DE0-Nano-Soc Project Creation

Introduction

This document describes how to setup an HPS Quartus and ARM DS-5 project for DE0-Nano-Soc board. You will then be able to use it as a base for custom design on your board.

This reference design is made for Quartus Prime Lite 18.1 et SocEDS 18.1 with ARM DS-5 installed from SocEDS 18.1.

Be sure to have installed Quartus and SocEDS in your Ubuntu 18.04 (64 bits) machine before continuing.

This project exposes how to create a HPS + FPGA system which uses peripherals through the FPGA, in this case PIOs.

The main system to install on the board is the Debian 10 provided on Digikey website. Follow the procedure to get your system running with latest 4.14.130 kernel and debian rootfs.

DE0 Nano Soc linux system install
https://www.digikey.com/eewiki/display/linuxonarm/DE0-Nano-SoC+Kit

Setting up the project

Cloning the repository

Start by cloning the template project from github into your QuartusProject directory:

```
cd ~/QuartusProjects
git clone https://github.com/lochej/DE0_HPS_Example.git
cd DE0_HPS_Example
mkdir sw
mkdir hw
```

The sub-directories **sw** and **hw** will hold your application.

All the specific hardware Verilog files go into the **hw** folder.
The software ARM DS-5 project and HPS linux files will go into you **sw** folder.

This project contains folder contains a Quartus Project file and the necessary files to synthesis the hardware for DE0-Nano-Soc board with HPS.

Preparing the Quartus Project

Open Quartus Prime Lite 18.1 and open the **DE0_HPS_Example.qpf** file.

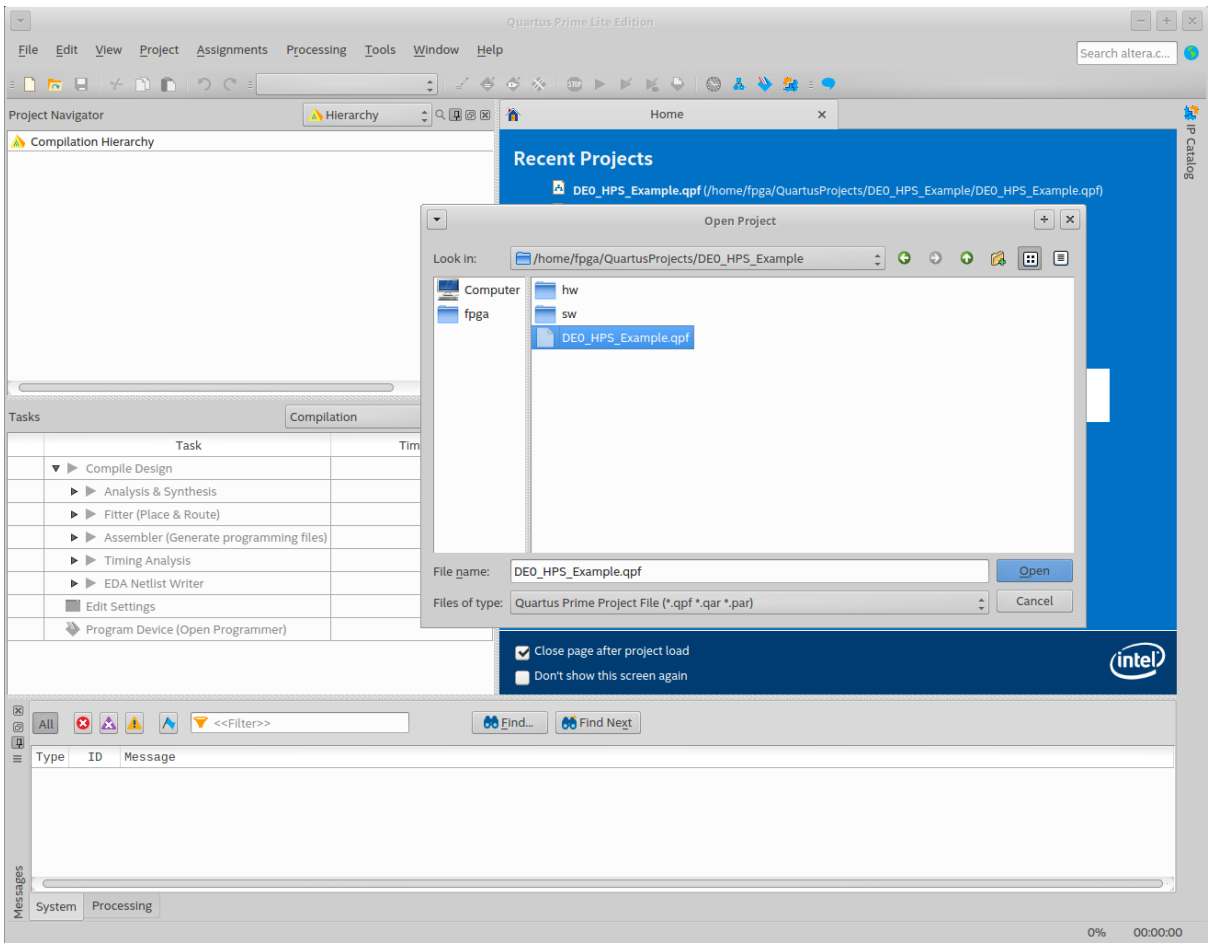

Figure 1 : Open the Quartus Project

Once the project is loaded, open **Qsys/Platform Designer** under **Tools->Platform Designer.**
Load the **DE0_HPS_Example.qsys** file.

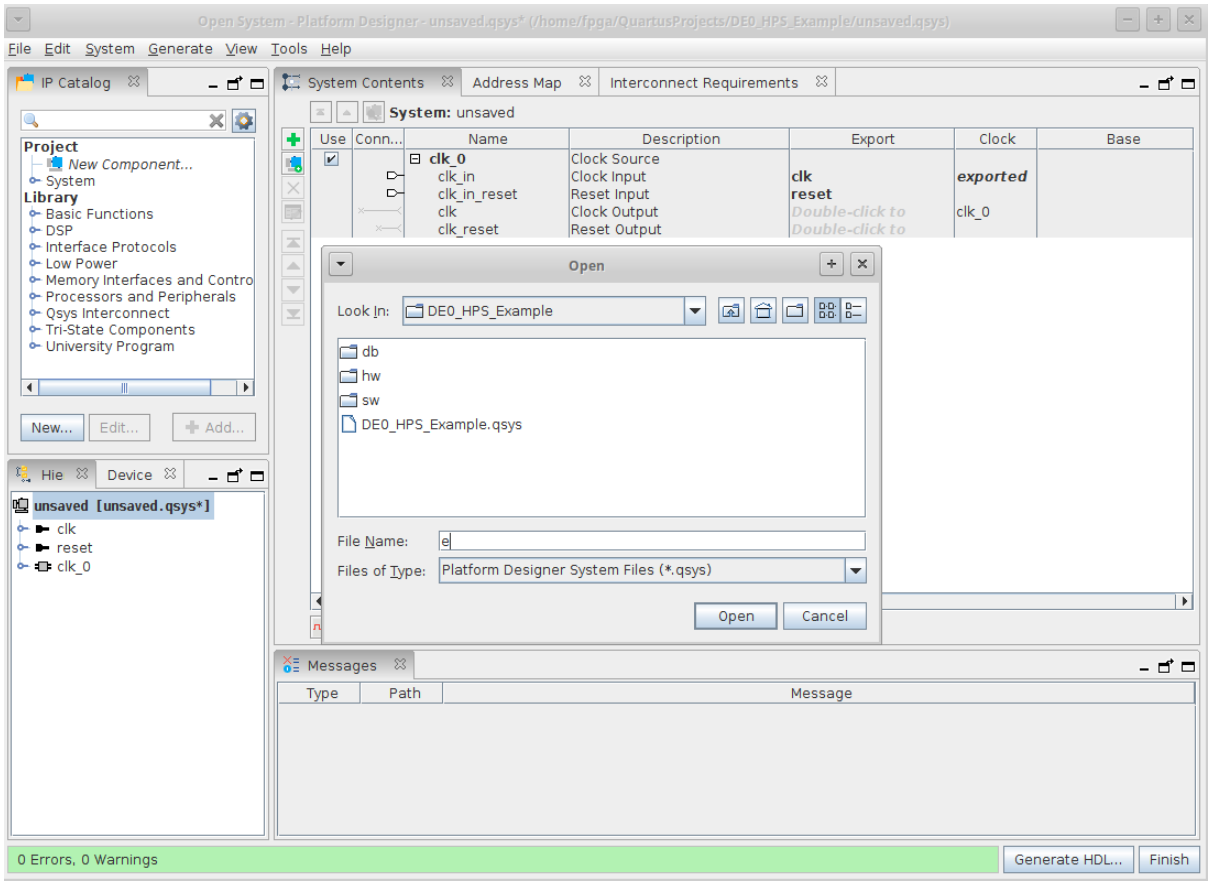

Figure 2 : Open the Qsys file

Once the Qsys project is loaded, click **Generate HDL** in the bottom right corner and select **Verilog** as HDL language.

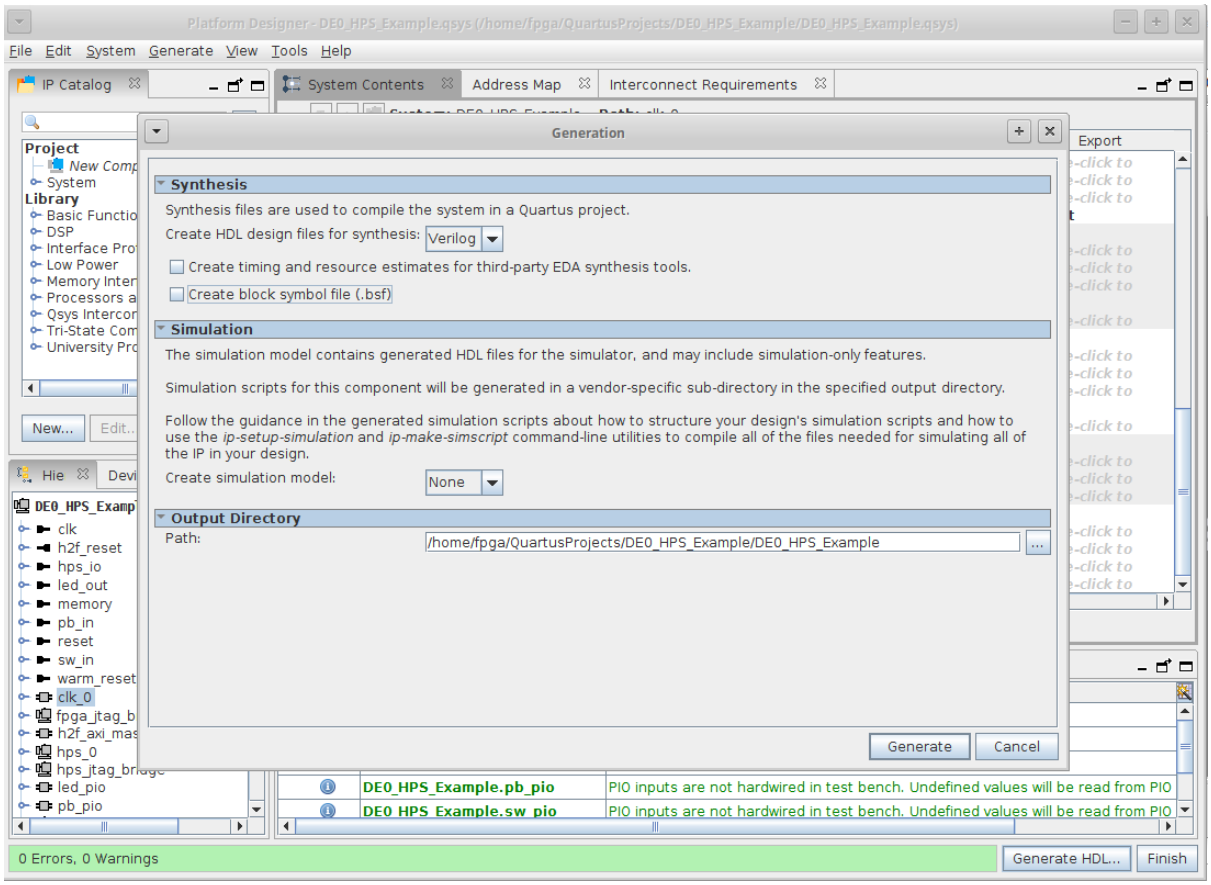

Figure 3 : Qsys Generate project

When you are done generating the HDL files, just click finish to return to Quartus Prime Lite.

Now Generate the full **Quartus Compilation** using the **CTRL+L key** combination or **Processing-> Start Compilation.**

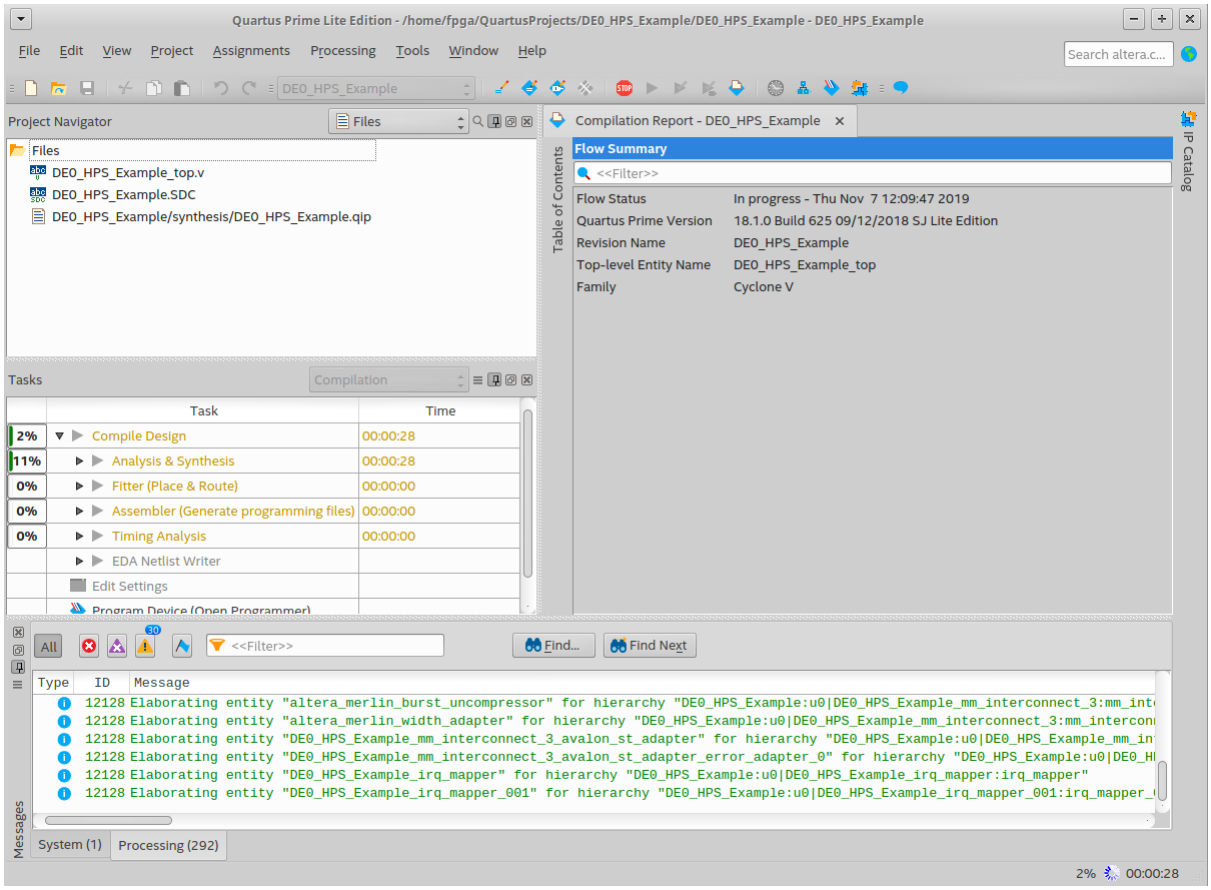

Figure 4 : Quartus Compiling Design

When compilation is finished, we are going to prepare the **rbf** FPGA programming file for use in the Linux file system.

We are going to program the FPGA from Linux using a **Passive Parallel 16 RBF file**.
This is the default configuration of the board with all **MSEL** buttons to '1' (pointing towards GPIO_0 port).

Convert the SOF file to RBF using the **File->Convert Programming File menu** in Quartus.
In the opened window, select **Raw Binary File (.rbf)** in the **Programming file type** selector.

Rename the file **output_file.rbf** to **fpga_firmware.rbf**.

On the right of the **Input files to convert** section, hit the **Add file** button and select your **DE0_HPS_Example.sof** file.

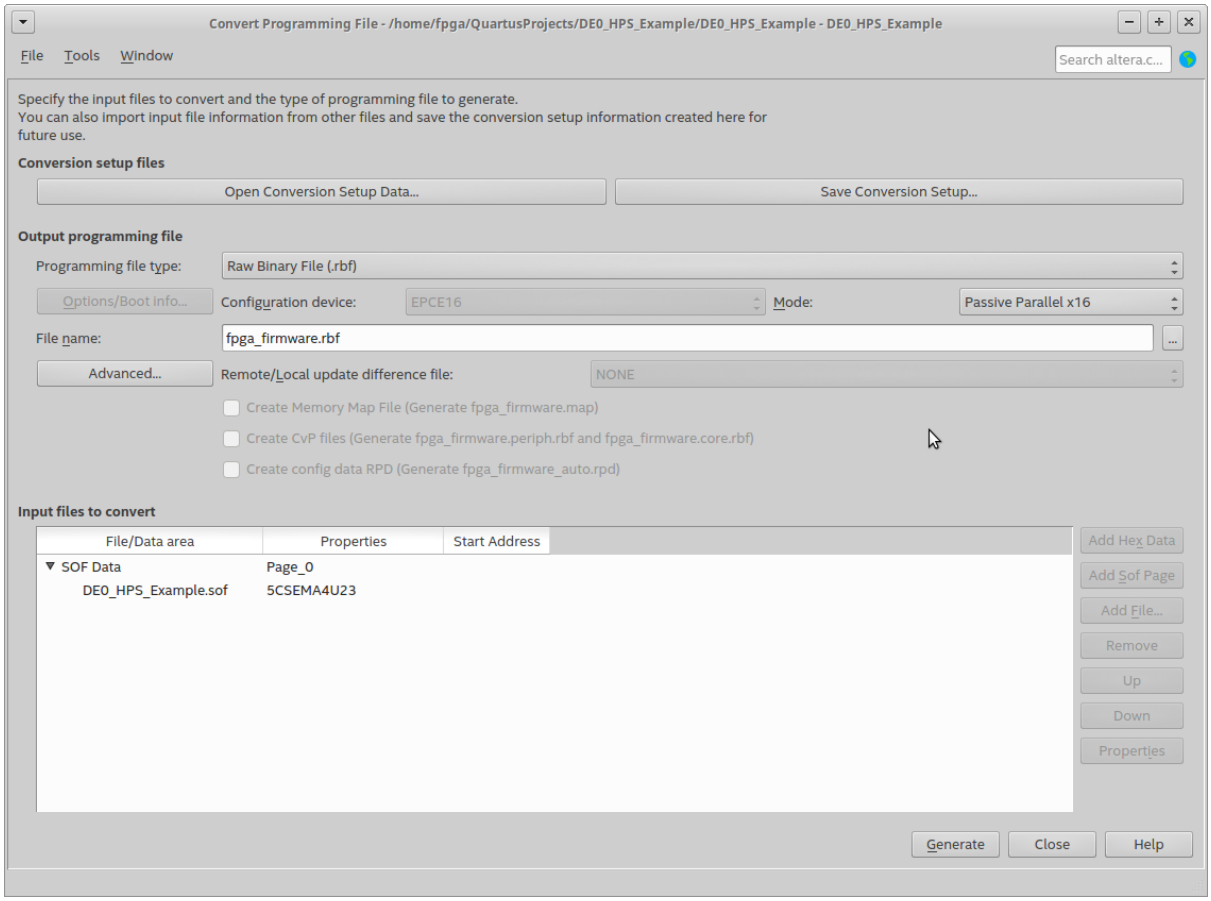

Figure 5 : Converting programming file to RBF PP16

We are done with the Quartus Hardware compilation and our FPGA files are ready .

We will now follow on with by creating the files for the Linux part.

Creating the Linux Device Tree files and HPS system header files

Why a device-tree ?

Most of the time, you will use standard IP core from Intel FPGA IP Catalog in your Qsys design. For example, there exists **a linux sysfs driver** for *Intel FPGA PI, UART, SPI, I2C and DMA IP cores*. To let Linux kernel know what peripherals are implemented in the FPGA, you need a **custom Device Tree for your design.** The device tree describes the peripherals of your system and where to access them in memory. So when you create components in the FPGA and want them to be accessible from the Linux file system, you will need to specify them in a device-tree.

From Linux Kernel 4.4 and on, the FPGA Manager, FPGA Bridge and FPGA Region drivers have appeared in the linux kernel code. This is particularly true for **Linux 4.14.130 LTSi Kernel**.

This new components implement all you need to flash the FPGA and hot mount new components driver under the Linux file system with the use of **Device Tree Overlays (dtbo).** You don't have to

recompile a new kernel each time you modify your design, just push the device tree overlay to your system and it will load everything dynamically.

You just need to make sure that your HPS linux system kernel has all the peripheral drivers you need or the kernel modules to load them.

This new API has brought many improvements hover past techniques. By using a device-tree overlay, Linux will only probe the drivers when the FPGA is programmed. This prevents the drivers to be loaded and the FPGA not programmed leaving the system in an undefined state…

*So the goal here is to use the information in our Qsys design and the knowledge of our address map to declare our peripherals in a **device tree overlay** to dynamically load our HPS Linux FS and prepare the FPGA.*

Generating the full system device-tree

Quartus offers the **sopc2dts** command line tool which creates a **device-tree source** from you Qsys generated design **DE0_HPS_Example.sopcinfo**.

For this, I have included a friendly script that calls the SocEDS shell for you and generates the device-tree source under **sw/dts/** folder. The name of the script is: **create_linux_files.sh**

This command should be called from SocEDS embedded command shell. The script assumes that you SocEDS shell is installed under **~/intelFPGA_lite/18.1/embedded/embedded_command_shell.sh** replace with your path if different.

Under the hood, he script calls the following command to create the dts file.

```
sopc2dts DE0_HPS_Example.sopcinfo > sw/dts/hps_fpga_device_tree.dts
```

This command creates a full device-tree for the whole system. That's not exactly what we want.

We will just use this device-tree source file to create our own a device-tree overlay which just loads the FPGA part.

Writing your device tree overlay

Now that we have the full system device-tree, we will have a look at how to use it to create our overlay file.

The base template for an FPGA device tree overlay is the following:

FPGA Device tree overlay base template

```
/dts-v1/;
/plugin/;
/ {
        fragment@0 {
                target-path = "/soc/base-fpga-region";
                #address-cells = <1>;
                #size-cells = <1>;
```

```
                __overlay__ {
                        #address-cells = <1>;
                        #size-cells = <1>;
                        firmware-name = "fpga_firmware.rbf";
                };
        };
};
```

In this device tree base overlay, we have just specified that we want our FPGA programmed with the **fpga_firmware.rbf** file.

On your linux FS, you place **fpga_firmware.rbf** under **/lib/firmware/fpga_firmware.rbf.**

This path is the default path to which the linux FS will try to find you firmware file.

The **target-path** indicates where to plug in our device tree overlay in the live tree (the one currently running on the system, loaded on boot). In our case, we specify the **"/soc/base-fpga-region"** which is the region where our FPGA stands.

You can check for your current device tree using by looking at the **/sys/firmware/devicetree/base/soc** folder on your HPS linux system.

```
root@arm:/sys/firmware/devicetree/base/soc# ls
'#address-cells'       gpio@ff70a000      snoop-control-unit@fffec000
 amba                 i2c@ffc04000       spi@ff705000
 base-fpga-region      i2c@ffc05000       spi@fff00000
 can@ffc00000          i2c@ffc06000       spi@fff01000
 can@ffc01000          i2c@ffc07000       sram@ffff0000
 clkmgr@ffd04000       interrupt-parent    sysmgr@ffd08000
 compatible           l2-cache@fffef000   timer0@ffc08000
 device_type          l3regs@0xff800000   timer1@ffc09000
 dwmmc0@ff704000       name               timer2@ffd00000
 eccmgr               nand@ff900000       timer3@ffd01000
 ethernet@ff700000     ranges             timer@fffec600
 ethernet@ff702000     rstmgr@ffd05000     usb@ffb00000
 fpga_bridge@ff400000  sdramedac          usb@ffb40000
 fpga_bridge@ff500000  sdr@ffc25000       usbphy
 fpgamgr@ff706000      serial0@ffc02000    watchdog@ffd02000
 gpio@ff708000         serial1@ffc03000    watchdog@ffd03000
 gpio@ff709000        '#size-cells'
```

Now it is time to add our peripherals to the system. In our case, all PIOs are connected to the **LightWeight HPS to FPGA Bridge.** All the peripherals on this bus will be mapped to an address starting at **0xFF200000**.

So let's use the peripherals instantiation in the generated DTS file to copy and paste them in our device tree overlay.

Extract from sw/dts/hps_fpga_device_tree.dts

```
… //beginning of the device tree file
sopc0: sopc@0 {
                device_type = "soc";
                ranges;
                #address-cells = <1>;
                #size-cells = <1>;
                compatible = "ALTR,avalon", "simple-bus";
                bus-frequency = <0>;

                hps_0_bridges: bridge@0xc0000000 {
                        compatible = "altr,bridge-18.1", "simple-bus";
                        reg = <0xc0000000 0x20000000>,
                                <0xff200000 0x00200000>;
                        reg-names = "axi_h2f", "axi_h2f_lw";
                        #address-cells = <2>;
                        #size-cells = <1>;
                        ranges = <0x00000001 0x00000030 0xff200030 0x00000008>,
                                <0x00000001 0x00000020 0xff200020 0x00000010>,
                                <0x00000001 0x00000010 0xff200010 0x00000010>,
                                <0x00000001 0x00000000 0xff200000 0x00000010>;

                        sysid_qsys_0: sysid@0x100000030 {
                                compatible = "altr,sysid-18.1", "altr,sysid-1.0";
                                reg = <0x00000001 0x00000030 0x00000008>;
                                id = <1193046>;
                                timestamp = <1573124790>;
                        }; //end sysid@0x100000030 (sysid_qsys_0)

                        led_pio: gpio@0x100000020 {
                                compatible = "altr,pio-18.1", "altr,pio-1.0";
                                reg = <0x00000001 0x00000020 0x00000010>;
                                altr,gpio-bank-width = <8>;
                                resetvalue = <0>;
                                #gpio-cells = <2>;
                                gpio-controller;
                        }; //end gpio@0x100000020 (led_pio)

                        sw_pio: gpio@0x100000010 {
                                compatible = "altr,pio-18.1", "altr,pio-1.0";
                                reg = <0x00000001 0x00000010 0x00000010>;
                                interrupt-parent = <&hps_0_arm_gic_0>;
                                interrupts = <0 41 1>;
                                altr,gpio-bank-width = <4>;
                                altr,interrupt-type = <3>;
                                altr,interrupt_type = <3>;
                                edge_type = <2>;
                                level_trigger = <0>;
                                resetvalue = <0>;
                                #gpio-cells = <2>;
                                gpio-controller;
```

```
                        }; //end gpio@0x100000010 (sw_pio)

                        pb_pio: gpio@0x100000000 {
                                compatible = "altr,pio-18.1", "altr,pio-1.0";
                                reg = <0x00000001 0x00000000 0x00000010>;
                                interrupt-parent = <&hps_0_arm_gic_0>;
                                interrupts = <0 40 1>;
                                altr,gpio-bank-width = <2>;
                                altr,interrupt-type = <3>;
                                altr,interrupt_type = <3>;
                                edge_type = <2>;
                                level_trigger = <0>;
                                resetvalue = <0>;
                                #gpio-cells = <2>;
                                gpio-controller;
                        }; //end gpio@0x100000000 (pb_pio)
                }; //end bridge@0xc0000000 (hps_0_bridges)
... //following of the device tree file
```

You need to copy the sections in bold to add your new peripherals to your device tree overlay.

Paste the bold section after the firmware name section. Remember to add the fpga_bridge name that we will be using. It is the lightweight bridge declared as **<&fpga_bridge0>. Beware of the #address-cells** argument.

More info about the phandles of bridges here :

https://github.com/altera-opensource/linux-socfpga/blob/master/arch/arm/boot/dts/socfpga.dtsi

IMPORTANT : The phandle of the interrupt controller on Kernel 4.14.130 is **<&intc>** and not **<&hps_0_arm_gic_0>** . So we have to change is for push button PIOs and Switch PIO !

Resulting Device tree overlay file with our components

```
/dts-v1/;
/plugin/;
/ {
        fragment@0 {
                target-path = "/soc/base-fpga-region";
                #address-cells = <1>;
                #size-cells = <1>;
                __overlay__ {
                        #address-cells = <2>;
                        #size-cells = <1>;

                        firmware-name = "fpga_firmware.rbf";

                        fpga-bridges = <&fpga_bridge0>;

                        ranges = <0x00000001 0x00000030 0xff200030 0x00000008>,
                                 <0x00000001 0x00000020 0xff200020 0x00000010>,
```

```
                        <0x00000001 0x00000010 0xff200010 0x00000010>,
                        <0x00000001 0x00000000 0xff200000 0x00000010>;

                sysid_qsys_0: sysid@0x100000030 {
                        compatible = "altr,sysid-18.1", "altr,sysid-1.0";
                        reg = <0x00000001 0x00000030 0x00000008>;
                        id = <1193046>;
                        timestamp = <1573124790>;
                }; //end sysid@0x100000030 (sysid_qsys_0)

                led_pio: gpio@0x100000020 {
                        compatible = "altr,pio-18.1", "altr,pio-1.0";
                        reg = <0x00000001 0x00000020 0x00000010>;
                        altr,gpio-bank-width = <8>;
                        resetvalue = <0>;
                        #gpio-cells = <2>;
                        gpio-controller;
                }; //end gpio@0x100000020 (led_pio)

                sw_pio: gpio@0x100000010 {
                        compatible = "altr,pio-18.1", "altr,pio-1.0";
                        reg = <0x00000001 0x00000010 0x00000010>;
                        interrupt-parent = <&intc>;
                        interrupts = <0 41 1>;
                        altr,gpio-bank-width = <4>;
                        altr,interrupt-type = <3>;
                        altr,interrupt_type = <3>;
                        edge_type = <2>;
                        level_trigger = <0>;
                        resetvalue = <0>;
                        #gpio-cells = <2>;
                        gpio-controller;
                }; //end gpio@0x100000010 (sw_pio)

                pb_pio: gpio@0x100000000 {
                        compatible = "altr,pio-18.1", "altr,pio-1.0";
                        reg = <0x00000001 0x00000000 0x00000010>;
                        interrupt-parent = <&intc>;
                        interrupts = <0 40 1>;
                        altr,gpio-bank-width = <2>;
                        altr,interrupt-type = <3>;
                        altr,interrupt_type = <3>;
                        edge_type = <2>;
                        level_trigger = <0>;
                        resetvalue = <0>;
                        #gpio-cells = <2>;
                        gpio-controller;
                }; //end gpio@0x100000000 (pb_pio)

        };
};
```

```
};
```

Notice that the **ranges** attribute describes the address map from Qsys to the real FPGA lightweight bridge address map.

Note: As we are using the lightweight bridge which only uses 32 bits addresses, we could totally remove the "0x00000001" on every of our components and switch **#address-cells** to **1.** I don't know why but you can forget about this.

The equivalent device tree overlay that would also work with **#address-cells** to **1 (32 bits)** is the following.

Note: We have to add the **altr,ngpio** attribute to our PIO peripherals to indicate the width of the registers.

FINAL Device tree overlay

```
/dts-v1/;
/plugin/;
/ {
        fragment@0 {
                target-path = "/soc/base-fpga-region";
                #address-cells = <1>;
                #size-cells = <1>;
                __overlay__ {
                        #address-cells = <1>;
                        #size-cells = <1>;

                        firmware-name = "fpga_firmware.rbf";

                        fpga-bridges = <&fpga_bridge0>;

                        ranges = <0x00000030 0xff200030 0x00000008>,
                                 <0x00000020 0xff200020 0x00000010>,
                                 <0x00000010 0xff200010 0x00000010>,
                                 <0x00000000 0xff200000 0x00000010>;

                        sysid_qsys_0: sysid@30 {
                                compatible = "altr,sysid-18.1", "altr,sysid-1.0";
                                reg = <0x30 0x00000008>;
                                id = <1193046>;
                                timestamp = <1573124790>;
                        };

                        led_pio: gpio@20 {
                                compatible = "altr,pio-18.1", "altr,pio-1.0";
                                reg = <0x20 0x00000010>;
                                altr,gpio-bank-width = <8>;
                                altr,ngpio = <8>;
                                resetvalue = <0>;
                                #gpio-cells = <2>;
```

```
                        gpio-controller;
                    };

                    sw_pio: gpio@10 {
                        compatible = "altr,pio-18.1", "altr,pio-1.0";
                        reg = <0x10 0x00000010>;
                        interrupt-parent = <&intc>;
                        interrupts = <0 41 1>;
                        altr,gpio-bank-width = <4>;
                        altr,ngpio = <4>;
                        altr,interrupt-type = <3>;
                        altr,interrupt_type = <3>;
                        edge_type = <2>;
                        level_trigger = <0>;
                        resetvalue = <0>;
                        #gpio-cells = <2>;
                        gpio-controller;
                    };

                    pb_pio: gpio@0 {
                        compatible = "altr,pio-18.1", "altr,pio-1.0";
                        reg = <0x0 0x00000010>;
                        interrupt-parent = <&intc>;
                        interrupts = <0 40 1>;
                        altr,gpio-bank-width = <2>;
                        altr,ngpio = <2>;
                        altr,interrupt-type = <3>;
                        altr,interrupt_type = <3>;
                        edge_type = <2>;
                        level_trigger = <0>;
                        resetvalue = <0>;
                        #gpio-cells = <2>;
                        gpio-controller;
                    };
                };
        };
};
```

This final device tree overlay is completely clean.

Save it with the name ***fpga_overlay.dts.***

Now it makes more sense since the addresses directly correlate with the one we can read in Qsys. I suppose that sopc2dts utility add 0x00000001 and switches the address to 2 cells because it is easier to generate text. ***It also is the correct structure for our Kernel 4.14.130 !***

Our device tree is ready to use in our FPGA system now.

Loading the device tree overlay in Linux

Now that we have our FPGA firmware and device tree source available, we will load them onto the DE0-Nano-Soc board.

Copy the **fpga_firmware.rbf** to the **/lib/firmware** directory on your board.

Install the device-tree-compiler on the board to compile your device tree overlay.

```
sudo apt-get install device-tree-compiler
```

Compile the device tree source on the board using this command

```
dtc -@ -I dts -O dtb fpga_overlay.dts > fpga_overlay.dtbo
```

The device tree overlay blob overlay has been compiled. We can now load the FPGA firmware by mounting the device tree overlay in the system.

```
mkdir /sys/kernel/config/device-tree/overlays/fpga

cat fpga_overlay.dtbo > /sys/kernel/config/device-tree/overlays/fpga/dtbo
```

And we are done, the kernel will print:

fpga_manager fpga0: writing fpga_firmware.rbf to Altera SOCFPGA FPGA Manager

Your FPGA is not fully configured and the PIOs declared in the device-tree overlay have been loaded into your Sysfs GPIO class:

```
root@arm:/sys/class/gpio# ls
export      gpiochip1951  gpiochip1963  gpiochip2019
gpiochip1949  gpiochip1955  gpiochip1990  unexport
```

Look at the **gpiochip1951, gpiochip1949 and gpiochip1955**, these are our **FPGA PIO** !
You can use them as regular sysfs GPIOs in Linux.

1. *gpiochip1949 is the push buttons PIO*
2. *gpiochip1951 is the switches PIO*
3. *gpiochip1955 is the LED PIO*

You can identify the GPIOs with their **ngpio** property.

```
cd /sys/class/gpio
cat gpiochip1949/ngpio #FPGA pio push buttions KEY retuns 2
cat gpiochip1951/ngpio #FPGA pio Switches returns 4
cat gpiochip1955/ngpio # FPGA pio LEDs returns 8
```

Driving the SysFS FPGA GPIOs

Export the GPIO to make it available in sysfs:

```
cd /sys/class/gpio/
echo 1949 > export  #export FPGA push button 0
echo 1950 > export #export FPGA push button 1
cat gpio1949/value #read FPGA push button 0 value
cat gpio1950/value #read FPGA push button 1 value
```

You can do the same for the switches with 1951 to 1951+3.

To write the FPGA LEDs export one of the 1955 to 1955+7 GPIOs and set the direction to **out**.

```
echo 1955 > export  #export FPGA LED 0
echo out > gpio1955/direction #set gpio to output
echo 1 > gpio1955/value #LED should be on !
```

For now we have only used the GPIOs with the driver from Linux sysfs. It will not always be the case and sometimes there won't exist any driver for your custom Avalon component. What Qsys will generate is just the address map for your driver.

Generating the system address map header

When accessing the FPGA peripheral from Linux, you don't always get linux driver for that.

So you need to know the address of your peripherals to access to their register through the HPS to FPGA bridges.

Generate your system header file using the following command:

```
sopc-create-header-files DE0_HPS_example.sopcinfo --single sw/include/hps_soc_system.h --module hps_0
```

This will generate the following header file in **sw/include/hps_soc_system.h**:

HPS Soc system system header file
#ifndef _ALTERA_HPS_SOC_SYSTEM_H_

```
#define _ALTERA_HPS_SOC_SYSTEM_H_

/*
 * This file was automatically generated by the swinfo2header utility.
 *
 * Created from SOPC Builder system 'DE0_HPS_Example' in
 * file './DE0_HPS_Example.sopcinfo'.
 */

/*
 * This file contains macros for module 'hps_0' and devices
 * connected to the following masters:
 *   h2f_axi_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 'pb_pio', class 'altera_avalon_pio'
 * The macros are prefixed with 'PB_PIO_'.
 * The prefix is the slave descriptor.
 */
#define PB_PIO_COMPONENT_TYPE altera_avalon_pio
#define PB_PIO_COMPONENT_NAME pb_pio
#define PB_PIO_BASE 0x0
#define PB_PIO_SPAN 16
#define PB_PIO_END 0xf
#define PB_PIO_IRQ 0
#define PB_PIO_BIT_CLEARING_EDGE_REGISTER 1
#define PB_PIO_BIT_MODIFYING_OUTPUT_REGISTER 0
#define PB_PIO_CAPTURE 1
#define PB_PIO_DATA_WIDTH 2
#define PB_PIO_DO_TEST_BENCH_WIRING 0
#define PB_PIO_DRIVEN_SIM_VALUE 0
#define PB_PIO_EDGE_TYPE ANY
#define PB_PIO_FREQ 50000000
#define PB_PIO_HAS_IN 1
#define PB_PIO_HAS_OUT 0
#define PB_PIO_HAS_TRI 0
#define PB_PIO_IRQ_TYPE EDGE
#define PB_PIO_RESET_VALUE 0

/*
 * Macros for device 'sw_pio', class 'altera_avalon_pio'
 * The macros are prefixed with 'SW_PIO_'.
 * The prefix is the slave descriptor.
 */
```

```
#define SW_PIO_COMPONENT_TYPE altera_avalon_pio
#define SW_PIO_COMPONENT_NAME sw_pio
#define SW_PIO_BASE 0x10
#define SW_PIO_SPAN 16
#define SW_PIO_END 0x1f
#define SW_PIO_IRQ 1
#define SW_PIO_BIT_CLEARING_EDGE_REGISTER 1
#define SW_PIO_BIT_MODIFYING_OUTPUT_REGISTER 0
#define SW_PIO_CAPTURE 1
#define SW_PIO_DATA_WIDTH 4
#define SW_PIO_DO_TEST_BENCH_WIRING 0
#define SW_PIO_DRIVEN_SIM_VALUE 0
#define SW_PIO_EDGE_TYPE ANY
#define SW_PIO_FREQ 50000000
#define SW_PIO_HAS_IN 1
#define SW_PIO_HAS_OUT 0
#define SW_PIO_HAS_TRI 0
#define SW_PIO_IRQ_TYPE EDGE
#define SW_PIO_RESET_VALUE 0

/*
 * Macros for device 'led_pio', class 'altera_avalon_pio'
 * The macros are prefixed with 'LED_PIO_'.
 * The prefix is the slave descriptor.
 */
#define LED_PIO_COMPONENT_TYPE altera_avalon_pio
#define LED_PIO_COMPONENT_NAME led_pio
#define LED_PIO_BASE 0x20
#define LED_PIO_SPAN 16
#define LED_PIO_END 0x2f
#define LED_PIO_BIT_CLEARING_EDGE_REGISTER 0
#define LED_PIO_BIT_MODIFYING_OUTPUT_REGISTER 0
#define LED_PIO_CAPTURE 0
#define LED_PIO_DATA_WIDTH 8
#define LED_PIO_DO_TEST_BENCH_WIRING 0
#define LED_PIO_DRIVEN_SIM_VALUE 0
#define LED_PIO_EDGE_TYPE NONE
#define LED_PIO_FREQ 50000000
#define LED_PIO_HAS_IN 0
#define LED_PIO_HAS_OUT 1
#define LED_PIO_HAS_TRI 0
#define LED_PIO_IRQ_TYPE NONE
#define LED_PIO_RESET_VALUE 0

/*
 * Macros for device 'sysid_qsys_0', class 'altera_avalon_sysid_qsys'
 * The macros are prefixed with 'SYSID_QSYS_0_'.
 * The prefix is the slave descriptor.
 */
#define SYSID_QSYS_0_COMPONENT_TYPE altera_avalon_sysid_qsys
#define SYSID_QSYS_0_COMPONENT_NAME sysid_qsys_0
```

```
#define SYSID_QSYS_0_BASE 0x30
#define SYSID_QSYS_0_SPAN 8
#define SYSID_QSYS_0_END 0x37
#define SYSID_QSYS_0_ID 1193046
#define SYSID_QSYS_0_TIMESTAMP 1573124790


#endif /* _ALTERA_HPS_SOC_SYSTEM_H_ */
```

As you can see, this header file contains the base addresses for each peripheral in the system. Mainly, you will use this information manually access the registers of the peripherals through your own driver.

It will be usefull for the ARM DS-5 project to have this file.

ARM DS-5 is the IDE provided with Quartus SocEDS 18.1 and allows you to use Eclipse as a compiling and development environment.

Create an ARM DS-5 project

Start AMR-DS5 IDE and select your workspace as the **sw** folder of the project.

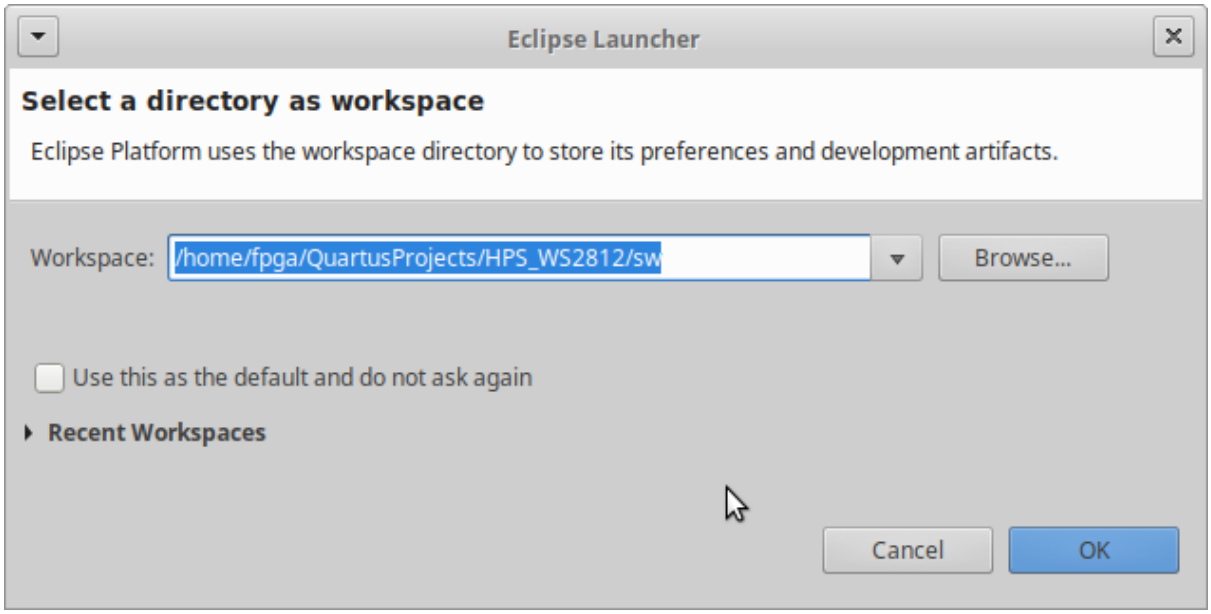

Figure 6 : Selecting Workspace in ARM DS-5

In the example above, replace **HPS_WS2812** by your DE0_HPS_Example project.

Create a new C project.

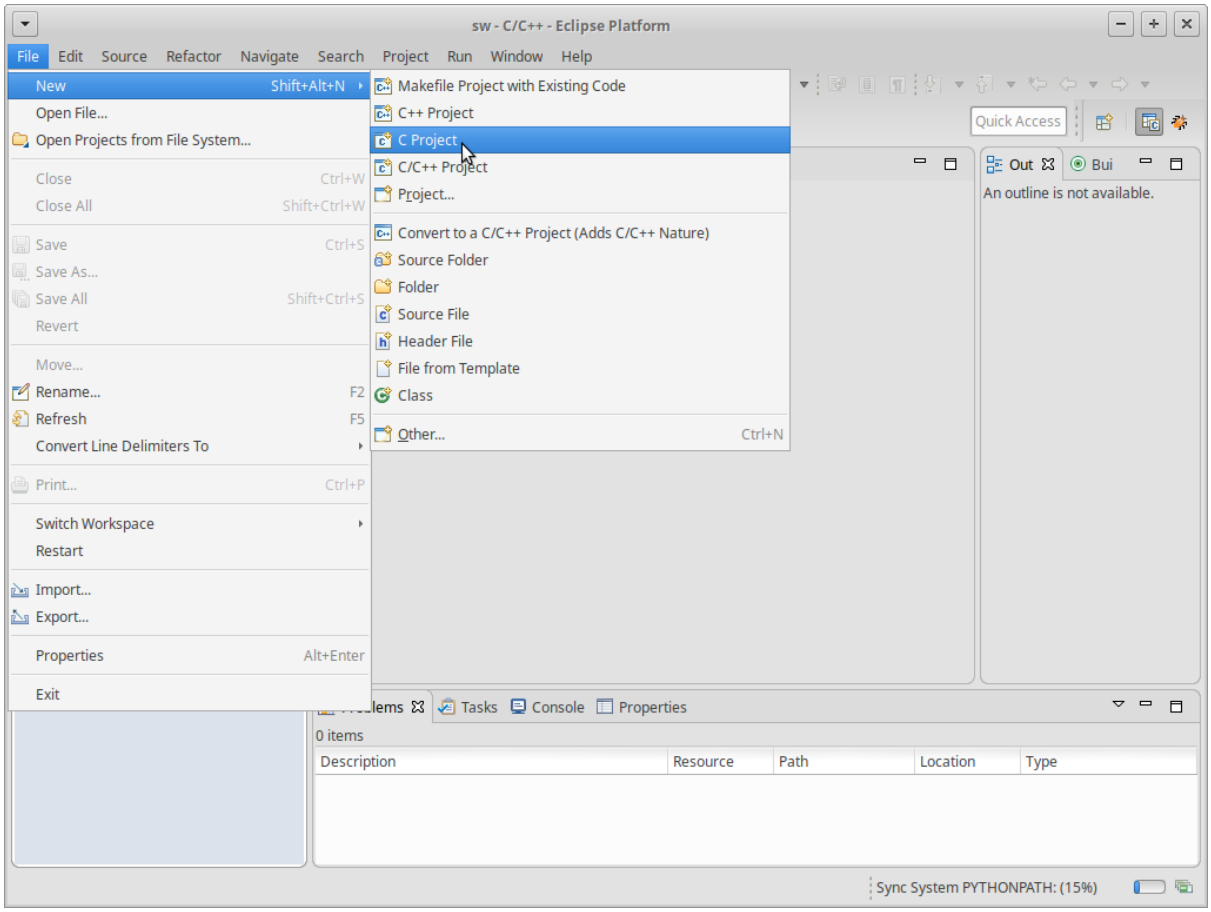

Figure 7: Create a new C project

Then set your project name and select GNU GCC compiler in the project wizard.

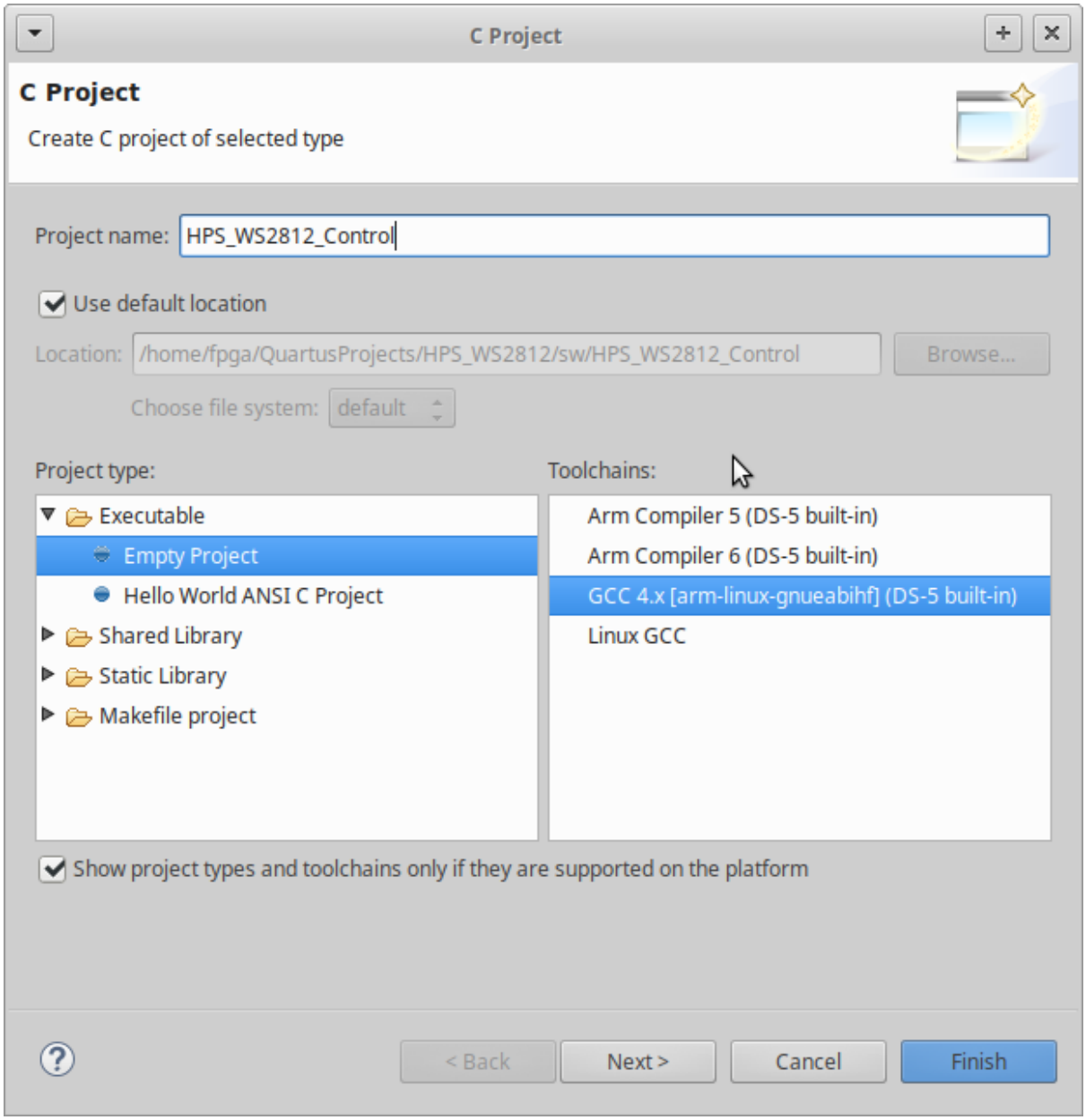

Figure 8 : ARM DS-5 Project wizard

Hit finish button and then open the project properties by right clicking on you new project and selecting **Properties (Alt+Enter)**.

Navigate to **C/C++ General / Path and Symbols**. Select **GNU C** and click **Add.** We are going to insert all the header files needed to compile for Cyclone 5 HPS.

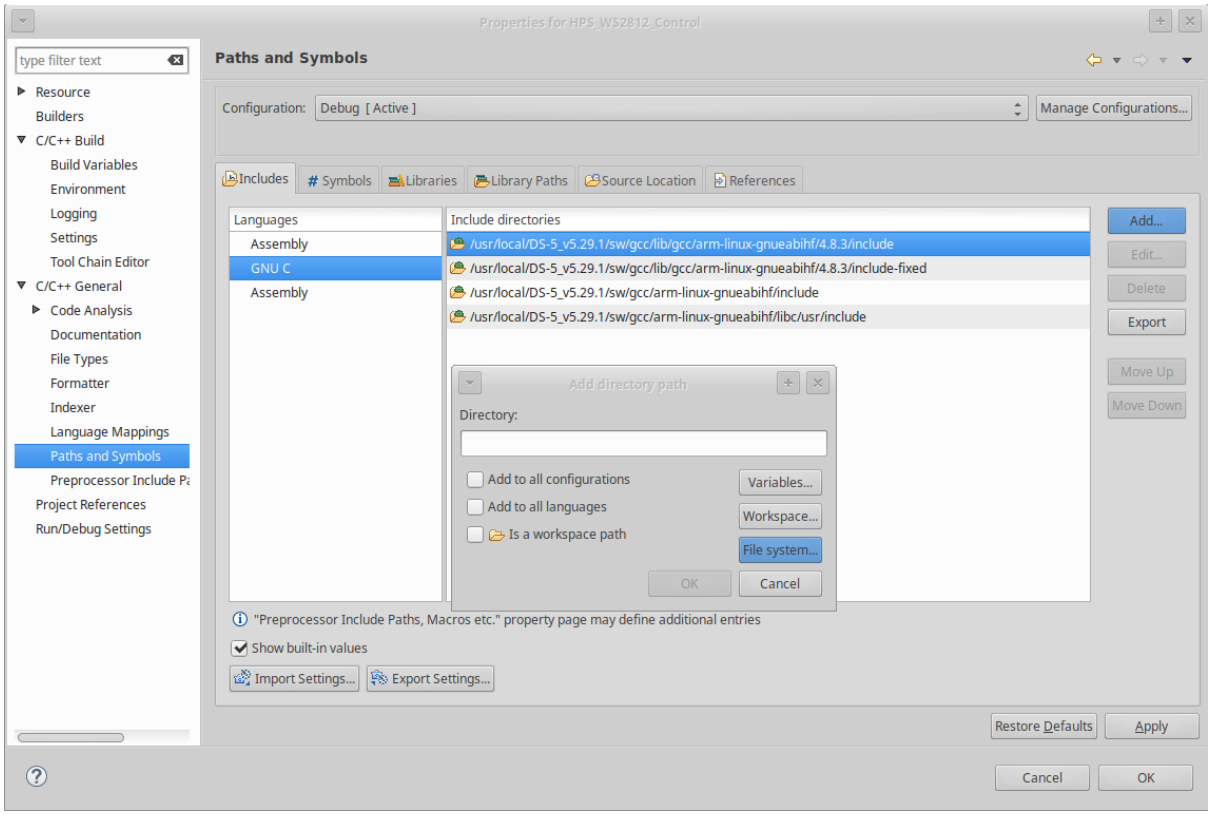

Figure 9 : Add new path to project

Add the two paths in Bold. The first one id the path to Altera hardware libraries to access Altera register write macros. The second one is the path to our hps_soc_system.h file.

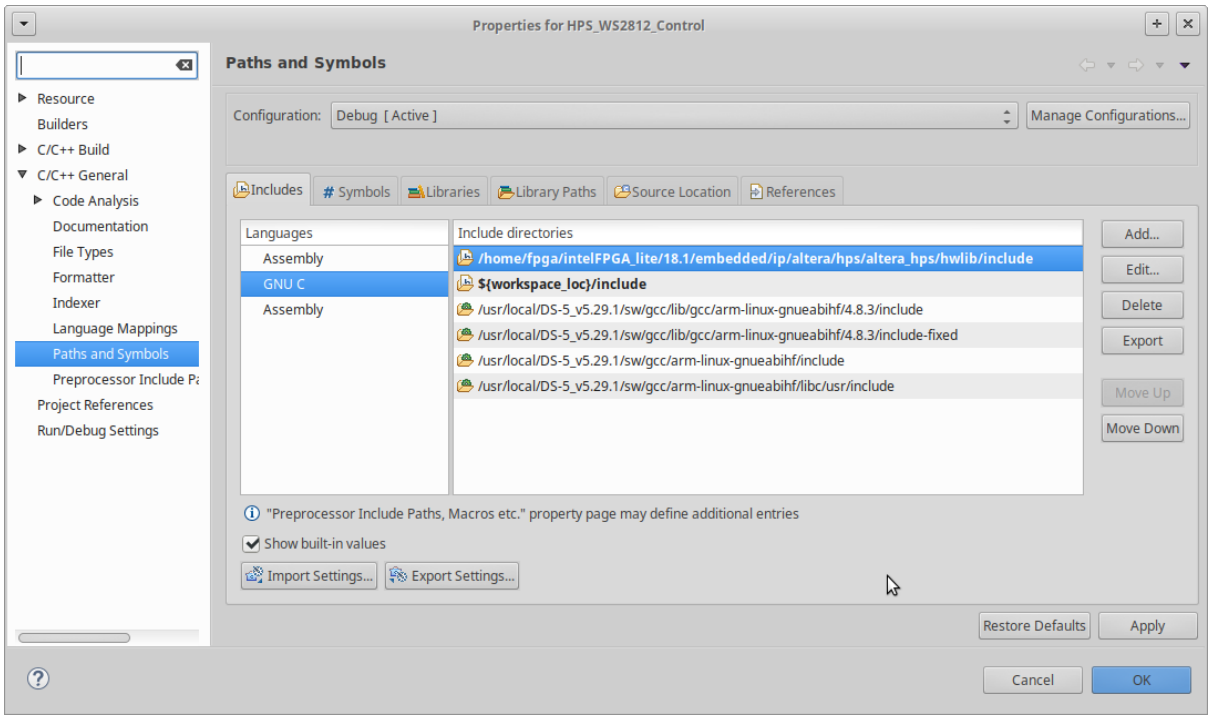

Figure 10: Include paths

If you add any other include paths to your project, this will be the place to put them.

In your project, create a new C **source file** and give it the **main.c** file name.

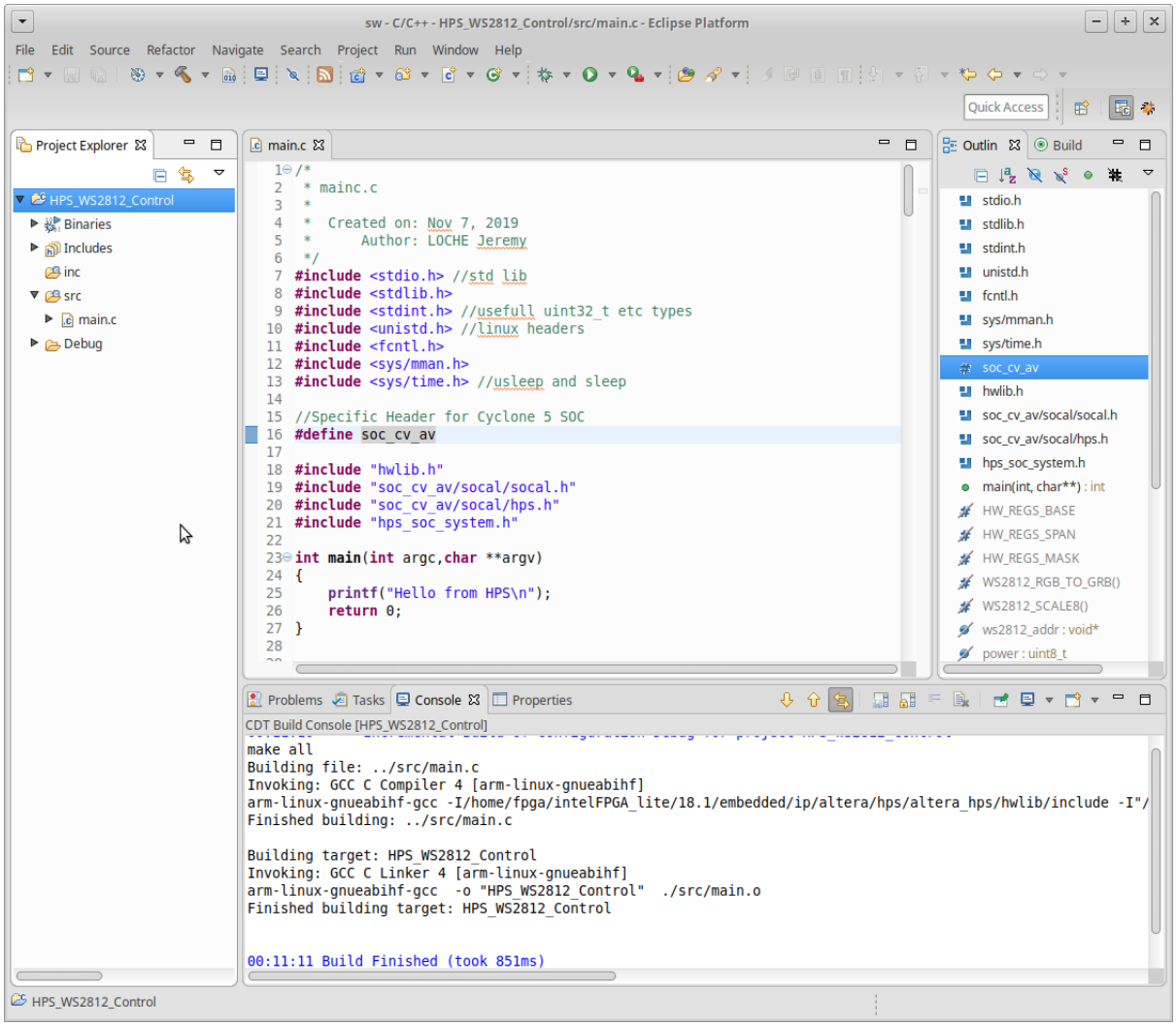

Figure 11: Compiling main.c

Build your project by clicking on the little hammer or **Project->Build project** menu.

You are now ready to build your application for your HPS system! You have a basic project template to work with. If the compiler doesn't find your **hps_soc_system.h** file, just copy it to your project.

If you are accessing the FPGA peripherals through one of your HPS to FPGA Bridge, you will need map the virtual address space of your Linux program to the FPGA bridge addresses. With this additional code to your template, you will be able to get a pointer to the base addresses of your FPGA peripherals that you will be able to manipulate to talk to your peripherals.

Below is an example on how to create a fast FPGA LED animation lighting up the LEDs one after the other from LED[0] to LED[7] on the board.

Manually controlling the LED PIO from the HPS
/* * mainc.c *

```
 *  Created on: Nov 7, 2019
 *      Author: LOCHE Jeremy
 */
#include <stdio.h> //std lib
#include <stdlib.h>
#include <stdint.h> //usefull uint32_t etc types
#include <unistd.h> //linux headers
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/time.h> //usleep and sleep

//Specific Header for Cyclone 5 SOC
#define soc_cv_av

#include "hwlib.h"
#include "soc_cv_av/socal/socal.h" //alt read and write functions
#include "soc_cv_av/socal/hps.h"
#include "hps_soc_system.h"


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


void *led_pio_base;

int main(int argc,char **argv)
{
	printf("Hello World HPS\n");

	void *virtual_base;
	int fd;

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

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

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

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

```
	}

	//Map the virtual address to the real address of the LED PIO.
	led_pio_base= virtual_base + ( ( unsigned long  )( ALT_LWFPGASLVS_OFST +
LED_PIO_BASE ) & ( unsigned long)( HW_REGS_MASK ) );

	uint8_t led_sel=0;
	uint32_t led_mask=0;
	while(1)
	{
		//Calculate the LED animation
		led_sel=(led_sel+1) % LED_PIO_DATA_WIDTH;
		led_mask = (1<<led_sel);

		//Write the 32 bit register of the LED PIO
		alt_write_word(led_pio_base,led_mask);

		//Small delay for the animation.
		usleep(200*1000);

	}

	return 0;
}
```

Of course, this is an example with the LED PIO, but you could also access the Push button and Switch PIOs.