be on or powered on at the same time as the FPGA portion. Table 1 summarizes the possible configurations.

Table 1: Possible HPS and FPGA Power Configurations

HPS Power	FPGA Power
On	On
On	Off
Off	Off

2.3 Features of the HPS

Fig. 2 represents the block diagram of HPS system with the interface with the FPGA portion through the bridges. Some of these blocks can be a bus master. Masters takes control of the bus for different data transfers. The bus master capability is implemented by:

- MPU subsystem is featuring dual ARM Cortex-A9 MPCore processors. This includes an interrupt controller., one general-purpose timer, one watchdog timer and one Memory management unit (MMU) per processor. The HPS masters the L2 cache memory, the L3 interconnect and the SDRAM controller subsystem.
- General-purpose Direct Memory Access (DMA) controller.
- Two Ethernet media access controllers (EMACs)
- Two USB 2.0 On-The-Go (OTG) controllers
- NAND flash controller
- Secure Digital (SD) / MultiMediaCard (MMC) controller
- Two serial peripheral interface (SPI) master controllers
- ARM CoreSight debug components

The slaves devices included in the system are the following:

- Quad SPI flash controller
- Two SPI slave controllers
- Four inter-integrated circuit (I^2C) controllers
- 64 KB on-chip RAM
- 64 KB on-chip boot ROM
- Two UARTs
- Four timers
- Two watchdog timers
- Three general-purpose I/O (GPIO) interfaces

- Two controller area network (CAN) controllers
- System manager
- Clock manager
- Reset manager
- Scan manager
- FPGA manager

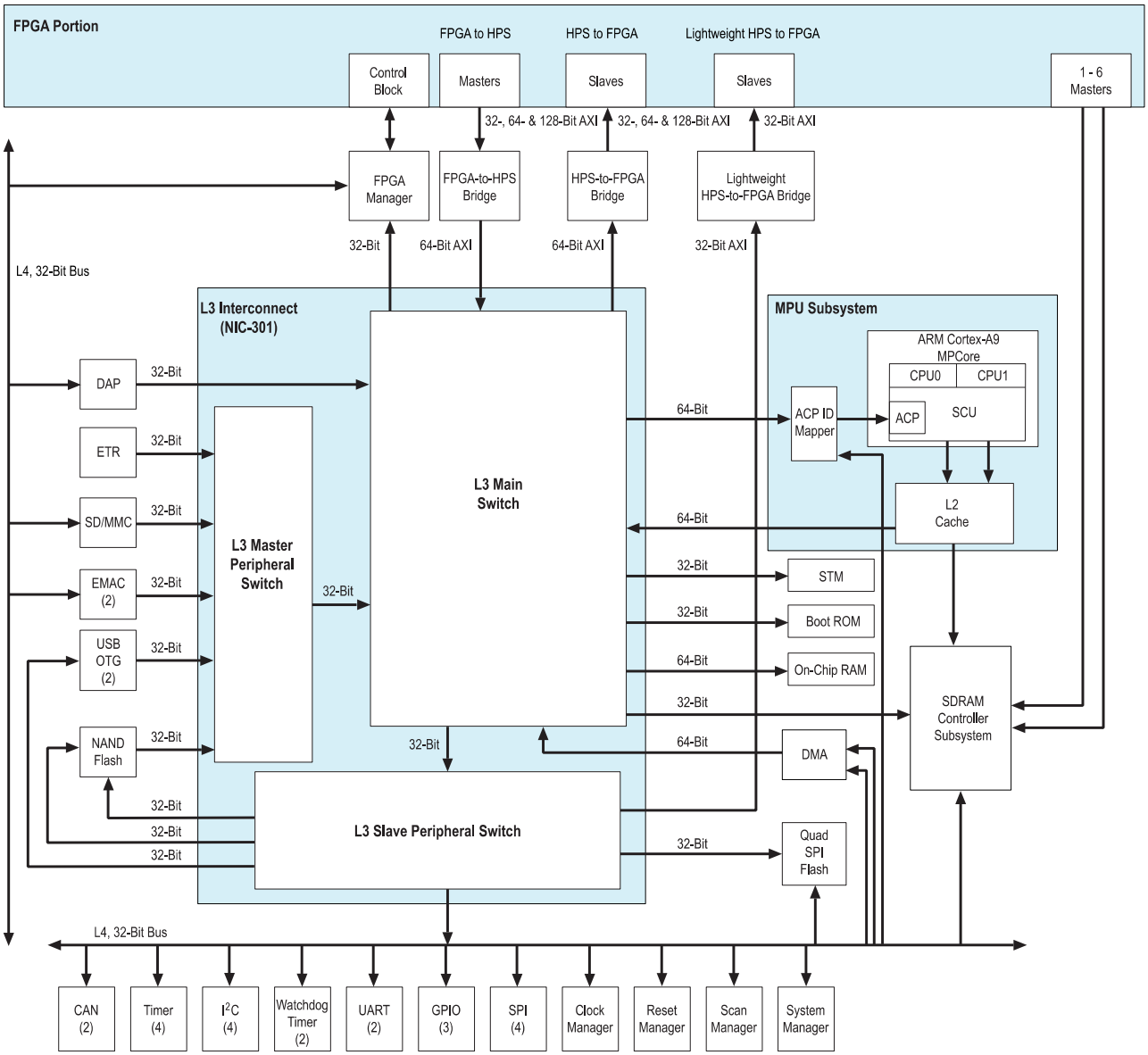


Fig. 2: HPS Block Diagram [1, pp. 1-3]

The HPS hardware component needs a piece of software that configures the different options provided by the HPS hardware. This does not mean that the HPS is a softcore processor. As such, the HPS component has a *small footprint* in the FPGA fabric, as its only purpose is to connect the soft and hard logic together. Therefore, it is possible to use the Cyclone V SoC in 3 different configurations:

- FPGA-only
- HPS-only
- HPS & FPGA

The following paragraphs describe some basic details of the different hardware elements included in the HPS.

2.3.1 SDRAM Controller Subsystem

The SDRAM controller subsystem is mastered by HPS masters and FPGA fabric masters. It supports DDR2, DDR3, and LPDDR2 devices. It is composed of 2 parts:

- SDRAM controller
- DDR PHY (interfaces the single port memory controller to the HPS I/O)

2.3.2 Support Peripherals

2.3.2.1 System Manager

The system manager is an essential HPS component. It offers a few important features:

- Pin multiplexing (term used for the software configuration of the HPS I/O pins by the preloader software)
- Freeze controller that places I/O elements into a safe state for configuration
- Low-level control of peripheral features not accessible through the control and status registers (CSRs)

2.3.2.2 FPGA Manager

The FPGA manager offers the following features:

- Manages the configuration of the FPGA portion of the device
- Monitors configuration-related signals in the FPGA
- Provides 32 general-purpose inputs and 32 general-purpose outputs to the FPGA fabric

2.3.3 Interface Peripherals

2.3.3.1 GPIO Interfaces

The HPS provides three GPIO interfaces and offers the following features:

- Supports digital de-bounce
- Configurable interrupt mode
- Supports up to 71 I/O pins and 14 input-only pins, based on device variant
- Supports up to 67 I/O pins and 14 input-only pins

2.3.4 On-Chip Memory

The following on-chip memories are different from any on-chip memories located in the FPGA fabric.

2.3.4.1 On-Chip RAM

The on-chip RAM offers the following features:

- 64 KB size
- High performance for all burst lengths

2.3.4.2 Boot ROM

The boot ROM offers the following features:

- 64 KB *size* (the code in the boot ROM cannot be changed)
- Contains the code required to support HPS boot from cold or warm reset
- Used exclusively for booting the HPS

2.4 DE1-SoC: Development board using a Cyclone V

Before continung explaining more details of the Cyclone V SoC, it is interesting to introduce the development system used in this tutorial. There is an ecosystem of hardware platforms supporting the use of SoC with the most common software tools provided by ALTERA and XILINX. One of this is the Terasic DE1-SoC board (see Fig. 3). The DE1-SoC board has many features that allow users to implement a wide range of electronic embedded systems. See Table 2 for details.

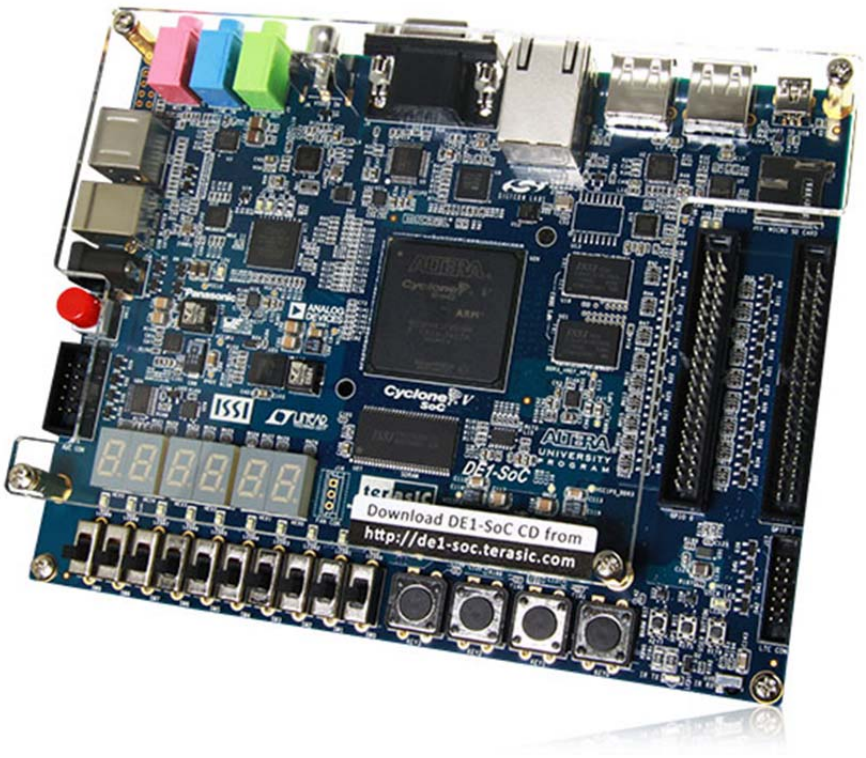


Fig. 3: Terasic DE1-SoC Board [2]

Table 2: Main features of DE1-SoC

Feature	Details
FPGA Device	Cyclone V SoC 5CSEMA5F31C6 Device Dual-core ARM CORTEX-A9 (HPS) . 85K Programmable Logic Elements 4450 Kbits embedded memory 6 Fractional PLLs 2 Hard Memory Controllers (only seems to be used for the HPS DDR3 SDRAM, not the FPGA SDRAM)
Configuration and Debug	Quad Serial Configuration device – *EPCQ256* on FPGA On-Board *USB BLASTER II* (Normal type B USB connector)
Memory Device	64 MB (32Mx16) SDRAM on FPGA 1 GB (2x256Mx16) DDR3 SDRAM on HPS Micro SD Card Socket on HPS
Communication	Two Port USB 2.0 Host (ULPI interface with USB type A connector) USB to UART (micro USB type B connector) 10/100/1000 Ethernet PS/2 mouse/keyboard IR Emitter/Receiver
Connectors	Two 40-pin Expansion Headers One 10-pin ADC Input Header One LTC connector (One Serial Peripheral Interface (SPI) Master, one I2C and

	one GPIO interface)
Display	24-bit VGA DAC
Audio	24-bit CODEC, line-in, line-out, and microphone-in jacks
Video Input	TV Decoder (NTSC/PAL/SECAM) and TV-in connector
ADC	Fast throughput rate: 1 MSPS Channel number: 8 Resolution: 12 bits Analog input range: 0 ~ 2.5 V or 0 ~ 5V as selected via the RANGE bit in the control register
Switches, Buttons, and Indicators	4 User Keys (FPGA x4) 10 User switches (FPGA x10) 11 User LEDs (FPGA x10; HPS x 1) 2 HPS Reset Buttons (HPS_RST_n and HPS_WARM_RST_n) Six 7-segment displays
Sensors	G-Sensor on HPS
Power	12V DC input

The block diagram identifying the hardware elements available in the De1-SoC device is represented in Fig. 4. Fig. 5 and Fig. 6 show the pictures of the PCBs of this product. Fig. 6 also contains some graphical details indicating where the peripheral are connected to, HPS or FPGA.

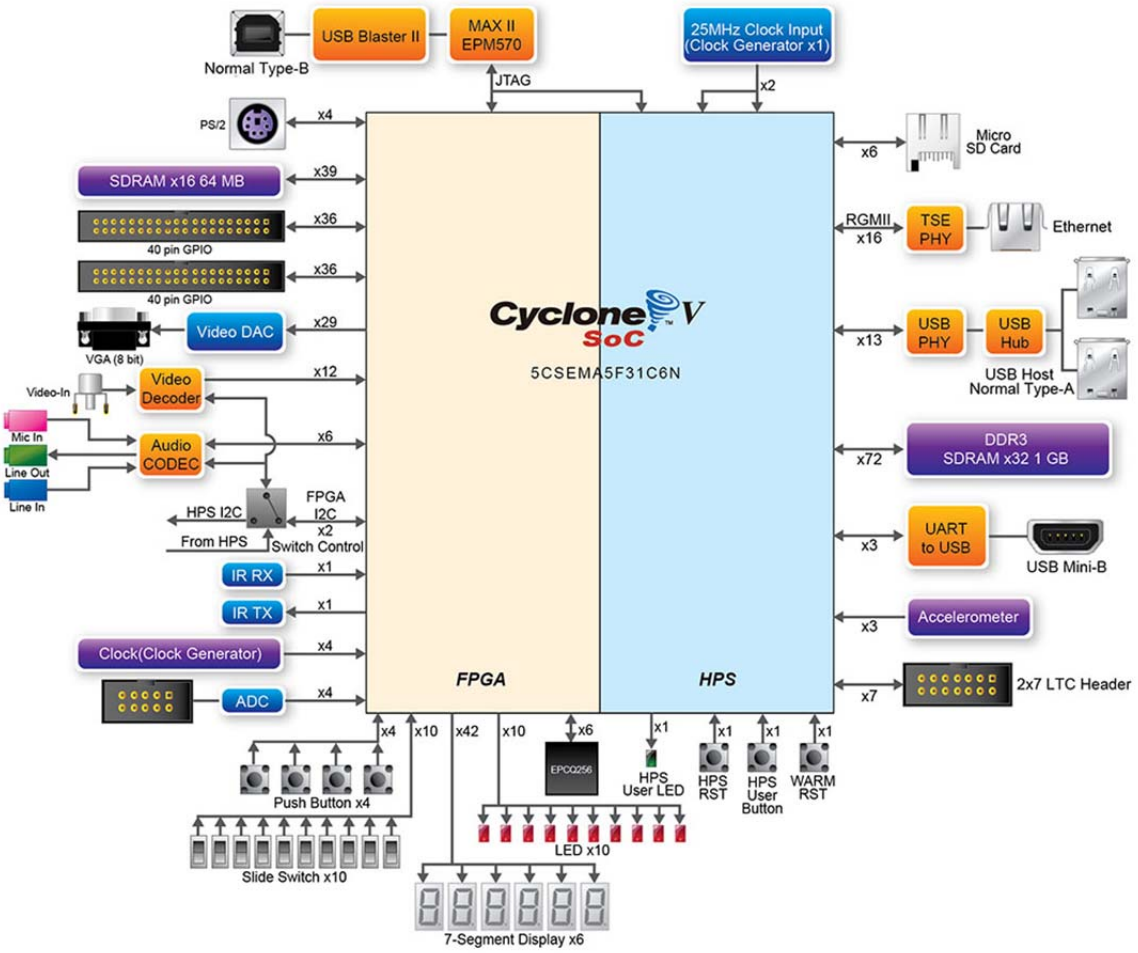


Fig. 4: Block Diagram of the DE1-SoC Board [2]

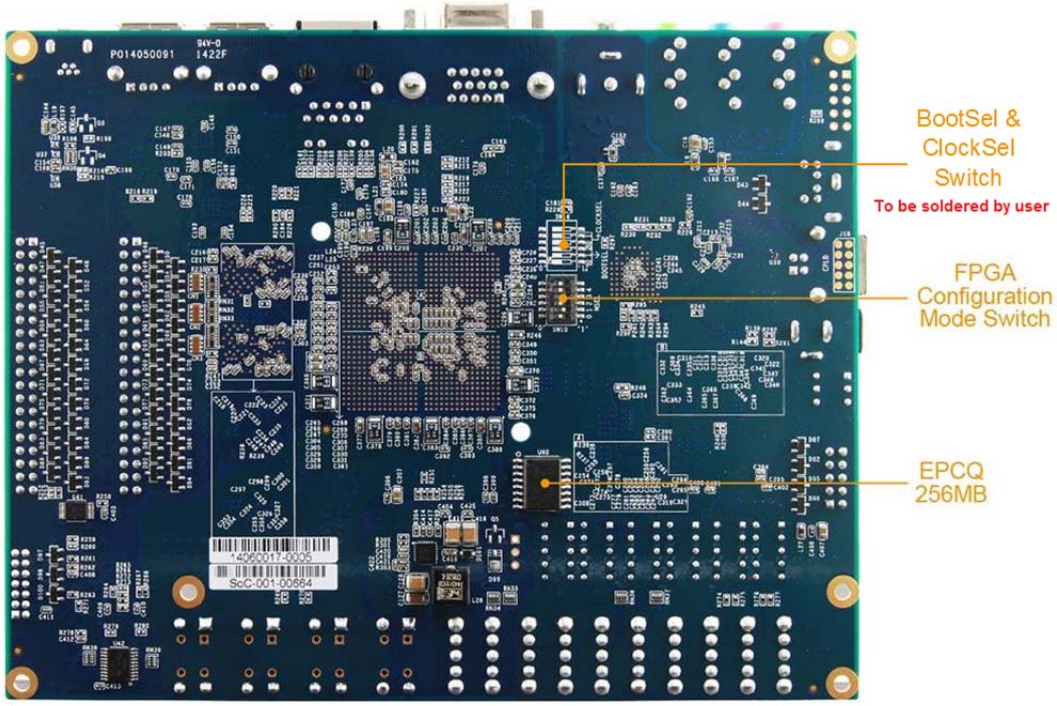


Fig. 5: Back layout details [2].

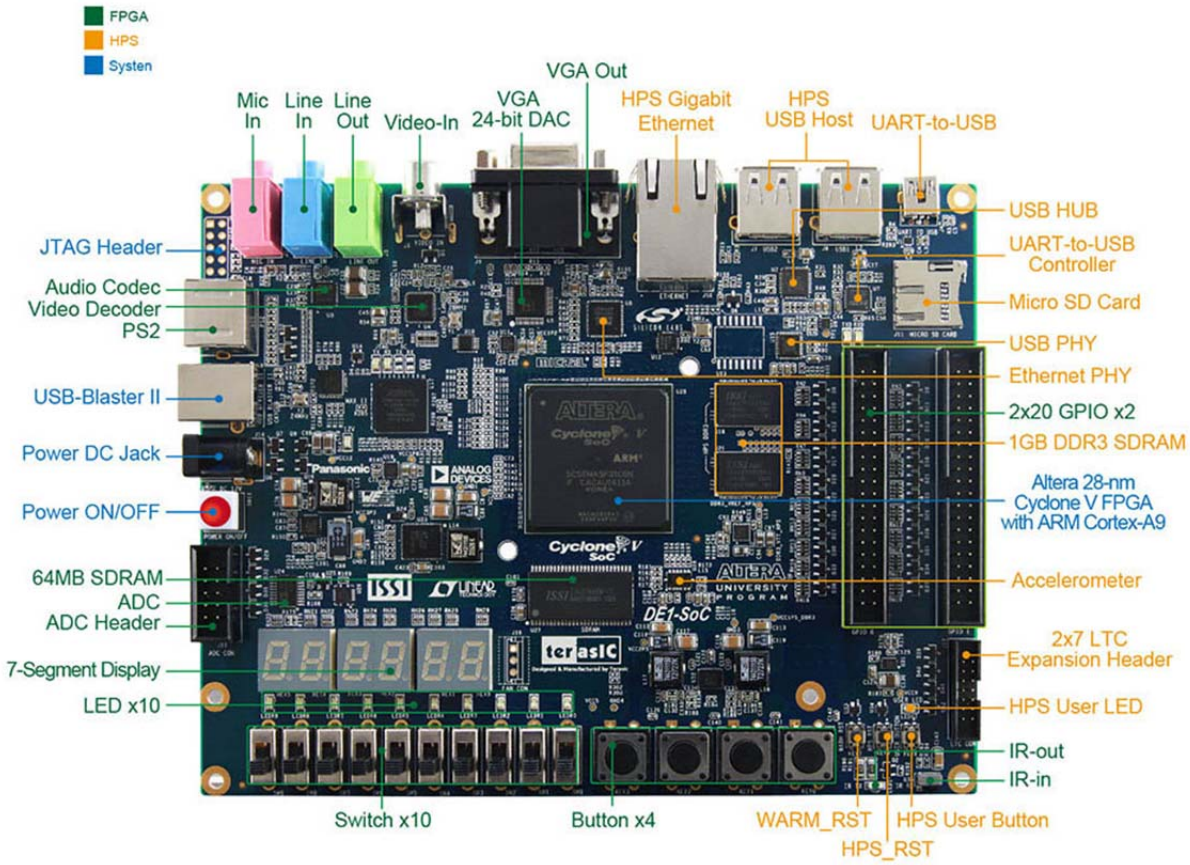


Fig. 6: Front layout details [2]. Green for peripherals directly connected to the FPGA Orange for peripherals directly connected to the HPS Blue for board control

2.5 HPS-FPGA Interfaces in Cyclone V

The HPS-FPGA interfaces provide a variety of communication channels between the HPS and the FPGA fabric. The HPS-FPGA interfaces include:

- **FPGA-to-HPS bridge** – a high-performance bus with a configurable data width of 32, 64, or 128 bits. It allows the FPGA fabric to master transactions to slaves in the HPS. This interface allows the FPGA fabric to have full visibility into the HPS address space.
- **HPS-to-FPGA bridge** – a high-performance bus with a configurable data width of 32, 64, or 128 bits. It allows the HPS to master transactions to slaves in the FPGA fabric. Sometimes this is called "*heavyweight*" HPS-to-FPGA bridge to distinguish its "*lightweight*" counterpart (see below).
- **Lightweight HPS-to-FPGA bridge** – a bus with a 32-bit fixed data width. It allows the HPS to master transactions to slaves in the FPGA fabric.
- **FPGA manager interface** – signals that communicate with FPGA fabric for boot and configuration.
- **Interrupts** – allow to supply interrupts directly to the MPU interrupt controller.
- **HPS debug interface** – an interface that allows the HPS debug control domain to extend into the FPGA.

2.6 HPS Address Map

2.6.1 HPS Address Spaces

The HPS address map specifies the address of slaves, such as memory and peripherals, as viewed by the HPS masters. The HPS has three address spaces:

Table 3: HPS Address Spaces [1, pp. 1-13]

Name	Description	Size
MPU	MPU subsystem	4 GB
L3	L3 interconnect	4 GB
SDRAM	SDRAM controller subsystem	4 GB

Fig. 7 shows the relationships between the different HPS address spaces. The figure is not to scale.

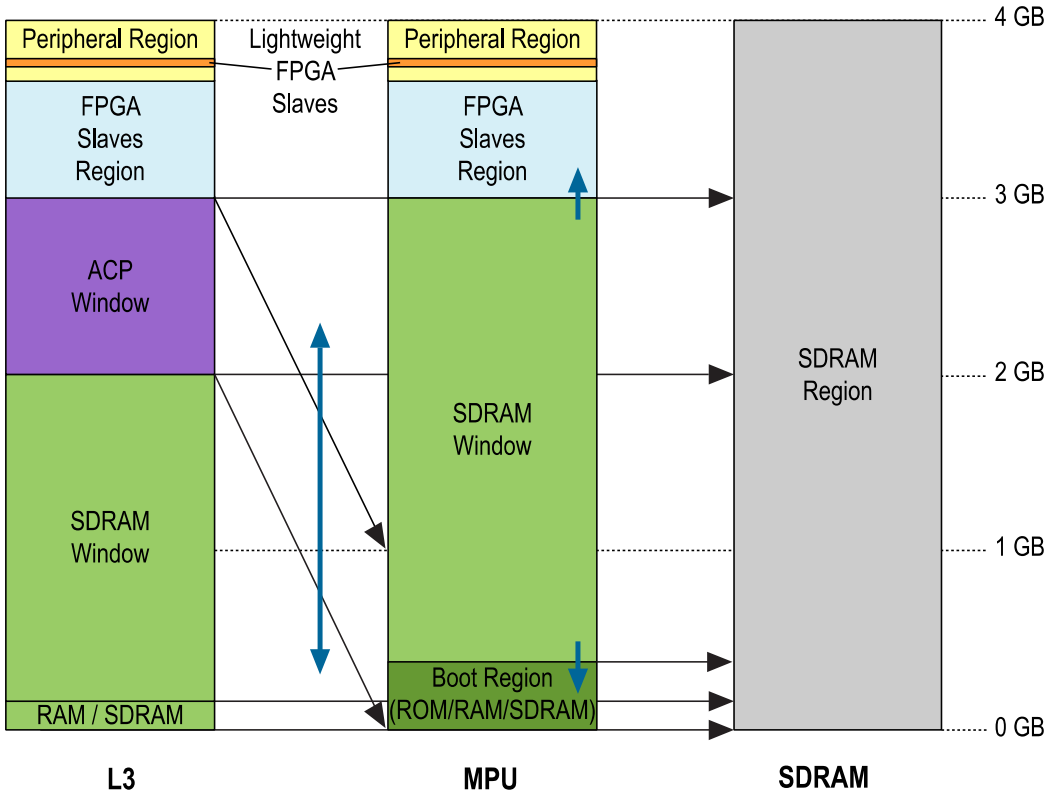


Fig. 7: HPS Address Space Relations [1, pp. 1-14]

The window regions provide access to other address spaces. The thin black arrows indicate which address space is accessed by a window region (arrows point to accessed address space).
The SDRAM window in the MPU can grow and shrink at the top and bottom (short blue vertical arrows) at the expense of the FPGA slaves and boot regions. The ACP window can be mapped to any 1 GB region in the MPU address space (blue vertical bidirectional arrow), on gigabyte-aligned boundaries.
Table 4 shows the base address and size of each region that is common to the L3 and MPU address spaces.

Table 4: Common Address Space Regions [1, pp. 1-15]

Region Name	Description	Base Address	Size
FPGA slaves	FPGA slaves connected to the HPS-to-FPGA bridge	0xC0000000	960 MB
HPS peripherals	Slaves directly connected to the HPS (corresponds to all orange colored elements on Fig. 6)	0xFC000000	64 MB
Lightweight FPGA slaves	FPGA slaves connected to the lightweight HPS-to-FPGA bridge	0xFF200000	2 MB

2.6.2 HPS Peripheral Region Address Map

Table 5 lists the slave identifier, slave title, base address, and size of each slave in the HPS peripheral region. The Slave Identifier column lists the names used in the HPS register map file provided by Altera (more on this later).

Table 5: HPS Peripheral Region Address Map [1, pp. 1-16]

Slave Identifier	Slave Title	Base Address	Size
STM	STM	0xFC000000	48 MB
DAP	DAP	0xFF000000	2 MB
LWFPGASLAVES	FPGA slaves accessed with lightweight HPS-to-FPGA bridge	0xFF200000	2 MB
LWHPS2FPGAREGS	Lightweight HPS-to-FPGA bridge GPV	0xFF400000	1 MB
HPS2FPGAREGS	HPS-to-FPGA bridge GPV	0xFF500000	1 MB
FPGA2HPSREGS	FPGA-to-HPS bridge GPV	0xFF600000	1 MB
EMAC0	EMAC0	0xFF700000	8 KB
EMAC1	EMAC1	0xFF702000	8 KB
SDMMC	SD/MMC	0xFF704000	4 KB
QSPIREGS	Quad SPI flash controller registers	0xFF705000	4 KB
FPGAMGRREGS	FPGA manager registers	0xFF706000	4 KB
ACPIDMAP	ACP ID mapper registers	0xFF707000	4 KB
GPIO0	GPIO0	0xFF708000	4 KB
GPIO1	GPIO1	0xFF709000	4 KB
GPIO2	GPIO2	0xFF70A000	4 KB
L3REGS	L3 interconnect GPV	0xFF800000	1 MB
NANDDATA	NAND controller data	0xFF900000	1 MB
QSPIDATA	Quad SPI flash data	0xFFA00000	1 MB
USB0	USB0 OTG controller registers	0xFFB00000	256 KB
USB1	USB1 OTG controller registers	0xFFB40000	256 KB
NANDREGS	NAND controller registers	0xFFB80000	64 KB
FPGAMGRDATA	FPGA manager configuration data	0xFFB90000	4 KB
CAN0	CAN0 controller registers	0xFFC00000	4 KB
CAN1	CAN1 controller registers	0xFFC01000	4 KB
UART0	UART0	0xFFC02000	4 KB
UART1	UART1	0xFFC03000	4 KB
I2C0	I2C0	0xFFC04000	4 KB
I2C1	I2C1	0xFFC05000	4 KB
I2C2	I2C2	0xFFC06000	4 KB
I2C3	I2C3	0xFFC07000	4 KB
SPTIMER0	SP Timer0	0xFFC08000	4 KB
SPTIMER1	SP Timer1	0xFFC09000	4 KB
SDRREGS	SDRAM controller subsystem registers	0xFFC20000	128 KB
OSC1TIMER0	OSC1 Timer0	0xFFD00000	4 KB
OSC1TIMER1	OSC1 Timer1	0xFFD01000	4 KB
L4WD0	Watchdog0	0xFFD02000	4 KB
L4WD1	Watchdog1	0xFFD03000	4 KB

Slave Identifier	Slave Title	Base Address	Size
CLKMGR	Clock manager	0xFFD04000	4 KB
RSTMGR	Reset manager	0xFFD05000	4 KB
SYSMGR	System manager	0xFFD08000	16 KB
DMANONSECURE	DMA nonsecure registers	0xFFE00000	4 KB
DMASECURE	DMA secure registers	0xFFE01000	4 KB
SPIS0	SPI slave0	0xFFE02000	4 KB
SPIS1	SPI slave1	0xFFE03000	4 KB
SPIM0	SPI master0	0xFFF00000	4 KB
SPIM1	SPI master1	0xFFF01000	4 KB
SCANMGR	Scan manager registers	0xFFF02000	4 KB
ROM	Boot ROM	0xFFFD0000	64 KB
MPUSCU	MPU SCU registers	0xFFFEC000	8 KB
MPUL2	MPU L2 cache controller registers	0xFFFEF000	4 KB
OCRAM	On-chip RAM	0xFFFF0000	64 KB

Every peripheral has a base address at which a certain number of registers can be found. You can then read and write to a certain set of these registers to modify the peripheral's behaviour. You do not need to hard-code any base address or peripheral register map in your programs, as Altera provides a header file for each one.

Three directories contain all HPS-related header files:

1. "<altera_install_directory>/<version>/embedded/ip/altera/hps/altera_hps/hwlib/include". Contains high-level header files that typically contain a few functions**,** which facilitate control over the HPS components. These functions are all part of Altera's HWLIB, which was created to make programming the HPS easier. This directory contains code that is common to the Cyclone V, Arria V, and Arria 10 devices.
2. **"<altera_install_directory>/<version>/embedded/ip/altera/hps/altera_hps/hwlib/include/soc_cv_av".** Same as above, but more specifically for the Cyclone V and Arria V FPGA families.
3. "<altera_install_directory>/<version>/embedded/ip/altera/hps/altera_hps/hwlib/include/soc_cv_av/socal". Contains low-level header files that provide a peripheral's bit-level register details. For example, any bits in a peripheral's register that correspond to undefined behavior will be specified in these header files.

To illustrate the differences between the high and low-level header files, we can compare the ones related to the FPGA manager peripheral:

1. "…/hwlib/include/soc_cv_av/alt_fpga_manager.h"

```
ALT_STATUS_CODE alt_fpga_reset_assert(void);
ALT_STATUS_CODE alt_fpga_configure(const void* cfg_buf, size_t cfg_buf_len);
```

2. "…/hwlib/include/soc_cv_av/socal/alt_fpgamgr.h"

```
/* The width in bits of the ALT_FPGAMGR_CTL_EN register field. */
#define ALT_FPGAMGR_CTL_EN_WIDTH      1
/* The mask used to set the ALT_FPGAMGR_CTL_EN register field value. */
#define ALT_FPGAMGR_CTL_EN_SET_MSK    0x00000001
/* The mask used to clear the ALT_FPGAMGR_CTL_EN register field value. */
#define ALT_FPGAMGR_CTL_EN_CLR_MSK    0xfffffffe
```

An important header file is **"…/hwlib/include/soc_cv_av/socal/hps.h"**. It contains the HPS component's full register map, as provided.

Note, however, that there exists no header file for the "*heavyweight*" HPS-to-FPGA bridge, as it is not located in the "HPS peripherals" region. Indeed, the "*heavyweight*" HPS-to-FPGA bridge is not considered an HPS peripheral, whereas the *"lightweight"* HPS-to-FPGA bridge is. Therefore, in order to use the *"heavyweight"* HPS-to-FPGA bridge, you will have to define a macro in your code, as follows:

```
#define ALT_HWFPGASLVS_OFST    0xc0000000
```

HWLIB can only be directly used in a bare-metal application**, as it directly references physical addresses. The library can unfortunately not be used directly in a Linux device driver because it uses standard header files that are not available in the kernel. Needless to say that a user-space Linux program cannot use the library directly either, as the Linux kernel would terminate a user process that tries to access any of these physical addresses directly. We will see later how to access a physical address from user space mapping the memory.**

2.7 HPS Booting and FPGA Configuration (text reproduced partially from [3])

Before being able to use the Cyclone V SoC, it is necessary to understand how the HPS boots and how the FPGA is configured.

2.7.1 HPS Boot and FPGA Configuration Ordering

The HPS boot starts when the processor is released from reset (for example, on power up) and executes code in the internal boot ROM at the reset exception address. The boot process ends when the code in the boot ROM jumps to the next stage of the boot software. This next stage of the boot software is referred to as the preloader**¡Error! No se encuentra el origen de la referencia.** Fig. 8 illustrates this *initial* incomplete HPS boot flow.

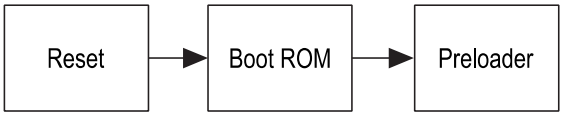


Fig. 8: Simplified HPS Boot Flow [1, pp. A-3]

The processor can boot from the following sources:

- NAND flash memory through the NAND flash controller
- SD/MMC flash memory through the SD/MMC flash controller
- SPI and QSPI flash memory through the QSPI flash controller using *Slave Select 0*
- FPGA fabric on-chip memory

The choice of the boot source is made by modifying the BOOTSEL and CLKSEL values before the device is powered up. Therefore, the Cyclone V device normally uses a physical DIP switch to configure the BOOTSEL and CLKSEL.

[DE1-SoC boot]: The DE1-SoC can ONLY BOOT from SD/MMC flash memory, as its BOOTSEL and CLKSEL values are hard-wired on the board. Although its HPS contains all necessary controllers, the board does not have a physical DIP switch to modify the BOOTSEL and CLKSEL values. The actual location of the DIP switch is present underneath the board, as can be seen in Fig. 5, but a switch is not soldered.

The configuration of the FPGA portion of the device starts when the FPGA portion is released from reset state (for example, on power up). The control block (CB) in the FPGA portion of the device is responsible for obtaining an FPGA configuration image and configuring the FPGA. The FPGA configuration ends when the configuration image has been fully loaded, and the FPGA enters user mode. The FPGA configuration image is provided by users and is typically stored in non-volatile flash-based memory. The FPGA CB can obtain a configuration image from the HPS through the FPGA manager, or from another external source, such as the *Quartus Prime Programmer*.

Fig. 9, Fig. 10 and Fig. 11 illustrate the possible HPS boot and FPGA configuration schemes. Note that Cyclone V devices can also be fully configured through a JTAG connection. Fig. 9 shows the scheme where the FPGA configuration and the HPS boot occur independently. The FPGA configuration obtains its image from a non-HPS source (*Quartus Prime Programmer*), while the HPS boot obtains its configuration image from a non-FPGA fabric source.

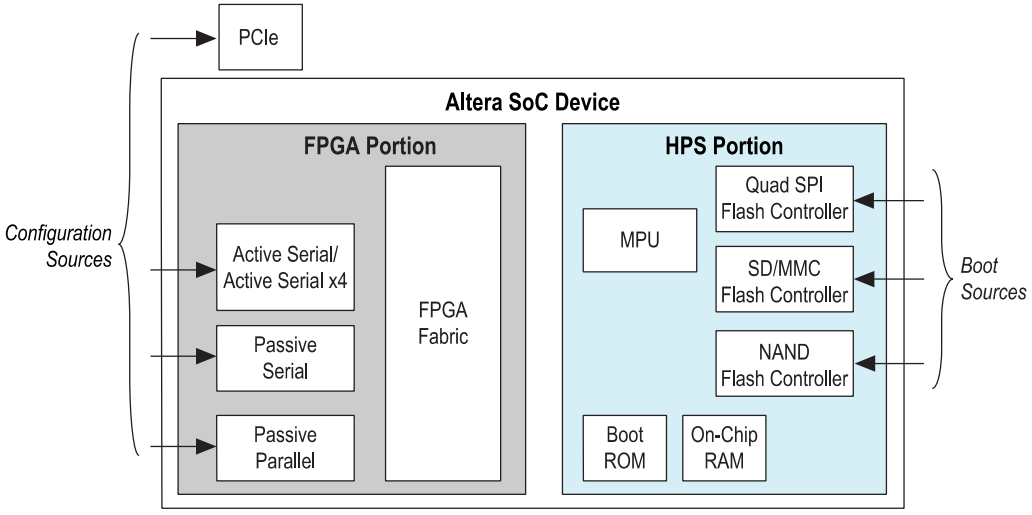


Fig. 9: Independent FPGA Configuration and HPS Booting [1, pp. A-2]

Fig. 10 shows the scheme where the FPGA is first configured through the *Quartus Prime Programmer*, then the HPS boots from the FPGA fabric. The HPS boot waits for the FPGA fabric to be powered on and in user mode before executing. The HPS boot ROM code executes the preloader from the FPGA fabric over the HPS-to-FPGA bridge. The preloader can be obtained from the FPGA on-chip memory, or by accessing an external interface (such as a larger external SDRAM).

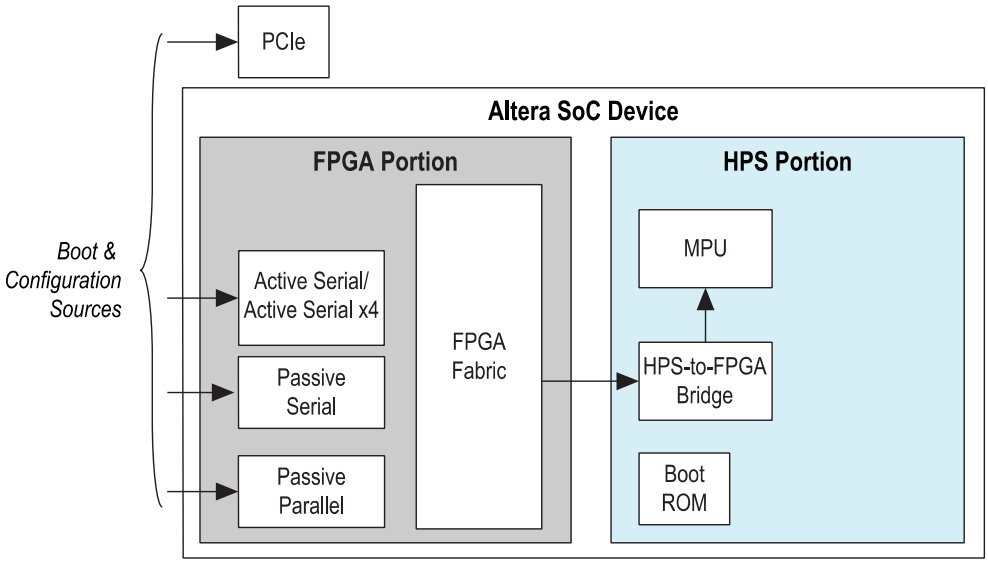


Fig. 10: FPGA Configuration before HPS Booting (HPS boots from FPGA) [1, pp. A-2]

¡Error! No se encuentra el origen de la referencia. shows the scheme under which the HPS first boots from one of its non-FPGA fabric boot sources, then software running on the HPS configures the FPGA fabric through the FPGA manager. The software on the HPS obtains the FPGA configuration image from any of its flash memory devices or communication interfaces, such as the SD/MMC memory, or the Ethernet port. The software is provided by users, and the boot ROM is not involved in configuring the FPGA fabric.

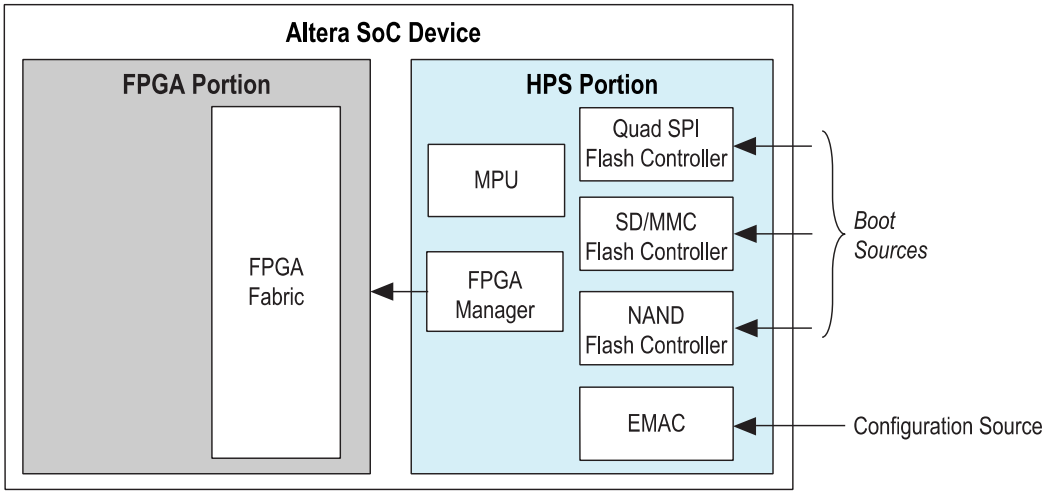


Fig. 11: HPS Boots and Performs FPGA Configuration [1, pp. A-3]

2.7.2 Zooming In On the HPS Boot Process

Booting software on the HPS is a multi-stage process. Each stage is responsible for loading the next one. The first software stage is the *boot ROM*. The boot ROM code locates and executes the second software stage, called the *preloader*. The preloader locates, and if present, executes the next software stage. The preloader and subsequent software stages are collectively referred to as user software.

The reset, boot ROM, and preloader stages are always present in the HPS boot flow. What comes after the preloader then depends on the type of application you want to run. The HPS can execute two types of applications:

- Bare-metal applications (no operating system)
- Applications on top of an operating system (Linux)

Fig. 12 shows the HPS' available boot flows. The Reset and Boot ROM stages are the only fixed parts of the boot process. Everything in the user software stages can be customized. Although the DE1-SoC has a **DUAL**-processor HPS (CPU0 and CPU1), the boot flow only executes on CPU0 and CPU1 is maintained under reset.

If you want to use both processors of the DE1-SoC, then User software executing on CPU0 is responsible for releasing CPU1 from reset.

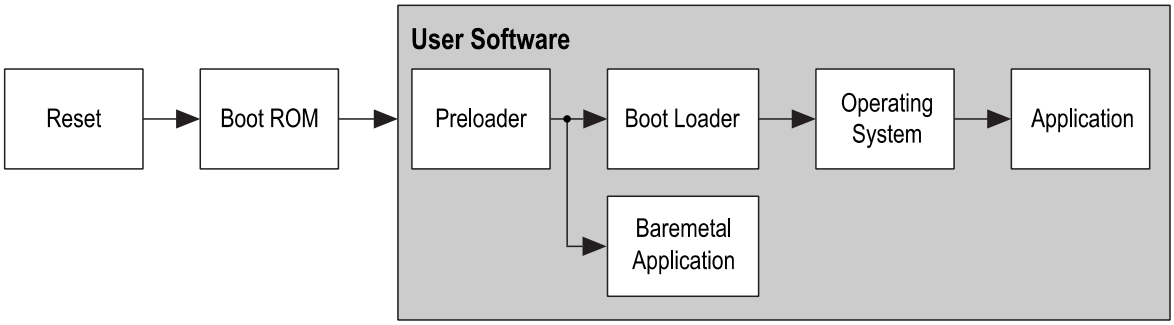


Fig. 12: HPS Boot Flows [1, pp. A-3]

2.7.2.1 Preloader

The preloader is one of the most important boot stages. It is actually, what one would call the boot "*source*", as all stages before it are unmodifiable. The preloader can be stored on the external flash-based memory, or in the FPGA fabric. The preloader typically performs the following actions:

- Initialize the SDRAM interface
- Configure the HPS I/O through the scan manager
- Configure pin multiplexing through the system manager
- Configure HPS clocks through the clock manager
- Initialize the flash controller (NAND, SD/MMC, QSPI) that contains the next stage boot software
- Load the next boot software into the SDRAM and pass control to it

The preloader does not release CPU1 from reset. The subsequent stages of the boot process are responsible for it if they want to use the extra processor.

2.8 Using FPGA-only

Exclusively using the FPGA part of the Cyclone V is easy, as the design process is identical to any other Altera FPGA. You can build a complete design in *Quartus Prime* & *Qsys*, simulate it in *ModelSim-Altera*, and then program the FPGA through the *Quartus Prime Programmer*. The DE1-SoC has a lot of pins, which makes it tedious to start an FPGA design. It is recommended to use the entity in for your top-level VHDL file, as it contains all the board's FPGA and HPS pins.

After having defined a top-level module, it is necessary to map your design's pins to the ones available on the DE1-SoC. The TCL script in can be executed in *Quartus Prime* to specify the board's device ID and all its pin assignments**.** In order to execute the TCL script, place it in your quartus working directory, and then run it through the "Tools > Tcl Scripts…" menu item in *Quartus Prime*.

2.9 Using HPS & FPGA

2.9.1 Bare-metal Application

On the one hand, bare-metal software enjoys the advantage of having no OS overhead. This has many consequences, the most visible of which are that code executes at native speed as no context switching is ever performed, and additionally, that code can directly address the HPS peripherals using their physical memory-mapped addresses, as no virtual memory system is being used. This is very useful when trying to use the HPS as a high-speed microcontroller. Such a programming environment is very similar to the one used by other microcontrollers.

On the other hand, the bare-metal code has one great disadvantage, as the programmer must continue to configure the Cyclone V to use all its resources. For example, the preloader does not release CPU1 from reset, and that it is up to the user software to perform this, which is the bare-metal application itself in this case. Furthermore, supposing CPU1 is available for use, it is still difficult to run multi-threaded code, as an OS generally handles program scheduling and CPU affinity for the programmer. The programmer must now manually assign code fragments to each CPU.

2.9.2 Application over an Operating System (Linux)

Running code on a Linux operating system has several advantages. First of all, the kernel releases CPU1 from reset upon boot, so all processors are available. Furthermore, the kernel initializes and makes most, if not all HPS peripherals available for use by the programmer. This is possible since the Linux kernel has access to a huge amount of device drivers. Multi-threaded code is also much easier to write, as the programmer has access to the familiar Pthreads system calls. Finally, the Linux kernel is not restricted to running compiled C programs. Indeed, you can always run code written in another programming language providing you first install the runtime environment required (that must be available for ARM processors).

However, running an embedded application on top of an operating system also has disadvantages. Due to the virtual memory system put in place by the OS, a program cannot directly access the HPS peripherals through their physical memory-mapped addresses. Instead, one first needs to map the physical addresses of interest into the running program's virtual address space. Only then will it be possible to access a peripheral's registers. Ideally, the programmer should write a device driver for each specific component that is designed to have a clean interface between user code, and device accesses.

At the end of the day, bare-metal applications and applications running code on top of Linux can do the same things. Generally speaking, programming on top of Linux is superior and much easier compared to bare-metal code, as its advantages greatly outweigh its drawbacks.

3 USING THE CYCLONE V – CREATING AND TESTING A BASIC BSP (BOARD SUPPORT PACKAGE)

The details below give step-by-step instructions to create a full system from scratch.

3.1 Goals

Let's start by defining what we want to achieve in this tutorial. We want to create a system in which both the HPS and FPGA can do some computation simultaneously. More specifically, we want the following capabilities:

1. The ***HPS*** must be able to use the LED and button that are directly connected to the ***HPS PORTION*** of the device. Pressing the button should toggle the LED.
2. The ***HPS*** must be able to use two buttons and the six 7-segment displays connected to the ***FPGA PORTION*** of the device. The HPS will increment and decrement a counter that will be shown on the 7-segment displays. Pressing the first button should invert the counting direction, and pushing the second button should reset the counter to 0.
3. The ***HPS*** must be able to use the ethernet port on the board.
4. The ***HPS*** must be able to use the microSD card port on the board to which we will write anything we want.

3.2 Project Structure

The complete development process creates several files and folders. We will use the folder structure shown in Fig. 13 to organize our project. In this demo, we will use "`soc_system`" as the project name.

- The "`cyclonevbsp`" is the root folder with the quartus project inside.
- The "`soc_system`", "`hps_isw_handoff`", and "`output_files`" `subfolders are generated by Quartus tools as the tutorial progresses.`
- The "`software`" directory contains the basic software for the initialization of the SoC system.

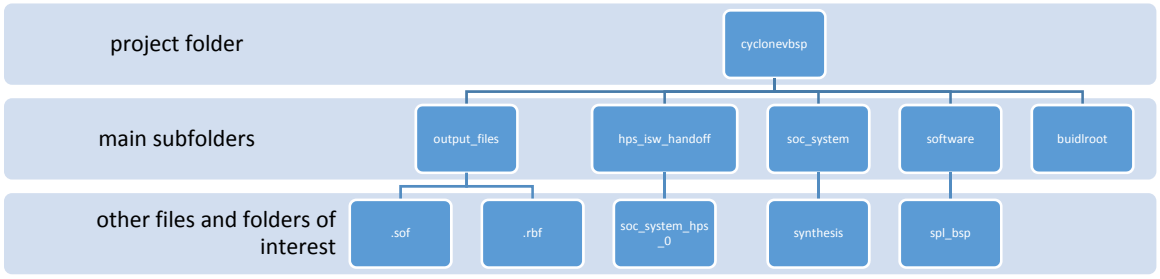


Fig. 13: Project folder organization

[Project organization]: Maintain the above project organization and the names suggested for the different elements. This helps you and the instructor to find errors.

3.3 Quartus Prime Lite 16.0 Setup

1. Create a new *Quartus Prime* project using "File->New Project Wizard. You will see a window similar to Fig. 14. Pressing Next button you will be prompted to select the project name and destination, as all other settings (Fig. 15). For this tutorial, we will call our project "soc_sytem", the top level file is the "soc_system_top" and will store it in "`cycloneVbsp`" `folder`. In the "Project Type window" select an "Empty Project option" as shown in Fig. 15. Do not "Add Files" in this step and select a Cyclone V SoC 5CSEMA5F31C6 device (Fig. 17). Finally in the "EDA Tools Settings" define the format file for ModelSim-Altera as VHDL (Fig. 18). Press Finish to create the project.

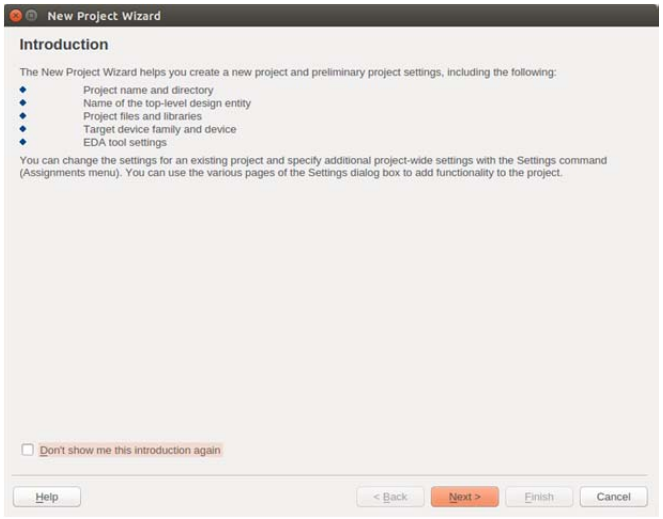


Fig. 14: Creating a New Project using the wizard.

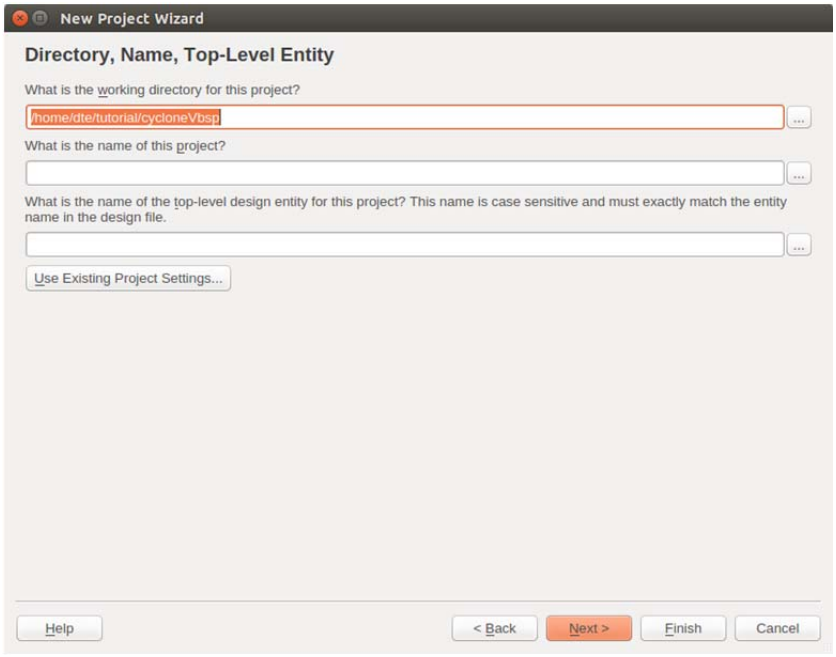


Fig. 15: Selecting the project location, name and the top level entity.

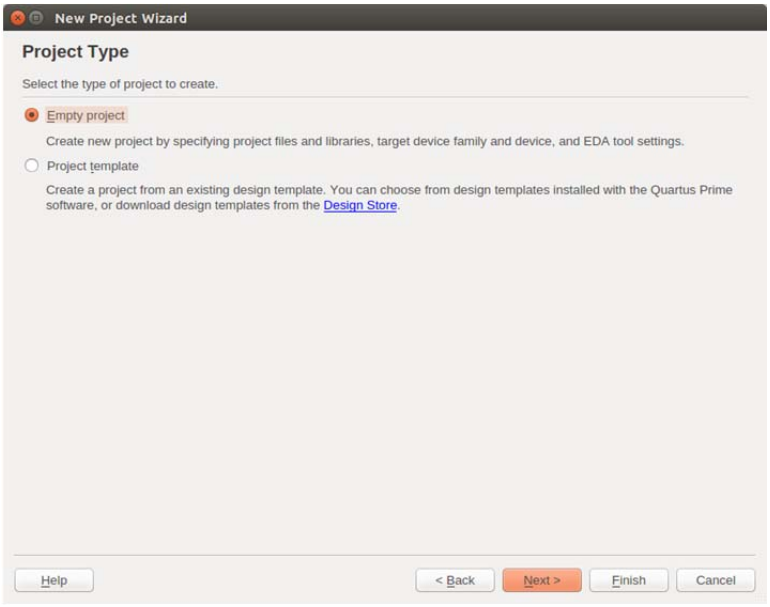


Fig. 16: Project type selection.

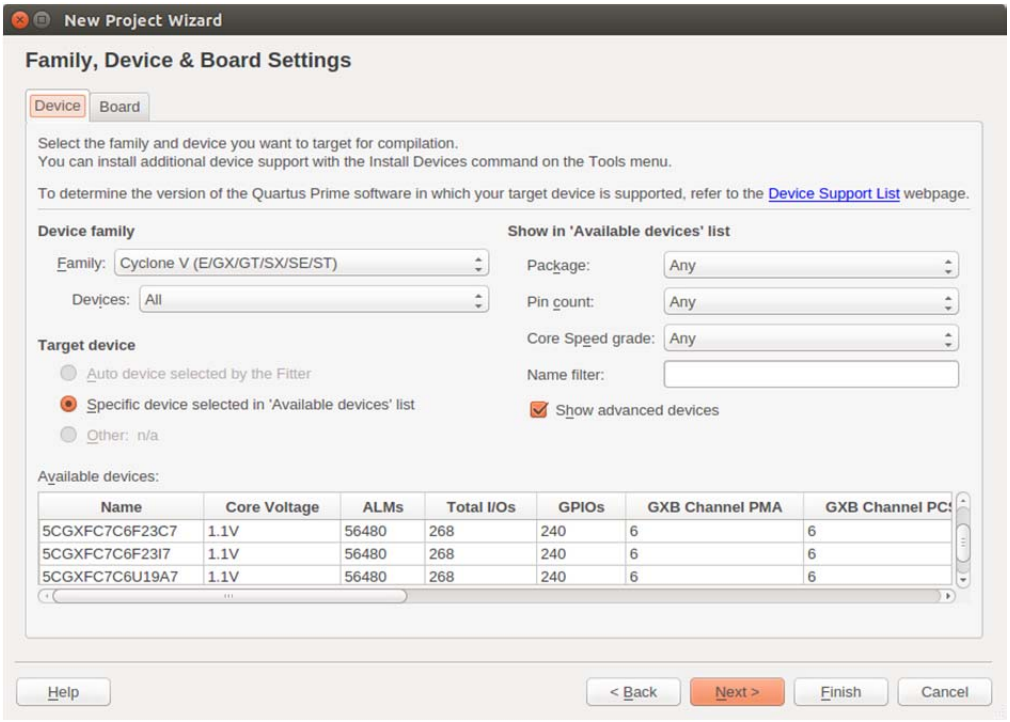


Fig. 17: Selecting the Device.

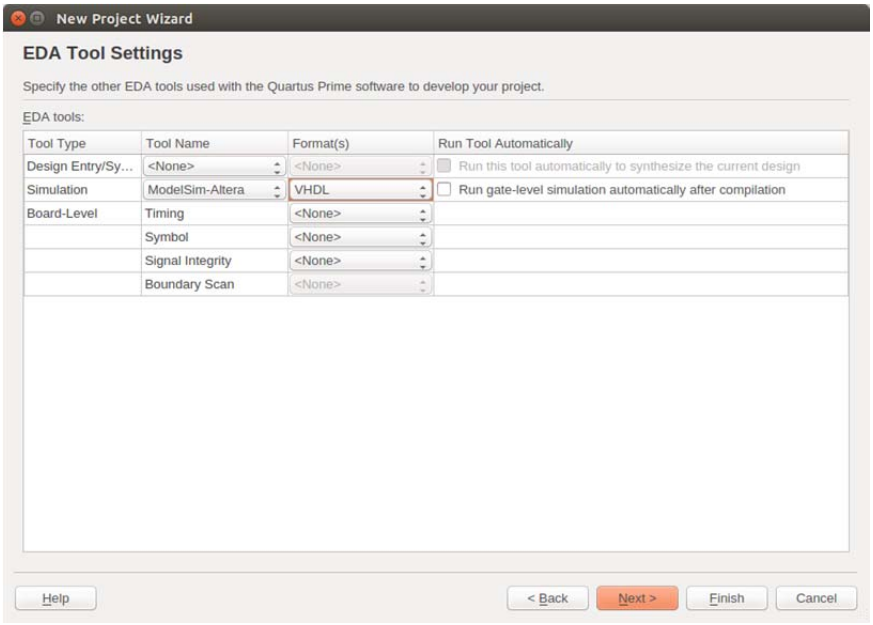


Fig. 18: Selecting VHDL for ModelSim Tool.

2. Download `soc_system_top.vhd` [4] and save it in "`cycloneVbsp`". We use this file as the project's top-level VHDL file, as it contains a complete list of pin names available on the DE1-SoC for use in your designs.
3. Download `pin_assignment_DE1_SoC.tcl` and save it in "`cycloneVbsp`". This script assigns pin locations and I/O standards to all pins names in "`soc_system_top.vhd`". Execute the TCL script by using "`Tools > Tcl Scripts…`" in *Quartus Prime*. A new window will be prompted. Select the file and press Run. After some tenths of seconds, a message is displayed in a dialog box indicating that the file has been processed. Close the window.

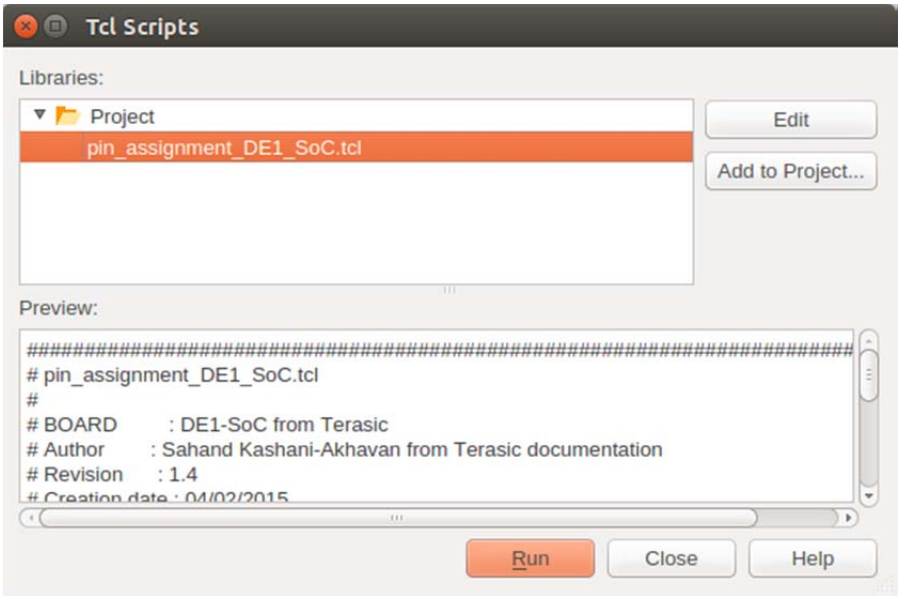


Fig. 19: Pin assignment using TCL script.

At this stage, all general *Quartus Prime* settings have been performed, and we can start creating our design. We want to use the HPS, therefore we will use the *Qsys* tool to create the system.

4. Launch the *Qsys* tool (Tools->Qsys) and create a new system. Save it under the name "soc_system.qsys".

3.4 System Design with Qsys – HPS

In this section, we are going to assemble all system components needed to allow the HPS to boot the Linux embedded system.

[Note]: When using Qsys to manipulate any signal or menu item related to the HPS, the GUI will seem as though it is not responding, but this is not the case. The GUI is just checking all parameters in the background, which makes the interface hang momentarily. It is working correctly behind the scenes.

3.4.1 Instantiating the HPS Component

1. To use the HPS, add an "Arria V/Cyclone V Hard Processor System" to the system. In the IP catalog select it in the "Processor and Peripherals->Hard Processor Systems". This opens a window allowing to select the HPS' parameters. There are four tabs that control various aspects of the HPS' behaviour, as shown in Fig. 20.

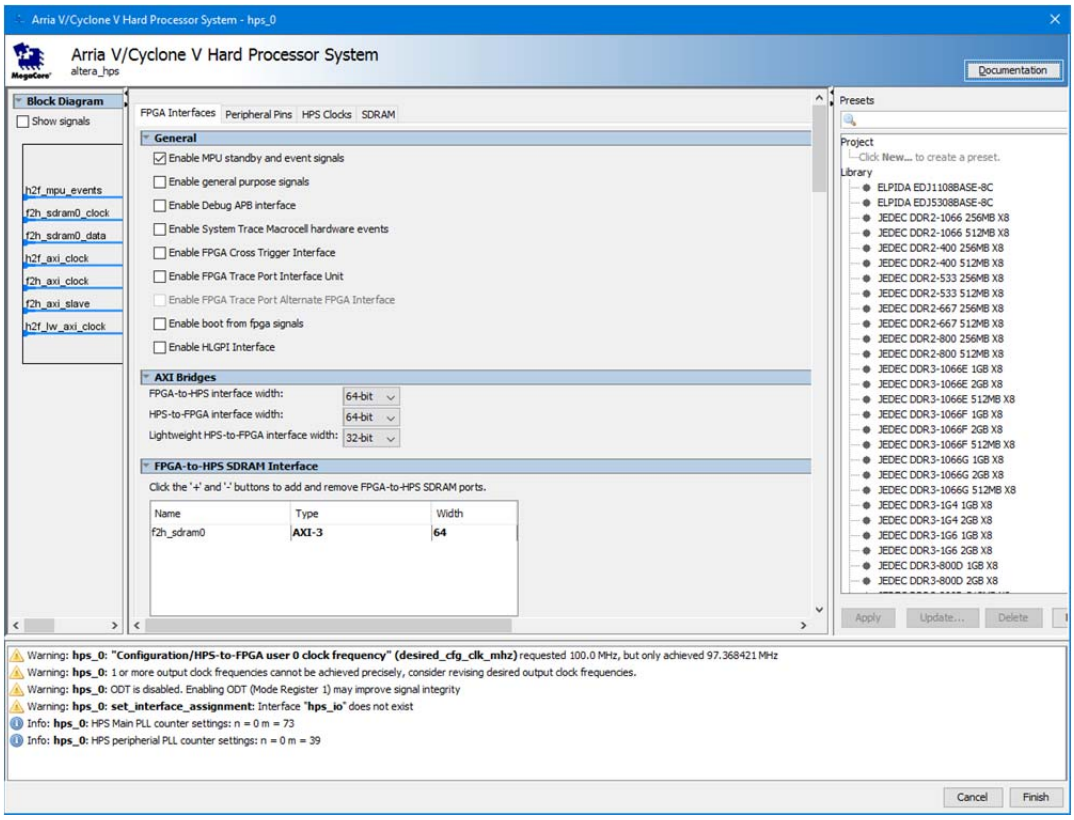


Fig. 20: HPS Component Parameters

3.4.1.1 FPGA Interfaces Tab

This tab configures everything related to the interfaces between the HPS and the FPGA.

2. We want to use the HPS to access FPGA peripherals, so we need to enable one of the following buses:
    a. HPS-to-FPGA AXI bridge
    b. Lightweight HPS-to-FPGA AXI bridge

    Since we are not going to be using any high-performance FPGA peripherals in this tutorial, we choose to enable the Lightweight HPS-to-FPGA AXI bridge.
    - Set the FPGA-to-HPS interface width to "Unused".
    - Set the HPS-to-FPGA interface width to "Unused".

- Set the Lightweight HPS-to-FPGA interface width to "32 bits"

By default, *Qsys* checks "Enable MPU standby and event signals", but we are not going to use this feature, so

- Uncheck "Enable MPU standby and event signals".

Qsys also adds an FPGA-to-HPS SDRAM port by default, which we are not going to use either, so

- Remove the port listed under "FPGA-to-HPS SDRAM Interface". (OpenCL based designs uses this port, in such case the port shouldn't be removed)

Later in our specific peripheral design, we are using interrupts as a way to control its behaviour, so FPGA-to-HPS Interrupts should be enabled. Therefore,

- Check "Enable FPGA-to-HPS Interrupts" option, just at the beginning of the Interrupts section

3.4.1.2 Peripheral Pins Tab

This tab configures the HPS' peripheral pins that are available on the SoC device. Most device pins have various sources and are *multiplexed*. The pins can be configured to be sourced by the FPGA part, or by the different HPS peripherals. We want to use the HPS to access the button and LED available in the DE1-SoC. Reviewing the DE1-SOC schematics in DE1-SoC.pdf [5] we can identify these pins as "HPS_KEY" and "HPS_LED". They are connected to pins G21 and A24. Additionally, these pins are identified on the device's top-level entity file as HPS_LED and HPS_KEY_N

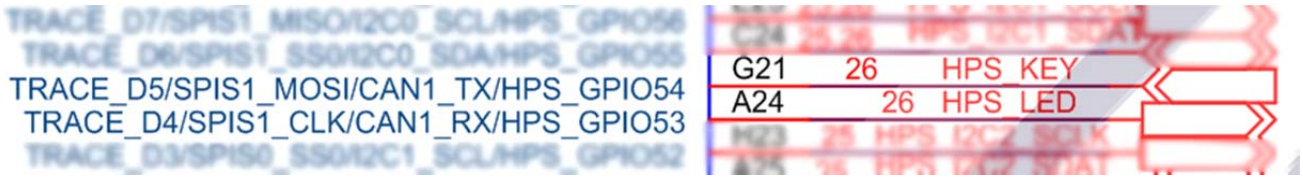


Fig. 21: Detail of connection of HPS_KEY & HPS_LED on DE1-SoC Schematics to pins G21 and A24

The *Qsys* GUI doesn't make any reference to pins G21 and A24, as they depend on the device being used, and cannot be generalized to other Cyclone V devices. However, the GUI does have references to what is displayed on the left side of Fig. 21. We will examine the details of pin G21, to which "HPS_KEY_N" is connected. The schematic shows that pin G21 is connected to 4 possible sources:

a. TRACE_D5
b. SPIS1_MOSI
c. CAN1_TX
d. HPS_GPIO54

This can be seen in *Qsys*, as shown in Fig. 22.

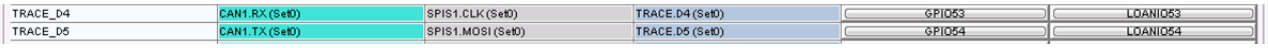


Fig. 22: HPS_KEY & HPS_LED on Qsys Peripheral Pins Tab

Depending on how you configure the Peripheral Pins tab, you can configure pin G21 to use any of the sources above. For example, if you want to use this pin as the SPI Master Output Slave Input control signal, you would use the configuration shown in Fig. 23.

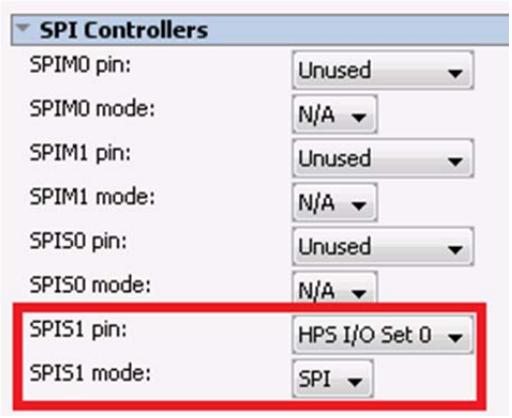


Fig. 23 Using Pin G21 for SPI MOSI

However, if you don't want to use any of the peripherals available at the top of the *Peripheral Pins* tab, then you can always use one of the two buttons on the right side of Fig. 22:

- ***GPIO****XY*: Configures the pin to be connected to the ***HPS' GPIO*** peripheral.
- ***LOANIO****XY*: Configures the pin to be connected to the ***FPGA*** fabric (part). This pin can be exported from *Qsys* to be used by the FPGA.

3.4.1.2.1 Configuration

3. We want the HPS to directly control the "HPS_KEY_N" and "HPS_LED" pins. To do this, we connect pins G21 and A24 to the HPS' GPIO peripheral.
    e. Click on the **"GPIO53"** button. This corresponds to pin A24, which is connected to **"HPS_LED"**.
    f. Click on the **"GPIO54"** button. This corresponds to pin G21, which is connected to **"HPS_KEY_N"**.

4. We want to connect to DE1-SoC using the Ethernet interface using TCP/IP with an SSH protocol connection later in the tutorial, so we need to enable the Ethernet MAC interface.
    c. Configure **"EMAC1 pin"** to **"HPS I/O Set 0"** and the **"EMAC 1 mode"** to **"RGMII"**, as shown in Fig. 24
    d. Click on the **"GPIO35"** button. This corresponds to pin C19, which is connected to **"HPS_ENET_INT_N"** (see schematics Bank 7B, pin C19).

Fig. 24: Ethernet MAC configuration

5. Our system will boot from the microSD card slot, so we need to enable the SD/MMC controller.
    e. Configure **"SDIO pin"** to **"HPS I/O Set 0"** and **"SDIO mode"** to **"4-bit Data"**, as shown in Fig. 25.

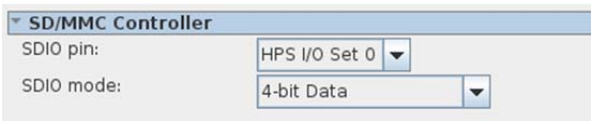


Fig. 25: SD/MMC configuration

6. In the early stages of the development of a SoC-based system, we need to use a serial line (RS232). We will do this through a serial UART connection, so we need to enable the UART controller.
    f. Configure "UART0 pin" to "HPS I/O Set 0" and "UART0 mode" to "No Flow Control", as shown in Fig. 26.

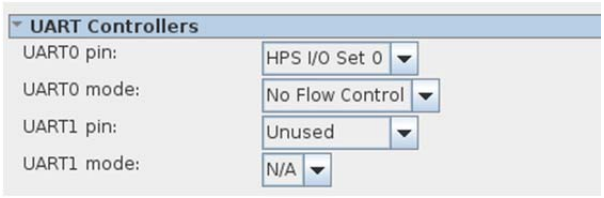


Fig. 26: UART configuration

7. Although not needed to satisfy the design goals that has been defined, we enable all the remaining HPS peripherals so future designs can use any of them if needed. Adding these peripherals does not increase FPGA resource usage as they are all hard peripherals connected directly to the HPS.
    a. Configure the Quad SPI Flash controller (Fig. 27), USB controllers (Fig. 28), SPI controllers (Fig. 29), and the I^2C controllers (Fig. 30).
    b. Click on the "GPIO09" button. This corresponds to pin B15, which is connected to "HPS_CONV_USB_N".
    c. Click on the "GPIO35" button. This corresponds to pin C19, which is connected to "HPS_ENET_INT_N".
    d. Click on the "GPIO40" button. This corresponds to pin H17, which is connected to "HPS_LTC_GPIO".
    e. Click on the "GPIO48" button. This corresponds to pin B26, which is connected to "HPS_I2C_CONTROL".
    f. Click on the "GPIO61" button. This corresponds to pin B22, which is connected to "HPS_GSENSOR_INT".

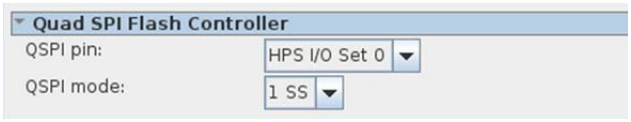


Fig. 27: Quad SPI Controller connection in HPS

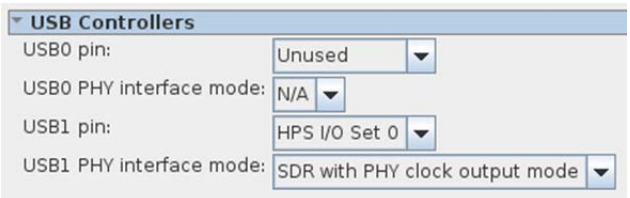


Fig. 28: USB controller pin setup

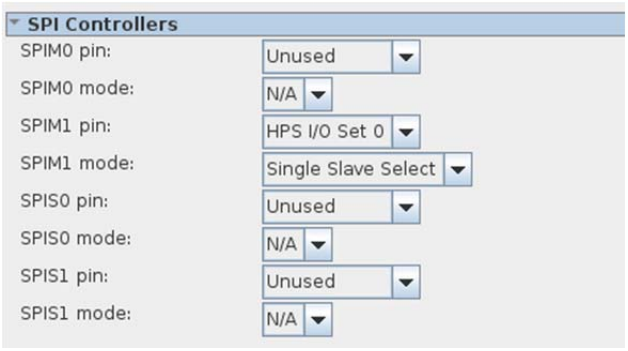


Fig. 29: SPI master controller configuration

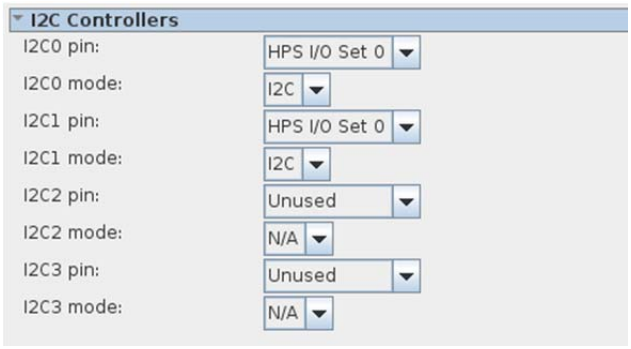
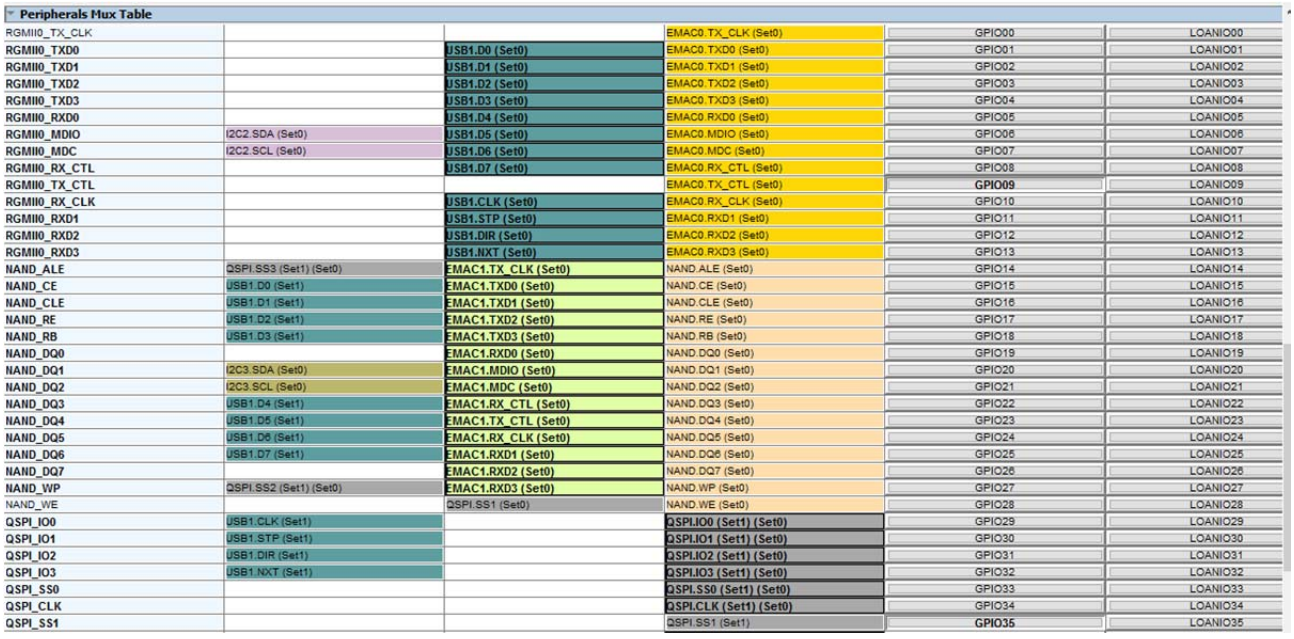


Fig. 30: I2C Master configuration

At this stage, you should have the same configuration shown in Fig. 31. Close the properties windows of HPS.



Peripherals Mux Table

RGMII0_TX_CLK			EMAC0.TX_CLK (Set0)	GPIO00	LOANIO00
RGMII0_TXD0		USB1.D0 (Set0)	EMAC0.TXD0 (Set0)	GPIO01	LOANIO01
RGMII0_TXD1		USB1.D1 (Set0)	EMAC0.TXD1 (Set0)	GPIO02	LOANIO02
RGMII0_TXD2		USB1.D2 (Set0)	EMAC0.TXD2 (Set0)	GPIO03	LOANIO03
RGMII0_TXD3		USB1.D3 (Set0)	EMAC0.TXD3 (Set0)	GPIO04	LOANIO04
RGMII0_RXD0		USB1.D4 (Set0)	EMAC0.RXD0 (Set0)	GPIO05	LOANIO05
RGMII0_MDIO	I2C2.SDA (Set0)	USB1.D5 (Set0)	EMAC0.MDIO (Set0)	GPIO06	LOANIO06
RGMII0_MDC	I2C2.SCL (Set0)	USB1.D6 (Set0)	EMAC0.MDC (Set0)	GPIO07	LOANIO07
RGMII0_RX_CTL		USB1.D7 (Set0)	EMAC0.RX_CTL (Set0)	GPIO08	LOANIO08
RGMII0_TX_CTL			EMAC0.TX_CTL (Set0)	GPIO09	LOANIO09
RGMII0_RX_CLK		USB1.CLK (Set0)	EMAC0.RX_CLK (Set0)	GPIO10	LOANIO10
RGMII0_RXD1		USB1.STP (Set0)	EMAC0.RXD1 (Set0)	GPIO11	LOANIO11
RGMII0_RXD2		USB1.DIR (Set0)	EMAC0.RXD2 (Set0)	GPIO12	LOANIO12
RGMII0_RXD3		USB1.NXT (Set0)	EMAC0.RXD3 (Set0)	GPIO13	LOANIO13
NAND_ALE	QSPI.SS3 (Set1) (Set0)	EMAC1.TX_CLK (Set0)	NAND.ALE (Set0)	GPIO14	LOANIO14
NAND_CE	USB1.D0 (Set1)	EMAC1.TXD0 (Set0)	NAND.CE (Set0)	GPIO15	LOANIO15
NAND_CLE	USB1.D1 (Set1)	EMAC1.TXD1 (Set0)	NAND.CLE (Set0)	GPIO16	LOANIO16
NAND_RE	USB1.D2 (Set1)	EMAC1.TXD2 (Set0)	NAND.RE (Set0)	GPIO17	LOANIO17
NAND_RB	USB1.D3 (Set1)	EMAC1.TXD3 (Set0)	NAND.RB (Set0)	GPIO18	LOANIO18
NAND_DQ0		EMAC1.RXD0 (Set0)	NAND.DQ0 (Set0)	GPIO19	LOANIO19
NAND_DQ1	I2C3.SDA (Set0)	EMAC1.MDIO (Set0)	NAND.DQ1 (Set0)	GPIO20	LOANIO20
NAND_DQ2	I2C3.SCL (Set0)	EMAC1.MDC (Set0)	NAND.DQ2 (Set0)	GPIO21	LOANIO21
NAND_DQ3	USB1.D4 (Set1)	EMAC1.RX_CTL (Set0)	NAND.DQ3 (Set0)	GPIO22	LOANIO22
NAND_DQ4	USB1.D5 (Set1)	EMAC1.TX_CTL (Set0)	NAND.DQ4 (Set0)	GPIO23	LOANIO23
NAND_DQ5	USB1.D6 (Set1)	EMAC1.RX_CLK (Set0)	NAND.DQ5 (Set0)	GPIO24	LOANIO24
NAND_DQ6	USB1.D7 (Set1)	EMAC1.RXD1 (Set0)	NAND.DQ6 (Set0)	GPIO25	LOANIO25
NAND_DQ7		EMAC1.RXD2 (Set0)	NAND.DQ7 (Set0)	GPIO26	LOANIO26
NAND_WP	QSPI.SS2 (Set1) (Set0)	EMAC1.RXD3 (Set0)	NAND.WP (Set0)	GPIO27	LOANIO27
NAND_WE		QSPI.SS1 (Set0)	NAND.WE (Set0)	GPIO28	LOANIO28
QSPI_IO0	USB1.CLK (Set1)		QSPI.IO0 (Set1) (Set0)	GPIO29	LOANIO29
QSPI_IO1	USB1.STP (Set1)		QSPI.IO1 (Set1) (Set0)	GPIO30	LOANIO30
QSPI_IO2	USB1.DIR (Set1)		QSPI.IO2 (Set1) (Set0)	GPIO31	LOANIO31
QSPI_IO3	USB1.NXT (Set1)		QSPI.IO3 (Set1) (Set0)	GPIO32	LOANIO32
QSPI_SS0			QSPI.SS0 (Set1) (Set0)	GPIO33	LOANIO33
QSPI_CLK			QSPI.CLK (Set1) (Set0)	GPIO34	LOANIO34
QSPI_SS1			QSPI.SS1 (Set1)	GPIO35	LOANIO35

SDMMC_CMD		USB0.D0 (Set0)	SDIO.CMD (Set0)	GPIO36	LOANIO36
SDMMC_PWREN		USB0.D1 (Set0)	SDIO.PWREN (Set0)	GPIO37	LOANIO37
SDMMC_D0		USB0.D2 (Set0)	SDIO.D0 (Set0)	GPIO38	LOANIO38
SDMMC_D1		USB0.D3 (Set0)	SDIO.D1 (Set0)	GPIO39	LOANIO39
SDMMC_D4		USB0.D4 (Set0)	SDIO.D4 (Set0)	GPIO40	LOANIO40
SDMMC_D5		USB0.D5 (Set0)	SDIO.D5 (Set0)	GPIO41	LOANIO41
SDMMC_D6		USB0.D6 (Set0)	SDIO.D6 (Set0)	GPIO42	LOANIO42
SDMMC_D7		USB0.D7 (Set0)	SDIO.D7 (Set0)	GPIO43	LOANIO43
SDMMC_FB_CLK_IN		USB0.CLK (Set0)		GPIO44	LOANIO44
SDMMC_CCLK_OUT		USB0.STP (Set0)	SDIO.CLK (Set0)	GPIO45	LOANIO45
SDMMC_D2		USB0.DIR (Set0)	SDIO.D2 (Set0)	GPIO46	LOANIO46
SDMMC_D3		USB0.NXT (Set0)	SDIO.D3 (Set0)	GPIO47	LOANIO47
TRACE_CLK			TRACE.CLK (Set0)	GPIO48	LOANIO48
TRACE_D0	UART0.RX (Set0)	SPIS0.CLK (Set0)	TRACE.D0 (Set0)	GPIO49	LOANIO49
TRACE_D1	UART0.TX (Set0)	SPIS0.MOSI (Set0)	TRACE.D1 (Set0)	GPIO50	LOANIO50
TRACE_D2	I2C1.SDA (Set0)	SPIS0.MISO (Set0)	TRACE.D2 (Set0)	GPIO51	LOANIO51
TRACE_D3	I2C1.SCL (Set0)	SPIS0.SS0 (Set0)	TRACE.D3 (Set0)	GPIO52	LOANIO52
TRACE_D4	CAN1.RX (Set0)	SPIS1.CLK (Set0)	TRACE.D4 (Set0)	GPIO53	LOANIO53
TRACE_D5	CAN1.TX (Set0)	SPIS1.MOSI (Set0)	TRACE.D5 (Set0)	GPIO54	LOANIO54
TRACE_D6	I2C0.SDA (Set0)	SPIS1.SS0 (Set0)	TRACE.D6 (Set0)	GPIO55	LOANIO55
TRACE_D7	I2C0.SCL (Set0)	SPIS1.MISO (Set0)	TRACE.D7 (Set0)	GPIO56	LOANIO56
SPIM0_CLK	UART0.CTS (Set2) (Set1) (Set0)	I2C1.SDA (Set1)	SPIM0.CLK (Set0)	GPIO57	LOANIO57
SPIM0_MOSI	UART0.RTS (Set2) (Set1) (Set0)	I2C1.SCL (Set1)	SPIM0.MOSI (Set0)	GPIO58	LOANIO58
SPIM0_MISO	UART1.CTS (Set0)	CAN1.RX (Set1)	SPIM0.MISO (Set0)	GPIO59	LOANIO59
SPIM0_SS0	UART1.RTS (Set0)	CAN1.TX (Set1)	SPIM0.SS0 (Set0)	GPIO60	LOANIO60
UART0_RX	SPIM0.SS1 (Set0)	CAN0.RX (Set0)	UART0.RX (Set1)	GPIO61	LOANIO61
UART0_TX	SPIM1.SS1 (Set0)	CAN0.TX (Set0)	UART0.TX (Set1)	GPIO62	LOANIO62
I2C0_SDA	SPIM1.CLK (Set0)	UART1.RX (Set0)	I2C0.SDA (Set1)	GPIO63	LOANIO63
I2C0_SCL	SPIM1.MOSI (Set0)	UART1.TX (Set0)	I2C0.SCL (Set1)	GPIO64	LOANIO64
CAN0_RX	SPIM1.MISO (Set0)	UART0.RX (Set2)	CAN0.RX (Set1)	GPIO65	LOANIO65
CAN0_TX	SPIM1.SS0 (Set0)	UART0.TX (Set2)	CAN0.TX (Set1)	GPIO66	LOANIO66

To enable a HPS pin to work as a Loan IO or as a GPIO pin, Click on the GPIO or Loan IO button on the Peripherals Mux table. The Specific peripherals are enabled in the Drop boxes above the Peripherals Mux table.

Fig. 31: Exported peripheral pins. Verify that your design export exactly the same pins.

8. In *Qsys*' displays the "`System Contents`" tab:
    - Export "`hps_0.hps_io`" under the name "`hps_0_io`". This is a conduit that contains all the pins configured in the *Peripheral Pins* tab. We will connect these to our top-level entity later.

3.4.1.3 HPS Clocks Tab

Open HPS again and select HPS Clocks tab. This tab configures the HPS clocking system. We will use the default settings here, so we only need to change one thing to get the maximum precision for the output clocks.

9. In "Output Clocks" tab:
    - Unselect the "`Use default MPU clock frequency`" mark and set MPU clock frequency to 800MHz and.

3.4.1.4 SDRAM Tab

This tab configures the memory subsystem of the HPS.

10. We need to configure all clocks and timings related to the memory used in our system. The DE1-SoC uses DDR3 memory, so we need to consult its datasheet to find all the settings. The datasheet is available at [6]. Based on the memory's datasheet, we can fill in the following memory settings (pay attention to the values):
    - SDRAM Protocol: DDR3
    - PHY Settings:
        - Clock
            - Memory clock frequency: 400.0 MHz
            - PLL reference clock frequency: 25.0 MHz
        - Advanced PHY Settings:
            - Supply Voltage: 1.5V DDR3
    - Memory Parameters:
        - Memory vendor: Other
        - Memory device speed grade: 800.0 MHz
        - Total interface width: 32
        - Number of chip select/depth expansion: 1
        - Number of clocks: 1
        - Row address width: 15

        - Column address width: 10
        - Bank-address width: 3
        - Enable DM pins
        - DQS# Enable
        - Memory Initialization Options:
            - Mirror Addressing: 1 per chip select: 0
            - Mode Register 0:
                - Burst Length: Burst chop 4 or 8 (on the fly)
                - Read Burst Type: Sequential
                - DLL precharge power down: DLL off
                - Memory CAS latency setting: 7
            - Mode Register 1:
                - Output drive strength setting: RZQ/6
                - ODT Rtt nominal value: RZQ/6
            - Mode Register 2:
                - Auto selfrefresh method: Manual
                - Selfrefresh temperature: Normal
                - Memory write CAS latency setting: 7
                - Dynamic ODT (Rtt_WR) value: Dynamic ODT off
- Memory Timing:
    - tIS (base): 190 ps
    - tIH (base): 140 ps
    - tDS (base): 30 ps
    - tDH (base): 65 ps
    - tDQSQ: 125 ps
    - tQH: 0.38 cycles
    - tDQSCK: 255 ps
    - tDQSS: 0.25 cycles
    - tQSH: 0.4 cycles
    - tDSH: 0.2 cycles
    - tDSS: 0.2 cycles
    - tINIT: 500 us
    - tMRD: 4 cycles
    - tRAS: 36.0 ns
    - tRCD: 13.125 ns
    - tRP: 13.125 ns
    - tREFI: 7.8 us
    - tRFC: 300.0 ns
    - tWR: 15.0 ns
    - tWTR: 4 cycles
    - tFAW: 45.0 ns
    - tRRD: 7.5 ns
    - tRTP: 7.5 ns
- Board Settings:
    - Setup and Hold Derating:
        - Use Altera's default settings
    - Channel Signal Integrity:
        - Use Altera's default settings
    - Board Skews:

- Maximum CK delay to DIMM/device: 0.03 ns
- Maximum DQS delay to DIMM/device: 0.02 ns
- Minimum delay difference between CK and DQS: 0.06 ns
- Maximum delay difference between CK and DQS: 0.12 ns
- Maximum skew within DQS group: 0.01 ns
- Maximum skew between DQS groups: 0.06 ns
- Average delay difference between DQ and DQS: 0.05 ns
- Maximum skew within address and command bus: 0.02 ns
- Average delay difference between address and command and CK: 0.01 ns

11. In *Qsys*' "System Contents" tab:
    - Export "`hps_0.memory`" under the name "`hps_0_ddr`".
    - Export "`hps_0.h2f_reset`" under the name "`hps_0_h2f_reset`".

Now that you've placed all of the components, you must connect all the interfaces together. All of the possible connections are indicated by light grey lines. To make an actual connection, simply click on the empty bubbles at the intersections of lines. A connection which is actually made will turn black, and the bubble will be filled in.

12. Connect the system as shown in Fig. 32 below:

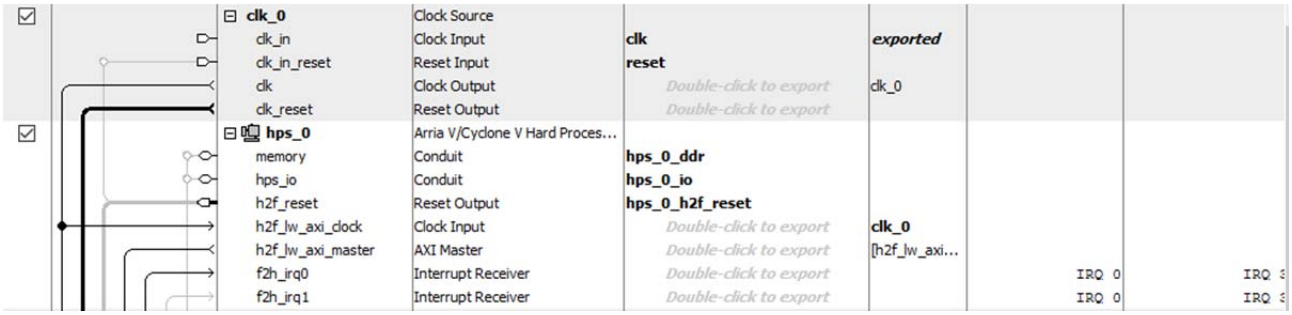


Fig. 32: Adding the "Standalone" HPS to the System

3.4.2 Adding SYSID peripheral.

13. Add a SYSID peripheral selecting it in `Basic Functions->Simulation Debug and Verification->Debug Performance->System ID`. Connect SYSID clock and Avalon Bus as depicted in Fig. 33.

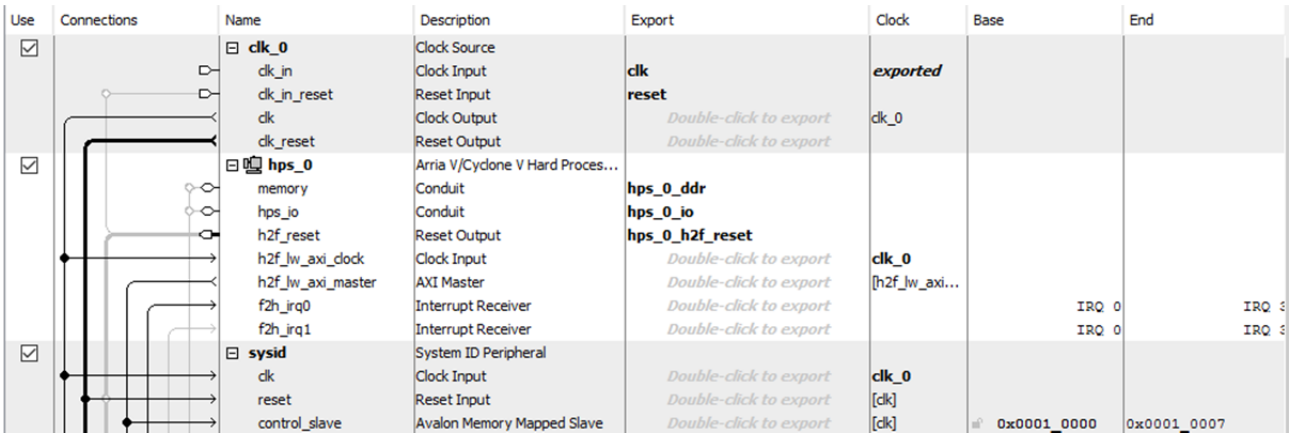


Fig. 33: Connection of the peripheral to the HPS

3.4.3 Adding the JTAG UART interface.

14. Add a JTAG UART peripheral selecting it in Interface Protocols->Serial. Leave the default parameters. This peripheral generates interrupts. Edit the HPS system and check the FPGA to HPS interrupts. Two new terminals will be added to the HPS. Connect the JTAG UART as depicted in Fig. 34

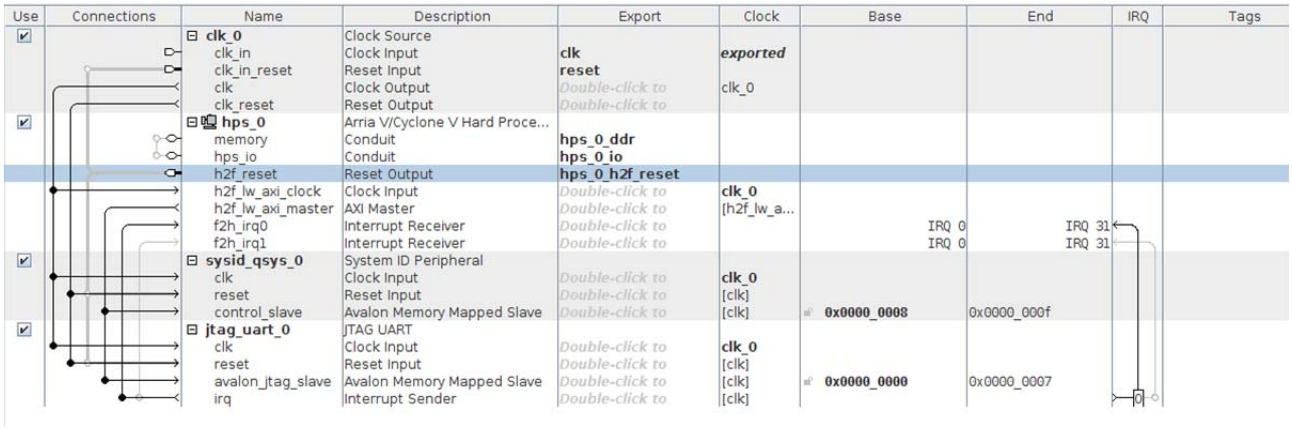


Fig. 34: Connection of the JTAG UART element.

3.4.4 Assigning the memory map for the peripherals.

15. In the main *Qsys* window, select "`System > Assign Base Addresses`" to get rid of any error messages regarding memory space overlaps among the different components of the system.

3.4.5 Generating the Qsys System

16. Click on the "`Generate HDL`" button.
17. Select "VHDL" for "`Create HDL design files for synthesis`".
18. Click on the "`Generate`" button to generate the system.
19. Save the design and exit *Qsys*. When asked if you want to generate the design, select "No", as we have already done it in the previous step.

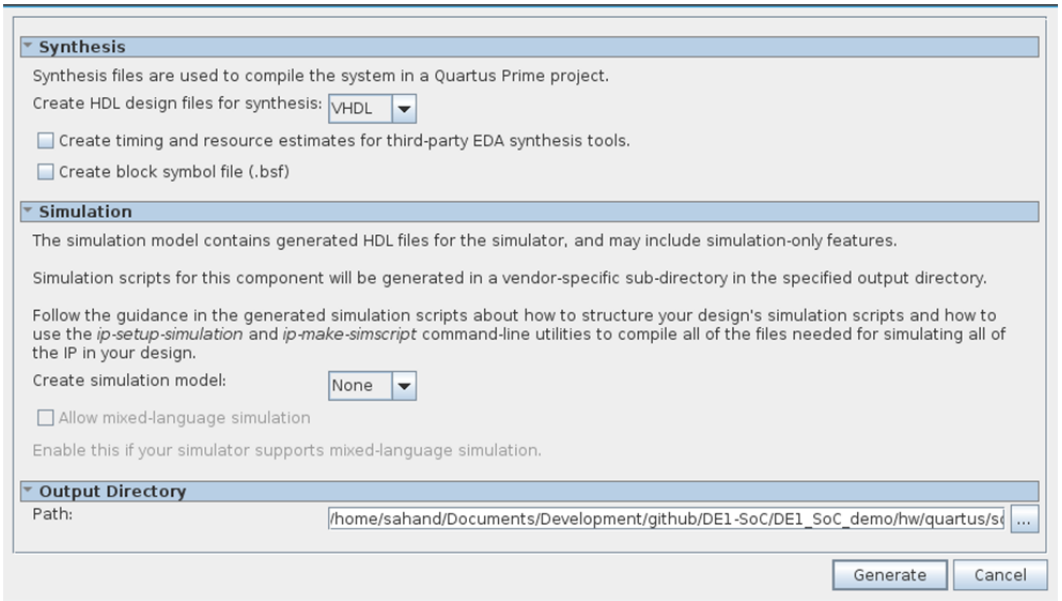


Fig. 35 Generating the SoC System with QSYS

3.5 Instantiating the Qsys System

You now have a complete *Qsys* system. The system will be available as an instantiable component in your design files. However, in order for *Quartus Prime* to see the *Qsys* system, you will have to add the system's files to your *Quartus Prime* project.

1. Add "`./soc_system/synthesis/soc_system.qip`" to the *Quartus Prime* project by using "`Project > Add/Remove Files in Project…`".
2. To use the *Qsys* system in your design, you have to declare its component, and then instantiate it. Qsys already provides you with a component declaration. You can find it among the numerous files that were generated. The one we are looking for is "`./soc_system/soc_system.cmp`".
3. Copy the component declaration code in the declarative zone of the architecture body of "`./soc_system_top.vhd`", just above the *process* keyword. Be sure to instantiate the component and assign all the correct pins of the DE1-SoC board. For our demo project, we would use the instantiation shown below. Don't forget to **remove** the port "hps_p_h2f_reset_reset_n" because we are not using this resource for the time being.

```
soc_system_inst : component soc_system
  port map(
      clk_clk                                => CLOCK_50,
      hps_0_ddr_mem_a                        => HPS_DDR3_ADDR,
      hps_0_ddr_mem_ba                       => HPS_DDR3_BA,
      hps_0_ddr_mem_ck                       => HPS_DDR3_CK_P,
      hps_0_ddr_mem_ck_n                     => HPS_DDR3_CK_N,
      hps_0_ddr_mem_cke                      => HPS_DDR3_CKE,
      hps_0_ddr_mem_cs_n                     => HPS_DDR3_CS_N,
      hps_0_ddr_mem_ras_n                    => HPS_DDR3_RAS_N,
      hps_0_ddr_mem_cas_n                    => HPS_DDR3_CAS_N,
      hps_0_ddr_mem_we_n                     => HPS_DDR3_WE_N,
      hps_0_ddr_mem_reset_n                  => HPS_DDR3_RESET_N,
      hps_0_ddr_mem_dq                       => HPS_DDR3_DQ,
      hps_0_ddr_mem_dqs                      => HPS_DDR3_DQS_P,
      hps_0_ddr_mem_dqs_n                    => HPS_DDR3_DQS_N,
      hps_0_ddr_mem_odt                      => HPS_DDR3_ODT,
      hps_0_ddr_mem_dm                       => HPS_DDR3_DM,
      hps_0_ddr_oct_rzqin                    => HPS_DDR3_RZQ,
      hps_0_io_hps_io_emac1_inst_TX_CLK      => HPS_ENET_GTX_CLK,
      hps_0_io_hps_io_emac1_inst_TX_CTL      => HPS_ENET_TX_EN,
      hps_0_io_hps_io_emac1_inst_TXD0        => HPS_ENET_TX_DATA(0),
      hps_0_io_hps_io_emac1_inst_TXD1        => HPS_ENET_TX_DATA(1),
      hps_0_io_hps_io_emac1_inst_TXD2        => HPS_ENET_TX_DATA(2),
      hps_0_io_hps_io_emac1_inst_TXD3        => HPS_ENET_TX_DATA(3),
      hps_0_io_hps_io_emac1_inst_RX_CLK      => HPS_ENET_RX_CLK,
      hps_0_io_hps_io_emac1_inst_RX_CTL      => HPS_ENET_RX_DV,
      hps_0_io_hps_io_emac1_inst_RXD0        => HPS_ENET_RX_DATA(0),
      hps_0_io_hps_io_emac1_inst_RXD1        => HPS_ENET_RX_DATA(1),
      hps_0_io_hps_io_emac1_inst_RXD2        => HPS_ENET_RX_DATA(2),
      hps_0_io_hps_io_emac1_inst_RXD3        => HPS_ENET_RX_DATA(3),
      hps_0_io_hps_io_emac1_inst_MDIO        => HPS_ENET_MDIO,
      hps_0_io_hps_io_emac1_inst_MDC         => HPS_ENET_MDC,
      hps_0_io_hps_io_qspi_inst_CLK          => HPS_FLASH_DCLK,
      hps_0_io_hps_io_qspi_inst_SS0          => HPS_FLASH_NCSO,
      hps_0_io_hps_io_qspi_inst_IO0          => HPS_FLASH_DATA(0),
      hps_0_io_hps_io_qspi_inst_IO1          => HPS_FLASH_DATA(1),
      hps_0_io_hps_io_qspi_inst_IO2          => HPS_FLASH_DATA(2),
      hps_0_io_hps_io_qspi_inst_IO3          => HPS_FLASH_DATA(3),
      hps_0_io_hps_io_sdio_inst_CLK          => HPS_SD_CLK,
      hps_0_io_hps_io_sdio_inst_CMD          => HPS_SD_CMD,
      hps_0_io_hps_io_sdio_inst_D0           => HPS_SD_DATA(0),
```

```
      hps_0_io_hps_io_sdio_inst_D1          => HPS_SD_DATA(1),
      hps_0_io_hps_io_sdio_inst_D2          => HPS_SD_DATA(2),
      hps_0_io_hps_io_sdio_inst_D3          => HPS_SD_DATA(3),
      hps_0_io_hps_io_usb1_inst_CLK         => HPS_USB_CLKOUT,
      hps_0_io_hps_io_usb1_inst_STP         => HPS_USB_STP,
      hps_0_io_hps_io_usb1_inst_DIR         => HPS_USB_DIR,
      hps_0_io_hps_io_usb1_inst_NXT         => HPS_USB_NXT,
      hps_0_io_hps_io_usb1_inst_D0          => HPS_USB_DATA(0),
      hps_0_io_hps_io_usb1_inst_D1          => HPS_USB_DATA(1),
      hps_0_io_hps_io_usb1_inst_D2          => HPS_USB_DATA(2),
      hps_0_io_hps_io_usb1_inst_D3          => HPS_USB_DATA(3),
      hps_0_io_hps_io_usb1_inst_D4          => HPS_USB_DATA(4),
      hps_0_io_hps_io_usb1_inst_D5          => HPS_USB_DATA(5),
      hps_0_io_hps_io_usb1_inst_D6          => HPS_USB_DATA(6),
      hps_0_io_hps_io_usb1_inst_D7          => HPS_USB_DATA(7),
      hps_0_io_hps_io_spim1_inst_CLK        => HPS_SPIM_CLK,
      hps_0_io_hps_io_spim1_inst_MOSI       => HPS_SPIM_MOSI,
      hps_0_io_hps_io_spim1_inst_MISO       => HPS_SPIM_MISO,
      hps_0_io_hps_io_spim1_inst_SS0        => HPS_SPIM_SS,
      hps_0_io_hps_io_uart0_inst_RX         => HPS_UART_RX,
      hps_0_io_hps_io_uart0_inst_TX         => HPS_UART_TX,
      hps_0_io_hps_io_i2c0_inst_SDA         => HPS_I2C1_SDAT,
      hps_0_io_hps_io_i2c0_inst_SCL         => HPS_I2C1_SCLK,
      hps_0_io_hps_io_i2c1_inst_SDA         => HPS_I2C2_SDAT,
      hps_0_io_hps_io_i2c1_inst_SCL         => HPS_I2C2_SCLK,
      hps_0_io_hps_io_gpio_inst_GPIO09      => HPS_CONV_USB_N,
      hps_0_io_hps_io_gpio_inst_GPIO35      => HPS_ENET_INT_N,
      hps_0_io_hps_io_gpio_inst_GPIO40      => HPS_LTC_GPIO,
      hps_0_io_hps_io_gpio_inst_GPIO48      => HPS_I2C_CONTROL,
      hps_0_io_hps_io_gpio_inst_GPIO53      => HPS_LED,
      hps_0_io_hps_io_gpio_inst_GPIO54      => HPS_KEY_N,
      hps_0_io_hps_io_gpio_inst_GPIO61      => HPS_GSENSOR_INT,
      reset_reset_n                         => '1'
  );
```

4. After finishing the design, ***COMMENT*** all unused pins from the top-level VHDL file. Your top-level entity should look like this.

```
entity soc_system_top is
    port(
     -- -- ADC
     -- ADC_CS_n          : out   std_logic;
     -- ADC_DIN           : out   std_logic;
     -- ADC_DOUT          : in    std_logic;
     -- ADC_SCLK          : out   std_logic;

     -- -- Audio
     -- AUD_ADCDAT        : in    std_logic;
     -- AUD_ADCLRCK       : inout std_logic;
     -- AUD_BCLK          : inout std_logic;
     -- AUD_DACDAT        : out   std_logic;
     -- AUD_DACLRCK       : inout std_logic;
     -- AUD_XCK           : out   std_logic;

     -- -- CLOCK
        CLOCK_50          : in    std_logic;
     -- CLOCK2_50         : in    std_logic;
     -- CLOCK3_50         : in    std_logic;
```

```
      -- CLOCK4_50        : in    std_logic;

      -- -- SDRAM
         DRAM_ADDR        : out   std_logic_vector(12 downto 0);
         DRAM_BA          : out   std_logic_vector(1 downto 0);
         DRAM_CAS_N       : out   std_logic;
         DRAM_CKE         : out   std_logic;
         DRAM_CLK         : out   std_logic;
         DRAM_CS_N        : out   std_logic;
         DRAM_DQ          : inout std_logic_vector(15 downto 0);
         DRAM_LDQM        : out   std_logic;
         DRAM_RAS_N       : out   std_logic;
         DRAM_UDQM        : out   std_logic;
         DRAM_WE_N        : out   std_logic;

      -- -- I2C for Audio and Video-In
      -- FPGA_I2C_SCLK    : out   std_logic;
      -- FPGA_I2C_SDAT    : inout std_logic;

      -- -- SEG7
         HEX0_N           : out   std_logic_vector(6 downto 0);
         HEX1_N           : out   std_logic_vector(6 downto 0);
         HEX2_N           : out   std_logic_vector(6 downto 0);
         HEX3_N           : out   std_logic_vector(6 downto 0);
         HEX4_N           : out   std_logic_vector(6 downto 0);
         HEX5_N           : out   std_logic_vector(6 downto 0);

      -- -- IR
      -- IRDA_RXD         : in    std_logic;
      -- IRDA_TXD         : out   std_logic;

      -- -- KEY_N
         KEY_N            : in    std_logic_vector(3 downto 0);

      -- -- LED
         LEDR             : out   std_logic_vector(9 downto 0);

      -- -- PS2
      -- PS2_CLK          : inout std_logic;
      -- PS2_CLK2         : inout std_logic;
      -- PS2_DAT          : inout std_logic;
      -- PS2_DAT2         : inout std_logic;

      -- -- SW
         SW               : in    std_logic_vector(9 downto 0);

      -- -- Video-In
      -- TD_CLK27         : inout std_logic;
      -- TD_DATA          : out   std_logic_vector(7 downto 0);
      -- TD_HS            : out   std_logic;
      -- TD_RESET_N       : out   std_logic;
      -- TD_VS            : out   std_logic;

      -- -- VGA
      -- VGA_B            : out   std_logic_vector(7 downto 0);
      -- VGA_BLANK_N      : out   std_logic;
      -- VGA_CLK          : out   std_logic;
      -- VGA_G            : out   std_logic_vector(7 downto 0);
      -- VGA_HS           : out   std_logic;
      -- VGA_R            : out   std_logic_vector(7 downto 0);
      -- VGA_SYNC_N       : out   std_logic;
```

```
      -- VGA_VS           : out   std_logic;

      -- -- GPIO_0
      -- GPIO_0           : inout std_logic_vector(35 downto 0);

      -- -- GPIO_1
      -- GPIO_1           : inout std_logic_vector(35 downto 0);

      -- -- HPS
         HPS_CONV_USB_N   : inout std_logic;
         HPS_DDR3_ADDR    : out   std_logic_vector(14 downto 0);
         HPS_DDR3_BA      : out   std_logic_vector(2 downto 0);
         HPS_DDR3_CAS_N   : out   std_logic;
         HPS_DDR3_CK_N    : out   std_logic;
         HPS_DDR3_CK_P    : out   std_logic;
         HPS_DDR3_CKE     : out   std_logic;
         HPS_DDR3_CS_N    : out   std_logic;
         HPS_DDR3_DM      : out   std_logic_vector(3 downto 0);
         HPS_DDR3_DQ      : inout std_logic_vector(31 downto 0);
         HPS_DDR3_DQS_N   : inout std_logic_vector(3 downto 0);
         HPS_DDR3_DQS_P   : inout std_logic_vector(3 downto 0);
         HPS_DDR3_ODT     : out   std_logic;
         HPS_DDR3_RAS_N   : out   std_logic;
         HPS_DDR3_RESET_N : out   std_logic;
         HPS_DDR3_RZQ     : in    std_logic;
         HPS_DDR3_WE_N    : out   std_logic;
         HPS_ENET_GTX_CLK : out   std_logic;
         HPS_ENET_INT_N   : inout std_logic;
         HPS_ENET_MDC     : out   std_logic;
         HPS_ENET_MDIO    : inout std_logic;
         HPS_ENET_RX_CLK  : in    std_logic;
         HPS_ENET_RX_DATA : in    std_logic_vector(3 downto 0);
         HPS_ENET_RX_DV   : in    std_logic;
         HPS_ENET_TX_DATA : out   std_logic_vector(3 downto 0);
         HPS_ENET_TX_EN   : out   std_logic;
         HPS_FLASH_DATA   : inout std_logic_vector(3 downto 0);
         HPS_FLASH_DCLK   : out   std_logic;
         HPS_FLASH_NCSO   : out   std_logic;
         HPS_GSENSOR_INT  : inout std_logic;
         HPS_I2C_CONTROL  : inout std_logic;
         HPS_I2C1_SCLK    : inout std_logic;
         HPS_I2C1_SDAT    : inout std_logic;
         HPS_I2C2_SCLK    : inout std_logic;
         HPS_I2C2_SDAT    : inout std_logic;
         HPS_KEY_N        : inout std_logic;
         HPS_LED          : inout std_logic;
         HPS_LTC_GPIO     : inout std_logic;
         HPS_SD_CLK       : out   std_logic;
         HPS_SD_CMD       : inout std_logic;
         HPS_SD_DATA      : inout std_logic_vector(3 downto 0);
         HPS_SPIM_CLK     : out   std_logic;
         HPS_SPIM_MISO    : in    std_logic;
         HPS_SPIM_MOSI    : out   std_logic;
         HPS_SPIM_SS      : inout std_logic;
         HPS_UART_RX      : in    std_logic;
         HPS_UART_TX      : out   std_logic;
         HPS_USB_CLKOUT   : in    std_logic;
         HPS_USB_DATA     : inout std_logic_vector(7 downto 0);
         HPS_USB_DIR      : in    std_logic;
         HPS_USB_NXT      : in    std_logic;
         HPS_USB_STP      : out   std_logic
```

```
    );
end entity soc_system_top; end entity DE1_SoC_top_level;
```

3.6 HPS DDR3 Pin assignments

In a normal FPGA design flow, you would be able to compile your design at this stage. However, this isn't possible at the moment in our design. The reason is that the HPS' DDR3 pins assignments have not been performed yet. How is this possible? We said earlier that our TCL script assigns pin locations and I/O standards to all pins names in "soc_system_top.vhd". The truth is that it assigns values for all pin names, except those related to the HPS' DDR3 memory. The reason is that the DDR3 pin assignments depend on how you parameterize the HPS memory timings in *Qsys*. Our TCL script could not have known what timings you were going to use, so it doesn't set those pin locations and I/O standards.

However, *Qsys* knows what the parameters are (since you provided it with all the necessary information), and it has generated a custom TCL script for the HPS DDR3 pin assignments.

1. Start the "Analysis and Synthesis" flow to perform a preliminary analysis of the system.
2. Go to "Tools > Tcl Scripts…" in *Quartus Prime*.

[Note]: If at this point you do not see the same thing as on Fig. 36, then close and relaunch quartus prime again. Some versions of quartus prime suffer from a bug, where the program doesn't correctly detect tcl files generated by qsys. You should see the same thing as on Fig. 36.

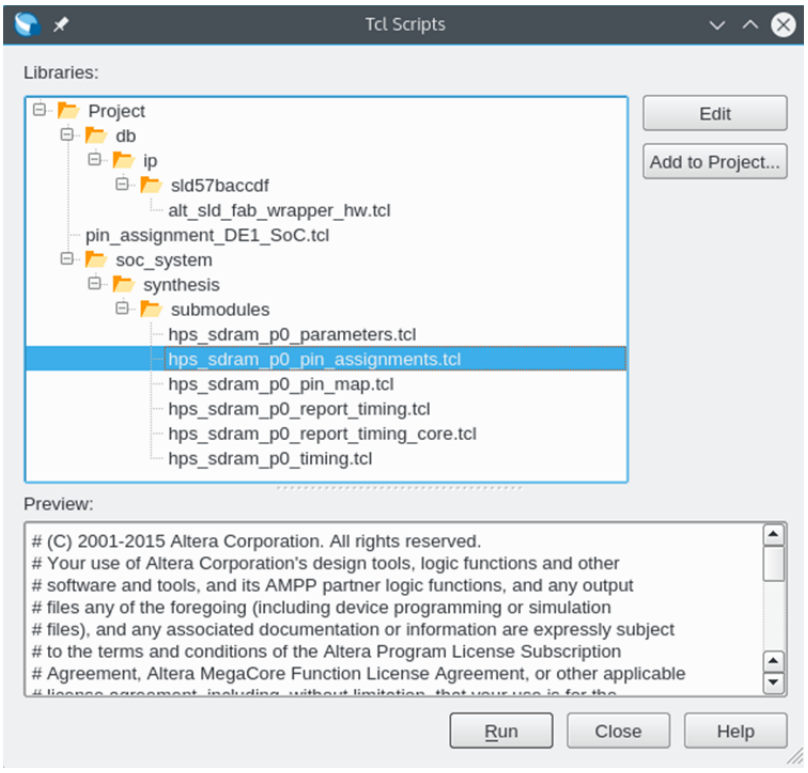


Fig. 36: Correct HPS DDR3 Pin Assignment TCL Script Selection

3. Execute "hps_sdram_p0_pin_assignments.tcl".

3.7 Compiling the design. Generation of the sof and rbf file

You can now start the full compilation of your design with the "Start Compilation" flow. You could find errors in the compilation. In that case, review the previous steps. Probably your errors are related to the use of different labels. The compilation time depends on your computer performance, but it will be in the order of 6 to 15 minutes.

The output of the compilation is a file with extension .sof. The HPS (the ARM processor) must program the FPGA by using a *Raw Binary File* (`.rbf`). Therefore, we must convert the `.sof` file to the `.rbf`. Execute the following command to convert the .sof file to an .rbf file. You need to open a SoC EDS Command Shell. Change to the folder with the sof file and execute:

```
$<altera>/16.0/embedded/embedded_command_shell.sh
```

```
$quartus_cpf -c -o bitstream_compression=on soc_system_top.sof
soc_system.rbf
```

3.8 Testing DE1-SoC with Linux prebuilt binaries.

The objective of this step is to verify the hardware implementation in the SoC system using an embedded Linux OS. The prebuilt binaries are provided to boot the system. The booting process of the Cyclone5 SoC with its AMR A9 processor is depicted in Fig. 37. This figure summarizes the complete boot process in the DE1-SoC. In a later paragraph, we will provide more details of this. In this point, we are going to focus in the initial stages of the process. The boot process needs an application known as a preloader. It provides the ability to load a loader, named u-boot that will boot the Linux kernel. The next paragraph explains how to generate this preloader. The DE1-SoC uses an SD Card for this booting process. There is another paragraph explaining how to format the SD Card. Finally, the binaries with the different files are downloaded and written in the SDCard to perform the boot.

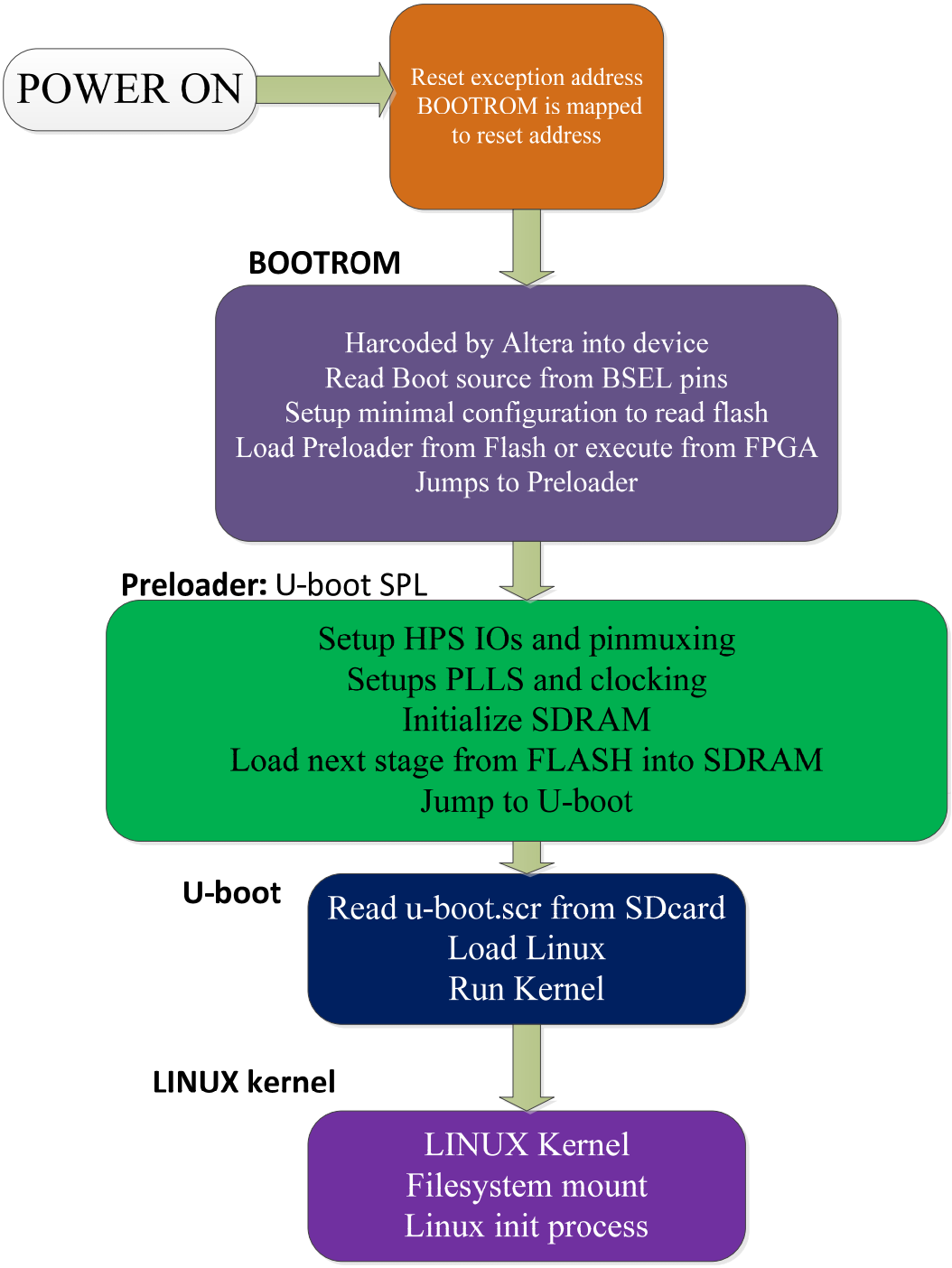


Fig. 37: Booting process for DE1-SOC.

3.8.1 Preloader Generation

1. Start an Embedded Command Shell (you will this in the terminal windows, see Fig. 38) and go to the project folder (assumed here to be saved in the home folder):

```
$ ~/altera_lite/16.0/embedded/embedded_command_shell.sh
$ cd ~/"name of your Project folder"
```

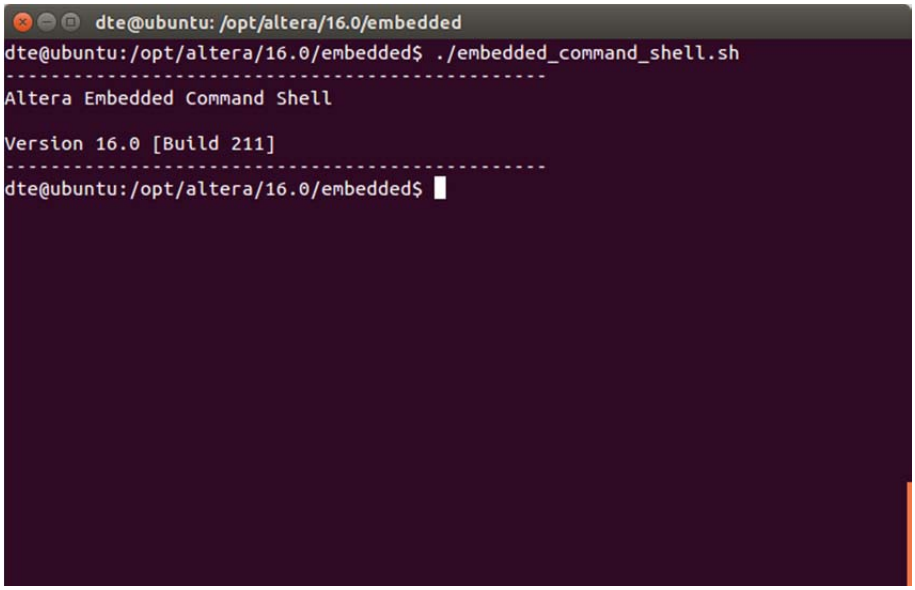


Fig. 38: Terminal window running Altera Embedded Command Shell

2. Start the Preloader Generator **BSP Editor running this command**

```
$ bsp-editor
```

3. You will see a screen as depicted in Fig. 39. Choose "File > New HPS BSP…" and complete the information requested (see).

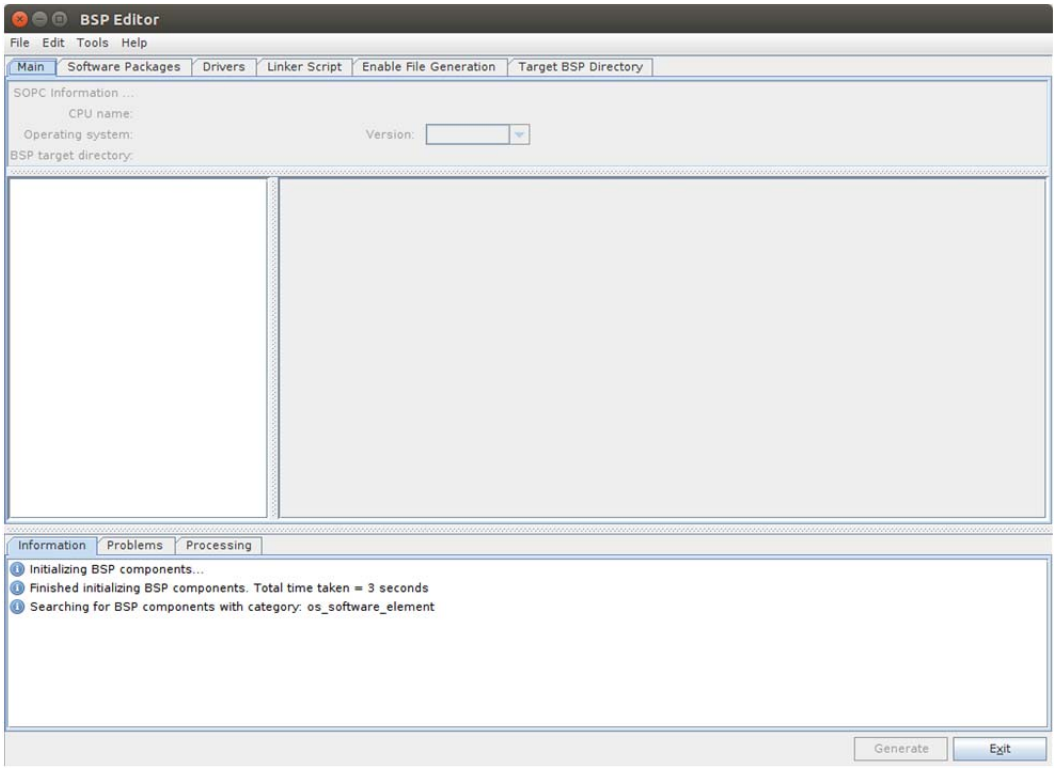


Fig. 39: bsp-editor main window

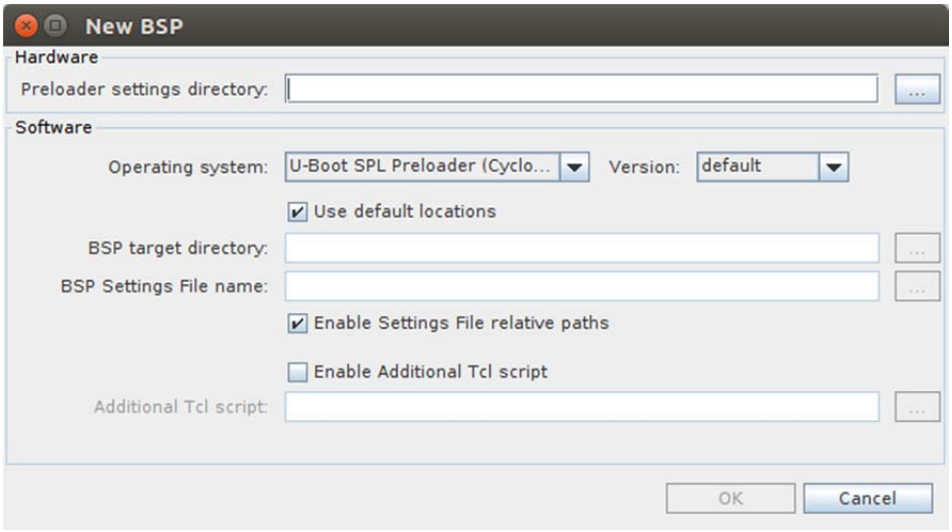


Fig. 40: Information necessary for building the preloader.

4. The preloader will need to know which of the HPS' peripherals were enabled so it can appropriately initialize them in the boot process. Under **"Preloader settings directory"**, select the **"./hps_isw_handoff/soc_system_hps_0"** directory.
5. This directory contains settings about the HPS' ***HARD*** peripherals, as configured in the **"Arria V/Cyclone V Hard Processor System" component** in *Qsys*.
6. **"Use default locations"** checkbox and under the **"BSP target directory**. You should have something similar to Fig. 41.

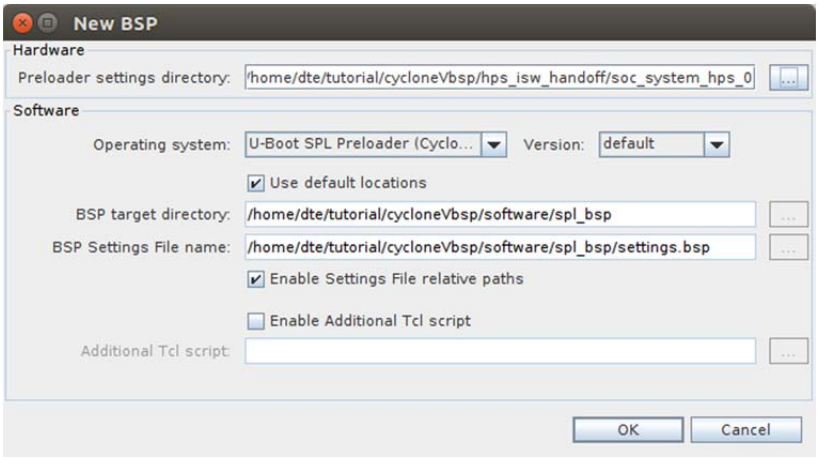


Fig. 41: New BSP Dialog

7. Press the "OK" button. You should then arrive at a page with many settings, as shown in Fig. 42. The **New BSP** window will have all the settings populated, based on the handoff folder. Take some time to read through them to see what the preloader can do.

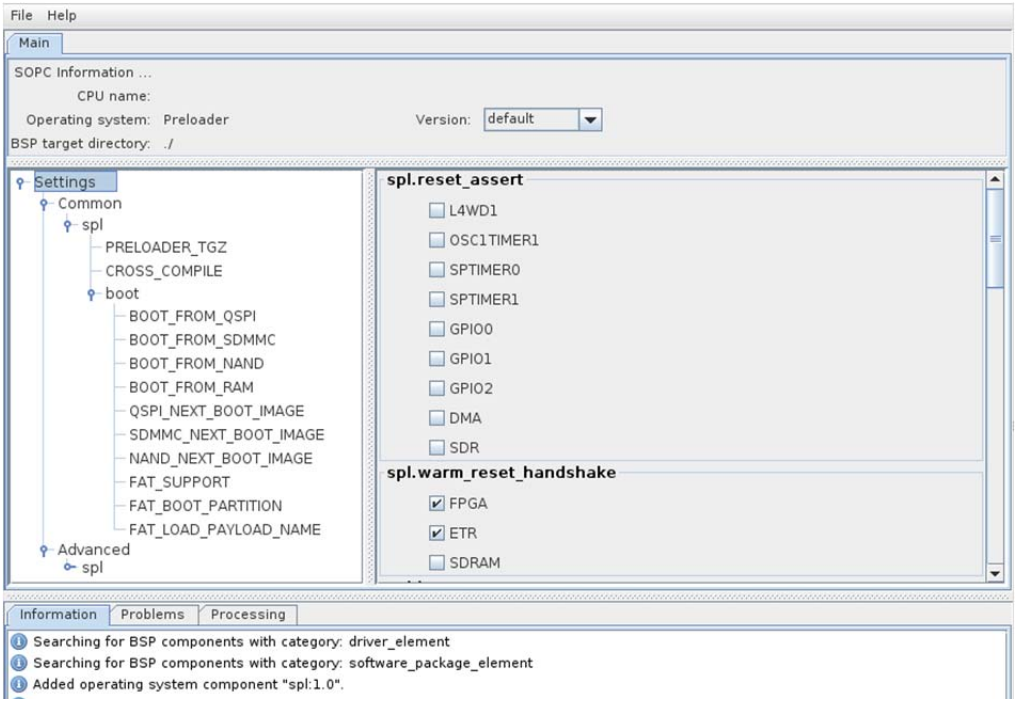


Fig. 42: Preloader Settings Dialog

8. Press the "`Generate`" button to finish. You can then exit the `bsp-editor` pressing "`Exit`".

This bsp-editor has generated all the files supporting the construction of a Singe U-boot Preloder. The folder contains some source files (c and h) with the specific configuration of your HPS.

9. Execute the following command to build the preloader.

```
$ cd software/spl_bsp
$ make
```

The files listed in Table 6 are built in the "./**cyclonevbsp/software/spl_bsp/"** folder. We are going to use the preloader-mlpimage.bin. This file includes a special header necessary in order to validate the image to be loaded. The u-boot image is "signed" using the mkpimage tool [3].

Table 6: Files generated when compiling the u-boot SPL

File	Description
uboot-socfpga/spl/u-boot-spl	Preloader ELF file
uboot-socfpga/spl/u-boot-spl.bin	Preloader binary file
preloader-mkpimage.bin	Preloader image with the BootROM required header

If you ever decide to move the project directory defined, you will have to regenerate the preloader. Unfortunately, the script provided by Altera which generates the preloader hard-codes multiple absolute paths directly in the resulting files, rendering them useless once moved.

3.8.2 Generating the device tree.

Another important file to be generated to boot the embedded Linux is the device tree. This file (the binary version –device tree blob-) contains the description of the hardware of our embedded platform. In a static hardware configuration this file never changes but if we have a SoC with a configurable hardware such us the FPGA part in our SoC, therefore, we need to know the method to generate the device tree.

This file is generated using the hardware description available in the "soc_system.sopinfo" file and additional support files named soc_system_board_info.xml and hps_common_board_info.xml (files available also in [4]). The command necessary (in a SoC EDS command shell) to obtain the device tree blob file is this:

```
$ sopc2dts --input soc_system.sopcinfo\
  --output soc_system.dtb  --type dtb\
  --board soc_system_board_info.xml\
  --board hps_common_board_info.xml\
  --clocks
```

Every time that you change the hardware in the FPGA you need to obtain a new device tree. The previous command always obtains the device tree in binary form. If you change –type dtb by –type dts you will obtain the device tree source file (the output filename should be soc_system.dts).

3.8.3 Creating the u-boot.scr script.

The FPGA can be configured (also known as 'programmed') in several ways:

- From an external configuration flash memory,
- With the Quartus Programmer tool,
- From HPS software (from the preloader, from u-boot, or from Linux)

The FPGA is configured from U-boot with the help of a script. This scheme ensures that the Linux boot succeeds even if the FPGA image is not present on the SD card.

If any failure during FPGA programming in U-Boot, it will return an error message with a value. This value will represent the stages where the programming failed. Here are the numbers and error stages.

Table 7: Error displayed by u-boot when configuring the FPGA.

Error No	Description
-1	The FPGA is not able to enter reset phase after FPGA reset request being made at FPGA Manager
-2	The FPGA is not able to enter configuration phase after FPGA reset release request being made at FPGA Manager
-3	The FPGA is having config error after enable AXI configuration at FPGA Manager
-4	The FPGA is having timeout from entering config done phase after enable AXI configuration at FPGA Manager
-5	The FPGA is having timeout from getting dclkcnt done signal after enable the dclkcnt delay.
-6	The FPGA is having timeout from entering init phase or user mode after disable AXI configuration at FPGA Manager
-7	The FPGA is having timeout from getting dclkcnt done signal after enable the dclkcnt delay. This is happen after FPGA reaching init phase or user mode
-8	The FPGA is having timeout from getting into final stage which is user mode

If the FPGA image is not present, Linux will still boot, but the FPGA will need to be configured using another method.
These are the steps needed to create the script:

1. On host PC, create text file “boot.script” with the following contents:

```
fatload mmc 0:1 $fpgadata soc_system.rbf;
fpga load 0 $fpgadata $filesize;
setenv fdtimage soc_system.dtb;
run bridge_enable_handoff;
run mmcload;
run mmcboot;
```

2. Add the U-boot header to the “boot.script” file to create the u-boot.scr file. Execute this in a SoC EDS command shell:

```
$ ./software/spl_bsp/uboot-socfpga/tools/mkimage  -A arm -O linux -T script -
C none -a 0 -e 0 -n "My script" -d boot.script u-boot.scr
```

3.8.4 Creating the sdcard image

This section presents the layout of the SD card image used by the de1-soc, together with the instructions on how to create this SD card image, and also how to update individual elements on the SD card. Moreover, it will be presented details on how to create the bootable SD card image. Fig. 43 presents the layout of the SD

card that is used by the de1-soc. Partition 3 is a custom partition with type 0xA2. It is required by the boot ROM, which will identify it from the MBR and load the Preloader from the beginning of it.

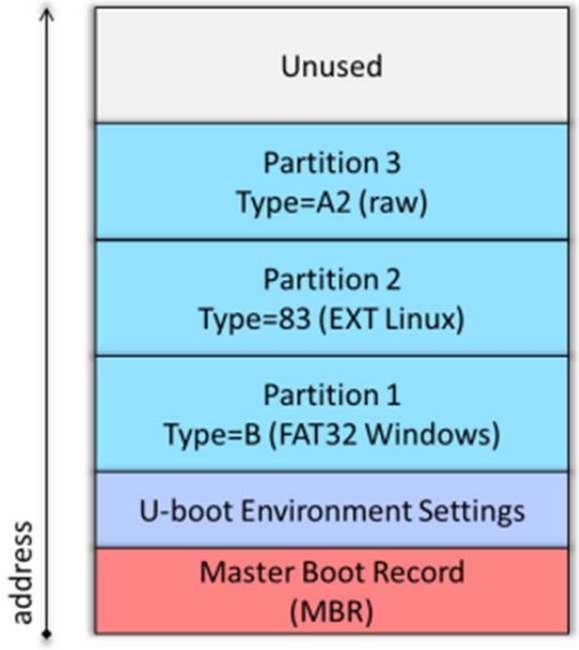


Fig. 43: SD card organization

Table 8 summarizes the information that is stored on the SD card and how each file is generated:

Table 8: Organization of the partitions in the SD card.

Location	File System	File Name	Description	Source of File
Partition 1	FAT32	soc_system.dtb	Device Tree Blob file	Build with `sopc2dts` utility
Partition 1	FAT32	soc_system.rbf	Compressed FPGA configuration file	Build with quartus, convert .sof to .rbf
Partition 1	FAT32	u-boot.scr	U-boot script for configuring FPGA	Build with boot.script and `mkimage` utility
Partition 1	FAT32	zImage	Compressed Linux kernel image file	Generated by Buildroot with `make linux-build`
Partition 2	EXT3	*various*	Linux root filesystem	Generated by Buildroot with `make`
Partition 3	A2 raw	preloader-mkpimage.bin	Preloader image	Generated with SoC EDS BSP Editor based on Quartus handoff information
Partition 3	A2 raw	u-boot.img	U-boot image	Generated by buildroot with `make uboot-build`

The Altera Golden Software Reference Design (GSRD) includes a Python script that can be used to create a bootable SD card image from the items mentioned in the table above. The script is named make_sdimage.py and is available also at [4]. In order to obtain more information about the script and its use, run it with the '-h' option:

```
$ sudo ~/make_sdimage.py -h
usage: make_sdimage.py [-h] [-P PART_ARGS] [-s SIZE] [-n IMAGE_NAME] [-f]

Creates an SD card image for Altera's SoCFPGA SoC's
optional arguments:
-h, --help     show this help message and exit
-P PART_ARGS   specifies a partition. May be used multiple times. file[,file
               ,...],num=,format=,
               size=[,type=ID]
-s SIZE        specifies the size of the image. Units K|M|G can be used.
-n IMAGE_NAME  specifies the name of the image.
-f             deletes the image file if exists

Usage: PROG [-h] -P  [-P ...] -P
```

When building the files using the tool flows, use the Table 9 to help locate the files after the build completes. It can be helpful to copy all the required files to a sdcard directory along with the make_sdimage.py script and run the script from that sdcard directory. The names used in the table match those in the *cyclonevbsp* project and the example usage of the script shown below the table. Note that the file names listed in the table are links to the build output target files. You will need to follow the links to their target destination file and copy that file to the sdcard directory. The target file names include one or more of the following suffixes: machine name, version number, date/timestamp. The make_sdimage.py script does not support the use of links when specifying the filenames.

Table 9: List of files needed to build the SD card

Item	Source	File Location	Copy to sdcard directory as
Preloader image	Quartus EDS	./cyclonevbsp/software/spl_bsp/preloader-mkpimage.bin*	preloader-mkpimage.bin
Compressed FPGA configuration file		./cyclonevbsp /output_files/soc_system.rbf	soc_system.rbf
Device Tree Blob file		./cyclonevbsp /soc_system.dtb	soc_system.dtb
U-boot script for configuring FPGA		./cyclonevbsp /u-boot.scr	u-boot.scr
U-boot image	Buildroot	.<builroot>/output/images/	u-boot.img
Compressed Linux kernel image file		.<builroot>/output/images/	zImage
Linux root filesystem		.<builroot>/output/images/	*extract to rootfs.tar* (see below)

10. How to extract the rootfs.tar. In the sdcard folder

```
$ mkdir rootfs
$ cd  rootfs
$ sudo tar -xvf ../rootfs.tar
```

So far, some of these files have been created: soc_system.rbf, soc_system.dtb, preloader_mkpimage.bin and u-boot.src. Later, in the following chapters, the method for creating the rest of the files will be described. So, for the time being just, download the prebuilt binaries (zImage, u-boot.img and rootfs.tar) from this link [4].

11. Assemble the rebuilt binaries according to the table above.
12. Call the make_sdimage.py script:

```
$ sudo /usr/bin/python/make_sdimage.py \
    -f \
    -P preloader-mkpimage.bin,u-boot.img,num=3,format=raw,size=10M,type=A2 \
    -P rootfs/*,num=2,format=ext3,size=1200M \
    -P zImage,u-boot.scr,soc_system.rbf,soc_system.dtb,num=1,format=vfat,size=300M \
    -s 1700M  \
    -n de1soc-sd-card.img
```

3.8.5 Copying the image to the SD Card (Linux Host)

Determine the device name by running the command below before and after plugging in the SD card reader into the host.

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

The device name can also be identified using a GUI utility on your Linux host such as **gnome-disks**.
Next, clone the image to the SD card

```
$ sudo dd if=de1soc-sd-card.img of=/dev/sdx bs=1M
```

Replace "sdx" with the device name of the SD card on your host system. After the clone command finishes, flush the file system buffers:

```
$ sudo sync
```

The cloning process can take several minutes, and there is no feedback or progress indicator during the cloning process, leaving you to wonder if the system is hung. You can use the pv command to monitor the progress of the cloning process through a pipe.

$ sudo apt-get install pv

$ pv -<xxxx>.img | sudo dd of=/dev/sd<x> bs=1M

(replacing "sdx" with the device name of the SD card on your host system)

When the progress indicator shows 100%, wait for the command prompt to return and then flush the buffers:

$ sudo sync

It is time-consuming to write the whole SD image to the card each time a modification is made. Therefore it is often preferable to create the SD card and write it to the card once, then update different elements individually. The following table presents how each item can be updated individually:

Table 10: Procedure to copy files manually to the SD

File	Update Procedure

zImage	Mount DOS SD card partition 1 and replace file with new one:
	$ sudo mkdir sdcard
	$ sudo mount /dev/sdx1 sdcard/
	$ sudo cp <file_name> sdcard/
	$ sudo umount sdcard

soc_system.rbf	
soc_system.dtb	
u-boot.scr	
preloader-mkpimage.bin	$ sudo dd if=preloader-mkpimage.bin of=/dev/sdx3 bs=64k seek=0
u-boot-sockit.img	$ sudo dd if=u-boot.img of=/dev/sdx3 bs=64k seek=4
root filesystem	$ sudo dd if=rootfs.ext3 of=/dev/sdx2

Replace "sdx" in the command above with the device name of the SD card on your host system. You can find out the device name by running $ cat /proc/partitions before and after plugging in the card reader into the host.

3.8.6 Copying the image to the SD Card (Windows Host)

This section explains how to create the SD card necessary to boot Linux, using the SD card image.
The steps required to create the SD card are:

1. Get Disk Imaging software: Win32diskimager [7],
2. Insert the SD Adaptor. Use Windows Explorer to determine the drive letter.
3. Select the Image File. Navigate to de1soc-sdcard.img
4. Press the Write Button.

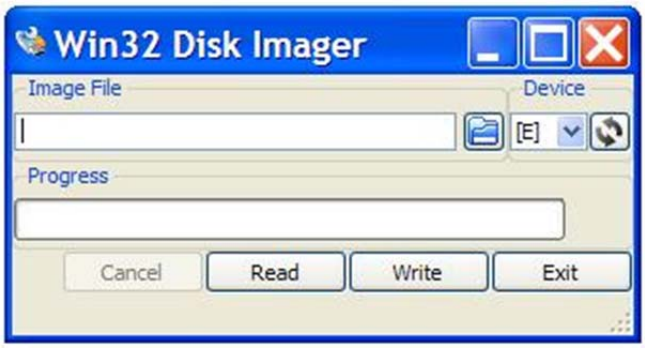


Fig. 44: Win32 Disk Imager utility

3.8.7 Connecting the DE1-SoC.

1. Check DE1-SoC switches and set as follows (Fig. 45):
    - MSEL0=0
    - MSEL1=1
    - MSEL2=0
    - MSEL3=1
    - MSEL4=0

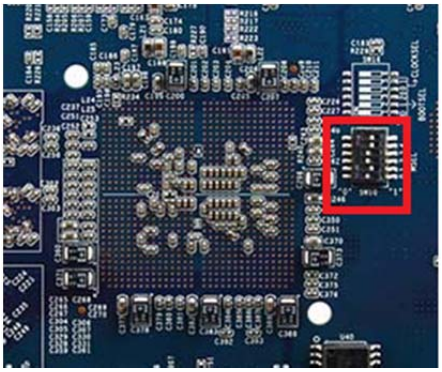


Fig. 45: MSEL switches on the bottom layer of DE1-soc.

2. Connect the DE1-SoC as shown in Fig. 46 :
    a. USB-Blaster cable is connected to the host computer.
    b. USB to microUSB (this is an RS232-USB cable) adapter. This adapter will provide the serial I
    :

> **[Connecting the USB serial interface in the VM]:** In order to connect the serial interface on your virtual machine you need to connect it selecting un VMWare Player the option Player->Removable Devices->Future Devices TTL 232R 3V3->Connect

    I
    nterface to be used as a console in the Linux operating system.
    c. Ethernet cable to the Ethernet switch. The network infrastructure must have a DHCP running.
    d. Power cable. Do not power on the card pressing the red button.

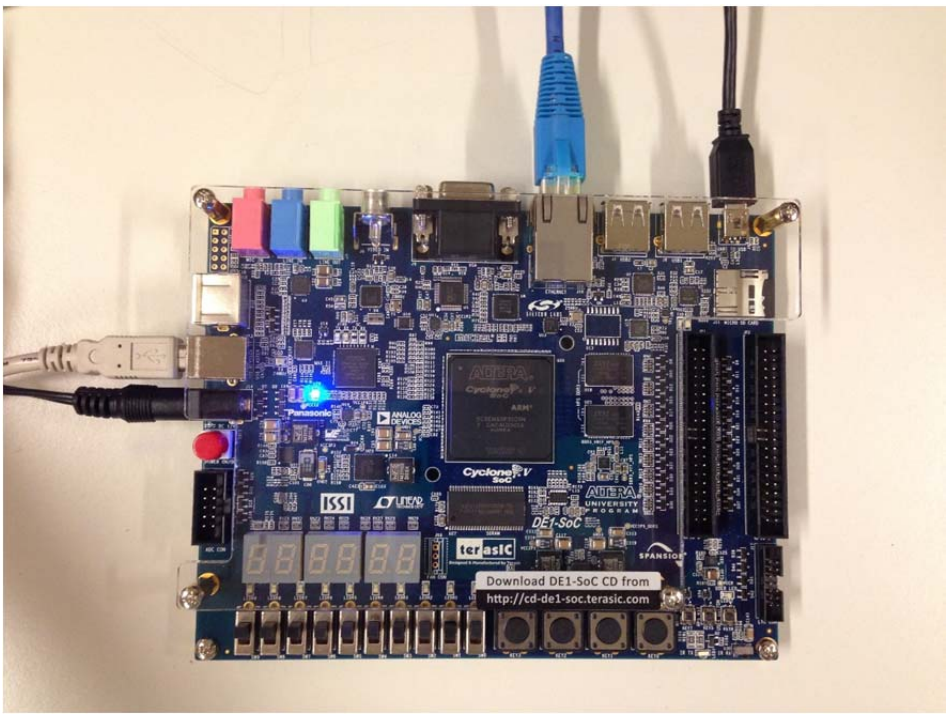


Fig. 46: DE1-SoC Wiring

3. Insert the SD card into the slot.

4. Open the putty application (Fig. 47) on the Linux host (execute “sudo putty” in a Linux terminal) and open a session to the serial line (use /dev/ttyUSB0 with 115200 baud rate). You will see a terminal with a black screen.

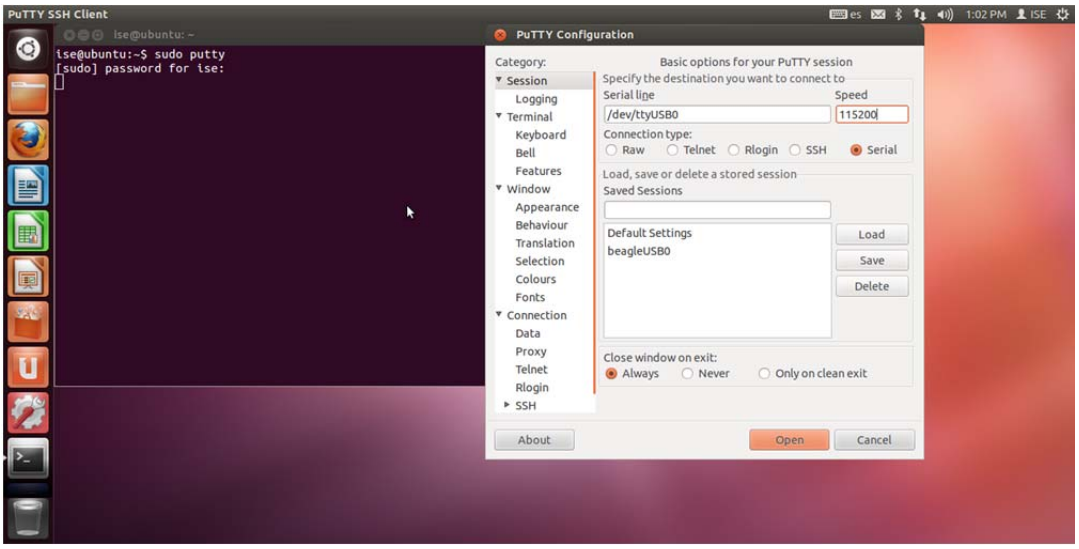


Fig. 47: Putty program main window.

[Serial interface identification in Linux]: In Linux, the serial devices are identified typically with the names /dev/ttyS0, /dev/ttyS1, etc. In the figure, the example has been checked with a serial port implemented with an USB-RS232 converter. This is the reason of why the name is /dev/ttyUSB0. In your computer, you need to find the identification of your serial port. Use Linux **dmesg** command to do this.

Apply the power pressing the red button. After some seconds, you will see a lot of messages displaying in the terminal. Linux kernel is booting and the operating system is running their configuration and initial daemons. If the system boots correctly you will see an output like the represented in Fig. 48. Introduce the user name (root), the passwd (hola) and the Linux shell will be available for you.

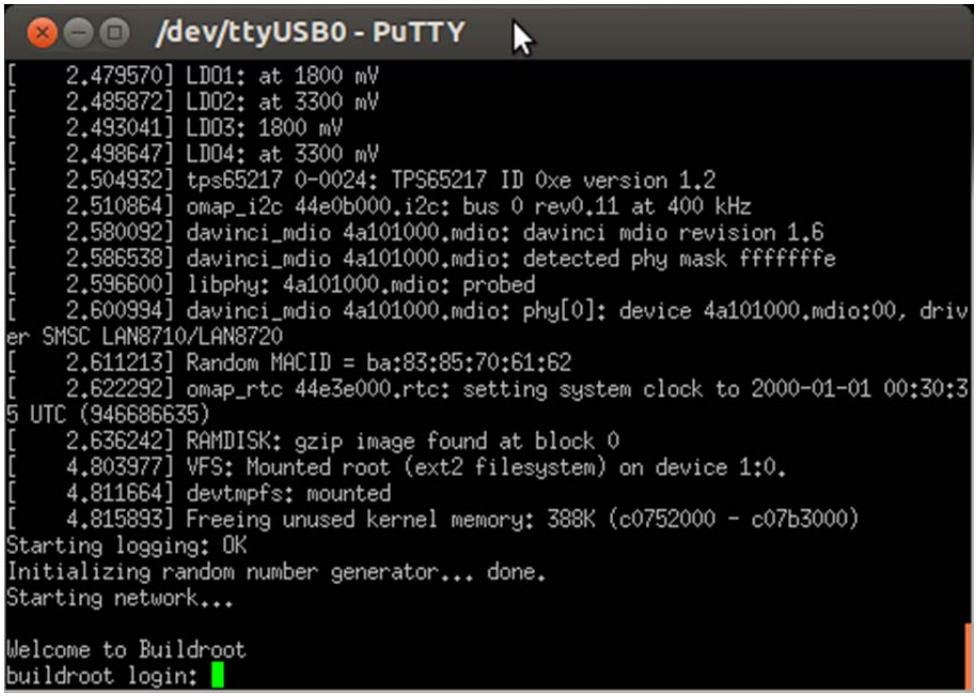


Fig. 48: Running Linux. Your screen could be different.

4 BUILDING LINUX USING BUILDROOT

4.1 Starting the VMware

Start VMware Player and open the Ubuntu Virtual Machine. Wait until the welcome screen is displayed (see Fig. 49 and Fig. 50). Login as *"ubuntuBuildroot"* user using the password "dte.2016". An Ubuntu tutorial is available at Moodle site.

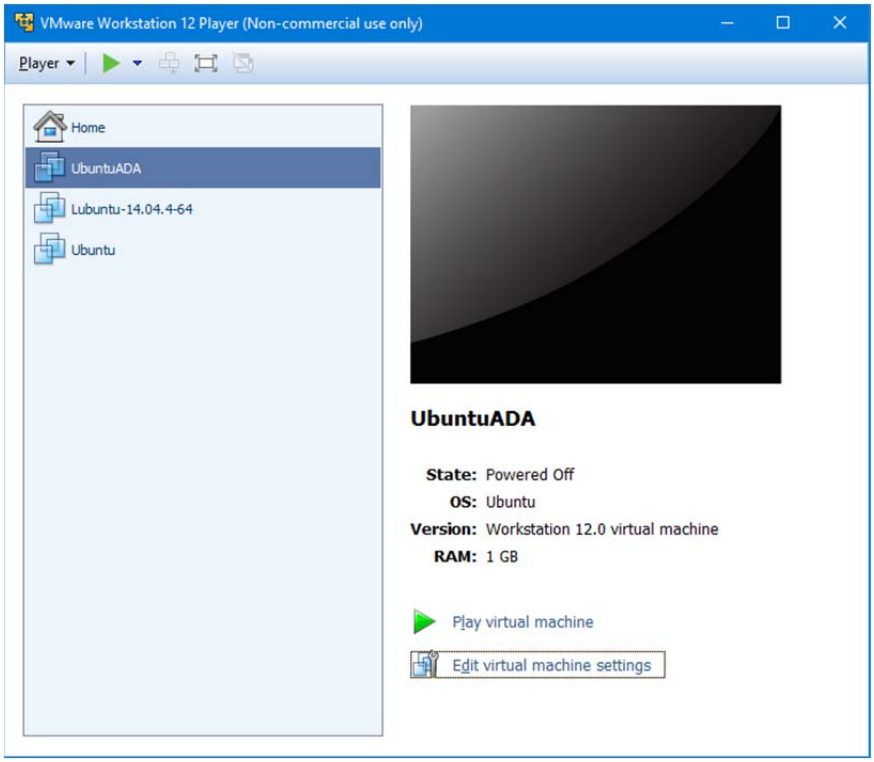


Fig. 49: Main screen of VMware player with some VM available to be executed.

Fig. 50: Ubuntu Virtual Machine login screen.

Open the **Firefox** web browser and download from https://buildroot.org/ the version identified as buidlroot2016-08 (use the download link, see Fig. 51, and navigate searching for a specific release, https://buildroot.org/downloads/). Save the file to the **Documents** folder in your account (Fig. 52).

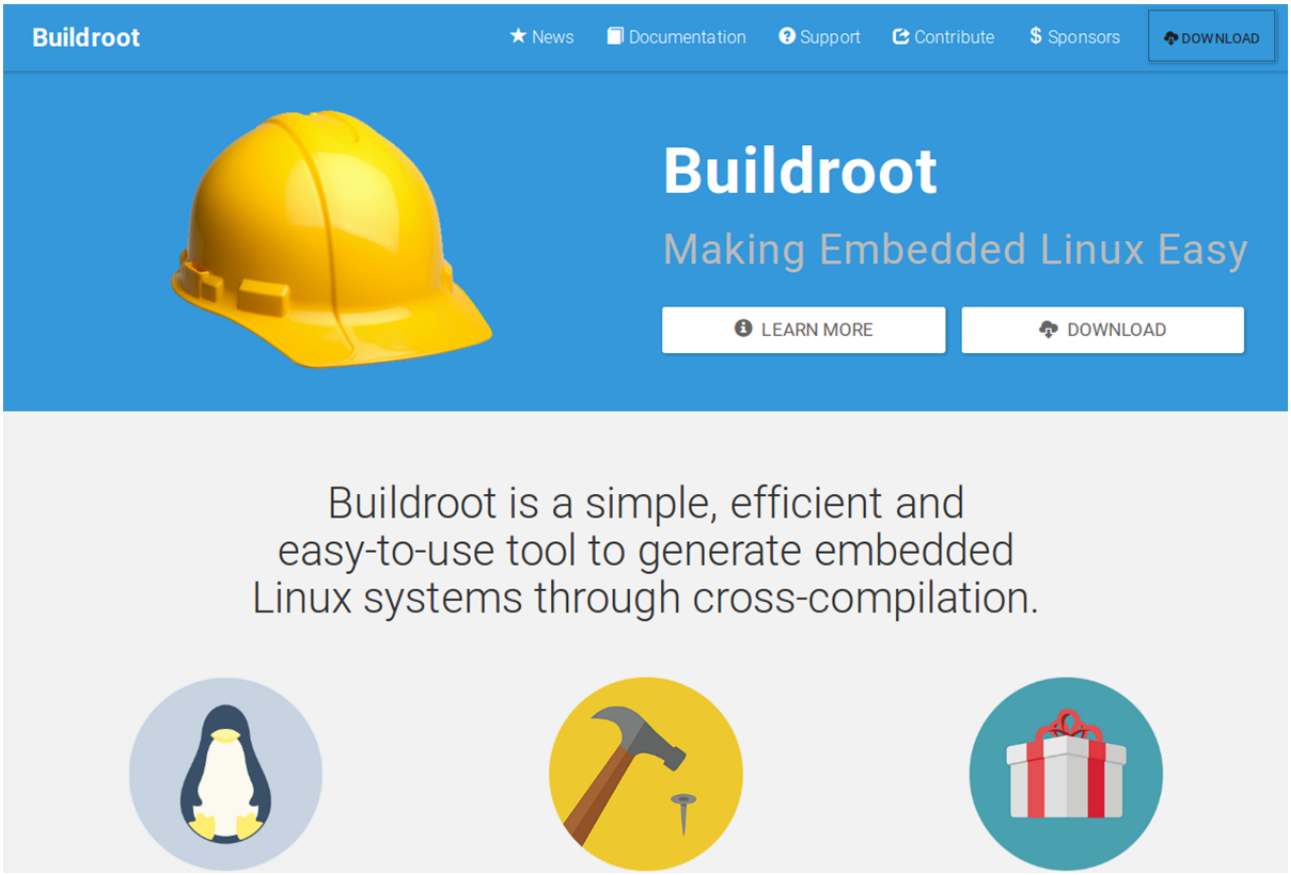


Fig. 51 Buildroot homepage.

Buildroot is a tool to generate embedded Linux systems in our PC, and then this Linux will be installed on the target.

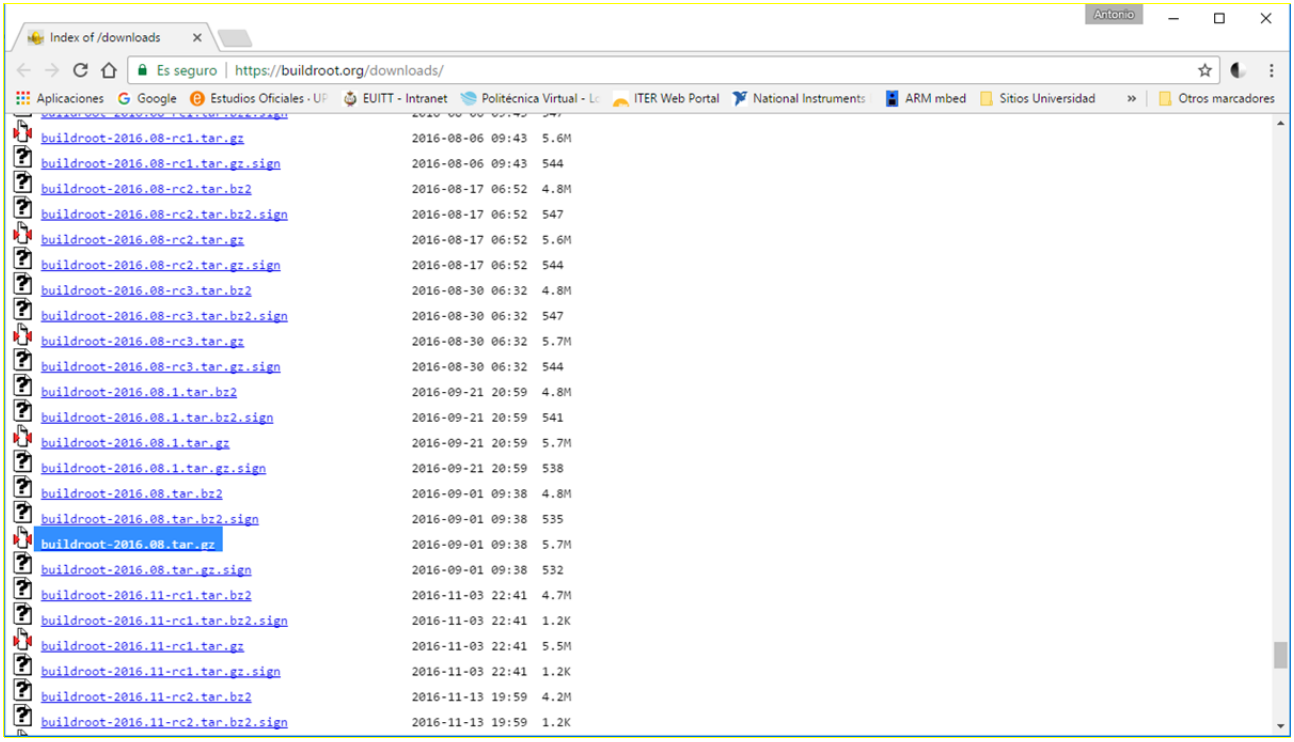


Fig. 52: Downloading Buildroot source code.

Copy the file to the “Documents” folder and decompress the file (Fig. 53).

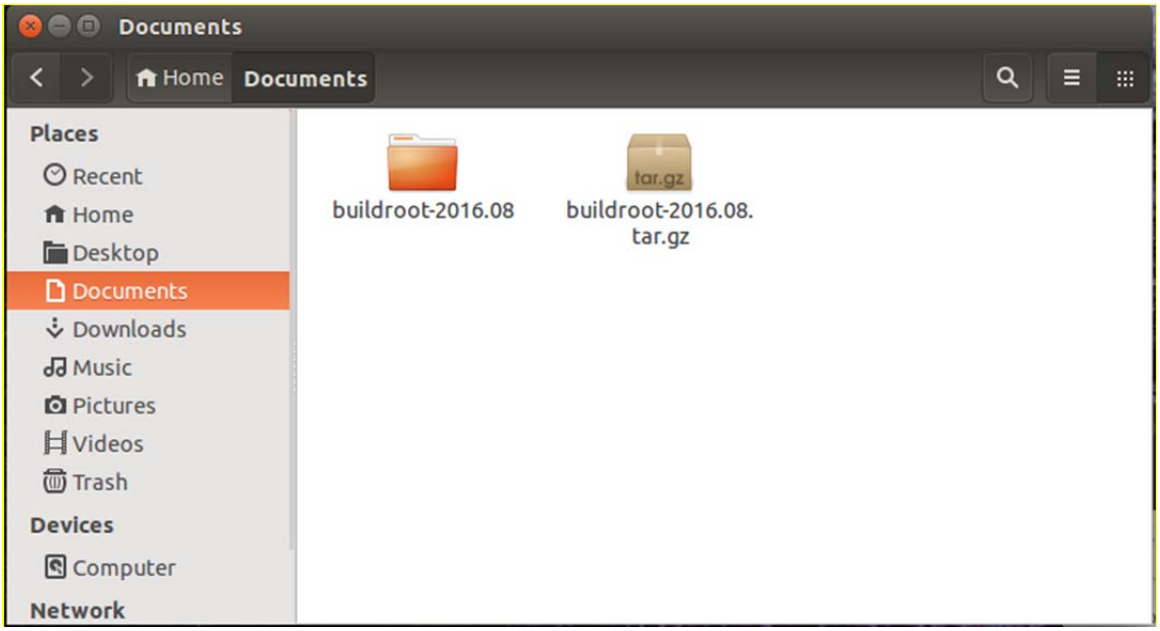


Fig. 53: Buildroot folder (the folder name depends on the version downloaded).

Right-click in the window and execute “Open in Terminal” or execute from Dash home the Terminal application as is shown in the Fig. 54 (if “Open in Terminal” is not available, search how to install it in Ubuntu).

Fig. 54: Dash home, Terminal application

In some seconds a command window is displayed. Then, execute these commands:

```
xxx@ubuntu:~/Documents$ cd buildroot-2016.05
xxx@ubuntu:~/Documents/buildroot-2016.05$ make altera_sockit_defconfig
xxx@ubuntu:~/Documents/buildroot-2016.05$ make xconfig {or make menuconfig}
```

[Help]: make altera_sockit_defconfig generates a basic configuration (not the definitive one) for our system.

[Help]: You can use either “make xconfig” or “make menuconfig”. If “make xconfig” returns an error try “make menuconfig”.

[Help]: In Linux “TAB” key helps you to autocomplete de commands, folders and files names. You can find a description of “make” application in this link https://www.gnu.org/software/make/manual/make.pdf

In some seconds you will see a new window with content similar to Fig. 55 (or Fig. 56).

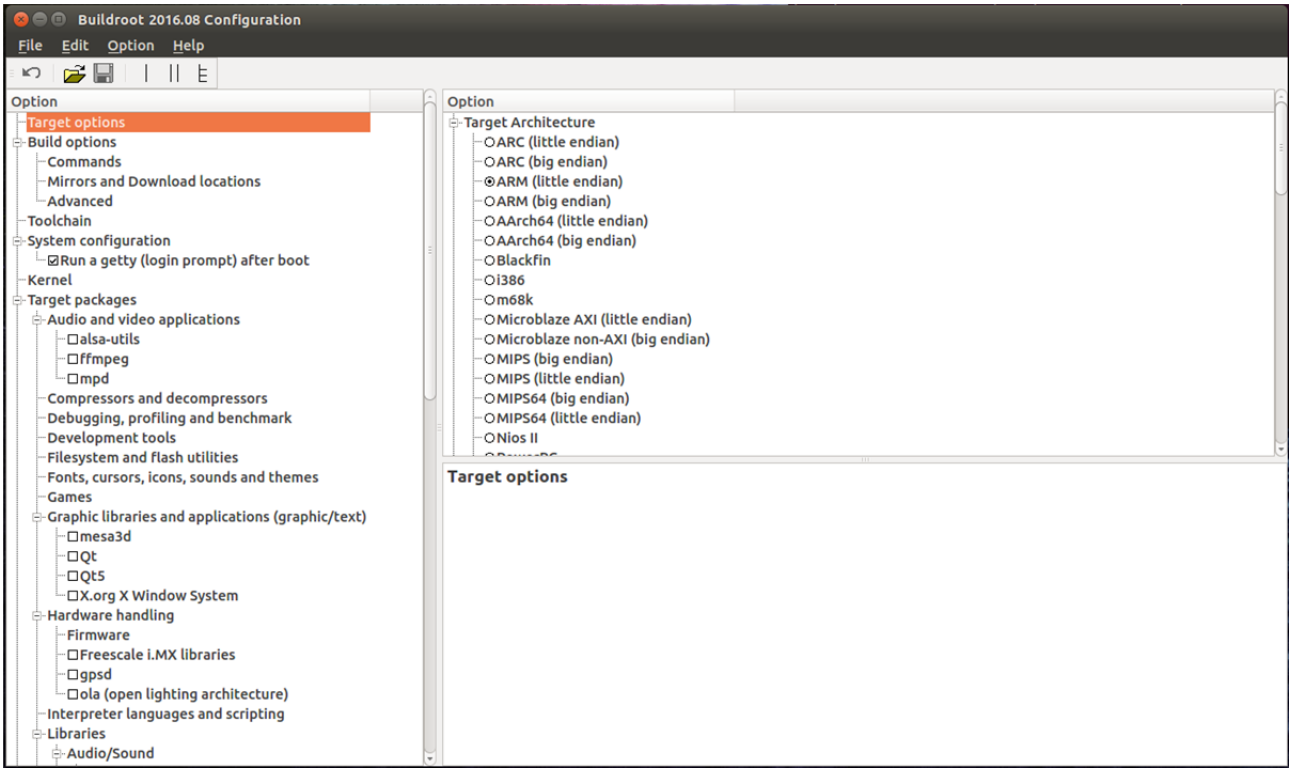


Fig. 55: Buildroot setup screen (make xconfig). The content of this windows depends on the parameter selected.

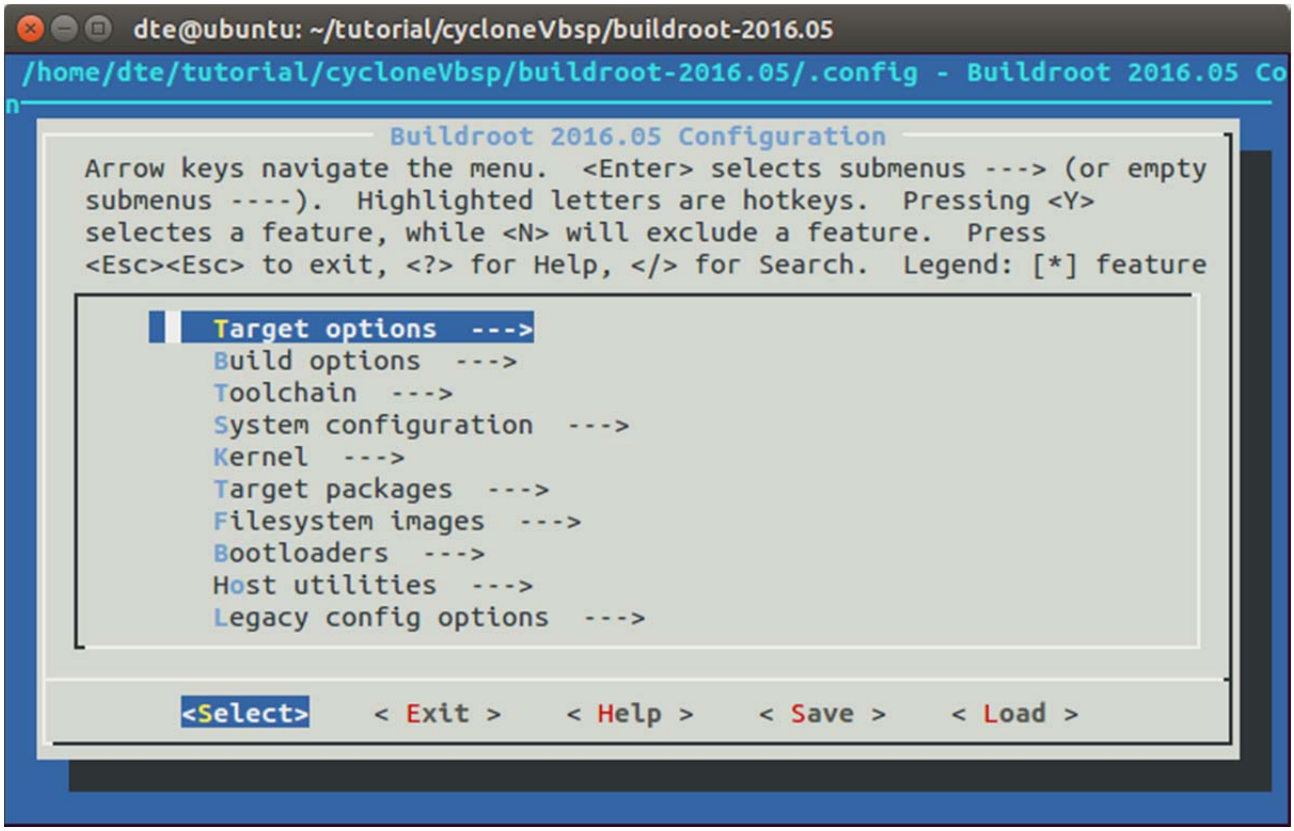


Fig. 56: Buildroot setup screen (make menuconfig).

4.2 Configuring Buildroot.

Once **Buildroot** configuration is started, it is necessary to configure the different items. You need to navigate through the different menus and select the elements to install. Table XI contains the specific configuration of **Buildroot** for installing it in the De1-SoC. Depending on the version downloaded the organization and the items displayed can be different.

[Help]: The Buildroot configuration is an iterative process. To set up your embedded Linux system probably you will need to execute the configuration several times.

Table XI: Parameters for Buildroot configuration

Main Item	Subitem	Value	Comments
Target options	Target Architecture	ARM (little endian)	
	Target Architecture Variant	Cortex-A9	
	Enable NEON SIMD extension support		
	Enable VFP extension support		
	Target ABI	EABIhf	An embedded-application binary interface (EABI) specifies standard conventions for file formats, data types, register usage, stack frame organization, and function parameter passing of an embedded software program.
	Floating point strategy	NEON	
	ARM Instruction Set	ARM	
Build options		Default values	How Buildroot will build the code. Leave default values.
Toolchain			Cross Compiler, linker, libraries to be built to compile our embedded application
	Toolchain Type	Buildroot toolchain	Embedded system will be compiled with tools integrated into Buildroot
	Kernel Headers	Same as kernel being built	Source header files of the Linux Kernel. Navigate to kernel option and select it. The option is now displayed.
	Custom kernel headers series	3.10.x	
	C library	glibc	
	Glibc version	2.23	

Main Item	Subitem	Value	Comments
	Binutils Version	binutils 2.25.1	Binutils contains tools to manage the binary files obtained in the compilation of the different applications
	GCC compiler Version	gcc 4.9.x	GCC tools version to be installed
	Enable C++ support	Yes	Including support for C++ programming, compiling, and linking.
	Build cross gdb for the host	GDB Debugger Version, gdb 7.10.x	It Includes the support for GDB. GCC debugger.
	Enable MMU support	Yes	Mandatory if building a Linux system
System Configuration			
	System Hostname	**de1-soc**	Name of the embedded system
	System Banner	**Free label**	Banner
	Passwords encoding	md5	
	Init System	BusyBox	
	/dev management	Dynamic using devtmpfs only	
	Path to permissions table	**system/device_table.txt**	Text files with permissions for /dev files
	Root FS skeleton	Default target skeleton	Linux folder organization for the embedded system
	Enable root login with password	**Root password: <your value>**	
	/bin/sh	**BusyBox**	
	Remount root filesystem…	yes	
	Network Interface to configure through DHCP	eth0	
	Path to user tables	/home/<user>/ut/test	This file contains the list of user to be created in the embedded Linux system. Use this link to understand how to complete the file. https://buildroot.org/downloads/manual/manual.html#makeuser-syntax.
	Custom Scripts to run …	none	These scripts are executed after Buildroot has finished certain tasks
	Run a getty: Port to	**ttyS0**	Linux device file with the port

Main Item	Subitem	Value	Comments
	run a getty		to run getty (login) process. Uses ttyS0 for serial port
	Baud rate to use	115200	
	TERM environment variable	Vt100	
Linux Kernel			
	Kernel version	Custom Git Repository	
	URL of custom repository	**https://github.com/altera-opensource/linux-socfpga.git**	
	Custom repository version	ACDS16.0_REL_GSRD_PR	
	Custom kernel patches		
	Kernel configuration	Using an in tree defconfig file Defconfig name: socfpga	
	Kernel binary format	**zImage**	
	Kernel compression format	gzip	
	Build Device Tree Blob	Use a custom device tree file	
	Device Tree Source Filenames	<your own path>	You must copy the .dts file generated in previous chapters to this path. If you have generated only the dtb repeat the process to obtain a dts.
	Linux Kernel Extensions	Nothing	
	Linux kernel tools	Nothing	
Target Packages			
	BusyBox configuration file to use	**package/busybox/busybox.config**	
	Audio and video applications	Default values	
	Compresssors and decompressors	Default values	
	Debugging, profiling and benchmark	**Gdb, gdbserver**	
	Developments tools	Default values	
	Filesystem and flash utilities	Default values	
	Games	Default values	
	Graphic libraries and	Default values	

Main Item	Subitem	Value	Comments
	applications (graphic/text)		
	Hardware handling	Default values	
	Interpreters language and scripting	**Perl** **No need Perl libraries/modules**	Perl is used by OpenCL Runtime
	Libraries	Default	
	Mail	Default	
	Miscellaneous	Default	
	Networking applications	**openssh**	
	Package managers Real Time Shell and utilities System Tools Text Editors and viewers	Default	
	Real-Time	Default	
	Security	Default	
	Shell and utilities	Default	
	System tools	Default	
	Text editors and viewers	Default	
Filesystem Images			
	ext2/3/4 root filesystem	ext3 Exact size in blocks: 204000 Compression method **no compression tar**	Very important to have space in the filesystem to copy applications, etc.
Bootloaders			
	U-Boot	Build system: Legacy U-Boot board name: **socfpga_cyclone5** U-Boot Version: Custom Git repository URL of custom repository: http://github.com/altera-opensource/u-boot-socfpga Custom repository version: ACDS16.0_REL_GSRD_PR U-Boot binary format: u-boot.img The patch should not be activated!!!!	You need to provide, in the field "U-boot SPL binary image name", the name of the file to be generated. This SPL is the preloader but we are not going to use it. Instead we use the preloader generated by Altera SoC tools.
Host utilities		Nothing	
Legacy config options		Default values	

Once you have configured all the menus, you need to exit saving the values (File->Quit).

[Help]: The **Buildroot** configuration is stored in a file named as ".config". You should have a backup of this file. If you want to build another Buildroot in another directory with a different configuration but reusing some of the packages downloaded, you can copy the "dl" folder to a new installation of Buildroot and avoid the time needed to download the source packages. The slowest download is always the kernel source code because it downloads approximately 1Million of files.

4.3 Important conclusions about the configuration of Buildroot.

The configuration of an embedded system to be used with Linux requires the use of stable u-boot versions and Linux kernel versions. The step completed previously has used the stable configurations provided by Altera in the Golden Design References. Check Altera Git on GitHub with the new updates introduced by this manufacturer.

4.4 Compiling Buildroot.

In the Terminal Window executes the following command:

```
xxx@ubuntu:~/Documents/buildroot-2016.08$ make
```

If everything is correct, you will see a final window similar to the represented in Fig. 57.

[Time for this step]: In this step Buildroot is going to connect, using the internet, to different repositories. After downloading the code, Buildroot is going to compile the applications and generates a lot of files and folders. Depending on your internet speed access and the configuration was chosen, this step could take up to **two hourss and a half**.

Warning. If you have errors in the configuration of Buildroot, you could obtain errors in this compilation phase. Check your configuration correctly. Use "make clean" to clean up your partial compilation.

Warning. dl subfolder in your Buildroot folder contains all the packages download from the internet. If you want to move your Buildroot configuration from one computer to another avoiding the copy of the virtual machine you can copy this folder.

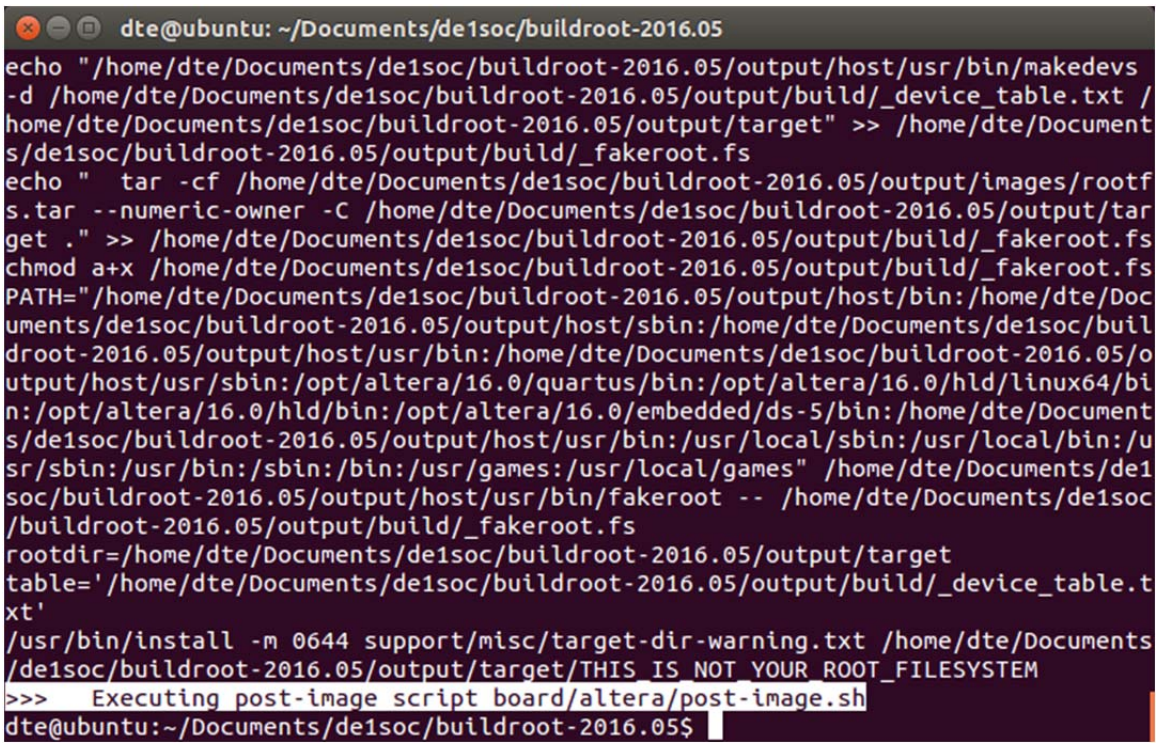


Fig. 57: Successful compilation and installation of Buildroot

Buildroot has generated some folders with different files and subfolders containing the tools for generating your Embedded Linux System.

4.5 Buildroot Output.

The main output files of the execution of the previous steps can be located at the folder "./output/images". Fig. 58 summarizes the use of **Buildroot**. Buildroot generates a boot loader, a kernel image, and a file system.

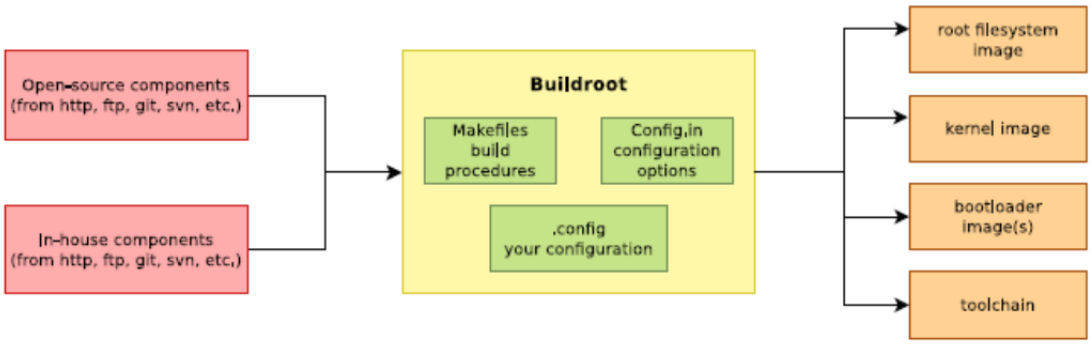


Fig. 58: Schematic representation of the Buildroot tool. Buildroot generates the root file system, the kernel image, the bootloader and the toolchain. Figure copied from "Free Electrons" training materials (http://free-electrons.com/training/)

In our specific case, the folder content is shown in Fig. 59

Fig. 59: images folder contains the binary files for our embedded system.

The u-boot.img file contains the u-boot binary for booting your Linux. Linux image is zImage. The files with dtb extension are the device tree in the binary form used by Linux in the booting phase to discover the hardware available in the DE1-SoC. The file rootfs.ext2 contains all the files of the file system of your embedded Linux. To boot the DE1-SoC from SD card firstly, you need to format it adequately and copy the files manually or using the script aforementioned.

4.6 Re-create the SDcard.

Now you can boot the DE1-SoC using your files. Repeat the steps described in 3.8.4.

4.7 Basic test your embedded Linux System

Your embedded Linux system is running in the DE1SoC, execute the following commands and analyse the output information.

```
$dmesg
….
$uname –r
….
$ifconfig
….
$ps –ax
```

4.7.1 Exercise 1:

Answer the following questions:

a) Which are the kernel parameters used to boot the system? Explain the meaning of them.
b) What IP address has been assigned to De1SOC? Who is providing this IP?
c) How many processes are running on the Linux OS?
d) Which devices have been detected in the system boot?
e) Try to access to de1-soc using putty with an ssh connection. First, try to access with root account and later with another account. What happens? Find a possible solution to this.

5 USING INTEGRATED DEVELOPMENT ENVIRONMENT: ECLIPSE/CDT

5.1 Adding cross-compiling tools to PATH variable.

Using a text editor, edit the **.profile** file (available in your home directory). Add a line at the end of the file containing: PATH="<your Buildroot installation>/output/host/usr/bin:$PATH". This adds to the PATH environment variable the location of the cross-compiling tools. You must log out and login again.

5.2 Cross-Compiling application using Eclipse.

How will a program be compiled? Remember that we are developing cross applications. We are developing and compiling the code on a Linux x86 machine, and we are executing it in an ARM architecture (see Fig. 60).

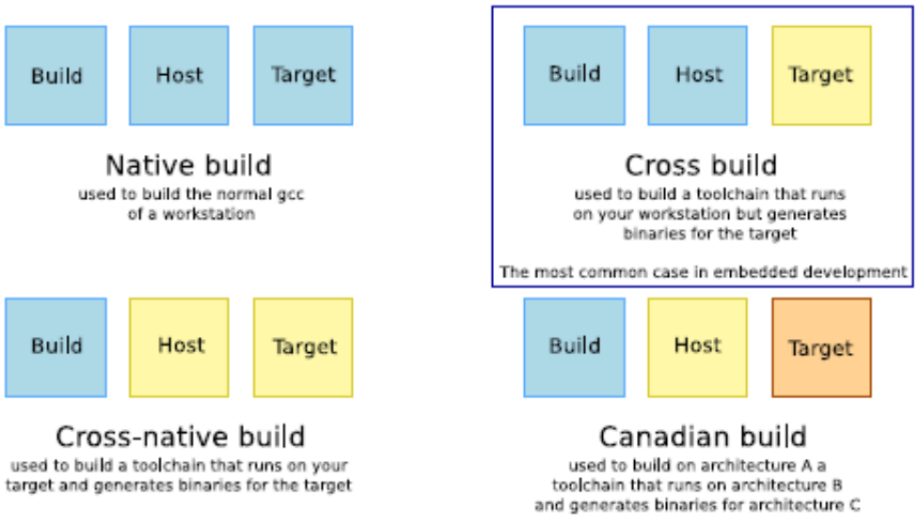


Fig. 60: Summary of the different configurations for developing applications for embedded systems. Figure copied from "Free Electrons" training materials (http://free-electrons.com/training/)

The first question is where the cross-compiler is located. The answer is this: in the folder "<buildroot>/output/host/usr/bin". If you inspect the content of this folder, you can see the entire compiling, linking and debugging tool (see Fig. 61). These programs are executed on your x86 computer, but they generate code for ARM processor.

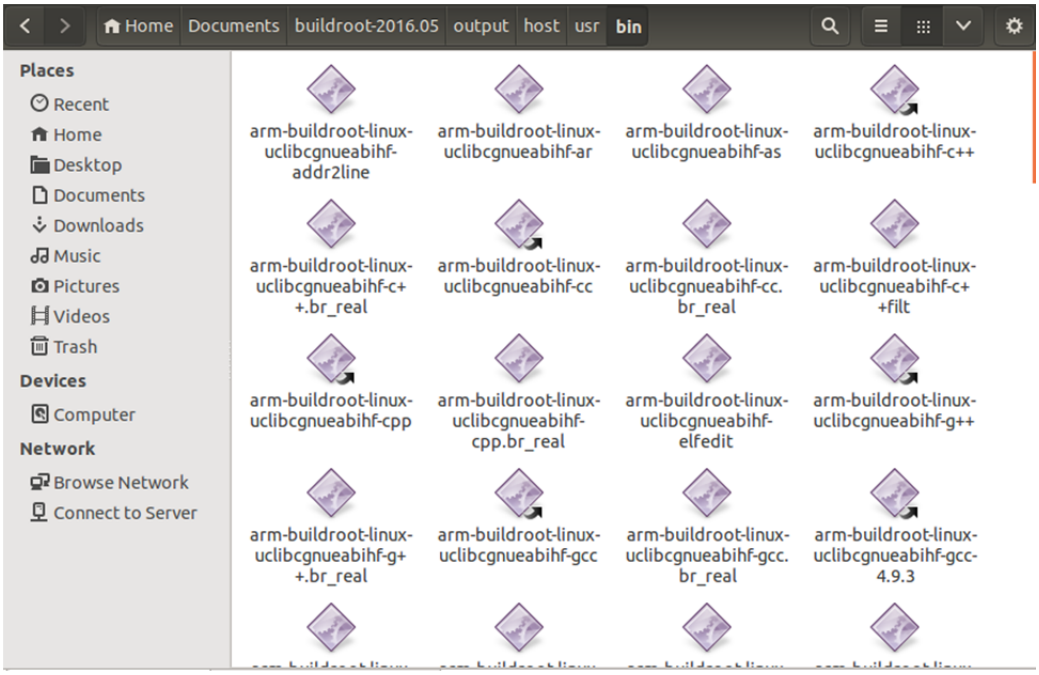


Fig. 61: Cross compiling tools installed in the host computer

In a Terminal window start Eclipse with the following command:

```
xxxx@ubuntu:~$ eclipse
```

The popup window invites you to enter the workspace (see Fig. 62). The workspace is the folder that will contain all the Eclipse projects created by the user. You can have as many workspaces as you want. Please specify a folder in your account.

[Help]: The figures displayed in the following paragraphs can be different depending on the Eclipse version installed.

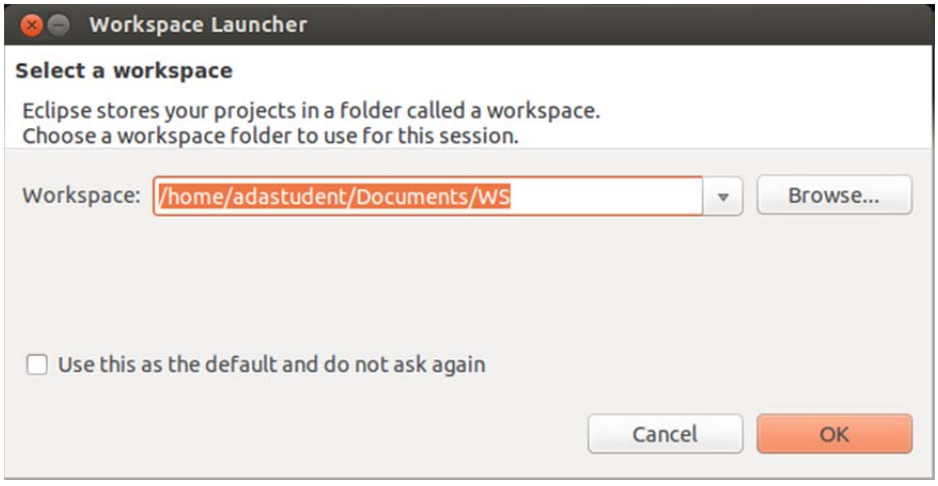


Fig. 62: Selection of the workspace for Eclipse. Use a folder in your account.

Select Ok, and the welcome window of Eclipse will be shown (Fig. 63). Next, close the welcome window, and the main Eclipse window will be displayed (Fig. 64).

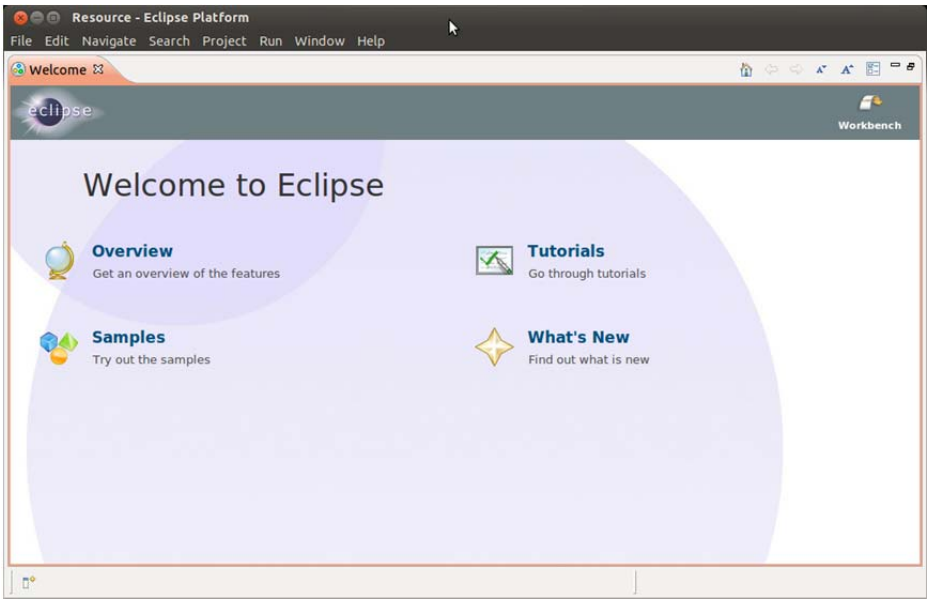


Fig. 63: Eclipse welcome window.

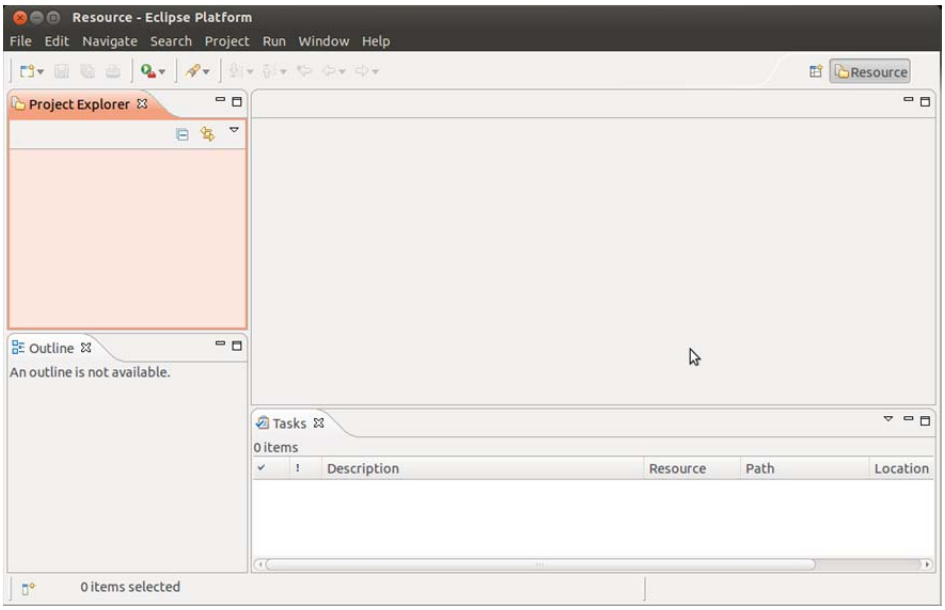


Fig. 64: Eclipse main window.

Create an Eclipse C/C++ project (File->New->Project->C/C++project->C project) selecting the hello world example (see Fig. 65). Specify the project name and the toolchain to be used. In this case a Cross GCC. Press Next.

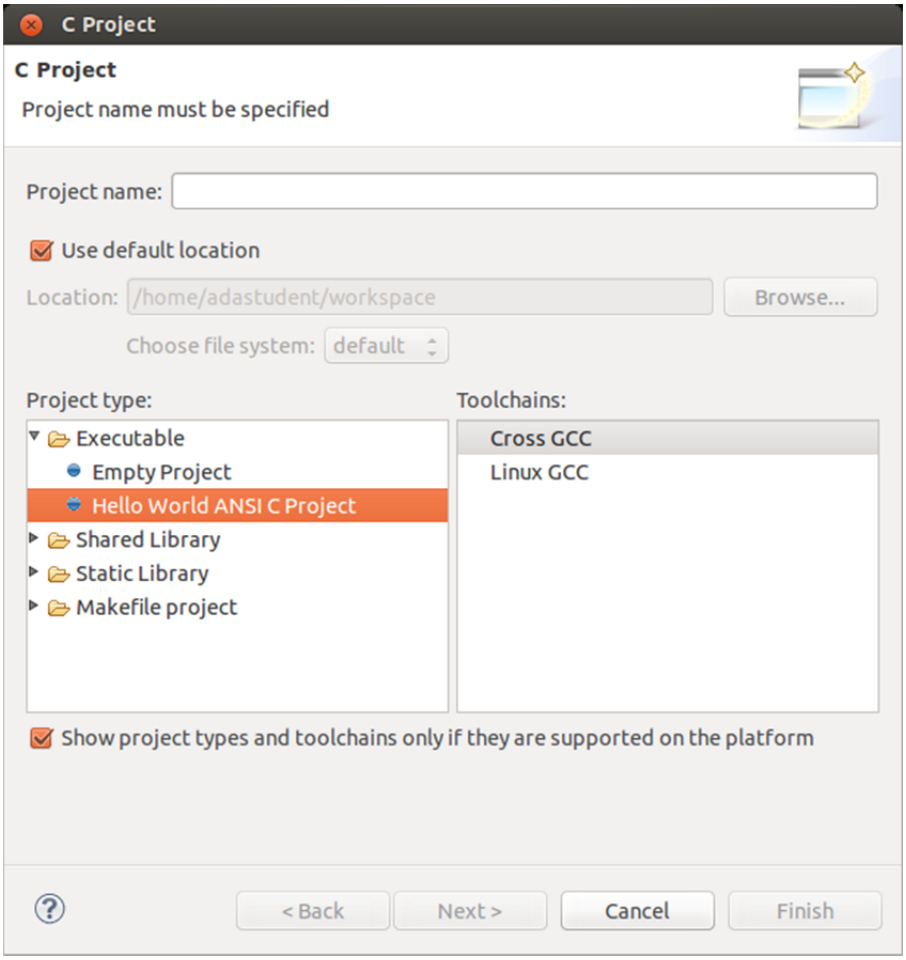


Fig. 65: Basic C project creation in Eclipse

There is a window (Fig. 66) requesting the Cross Compiler prefix and path, leave both inputs blank and click on the Finish button. You will obtain your first project created with eclipse.

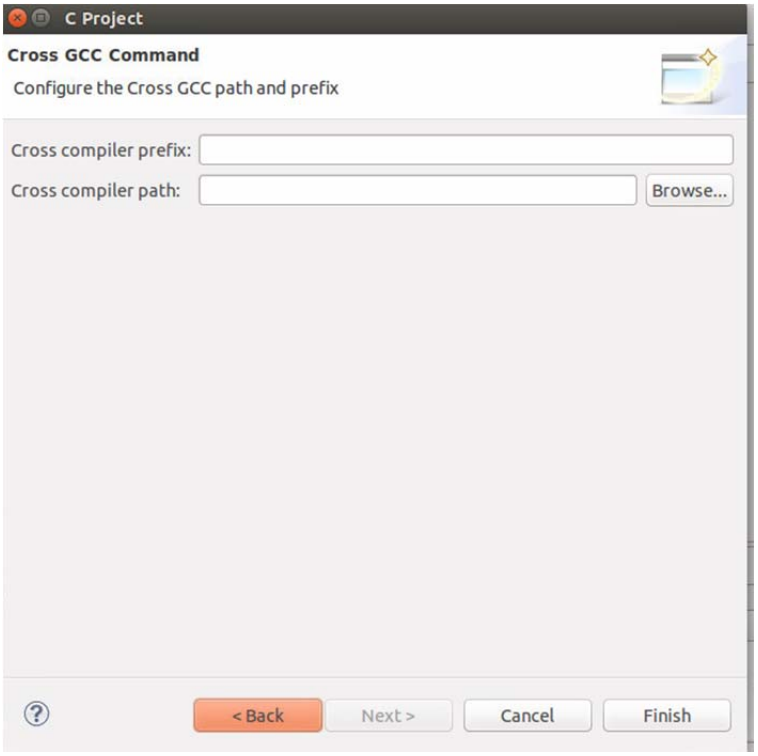


Fig. 66: Cross-compiler prefix and path window.

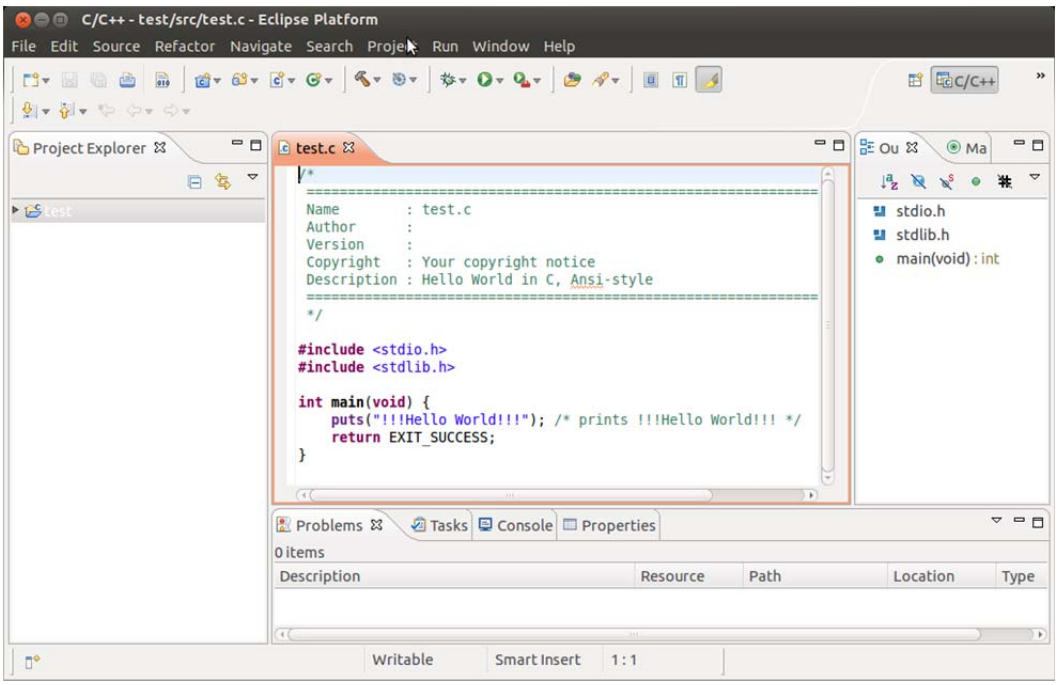


Fig. 67: Hello world example.

The next step (mandatory) is the Eclipse project configuration for managing the Cross-tools. In Project -> Properties configure the C/C++ Build Setting as the Fig. 68 and Fig. 69 shown. Pay attention that Prefix requires a string ending in a hyphen.

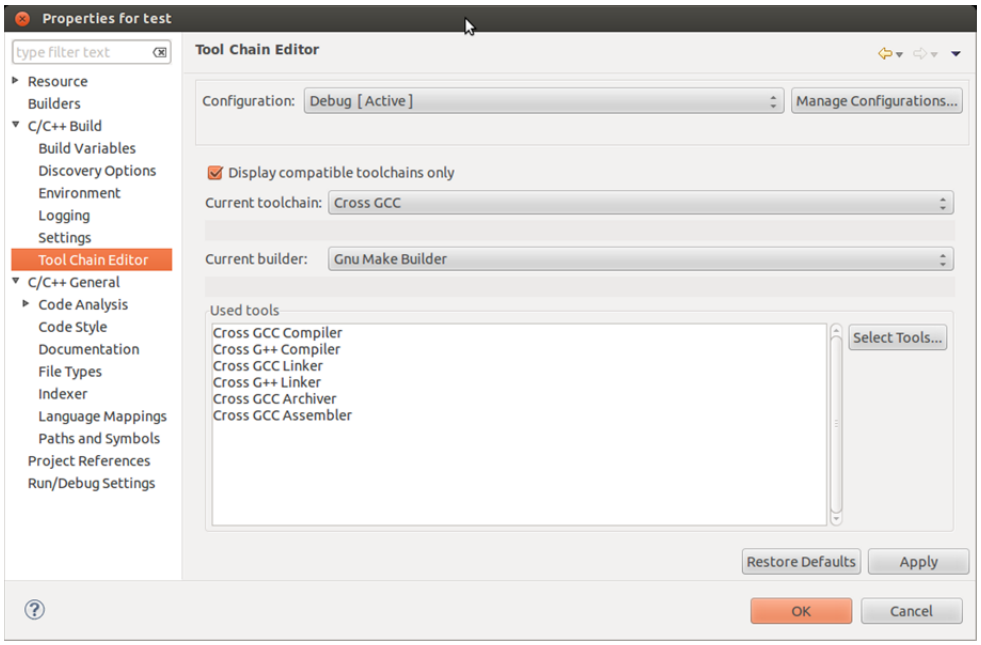


Fig. 68: Tool Chain Editor should be configured to use Cross GCC.

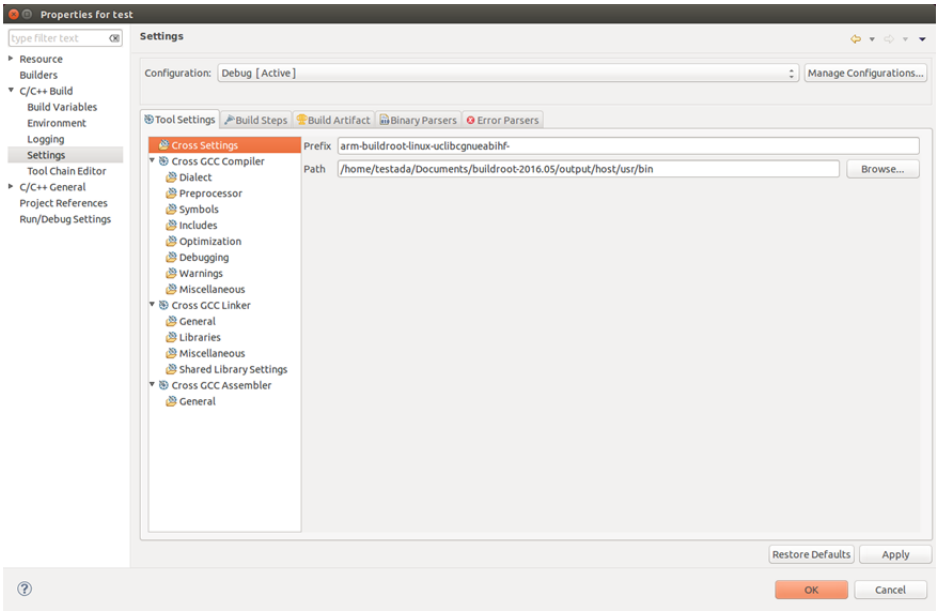


Fig. 69: Cross tools locate on (path).The path shown in this figure is an example.Use always the path of your toolchain.

The next step is to configure the search paths for the compiler and linker and the different tools to use. Complete the different fields with the information included in Fig. 70 and Fig. 71. Please consider the paths. The figures are showing examples for a specific user account.

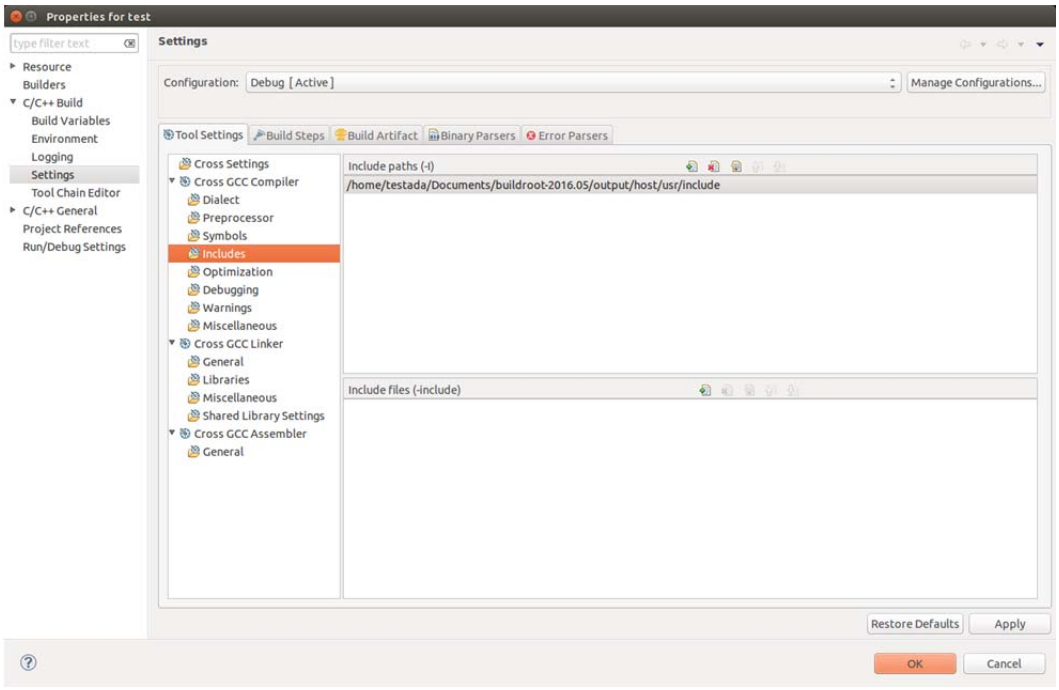


Fig. 70: Include search path.

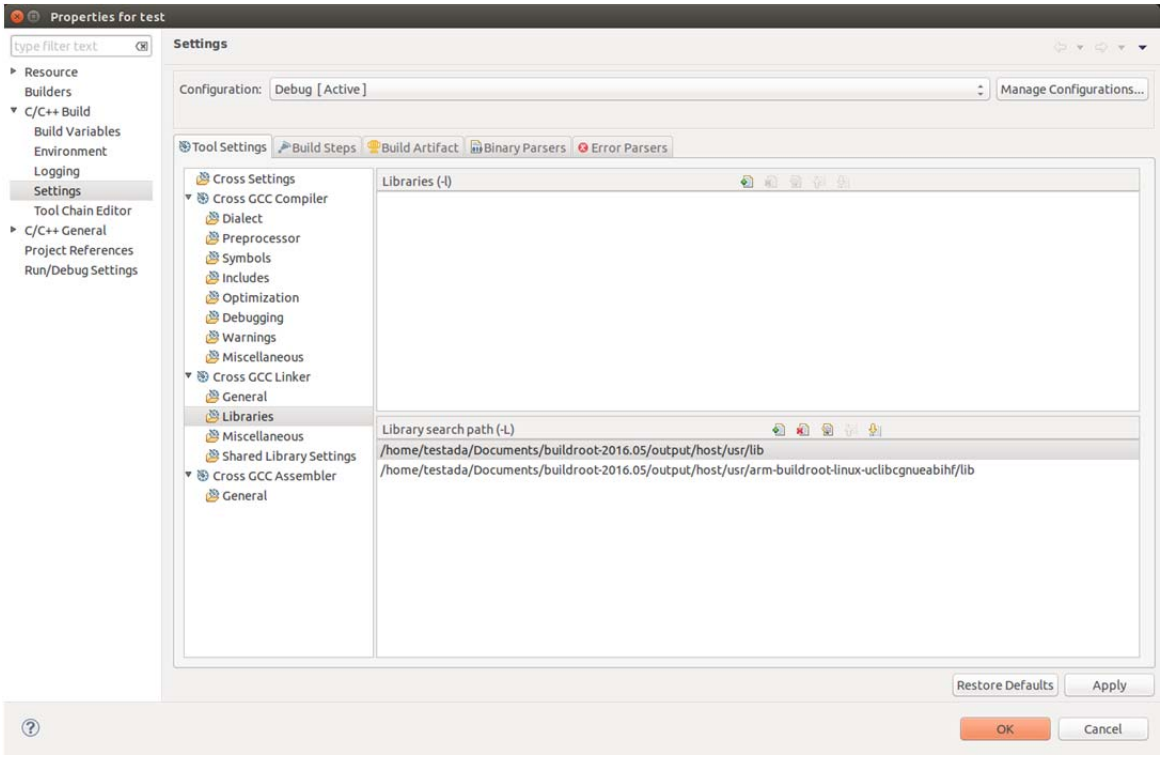


Fig. 71: Libraries search path.

Once you have configured the cross chain in Eclipse, you can build your project using Project->Build Project. If everything is correct, you will see the eclipse project as represented in Fig. 73.

[Console in Eclipse]: Have a look at the messages displayed in the Console. You will see how eclipse is calling the cross compiler with different parameters.

To copy the executable to the target, you have different options. You can use the Linux application called "scp" or other similar applications. In our case, we are going to use "Connect to Server…." utility included in Ubuntu (under Places menu). Specify in Server Address ssh://<ip address>

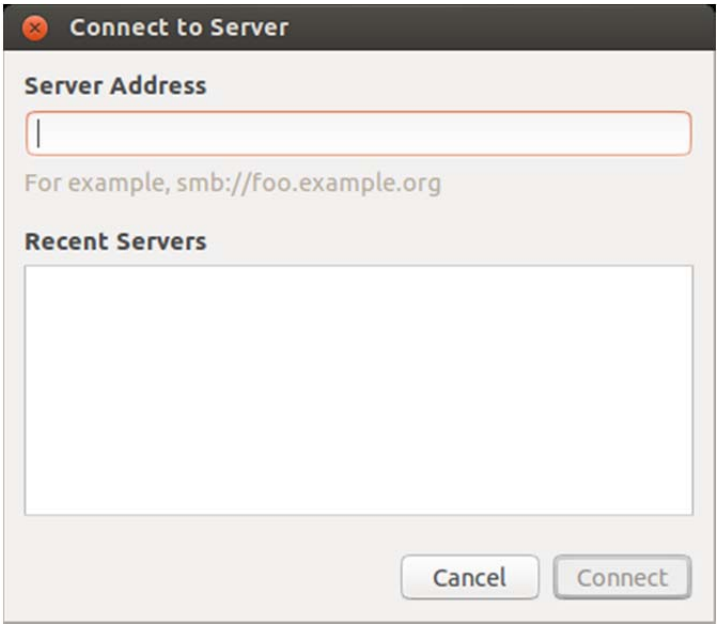


Fig. 72: Pop-up window when executing "Connect to Server."

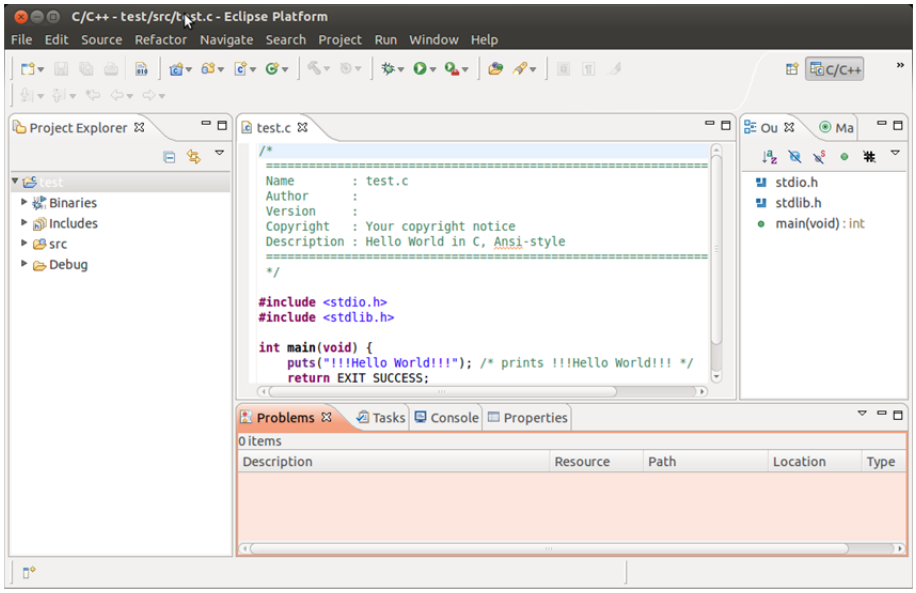


Fig. 73: Eclipse project compiled (Binaries has been generated).

You can run the program on the target using putty (remember that once you have a network connection available in the DE1-SoC, you can also use putty to connect to it).

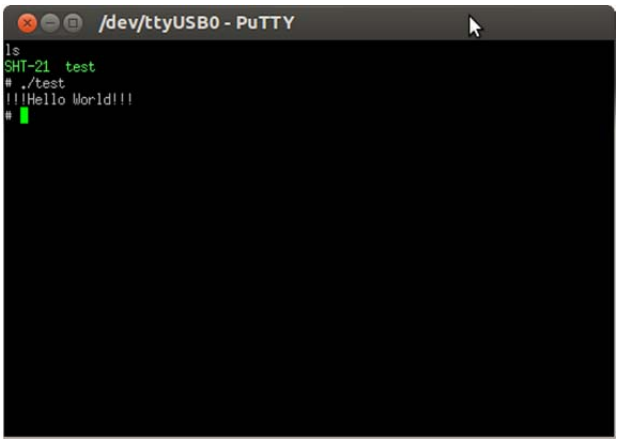


Fig. 74: Run test program in Raspberry PI

Warning. If you experiment problems using ssh, delete the .ssh folder in your home directory.

5.3 Automatic debugging using gdb and gdbserver.

You can directly debug the program running in the DE1-SoC using Eclipse. There are two methods to do it: manually and automatically. In the manual method, firstly, you need to copy the executable program to the DE1-SoC, change the file permissions to "executable" and execute the program to be debugged using gdbserver utility. Of course, this is a time-consuming process and very inefficient. The alternative solution is to use the automatic debugging. To debug your applications, we need to define a debug session and configure it. Firstly, Select Run->Debug Configurations and generate a new configuration under C/C++ Remote Application. You need to complete the different tabs available in this window. The first one is the main tab (see Fig. 75). You need to configure here: the path to the C/C++ application to be debugged, the project name; the connection with the target (you will need to create a new one using the IP address of your BBB); the remote path where your executable file will be downloaded; and the mode for the debugging (Automatic Remote Debugging Launcher). Secondly, in the argument tab, you can specify the argument of your executable program. Very important here is that you can also specify the path to the working directory where the executable will be launched.

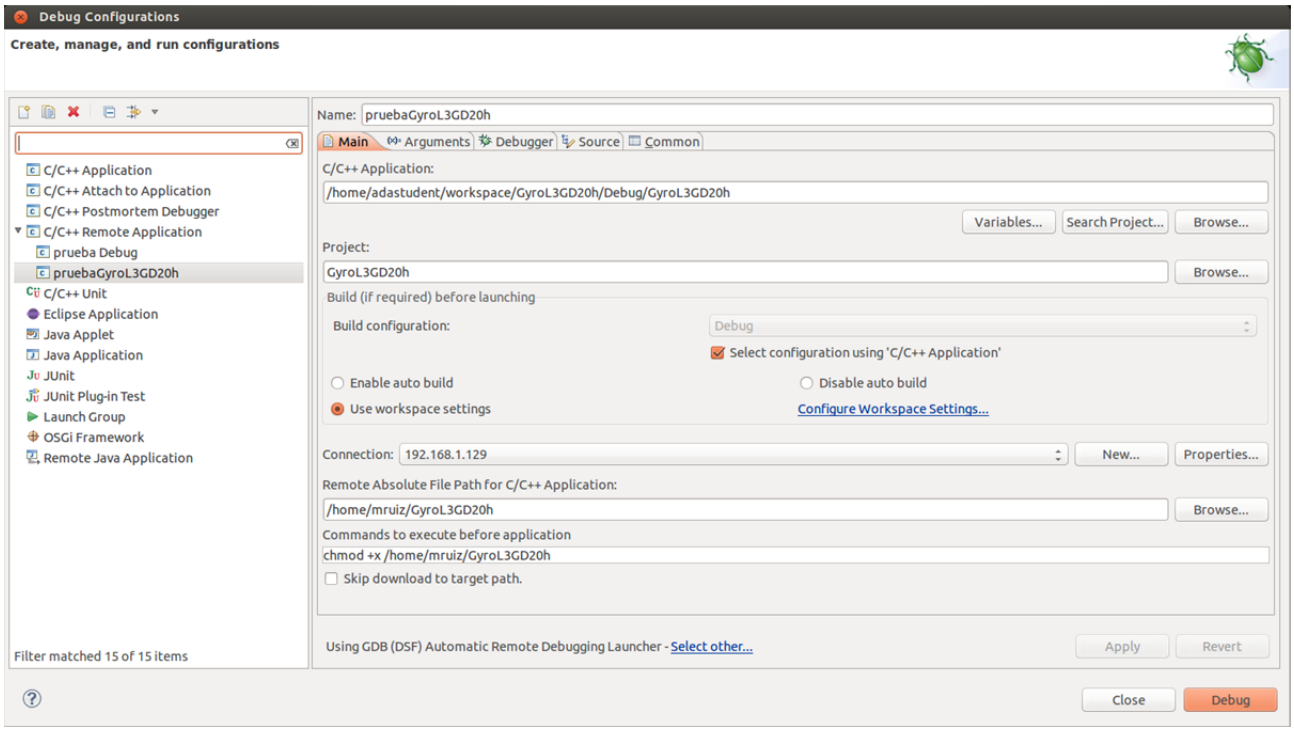


Fig. 75: Creating a Debug Configuration

In the debugger window (main tab) you need to configure the path of your gdb application. Remember that we are working with a cross-compiler, cross-debugging, therefore, you need to provide here the correct path of your gdb. The GDB command file must be specified, providing a path to an empty file. In the Gdbserver settings tab, you need to provide a path to the gdbserver in the target and the port used (by default 2345).

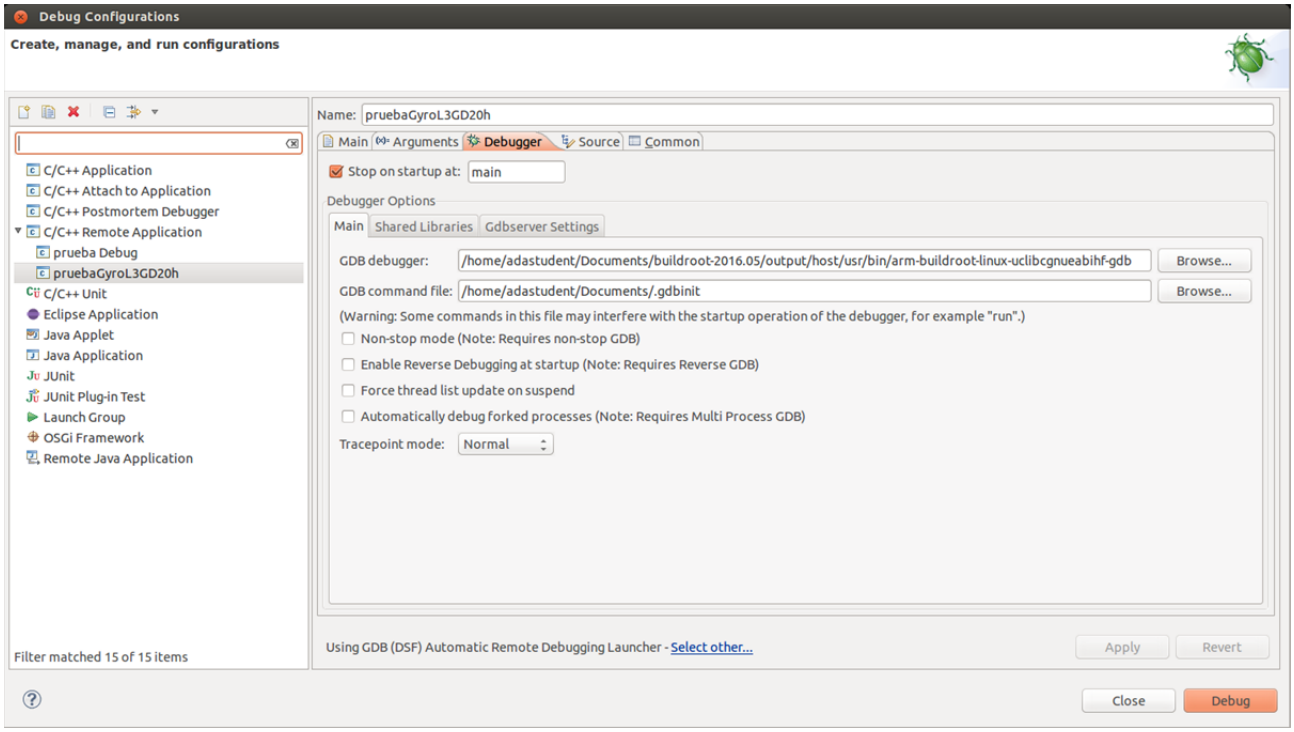


Fig. 76: Debug configuration including the path locating the cross gdb tool.

Now, press Debug in Eclipse window, and you can debug your application remotely.

6 DESIGNING CUSTOM PERIPHERALS IN THE FPGA.

It is common in many development system kits to try out firstly with some basic examples using the basic input-output elements of the boards, such as displays, LEDs, switches and/or buttons. Making some funny application consisting of blinking the LEDs or showing info in the displays and controlling the behavior through the switches or buttons. These applications are easy to create and, at the same time, gets you familiar with using the design tools, the procedures and the different techniques involved in embedded systems design. Later, there will be an opportunity to increase the design complexity to make more powerful and interesting applications.

6.1 Creating an external peripheral connected to HPS parallel I/O: Displays Test.

This first peripheral is aimed to test the correct functioning of the six displays mounted in the DE1SoC board. The solution is an external hardware element, placed at the top level of the design hierarchy, and connected to the HPS trough a standard parallel IO port, by which the user can control its operation.

6.1.1 VHDL Hardware Model.

The code below is the VHDL model of the hardware to be connected to the HPS PIO. You can find this code in the file “displays_test.vhd” [4].

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity displays_test is
port(  clk:         in     std_logic;
       nRst:        in     std_logic;
       disp_ena_n:  in     std_logic_vector(5 downto 0);
       disp0:       buffer std_logic_vector(6 downto 0);
       disp1:       buffer std_logic_vector(6 downto 0);
       disp2:       buffer std_logic_vector(6 downto 0);
       disp3:       buffer std_logic_vector(6 downto 0);
       disp4:       buffer std_logic_vector(6 downto 0);
       disp5:       buffer std_logic_vector(6 downto 0));
end entity;

architecture rtl of displays_test is
       signal counter:     std_logic_vector(22 downto 0);
       signal segments:    std_logic_vector(6 downto 0);
       signal num_disp:    std_logic_vector(2 downto 0);
       signal tic:         std_logic;
       constant tenth_sec: natural := 4999999;

begin
       process(clk, nRst)
       begin
          if nRst = '0' then
              counter <= (others => '0');

          elsif clk'event and clk = '1' then
              if tic = '1' then
                  counter <= (others => '0');
              else
                  counter <= counter + 1;
              end if;
          end if;
       end process;

       tic <= '1' when counter = tenth_sec else '0';
```

```
        process(clk, nrst)
        begin
            if nRst = '0' then
                 segments <= "1111110";
            elsif clk'event and clk = '1' then
                 if tic = '1' then
                      if segments = 0 then
                          segments <= "1111110";
                      else
                          segments <= segments(5 downto 0)&'0';
                      end if;
                 end if;
            end if;
        end process;

        process(clk, nrst)
        begin
            if nRst = '0' then
                      num_disp <= (others => '0');

            elsif clk'event and clk = '1' then
               if segments = 0 and tic = '1' then
                   if num_disp = 5 then
                      num_disp <= (others => '0');
                   else
                      num_disp <= num_disp + 1;
                   end if;
               end if;
            end if;
        end process;

        disp0 <= segments          when disp_ena_n(0) = '0' and num_disp = 0 else
                 (others => '0')   when disp_ena_n(0) = '0' and num_disp > 0 else
                 (others => '1');

        disp1 <= segments          when disp_ena_n(1) = '0' and num_disp = 1 else
                 (others => '0')   when disp_ena_n(1) = '0' and num_disp > 1 else
                 (others => '1');

        disp2 <= segments          when disp_ena_n(2) = '0' and num_disp = 2 else
                 (others => '0')   when disp_ena_n(2) = '0' and num_disp > 2 else
                 (others => '1');

        disp3 <= segments          when disp_ena_n(3) = '0' and num_disp = 3 else
                 (others => '0')   when disp_ena_n(3) = '0' and num_disp > 3 else
                 (others => '1');

        disp4 <= segments          when disp_ena_n(4) = '0' and num_disp = 4 else
                 (others => '0')   when disp_ena_n(4) = '0' and num_disp > 4 else
                 (others => '1');

        disp5 <= segments          when disp_ena_n(5) = '0' and num_disp = 5 else
                 (others => '1');


    end architecture;
```

Fig. 77: VHDL source code for the hardware controlling the displays.

6.1.2 Exercise 2: analyse the code and answer these questions:

a) Describe the waveform of the signal “tic”
b) What kind of subsystem is modelled by the process named “segment_controller”?
c) What kind of subsystem is modelled by the process named “displays_controller”?
d) What is the goal of the combinational logic under “enable control”?
e) Describe the global operation of the “displays_test” peripheral

6.1.3 Adding the “displays_ctrl” parallel IO port to the system.

We are going to use a standard 6 bits PIO component (available in the IP catalog in QSYS) to drive the “*disp_ena_n*” bus. To achieve this, it is necessary, firstly to create this element with *QSYS* and, secondly, to connect PIO outputs with the displays enable inputs. This connection has to be done in the top VHDL design file. The former step is to add this PIO component:

1. Open Quartus project “`soc_system.qpf`” and launch Qsys, then open “`soc_system.qsys`” as described previously. Following the same method as when the hps, clk, sysid and jtag_uart were instantiated, it is time to place a new component in our basic system. Now the PIO (Parallel IO) component is going to be instantiated and connected in our system
2. Type `PIO` in the Search Window of IP catalog.
3. Double click on the `PIO (Parallel I/O)` to add to your system. This is an Altera parallel input/output peripheral with Avalon bus.

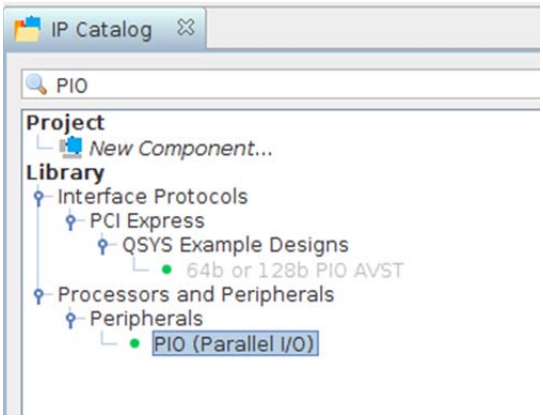


Fig. 78: Selecting the PIO IP in the catalog.

As shown in the Fig. 79, in the properties window, edit the following information:

4. Set the `Width` to `6`, which is the number of enable signals for the Displays on the board connected to the FPGA I/O pins.
5. Ensure that the direction is set to `Output`.
6. Select `Finish.`
7. Change the default name of the PIO component to be `displays_ctrl`. To change the name, select the component (click to highlight), right click, and select `Rename`.

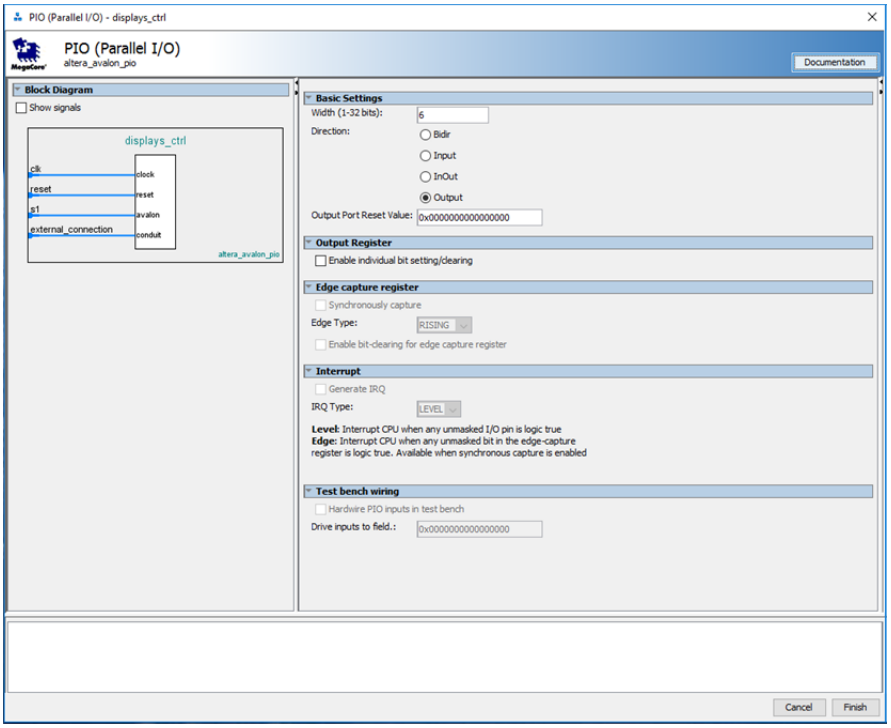


Fig. 79: Parameters in the PIO IP.

The output port lines will drive the `disp_ena_n` inputs of the `displays_test` hardware module. Therefore, they must be connected to each other in the top design file, and the port signals will need to be exported to the top level of the project. To do this:

8. Double click in the export column associated with the external connection of the `displays_ctrl` and the following should automatically show up: `displays_ctrl_external_connection`. If not, then type it.
9. Connect the inputs of the `displays_ctrl` component to the `hps_0` as shown in the Fig. 80.
10. Double click the column "*Base*" in the line *s1 (Avalon Memory-Mapped Slave)* and write the offset address `0x0003_0000`. Considering that the base address of the *h2f_lw_axi_master* is "0xff20_0000", as can be seen in the *cyclone5_handbook. Volume 3. Page 1-16. Table1-2(common address space regions)*, the base address of our parallel port is "0xff23_0000". The size of Altera PIO IP is fixed to "0xf", as can also be seen in the Fig. 80.
11. Finally, generate the HDL pressing the button.

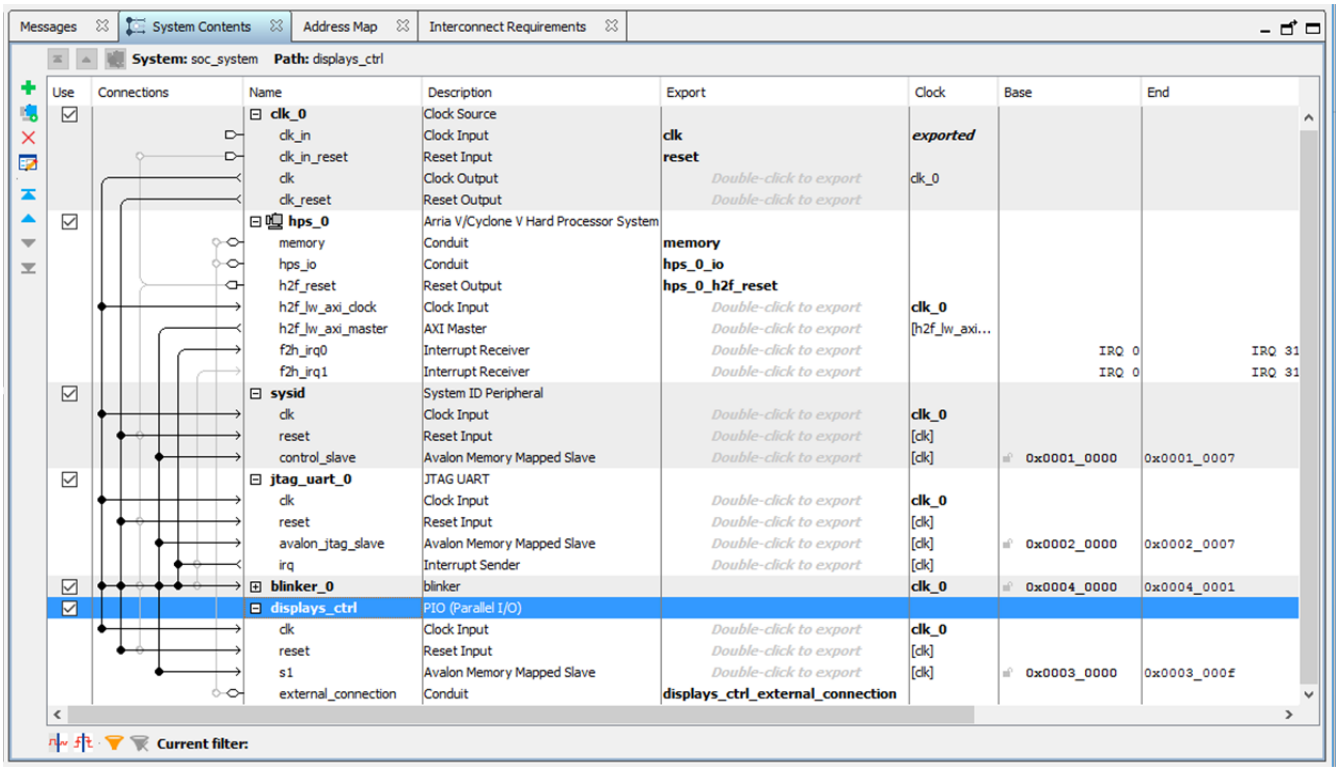


Fig. 80: Details of the connection of displays_ctrl IP (Avalon PIO) with the hps_0 (HPS).

6.1.4 Implementing the connection of the display control hardware module with the soc_system.

Once the *displays_ctrl* Parallel IO port has been included in the *soc_system,* and the *displays_test* module has been designed, the top level of the project (described in VHDL) must be modified.

12. Make a new directory, called "`ip`" in the Quartus project root folder, and inside another directory with the name "`displays_test`". Copy the "displays_test.vhd" file into this new folder.
13. Include the new design file "displays_test.vhd" in the project, as shown in Fig. 81, get by menu option "`Assignments->settings`", category "`files`". Select the file in the navigation bar and push "add" button.

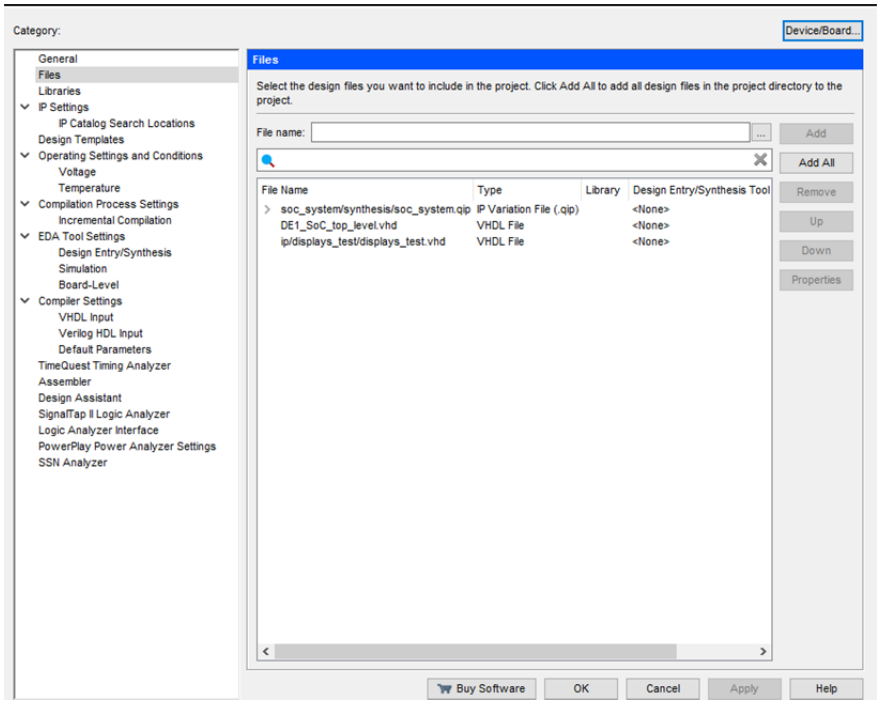


Fig. 81: Adding displays_test.vhd file to project.

Open the "*soc_system_top.vhd*" file and make the following changes:

14. In the entity declaration uncomment the items related to the LELDs, switches, buttons and displays (see Fig. 82).

```
58
59      -- -- SEG7
60         HEX0_N          : out   std_logic_vector(6 downto 0);
61         HEX1_N          : out   std_logic_vector(6 downto 0);
62         HEX2_N          : out   std_logic_vector(6 downto 0);
63         HEX3_N          : out   std_logic_vector(6 downto 0);
64         HEX4_N          : out   std_logic_vector(6 downto 0);
65         HEX5_N          : out   std_logic_vector(6 downto 0);
66
67      -- -- IR
68      -- IRDA_RXD        : in    std_logic;
69      -- IRDA_TXD        : out   std_logic;
70
71      -- -- KEY_N
72         KEY_N           : in    std_logic_vector(3 downto 0);
73
74      -- -- LED
75         LEDR            : out   std_logic_vector(9 downto 0);
76
77      -- -- PS2
78      -- PS2_CLK         : inout std_logic;
79      -- PS2_CLK2        : inout std_logic;
80      -- PS2_DAT         : inout std_logic;
81      -- PS2_DAT2        : inout std_logic;
82
83      -- -- SW
84         SW              : in    std_logic_vector(9 downto 0);
85
86      -- -- Video-In
87      -- TD_CLK27        : inout std_logic;
88      -- TD_DATA         : out   std_logic_vector(7 downto 0);
89      -- TD_HS           : out   std_logic;
90      -- TD_RESET_N      : out   std_logic;
91      -- TD_VS           : out   std_logic;
92
```

Fig. 82: Identification of the ports for displays, etc.

15. In the "architecture body of `soc_system_top`" declare the signal needed to connect the `hps reset output` signal and the `displays_ena_n` signal to the `displays_test` module.
16. In the component declaration include de output corresponding to the `displays_ctrl port`, referred as "`displays_ctrl_external_connection_export`" , and the "`hps_0_h2f_reset_reset_n`" port

```
151         HPS_USB_NXT      : in    std_logic;
152         HPS_USB_STP      : out   std_logic
153     );
154 end entity soc_system_top;
155
156 architecture rtl of soc_system_top is
157 signal hps_fpga_reset_n:   std_logic;
158 signal displays_ena_n:        std_logic_vector(5 downto 0);
159
160     component soc_system is
161         port(
162             displays_ctrl_external_connection_export    : out   std_logic_vector(5 downto 0);
163             clk_clk                                     : in    std_logic                       := 'X';
164             hps_0_h2f_reset_reset_n                     : out   std_logic;
165             hps_0_io_hps_io_emac1_inst_TX_CLK           : out   std_logic;
166             hps_0_io_hps_io_emac1_inst_TX_CTL           : out   std_logic;
167             hps_0_io_hps_io_emac1_inst_TXD0             : out   std_logic;
```

Fig. 83: Adding "signals"

17. In the *port map* zone of the emplacement sentence of the *soc_system*, include the lines corresponding to the connection between the parallel port output and the *displays_ena_n* signal and the connection between the output port *hps_0_h2f_reset_reset_n* to the signal *hps_fpga_reset_n*.

```
237             hps_0_ddr_mem_dm                    : out   std_logic_vector(3 downto 0);
238             hps_0_ddr_oct_rzqin                 : in    std_logic                       := 'X';
239             reset_reset_n                       : in    std_logic                       := 'X'
240         );
241     end component soc_system;
242
243 begin
244     soc_system_inst : component soc_system
245         port map(
246             displays_ctrl_external_connection_export    => displays_ena_n,
247             clk_clk                                     => CLOCK_50,
248             hps_0_h2f_reset_reset_n                     => hps_fpga_reset_n,
249             hps_0_io_hps_io_emac1_inst_TX_CLK           => HPS_ENET_GTX_CLK,
250             hps_0_io_hps_io_emac1_inst_TX_CTL           => HPS_ENET_TX_EN,
251             hps_0_io_hps_io_emac1_inst_TXD0             => HPS_ENET_TX_DATA(0),
252             hps_0_io_hps_io_emac1_inst_TXD1             => HPS_ENET_TX_DATA(1),
253             hps_0_io_hps_io_emac1_inst_TXD2             => HPS_ENET_TX_DATA(2),
```

Fig. 84: Mapping the output of PIO system with dispays_ena_n.

18. Write the emplacement sentence for the *displays_test* module as shown in Fig. 85.

```
321         );
322
323         displays_test:
324         entity work.displays_test(rtl)
325          port map(clk       => CLOCK_50,
326                   nRst      => hps_fpga_reset_n,
327                   disp_ena_n  => displays_ena_n,
328                   disp0     => HEX0_N,
329                   disp1     => HEX1_N,
330                   disp2     => HEX2_N,
331                   disp3     => HEX3_N,
332                   disp4     => HEX4_N,
333                   disp5     => HEX5_N
334          );
335 end;
336
```

Fig. 85: mapping the connection with the hardware managing the displays

19. Save the project files and compile the project. This compilation should not take more than 15 minutes. Exit Q*uartus* when all errors are resolved.

6.1.5 Creating the SD Image.

Having into account that the hardware system has changed with the inclusion of a new element, the *displays_test* module, and the Parallel Output port *displays_ctrl*, it is necessary to create a new device tree file (*soc_system.dtb*), a new programming file (*soc_system.rbf*) and a new preloader (*preloader_mkpimage.bin*). To do so, follow these steps:

20. Following the instructions set in section 3.7 create the new *soc_system.rbf* file.
21. Following the instructions are given in section 3.8.1 generate the new preloader (*preloader_mkpimage.bin*)

22. Generate the new *soc_system.dtb* as described in section 3.8.2
23. Using the same *zimage, rootfs, uboot* and *uboot.src* that were used in chapter 7, generate the sdcard as described in 3.8.4, 3.8.5and 3.8.6
24. Boot the de1-soc. If there not have been any problem, now you could see the board six seven segments displays doing a funny blinking.

6.1.6 Testing the hardware using Linux GPIO .

The kernel configuration applied previously using Buidlroot includes as a software Device Driver the General Purpose Input Output (GPIO) support for Altera GPIO. This means that when using standard Altera Parallel IO IP we can take advantage of this software device driver already implemented by Altera for this kind of peripheral, instead of having to develop our own driver. The Linux kernel has been configured to generate this driver as a kernel loadable module. Buildroot locates it in the rootfs under the /lib/modules/<kernel-version>/kernel/drivers/gpio folder with the name "*gpio-altera.ko*". In this section, we will load manually this module that generates *entry-files* in Linux sys folder. This will allow us to manage the output lines in the PIO peripheral *displays_ctrl* connected to the *input enable* of any of the six seven segment displays available in the de1-soc board.

[sys folder]: /sys is a virtual file system that can be accessed to set or obtain information about the kernel's view of the system.

Boot the de1-soc and open a serial console session with putty. Now the first thing to do is loading the altera-gpio driver module:

```
$ cd /lib/modules/3.10.31-ltsi/kernel/drivers/gpio/
$ modprobe gpio-altera
```

A new folder will be created in the sys directory. There is one folder of each one of the GPIO lines managed with this driver. It is necessary to identify which is the entry corresponding to the *displays_ctrl port*.

```
$ cd /sys/class/gpio/
$ ls -l
```

A list of the available gpio lines is shown. An easy way to identify the port we are searching for is to get the address of the port. This address is kept in the *label* file inside the *gpioN* folder. Please, execute this command for each of the folders until the address 0xff230000 is shown. Remember that this was the address given to the *displays_ctrl* port in *qsys*.

```
$ cat gpiochipN/label
```

Usually, the number assigned to this port is 165. Therefore, for now on we will suppose the node for the displays port is 165. If this is not your case, replace this number with the one you have found. The next step is to create the resources to manage this gpio line. Execute this command:

```
$ echo 165 > export
$ cd gpio165
$ ls gpio165
```

The output received are the entry files available to control the behavior of the gpio pin. With the following commands, the pin is being configured as an output and take the "1" value. After its completion, the first display, HEX0, should be blank.

```
$ echo out > direction
$ echo 1 > value
```

This procedure should be replicated to control the rest of the output lines of the displays_ctrl port, replacing N = 165 for: 166, 167, 168, 169, and 170.
Of course, it is possible to erase the nodes created in the sys folder executing this command.

```
$ echo 165 > unexport
```

[Help]: The details about the use of GPIO Linux device driver can be found in the Linux kernel source under the folder Documentation/gpio/driver.txt

6.2 Creating Avalon Memory-Mapped peripheral. Blinker.

In the previous section, a hardware module connected to a PIO peripheral (with Avalon interface) port was designed. This time we will face up to design a complete custom peripheral with Avalon slave interface directly controlled by the ARM processor in the HPS. We will access it through straightforward readings and writings in its configuration, control and status registers. Such type of peripherals are not placed on the top level of the project, on the contrary, they must be connected to the system using *qsys* as a new element in the soc-system (we are going to design a complete custom IP block that will be available in the IP catalog of Qsys application). Previously it has to be created and added to the local components library. This section describes the whole process, from the modelling stage in a formal language like VHDL, to the connection in the system and the generation of the new BSP so that the new peripheral can be used from a Linux user space application. We will see how to generate a C application able to access to memory in user space avoiding the complexity of developing a specific Linux kernel device driver for the peripheral (we will afford this later). We maintain the principle of reducing the complexity of the peripheral to focus attention on the procedural aspects of the design process. The peripheral is called "blinker" and is based on the use of the switches, the buttons and the LEDs to carry out a funny game of blinking lights.

6.2.1 Blinker operation modes.

It is possible to control the functioning of blinker both interacting with the push buttons placed in the DE1SoC board, and through writings and readings to or from its control registers. *Blinker* has two modes of operation:

- MODE SWEEP: the lower 8 LEDs (LED0 - LED7) blink one by one, according to the following pattern: first LED0 is on, whereas the rest remain in an off state, then the second is on while the rest are off, then the third. When reached the eighth led, the sweep is realized backward, seventh, sixth, and so on.
- MODE BLINK: the information settled in the eighth lower switches (SW0 – SW7) is blinking in the LEDs.

This operation mode can be switched pushing the KEY3, o writing over the bit 0 of the CONFIG register. The speed of blinking or shifting can be reduced pushing button KEY1, or increased pushing button KEY0. The speed can also be changed directly to a specific value by writing to the SPEED register. The blink or shift can be stopped or resumed pushing the KEY2 button or writing over the bit 1 of the CONFIG register.

This peripheral can generate an interrupt if enabled when one of the buttons have been pushed by the user. Bit 2 of the CONFIG register acts as the enable/disable signal for the interrupt.

As it has been mentioned *blinker* can be operated by buttons or by its internal register. Table 12shows all the information related with the internal registers.

Table 12: Registers map of blinker peripheral.

Name	Address	Mode	Description	Initial Value
SPEED	0	Write only	Speed of blinking/shifting of LEDS 1: slowest, 15: fastest	0x8
CONFIG	1	Write only	Configuration register Bit 0: mode of operation (0: sweep, 1: blink) Bit 1: 0 (running), 1 (stopped) Bit 2: 0 (interrupt disabled), 1 (interrupt enabled)	0x1
LEDS	0	Read only	Current position of the LEDS 0: off state 1: on state	N.A
STATE	1	Read only	Peripheral status Bit 7 – 4: speed Bit 2: 0 (interrupt disabled), 1 (interrupt enabled) Bit 1: 0 (running), 1 (stopped) Bit 0: mode of operation (0: sweep, 1: blink)	N.A

6.2.2 Avalon Memory-Mapped Interface basics.

Our final goal is to control a memory mapped peripheral from software applications. As we can see in Fig. 7 and Table 5 the slaves peripheral implemented in the FPGA are available in the memory map starting in the address 0xFF200000. Then, HPS ARM processor can send commands to a peripheral by writing to the peripherals address space and get information back by reading from the peripheral's address space. On Altera's FPGAs, the easiest bus to interface the processor with a peripheral is the Avalon Multi Master interface. Avalon MM is a master-slave protocol, with a CPU being the master and the peripherals being the slaves. The Avalon Interface Specifications document [8] lists all signal roles for the Avalon-MM interface. However, the specification does not require all signals existing in an Avalon-MM interface. There is no one signal that is always required. Table 13 shows the basic set of signals, this is not an exhaustive list of course, but these are the ones that are involved in simple applications like the one we are developing. They allow reading, writing and interrupts management.

Table 13: Avalon interface basic signals

Name	Direction	Width	Description
address	input	1 to 64	the address of the slave being accessed
read	input	1	indicates whether a read operation is requested
readdata	output	8, 16, 32, 64, …1024	the data that will be read
write	input	1	indicates whether a write operation is requested
writedata	input	8, 16, 32, 64, .., 1024	the data to be written
irq	input	1	Interrupt Request. A slave asserts IRQ when it needs service.

The Avalon-MM interface is synchronous. Each Avalon-MM interface is synchronized to an associated clock interface. Fig. 86 describes the simplest Avalon-MM interface read and write bus cycle. The chip-select generation is transparent to the slave. Therefore it should not be contemplated in the hardware design.

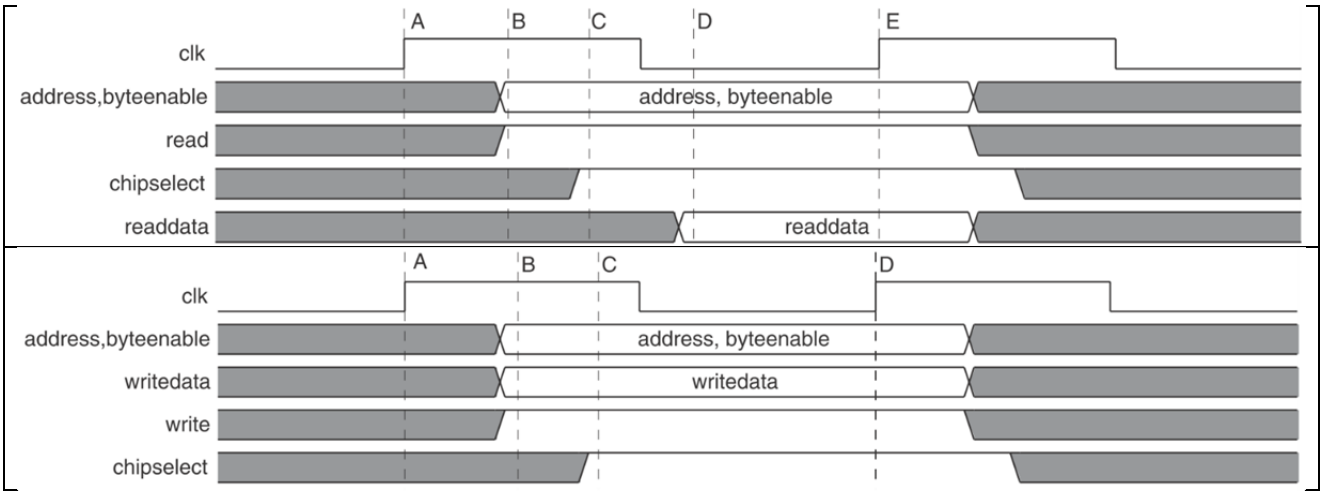


Fig. 86: Read and write cycles using Avalon Bus

In order to detect changes in the state of the FPGA peripherals, the HPS has to poll continuously the FPGA over the bus. If the state changes infrequently, but we want the software to get notified of the change quickly, polling can be rather inefficient. In this case, it would be better if the peripheral implemented in the FPGA could asynchronously notify the HPS of change. The way the FPGA can notify some event without needing the processor to poll the peripheral to know its state is through interrupts mechanism. Interrupts are essentially signals going from the FPGA to an interrupt controller on the HPS. The FPGA can make an interrupt request (IRQ) by asserting the interrupt signal high. When an IRQ reaches the HPS, this saves its current state and jumps to an interrupt service routine (ISR). The ISR should service the IRQ by reading or writing some data from the peripheral. Once the ISR has returned, the processor jumps back to its original state.

As with other signals sent between FPGA and HPS on the Cyclone V, interrupt signals go through an Avalon interface. The interrupt interface is quite simple, only a single one-bit IRQ signal is required. According to the Avalon interrupt interface specification. The IRQ signal should not be deasserted until the slave has determined that it has been serviced. In our case, we consider the IRQ serviced once one of its registers is read.

6.2.3 Hardware Modelling in VHDL of a peripheral with Avalon interface .

Bellow theses lines (see Fig. 87) there is some VHDL code to model the *blinker* peripheral with its own Avalon interface. The hardware modelled try to operate as aforementioned. However, the code is incomplete, therefore it must be completed before its incorporation into the rest of the system. The parts in which additional code must be included have been marked with the text --TBC (To Be Completed). Thus, the first activity proposed is the analysis of the proposed code and the completion of it. If necessary, the student might make a *testbench* to ensure the correct behavior of the model.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;


entity blinker is
	port(	clk:		in	std_logic;
		reset:		in	std_logic;

		switches:	--TBC
		buttons:	--TBC
		leds:		--TBC

		address:	--TBC
		write:		in	std_logic;
		writedata:	in	std_logic_vector(7 downto 0);
		read:		in	std_logic;
		readdata:	-- TBC
		irq:		buffer std_logic
		);
end entity;

architecture rtl of blinker is

-- Pulse shaper signals
signal sync_buttons:std_logic_vector(3 downto 0);
signal cur_value:   std_logic_vector(3 downto 0);
signal last_value:  std_logic_vector(3 downto 0);

-- Speed Control signals
signal speed:       std_logic_vector(3 downto 0);
signal delay:       std_logic_vector(3 downto 0);
signal faster:      std_logic;
signal slower:      std_logic;

-- Bliker signals
signal count:       std_logic_vector(23 downto 0);
signal position:    std_logic_vector(3 downto 0);
signal running:     std_logic;
signal mode:        std_logic;
signal pause:       std_logic;
signal change_mode: std_logic;

-- Interrupt controller signals
signal ena_irq:     std_logic;
```

```
begin
	-- Pulse shaper ----------------------------------------------------------
	pulse_shaper:
	process(clk, reset)
	begin
		if reset = '1' then
			cur_value <= "1111";
			last_value <= "1111";
		elsif clk'event and clk = '1' then
			cur_value <= buttons;
			last_value <= cur_value;
		end if;
	end process;

	sync_buttons <= -- TBC

	-- speed control ---------------------------------------------------------
	faster <= sync_buttons(0);
	slower <= sync_buttons(1);
	process(clk, reset)
	begin
		if (reset = '1') then
			speed <= -- TBC

		elsif clk'event and clk = '1' then
			if (faster = '1' and speed /= 15) then
				-- TBC

			elsif (slower = '1' and speed /= 1) then
				-- TBC

			elsif write = '1' and address = '0' then
				-- TBC
			end if;
		end if;
	end process;

	-- blinker ---------------------------------------------------------------
	change_mode <= sync_buttons(3);
	pause <= sync_buttons(2);

	-- leds manager
	process(position, switches, mode)
	begin
		if mode = '0' then
			case position is
				when "0000" => leds <= "00000001";
				when "0001" => leds <= "00000010";
				when "0010" => leds <= "00000100";
				when "0011" => leds <= "00001000";
				when "0100" => leds <= "00010000";
				when "0101" => leds <= "00100000";
				when "0110" => leds <= "01000000";
				when "0111" => leds <= "10000000";
				when "1000" => leds <= "01000000";
				when "1001" => leds <= "00100000";
				when "1010" => leds <= "00010000";
				when "1011" => leds <= "00001000";
				when "1100" => leds <= "00000100";
				when "1101" => leds <= "00000010";
				when others => leds <= "00000000";
			end case;
		else
			if position(0) = '0' then
				leds <= switches;
			else
				leds <= not switches;
			end if;
		end if;
	end process;
```

```
-- Mode controller
process(clk, reset)
begin
	if reset = '1' then
		-- TBC

	elsif clk'event and clk = '1' then
		if change_mode = '1' then
			mode <= not mode;

		elsif write = '1' and address = '1' then
			mode <= writedata(1);

		end if;
	end if;
end process;

-- running controller
process(clk, reset)
begin
	if reset = '1' then
		running <= '1';

	elsif clk'event and clk = '1' then
		if -- TBC
			running <= not running;

		elsif write = '1' and address = '1' then
			-- TBC

		end if;
	end if;
end process;


-- shift generation
process(clk, reset)
begin
	if (reset = '1') then
		count <= X"000000";
		position <= "0000";

	elsif clk'event and clk = '1' then
		if (running = '1') then
			if (count = 0) then
				count <= delay&X"00000";

				if (position = 13) then
					position <= "0000";

				else
					position <= position + 1;

				end if;
			else
				count <= count - 1;

			end if;
		end if;
	end if;
end process;

delay <= 16 - speed;
```

```
	-- Avalon read control -------------------------------------------------

	readdata <= speed&'0'&ena_irq&mode&running     when read = '1' and -- TBC else
		    leds                               when read = '1' and -- TBC else
		    (others => '0');


	-- Avalon interrupt control --------------------------------------------

	-- enable control
	process(clk, reset)
	begin
		if reset = '1' then
			ena_irq <= '0';

		elsif clk'event and clk = '1' then
			if write = '1' and address = '1' then
				ena_irq <= -- TBC

			end if;
		end if;
	end process;

	-- interrupt generation control
	process(clk, reset)
	begin
		if reset = '1' then
			irq <= '0';

		elsif clk'event and clk = '1' then
			if ((sync_buttons(0) = '1') or (sync_buttons(1) = '1') or
			    (sync_buttons(2) = '1') or (sync_buttons(3) = '1'))
			     and -- TBC
			       irq <= '1';

			elsif read = '1' and address = '1' then
				irq <= '0';

			end if;
		end if;
	end process;

end architecture;
```

Fig. 87: Incomplete VHDL code of the blinker peripheral with Avalon Interface.

6.2.4 Exercise 3: analyse the code and answer these questions:

f) Analyse and complete the model. Make a testbench to probe the correct operation of the model.
g) What is the relation between the “speed” value and the frequency of blinking?
h) What register should be read to deactivate the int request signal?
i) What is the width of the Avalon data bus used?
j) What is the utility of the piece of code called “pulse shaper”?
k) What part of the code prevents the value of speed to go beyond limits?
l) Are *faster, slower, change_mode* and *pause* signals really needed?
m) What is the mission of signals *count* and *position*?
n) What part of the code ensures the registers take their initial value at the start of operation?

6.2.5 Building the system in QSYS.

Now that you have an Avalon peripheral, we can hook it up to the processor. For this, we will need to use Altera's Qsys tool.

1. You can open Qsys from Quartus by going to "`Tools->Qsys`". You can also click the Qsys icon in the toolbar. When asked, open the file *soc_system.qsys* to load the system we have been creating till now.
2. Now you will need to add the *blinker* to the system. Since this is a custom module, you will first need to create a new *qsys* component for it. Go to the "`Library`" window and double-click on "`New Component`" (see Fig. 88).

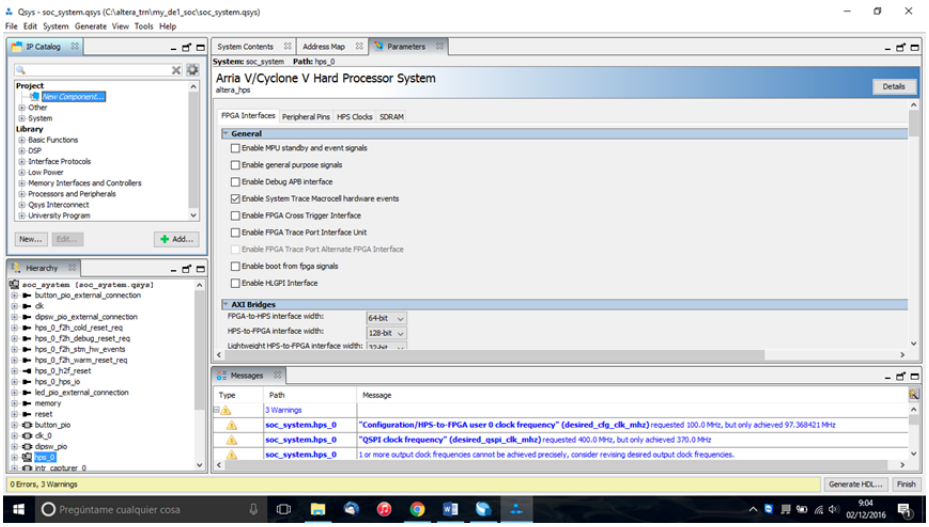


Fig. 88: Creation of a new component for the Library

In the newly opened window, select the following options.

3. Under the "`Component Type`" tab, change "`Name`" and "`Display name`" to "*blinker*" (see Fig. 89).

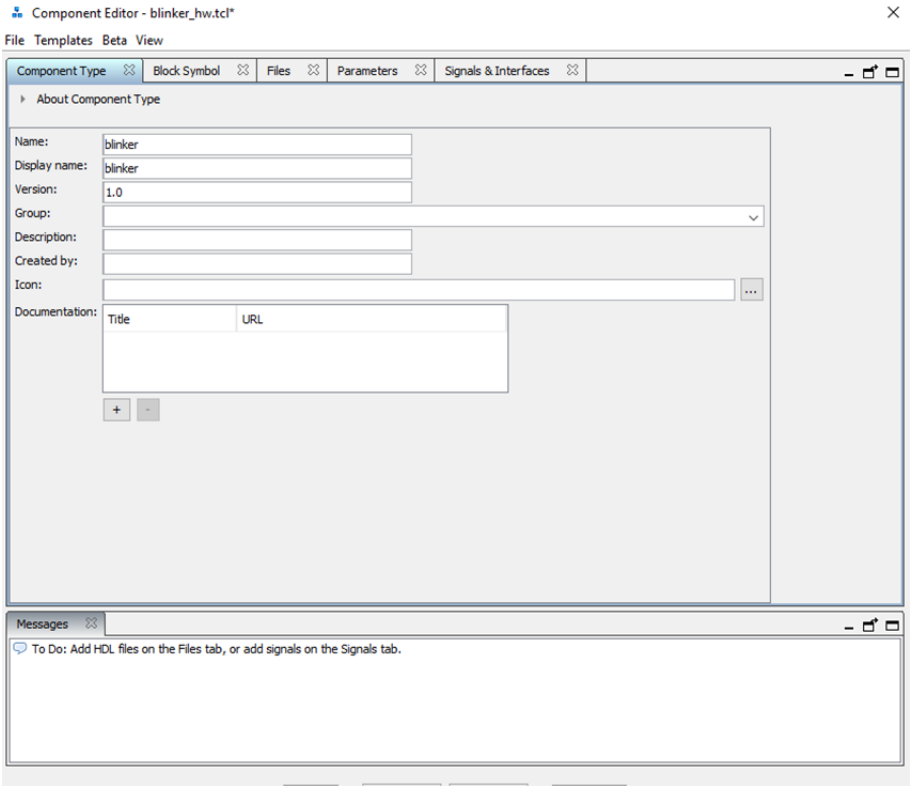


Fig. 89: Edition of the blinker component

4. Go to the "`Files`" tab and click the "+" button under "`Synthesis Files`" to add a new file to this component. Choose the "blinker.vhd" file (see Fig. 90).

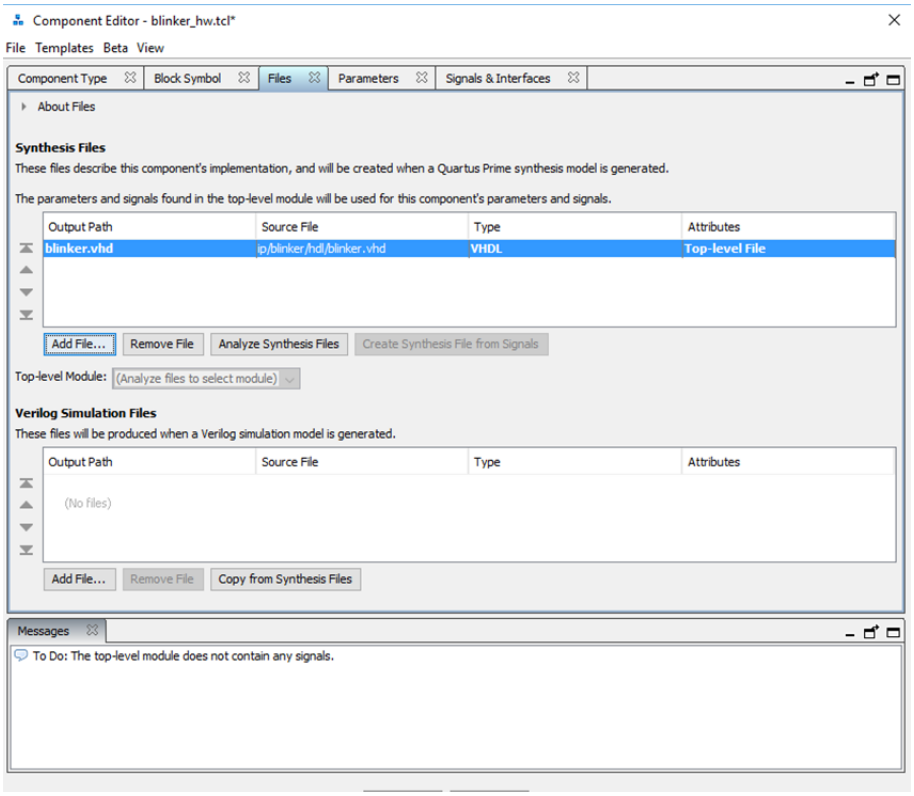


Fig. 90: Inclusion of blinker.vhd file for the component.

5. Click "`Analyze Synthesis Files`" to check the file for syntax errors and pull out the signals.

6. Select "`View->Signals`" (Fig. 91) where you will indicate the purpose of the signals in the module. A new tab is displayed as shown in
7. Fig. 92.

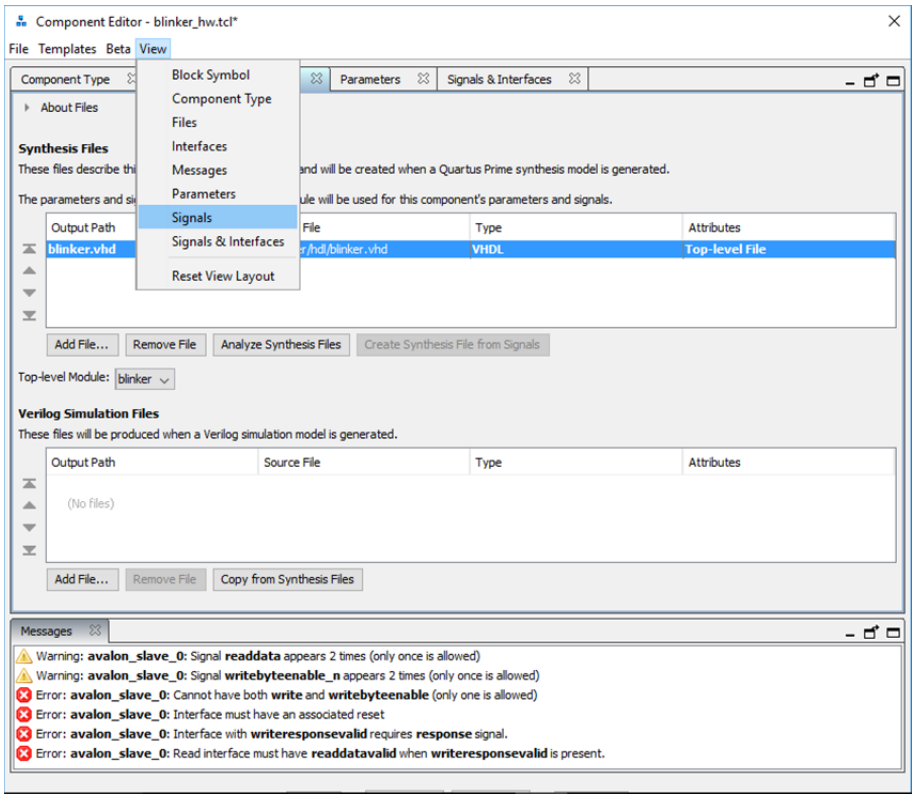


Fig. 91: Accessing signals view for the component

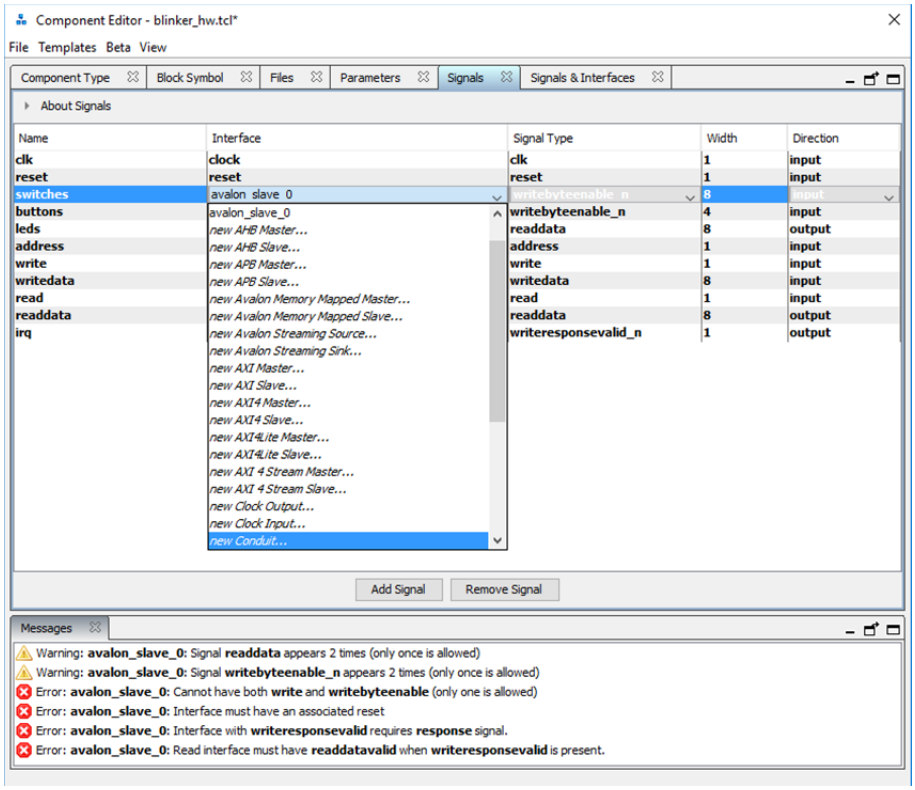


Fig. 92: Signals tab

8. Make sure that the "*write*", "*read*", "*readdata*" and "*writedata*" signals are on an avalon slave interface called "`avalon_slave_0`" and that the signal types are "`write`", "`read`", "`readdata`" and "`writedata`", respectively.
9. Make sure "*clk*" and "*reset*" are on "`clock`" and "`reset`" interfaces with signal types "`clock`" and "`reset`" respectively.
10. Change the interface for "*switches*" to "`new Conduit`". This will create an interface called "`conduit_end`".
11. Assign "*buttons*" and "*leds*" to also be on the "`conduit_end`" interface. The conduit interface type means that the signals will not be used internally by the *Qsys* interconnect and will instead be exported out to the top-level.
12. Make sure to assign "*irq*" to an "`Interrupt Sender`" interface and set the signal type to "`irq`" (see Fig. 93).

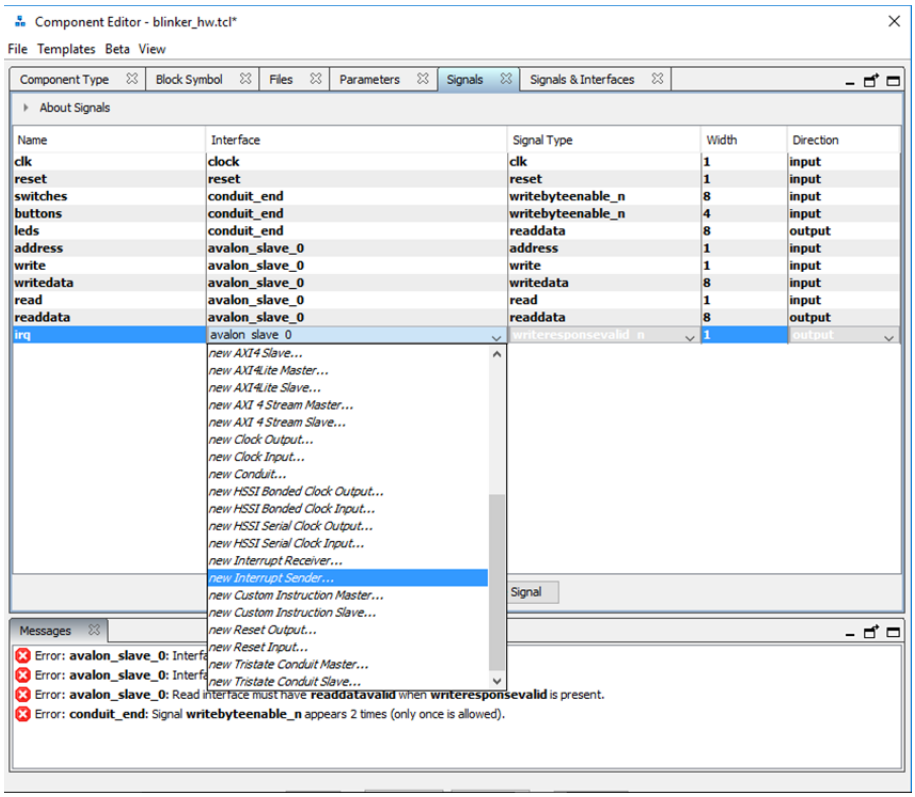


Fig. 93: Assigning interface to irq signal

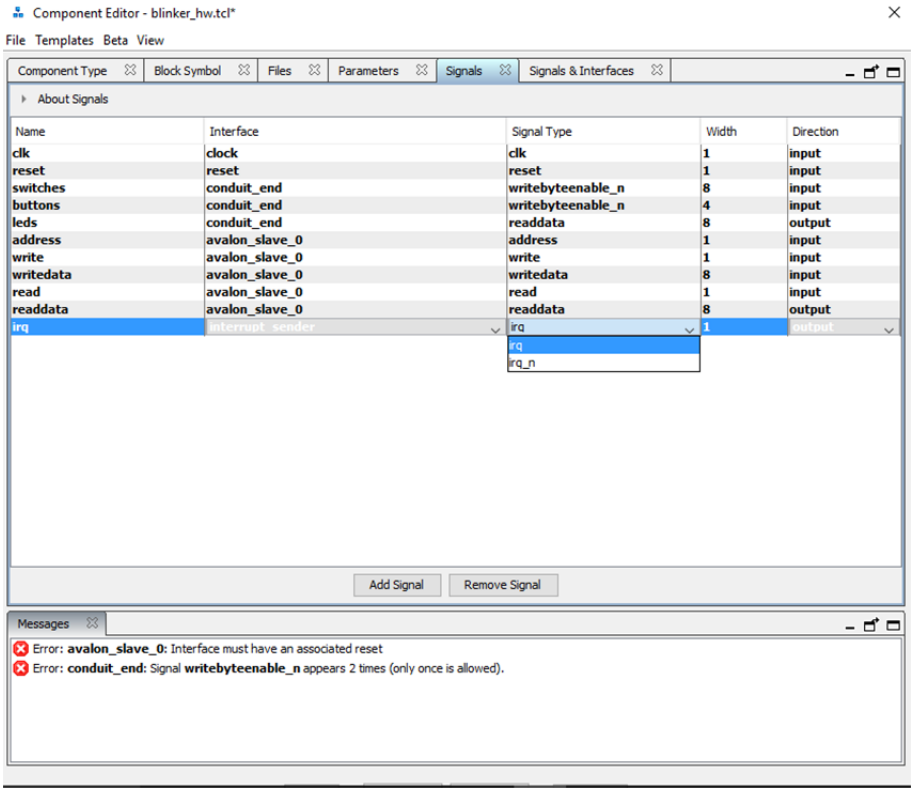


Fig. 94: Assigning signal type

13. Change the signal type for all of the conduit signals to "`switches`", "`buttons`" and "`leds`".

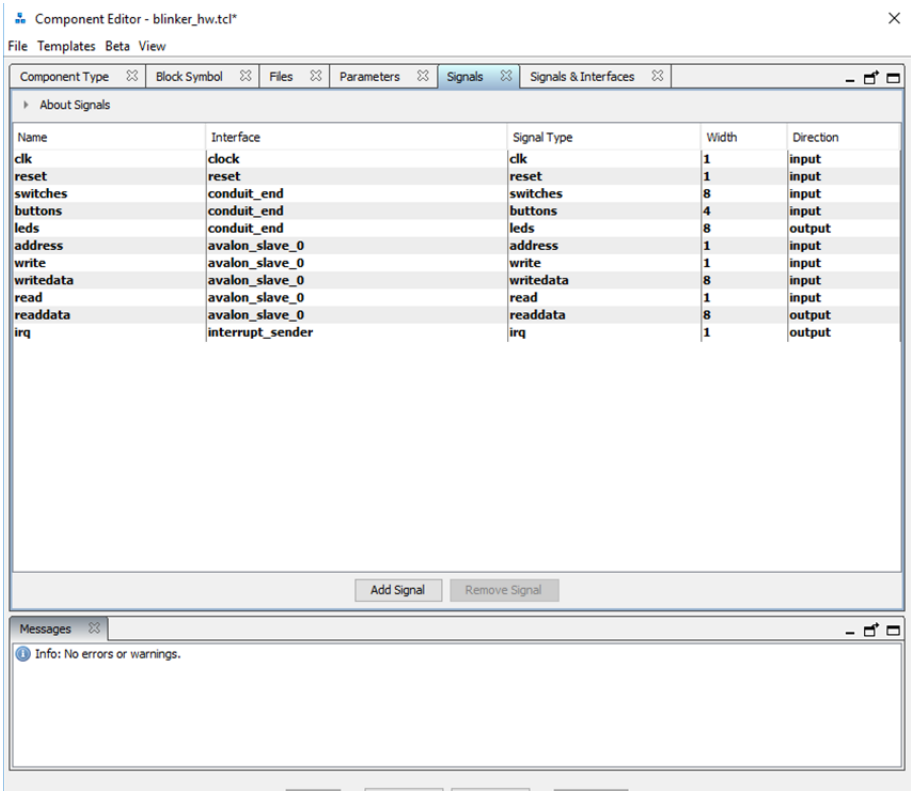


Fig. 95: Final assignation of interfaces and Signal Types.

14. Go to the "`Interfaces`" tab. Make sure there are four interfaces: "clock", "reset", "conduit_end", and "avalon_slave_0". If there are others, you can remove then using "`Remove Interfaces with no Signals`".

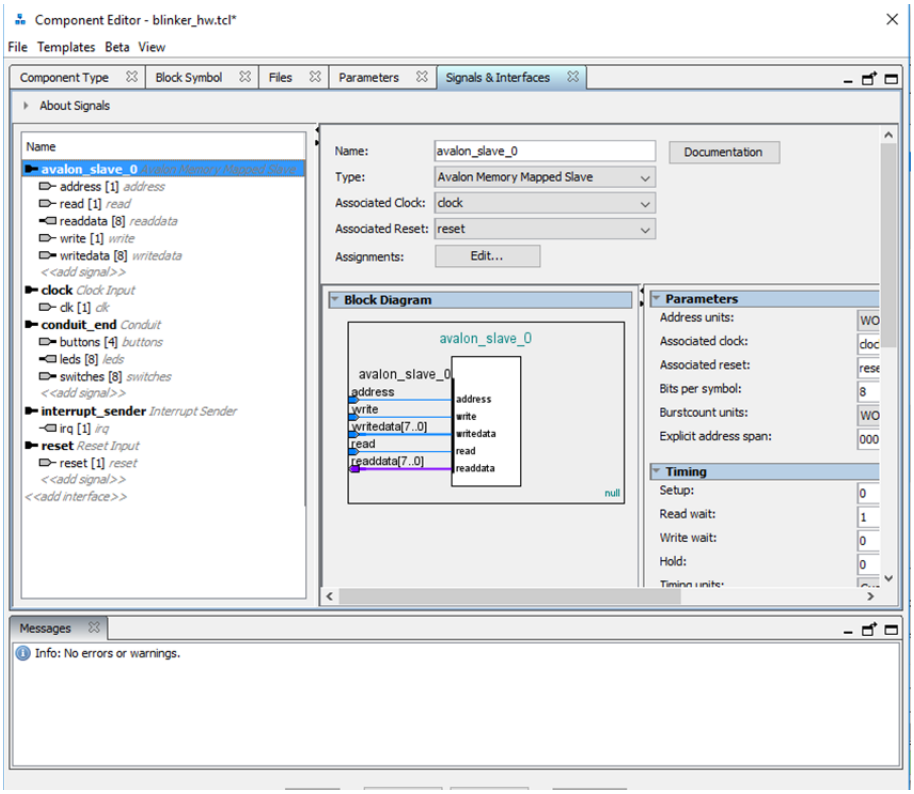
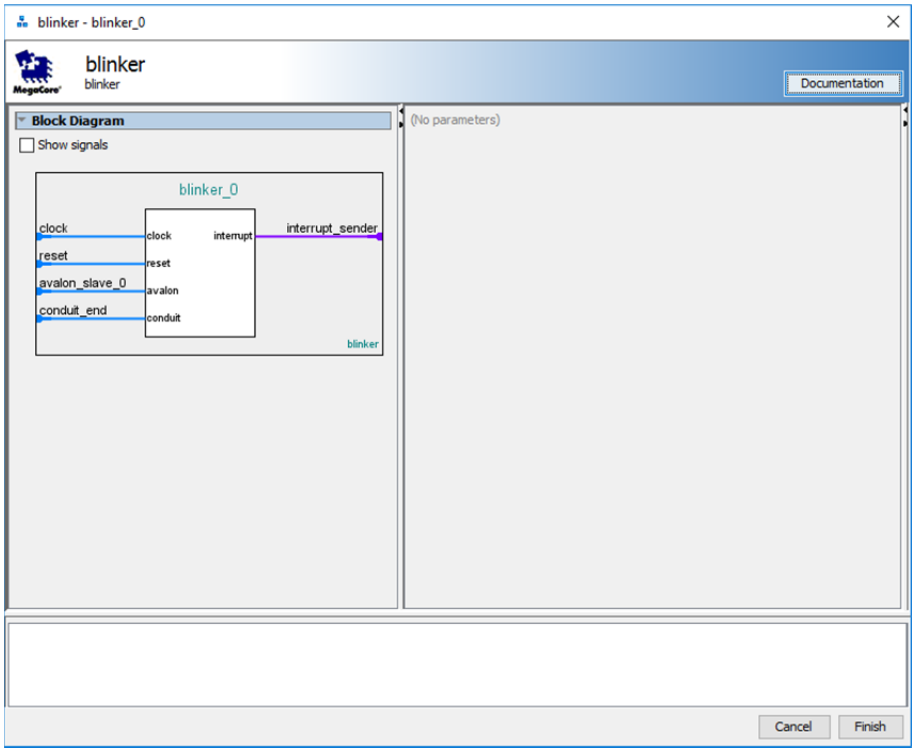


Fig. 96: Reviewing signals and interfaces.

15. Make sure "*reset*" has "`clock`" as its associated clock.
16. Make sure that "*avalon_slave_0*" has "`clock`" as its associated clock and "`reset`" as its associated reset.
17. Press "`Finish`" and save this component. You should see a new file called "blinker.tcl" in your project directory and a component named "blinker" under "`Project`" in the library window. Add this component to your system. You can just press "`Finish`" in the add dialog as there are no options.



Now you must connect the new component to the rest of the system.

18. Connect the "*clk*" output of the "*clk_0*" component to the "*clock*" input of "*blinker_0".*
19. Connect the "*clk_reset*" output of "*clk_0*" to "*reset*" of "*blinker_0*".
20. Connect "*h2f_lw_axi_master*" of "*hps_0*" to "*avalon_slave_0*" of "*blinker_0*".
21. Export "*conduit_end*" of "*blinker_0*" as "*blinker_external_connection*".
22. Assign base address offset of *0x0004_0000*. Lock the address with the symbol at the right of the address.
23. Click on the IRQ to connect the interrupt_sender to the interrupt receiver in the HPS. Assign interrupt 1.

In the end, your "System contents" window should look something like Fig. 97.

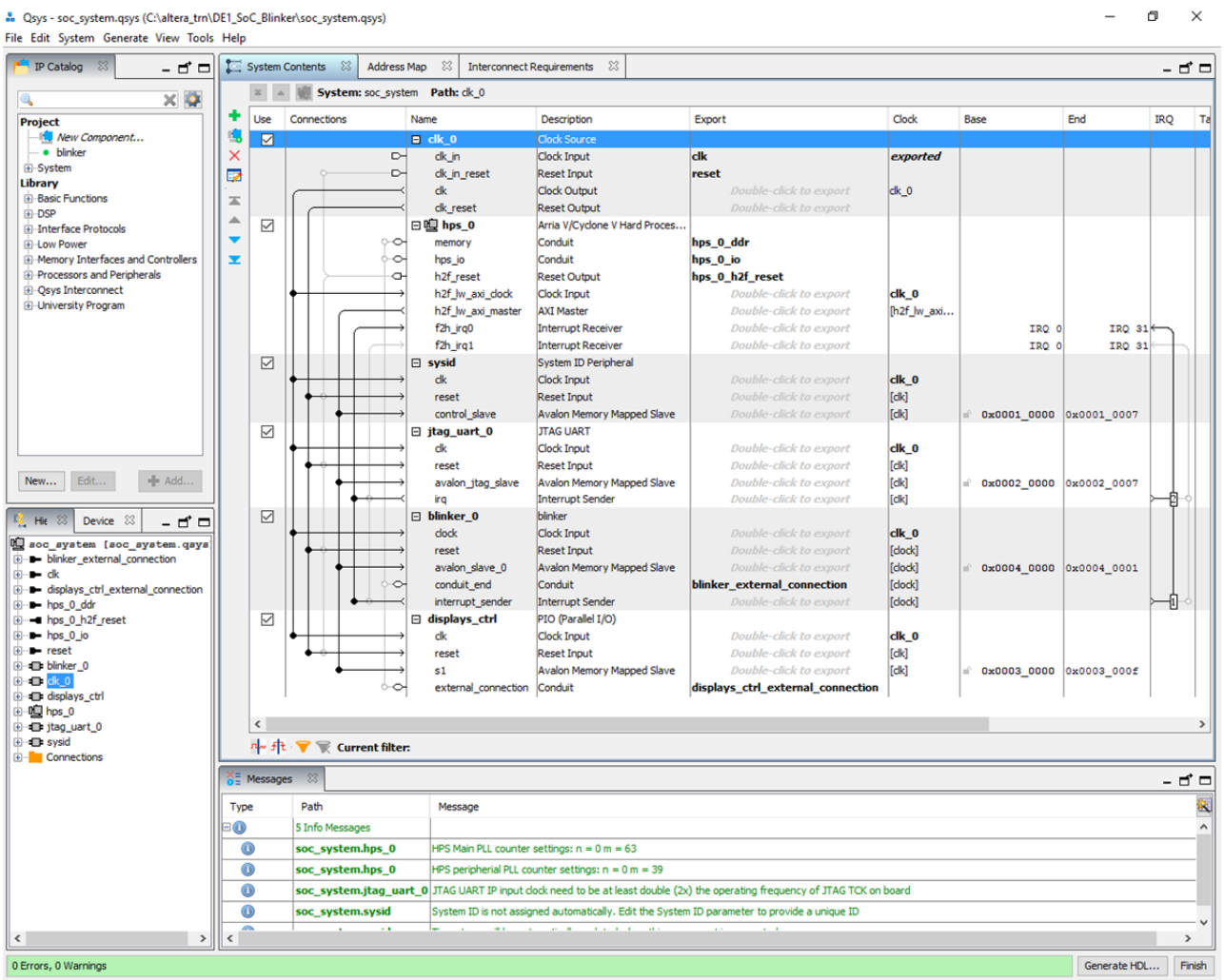


Fig. 97: Final connection of the component in Qsys.

24. You have now finished the system, so save it as "*soc_system.qsys*". You can now generate the system by clicking "`Generate -> Generate`" from the menu. In the "`Generation`" dialog, make sure "`Create HDL design files for synthesis`" is set to VHDL. You can also change the "`Output Directory`" to a directory of your choosing. By default, it will be a subdirectory of your project directory called "*soc_system*". Press the "`Generate`" button, and *Qsys* will begin producing VHDL files for this system. Once the system finishes generation successfully, you can close *Qsys*.

Once you have added "*blinker*" to the "*soc_system.qsys*", you must add it to the top-level file, "*soc_system_top.vhd*". First, the top-level inputs will have to change to accommodate the exported "*switches*", "*buttons*" and "*leds*" interfaces of the system.

25. Open the "*soc_system.cmp*" file, you can find it in folder "*soc_system*". It contains the component instance for *soc_system*. In the beginning, there are three lines that show the corresponding three elements of the interface, "`blinker_external_connection_xxxx`".
26. Copy these three lines in the component instantiation of *soc_system* in the top level file of the project "*soc_system_top.vhd*".

```
        HPS_USB_NXT      : in    std_logic;
        HPS_USB_STP      : out   std_logic
    );
end entity soc_system_top;

architecture rtl of soc_system_top is
signal hps_fpga_reset_n:   std_logic;
signal displays_ena_n:        std_logic_vector(5 downto 0);

    component soc_system is
        port(
            blinker_external_connection_switches  : in    std_logic_vector(7 downto 0)  := (others => 'X'); -- switches
            blinker_external_connection_buttons   : in    std_logic_vector(3 downto 0)  := (others => 'X'); -- buttons
            blinker_external_connection_leds      : out   std_logic_vector(7 downto 0);                     -- leds
            displays_ctrl_external_connection_export    : out   std_logic_vector(5 downto 0);                     -- ex
            clk_clk                               : in    std_logic                     := 'X';
            hps_0_h2f_reset_reset_n               : out   std_logic;                                      -- reset_n
            hps_0_io_hps_io_emac1_inst_TX_CLK     : out   std_logic;
            hps_0_io_hps_io_emac1_inst_TX_CTL     : out   std_logic;
            hps_0_io_hps_io_emac1_inst_TXD0       : out   std_logic;
            hps_0_io_hps_io_emac1_inst_TXD1       : out   std_logic;
```

Fig. 98: Adding new conduits to the soc_system component definition in top hierarchy file.

Since these three ports must be physically connected to the digital I/O where the KEY_Nn, LEDRn and SWn elements of the board are actually connected, it is necessary to add these connections to the `port map` definition.

```
            hps_0_ddr_oct_rzqin                   : in    std_logic                     := 'X';             -- oct_rzqi
            reset_reset_n                         : in    std_logic                     := 'X'
        );
    end component soc_system;

begin
    soc_system_inst : component soc_system
        port map(
            blinker_external_connection_buttons   => KEY_N,
            blinker_external_connection_leds      => LEDR(7 downto 0),
            blinker_external_connection_switches  => SW(7 downto 0),
            displays_ctrl_external_connection_export    => displays_ena_n,
            clk_clk                               => CLOCK_50,
            hps_0_h2f_reset_reset_n               => hps_fpga_reset_n,
            hps_0_io_hps_io_emac1_inst_TX_CLK     => HPS_ENET_GTX_CLK,
            hps_0_io_hps_io_emac1_inst_TX_CTL     => HPS_ENET_TX_EN,
```

Fig. 99: Mapping the physical connection.

27. Now the system is complete in *quartus*, it is only left to compile the whole project "`Processing-> Start Compilation`".

As you know this process may take some time depending on the computer. At the end, the `soc_system.sof` will be available.

6.2.6 Programming the FPGA with Quartus Programmer.

In previous sections you have configured the FPGA, using u-boot, when the HPS is booting from the *SDcard.* But for debugging purposes when we want to verify only the hardware implementation is time-saving to configure the FPGA directly from *Quartus. This avoids* to copy the *.rbf* file to the *SDcard and reboot the DE1-SoC*. This is particularly useful when in the phase of designing the system, where test operations are frequently done. This section shows how to carry out this process. Ensure that the USB Blaster cable is connected as was described in section 3.8.7, and that the DE1-SoC is powered on.

1. Before starting the use of Programmer verify that your Altera USB-blaster is detected on your computer. In Linux Ubuntu you can verify it following the instructions described in [9].
2. Within Quartus Prime, select "`Tools -> Programmer`"
3. Select "`hardware setup`" (top left side of the Programmer window, Fig. 100) and ensure that the currently selected hardware is `DE-SoC [USB-1]`. It should be selected by default. If not currently selected, double click on `DE-SoC` in the available hardware items list.

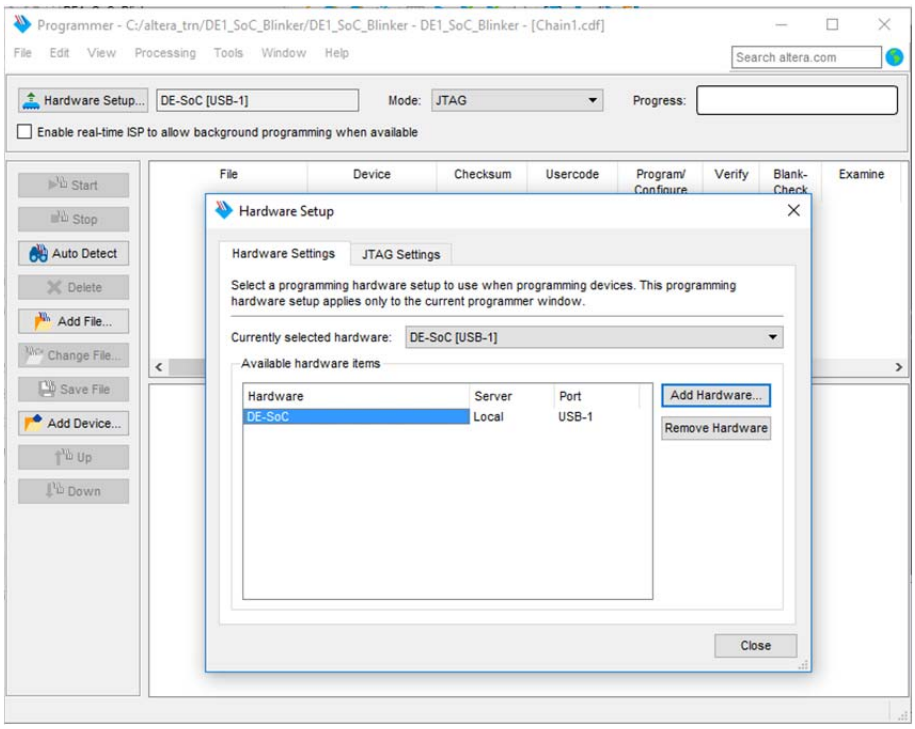


Fig. 100: Selection of the USB-blaster in Programmer.

4. Select `Close` and then "`Auto Detect`" when returned to the main programmer window (see Fig. 101).

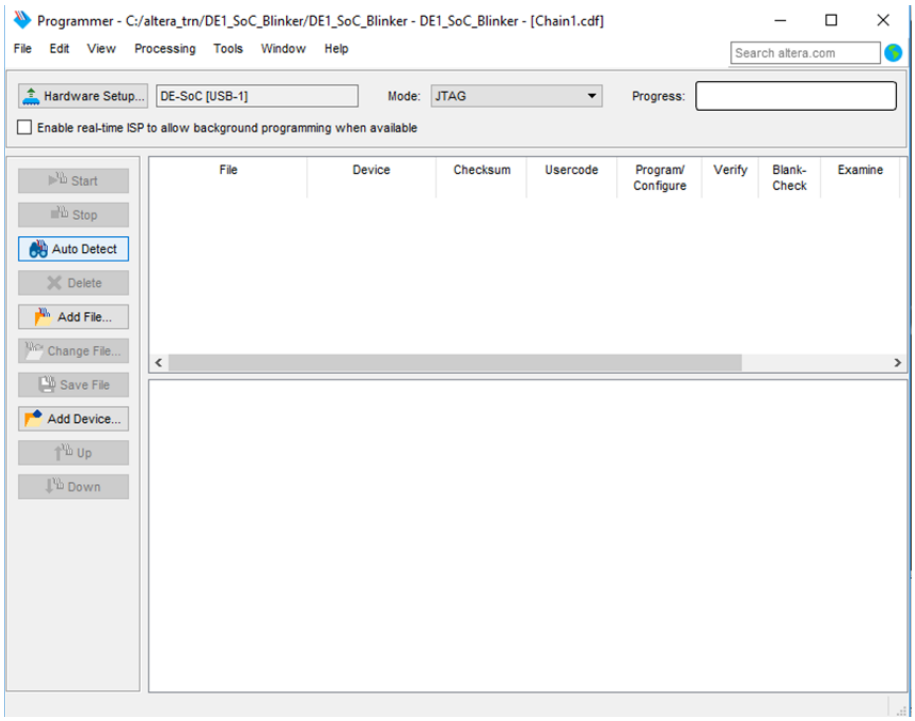


Fig. 101: Programmer with the USB-blaster for DE1-SoC added.

5. Select the correct device: "`5CSEMA5`" and then `OK (see` Fig. 102`).`

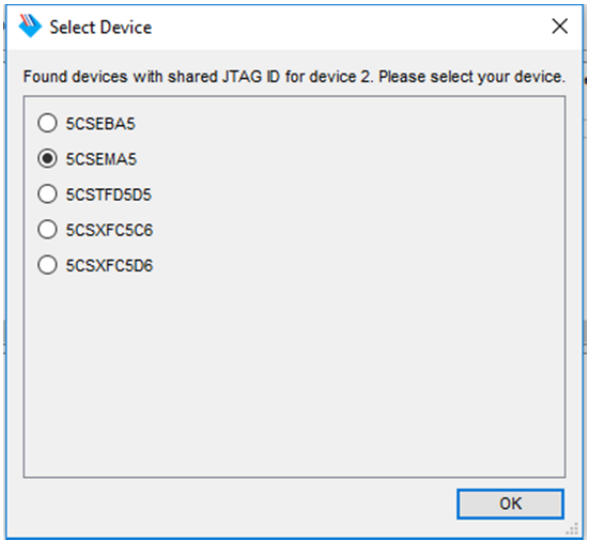


Fig. 102: Selection of the FPGA device.

The Programming window should now appear as shown in Fig. 103.

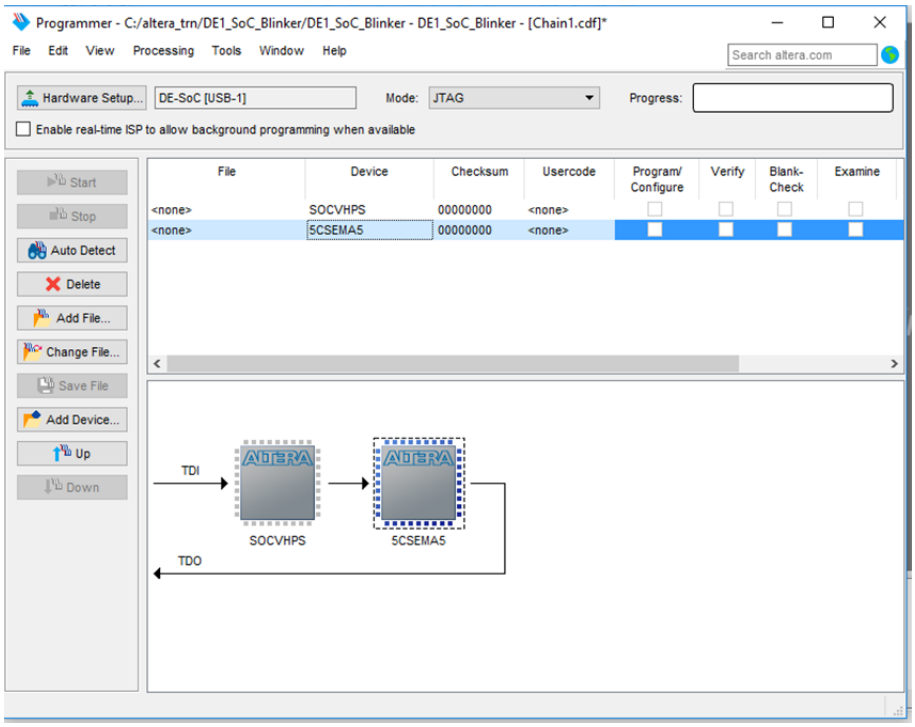


Fig. 103: Devices detected by JTAG interface (HPS and FPGA).

6. Select the row for the 5CSEMA5
7. Select, `Change File`
8. If you compiled the design, select the “cyclonevbsp.sof” in the .\output_files directory
9. Select `Open.`
10. Select the `checkbox` in the `Program/Configure` column, and a check will appear. The window should now look as shown in Fig. 104. If not, delete any extra device.
11. Press the `Start` button to program the FPGA. After programming the FPGA, the progress indicator should indicate 100% (Successful) as shown in Fig. 104. There should be no error message and, the LEDs will start blinking. Feel free to operate the blinker with the buttons and switches and verify its correct operation.

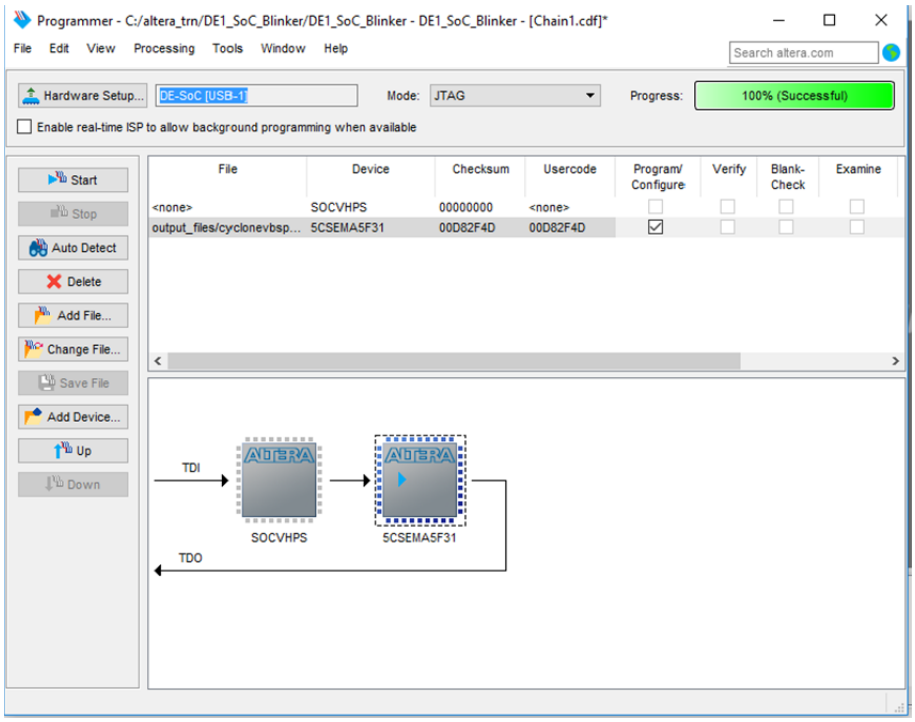


Fig. 104: FPGA configured using Programmer

12. Select `File -> Exit` to close the Programmer window.
13. Select the "No" button as shown to complete closing the Programmer window without saving.

6.2.7 Creating the new BSP.

The hardware design of our peripheral has reached the end, now the construction of the demo system is finished. However, there remain a bunch of tasks to make visible the new peripheral *blinker* to the software layers above. If we want the *blinker* to be recognized by the operating system and consequently be able to be used by software applications we need to include it in the device tree blob, that is, to introduce the corresponding node in the *soc_system.dts* file. This operation has been already described but there is a slight difference this time that is related to the further development of a driver. Let's see the procedure in detail:

1. Open a *SoC EDS Command Shell*
2. Create the new *soc_system.rbf* and *preloader_mkpimage.bin* files following the usual process. Copy the newly created files to *your image folder*, to get them ready to form part of the new SDcard image.

To generate the "*soc_system.dtb*" file, however, we will follow a different path, though.

3. First of all, we need to make the new "*soc_system.dts*" file, executing this command.

```
$ sopc2dts --input soc_system.sopcinfo\
  --output soc_system.dts  --type dts\
  --board soc_system_board_info.xml\
  --board hps_common_board_info.xml\
  --clocks
```

After completion, the *soc_system.dts* should now appear among the pile of files that populate our root folder project. This would be enough, for the moment, to give access to the peripheral from a user application, using the *mmap* system call on the "/dev/mem" device file, which represents physical memory,

to map the HPS-to-FPGA bridge's memory into the process memory. However, we must perform an additional task if we want the hardware to be managed through some driver that sets up an interface between our userspace code and the hardware. This allows us to avoid having to *mmap* "/dev/mem", which is quite unsafe.

4. Open the *soc_system.dts* file and search for the *blinker* and *displays_ctrl* nodes. As you see, the *displays_ctrl* hardware block has an associated driver, "`altr,pio-16.0`" or "`altr,pio-1.0`", that's the reason we could use the driver loaded at kernel boot, *gpio-altera.ko*, to write to this port. Though, you may also be aware that *blinker* does not have any associated driver.

```
blinker_0: unknown@0x100040000 {
     reg = <0x00000001 0x00040000 0x00000002>;
     interrupt-parent = <&hps_0_arm_gic_0>;
     interrupts = <0 41 4>;
     clocks = <&clk_0>;
}; //end unknown@0x100040000 (blinker_0)

displays_ctrl: gpio@0x100030000 {
     compatible = "altr,pio-16.0", "altr,pio-1.0";
     reg = <0x00000001 0x00030000 0x00000010>;
     clocks = <&clk_0>;
     altr,gpio-bank-width = <6>;
     resetvalue = <0>;
     #gpio-cells = <2>;
     gpio-controller;
}; //end gpio@0x100030000 (displays_ctrl)
```

5. To associate *blinker* to a driver, whose development we will undertake soon, we must modify this node so that the following appears

```
blinker_0: blinker@0x100040000 {
     compatible = "blinker,driver-1.0";
     reg = <0x00000001 0x00040000 0x00000002>;
     interrupt-parent = <&hps_0_arm_gic_0>;
     interrupts = <0 41 4>;
     clocks = <&clk_0>;
}; //end blinker@0x100040000 (blinker_0)
```

Now, this *device tree source* (soc_system.dts) must be compiled into a *device tree blob* (soc_system.dtb). There are there two possible methods. The first is by using the Device Tree Compiler executing this command in the "SoC EDS command shell":

```
$ dtc -I dts -O dtb -o soc_system.dtb soc_system.dts
```

The second one is by ordering Buildroot to compile the *.dts* file and generate the *.dtb*, together with the rest of the binaries. For this to be done it is only needed to fill the "`Device Tree Source Filenames`" field in the "`Linux Kernel`" section with this information: `$<yourpathtofolder>/soc_system.dts`, and to copy the soc_system.dts just created to the proper folder. Then, outside the "SoC EDS command shell" navigate to the Buildroot's base directory and enter the command:

```
$ make linux-rebuild
```

6. Choose the method you like the most, though if you select the first do not forget to copy the newly generated *soc_system.dtb* to the Buildroot's Image binaries folder, before making the new SDcard image.

6.2.8 Customizing your embedded Linux distribution.

Before generating the *SDcard,* we think this the right time to show some more interesting features of Buildroot that might be helpful to customize your embedded Linux distribution. Sometimes, it is necessary to modify, remove or add some files to the *rootfs* that have been generated automatically by Buildroot. These tasks must be done before the final image is created. Buildroot offers you the chance to carry out these tasks in an automatic way too. We are speaking about de "*custom scripts to run before creating filesystem im*ages" and the "*custom scripts to run after creating filesystem images*" items in the "*system configuration*" section of the Buildroot configuration options.

Why is this feature useful in our case? Well, on the one hand, the prompt used by the default configuration of Buildroot is slightly misleading, because it is not shown where you are placed inside the *rootfs* at all times. The prompt scheme is kept in the profile file, inside the "*/etc/*" folder, whose content is shown below. You should edit this file with any text editor tool, "*vi*" for example, and write the text in bold format. The problem is that this solution is not permanent if you generate a new image, these changes are lost, and you must repeat the process to get your favourite prompt.

```
#profile file in /etc/

/export PATH=/bin:/sbin:/usr/bin:/usr/sbin

if [ "$PS1" ]; then
     if [ "`id -u`" -eq 0 ]; then
           export PS1='\u@\h:\w\$ '
     else
           export PS1='$ '
     fi
fi

export PAGER='/bin/more '
export EDITOR='/bin/vi'

# Source configuration files from /etc/profile.d
for i in /etc/profile.d/*.sh ; do
     if [ -r "$i" ]; then
           . $i
fi
     unset i
done
```

On the other hand, the Buildroot default configuration for the *ssh* package is "*remote root accesses not allowed*". This is safer than other configurations, but awkward in our case because our only user is root (unless you have added one using the user tables) and it is easier to debug with eclipse using *root* as the default user. To allow remote *root* accesses involve some changes in *sshd* configuration files. Below these lines an excerpt of the "*/etc/ssh/sshd_config*" file is shown. The line *PermitRootLogin* must be uncommented and with "*yes*" as an option. Again, you may want this change to be permanent even when you make a new *SDcard* image.

```
#sshd_config file in /etc/ssh/
.
.
.
# Authentication:

#LoginGraceTime 2m
PermitRootLogin yes
#StrictModes yes
#MaxAuthTries 6
#MaxSessions 10
.
.
.
```

Besides, you may want your executable applications to be placed in your "*/home/*" folder when creating the new image. All of these things can be accomplished by the same procedure, the use of a post build script that is launched by Buildroot after building the packages and before constructing the *rootfs*. Having into account that the files that finally will form the *rootfs* are located in the *"/output/target/"* folder, this script should copy files above in the proper directories and then allow Buildroot to construct the *rootfs*. Let's illustrate this with an example.

1. Write these lines into a text file and save it under the name "*my-de1soc-post-build.sh*" in the folder "*<buildrootbasedirectory>/boards/altera/de1-soc/*".

```
#!/bin/sh
# post-build.sh for DE1-SOC
# 2016, "Antonio Carpeño" <antonio.cruiz@upm.es>
# 2016, "Mariano Ruiz" <mariano.ruiz@upm.es>

# copy "profile" to <target>/etc/ to set new prompt
cp $BASE_DIR/../board/altera/de1-soc/system_config_files/profile
$BASE_DIR/target/etc/

# copy "sshd_config" to <target>/etc/ssh to allow ssh root access
cp $BASE_DIR/../board/altera/de1-soc/system_config_files/sshd_config
$BASE_DIR/target/etc/ssh/

# copy user space programs for testing blinker drivers modules
cp $BASE_DIR/../board/altera/de1-soc/user_applications/*
$BASE_DIR/target/root/
```

2. Execute "make xconfig" inside the Buildroot's base directory.
3. Go to "system configuration" section and fill the item "custom scripts to run before creating filesystem images" with the text "<pathtomyfolderwiththefile>/my-de1soc-post-build.sh"
4. Enter "make". The new *rootfs* will be generated and left in the "output/image/" folder as usual.

It is also possible to run a script after creating the *rootfs*. One of the usual tasks that are done in this script is the image generation. If you want to automatize the image generation, namely, to copy the spl-bsp files, to extract the *rootfs* into a directory, and to execute the script that eventually makes the image file; you can use the following script.

```
#!/bin/sh
```

```
# post-image.sh for DE1-SOC
# 2016, "Antonio Carpeño" <antonio.cruiz@upm.es>
# 2016, "Mariano Ruiz" <mariano.ruiz@upm.es>

# copy "spl-bsp, dtb and bitfile to image directory
cp $BASE_DIR/../board/altera/de1-soc/spl_bsp/* $BASE_DIR/images/
cp $BASE_DIR/../board/altera/de1-soc/SD_Image/* $BASE_DIR/images/

# extract rootfs and set root as owner

cd $BASE_DIR/images/
sudo rm -Rf rootfs
mkdir rootfs
sudo tar -xf rootfs.tar -C rootfs

sudo /usr/bin/python make_sdimage.py \
-f \
-P preloader-mkpimage.bin,u-boot.img,num=3,format=raw,size=10M,type=A2 \
-P rootfs/*,num=2,format=ext3,size=1200M \
-P zImage,u-boot.scr,soc_system.rbf,soc_system.dtb,num=1,format=vfat,size=300M \
-s 1700M  \
-n de1soc-sd-card-buildroot.img
```

Note that this script expects a structure of folders and files like the one shown below.

```
<buildrootbasedir>/boards/altera/de1-soc/
                                  /spl_bsp/   preloader-mkpimage.bin
                                              soc_system.dts
                                              soc_system.rbf

                                  /SD-Image/  make_sdimage.py
                                              u.boot.scr

                                  /system_config_files/   profile
                                                          sshd_config


    /user_applications/yourlistofexecutablefiles
```

To order Buildroot the execution of the script after the construction of the *rootfs* follow these steps. If you prefer to generate the image as usual then you should jump to step 40.

5. `Write` the script code into a text file and save it under the name "*my-de1soc-post-image.sh*" in the folder "*<buildrootbasedirectory>/boards/altera/de1-soc/*".
6. Execute "`make xconfig`" (or make menuconfig) inside the buildroot's base directory.
7. Go to "`system configuration`" section and fill the item "`custom scripts to run after creating filesystem images`" with the text "`$(TOPDIR)/board/altera/de1-soc/my-de1soc-post-image.sh`"
8. Enter "`make`". The new *rootfs* will be generated and left in the "`output/image/`" folder as usual.
9. If you chose not to use the previous script, you can do it manually. Thus, use the same *zimage, rootfs, uboot* and *uboot.src* that were used in chapter 7, and generate the SDcard as you already know.
10. Boot the de1-soc. If there not have been any problem, now you could see both the board six seven segments displays and the LEDs playing its familiar blink.

6.2.9 Controlling *blinker* from the HPS.

The final step is to control the *blinker* from software, that is, we will have to read from and write to the addresses of the *blinker* registers. The proper way to do this is by writing a device driver that sets up an interface between our *user space* code and the hardware. This allows us to avoid having to *mmap* "/dev/mem", which is not quite safe. As you will see, our driver will export a bunch of files in sysfs (the /sys filesystem) which we can, for example, write a number to and have that number set as the configuration or the speed value in our hardware. But we are doing this a little further. To carry on this first test of our hardware, it can be much less overhead to simply poke around the hardware through a user space application that *mmap* the hardware.

[/dev/mem]: The /dev/mem is a character device file that is an image of the main memory of the processor. It provides access to the physical memory in your Linux system. Opening this file and reading and writing can provide access to all the physical devices connected to the processor. This includes the RAM memory and also the peripherals. Incorrect access can produce unexpected behaviours in your system even a crash. Depending on the kernel version in use the access to specific memory ranges can be restricted.

[mmap]: The."mmap" system call (#include <sys/mman.h> creates a new mapping in the virtual address space of the calling process. When you provide a file descriptor in fd parameter, the mapping is implemented in this file. When you pass to mmap the fd obtained when opening /dev/mem device you obtain a virtual address in addr parameter. The offset parameter is used to know the starting point of the memory area to be maped. Calling mmap using the /dev/mem fd with an offset equivalent to the position of a peripheral in the memory map of the HPS allows getting a pointer in user space (this pointer points to a virtual memory address that is mapped to the physical location of the peripheral)

If we want to write a Linux user space application to interact with our FPGA hardware one of the first things we have to solve is how to bring the hardware architecture details into our software development environment. As you know, this peripheral is connected to the *lightweight HPS-to-FPGA* bridge. As has been said the lightweight bridge's region of memory begins at physical address 0xff200000, so to find the address of a *blinker* register, simply add the peripheral's base offset as shown by Qsys, plus the offset address of the particular register you want to access. For example, the *config* register was assigned the offset 0x1 and the offset of the base address of *blinker* as was assigned in Qsys is 0x00040000, so the physical address is simply 0xff240001. With this information and using mmap system call, we can develop our program. The problem is that we are hard-coding the addresses of the peripherals and this is very risky when you are developing embedded systems. To avoid the use of hard-coded addresses, we can use the header files available in Altera's SoC EDS tool. One part of these tools is oriented to the development of bare-metal applications. There are some header files that ease the access to the common resources belonging to the HPS. The *lightweight HPS-to-FPGA* bridge is one of these resources. In Fig. 105 there is the fragment of code of the file "hps.h". There you can see some definitions and macros related with the *lw_h2f_bridge*. The file is located, in this path: "<altera>/16.0/embedded/ip/altera/hps/altera_hps/hwlib/include/soc_cv_av/socal".

```
/*
 * Component Instance : lwfpgaslaves
 *
 * Instance lwfpgaslaves of component ALT_LWFPGASLVS.
 *
 *
 */
/* The base address byte offset for the start of the ALT_LWFPGASLVS component. */
#define ALT_LWFPGASLVS_OFST        0xff200000
/* The start address of the ALT_LWFPGASLVS component. */
#define ALT_LWFPGASLVS_ADDR        ALT_CAST(void *, (ALT_CAST(char *, ALT_HPS_ADDR) +
ALT_LWFPGASLVS_OFST))
/* The lower bound address range of the ALT_LWFPGASLVS component. */
#define ALT_LWFPGASLVS_LB_ADDR     ALT_LWFPGASLVS_ADDR
/* The upper bound address range of the ALT_LWFPGASLVS component. */
#define ALT_LWFPGASLVS_UB_ADDR     ALT_CAST(void *, ((ALT_CAST(char *, ALT_LWFPGASLVS_ADDR) +
0x200000) - 1))
```

Fig. 105: macro definition for the addresses of Light Weight HPS interface to FPGA.

But, this file does not include any information about the remaining custom elements you instantiated and connected to the *HPS* with Qsys. However, as you know the Altera Quartus, Qsys and SoC EDS tools provide various output files that can help us. These files are generated in the Quartus project directory as a result of Qsys system generation and Quartus compilation. The "*sopcinfo*" file was created by Qsys during the system generation and can be used to create the system header files and the device trees for the software environment. The "*hps_isw_handoff*" directory was created by Quartus during compilation and contains information about the HPS configuration. As you did in previous chapters, this collection of files is used by the BSP generator to create a customized preloader for the HPS core to run at boot time to configure the hardware.

You have already used the "sopc2dts" utility to create device trees representing the Qsys system. And now, using the "sopc-create-header-files" utility, it is possible to generate the header files from your SOPC Builder system description. To create the header files in cpp format (with a .h suffix) use the following command-line from a "SoC EDS command shell".

```
$ sopc-create-header-files \
"<pathtoyourproject>\soc_system.sopcinfo" \
--single hps_0.h \
--module hps_0
```

The *sopc-create-header-files* utility can create a memory map view from the perspective of any of the masters in the Qsys system. The output file "*hps_0.h*" is the header file created for the *lightweight HPS-to-FPGA master* view of the memory mapped system. The macros in these headers can be useful when writing applications and drivers that interact with peripherals connected to this bridge. Fig. 106 shows the portion of the code included in this file that is related with the *displays_ctrl* and *blinker* peripherals.

```
#ifndef _ALTERA_HPS_0_H_
#define _ALTERA_HPS_0_H_

/*
 * This file was automatically generated by the swinfo2header utility.
 *
 * Created from SOPC Builder system 'soc_system' in
 * file 'C:\altera_trn\DE1_SoC_Blinker\soc_system.sopcinfo'.
 */

/*
 * This file contains macros for module 'hps_0' and devices
 * connected to the following master:
 *   h2f_lw_axi_master
 *
 * Do not include this header file and another header file created for a
 * different module or master group at the same time.
 * Doing so may result in duplicate macro names.
 * Instead, use the system header file which has macros with unique names.
 */


/*
 * Macros for device 'displays_ctrl', class 'altera_avalon_pio'
 * The macros are prefixed with 'DISPLAYS_CTRL_'.
 * The prefix is the slave descriptor.
 */
#define DISPLAYS_CTRL_COMPONENT_TYPE altera_avalon_pio
#define DISPLAYS_CTRL_COMPONENT_NAME displays_ctrl
#define DISPLAYS_CTRL_BASE 0x30000
#define DISPLAYS_CTRL_SPAN 16
#define DISPLAYS_CTRL_END 0x3000f
#define DISPLAYS_CTRL_BIT_CLEARING_EDGE_REGISTER 0
#define DISPLAYS_CTRL_BIT_MODIFYING_OUTPUT_REGISTER 0
#define DISPLAYS_CTRL_CAPTURE 0
#define DISPLAYS_CTRL_DATA_WIDTH 6
#define DISPLAYS_CTRL_DO_TEST_BENCH_WIRING 0
#define DISPLAYS_CTRL_DRIVEN_SIM_VALUE 0
#define DISPLAYS_CTRL_EDGE_TYPE NONE
#define DISPLAYS_CTRL_FREQ 50000000
#define DISPLAYS_CTRL_HAS_IN 0
#define DISPLAYS_CTRL_HAS_OUT 1
#define DISPLAYS_CTRL_HAS_TRI 0
#define DISPLAYS_CTRL_IRQ_TYPE NONE
#define DISPLAYS_CTRL_RESET_VALUE 0

/*
 * Macros for device 'blinker_0', class 'blinker'
 * The macros are prefixed with 'BLINKER_0_'.
 * The prefix is the slave descriptor.
 */
#define BLINKER_0_COMPONENT_TYPE blinker
#define BLINKER_0_COMPONENT_NAME blinker_0
#define BLINKER_0_BASE 0x40000
#define BLINKER_0_SPAN 2
#define BLINKER_0_END 0x40001


#endif /* _ALTERA_HPS_0_H_ */
```

Fig. 106: Header file generated with sopc-create-header-files

With the previous information, we are ready to present an example of how to use the blinker peripheral. The example does not cover the whole set of feasible operations supported by its internal registers, in other words, not all the commands have been implemented. The implementation of the parsing of the command line is based on the function *getopt_long* responsible for parsing the string introduced by the user. The example uses a header file -blinker_devmem_test.h- where the relative internal address of the blinker registers have been defined, together with the “usage” and “help” string that shows the list of valid commands. The help string is particularly useful as far as it gives some clues about the whole set of commands that must be supported by the application. The completion of the code will be suggested a little later.

```
//blinker_devmem_test.h

// expected values for the blinker hardware features
#define BLINKER_SYSFS_ENTRY   "/sys/bus/platform/devices/ff240000.blinker"
#define BLINKER_PROCFS_ENTRY "/proc/device-tree/sopc@0/bridge@0xff200000/blinker@0x100040000"

// Offset address of the blinker registers
#define BLINKER_REG_SPEED_OFFSET      0x0
#define BLINKER_REG_CONFIG_OFFSET     0x1
#define BLINKER_REG_LEDS_OFFSET       0x0
#define BLINKER_REG_STATE_OFFSET      0x1


//
// usage string
//
#define USAGE_STR "\
\n\
Usage: blinker_devmem [ONE-OPTION-ONLY]\n\
  -d, --write_speed speed_value\n\
  -c, --write_config config_value\n\
  -s, --read_state\n\
  -l, --read_leds\n\
  -h, --help\n\
\n\
"

//
// help string
//
#define HELP_STR "\
\n\
Only one of the following options may be passed in per invocation:\n\
\n\
  -d, --write_speed speed_value\n\
Set the speed of blink\n\
15: max speed\n\
1:  min speed\n\
\n\
  -c, --write_config config_value\n\
Set the operation mode. Bit pattern: 000000ms.\n\
m: 0 (sweep)   1 (blink)\n\
s: 0 (stopped) 1 (running)\n\
\n\
  -s, --read_state\n\
Read the current configuration and speed.\n\
\n\
  -l, --read_leds\n\
Read the current leds position.\n\
\n\
  -h, --help\n\
Display this help message.\n\
\n\
```

Fig. 107: blinker_devmem_test.h file content

If you are not familiar with the function *getopt_long* you should review the information in this link [10]. Once familiarized with this procedure for parsing the command line, take some time for analysing the code. After the code, a brief explanation about the application is given, which can be useful for a better understanding.

```c
// blinker_devmem_test.c

#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <stdlib.h>
#include <unistd.h>
#include <getopt.h>
#include <error.h>
#include <dirent.h>
#include <string.h>
#include "hps.h"
#include "hps_0.h"
#include "blinker_devmem_test.h"

// command line parameters identification
void *command_write_config    = NULL;
void *command_read_state      = NULL;
void *command_help            = NULL;

// parameter value when applicable
unsigned char param_value;

// function prototypes
void validate_soc_system(void);
void parse_cmdline(int argc, char **argv);
void do_write_config(void *blinker_driver_map);
void do_read_state(void *blinker_driver_map);
void do_help(void);

int main(int argc, char **argv) {

        int devmem_filedesc;
        void *blinker_map;
        int result;

        // Validate the features of the actual hardware
        validate_soc_system();

        // parse the command line arguments
        parse_cmdline(argc, argv);

        // open the /dev/mem device
        devmem_filedesc = open("/dev/mem", O_RDWR | O_SYNC);
        if(devmem_filedesc < 0) {
                printf("devmem open");
                exit(EXIT_FAILURE);
        }

        // map the base of the blinker hardware
        blinker_map = mmap(NULL, sysconf(_SC_PAGE_SIZE), PROT_READ|PROT_WRITE, MAP_SHARED,
                                devmem_filedesc, ALT_LWFPGASLVS_OFST + BLINKER_0_BASE);
        if(blinker_map == MAP_FAILED) {
                printf("devmem mmap");
                close(devmem_filedesc);
                exit(EXIT_FAILURE);
        }

        // perform the operation selected by the command line arguments
        if(command_write_config       != NULL) do_write_config(blinker_map);
        if(command_read_state         != NULL) do_read_state(blinker_map);
        if(command_help               != NULL) do_help();

        // unmap the blinker and close the /dev/mem file descriptor
        result = munmap(blinker_map, sysconf(_SC_PAGE_SIZE));
        if(result < 0) {
                printf("devmem munmap");
                close(devmem_filedesc);
                exit(EXIT_FAILURE);
        }

        close(devmem_filedesc);
        exit(EXIT_SUCCESS);
}
```

```c
/*****************************************************************
 * validate_soc_system()
 * This function validate the features of the system hardware that we expect
 * to found. If something is wrong it is a clue that blinker may not be in the
 * system we are running, so we'd better don't touch anything.
 *****************************************************************/
void validate_soc_system(void) {
        const char *dirname;
        DIR *ds;

        // verify that the blinker device entry exists in the sysfs
        dirname = BLINKER_SYSFS_ENTRY;
        ds = opendir(dirname);
        if(ds == NULL) {
                printf("ERROR: blinker may not be present in the system hardware");
                exit(EXIT_FAILURE);
        }
        if(closedir(ds)) {
                printf("ERROR: blinker may not be present in the system hardware");
                exit(EXIT_FAILURE);
        }

        // verify that the blinker device entry exists in the procfs
        dirname = BLINKER_PROCFS_ENTRY;
        ds = opendir(dirname);
        if(ds == NULL) {
                printf("ERROR: blinker may not be present in the system hardware");
                exit(EXIT_FAILURE);
        }
        if(closedir(ds)) {
                printf("ERROR: blinker may not be present in the system hardware");
                exit(EXIT_FAILURE);
        }

        // verify that the blinker base address is page aligned
        if((ALT_LWFPGASLVS_OFST + BLINKER_0_BASE) & (sysconf(_SC_PAGE_SIZE) - 1)) {
                printf("ERROR: blinker base 0x%08X is not page aligned",
                                ALT_LWFPGASLVS_OFST + BLINKER_0_BASE);
                printf("Page size = 0x%08lX", (sysconf(_SC_PAGE_SIZE) - 1));
        }
}

/*****************************************************************
 * commands functions implementation
 *****************************************************************/

void do_write_config(void *blinker_driver_map){
        volatile unsigned char *blinker_config_reg =
                        blinker_driver_map + BLINKER_REG_CONFIG_OFFSET;

        *blinker_config_reg = param_value;
}

void do_read_state(void *blinker_driver_map){
        volatile unsigned char *blinker_state_reg =
                        blinker_driver_map + BLINKER_REG_STATE_OFFSET;

        unsigned char dato = *blinker_state_reg;
        printf("configuration = %u\n", dato & 0x0F);
        printf("speed = %u\n", (dato & 0xF0)>>4);
}

void do_help(void) {
        puts(HELP_STR);
        puts(USAGE_STR);
}
```

```c
/*****************************************************************
 * parse_cmdline()
 * This function reads the command line arguments, check they are correct
 * informs about any wrong use and marks the selected command
 *****************************************************************/

void parse_cmdline(int argc, char **argv) {
        int command;
        int opt_index = 0;
        int action_count = 0;

        static struct option long_options[] = {
                {"write_config",        required_argument,      NULL, 'c'},
                {"read_state",                  no_argument,            NULL, 's'},
                {"help",                        no_argument,            NULL, 'h'},
                {NULL, 0, NULL,         0}
        };

        // parse the command line arguments
        while(1) {
                command = getopt_long( argc, argv, "c:sh", long_options, &opt_index);

                if(command == -1)
                        break;
                switch(command) {
                case 0:
                        puts(USAGE_STR);
                        error(1, 0, "ERROR: wrong command line options.");
                        break;
                case 'c':
                        command_write_config = &command_write_config;
                        param_value = atoi(optarg);
                        break;
                case 's':
                        command_read_state = &command_read_state;
                        break;
                case 'h':
                        command_help = &command_help;
                        break;
                default:
                        puts(USAGE_STR);
                        error(1, 0, "ERROR: wrong command line options.");
                        break;
                }
        }

        // There was any extra characters on the command line. The program exit here
        if(optind < argc) {
                printf("extra characters on command line: ");
                while(optind < argc) printf("%s\n", argv[optind++]);
                puts(USAGE_STR);
                error(1, 0, "ERROR: wrong command line options");
        }

        // verify that the user only request one action to perform
        if(command_write_config       != NULL) action_count++;
        if(command_read_state         != NULL) action_count++;
        if(command_help               != NULL) action_count++;

        if(action_count == 0) {
                puts(USAGE_STR);
                error(1, 0, "ERROR: no options parsed");
        }

        if(action_count > 1) {
                puts(USAGE_STR);
                error(1, 0, "ERROR: too many options parsed");
        }
}
```

Fig. 108: User space application C source code accessing blinker peripheral

In the main function, we can see how our application starts off, beginning with the attempt to validate the hardware that is present in the system by calling the *validate_soc_system()* function. This function uses the run time *device tree models* to validate the system that we are running, in other words, try to find the entries in the *sysfs* and in the *procfs* which would be a hint of the existence of the peripheral *blinker* in the hardware system. If these entries are not found the most sensible thing to do is to exit the application, on the contrary, we may be touching the hardware in an uncontrolled manner. This function also tests the alignment of the blinker's base address to a page size (4096 bytes on Linux).When there is no doubt about the adequacy of the hardware system in which our application is running, the next thing to do is to parse the command line to figure out what operation the user wish to be carried out. Most of the code has for objective to control the possible error in the command line format.

Then, we open the */dev/mem* file and map the base addresses to our *blinker* hardware. The *mmap()* call maps a single page of memory beginning at 0xff240000 into the process's memory space. The first argument to *mmap()* is the virtual memory address we want the mapped memory to start at. By leaving it *null*, we allow the kernel to use the next memory address available. The second argument is the size of the region we want to map. The size will always be a multiple of the page size, so we specify the size of a single page (the minimum) even though we only need two bytes. Any error in one of these operations causes a sudden exit of the application.

The next thing to be done by the application code is to launch the function that implements the parsed command. As you see the readings and writings are carried out by means of pointers. These pointers are generated from the pointer returned by the *mmap*() function, pointing to the base address of our peripheral, plus the offset of the register involved in the operation. Notice that these pointers are declared with the *volatile* keyword. This tells the compiler that the value stored at this memory address can change without being written to from software. This disables certain compiler optimizations that can cause incorrect behavior.

Finally, when we are finished and before exiting the program, it is mandatory to *unmap* the resources and close the */dev/mem* file.

6.2.10 Exercise 4: compiling and debugging the application (optional)

Take a look to the HELP_STR in the "*blinker_devmem_test.h"* header file and you'll see that there are two commands not implemented by the software, namely: *read_leds* and *write_speed*. Create a folder to keep your project files and copy ll the files you need: *hps.h, hps_0.h, blinker_devmem_test.h and blinker_devmem_test.c*. Complete the code so that the whole set of commands have their associated function implementing the requested operation. Be aware that it may be needed changes in other functions as well.

To compile the application, it should be enough with the *Makefile* shown in Fig. 109, which you can use, with the necessary changes, to compile other applications you develop in the future.

```
C_SRC := blinker_devmem_test.c
CFLAGS := -g -O0 -Werror -Wall -I${SOCEDS_DEST_ROOT}/ip/altera/hps/altera_hps/hwlib/include
-I${SOCEDS_DEST_ROOT}/ip/altera/hps/altera_hps/hwlib/include/soc_cv_av -D soc_cv_av

# using Buildroot toolchain
CROSS_COMPILE := arm-buildroot-linux-gnueabihf-

CC := $(CROSS_COMPILE)gcc
NM := $(CROSS_COMPILE)nm

ifeq ($(or $(COMSPEC),$(ComSpec)),)
RM := rm -rf
else
RM := cs-rm -rf
endif

ELF ?= $(basename $(firstword $(C_SRC)))
OBJ := $(patsubst %.c,%.o,$(C_SRC))

.PHONY: all
all: $(ELF)

.PHONY:
clean:
	$(RM) $(ELF) $(OBJ) *.objdump *.map

$(OBJ): %.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

$(ELF): $(OBJ)
	$(CC) $(CFLAGS) $(OBJ) -o $@ $(LDFLAGS)
	$(NM) $@ > $@.map
```

Fig. 109: Makefile for the blinker user space application

The Makefile needs to know where are the cross compiler tools; this is accomplished including in the environment variable PATH which has to be set to point to the folder where the cross compiler for ARM architectures was installed. In a Linux bash terminal window, the environment variable can be modified for a particular session in the "*.bashrc*" file. This file use to be found in your home directory if you can not see it perhaps it is because you ca not see any hidden file. To show the hidden files, you can enter CTRL+H and then you should find it without further problems. So, edit this file to include the following line:

```
PATH=$PATH:/home/dte/Documents/buildroot-2016.08/output/host/usr/bin
```

Enter these commands to order the compilation and to copy the executable to the *home* folder in the target.

```
$ make
$ scp blinker_devmem_test root@<yourtargetdir>:/root/
```

Answer "*yes*" when you are prompted to continue and enter the password of the *root user* when asked.

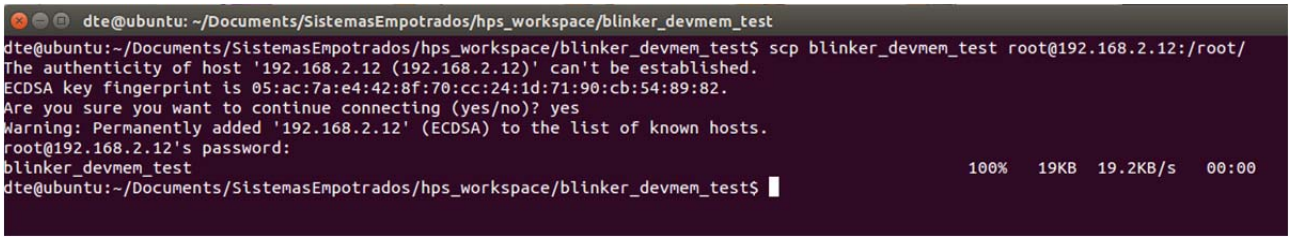


Fig. 110: Copying the file using the scp command.

Now you should test the blinker using the different commands to change the mode of operation, the speed of blinking, etc. Remember that you can interact with the blinker by the push buttons and the switches on the board as well. If something is wrong and you need to debug your code you could use *eclipse* to do it. It was described in chapter 8 how to create an Eclipse project from scratch. Here you will find how to create a project from existing code and makefile files. Open the Eclipse environment and select "`file -> import`"(see Fig. 111).

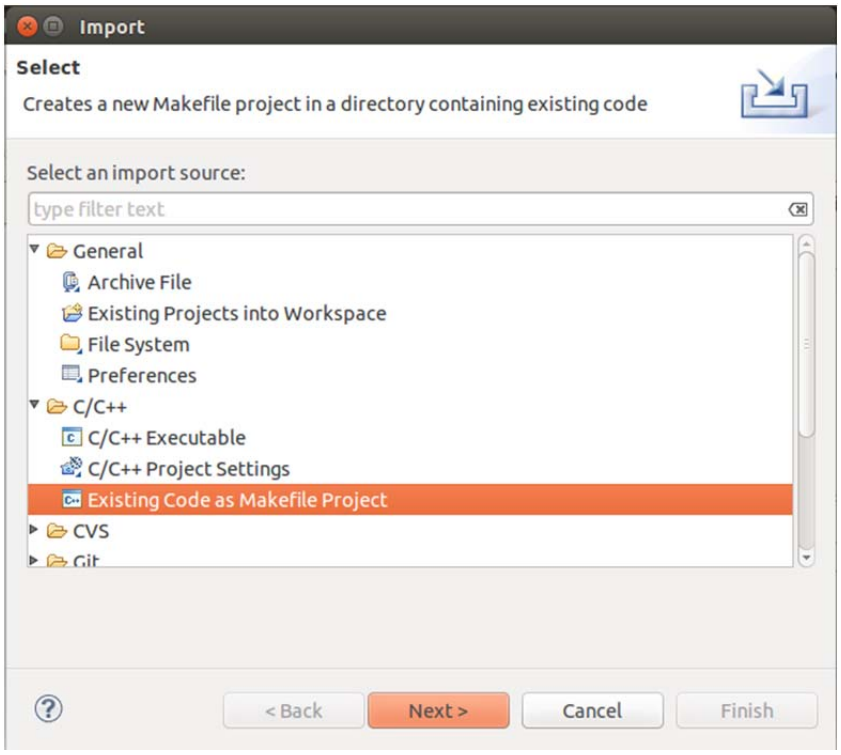


Fig. 111: Creating a new project using an existing Makefile

Select the option "`existing Code as Makefile Project`" in the `C/C++` section. Click "`Next`" and another window will appear (see Fig. 112).

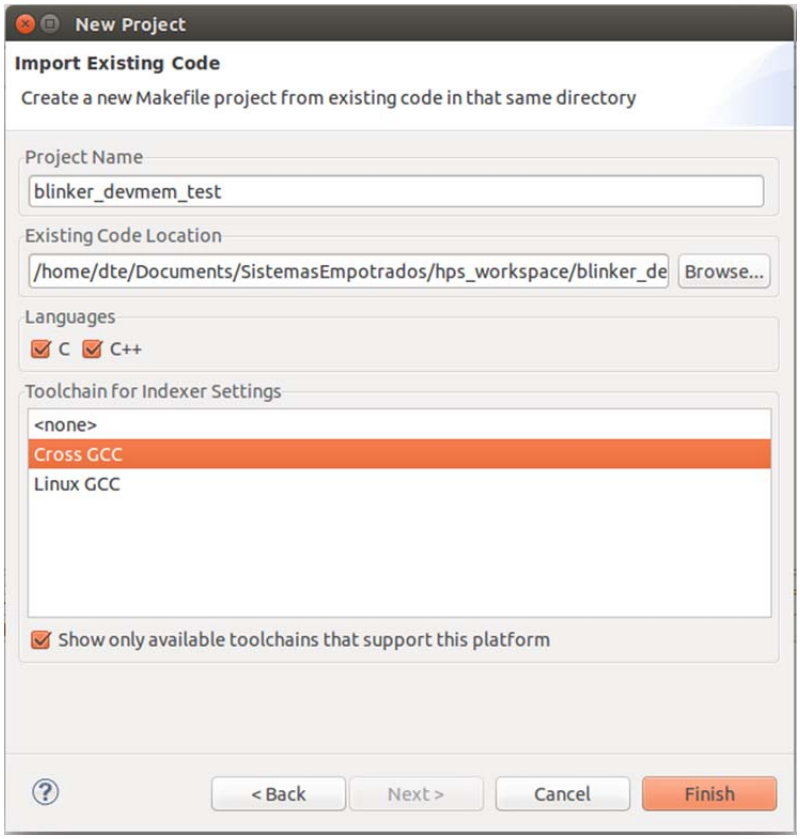


Fig. 112: Selecting the cross-compiling.

Write "*blinker_devmem_test*" in the "`Project name`" field and navigate to the folder's path with the "`Browse`" button. Select "`Cross GCC`" as Toolchain. Then select "`Finish`". The new project should be ready in the "`project explorer`" tab (see Fig. 113).

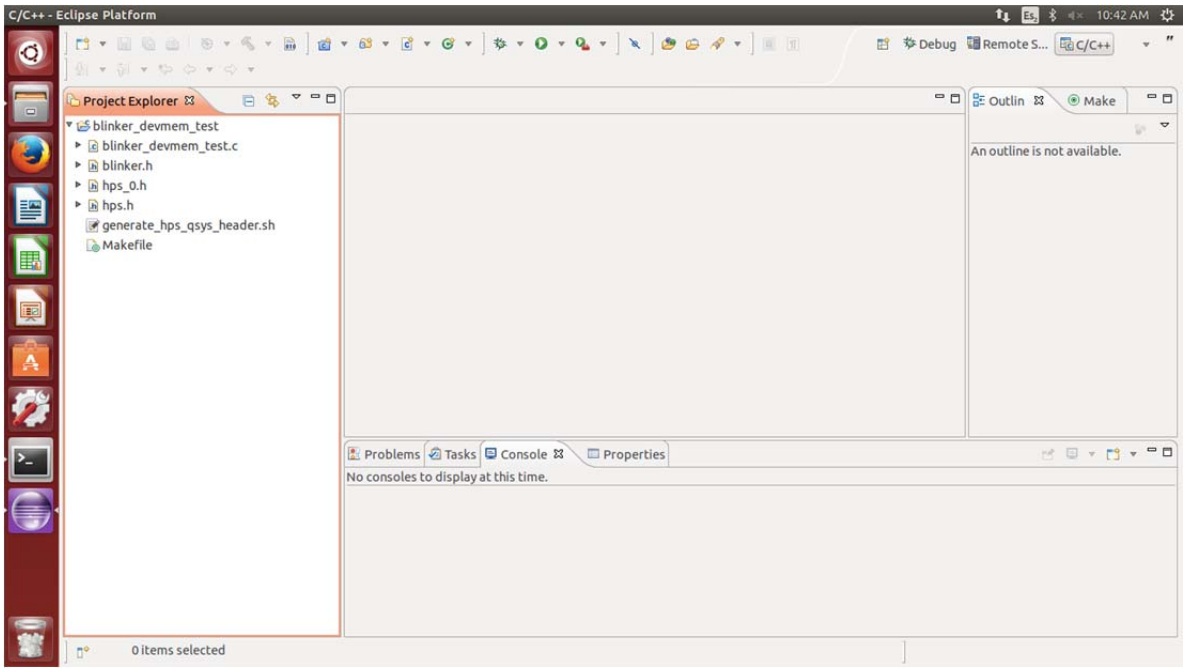


Fig. 113: Eclipse project using and existing Makefile.

Now you can execute your code step-by-step, watch the variables content, and debug your code as was described in Chapter 8.

7 KERNEL MODULES DEVELOPMENT

Using /dev/mem to map IO space into your user space application is a powerful resource for easily access the internal register of our peripheral. It is a quick and convenient thing to do when you need to access the IO space of your FPGA hardware avoiding the complex task of developing a proper device driver. However, this approach has some serious limitations, because:

- Mapping /dev/mem will not allow you to have access to kernel space memory or any other user space memory outside your own application's memory.
- Mapping /dev/mem will not allow you to register an interrupt handler or interact with hardware interrupts in any way.

To avoid these limitations and achieve overall control of your hardware with complete safety and being able to use whatever the input/output method, is mandatory to create a kernel module that gives whole access to user space applications in a secure way but with all the privileges of kernel processes.

The kernel image is a single file, resulting from the linking of all object files result of the compilation process of the source code that corresponds to features enabled in the configuration files. This file is loaded into memory by the bootloader at boot time. Therefore, all included features are available as soon as the kernel starts, at a time where no filesystem exists. This can be right for some resources of the OS (scheduling, memory management, task management, some hardware controllers, network, hard drive, USB driver, etc. However, some other features like device drivers, filesystems, etc., can be compiled as loadable modules. These are plugins that can be loaded/unloaded dynamically to add/remove features to the kernel. Each module is stored as a separate file in the filesystem, and therefore access to a filesystem is mandatory to use modules.

This principle of using removable modules makes it easy to develop drivers without having to reboot the system. The strategy is to load the module, test it, unload it, make changes and rebuild, load again, and so on. This is also useful for keeping the kernel image size to the minimum required by our application. This characteristic is essential in GNU/Linux embedded distributions. Another advantage is to reduce boot time, because you don't waste time initializing devices and kernel features that don't need right now.

There are several types of drivers, which present advantages and fallbacks depending on the requirements of our specific application. We are presenting in the next sections the steps to create the three most used types of driver in the Linux embedded field: platform-device drivers, miscellaneous drivers, and user space input/output drivers.

7.1 **Platform-Device Drivers.**

On embedded systems, devices that form part directly of the system-on-chip are not usually connected through a bus that allows hotplugging, and an automatic provision of unique identifiers. However, we want these devices to be part of the device model. Such devices must be statically described in either the kernel source code or in the Device Tree used on the newer architectures which is a hardware description file, instead of being dynamically detected.

The Linux kernel has a special bus, called the platform bus, specifically designed for controlling such devices. The kernel supports platform drivers for handling platform devices, through a bus that works like any other bus (USB, PCI, etc.), except that devices are enumerated statically instead of being discovered dynamically.

7.1.1 Developing the platform-device driver. *Blinker_pd_module*.

Now we will write a device driver that sets up an interface between our userspace code and the hardware. The main goal of this class of driver is to export a set of files in the /sys filesystem, that we can write or read to carry out actual writings and readings of the internal registers of our hardware. By these operations, we can set the speed or the configuration of the blinker peripheral, or read the actual state of the LEDs, the speed that has been configured and the established mode of operation. The code is presented in the next pages, and we will go through it bit by bit in the rest of this section. The code is not complete to offer a completely functional driver, so in exercise 5, you will be asked for writing the missing code, compile it and test it in the de1-soc board.

blinker_module.h

```
/* Common useful macros */
#define SYSTEM_BUS_WIDTH   (8)

#define __IO_CALC_ADDRESS_NATIVE(BASE, REGNUM)\
       ((void *)(((unsigned char*)BASE) + ((REGNUM) * (SYSTEM_BUS_WIDTH/8))))

#define IORD(BASE, REGNUM) \
       *(unsigned long*)(__IO_CALC_ADDRESS_NATIVE ((BASE), (REGNUM)))

#define IOWR(BASE, REGNUM, DATA) \
       *(unsigned long*)(__IO_CALC_ADDRESS_NATIVE ((BASE), (REGNUM))) = (DATA)


/* BLINKER peripheral parameters */
#define BLINKER_BASE       0xff240000
#define BLINKER_SIZE       PAGE_SIZE


/* SPEED register */
#define BLINKER_SPEED_REG   0

#define IOADDR_BLINKER_SPEED(base) \
       __IO_CALC_ADDRESS_NATIVE(base, BLINKER_SPEED_REG)

#define IOWR_BLINKER_SPEED(base, data)  IOWR(base, BLINKER_SPEED_REG, data)

#define BLINKER_SPEED_MSK               (0x0F)


/* CONFIG register */
#define BLINKER_CONFIG_REG                      1
#define IOADDR_BLINKER_CONFIG(base)\
       IO_CALC_ADDRESS_NATIVE(base, BLINKER_CONFIG_REG)

#define IOWR_BLINKER_CONFIG(base, data)        IOWR(base, BLINKER_CONFIG_REG, data)
#define BLINKER_CONFIG_MSK                     (0x07)
#define BLINKER_START_MSK                      (0x01)
#define BLINKER_STOP_MSK                       (0xFE)
#define BLINKER_BLINK_MSK                      (0x02)
#define BLINKER_SHIFT_MSK                      (0xFD)
#define BLINKER_IENA_MSK                       (0x04)
#define BLINKER_IDIS_MSK                       (0xFB)


/* LEDS register */
#define BLINKER_LEDS_REG   0

#define IOADDR_BLINKER_LEDS(base)\
       __IO_CALC_ADDRESS_NATIVE(base, BLINKER_LEDS_REG)

#define IORD_BLINKER_LEDS(base)         IORD(base, BLINKER_LEDS_REG)
#define BLINKER_LEDS_MSK                (0x0F)


/* STATE register */
#define BLINKER_STATE_REG  1
#define IOADDR_BLINKER_STATE(base)\
       __IO_CALC_ADDRESS_NATIVE(base, BLINKER_STATE_REG)

#define IORD_BLINKER_STATE(base)        IORD(base, BLINKER_STATE_REG)
#define BLINKER_STATE_MSK               (0xFF)
```

blinker_module.h (continuation)

```
/* ioctl values */
#define OUR_IOC_TYPE            (0xEE)
#define IOC_SET_SPEED           _IOW(OUR_IOC_TYPE, 0, u8)
#define IOC_SET_RUNNING         _IO(OUR_IOC_TYPE, 1)
#define IOC_CLEAR_RUNNING       _IO(OUR_IOC_TYPE, 2)
#define IOC_ENA_INT             _IO(OUR_IOC_TYPE, 6)
#define IOC_DIS_INT             _IO(OUR_IOC_TYPE, 7)
#define IOC_SET_MODE            _IOW(OUR_IOC_TYPE, 3, u8)
#define IOC_GET_LEDS            _IOR(OUR_IOC_TYPE, 4, u8)
#define IOC_GET_CONFIG          _IOR(OUR_IOC_TYPE, 5, u8)


/*
 * ioctl values
 *
#define IOC_SET_SPEED           (0x4001EE00)
#define IOC_SET_MODE            (0x4001EE03)
#define IOC_SET_RUNNING         (0x0000EE01)
#define IOC_CLEAR_RUNNING       (0x0000EE02)
#define IOC_ENA_INT             (0x0000EE06)
#define IOC_DIS_INT             (0x0000EE07)
#define IOC_GET_LEDS            (0x8001EE04)
#define IOC_GET_CONFIG          (0x8001EE05)
 */


/*To decode a hex IOCTL code:

Most architectures use this generic format, but check
include/ARCH/ioctl.h for specifics, e.g. powerpc
uses 3 bits to encode read/write and 13 bits for size.

 bits    meaning
 31-30 00 - no parameters: uses _IO macro
       10 - read: _IOR
       01 - write: _IOW
       11 - read/write: _IOWR

 29-16 size of arguments

 15-8  ascii character supposedly unique to each driver

 7-0   function #


So for example 0x82187201 is a read with arg length of 0x218,
character 'r' function 1.
*/
```

The content of the “blinker_module.h” header file gives some useful address calculation macros and masks for easily use the control and status registers of the blinker module. On behalf of simplicity, we decided to use only one header file for the whole set of drivers examples developed for the blinker. It has definitions and macros needed for all them. Therefore, you might not find useful this second part of the code for the platform-device driver, but we will need it further. Next, the “blinker_pd_module.c” code is presented.

blinker_pd_module.c

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/device.h>
#include <linux/platform_device.h>
#include <linux/uaccess.h>
#include <linux/ioport.h>
#include <linux/io.h>
#include <linux/clk.h>

#include "blinker_module.h"

/* global variables */
static struct platform_driver blinker_platform_driver;
static void *blinker_map_mem;
static int g_blinker_driver_base_addr;
static int g_blinker_driver_size;
static unsigned long g_blinker_driver_clk_rate;


/*****************************************************************************
             SHOW AND STORE METHODS
*****************************************************************************/


ssize_t leds_show(struct device_driver *drv, char *buf)
{
       u8 data;

       data = ioread8(IOADDR_BLINKER_LEDS(blinker_map_mem));

       pr_info("Leds position: %d\n", data);

       return scnprintf(buf, PAGE_SIZE,"%u\n", data);
}


ssize_t config_store(struct device_driver *drv, const char *buf, size_t count)
{
       u8 data;

       if (buf == NULL) {
              pr_err("Error, string must not be NULL\n");
              return -EINVAL;
       }

       if (kstrtou8(buf, 10, &data) < 0) {
              pr_err("Could not convert string to integer\n");
              return -EINVAL;
       }

       if (data > 3 && data < 1) {
              pr_err("Invalid configuration data %d\n", data);
              return -EINVAL;
       }

       iowrite8(data, IOADDR_BLINKER_CONFIG(blinker_map_mem));

       pr_info("New configuration value: %d\n", data);

       return count;
}


static DRIVER_ATTR(config, S_IWUGO, NULL, config_store);
static DRIVER_ATTR(leds, S_IRUGO, leds_show, NULL);
```

blinker_pd_module.c (continuation)

```
/*****************************************************************************
            PROBE, REMOVE, RELEASE, INIT AND EXIT METHODS
*****************************************************************************/

/*
 * Array of of_device_id's specifying ".compatible" string for binding this driver to
 * any compatible device in the device tree when this module is inserted.
 * This driver "probe" method will be automatically triggered.
 */

static struct of_device_id blinker_driver_dt_ids[] = {
       {
        .compatible = "blinker,driver-1.0"},
       { /* end of table */ }
};

MODULE_DEVICE_TABLE(of, blinker_driver_dt_ids);



static int blinker_probe(struct platform_device *pdev)
{
       int ret = -EINVAL;
       struct resource *res;
       struct resource *blinker_driver_mem_region;
       struct clk *clk;
       unsigned long clk_rate;

       /* get the memory region allocated by blinker device */
       res = platform_get_resource(pdev, IORESOURCE_MEM, 0);

       if (res == NULL) {
              pr_err("IORESOURCE_MEM, 0 does not exist\n");
              goto bad_exit_return;
       }

       g_blinker_driver_base_addr = res->start;
       g_blinker_driver_size = resource_size(res);


       /* get clock resource */
       clk = clk_get(&pdev->dev, NULL);
       if (IS_ERR(clk)) {
              pr_err("clk not available\n");
              goto bad_exit_return;
       } else {
              clk_rate = clk_get_rate(clk);
       }

       g_blinker_driver_clk_rate = clk_rate;

       /* Create sysfiles entries */
       ret = driver_create_file(&blinker_platform_driver.driver, &driver_attr_leds);
       if (ret < 0) {
              pr_err("failed to create leds sysf entry.");
              goto bad_exit_create_leds_file;
       }


       /*****************************************************************/
       /* WRITE HERE THE CODE TO CREATE THE REST OF THE SYSFILES ENTRIES       */
       /* TO WRITE AND READ THE SPEED, CONFIG, STATE REGISTERS          */
       /*****************************************************************/
```

blinker_pd_module.c (continuation)

```
	/*
	 * reserve a memory region and remap it into an IO pointer
	 * for the blinker driver
	 */

	blinker_driver_mem_region =
		request_mem_region(g_blinker_driver_base_addr,
				   g_blinker_driver_size, "blinker");

	if (blinker_driver_mem_region == NULL) {
		pr_err("request_mem_region failed.");
		goto bad_exit_request_mem;
		ret = -EBUSY;
	}

	blinker_map_mem = ioremap(g_blinker_driver_base_addr, g_blinker_driver_size);


	if (blinker_map_mem == NULL) {
		pr_err("ioremap failed.");
		goto bad_exit_ioremap;
		ret = -EFAULT;
	}

	/* All the operations were successfully develop */
	pr_info("blinker device successfully connected\n");
	return 0;


	/* Some kind of error occurred during the process of the module insertion */

bad_exit_ioremap:
	release_mem_region(g_blinker_driver_base_addr, g_blinker_driver_size);

bad_exit_request_mem:
	driver_remove_file(&blinker_platform_driver.driver, &driver_attr_state);

bad_exit_create_state_file:
	driver_remove_file(&blinker_platform_driver.driver, &driver_attr_config);

bad_exit_create_config_file:
	driver_remove_file(&blinker_platform_driver.driver, &driver_attr_leds);

bad_exit_create_leds_file:
	driver_remove_file(&blinker_platform_driver.driver, &driver_attr_speed);

bad_exit_return:

	pr_err("blinker device connect FAILED");
	return ret;
}
```

blinker_pd_module.c (continuation)

```
static int blinker_remove(struct platform_device *pdev)
{
	/* Sysfiles entries removal */
	driver_remove_file(&blinker_platform_driver.driver, &driver_attr_speed);
	driver_remove_file(&blinker_platform_driver.driver, &driver_attr_leds);
	driver_remove_file(&blinker_platform_driver.driver, &driver_attr_config);
	driver_remove_file(&blinker_platform_driver.driver, &driver_attr_state);

	/* Deallocate resources */
	release_mem_region(g_blinker_driver_base_addr, g_blinker_driver_size);
	iounmap(blinker_map_mem);

	/* Removal operations successfully done */
	pr_info("blinker device successfully removed\n");

	return 0;
}

static struct platform_driver blinker_platform_driver = {
	.probe = blinker_probe,
	.remove = blinker_remove,
	.driver = {
		  .name = "blinker",
		  .owner = THIS_MODULE,
		  .of_match_table = blinker_driver_dt_ids,
		  },
	/* .shutdown = unused,
	   .suspend = unused,
	   .resume = unused,
	   .id_table = unused,
	 */
};

static int __init blinker_init(void)
{
	int ret;
	g_blinker_driver_base_addr = BLINKER_BASE;
	g_blinker_driver_size = BLINKER_SIZE;

	ret = platform_driver_register(&blinker_platform_driver);
	if (ret != 0) {
		pr_err("platform_driver_register returned %d\n", ret);
		goto bad_exit_driver_register;
	}

	pr_info("blinker module successfully inserted\n");
	return 0;

bad_exit_driver_register:
	pr_err("blinker module insert FAILED");
	return ret;
}

static void __exit blinker_exit(void)
{
	platform_driver_unregister(&blinker_platform_driver);

	pr_info("blinker driver successfully removed\n");
}

module_init(blinker_init);
module_exit(blinker_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Antonio Carpeño <antonio.cruiz@upm.es>");
MODULE_AUTHOR("Mariano Ruiz <mariano.ruiz@upm.es>");
MODULE_DESCRIPTION("blinker peripheral platform_driver example");
MODULE_VERSION("1.0");
```

As platform devices cannot be detected dynamically, they are defined statically by using a device tree source (.dts). It is a tree of nodes that models the hierarchy of devices in the system on chip. Each node can have some properties describing various properties of the devices: addresses, interrupts, clocks, etc. At boot time, the kernel is given a compiled version, the Device Tree Blob (.dtb), which is parsed to instantiate all the devices described in the DT.

Here is an extract of the soc_system.dts file created in previous chapters. Namely, the description of the node corresponding to the blinker peripheral

```
blinker_0: blinker@0x100040000 {
                        compatible = "blinker,driver-1.0";
                        reg = <0x00000001 0x00040000 0x00000002>;
                        interrupt-parent = <&hps_0_arm_gic_0>;
                        interrupts = <0 41 4>;
                        clocks = <&clk_0>;
                } ; //end blinker@0x100040000 (blinker_0)
```

- blinker@0x100040000 is the node name
- blinker_0 is a label, that can be referred to in other parts of the Device Tree as &blinker_0
- A device is bound to the corresponding driver using the “compatible” string.
- The rest of the lines are properties. Their values are usually strings, a list of integers, or references to other nodes.

From the information included in the blinker node of the device tree, “platform_device structures” are created. This structure defines the set of devices that the “blinker, driver-1.0” can manage. Take notice of the matching between the “compatible” string in both parts of the code.

```
static struct of_device_id blinker_driver_dt_ids[] = {
       {
        .compatible = "blinker,driver-1.0"},
       { /* end of table */ }
};
```

On the other hand, the MODULE_DEVICE_TABLE() macro allows “depmod” to extract at compile time the relation between device identifiers and drivers, so that drivers can be loaded automatically by “udev”.

```
MODULE_DEVICE_TABLE(of, blinker_driver_dt_ids);
```

The “platform-driver structure” inherits from the “device_driver structure”, which is defined by the device model. When the driver is loaded or unloaded, it must register or unregister itself from the kernel core, using `platform_driver_register` () and platform_driver_unregister() functions.

```
static struct platform_driver blinker_platform_driver = {
       .probe = blinker_probe,
       .remove = blinker_remove,
       .driver = {
                 .name = "blinker",
                 .owner = THIS_MODULE,
                 .of_match_table = blinker_driver_dt_ids,
                 },
};
```

The initialization function “blinker_init” is called when the module is loaded, and returns an error code: 0 on success, or a negative value on failure. Declared by the module_init() macro, the name of the function doesn't matter, even though <modulename>_init() is a convention. The cleanup function “blinker_exit” is called when the module is unloaded, and is declared by the module_exit() macro. These functions are responsible for registering and unregistering the device to the platform driver infrastructure.

```
static int __init blinker_init(void)
{
   ---------
   ---------
}

static void __exit blinker_exit(void)
{
   ---------
   ---------
}

module_init(blinker_init);
module_exit(blinker_exit);
```

Metadata information declared using several macros. This information can be retrieved by using the “modinfo” command.

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Antonio Carpeño <antonio.cruiz@upm.es>");
MODULE_AUTHOR("Mariano Ruiz <mariano.ruiz@upm.es>");
MODULE_DESCRIPTION("blinker peripheral platform_driver example");
MODULE_VERSION("1.0");
```

The “platform-driver structure” has two fields to set pointers to both useful methods. The blinker_probe() method receives as argument a structure describing the device, usually specialized by the platform bus infrastructure. This function is responsible for initializing the device, mapping I/O memory, registering the interrupt handlers. The bus infrastructure provides methods to get the addresses, interrupt numbers and other device-specific information. The available resources list will be built up by the kernel at boot time from the device tree so that you do not need to make any unnecessary lookups to the Device Tree when loading your driver. The blinker_remove() method is in charge of liberating any resource allocated by the probe() method.

```
static int blinker_probe(struct platform_device *pdev)
{
   ---------
   ---------
}


static int blinker_remove(struct platform_device *pdev)
{
   ---------
   ---------
}
```

Linux subscribes to the philosophy of “everything is a file”. That is, the standard way for userspace to communicate with drivers is through file IO operations. For reading and writing data to driver modules, the Linux kernel provides a filesystem called sysfs. sysfs is usually mounted in /sys, /sys/bus/ contains the list of buses, /sys/devices/ contains the list of devices, /sys/class enumerates devices by class. The way to create these sysfs is by calling the function:

```
driver_create_file(&blinker_platform_driver.driver, &driver_attr_leds);
```

You can create as sysfs entries as needed and, what is more useful, to link *show* and *store* methods which will be launched when writing or reading to or from these files, by a user space application or by command execution on a shell.

```
ssize_t leds_show(struct device_driver *drv, char *buf)
{
   ---------
   ---------
}

ssize_t config_store(struct device_driver *drv, const char *buf, size_t count)
{
   ---------
   ---------
}

static DRIVER_ATTR(config, S_IWUGO, NULL, config_store);
static DRIVER_ATTR(leds, S_IRUGO, leds_show, NULL);
```

7.1.2 Compiling the platform-device driver.

To compile the driver code it is necessary to use a cross compiler toolchain because the binary will be run on an ARM processor. Moreover, it is mandatory to compile the driver against the libraries and the headers of exactly the same version of the kernel where the modules will be loaded. This configuration is achieved by setting these variables as you can see in the first lines of the makefile.

```
ARCH=arm
CROSS_COMPILE=/home/dte/Documents/buildroot-2016.08/output/host/usr/bin/
 arm-buildroot-linux-gnueabihf-
OUT_DIR=/home/dte/Documents/buildroot-2016.08/output/build/linux-
ACDS16.0_REL_GSRD_PR
```

As you can see, we are using the same kernel sources and the same cross-compiling tools used by Buildroot to generate the kernel binaries, which we can find in the “/buildroot-2016.08/output/” directory.

The different targets in the make file allow compiling the source code of the driver with slightly different results. You can compile with the following commands executed inside the driver directory:

```
$ make
```

Then, you will get your “kernel object” file (.ko) which you can send to the de1-soc board (with “scp” command for example) and load them once the kernel is running with the “modprobe” command in a shell.

However, maybe you want to create an SD image with the .ko file already placed in the proper directory. And get your driver loaded into the kernel automatically at boot time, with “depmod”. This can be achieved by the command:

```
$ make modules_install
```

The .ko file will be copied to the target folder "/lib/modules/3.10.31-ltsi/kernel/drivers/extra", inside the directory specified by this variable:

```
DESTDIR = /home/dte/Documents/buildroot-2016.08/output/target
```

Besides, all the necessary dependencies will be included in /lib/modules/3.10.31-ltsi/kernel/modules.dep and /modules.dep.bin.

Then in the Buildroot base directory, you must execute this command, to generate the new "rootfs".

```
$ make
```

To uninstall a module you must delete the .ko file in the driver sources directory and execute "make modules_install" again.

You can also clean-up all the generated files, and force re-compilation, by using:

```
$ make clean
```

Here the makefile and kbuild complete contents are showed.

Makefile

```
ARCH=arm
CROSS_COMPILE=/home/dte/Documents/buildroot-2016.08/output/host/usr/bin/
              arm-buildroot-linux-gnueabihf-
OUT_DIR=/home/dte/Documents/buildroot-2016.08/output/build/linux-ACDS16.0_REL_GSRD_PR
DESTDIR = /home/dte/Documents/buildroot-2016.08/output/target
LINUX_VARIABLES = PATH=$(PATH)
LINUX_VARIABLES += ARCH=$(ARCH)
ifneq ("$(KBUILD_BUILD_VERSION)","")
       LINUX_VARIABLES += KBUILD_BUILD_VERSION="$(KBUILD_BUILD_VERSION)"
endif
LINUX_VARIABLES += CROSS_COMPILE=$(CROSS_COMPILE)
ifneq ("$(DEVICETREE_SRC)","")
       LINUX_VARIABLES += CONFIG_DTB_SOURCE=$(DEVICETREE_SRC)
endif
LINUX_VARIABLES += INSTALL_MOD_PATH=$(INSTALL_MOD_PATH)


ifndef OUT_DIR
    $(error OUT_DIR is undefined, bad environment, you point OUT_DIR to the linux/
       kernel build output directory)
endif

KDIR ?= $(OUT_DIR)

default:
       $(MAKE) -C $(KDIR) $(LINUX_VARIABLES) M=$(CURDIR)

all:
       $(MAKE) -C $(KDIR) $(LINUX_VARIABLES) M=$(CURDIR)

clean:
       $(MAKE) -C $(KDIR) $(LINUX_VARIABLES) M=$(CURDIR) clean

help:
       $(MAKE) -C $(KDIR) $(LINUX_VARIABLES) M=$(CURDIR) help

modules:
       $(MAKE) -C $(KDIR) $(LINUX_VARIABLES) M=$(CURDIR) modules

modules_install:
       $(MAKE) -C $(KDIR) $(LINUX_VARIABLES) M=$(CURDIR) INSTALL_MOD_PATH=$(DESTDIR)/
        modules_install
```

Kbuild

```
       obj-m  := blinker_pd_module.o
```

7.1.3 Testing *blinker_pd_module* from a Linux terminal

Once the driver has been compiled and copied to the DE1-SoC the first thing we must do is to load it into the kernel. There are two possible commands to do this. One is the "insmod" command; the other is the "modprobe" command. We will choose this second option because, while the first tries to load only the module indicated by the command line, the second tries to load all the modules the given module depends on, and then this module. Below these lines, the command entered, and the response you should receive are shown.

```
$ cd /lib/modules/3.10.31-ltsi/kernel/drivers/platform/
$ ls
blinker_pd_module.ko
$ modprobe blinker_pd_module
blinker device successfully connected
blinker module successfully inserted
$
```

After the successful load of the driver, the sysfs entry files should have been created. To verify this you can enter de following commands in the terminal.

```
$ cd /sys/bus/platform/drivers/blinker
$ ls -l
```

If there was not anything wrong, you should see the output in the terminal as shown in Fig. 114. The file names match the names used when these files were created in the probe() method, as expected.

Fig. 114: content of the "blinker" folder

Now, to communicate with the driver and change its behaviour you only have to write (echo commands) or read (cat commands) to or from the corresponding file. Here you have some examples. Try them and look the response received in the terminal and the changes observed in the blinker device.

```
$ echo 15 > speed
$ echo 1 > speed
$ echo 3 > config
$ echo 2 > config
$ cat leds
$ cat state
```

Reading from LEDs with "cat" command should reflect "1" in that positions where the led are on.

The messages sent (logged) by the blinker driver, as any other Kernel log messages are displayed through the dmesg command (diagnostic message).

Once finished the test you can unload the driver by two possible ways: rmmod <module_name> which tries to remove the given module; or modprobe -r <module_name>, which tries to remove the given module and all dependent modules. This time we suggest to use the later option. Below you can see the commands and the response offered by the blinker driver

```
$ rmmod blinker_pd_module.ko
blinker device successfully removed
blinker driver successfully removed
$
```

7.1.4 Exercise 5: blinker platform driver development and test

As it has been mentioned before the driver source code is not complete, therefore it is necessary to write the missing code. This is the goal of this exercise, to finish the "blinker_pd_module.c", compile it using the given makefile and test the results by command execution in a terminal, as described in the previous section. Please, follow the following steps:

a) Write the sentences to create the sysfs entries for the speed, config and state registers.
b) Write the show and store methods for controlling the speed and state registers. Do not forget to register these methods with the appropriate macros.
c) Compile the source code using your favorite method.
d) Test the .ko file in the DE1-SoC board.
e) Repeat this process until the correct functioning of the driver.

7.1.5 Adding *blinker_pd_module* to the kernel as a loadable module

After completion of exercise 5 we are sure of the correct operation of our blinker driver. It will be very useful to have the .ko file included in the SD image. However, if we want our driver to be automatically compiled by Buildroot anytime we create a new image we have to add the new driver to the kernel sources. To achieve this it is necessary to carry out the following operations.

1. Add your new source file to the appropriate source directory. In this case: "../kernel/drivers/platform/"

2. Describe the configuration interface for your new driver by adding the following lines to the Kconfig file in this directory:

```
config BLINKER_PLATFORM
     tristate "blinker platform module"
     default n
     help
       Select this configuration to enable the blinker_platform module
       Don't select as built-in together with blinker_uio or blinker_misc.
       Only one of then can be inserted at one time
```

3. Add this line in the Makefile file based on the Kconfig setting:

```
obj-$(CONFIG_BLINKER_PLATFORM)     += blinker_pd_module.o
```

4. Now you must set the appropriate configuration options to compile the kernel. Enter the following command in Buildroot base directory.

```
$ make linux-xconfig
```

5. Navigate to Device Drivers and select the “blinker driver module” as shown in the Fig. 115 and Fig. 116 to enable its compilation as a loadable module, if you select the “check mark” then the module will be included as part of the kernel and it won’t be necessary to load it with the modprobe command.

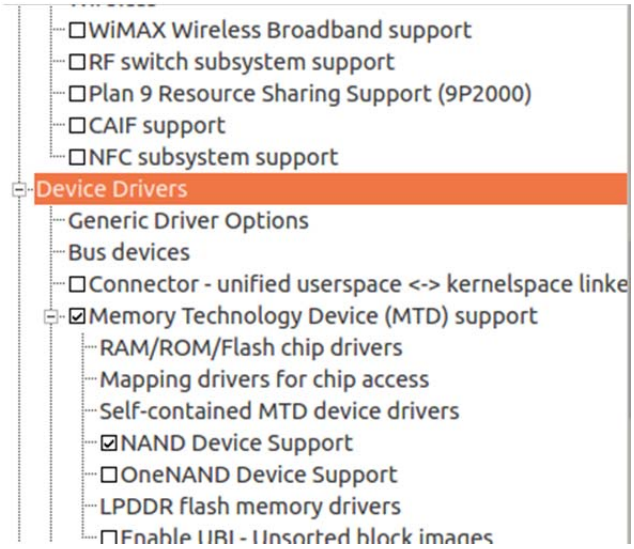


Fig. 115: Selecting a custom device driver.

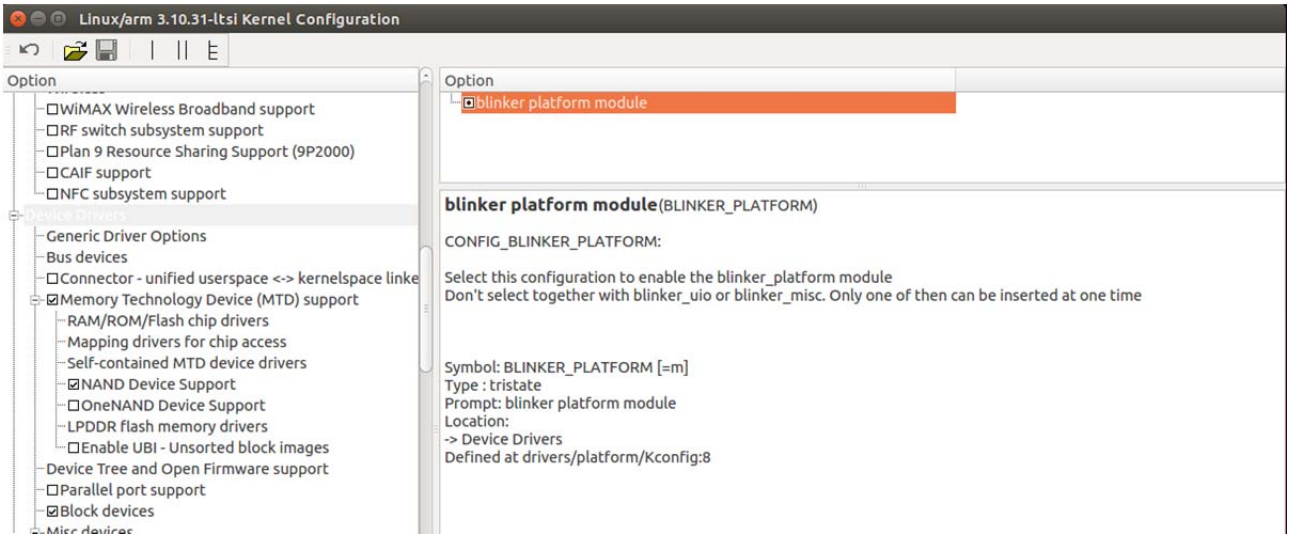


Fig. 116: Kernel configuration.

6. To order Buildroot to compile only the kernel without recompiling the rest of the source files you must enter this command.

```
$ make linux-rebuild
```

After the execution Buildroot will have generated the new kernel binaries “zImage”, the new .dtb file with the device tree blob, and the blinker driver module, in the /lib/modules/3.10.31-ltsi/kernel/drivers/platform/ directory as a .ko file in the /output/target directory in the Buildroot tree.

7. To create the new rootfs which include our new driver and create the new SD image you must execute this command.

```
$ make
```

7.1.6 Testing *blinker_pd_module* from a user space application

We have verified the correct operation of our driver directly writing and reading the sys entry files on a shell, but the most usual thing is to interact with the driver from a user space application. We are coping now with this topic creating a “blinker_pd_module_test” application. The code resembles very closely the one we develop in section 7.2.9.

First of all, we have to make some changes in the header file, introducing the following lines, which defines the path to the sysfs entries for the different internal registers.

blinker_pd_module_test.h

```
      // SYSFILES Entries for the blinker platform driver control
#define SYSFS_SPEED_FILE "/sys/bus/platform/drivers/blinker/speed"
#define SYSFS_CONFIG_FILE "/sys/bus/platform/drivers/blinker/config"
#define SYSFS_LEDS_FILE "/sys/bus/platform/drivers/blinker/leds"
#define SYSFS_STATE_FILE "/sys/bus/platform/drivers/blinker/state"
```

After this, we can interact with these files using the common “open”, “write”, “read” and “close” system calls, as shown in the excerpts of code shown below. They present an example of code writing in the “config” register and reading an example from the “leds” register. These interactions launch the “leds_show” and “config_store” methods of the blinker driver, in the same way that the “echo” and “cat” commands did before. In exercise 6 you will have to write the two methods that are left, that is, the “state_show” and the “speed_store” methods.

blinker_pd_module_test.c

```
int do_write_config(void){
	int f;
	int count=0;

	f = open(SYSFS_CONFIG_FILE, O_WRONLY);
	if (f == -1) {
		perror("Error opening config SYSF.
			Verify that blinker driver is correctly inserted");
		return -1;
	}

	count = write(f, param_value, strlen(param_value));
	if (count == -1) {
		perror("Error writing in config SYSF.
			Verify that blinker is installed");
		return -1;
	}
	close(f);
	if (f == -1) {
		perror("Error closing config SYSF.
			Verify that blinker driver correctly inserted");
		return -1;
	}
	return 0;
}


int do_read_leds(void){
	int f;
	int count=0;
	char buffer[16];
	size_t offset = 0;
	size_t bytes_read;

	f = open(SYSFS_LEDS_FILE, O_RDONLY);
	if (f == -1) {
		perror("Error opening leds SYSF.
			Verify that blinker driver is correctly inserted");
		return -1;
	}

	/* Read from the file until reading less than we asked for.
	   This indicates that we've reached the end of the file.  */
	do {
		/* Read the next line's size of bytes.  */
		bytes_read = read (f, buffer, sizeof (buffer));

		for (count = 0; count < bytes_read; ++count)
			/* Keep count of our position in the file.  */
			offset += bytes_read;
		} while (bytes_read == sizeof (buffer));

	if (bytes_read == -1) {
		perror("Error reading from leds SYSF.
			Verify that blinker is correctly installed");
		return -1;
	}

	close(f);
	if (f == -1) {
		perror("Error closing leds SYSF.
			Verify that blinker driver is inserted");
		return -1;
	}
	printf ("Leds position: %s", buffer);
	return 0;
}
```

7.1.7 Exercise 6: accessing the driver from a user space application

Following the same strategy that was used in section 7.2.9 you have to write an application that implements these options:

```
#define USAGE_STR "\
\n\
Usage: blinker_pd_module_test [ONE-OPTION-ONLY]\n\
  -d, --write_speed speed_value\n\
  -c, --write_config config_value\n\
  -s, --read_state\n\
  -l, --read_leds\n\
  -h, --help\n\
\n\"
```

It should be helpful to start from the skeleton of the “blinker_devmem_test.c” and “blinker_devmem_test.h” and make the pertinent modifications. Create an eclipse project to compile and test the application.

7.2 Miscellaneous Character Drivers.

Apart from the platform device drivers that we have been working on the previous section, in Linux there are three types of devices:

- Network devices, represented as network interfaces.
- Block devices, which are used with raw storage devices the kind of hard disks, USB flash memories, etc.
- Character devices, used to provide user space applications access to all other types of devices.

The Linux philosophy of representing most system objects as files allows applications to manipulate all system objects with the normal file API (open, read, write, close, etc.). This is the reason why devices are represented as files to the applications, which are by convention stored in the /dev directory. These files are called device files, a special type of file that associates a file name visible to user space applications to the triplet (type, major, minor) that the kernel understands. The Linux kernel identifies all block and character devices using a major and a minor number. The major number indicates the family of the device and the minor number the number of the device (when there are several devices of the same kind, serial ports, for example). Most major and minor numbers are statically allocated and identical across all Linux systems.

However, many device drivers are not implemented directly as character drivers, they are implemented under a framework, specific to a given device type (framebuffer, V4L, serial, etc.). The strategy of using frameworks allows to reuse the common parts of drivers for the same type of devices, and offer a consistent API to user space applications. Although, the user space applications still see them as character devices. The framework allows providing a coherent user space interface (read, write, ioctl, etc.) for every type of device, regardless of the driver specific characteristics.

The kernel offers a large number of frameworks covering a wide range of device types: input, network, video, audio, etc. However, some devices really do not fit in any of the existing frameworks. This is the case of highly customized devices implemented in an FPGA, or other weird devices for which implementing a complete framework is not useful. The drivers for such devices could be implemented directly as raw character drivers. However, a subsystem makes this work a little bit easier, the miscellaneous subsystem or misc drivers. Really, it is only a thin layer on top of the character driver API.

7.2.1 Developing the Misc driver. *Blinker_misc_module*.

This is the code that comprises the misc driver version for the blinker peripheral. This code is also available in the course git.

blinker_misc_module.c

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/device.h>
#include <linux/platform_device.h>
#include <linux/uaccess.h>
#include <linux/ioport.h>
#include <linux/io.h>
#include <linux/clk.h>
#include <linux/miscdevice.h>
#include <linux/fs.h>
#include <linux/ioctl.h>
#include <linux/interrupt.h>
#include <linux/wait.h>
#include <linux/sched.h>
#include "blinker_module.h"

/* global variables */
static struct semaphore g_dev_probe_sem;
static int g_platform_probe_flag;
static void *blinker_map_mem;
static int g_blinker_driver_base_addr;
static int g_blinker_driver_size;
static unsigned long g_blinker_driver_clk_rate;
static int g_blinker_driver_irq;
static uint32_t g_irq_count;
static spinlock_t g_irq_lock;
static wait_queue_head_t g_irq_wait_queue;
static int interrupt_flag;
static struct platform_driver blinker_platform_driver;

/*****************************************************************************
              MISC DEVICE DRIVER
*****************************************************************************/
struct blinker_dev {
        struct semaphore sem;
        unsigned long open_count;
        unsigned long release_count;
        unsigned long ioctl_count;
        unsigned long read_count;
        unsigned long write_count;
};
static struct blinker_dev the_blinker_dev = {
        .open_count = 0,
        .release_count = 0,
        .ioctl_count = 0,
        .write_count = 0,
        .read_count = 0,
};
static int blinker_dev_open(struct inode *ip, struct file *fp)
{
        struct blinker_dev *dev = &the_blinker_dev;

        if (down_interruptible(&dev->sem)) {
                pr_info("blinker_dev_open sem interrupted exit\n");
                return -ERESTARTSYS;
        }
        fp->private_data = dev;
        dev->open_count++;
        pr_info("open_count: 0x%08lX\n", dev->open_count);
        up(&dev->sem);
        return 0;
}
static int blinker_dev_release(struct inode *ip, struct file *fp)
{
        struct blinker_dev *dev = fp->private_data;

        if (down_interruptible(&dev->sem)) {
                pr_info("blinker_dev_open sem interrupted exit\n");
                return -ERESTARTSYS;
        }
        dev->release_count++;
        up(&dev->sem);
        pr_info("release_count: 0x%08lX\n", dev->release_count);
        return 0;
}
```

blinker_misc_module.c (continuation)

```
static long blinker_dev_ioctl(struct file *fp, unsigned int cmd, unsigned long arg)
{
        struct blinker_dev *dev = fp->private_data;
        u8 speed;
        u8 current_config;
        u8 mode;
        u8 leds;
        unsigned long flags;

        if (down_interruptible(&dev->sem)) {
                pr_info("blinker_dev_open sem interrupted exit\n");
                return -ERESTARTSYS;
        }

        dev->ioctl_count++;

        switch (cmd) {
        case IOC_SET_SPEED:
                if (get_user(speed, (uint32_t *)arg) < 0) {
                        up(&dev->sem);
                        pr_info("blinker_dev_ioctl get_user exit.\n");
                        return -EFAULT;
                }

                if (speed < 1 || speed > 15) {
                        up(&dev->sem);
                        pr_err("Invalid speed value %d\n", speed);
                        return -EINVAL;
                }

                /* acquire the irq_lock */
                spin_lock_irqsave(&g_irq_lock, flags);
                iowrite8(speed, IOADDR_BLINKER_SPEED(blinker_map_mem));
                /* release the irq_lock */
                spin_unlock_irqrestore(&g_irq_lock, flags);

                break;

        case IOC_SET_RUNNING:
                /* acquire the irq_lock */
                spin_lock_irqsave(&g_irq_lock, flags);

                current_config = ioread8(IOADDR_BLINKER_STATE(blinker_map_mem));
                iowrite8((current_config & BLINKER_CONFIG_MSK) | BLINKER_START_MSK,
                          IOADDR_BLINKER_CONFIG(blinker_map_mem));

                /* release the irq_lock */
                spin_unlock_irqrestore(&g_irq_lock, flags);
                break;

        case IOC_CLEAR_RUNNING:
                /* acquire the irq_lock */
                spin_lock_irqsave(&g_irq_lock, flags);

                current_config = ioread8(IOADDR_BLINKER_STATE(blinker_map_mem));
                iowrite8((current_config & BLINKER_CONFIG_MSK) & BLINKER_STOP_MSK,
                          IOADDR_BLINKER_CONFIG(blinker_map_mem));

                /* release the irq_lock */
                spin_unlock_irqrestore(&g_irq_lock, flags);
                break;

        case IOC_ENA_INT:
                /* acquire the irq_lock */
                spin_lock_irqsave(&g_irq_lock, flags);

                current_config = ioread8(IOADDR_BLINKER_STATE(blinker_map_mem));
                          iowrite8((current_config & BLINKER_CONFIG_MSK) | BLINKER_IENA_MSK,
                          IOADDR_BLINKER_CONFIG(blinker_map_mem));

                /* release the irq_lock */
                spin_unlock_irqrestore(&g_irq_lock, flags);

                break;
```

blinker_misc_module.c (continuation)

```
        case IOC_DIS_INT:
                /* acquire the irq_lock */
                spin_lock_irqsave(&g_irq_lock, flags);

                current_config = ioread8(IOADDR_BLINKER_STATE(blinker_map_mem));
                iowrite8((current_config & BLINKER_CONFIG_MSK) & BLINKER_IDIS_MSK,
                          IOADDR_BLINKER_CONFIG(blinker_map_mem));

                /* release the irq_lock */
                spin_unlock_irqrestore(&g_irq_lock, flags);

                break;
        case IOC_SET_MODE:
                if (get_user(mode, (uint32_t *)arg) < 0) {
                        up(&dev->sem);
                        pr_info("blinker_dev_ioctl get_user exit.\n");
                        return -EFAULT;
                }

                if (mode < 0 || mode > 1) {
                        up(&dev->sem);
                        pr_err("Invalid mode value %d\n", mode);
                        return -EINVAL;
                }

                /* acquire the irq_lock */
                spin_lock_irqsave(&g_irq_lock, flags);

                current_config = ioread8(IOADDR_BLINKER_STATE(blinker_map_mem));
                if (mode == 0) {
                        iowrite8((current_config & BLINKER_CONFIG_MSK) & BLINKER_SHIFT_MSK,
                                 IOADDR_BLINKER_CONFIG(blinker_map_mem));
                }
                else {
                        iowrite8((current_config & BLINKER_CONFIG_MSK) | BLINKER_BLINK_MSK,
                                IOADDR_BLINKER_CONFIG(blinker_map_mem));
                }

                /* release the irq_lock */
                spin_unlock_irqrestore(&g_irq_lock, flags);

                break;
        case IOC_GET_LEDS:
                leds = ioread8(IOADDR_BLINKER_LEDS(blinker_map_mem));

                if (put_user(leds, (uint32_t *)arg) < 0) {
                        up(&dev->sem);
                        pr_info("blinker_dev_ioctl put_user exit\n");
                        return -EFAULT;
                }

                break;
        case IOC_GET_CONFIG:
                current_config = ioread8(IOADDR_BLINKER_STATE(blinker_map_mem));

                if (put_user(current_config, (uint32_t *)arg) < 0) {
                        up(&dev->sem);
                        pr_info("blinker_dev_ioctl put_user exit\n");
                        return -EFAULT;
                }

                break;
        default:
                up(&dev->sem);
                pr_info("blinker_dev_ioctl bad cmd exit\n");
                return -EINVAL;
        }

        up(&dev->sem);
        pr_info("ioctl_count: 0x%08lX\n", dev->ioctl_count);
        return 0;
}
```

blinker_misc_module.c (continuation)

```
static ssize_t
blinker_dev_read(struct file *fp, char __user *user_buffer,
                    size_t count, loff_t *offset)
{
        struct blinker_dev *dev = fp->private_data;
        unsigned long flags;
        u8 state;

        if (down_interruptible(&dev->sem)) {
                pr_info("blinker_dev_read sem interrupted exit\n");
                return -ERESTARTSYS;
        }

        dev->read_count++;

        if (wait_event_interruptible(g_irq_wait_queue, interrupt_flag != 0)) {
                up(&dev->sem);
                pr_info("blinker_dev_read wait interrupted exit\n");
                return -ERESTARTSYS;
        }

        /* acquire the irq_lock */
        spin_lock_irqsave(&g_irq_lock, flags);

        interrupt_flag = 0;

        /* release the irq_lock */
        spin_unlock_irqrestore(&g_irq_lock, flags);

        memcpy_fromio(&state, IOADDR_BLINKER_STATE(blinker_map_mem), 1);
        if (copy_to_user(user_buffer, &state, 1)) {
                up(&dev->sem);
                pr_info("blinker_dev_read copy_to_user exit\n");
                return -EFAULT;
        }
        up(&dev->sem);
        pr_info("read_count: 0x%08lX\n", dev->read_count);
        return count;

}

static ssize_t
blinker_dev_write(struct file *fp,
                  const char __user *user_buffer, size_t count,
                  loff_t *offset)
{
        struct blinker_dev *dev = fp->private_data;
        u8 config;

        if (down_interruptible(&dev->sem)) {
                pr_info("blinker_dev_write sem interrupted exit\n");
                return -ERESTARTSYS;
        }

        dev->write_count++;


        if (copy_from_user(&config, user_buffer, 1)) {
                up(&dev->sem);
                pr_info("blinker_dev_write copy_from_user exit\n");
                return -EFAULT;
        }

        if (config < 1 || config > 7) {
                up(&dev->sem);
                pr_err("Invalid configuration value %d\n", config);
                return -EINVAL;
        }

        memcpy_toio(IOADDR_BLINKER_CONFIG(blinker_map_mem), &config, 1);
        up(&dev->sem);
        pr_info("write_count: 0x%08lX\n", dev->write_count);
        return count;
}
```

blinker_misc_module.c (continuation)

```
static const struct file_operations blinker_dev_fops = {
        .owner = THIS_MODULE,
        .open = blinker_dev_open,
        .release = blinker_dev_release,
        .unlocked_ioctl = blinker_dev_ioctl,
        .read = blinker_dev_read,
        .write = blinker_dev_write,
};

static struct miscdevice blinker_dev_device = {
        .minor = MISC_DYNAMIC_MINOR,
        .name = "blinker_misc",
        .fops = &blinker_dev_fops,
};

/*******************************************************************************
                IRQ HANDLER, SHOW AND STORE METHODS
*******************************************************************************/
irqreturn_t blinker_driver_interrupt_handler(int irq, void *dev_id)
{
        u8 data;

        spin_lock(&g_irq_lock);

        /* clear the interrupt by reading the state register*/
        data = ioread8(IOADDR_BLINKER_STATE(blinker_map_mem));

        /* increment the IRQ count */
        g_irq_count++;
        pr_info("IRQ handler executed. irq_count: 0x%08X\n", g_irq_count);

        data = ioread8(IOADDR_BLINKER_LEDS(blinker_map_mem));

        interrupt_flag = 1;
        spin_unlock(&g_irq_lock);
        wake_up_interruptible(&g_irq_wait_queue);

        return IRQ_HANDLED;
}

ssize_t leds_show(struct device_driver *drv, char *buf)
{
        ----------------------------
}

ssize_t state_show(struct device_driver *drv, char *buf)
{
        ----------------------------
}

ssize_t config_store(struct device_driver *drv, const char *buf, size_t count)
{
        ----------------------------

        /* acquire the irq_lock */
        spin_lock_irqsave(&g_irq_lock, flags);

        iowrite8(data, IOADDR_BLINKER_CONFIG(blinker_map_mem));

        /* release the irq_lock */
        spin_unlock_irqrestore(&g_irq_lock, flags);

        ----------------------------
}



ssize_t speed_store(struct device_driver *drv, const char *buf, size_t count)
{
        ----------------------------
}

static DRIVER_ATTR(speed, S_IWUGO, NULL, speed_store);
static DRIVER_ATTR(config, S_IWUGO, NULL, config_store);
static DRIVER_ATTR(leds, S_IRUGO, leds_show, NULL);
static DRIVER_ATTR(state, S_IRUGO, state_show, NULL);
```

blinker_misc_module.c (continuation)

```
static int blinker_probe(struct platform_device *pdev)
{
        ----------------------------
        int irq;

        /* acquire the probe lock */
        if (down_interruptible(&g_dev_probe_sem))
                return -ERESTARTSYS;

        /* check that any other device is using the driver */
        if (g_platform_probe_flag != 0)
                goto bad_exit_return;

        /* get blinker memory resource */
        ----------------------------

        /* get our interrupt resource */
        irq = platform_get_irq(pdev, 0);
        if (irq < 0) {
                pr_err("irq not available\n");
                goto bad_exit_return;
        }

        g_blinker_driver_irq = irq;

        /* get blinker clock resource */
        ----------------------------

        /* register our interrupt handler */
        init_waitqueue_head(&g_irq_wait_queue);
        spin_lock_init(&g_irq_lock);
        g_irq_count = 0;
        interrupt_flag = 0;
        ret = request_irq(g_blinker_driver_irq,
                          blinker_driver_interrupt_handler,
                          0,
                          blinker_platform_driver.driver.name,
                          &blinker_platform_driver);
        if (ret) {
                pr_err("request_irq failed");
                goto bad_exit_return;
        }

        /* create the sysfs entries */
        ----------------------------
        /* reserve blinker memory region */
        ----------------------------
        /* ioremap blinker memory region */
        ----------------------------
        /* register misc device blinker */
        sema_init(&the_blinker_dev.sem, 1);
        ret = misc_register(&blinker_dev_device);
        if (ret != 0) {
                pr_warn("Could not register misc device \"blinker_misc\"...");
                goto bad_exit_register_misc_device;
        }

        /* mark the driver as being used*/
        g_platform_probe_flag = 1;

        /* release the semaphore */
        up(&g_dev_probe_sem);

        pr_info("blinker device successfully connected\n");
        return 0;

        /* Exit with errors release the blocked resources */
        ----------------------------
bad_exit_freeirq:
        free_irq(g_blinker_driver_irq, &blinker_platform_driver);

        /* release the semaphore */
        up(&g_dev_probe_sem);
        pr_err("blinker device connect FAILED");
        return ret;
}
```

blinker_misc_module.c (continuation)

```
tatic int blinker_remove(struct platform_device *pdev)
{
        free_irq(g_blinker_driver_irq, &blinker_platform_driver);
        misc_deregister(&blinker_dev_device);

        /* remove sysfs entries and release memory resources */
        ---------------------------

        if (down_interruptible(&g_dev_probe_sem))
                return -ERESTARTSYS;

        /* mark the driver as free*/
        g_platform_probe_flag = 0;

        up(&g_dev_probe_sem);

        ---------------------------
}

static struct platform_driver blinker_platform_driver = {
        .probe = blinker_probe,
        .remove = blinker_remove,
        .driver = {
                   .name = "blinker_pd",
                   .owner = THIS_MODULE,
                   .of_match_table = blinker_driver_dt_ids,
                   },
};

static int __init blinker_init(void)
{
        ---------------------------

        sema_init(&g_dev_probe_sem, 1);

        ---------------------------

}

static void __exit blinker_exit(void)
{
        ---------------------------
}

module_init(blinker_init);
module_exit(blinker_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Antonio Carpeño <antonio.cruiz@upm.es>");
MODULE_AUTHOR("Mariano Ruiz <mariano.ruiz@upm.es>");
MODULE_DESCRIPTION("blinker peripheral misc_platform_driver example");
MODULE_VERSION("1.0");
```

Once the code has been presented it is time to cope with the explanation of its functionality. We are describing, bit by bit, the main language structures and functions that usually takes part in a misc driver. Some aspects are common to the platform device drivers you already know, that is why we will focus on the differences. The development of this driver will also serve to address some issues that were not dealt with previously, that is, how to safely access concurrently to shared resources and the use of interrupts.

The code starts with the definition of a structure whose only purpose is to keep some particular variables of our driver. After the definition it follows the instantiation of a variable of this type, with the assignation of the initial value. Like any other global variable, it is shared by any user space application that will be using the driver. Therefore, we will use them for semaphore exemplification purpose and will content just some useful statistics.

```
struct blinker_dev {
        struct semaphore sem;
        unsigned long open_count;
        unsigned long release_count;
        unsigned long ioctl_count;
        unsigned long read_count;
        unsigned long write_count;
};
```

Talking about concurrency, the kernel has the same constraints as any other multi-threaded application, that is, its state is global and visible for every thread. Concurrency happens because of the fact that several user space applications can be executing the driver methods simultaneously. In this case, Kernel preemption may cause the kernel to switch from the execution of one thread to another, and they may be using this shared structure. Otherwise, an interrupt can take place, and the interrupt handler will take over the current thread, and it is going to use shared resources too. The solution is the use of locking.

The simplest way to implement a lock for accessing a shared resource is the use of a semaphore to protect it, for which you must include the header file “linux/asm/semaphore.h”. You have to create the semaphore by using the type *struct semaphore* and then set it up with “void sema_init(struct semaphore *sem, int val)”, where *val* is the initial value to assign to a semaphore, which represents the number of resources available, 1 in our case.

Then you call “down” to grab the semaphore, and “up” to release it. Here, "down" implies the fact that the function decrements the value of the semaphore. If the semaphore is in use by another thread, the caller will be put to sleep for a while to wait for the semaphore becomes available. When this happens the kernel will grant access to the protected resources to this thread. There are three versions of down:

- ✓ void down(struct semaphore *sem)
  Decrements the value of the semaphore and waits as long as needed

- ✓ int down_interruptible(struct semaphore *sem)
  Does the same, but the operation is interruptible by the user

- ✓ int down_trylock(struct semaphore *sem)
  Never sleeps, so that if the semaphore is not available at the time of the call, this function returns immediately with a nonzero return value.

The interruptible version is almost always the one you need because it allows the process that is waiting on a semaphore to be interrupted by the user. However, the use of down_interruptible requires being careful, because when the operation is interrupted the caller does not grab the semaphore, returning a nonzero value. This value must be checked, so the application responds accordingly. When the function “down” has successfully finished it is said the thread is holding the semaphore, being now entitled to access the critical section protected by the semaphore. When the operations requiring mutual exclusion are complete, the

semaphore must be returned. This is accomplished calling the function “void up(struct semaphore *sem)”. There must be only one call to up. Another thing it is necessary to be aware is when an error is encountered while a semaphore is held, that semaphore must be released before returning.

A miscellaneous device must be described by the following structure, whose main fields are: *minor*, the minor number for the device, or MISC_DYNAMIC_MINOR to get a minor number automatically assigned; *name*, name of the device, which will be used to create the device node; *fops*, pointer to a struct file_operations structure.

```
static struct miscdevice blinker_dev_device = {
        .minor = MISC_DYNAMIC_MINOR,
        .name = "blinker_misc",
        .fops = &blinker_dev_fops,
};
```

Miscellaneous devices are normal character devices, and as it has been said, from an application, a character device is essentially a file located in /dev. Therefore, the driver of a character device must implement the regular operations with files. To achieve this, a character driver implements the operations described in the struct file_operations structure and register them. The Linux filesystem layer is responsible for calling the driver's operations when the user space application makes the corresponding system call, that is: the open() and release() system calls to open/close the device; the read() and write() system calls to read/write to/from the device; and the unlocked_ioctl() system call to call some driver-specific operations.

```
static const struct file_operations blinker_dev_fops = {
        .owner = THIS_MODULE,
        .open = blinker_dev_open,
        .release = blinker_dev_release,
        .unlocked_ioctl = blinker_dev_ioctl,
        .read = blinker_dev_read,
        .write = blinker_dev_write,
};
```

When the user space application opens the device file, the function “blinker_dev_open” is called. This function uses two parameters: a pointer to a *struct inode and a pointer to struct file. The former* is a structure that uniquely represents a file in the system. The later is a structure created every time a file is opened, contains information like the current position, the opening mode, etc. It also has a void *private_data pointer for free use, a pointer to the file structure is passed to all other operations.

```
static int blinker_dev_open(struct inode *ip, struct file *fp)
{
        struct blinker_dev *dev = &the_blinker_dev;

        if (down_interruptible(&dev->sem)) {
                pr_info("blinker_dev_open sem interrupted exit\n");
                return -ERESTARTSYS;
        }
        fp->private_data = dev;
        dev->open_count++;
        up(&dev->sem);
        return 0;
}
```

This other function is called when user space application closes the file.

```
static int blinker_dev_release(struct inode *ip, struct file *fp)
{
        ----------------------------
}
```

These two functions serve only to exemplification purposes; they only carry out some statistic calculation. However, they straightforwardly show the use of a semaphore. As you can see, any writing in a field of the global structure “the_blinker_dev” require to check the state of the lock previously and wait for its release

in case of it being locked with the down_interruptible call. Then, the release of the semaphore at the end of the operation, calling the up() function. This same operation will be seen in other methods further in the text.

The function “blinker_dev_read” is called when user space uses the read() system call on the device file. This function must read data from the device, write at most *count* bytes in the user space buffer u*ser_buffer*, and update the current position in the file *offset*. *fp* is a pointer to the same file structure that was passed in the open() operation. It must return the number of bytes read. You have to be careful because the read() operation usually block when there is not enough data to read from the device

```
static ssize_t blinker_dev_read
(struct file *fp, char __user *user_buffer, size_t count, loff_t *offset)
{
        ---------------------------
}
```

After increasing the counter of readings at the beginning, the main mission of the “blinker_dev_read” function is to wait for an IRQ event to occur. For that, the process will have to be put to sleep until this event takes place. There are several ways to make a kernel process sleep:

- ✓ void wait_event(queue, condition);

Sleeps until the task is woken up and the condition is true. It is important to be aware that it can not be interrupted, so the process can not be killed.

- ✓ int wait_event_killable(queue, condition);

Can be interrupted, but only by a fatal signal (SIGKILL). Returns -ERESTARSYS if interrupted.

- ✓ int wait_event_interruptible(queue, condition);

Can be interrupted by any signal. Returns -ERESTARTSYS if interrupted.

- ✓ int wait_event_timeout(queue, condition, timeout);

Also stops sleeping when the task is woken up and the timeout expired. Returns 0 if the timeout elapsed, non-zero if the condition was met.

- ✓ int wait_event_interruptible_timeout(queue, condition, timeout);

Same as above but with a timeout. Returns 0 if the timeout elapsed, -ERESTARTSYS if interrupted, a positive value if the condition was met.

We have chosen the interruptible version.

```
wait_event_interruptible(g_irq_wait_queue, interrupt_flag != 0))
```

The condition of exit is the evaluation of a flag, named *interrupt_flag*, which must be set up in the interrupt service routine (ISR). Note the use of a queue of processes waiting for the event which was declared globally.

```
static wait_queue_head_t g_irq_wait_queue;
```

After woken-up the function continues putting down the flag. However, again this is a shared resource so it is mandatory to ensure no other process is accessing the variable at this moment. This time we have used another locking mechanism, spinlocks.

Spinlocks are locks to be used for code that is not allowed to sleep, as interrupt handlers, or that doesn't want to sleep, as critical sections. Spinlocks never sleep and keep spinning in a loop until the lock is

available. To achieve this, spinlocks cause kernel preemption to be disabled on the CPU executing them. Therefore, the critical section protected by a spinlock is not allowed to sleep.

There are several ways of invoking a spinlock. If you do not want to disable interrupts because you only want to lock critical sections you do not want to sleep, you must use:

```
void spin_lock(spinlock_t *lock);
void spin_unlock(spinlock_t *lock);
```

Sometimes you want the lock can be accessed in both process and interrupt context, to prevent preemption by interrupts, you must disable and restore interrupts. Then, you must use:

```
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
```

When you want to protect shared data accessed in process context and a soft interrupt, then, you don't need to disable hardware interrupts. You have to use:

```
void spin_unlock_bh(spinlock_t *lock);
```

In our case we are protecting a shared resource that could be accessed from an interrupt, so, our choice was the following:

```
/* acquire the irq_lock */
spin_lock_irqsave(&g_irq_lock, flags);

interrupt_flag = 0;

/* release the irq_lock */
spin_unlock_irqrestore(&g_irq_lock, flags);
```

After having put down the interrupt flag, it is time for the read function of reading the *state* of the blinker peripheral from its STATE register and copying this information into user memory space. To read from an IO address in previous occasions we used the ioread8(), ioread16() or ioread32(), function, this is valid when you want to read only one, two or four bytes. But, when you want to read a block of data from IO space remapped to kernel memory then you must use the memcpy_fromio() function. That is the sentence used in our function, which copy 1 (last parameter) byte from memory address pointed by *blinker_map_mem* to the variable *state* (kernel memory):

```
memcpy_fromio(&state, IOADDR_BLINKER_STATE(blinker_map_mem), 1);
```

To send the state to user space you have to know that Kernel code is not allowed to directly access user space memory, using memcpy() or direct pointer dereferencing. There are specialized functions for this operation:

To copy a single value you must use:

- ✓ get_user(v, p);
  The kernel variable v gets the value pointed by the user space pointer p

- ✓ put_user(v, p);
  The value pointed by the user space pointer p is set to the contents of the kernel variable v.

To copy a buffer you must use:

- ✓ unsigned long copy_to_user(void __user *to, const void *from, unsigned long n);
- ✓ unsigned long copy_from_user(void *to, const void __user *from, unsigned long n);
  The return value must be checked. Zero on success, non-zero on failure. If non-zero, the convention is to return -EFAULT.

In our case we have used the following sentence to copy 1 (las parameter) byte from the variable *state* to the user memory space buffer *user_buffer*.

```
copy_to_user(user_buffer, &state, 1))
```

Another method is blinker_dev_write() which is called when user space uses the write() system call on the device file. This method operates the opposite of read(), must read at most *count* bytes from *user_buffer*, write it to the device, update *offset* and return the number of bytes written.

```
static ssize_t blinker_dev_write
(struct file *fp, const char __user *user_buffer, size_t count, loff_t *offset)
{
        ---------------------------
}
```

You can see an example of how to use the opposite functions to read from user space memory to kernel memory with copy_from_user() and write data from kernel memory to io remapped into memory space with memcpy_toio(), instead of iowrite8(), iowrite16() or iowrite32(), as we did on previous occasions.

```
copy_from_user(&config, user_buffer, 1)) {
memcpy_toio(IOADDR_BLINKER_CONFIG(blinker_map_mem), &config, 1);
```

One last method defined in the file_operations structure is the blinker_dev_ioctl(). It is called when the user space application uses the ioctl() system call. It is called unlocked because it doesn't disable the kernel preemption, former ioctl() method prevent the execution of any other process while is being executed, holding the Big Kernel Lock. This function allows to extend the driver capabilities beyond the limited read/write API, it is the favorite option when you want to change the behavior of the peripheral (control it) or know its state. This function is the heart of the control and configuration of the blinker driver. About its parameters, *cmd* is a number identifying the operation to perform, and *arg* is the optional argument passed as the third argument of the ioctl() system call. The semantic of *cmd* and *arg* is driver-specific. In our example, the set of commands were defined in the *blinker_module.h* file. Below you can find some examples of commands recognition and execution.

```
static long blinker_dev_ioctl(struct file *fp, unsigned int cmd, unsigned long arg)
{
        ---------------------------

        case IOC_SET_SPEED:
                if (get_user(speed, (uint32_t *)arg) < 0) {
                        up(&dev->sem);
                        pr_info("blinker_dev_ioctl get_user exit.\n");
                        return -EFAULT;
                }

                if (speed < 1 || speed > 15) {
                        up(&dev->sem);
                        pr_err("Invalid speed value %d\n", speed);
                        return -EINVAL;
                }

                /* acquire the irq_lock */
                spin_lock_irqsave(&g_irq_lock, flags);
                iowrite8(speed, IOADDR_BLINKER_SPEED(blinker_map_mem));
                /* release the irq_lock */
                spin_unlock_irqrestore(&g_irq_lock, flags);

                break;

        ---------------------------
```

```
        case IOC_GET_LEDS:
                leds = ioread8(IOADDR_BLINKER_LEDS(blinker_map_mem));

                if (put_user(leds, (uint32_t *)arg) < 0) {
                        up(&dev->sem);
                        pr_info("blinker_dev_ioctl put_user exit\n");
                        return -EFAULT;
                }
                break;

        ----------------------------
}
```

In this example, again, you can see the use of spinlock to manage a potential race with the irq handler when accessing the internal registers of the blinker with iowrite8(), and the use of the functions get_user() instead of copy_from_user() and put_user() instead of copy_to_user().

The rest of the code is the typical Platform Device Driver methods, show(), store(), probe(), remove(), init() and exit(). The show and store methods don't present anything new, but for the use of spinlock when writing the internal registers of the blinker peripheral. However, there is a completely new resource that is the use of interrupts.

Here is the interrupt handler for the unique blinker interrupt source when one button of the de1soc has been pushed.

```
irqreturn_t blinker_driver_interrupt_handler(int irq, void *dev_id)
{
        u8 data;

        spin_lock(&g_irq_lock);

        /* clear the interrupt by reading the state register*/
        data = ioread8(IOADDR_BLINKER_STATE(blinker_map_mem));

        /* increment the IRQ count */
        g_irq_count++;

        data = ioread8(IOADDR_BLINKER_LEDS(blinker_map_mem));

        interrupt_flag = 1;
        spin_unlock(&g_irq_lock);
        wake_up_interruptible(&g_irq_wait_queue);

        return IRQ_HANDLED;
}
```

Analyzing the code we can realize that the ISR only perform two tasks. On the one hand, the handler acknowledges the interrupt to the device, by reading the state register. After this, the blinker deactivates the interrupt line. Then the interrupt counter is increased. The LEDS register is also read, but nothing is done with this information. The interrupt flag is marked to signal the read() process that is waiting for this event, which is what the read function will ultimately pend on. Finally, the wake_up_interruptible() function awaken all sleeping processes registered in the g_irq_wait_queue.

This last action deserves and explanation. Typically, interrupt handlers do this when data sleeping processes are waiting for something become available. There are two possible options:

- ✓ wake_up(&queue);
  Wakes up all processes in the wait queue

- ✓ wake_up_interruptible(&queue);
  Wakes up all processes waiting in an interruptible sleep on the given queue

Concerning the probe() and remove() methods, we start our probe function in much the same way as before, performing similar tasks that the conventional platform driver of the previous section. But, checking that the driver is not already in use by any other device.

```
static int blinker_probe(struct platform_device *pdev)
{
        ---------------------------

        if (g_platform_probe_flag != 0)
                goto bad_exit_return;

        ---------------------------
}
```

However, some additional tasks are also needed. Among other resources, it is necessary to get the interrupt resource from the hardware.

```
irq = platform_get_irq(pdev, 0);
```

Once we have the interrupt resource we must register the interrupt handler in the platform driver infraestructure

```
ret = request_irq(....)
```

and declare a wait queue to store the list of threads waiting for an event.

```
static wait_queue_head_t g_irq_wait_queue;
init_waitqueue_head(&g_irq_wait_queue);
```

Then the probe() function initializes the spinlock

```
static spinlock_t g_irq_lock;
spin_lock_init(&g_irq_lock);
```

The function continues using misc_register() to register the blinker device with the kernel. And initialize the semaphore for controlling the access to the global variables

```
ret = misc_register(&blinker_dev_device);
sema_init(&the_blinker_dev.sem, 1);
```

Driver binding is performed automatically by the driver core, invoking driver probe() after finding a match between device and driver. When a driver is registered using platform_driver_register(), all unbound devices on that bus are checked for matches. Drivers usually register later during booting, or by module loading. However, in the blinker case the device use exclusive resources, switches, buttons, and LEDs. Only one device can make use of them. Therefore, only one device can use our blinker driver. To ensure this to happen we use a flag signaling that the driver is already in use and that any other device trying to use this driver cannot have success.

```
        g_platform_probe_flag = 1;
```

Our remove function simply removes our misc devices with misc_deregister(), release the interrupt handler, and down the flag of blinker driver in use.

```
static int blinker_remove(struct platform_device *pdev)
{
        ----------------------------
        free_irq(g_blinker_driver_irq, &blinker_platform_driver);
        misc_deregister(&blinker_dev_device);

        ----------------------------

        g_platform_probe_flag = 0;

        ----------------------------
}
```

To finish with the blinker_misc_module.c code we must say that the init() and exit() methods are quite similar to the conventional pplatform device driver studied before. There is only one difference, it is the initialization of the g_dev_probe_sem semaphore.

```
static int __init blinker_init(void)
{
        sema_init(&g_dev_probe_sem, 1);

        ----------------------------
}
```

7.2.2 Compiling *blinker_misc_module*.

To compile this driver, we can follow two possible strategies again, compile with the makefile, for which we have to modify the Kbuild file to contemplate the compiling of our new driver:

Kbuild

```
obj-m  := blinker_pd_module.o

obj-m  += blinker_misc_module.o
```

The second strategy is to use Buildroot to compile the source code and includes the kernel object code "blinker_misc_module.ko" as a loadable module into the /lib directory. To accomplish this you have to follow the same procedure as with the platform device driver, which we repeat here on behalf of commodity:

1. Add your new source file to the appropriate source directory. In this case: /kernel/drivers/mis/

2. Describe the configuration interface for your new driver by adding the following lines to the Kconfig file in this directory:

```
config BLINKER_MISC
     tristate "blinker misc module"
     default n
     help
       Select this configuration to enable the blinker_misc module
       Don't select as built-in together with blinker_uio or
       blinker_platform. Only one of then can be inserted at one time
```

1. Add this line in the Makefile file based on the Kconfig setting:

```
obj-$(CONFIG_BLINKER_MISC)   += blinker_misc_module.o
```

2. Now you must set the appropriate configuration options to compile the kernel. Enter the following command in Buildroot base directory

```
$ make linux-xconfig
```

3. Mark the "blinker misc module" as shown in the Fig. 117 and Fig. 118to enable its compilation as a loadable module, if you select the "check mark" then the module will be included as part of the kernel and it won't be necessary to load it with the modprobe command.

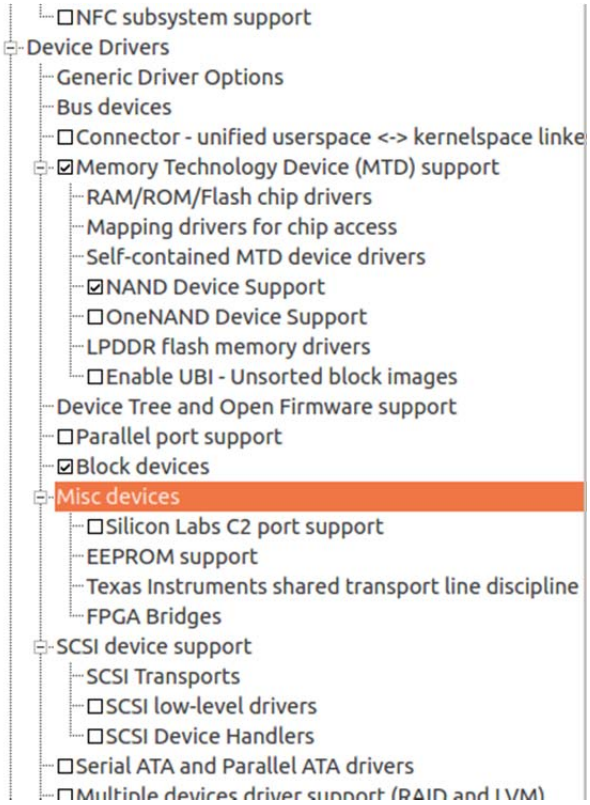


Fig. 117: Adding miscellanies drivers in the Kernel

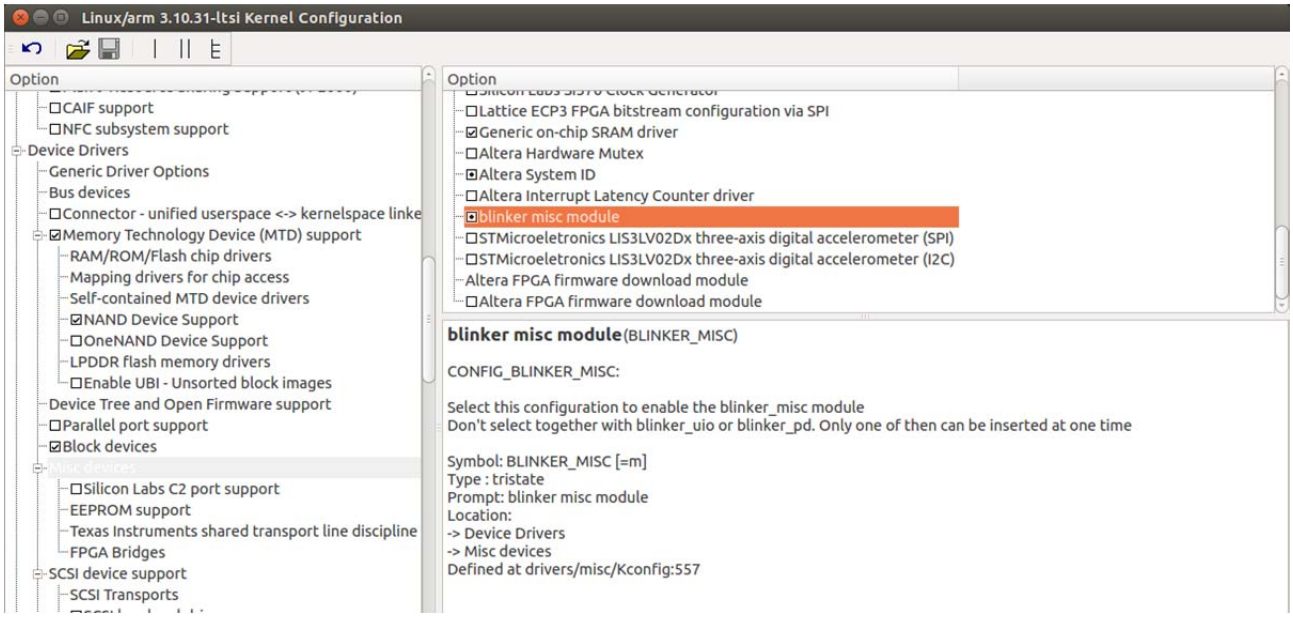


Fig. 118: Detail of the selection of the kernel driver.

4. To order Buildroot to compile only the kernel without recompiling the rest of the source files, remember, you must enter this command

```
$ make linux-rebuild
```

Again, after the execution, Builroot generates the new kernel binaries “zImage”, the new .dtb file with the device tree blob, and the blinker driver module, in the /lib/modules/3.10.31-ltsi/kernel/drivers/misc/ directory as a .ko file in the /output/target directory in the Buildroot tree.

5. Finally, to create the new rootfs, which includes our new driver and create the new SD image you must execute this command.

```
$ make
```

When the new image has been made and the DE1-SoC has booted up the first thing we must do is to load the new driver into the kernel. The process is shown below

```
$ cd /lib/modules/3.10.31-ltsi/kernel/drivers/misc/
$ ls
blinker_misc_module.ko
$ modprobe blinker_misc_module
blinker device successfully connected
blinker module successfully inserted
$
```

After the successful l load of the driver, the /sys and the /dev directories have been populated with our blinker driver files. To verify this, you can inspect these directories and see something similar to the image shown in the Fig. 119

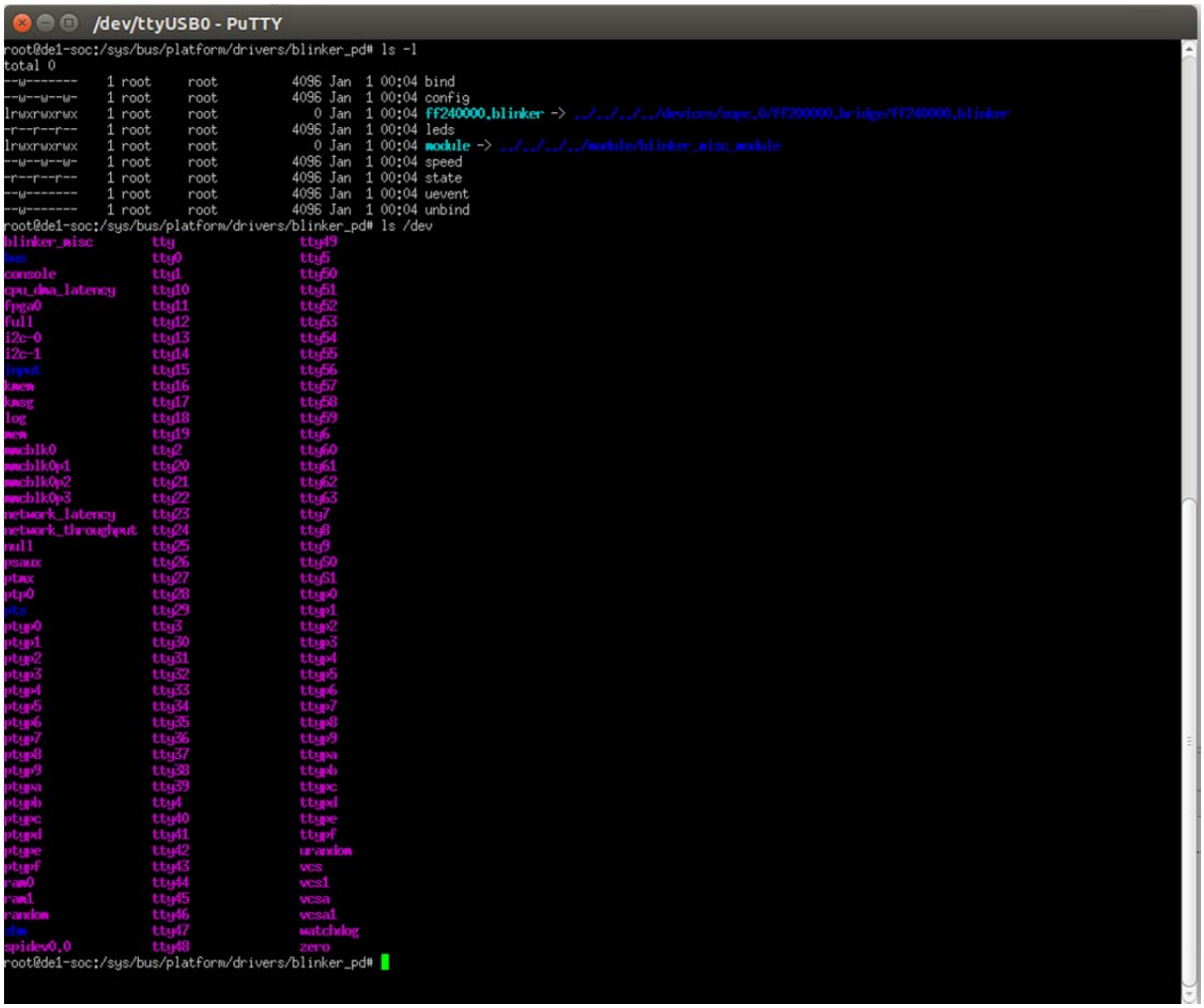


Fig. 119: folder with the miscellaneous driver

After the completion of the test tasks, remember, to remove your driver this is the command you have to enter

```
$ rmmod blinker_misc_module.ko
blinker device successfully removed
blinker driver successfully removed
$
```

7.2.3 Accessing *blinker_misc_module* from a user space application

Developing a user space application that uses the miscellaneous driver just created do not differ much of the user-space applications developed in previous sections of this document. Therefore, we are not showing the whole code, but instead, we are focusing on the differences. It is up to the student to add the missing code to create the complete application.

The header file “blinker_misc_module_test.h” must include the definition of the sysfs entries through which the internal registers can be reached. Moreover, it will be helpful to define the code numbers that identifies the commands for the ioctl() system call, which should match the ones defined in the “blinker_module.h”, the header file used for the driver code.

With regard to the “blinker_misc_module_test.c” take notice of the ioctl() and write() system calls, and how the wait_int() function makes use of the system call read() where the process will get blocked until the user push some button and the consequent interrupt arises. The main() program do not differ either of the usual code, but for the opening of the device file in /dev directory. The handler returned by this function will be used in the rest of the system calls. It is up to the student the use of the sysfs entries, located in /sys/bus/platform/drivers directory, which launch the show() and store() methods of the platform device driver, or the ioctl() system calls to perform the operations demanded by the long options entered on the command line. However, we strongly recommend the use of the former.

blinker_misc_module_test.h

```
//  sysfs files
#define SYSFS_SPEED_FILE "/sys/bus/platform/drivers/blinker/speed"
#define SYSFS_CONFIG_FILE "/sys/bus/platform/drivers/blinker/config"
#define SYSFS_LEDS_FILE "/sys/bus/platform/drivers/blinker/leds"
#define SYSFS_STATE_FILE "/sys/bus/platform/drivers/blinker/state"

// ioctl values
#define IOC_SET_SPEED         (0x4001EE00)
#define IOC_SET_MODE          (0x4001EE03)
#define IOC_SET_RUNNING       (0x0000EE01)
#define IOC_CLEAR_RUNNING     (0x0000EE02)
#define IOC_ENA_INT           (0x0000EE06)
#define IOC_DIS_INT           (0x0000EE07)
#define IOC_GET_LEDS          (0x8001EE04)
#define IOC_GET_CONFIG        (0x8001EE05)
```

blinker_misc_module_test.c

```c
/*****************************************************************
 * commands functions implementation
 *****************************************************************/

void do_set_mode(int dev_blinker_fd){
        int result;

        mode = atoi(param_value);

        result = ioctl(dev_blinker_fd, IOC_SET_MODE, &mode);
        if(result != 0) {
                perror("ioctl failed");
        }

        printf("Mode set to %u.\n", mode);
}

void do_set_config(int dev_blinker_fd){
        int result;

        config = atoi(param_value);

        result = write(dev_blinker_fd, &config, 1);
        if(result < 0) {
                perror("write configuration failed");
        }

        printf("Configuration set to %u.\n", config);
}

void do_wait_int(int dev_blinker_fd){
        int result;
        unsigned char config_number;

        printf("Waiting for the user to push any button\n");

        result = read(dev_blinker_fd, &config_number, 1);
        if(result < 0) {
                perror("waiting for interrupt failed");
        }

        printf ("Current configuration\n\tspeed -> %d\n\tconfig -> %d\n", (config_number >> 4) &
        0x0F, config_number & 0x0F);

}


/*****************************************************************
 * Main program
 *****************************************************************/
int main(int argc, char **argv) {
        int dev_blinker_fd;

        // parse the command line arguments
        ----------------------------

        // open the /dev/blinker_misc device
        dev_blinker_fd = open("/dev/blinker_misc", O_RDWR | O_SYNC);
        if(dev_blinker_fd < 0) {
                perror("dev_blinker open error");
                exit(EXIT_FAILURE);
        }

        // perform the operation selected by the command line arguments
        ----------------------------

        // close the /dev/blinker_misc device
        close(dev_blinker_fd);
        exit(EXIT_SUCCESS);
}
```

7.2.4 Exercise 7: Development of an application to test the blinker_misc_module.

Write and test a user space application that implements the following set of commands:

```
#define HELP_STR "\
\n\
Only one of the following options may be passed in per invocation:\n\
\n\
  -d, --set_speed speed_value\n\
Set the speed of the blinking.\n\
      15:  maximum speed.\n\
      1: minimum speed.\n\
\n\
  -m, --set_mode mode_value\n\
Set the mode of operation of the blinking.\n\
      mode_value: mode of operation 0 (shift) 1 (switches blinking).\n\
\n\
  -s, --start\n\
Start the operation of blinker.\n\
\n\
  -p, --stop\n\
Stop the operation of blinker.\n\
\n\
  -e, --ena_int\n\
Enable interrupt.\n\
\n\
  -i, --dis_int\n\
Disable interrupt.\n\
\n\
  -g, --get_config\n\
Read the current configuration and speed.\n\
\n\
  -l, --get_leds\n\
Read the current leds position.\n\
\n\
  -c, --set_config config_value\n\
Set the mode of operation of the blinking.\n\
bit pattern: 00000ems.\n\
      e: enable interrupt 0 (disabled) 1 (enabled).\n\
      m: mode of operation 0 (blinker) 1 (switches blinking).\n\
      s: 0 (stopped) 1 (running).\n\
\n\
  -w, --wait_int\n\
Wait for the user to push some button and show the new configuration.\n\
\n\
  -h, --help\n\
Display this help message.\n\
\n\
"
```

7.3 UserSpace Input Output (UIO) Drivers.

The development of Linux device driver is a complex task because debugging is not easy. Every time that you have an error and the kernel crashes, you need to reboot again your system. Finding the piece of code with the bug is not a simple task. This is the reason of developing drivers using the user-space approach. The development and implementation are the easiest but, on the contrary, you have some limitations. In the following paragraphs, we are going to present an example of the implementation of blinker using this approach. The main concept of a User Space Input/Output driver is that it gives access to the raw hardware registers to user space applications. Within kernel space it is not necessary, therefore, to control any of the hardware details. Thus, it is possible to implement device drivers in user space, avoiding the lavish thing of coping with the bugs in the development stage with the limited resources of any kernel process.

Using UIO for your device driver presents some advantages, such as:

- You only have one small kernel module to write and maintain.
- You can develop the main part of your driver in user space, with all the tools and libraries.
- You can kill and debug the driver code, the bugs in your driver won't crash the kernel.
- You can update your driver without recompiling the kernel.
- You do not need kernel coding skills.
- You can reuse code between devices more easily.
- You can write your drivers in any language.
- You can load and unload the driver while the kernel is running, whiles kernel code cannot be.
- You can use floating-point computation.
- You can reach higher performance with memory-mapped devices, thanks to the avoidance of system calls.

However, this kind of drivers has also some drawbacks, being the more relevant:

- It is less straightforward to handle interrupts.
- It increases interrupt latency in comparison with the kernel code.

7.3.1 Developing the UIO driver. *Blinker_uio_module*.

An UIO driver has a completely different form of operation that the one of the character driver. When an UIO driver is loaded into the kernel a device file is created inside the /dev directory. The UIO device can then be accessed through this file. Several sysfs attribute files are created as well, where some relevant information of the device characteristics and operation can be found. When there is more than one UIO device, which is the most common situation as you can imagine, the device files are called consecutively: /dev/uio0 for the first device, /dev/uio1 for the second, and so on.
The /dev/uioX file is used to access the address space of the device, using mmap() to access registers or memory locations of your peripheral. The difference with the use of /devmem is that the latter needs applications running in the context of privileged users, whereas the former can be used by non-privileged users. On the other hand, to handle interrupts it is only needed to read from /dev/uioX file. This is a blocking read that will eventually return as soon as an interrupt occurs. The integer value returned from the /dev/uioX reading throws the total interrupt count. This number should be used to figure out if you missed some interrupts.

Though this is a good summary of UIO drivers operation, here you can find almost the whole bunch of code for the “blinker_uio_module”. Later you will find detailed descriptions of this code.

blinker_uio_module.c

```c
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/clk.h>
#include <linux/io.h>
#include <linux/interrupt.h>
#include <linux/uio_driver.h>
#include "blinker_module.h"

/* global variables */
static struct semaphore g_dev_probe_sem;
static int g_platform_probe_flag;
static int g_blinker_driver_base_addr;
static int g_blinker_driver_size;
static int g_blinker_driver_irq;
static unsigned long g_blinker_driver_clk_rate;
static void *g_ioremap_addr;
static struct semaphore g_blinker_uio_dev_sem;
static struct platform_driver blinker_platform_driver;

/*****************************************************************************
              UIO DRIVER
              IRQ HANDLER, OPEN, RELEASE METHODS
*****************************************************************************/
irqreturn_t blinker_uio_interrupt_handler(int irq, struct uio_info *dev_info)
{
       /* clear the interrupt */
       // TO BE COMPLETED
}
static int blinker_uio_irqcontrol(struct uio_info *info, s32 irq_on)
{
       u8 current_config;
       if (irq_on) {
              /* Enable interrupt  */
              // TO BE COMPLETED

       } else {
              /* Disable interrupt  */
              // TO BE COMPLETED

              /* ensure there is no pending IRQ */
              // TO BE COMPLETED
       }
       return 0;
}
static int blinker_uio_open(struct uio_info *info, struct inode *inode)
{
       if (down_trylock(&g_blinker_uio_dev_sem) != 0)
              return -EAGAIN;
       return 0;
}
static int blinker_uio_release(struct uio_info *info, struct inode *inode)
{
       /* ensure there is no pending IRQ */
       // TO BE COMPLETED

       up(&g_blinker_uio_dev_sem);
       return 0;
}
static struct uio_info blinker_uio_info = {
       .name = "blinker_uio_module",
       .version = "1.0",
       .irq_flags = 0,
       .handler = blinker_uio_interrupt_handler,
       .open = blinker_uio_open,
       .release = blinker_uio_release,
       .irqcontrol = blinker_uio_irqcontrol,
};
```

blinker_uio_module.c (continuation)

```
/*****************************************************************************
             PLATFORM-DEVICE DRIVER
             PROBE, REMOVE, RELEASE, INIT AND EXIT METHODS
*****************************************************************************/

static int platform_probe(struct platform_device *pdev)
{
       int ret_val;
       struct resource *r;
       struct resource *blinker_driver_mem_region;
       int irq;
       struct clk *clk;
       unsigned long clk_rate;
       int i;

       ret_val = -EBUSY;

       /* acquire the probe lock */
       if (down_interruptible(&g_dev_probe_sem))
              return -ERESTARTSYS;

       if (g_platform_probe_flag != 0)
              goto bad_exit_return;

       ret_val = -EINVAL;

       /* get memory resource */
       r = platform_get_resource(pdev, IORESOURCE_MEM, 0);
       if (r == NULL) {
              pr_err("IORESOURCE_MEM, 0 does not exist\n");
              goto bad_exit_return;
       }

       g_blinker_driver_base_addr = r->start;
       g_blinker_driver_size = resource_size(r);

       /* get interrupt resource */
       irq = platform_get_irq(pdev, 0);
       if (irq < 0) {
              pr_err("irq not available\n");
              goto bad_exit_return;
       }

       g_blinker_driver_irq = irq;

       /* get clock resource */
       clk = clk_get(&pdev->dev, NULL);
       if (IS_ERR(clk)) {
              pr_err("clk not available\n");
              goto bad_exit_return;
       } else {
              clk_rate = clk_get_rate(clk);
       }

       g_blinker_driver_clk_rate = clk_rate;

       ret_val = -EBUSY;
```

blinker_uio_module.c (continuation)

```
	/* reserve memory region */
	blinker_driver_mem_region = request_mem_region(g_blinker_driver_base_addr,
	                                               g_blinker_driver_size,
	                                               "blinker_uio_driver_hw_region");
	if (blinker_driver_mem_region == NULL) {
		pr_err("request_mem_region failed: g_blinker_driver_base_addr\n");
		goto bad_exit_return;
	}

	/* ioremap memory region */
	g_ioremap_addr = ioremap(g_blinker_driver_base_addr, g_blinker_driver_size);
	if (g_ioremap_addr == NULL) {
		pr_err("ioremap failed: g_blinker_driver_base_addr\n");
		goto bad_exit_release_mem_region;
	}

	/* initialize uio_info struct uio_mem array */
	blinker_uio_info.mem[0].memtype = UIO_MEM_PHYS;
	blinker_uio_info.mem[0].addr = r->start;
	if (resource_size(r) > PAGE_SIZE) blinker_uio_info.mem[0].size =
	     resource_size(r);
	else blinker_uio_info.mem[0].size = PAGE_SIZE;
	blinker_uio_info.mem[0].name = "blinker_uio_driver_hw_region";
	blinker_uio_info.mem[0].internal_addr = g_ioremap_addr;

	for (i = 1; i < MAX_UIO_MAPS; i++)
		blinker_uio_info.mem[i].size = 0;

	/* initialize uio_info irq */
	blinker_uio_info.irq = g_blinker_driver_irq;

	/* register the uio device */
	sema_init(&g_blinker_uio_dev_sem, 1);
	ret_val = uio_register_device(&pdev->dev, &blinker_uio_info);
	if (ret_val != 0) {
		pr_warn("Could not register device \"blinker_uio\"...");
		goto bad_exit_iounmap;
	}

	g_platform_probe_flag = 1;
	up(&g_dev_probe_sem);
	return 0;

bad_exit_iounmap:
	iounmap(g_ioremap_addr);
bad_exit_release_mem_region:
	release_mem_region(g_blinker_driver_base_addr, g_blinker_driver_size);
bad_exit_return:
	up(&g_dev_probe_sem);
	pr_info("platform_probe bad_exit\n");
	return ret_val;
}
```

blinker_uio_module.c (continuation)

```
static int platform_remove(struct platform_device *pdev)
{
	uio_unregister_device(&blinker_uio_info);

	iounmap(g_ioremap_addr);
	release_mem_region(g_blinker_driver_base_addr, g_blinker_driver_size);

	if (down_interruptible(&g_dev_probe_sem))
		return -ERESTARTSYS;

	g_platform_probe_flag = 0;
	up(&g_dev_probe_sem);

	return 0;
}

static struct of_device_id blinker_driver_dt_ids[] = {
	{
	 .compatible = "blinker,driver-1.0"},
	{ /* end of table */ }
};

MODULE_DEVICE_TABLE(of, blinker_driver_dt_ids);

static struct platform_driver blinker_platform_driver = {
	.probe = platform_probe,
	.remove = platform_remove,
	.driver = {
		  .name = "blinker",
		  .owner = THIS_MODULE,
		  .of_match_table = blinker_driver_dt_ids,
		  },
};

static int blinker_init(void)
{
	int ret_val;

	sema_init(&g_dev_probe_sem, 1);

	ret_val = platform_driver_register(&blinker_platform_driver);
	if (ret_val != 0) {
		pr_err("blinker module insert FAILED.\n");
		return ret_val;
	}

	pr_info("blinker module successfully inserted\n");
	return 0;
}

static void blinker_exit(void)
{
	platform_driver_unregister(&blinker_platform_driver);

	pr_info("blinker module successfully removed\n");
}

module_init(blinker_init);
module_exit(blinker_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Antonio Carpeño <antonio.cruiz@upm.es>");
MODULE_AUTHOR("Mariano Ruiz <mariano.ruiz@upm.es>");
MODULE_DESCRIPTION("blinker peripheral uio_platform_driver example");
MODULE_VERSION("1.0");
```

First, we come across the interrupt handler. What you need to do in your interrupt handler depends on the specifics of your hardware and on how you want to handle with the interrupt service routine. However, it should be wise to try to keep your kernel interrupt handler light of code and try to move most of the code to user space. What is more, you might have an empty handler if your hardware does not require any action that you have to perform just after each interrupt.

On the other hand, your hardware might need some action to be performed after each interrupt, in such a case you must do it in your kernel module, without relying on the user space part of your driver. The reason is that the user space application can terminate unexpectedly; leaving your hardware in a state where suitable interrupt treatment is still needed. It could also be the case of applications in which you must read data from your device at each interrupt and save it into kernel memory. Doing that in the kernel part of your interrupt handler you could avoid the loss of data that could arise if your user space application misses an interrupt. In our case we have to clear the interrupt flag whenever the interrupt had been attended, therefore it would be appropriate to carry out this action in the interrupt handler of our driver. This is the only action that our handler will be performing.

It is common practice that the device driver supports interrupt sharing. This is only possible if the driver can figure out whether our device was the one that triggered the interrupt. This usual way of doing this is by looking at the interrupt flag in the status register. Our driver only performs its required actions if it checks that the IRQ flag is actually set, returning IRQ_HANDLED. If, on the contrary, the driver detects that it was not our device that caused the interrupt, it will do nothing and return IRQ_NONE, allowing the kernel to call the next possible interrupt handler.

```
irqreturn_t blinker_uio_interrupt_handler(int irq, struct uio_info *dev_info)
{
	/* clear the interrupt */


}
```

UIO drivers can also implement a “write” function which will be called when the user space application calls the writing system call to the /dev/uioX file. This resource is usually needed when some kind of operation on the device should cannot be accomplished in the user space process. Thus, a write to /dev/uioX will call the irqcontrol() function implemented by the driver. If a driver does not implement irqcontrol(), write() will return with -ENOSYS. In our blinker driver we write either 0 or 1 to disable or enable the interrupt.

```
static int blinker_uio_irqcontrol(struct uio_info *info, s32 irq_on)
{
	u8 current_config;
	if (irq_on) {
		/* Enable interrupt  */

	} else {
		/* Disable interrupt  */
		/* ensure there is no pending IRQ */
	}
	return 0;
}
```

When the user space application opens the device file, the function “blinker_uio_open” is called. This function uses these parameters: *struct inode* is a structure that uniquely represents a file in the system; *struct uio_info* is a structure that contains information about the uio device whose fields will be explained later.

```
static int blinker_uio_open(struct uio_info *info, struct inode *inode)
{
	--------------------------
}
```

This other function is called when user space application closes the file.

```
static int blinker_uio_release(struct uio_info *info, struct inode *inode)
{
	--------------------------
}
```

The *uio_info* structure keeps details of your driver for the UIO framework. Some fields are required, whereas others are optional.

- ✓ *const char *name*
  [Required]. The name of your driver as it will appear in sysfs. It is recommended using the name of your module for this field.
- ✓ *const char *version*
  [Required]. This string appears in /sys/class/uio/uioX/version entry file.
- ✓ *struct uio_mem mem[MAX_UIO_MAPS]*
  [Required if you have a memory that can be mapped]. For each mapping you need to fill one uio_mem structure, whose description will be presented later.
- ✓ *long irq*
  [Required]. If your hardware generates an interrupt, the driver must figure out the irq number during its initialization tasks. If you do not have any interrupt, you should set irq to UIO_IRQ_NONE.
- ✓ *unsigned long irq_flags*
  [Required if you've set irq to a hardware interrupt number]. The flags given here will be used in the call to request_irq().
- ✓ *int (*mmap)(struct uio_info *info, struct vm_area_struct *vma)*
  [Optional]. If you need a special mmap() function, you can set it here. If this pointer is not NULL, your mmap() will be called instead of the built-in one.
- ✓ *int (*open)(struct uio_info *info, struct inode *inode)*
  [Optional]. The pointer to your open() method
- ✓ *int (*release)(struct uio_info *info, struct inode *inode)*
  [Optional]. The pointer to your release() method.
- ✓ *int (*irqcontrol)(struct uio_info *info, s32 irq_on)*
  [Optional]. If you want to enable or disable interrupts from user space by writing to /dev/uioX, you should implement this function. The parameter irq_on will be 0 to disable interrupts and 1 to enable them.

```
static struct uio_info blinker_uio_info = {
	.name = "blinker_uio_module",
	.version = "1.0",
	.irq_flags = 0,
	.handler = blinker_uio_interrupt_handler,
	.open = blinker_uio_open,
	.release = blinker_uio_release,
	.irqcontrol = blinker_uio_irqcontrol,
};
```

As for the probe() function it is very similar to another kind of drivers we have already developed. It is necessary to get the memory resources, the interrupt and the clock. Then, this function remaps the io memory occupied by the internal registers of the blinker device to kernel memory, later the driver will put at user space applications disposal this map of memory. What can be seen in the piece of code shown below is how to register in the uio infrastructure this memory region in order to map it to user space. Even if it does not happen with our blinker peripheral, sometimes devices have more than one memory region to

be mapped to user space. If such a case, for each region, the probe() function have to set up a struct uio_mem in the mem[] array. Here is a description of the fields of the uio_mem structure:

- ✓ *char *name*
  [Optional]. A string identifier for this mapping, though the string can be empty.
- ✓ *int memtype*
  [Required]. Set this to UIO_MEM_PHYS if you have physical memory on your device to be mapped. Use UIO_MEM_LOGICAL for logical memory, allocated with kmalloc, for example. There is also UIO_MEM_VIRTUAL to map virtual memory.
- ✓ *unsigned long addr*
  [Required]. The address of memory that can be mapped. This address is the one that will appear in sysfs.
- ✓ *unsigned long size*
  The size, in bytes, of the memory pointed to by addr. You must initialize size with zero for all unused mappings.
- ✓ *void *internal_addr*
  If you want to have access to this memory region from within your kernel code, you must map it internally by using ioremap(). Addresses returned by this function cannot be mapped to user space. You must use internal_addr instead of addr to store such address.

These attributes appear under the /sys/class/uio/uioX directory. Though you have to be aware that this directory might be a symlink, and not a real directory. Each mapping will have its own directory in sys/class/uio/uioX directory, the first mapping will appear as /map0, and subsequent mappings will create directories /map1, /map2, and so on. All of these directories will contain four read-only files that show the attributes of the memory: *name*, *addr*, *size*, as were established in the uio_mem structure, and *offset*, which represents, in bytes, the offset that hat has to be added to the pointer returned by mmap() to get the actual device memory. This is important if the device's memory is not page aligned. Remember that pointers returned by mmap() are always page aligned. So, from user space, the different mappings are distinguished by adjusting the offset parameter of the mmap() call. To map the memory of mapping N, you have to use N times the page size as your offset, offset = N * getpagesize();

```
static int platform_probe(struct platform_device *pdev)
{
	--------------------------
	/* acquire the probe lock */
	--------------------------
	/* get memory resource */
	--------------------------
	/* get interrupt resource */
	--------------------------
	/* get clock resource */
	--------------------------
	/* reserve memory region */
	--------------------------
	/* ioremap memory region */
	--------------------------

	/* initialize uio_info struct uio_mem array */
	blinker_uio_info.mem[0].memtype = UIO_MEM_PHYS;
	blinker_uio_info.mem[0].addr = r->start;
	if (resource_size(r) > PAGE_SIZE) blinker_uio_info.mem[0].size = resource_size(r);
	else blinker_uio_info.mem[0].size = PAGE_SIZE;
	blinker_uio_info.mem[0].name = "blinker_uio_driver_hw_region";
	blinker_uio_info.mem[0].internal_addr = g_ioremap_addr;

	for (i = 1; i < MAX_UIO_MAPS; i++)
		blinker_uio_info.mem[i].size = 0;

	/* initialize uio_info irq */
	blinker_uio_info.irq = g_blinker_driver_irq;
```

```
	/* register the uio device */
	sema_init(&g_blinker_uio_dev_sem, 1);
	ret_val = uio_register_device(&pdev->dev, &blinker_uio_info);
	if (ret_val != 0) {
		pr_warn("Could not register device \"blinker_uio\"...");
		goto bad_exit_iounmap;
	}

	---------------------------
}


static int platform_remove(struct platform_device *pdev)
{
	uio_unregister_device(&blinker_uio_info);

	---------------------------
}

static struct of_device_id blinker_driver_dt_ids[] = {
	{
	 .compatible = "blinker,driver-1.0"},
	{ /* end of table */ }
};
```

The rest of the functions of the platform device driver and the definition structure are quite the same as before.

7.3.2 Making a UIO enabled Kernel. Adding *blinker_uio_module* as a built-in module

You already know how to build a driver as a loadable module for the kernel. This time we will show you how to compile the kernel and install it as a built-in module for the kernel, so it forms part of the kernel itself, without the necessity of loading it with modprobe. Having the driver as part of the kernel has the advantage of being able to use the driver at the very start of the system, now the kernel is put into execution. But it has also backfalls, as for example, not being able of unloading the module without having to rebuild the kernel. It is up to you to evaluate the pros and cons and select one of the two choices.

The process of creation of a built-in module is very similar to the one described in previous sections. It is only required to have into account some differences. In this occasion, the sources have to be copied into the /drivers/uio directory, and you should use this information for the makefile and the kconfig files of this directory.

```
config UIO_BLINKER
     tristate "blinker user space input output (UIO) module"
     default n
     help
       Select this configuration to enable the blinker_uio
       Don't select as buil-in together with blinker_misc or
       blinker_platform. Only one of then can be inserted at one time
```

```
obj-$(CONFIG_UIO_BLINKER)    += blinker_uio_module.o
```

When configuring the kernel prior to enable your uio driver is mandatory to have the kernel enabled for uio devices. You can achieve that by marking the options as shown in the Fig. 120 and Fig. 121 (be aware of the check mark in the selection, instead of the filled square of previous examples miscellaneous and platform-device drivers).

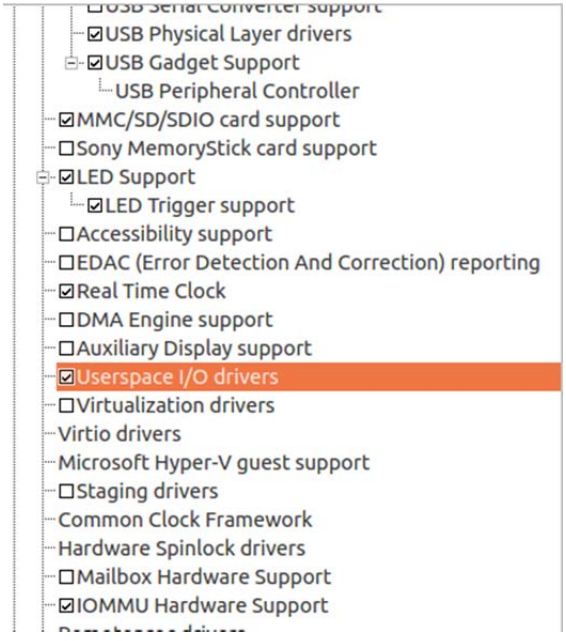


Fig. 120: Electing the UIO drivers in kernel configuration

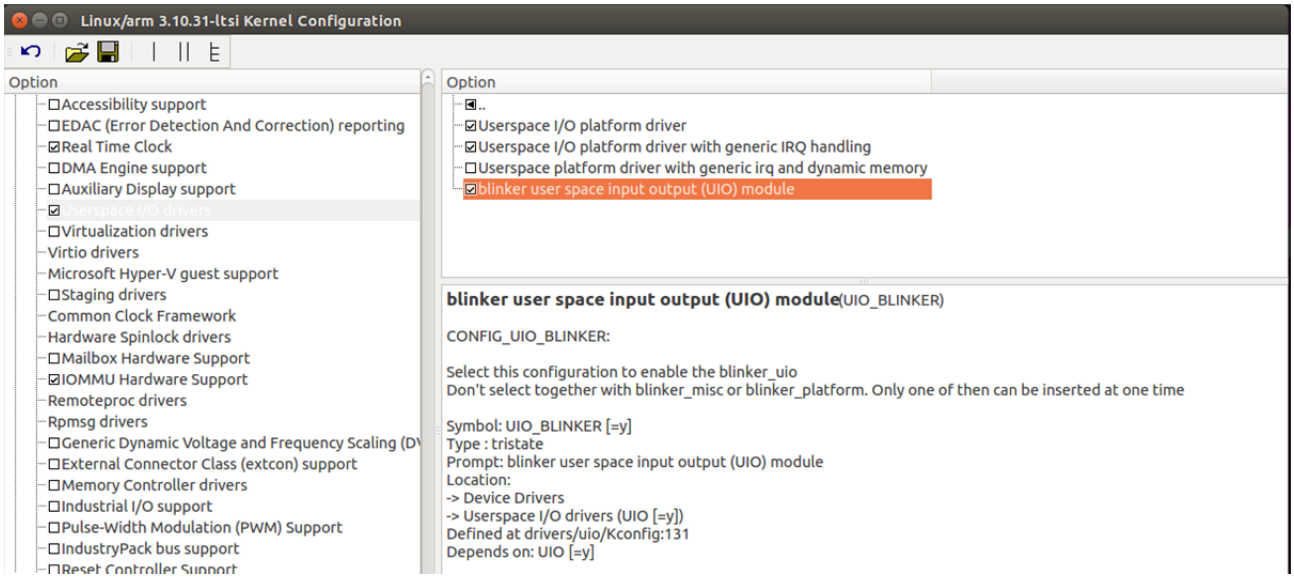


Fig. 121: Details of the selection of the UIO driver

From now on the process is identical as in the previous occasions, the Buildroot commands are exactly the same as before. Once the final SD image has been created, when booting the DE1-SoC your blinker_uio_driver.ko will be part of the kernel and you will not have to load it to start using it from your user space applications.

7.3.3 Testing *blinker_uio_module* from a user space application

This section shows you the way to write a user space application that makes use of the blinker_uio_driver. First, you can see the header file which includes some useful to definitions to test that the application is accessing the correct device in the sysfs. The function in charge of validating the system tries to find matches between this information and the actual attributes read from the /sys/class/uio/uioX. Please note that the latter is only a simlink to the actual directory that resides in "/sys/devices/sopc.0/ff200000.bridge/ff240000.blinker”

blinker_uio_module_test.h

```
// sysfs path to our device and other properties
#define BLINKER_DRIVER_DEV_SYSFS_ENTRY_DIR
"/sys/devices/sopc.0/ff200000.bridge/ff240000.blinker"
#define UIO_NAME "blinker_uio_module"
#define UIO_VERSION "1.0"
#define UIO_MAP_NAME "blinker_uio_driver_hw_region"
#define UIO_MAP_ADDR "0xff240000"
#define UIO_MAP_OFFSET "0x0"
#define UIO_MAP_SIZE "0x1000"
#define DIR_NAME_BUFFER_SIZE     (1024)
#define FILE_NAME_BUFFER_SIZE    (1024)
#define RESPONSE_BUFFER_SIZE     (4096)
#define DEV_NAME_BUFFER_SIZE     (1024) ("1.0");
```

Below you can find some of the functions that comprise the *blinker_uio_module_test* application.

blinker_uio_module_test.c

```c
/*****************************************************************
 * commands functions implementation
 *****************************************************************/

void do_write_config(void *blinker_driver_map){
// TO BE COMPLETED

}

void do_write_speed(void *blinker_driver_map){

// TO BE COMPLETED
}

void do_read_leds(void *blinker_driver_map){

// TO BE COMPLETED
	printf("leds position = %u \n", dato);
}

void do_read_state(void *blinker_driver_map){

// TO BE COMPLETED
	printf("configuration = %u\n", dato & 0x0F);
	printf("speed = %u\n", (dato & 0xF0)>>4);
}

void do_ena_int(void *blinker_driver_map,int uio_fd){
	unsigned long irq_control;

	irq_control = 1;
	write(uio_fd, &irq_control, sizeof(irq_control));
	printf("blinker interrupt enabled\n");

}

void do_dis_int(void *blinker_driver_map,int uio_fd){

// TO BE COMPLETED
	printf("blinker interrupt disabled\n");
}

void do_wait_int(void *blinker_driver_map,int uio_fd){
	unsigned long irq_count;

	// wait for an IRQ event
	printf("Waiting for an interrupt\n");

	read(uio_fd, &irq_count, 4);

	do_read_state(blinker_driver_map);

	printf("%lu interrupts received\n", irq_count);

}

void do_help(void) {
	puts(HELP_STR);
	puts(USAGE_STR);
}
```

blinker_uio_module_test.c (continuation)

```
/*****************************************************************
// This function attempts to validate some of the features that we expect to
// see in the system that we are running on.  Primarily this is an attempt to
// validate that the macros that we've defined for  BLINKER_DRIVER_PHYS_BASE and
// BLINKER_DRIVER_FREQ are accurate.  We do this by checking to see if the
// expected entries in the procfs and sysfs exist for our device.  Then we
// attempt to verify that the clocks setting for our device in the device-tree
// matches the clock that we expect to be driving it.
 *****************************************************************/
void validate_system_features(void) {
	DIR *dp;
	const char *dirname;
	int fd;
	char *filename;
	struct dirent *dir_entry;
	int found_uio_entry;
	int result;
	char dir_name_buffer[DIR_NAME_BUFFER_SIZE];
	char file_name_buffer[FILE_NAME_BUFFER_SIZE];
	char response_buffer[RESPONSE_BUFFER_SIZE];
	char *newline;

	// test to see that the blinker_driver device entry exists in the sysfs
	dirname = BLINKER_DRIVER_DEV_SYSFS_ENTRY_DIR;
	dp = opendir(dirname);
	if(dp == NULL) {
		perror("error opening blinker sysfs directory");
		exit(EXIT_FAILURE);
	}
	if(closedir(dp)) {
		perror("error closing directory");
		exit(EXIT_FAILURE);
	}

	// look for a uio entry
	strncpy(dir_name_buffer, BLINKER_DRIVER_DEV_SYSFS_ENTRY_DIR,
			DIR_NAME_BUFFER_SIZE);
	strncat(dir_name_buffer, "/uio", DIR_NAME_BUFFER_SIZE);
	dirname = dir_name_buffer;
	dp = opendir(dirname);

	if(dp == NULL) {
		perror("error opening /uio directory");
		exit(EXIT_FAILURE);
	}
	found_uio_entry = 0;

	do {
		dir_entry = readdir(dp);
		if(dir_entry != NULL) {
			if(dir_entry->d_type == DT_DIR) {
				if(strncmp(dir_entry->d_name, "uio", 3) == 0) {
				strncpy(g_uio_dev_name, dir_entry->d_name, NAME_MAX);
				strncat(dir_name_buffer, "/", DIR_NAME_BUFFER_SIZE);
				 strncat(dir_name_buffer, dir_entry->d_name,
				     DIR_NAME_BUFFER_SIZE);
				found_uio_entry = 1;
				break;
				}
			}
		}
	} while(dir_entry != NULL);

	if(closedir(dp)) {
		perror("error closing directory");
		exit(EXIT_FAILURE);
	}
```

blinker_uio_module_test.c (continuation)

```
        if(found_uio_entry == 0) {
                perror("unable to locate uio entry for device");
                exit(EXIT_FAILURE);
        }
        // test the uio device name
        strncpy(file_name_buffer, dir_name_buffer, FILE_NAME_BUFFER_SIZE);
        strncat(file_name_buffer, "/name", FILE_NAME_BUFFER_SIZE);
        filename = file_name_buffer;
        fd = open(filename, O_RDONLY);

        if(fd < 0) {
                perror("error opening /name file ");
                exit(EXIT_FAILURE);
        }
        result = read(fd, response_buffer, RESPONSE_BUFFER_SIZE);

        if(result < 0) {
                perror("failed to read from /name file");
                close(fd);
                exit(EXIT_FAILURE);
        }

        if(close(fd)) {
                perror("error closing /name file");
                exit(EXIT_FAILURE);
        }

        newline = strchr(response_buffer, '\n');

        if(newline != NULL)
                *newline = '\0';
        if(strcmp(response_buffer, UIO_NAME) != 0) {
                perror("uio name does not match expected");
                exit(EXIT_FAILURE);
        }

        // test the uio device version
        strncpy(file_name_buffer, dir_name_buffer, FILE_NAME_BUFFER_SIZE);
        strncat(file_name_buffer, "/version", FILE_NAME_BUFFER_SIZE);
        filename = file_name_buffer;
        fd = open(filename, O_RDONLY);

        if(fd < 0) {
                perror("error opening version file ");
                exit(EXIT_FAILURE);
        }
        result = read(fd, response_buffer, RESPONSE_BUFFER_SIZE);

        if(result < 0) {
                perror("failed to read from /version file");
                close(fd);
                exit(EXIT_FAILURE);
        }

        if(close(fd)) {
                perror("error closing /version file");
                exit(EXIT_FAILURE);
        }

        newline = strchr(response_buffer, '\n');

        if(newline != NULL)
                *newline = '\0';
        if(strcmp(response_buffer, UIO_VERSION) != 0) {
                perror("uio version does not match expected");
                exit(EXIT_FAILURE);
        }
```

blinker_uio_module_test.c (continuation)

```
        // test the uio device maps name
        strncpy(file_name_buffer, dir_name_buffer, FILE_NAME_BUFFER_SIZE);
        strncat(file_name_buffer, "/maps/map0/name", FILE_NAME_BUFFER_SIZE);
        filename = file_name_buffer;
        fd = open(filename, O_RDONLY);

        if(fd < 0) {
               perror("error opening /name file ");
               exit(EXIT_FAILURE);
        }
        result = read(fd, response_buffer, RESPONSE_BUFFER_SIZE);

        if(result < 0) {
               perror("failed to read from /name file");
               close(fd);
               exit(EXIT_FAILURE);
        }

        if(close(fd)) {
               perror("error closing /name");
               exit(EXIT_FAILURE);
        }
        newline = strchr(response_buffer, '\n');

        if(newline != NULL)
               *newline = '\0';
        if(strcmp(response_buffer, UIO_MAP_NAME) != 0) {
               perror("uio map name does not match expected");
               exit(EXIT_FAILURE);
        }

        // test the uio device maps addr
        strncpy(file_name_buffer, dir_name_buffer, FILE_NAME_BUFFER_SIZE);
        strncat(file_name_buffer, "/maps/map0/addr", FILE_NAME_BUFFER_SIZE);
        filename = file_name_buffer;
        fd = open(filename, O_RDONLY);

        if(fd < 0) {
               perror("error opening /addr file ");
               exit(EXIT_FAILURE);
        }
        result = read(fd, response_buffer, RESPONSE_BUFFER_SIZE);

        if(result < 0) {
               perror("failed to read from /address file");
               close(fd);
               exit(EXIT_FAILURE);
        }

        if(close(fd)) {
               perror("error closing /address");
               exit(EXIT_FAILURE);
        }

        newline = strchr(response_buffer, '\n');

        if(newline != NULL)
               *newline = '\0';

        if(strcmp(response_buffer, UIO_MAP_ADDR) != 0) {
               perror("uio map addr does not match expected");
               exit(EXIT_FAILURE);
        }
```

blinker_uio_module_test.c (continuation)

```
        // test the uio device maps offset
        strncpy(file_name_buffer, dir_name_buffer, FILE_NAME_BUFFER_SIZE);
        strncat(file_name_buffer, "/maps/map0/offset", FILE_NAME_BUFFER_SIZE);
        filename = file_name_buffer;
        fd = open(filename, O_RDONLY);

        if(fd < 0) {
                perror("error opening /offset file ");
                exit(EXIT_FAILURE);
        }
        result = read(fd, response_buffer, RESPONSE_BUFFER_SIZE);

        if(result < 0) {
                perror("failed to read from /offset file");
                close(fd);
                exit(EXIT_FAILURE);
        }

        if(close(fd)) {
                perror("error closing /offset file");
                exit(EXIT_FAILURE);
        }

        newline = strchr(response_buffer, '\n');

        if(newline != NULL)
                *newline = '\0';
        if(strcmp(response_buffer, UIO_MAP_OFFSET) != 0) {
                perror("uio map offset does not match expected");
                exit(EXIT_FAILURE);
        }

        // test the uio device maps size
        strncpy(file_name_buffer, dir_name_buffer, FILE_NAME_BUFFER_SIZE);
        strncat(file_name_buffer, "/maps/map0/size", FILE_NAME_BUFFER_SIZE);
        filename = file_name_buffer;
        fd = open(filename, O_RDONLY);

        if(fd < 0) {
                perror("error opening /size file ");
                exit(EXIT_FAILURE);
        }

        result = read(fd, response_buffer, RESPONSE_BUFFER_SIZE);

        if(result < 0) {
                perror("failed to read from /size file");
                close(fd);
                exit(EXIT_FAILURE);
        }

        if(close(fd)) {
                perror("error closing /size file");
                exit(EXIT_FAILURE);
        }

        newline = strchr(response_buffer, '\n');

        if(newline != NULL)
                *newline = '\0';
        if(strcmp(response_buffer, UIO_MAP_SIZE) != 0) {
                perror("uio map size does not match expected");
                exit(EXIT_FAILURE);
        }
}
```

blinker_uio_module_test.c (continuation)

```
/*****************************************************************
 * Main program
 *****************************************************************/
int main(int argc, char **argv) {
	int devuio_fd;
	void *blinker_driver_map;
	int result;
	char dev_name[DEV_NAME_BUFFER_SIZE];

	// Validate the features of the actual hardware
	validate_system_features();

	// parse the command line arguments
	parse_cmdline(argc, argv);

	// open() the /dev/uioX device
	strncpy(dev_name, "/dev/", DEV_NAME_BUFFER_SIZE);
	strncat(dev_name, g_uio_dev_name, DEV_NAME_BUFFER_SIZE);
	devuio_fd = open(dev_name, O_RDWR | O_SYNC);
	if(devuio_fd < 0) {
		perror("devuio open");
		exit(EXIT_FAILURE);
	}

	// map the base of the blinker hardware
	blinker_driver_map = mmap(NULL, sysconf(_SC_PAGE_SIZE), PROT_READ|PROT_WRITE,
	    MAP_SHARED, devuio_fd, 0);
	if(blinker_driver_map == MAP_FAILED) {
		perror("devuio mmap");
		close(devuio_fd);
		exit(EXIT_FAILURE);
	}


	// perform the operation selected by the command line arguments
	if(command_write_config    != NULL) do_write_config(blinker_driver_map);
	if(command_write_speed     != NULL) do_write_speed(blinker_driver_map);
	if(command_read_state      != NULL) do_read_state(blinker_driver_map);
	if(command_read_leds       != NULL) do_read_leds(blinker_driver_map);
	if(command_wait_int        != NULL) do_wait_int(blinker_driver_map, devuio_fd);
	if(command_ena_int         != NULL) do_ena_int(blinker_driver_map,devuio_fd);
	if(command_dis_int         != NULL) do_dis_int(blinker_driver_map, devuio_fd);
	if(command_help                  != NULL) do_help();

	// unmap the blinker and close the /dev/uiox device
	result = munmap(blinker_driver_map, sysconf(_SC_PAGE_SIZE));
	if(result < 0) {
		perror("devuio munmap");
		close(devuio_fd);
		exit(EXIT_FAILURE);
	}

	close(devuio_fd);
	exit(EXIT_SUCCESS);
}
```

Starting with the validate_system_features we have to take into account that information about all UIO devices is available in *sysfs*, so the first thing the driver must do is to find this directory entry what will be a clue that the device driver is loaded. Then, it should check the name and version saved in this directory, in the corresponding files, to make sure it is talking to the right device. After that, it should also verify that the memory mapping exists and has the proper size.
After these verifications all that is left is to map the device's memory to user space by calling mmap(). Provided that the blinker driver has just one mapping there is no need to use any offset. After you successfully mapped your device's memory, you can access it with an ordinary pointer. After that, the application can do the operation requested by the user call. Options requiring an ordinary read or write from/into the internal registers do not differ from the use of pointers accomplished by previous applications, so the student will have to fill this functions with the pertinent code. With regard to the *ena_int* and *dis_int* options, an example of the use of the *irq_control* system call is provided.

Finally, regarding the *wait_int* option, you must remember that a read() to the /dev/uioX blocks until an interrupt occurs. There is only one legal value for the count parameter of read(), and that is 4. Any other value for count causes read() to fail. This function returns a signed 32 bit integer which represents the interrupt count of your device, since the driver loading. You can check this value in order to get useful information, such as, if the value is one more than the value you read the last time, everything was fine. But, if the difference is greater than one, that hints that you missed some interrupts.

7.3.1 Exercise 8: Development of the blinker UIO driver and the user space test application.

In this exercise, you will fill the missing code in the blinker_uio_module.c file, and complete the code in the blinker_uio_module_test.c file consisting of a user space application that must implement the following set of commands:

```
#define HELP_STR "\
\n\
Only one of the following options may be passed in per invocation:\n\
\n\
  -d, --write_speed speed_value\n\
Set the speed of the blinking.\n\
      15:  maximum speed.\n\
      1: minimum speed.\n\
\n\
  -c, --write_config config_value\n\
Set the mode of operation of the blinking.\n\
bit pattern: 000000ms.\n\
      m: mode of operation 0 (blinker) 1 (switches blinking).\n\
      s: 0 (stopped) 1 (running).\n\
\n\
  -s, --read_state\n\
Read the current configuration and speed.\n\
\n\
  -e, --ena_int\n\
Enable interrupt.\n\
\n\
  -i, --dis_int\n\
Disable interrupt.\n\
\n\
  -l, --read_leds\n\
Read the current leds position.\n\
\n\
  -w, --wait_int\n\
Wait for the user to push some button and show the new configuration.\n\
\n\
  -h, --help\n\
Display this help message.\n\
\n\
"
```

8 PREPARING THE LINUX VIRTUAL MACHINE.

8.1 Download VMware Workstation Player.

The link https://www.vmware.com/support/pubs/player_pubs.html contains documentation describing the installation and basic use of VMware Workstation Player. Follow the instructions to setup the application on your computer.

8.2 Installing Ubuntu 14.04 LTS as a virtual machine.

[Ubuntu version]: It is mandatory to install Ubuntu 14.04 version. 16.04 version will generate compatibility problems.

The first step is to download Ubuntu 14.04 (64 bit PC-AMD64) from Ubuntu website using this link: http://releases.ubuntu.com/14.04/ . You will download an ISO image with this Linux operating System.
Run WMware player and install Ubuntu using the VMWare player instructions. Consider the following when creating the virtual machine: you need at least 150Gbytes of hard disk space (in multiple files), 3GByte of RAM, and if possible 2 processors. The installation time will be half an hour more or less depending of your computer. Moving a virtual machine from one computer to another is a time-consuming task, therefore, take this into account to minimize the development time.

8.3 Installing synaptic

If you need to install software packages, you can do it using the command apt-get. Another alternative process is the use of the synaptic utility. To use it you need to install it using this command:

```
$ sudo apt-get install synaptic
```

Once installed you can search and execute the synaptic program. When you click two times over the package, it will show all the dependent packages than would be installed.

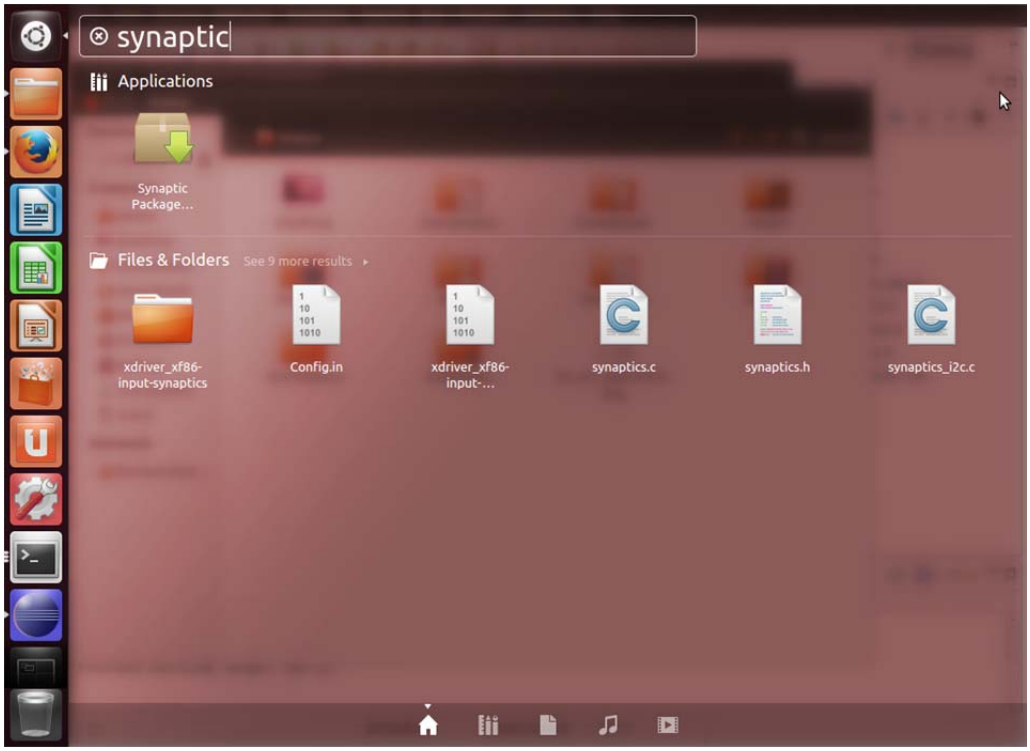


Fig. 122: Synaptic program from Dash

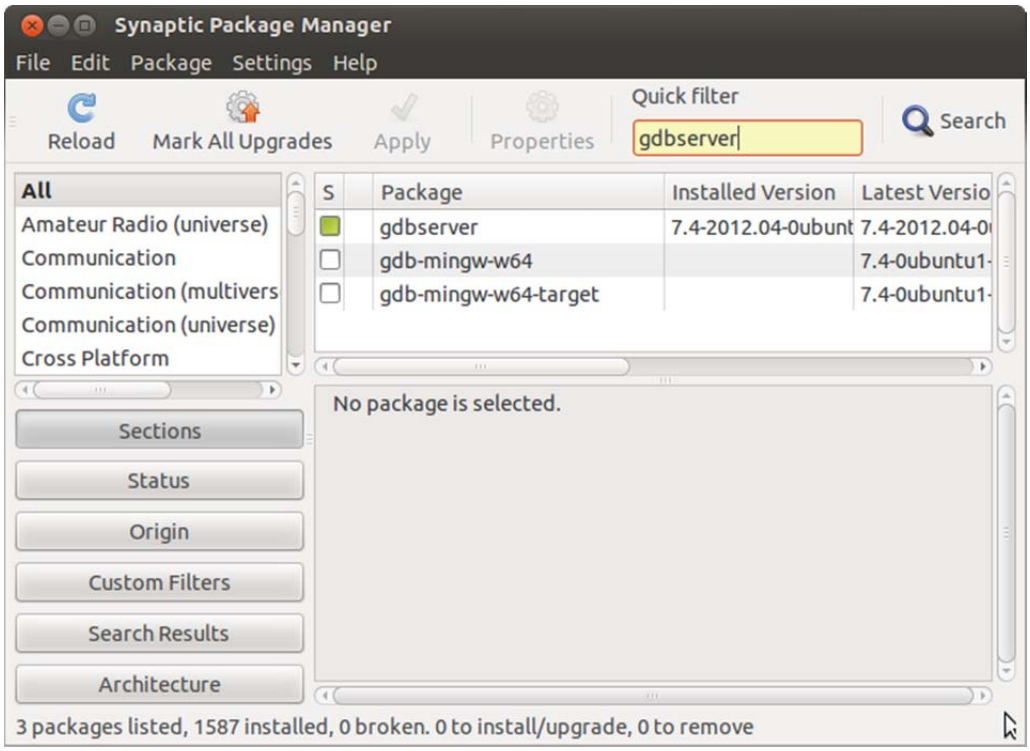


Fig. 123: Synaptic windows

8.4 Installing putty

You need to install:

- putty

8.5 Installing packages for supporting Buildroot.

Using buildroot requires some software packages that have to be installed in the VM. These are listed in this link http://buildroot.uclibc.org/downloads/manual/manual.html#requirement. You need to install:

- g++
- libqt-4-dev
- git

8.6 Installing packages supporting Eclipse

You need to install:

- eclipse-cdt (eclipse C/C++ programming)
- eclipse-rse (eclipse remote explorer)
- eclipse-cdt-launch-remote (eclipse for remote debugging)

If the there is some problems with the remote C/C++ editor you should change the default editor option for .c, .h and .c++ files:
window->preferences->general->editors->file Associations
"add" button for ".c", ".h" y ".c++", select C/C++ editor, and hit "default" button

8.7 Installing Altera design tools in Linux

To install quartus download "Quartus-lite-16.0.0.211-linux.tar", extract and execute the script “setup.sh”. In some versions of Ubuntu some libraries present problems, therefore it is necessary to carry out the following changes to the default installation:

a) Include this line in the ".bashrc" file of your home directory.

export LD_LIBRARY_PATH=~/altera/16.0/quartus/linux64${LD_LIBRARY_PATH:+:$LD_LIBRARY_PATH}

b) To remove the incompatible libraries, keeping a backup copy, and to install the compatible ones, enter these commands in the quartus installation ditectory (~/altera/16.0/quartus/linux64/).

```
#mv crashreporter crashreporter_backup
#mv crashreporter.dep crashreporter.dep_backup
#mv libcurl.so.4 libcurl.so.4_backup
#mv libccl_curl_drl.so libccl_curl_drl.so_backup
#mv libssl.so.1.0.0 libssl.so.1.0.0_backup
#mv libcrypto.so.1.0.0 libcrypto.so.1.0.0_backup
#sudo apt install libcrypto++9 libssl-dev libcurl3
```

To install the USB Blaster Driver you have to add a new file /etc/udev/rules.d/92-usbblaster.rules file. With this content

```
# USB-Blaster
SUBSYSTEM=="usb", ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6001",
MODE="0666"
SUBSYSTEM=="usb", ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6002",
MODE="0666"

SUBSYSTEM=="usb", ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6003",
MODE="0666"

# USB-Blaster II
SUBSYSTEM=="usb", ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6010",
MODE="0666"
SUBSYSTEM=="usb", ATTRS{idVendor}=="09fb", ATTRS{idProduct}=="6810",
MODE="0666"
```

Then reboot linux

Finally, we need to install the SoC Embedded Design Suite. Download "SoCEDSSetup-16.0.0.211-linux.run" From the Intel-FPGA downloads web page. After the download is finished execute the ".run" file. Being the 64 bit version DS-5 ARM compiler will crash for missing dependencies. To solve this matter, execute as sudo user the "ds-deps-ubuntu_64.sh" script (available in the ARM web page) to install the missing modules. If the ".run" was executed as a normal user (not sudo) is mandatory to run the "run_post_install_for_ARM_DS-5_v5.23.1.sh" script in the installation directory of DS-5.

9 TYPICAL UBUNTU VIRTUAL MACHINE PROBLEMS

9.1 Manually installation of VMware client tools in a Linux Virtual Machine.

Check the tag “Manually Install or Upgrade VMware Tools in a Linux Virtual Machine” in VMWare player help file to see how to install client tools.

9.2 Ubuntu presents a black screen after graphical login

http://www.ubuntugeek.com/ubuntu-tiphow-to-removeinstall-and-reconfigure-xorg-without-reinstalling-ubuntu.html

9.3 Ubuntu is using us keyboard and not a Spanish one.

In a terminal window change the keyboard with this command: loadkeys es or Use Configuration->text-entry-> add Spanish.

9.4 Opening terminals when navigating with nautilus.

Using synaptic install:

- nautilus-open-terminal