THE MICROZED CHRONICLES

Implementing Xilinx Zynq solutions using Vivado and software in SDK.

Adam Taylor CEng FIET
Chartered Engineer

Contents

List of Figures

Introduction

This is a collection of a number of blogs I have written for the Xilinx Xcell Daily blog on how to use the Xilinx Zynq based around the MicroZed.

This series is continued at

http://forums.xilinx.com/t5/Xcell-Daily-Blog/bg-p/Xcell

I can be contacted at aptaylor@theiet.org

Introduction

I recently received the Adam Taylor Edition of Avnet's Zynq-based MicroZed board, which was sent by the very kind people at Xilinx. I have been writing about the ZedBoard for a while now over on All Programmable Planet. For the original ZedBoard, I used the more traditional PlanAhead, Xilinx Platform Studio, and Software Design Kit (SDK) flow. With that in mind, I decided that for the MicroZed I would implement the system using the Xilinx Vivado Design Suite, which turned out to be surprisingly easy. My aim is to progress with the MicroZed in a similar manner to the ZedBoard: looking at creating the system, using the on-chip XADC, boot-loading the MicroZed, adding my own peripheral, and finally adding an operating system. I expect this will progress rapidly expect due to my familiarity with the ZedBoard.

Figure 1 Adam Taylor Edition of the MicroZed board

The first step is to download the MicroZed board definition and configuration, which are available athttp://www.zedboard.org/documentation/1519. The first file to download is the MicroZed board definition file, which should be extracted to your Xilinx implementation directory. In my case, the directory is loacated at C:\Xilinx\Vivado\2013.2\data\boards\zynq. This file provides the Vivado Design Suite with MicroZed configuration information. The second file you'll need is a TCL file containing the necessary preset information for the MicroZed. We'll run this TCL file once we have created a project.

After starting Vivado, the first step is to create a new project. My first MicroZed project will be an RTL project and will not contain any initial source code. The next step is to select the MicroZed 7010 board as a default target using the definition file just downloaded.

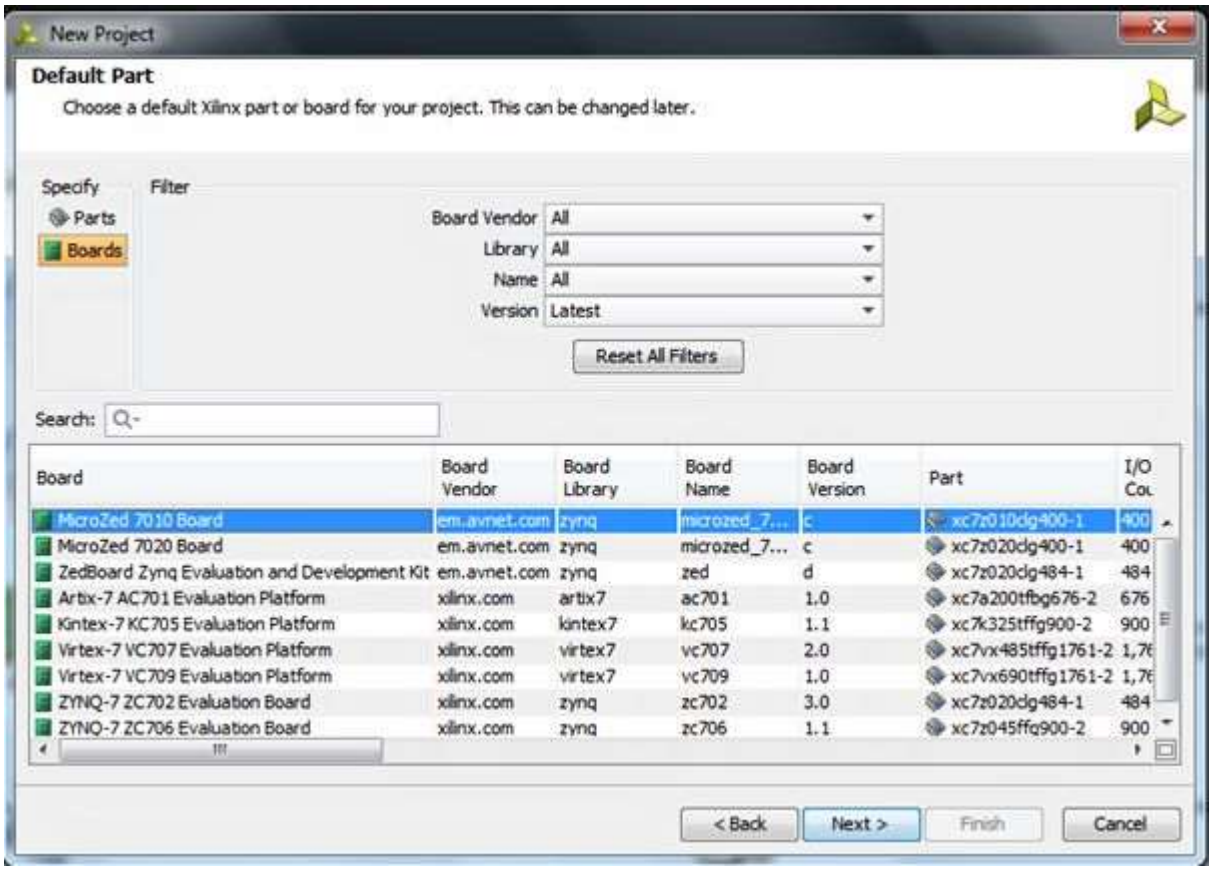


Figure 2 Creating the project

Now that the project is created, we need to add the Zynq SoC's processing system (PS). The best way to do this is to create a new block diagram and add in the Zynq PS from the IP library in Vivado. We can then create a block diagram by selecting the option under the flow-control window on the left-hand side of the Vivado screen.

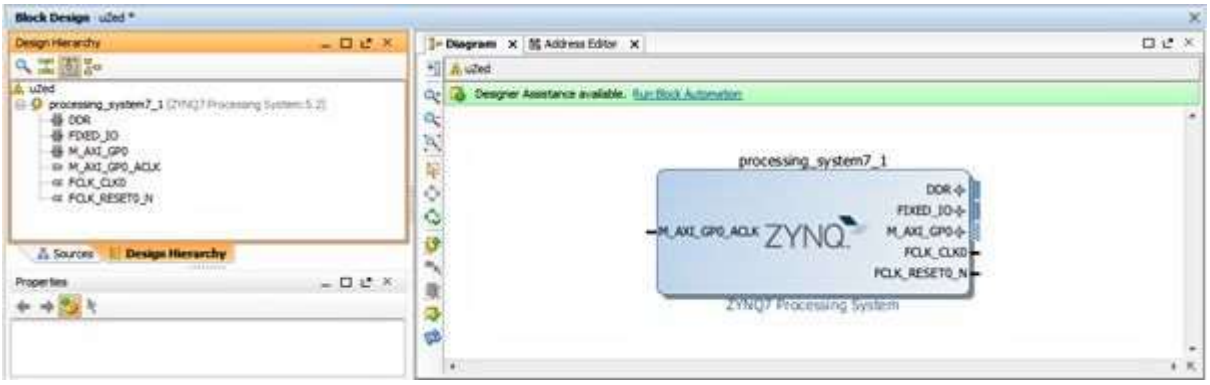


Figure 3 Creating the Block Diagram

With the PS now added to the block diagram, we need to define the system. We could do this by hand (as would be the case for a custom board). Rather helpfully, the MicroZed people have created

a TCL file that defines the MicroZed system. It's the preset file we downloaded at the start. This TCL file defines the PS bank voltages, buses, clocks, fabric clocks, DDR3 settings, external peripherals, and the MIO configuration.

Note: Remember to use linux style forward separators:

"C:/Users/Adam/Downloads/MicroZed_PS_properties_v02/MicroZed_PS_properties_v02.tcl"

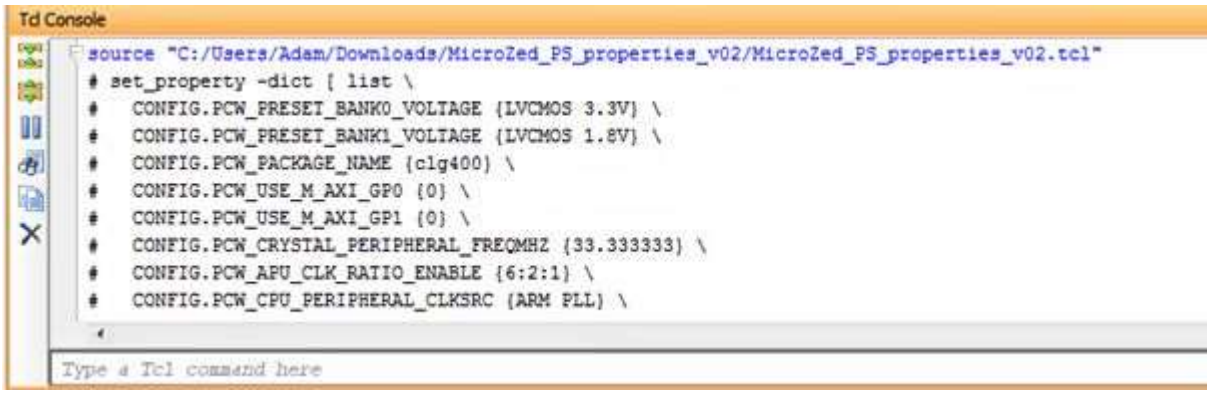


Figure 4 Running the TCL Script

Having applied this file, we then double click on the system and we see the Zynq PS design. Notice that this definition ties up with the capabilities of the MicroZed board's Ethernet, USB, DDR3 etc.

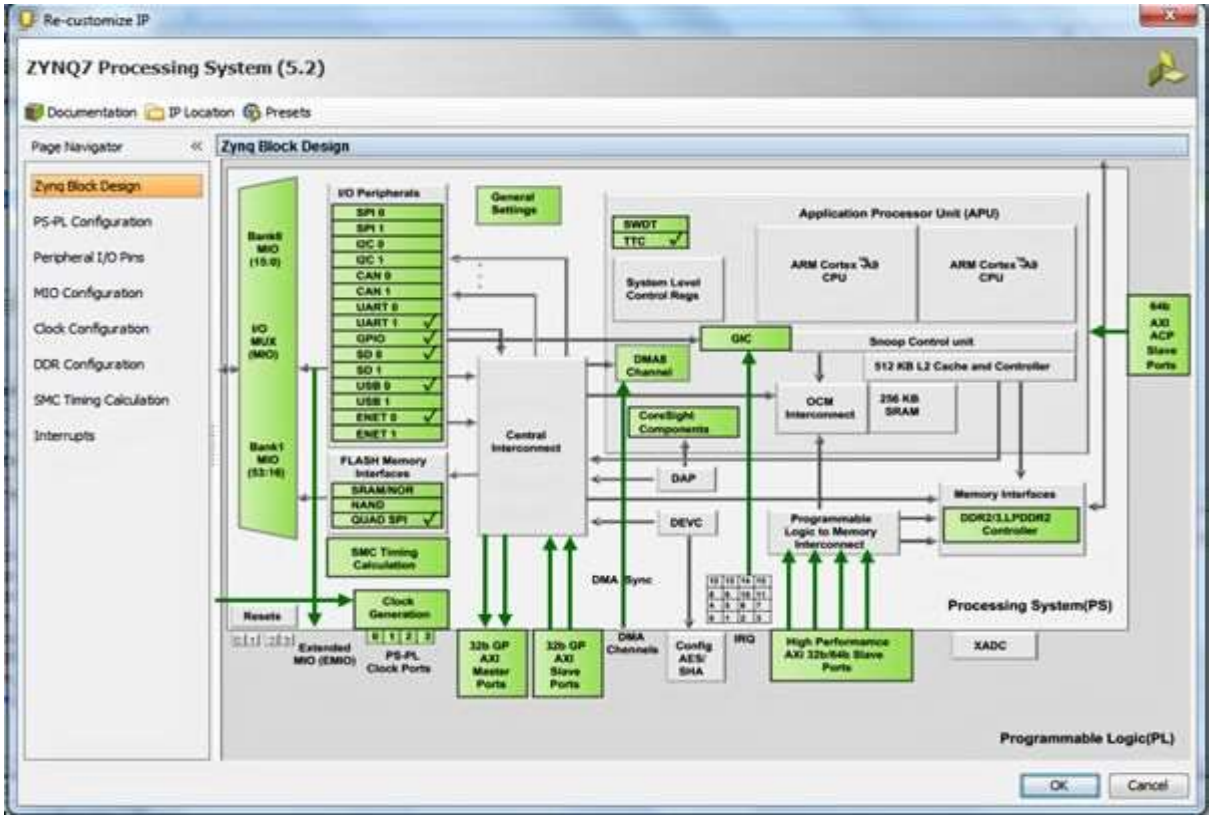


Figure 5 Zynq block Diagram

Once we're happy with the PS configuration, we need to declare the system's external I/O. In this case, we want to declare the DDR and the fixed I/O. Within the Zynq PS, the fixed I/O includes the MIO, clocks, and resets along with the DDR3 reference voltages. As these are fixed, no UCF file is required because we are not working with the programmable logic (PL) side of the Zynq. We will need to create UCFs later when we use the Zynq's PL side.

To add these external I/O declarations, you click on the "run design automation" option that appears at the top of the diagram. This will generate a warning. Clicking on "OK" allows you to proceed and you will then see outputs added to the fixed IO and the DDR within the block diagram.

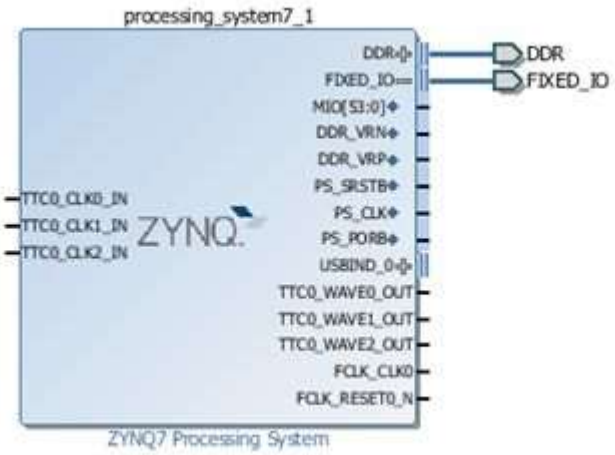


Figure 6 Processing System with the block diagram showing the PS IO

Now we're nearly ready to proceed to build the system. However, we must first validate the design to ensure that it is valid and contains no errors by selecting the "validate design" button on the left side of the Vivado screen.

Figure 7 Validation of the PS design

Having created a valid block diagram we will want to save this before we proceed. Once you have saved the design, the next step is to generate the files needed to implement the system, starting with the creation of the HDL wrapper. But first, we need to determine which language we're going to work in (VHDL or Verilog). We select the HDL via the Tools->Project Settings Menu.

Once we've selected our preferred language, we right click on the uzed.bd file under "sources" and select "Create HDL Wrapper" to generate the wrapper.

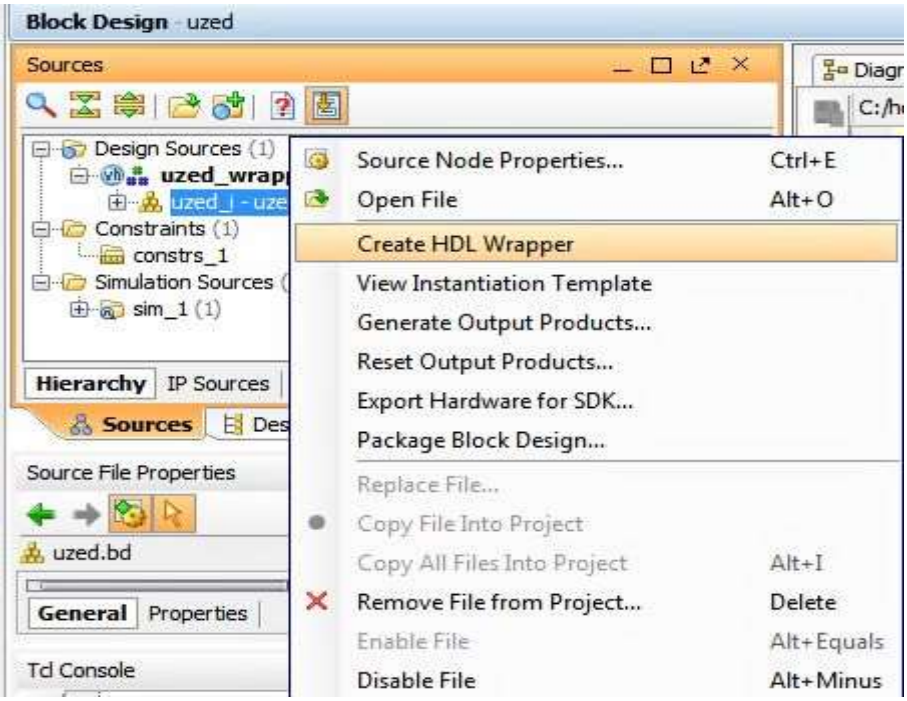


Figure 8 Creating the HDL wrapper for the design

We can also create the necessary synthesis and place-and-route files by selecting the "Generate Output Products…" option from the same menu that we used to generate the HDL wrapper.

Once these files have been created, it's time to generate the bitstream by selecting the "Generate Bitstream" option in the flow navigator on the left of the Vivado screen. When the bitstream generation completes, you will see:

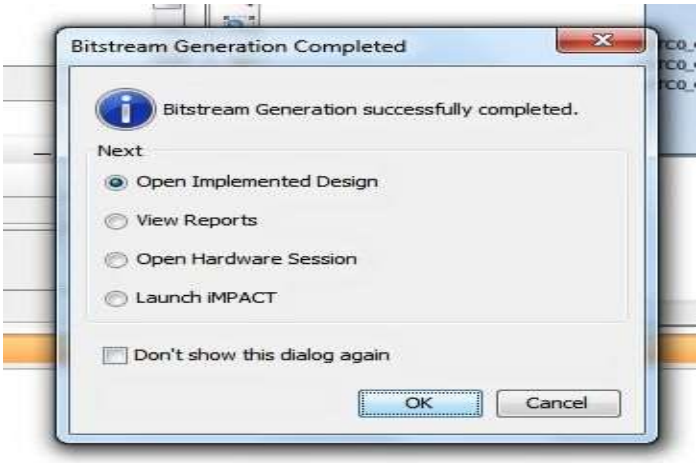


Figure 9 Opening the design once implemented

With the bitstream created, we then export the data into SDK. Then we're ready to write the software to run on the MicroZed's Zynq SoC. I'll discuss this process in my next blog, but a sneak peak of where we are headed appears below.

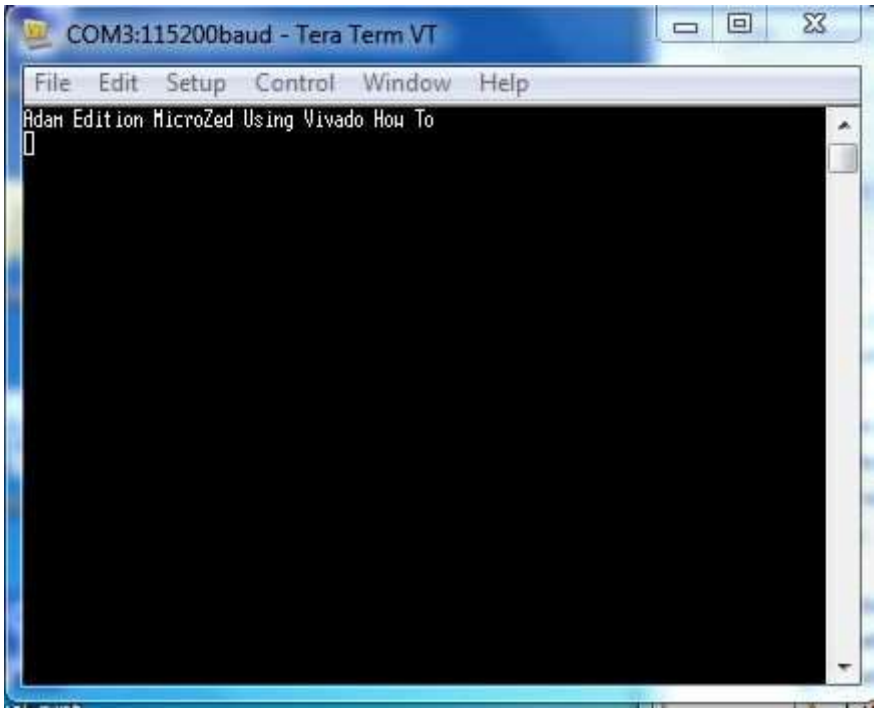


Figure 10 The desired outcome

I spent an afternoon bringing up the MicroZed and really enjoyed the experience. My next blog post will wrap up SDK project creation and boot loading with the MicroZed.

Note: Please do not be concerned if you see an error in Vivado version 2013.2 which reports "Failed to get a license:Internal Bit stream." This is a bug in the current version. You can check in the implementation log to make sure that the license was in fact obtained.

Setting the software Scene

In the last chapter I focused upon creating the MicroZed system using the Vivado Design suite. I'd progressed as far as creating the bit file, which is needed for device configuration. The next step is to export the system hardware definition to the Software Design Kit (SDK) to allow the software engineering team to write the application code for the processing system side of the Zynq All Programmable SoC.

This is a very simple task, achieved by using the export function under the File->Export->Export Hardware for SDK Option as seen below:

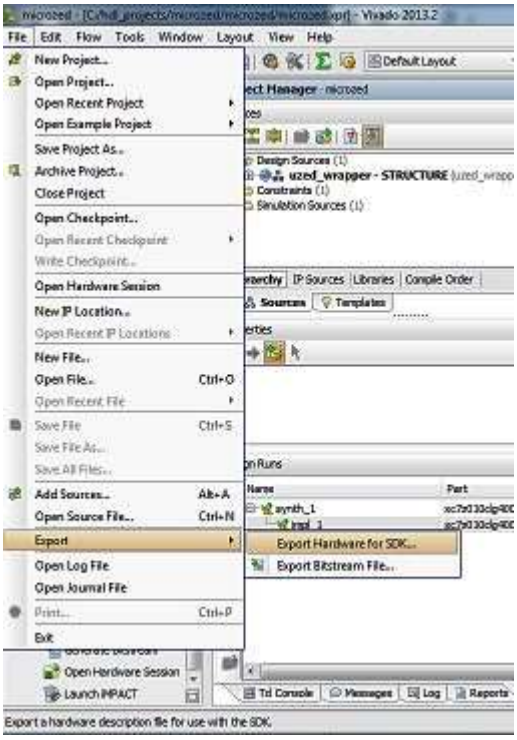


Figure 11 Exporting the hardware design to SDK

The "Export Hardware" function creates a folder under the SDK_Export directory within your project that defines the hardware definition of the system. This should automatically be imported into SDK but if it is not, you can create a new hardware specification underneath the File->New->Other option, which allows you to point it to your exported hardware definition.

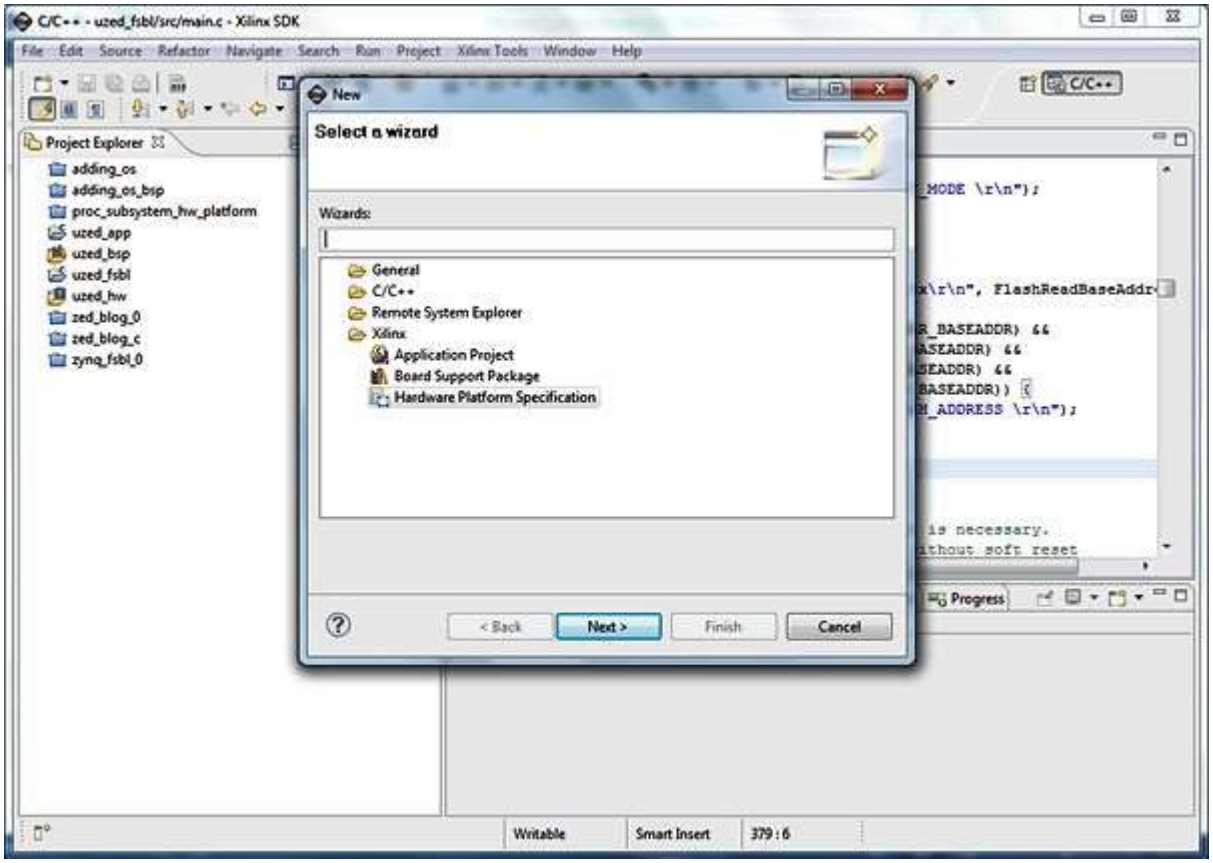


Figure 12 Creating a new hardware platform in SDK

A project within the SDK designed to run upon the Zynq requires three different definitions:

- Hardware Definition (uzed_hw), which defines the system hardware definition. This definition contains a file called system.xml that defines the makeup of your system: the address map, IP blocks present, and the device type. You will also see the configuration bit file that we will need later when we discuss configuration.

- Board Support Package (uzed_bsp), which contains the drivers needed to support the IP block within the design. The BSP mainly consists of C include files. Here the file system.mss defines the BSP settings. This file will also contain links to the drivers for the IP blocks and provides examples of how to use them. Within the include file here you will find Xparameters.h, which defines your system in great detail.

- The application itself (uzed_app). This is where you will actually put the software application.

  The project definitions reference each other in progression with the Hardware Definition being the lowest level. The Hardware Definition is referenced by the BSP which in turn is referenced by the application.

  Once we have imported the hardware project, we need to create a new BSP for our MicroZed. The simplest way to do this is by selecting File->New->Board Support Package, which allows you to select the name of the BSP, the processor upon which the application will run, and most importantly the hardware platform definition that you just imported.

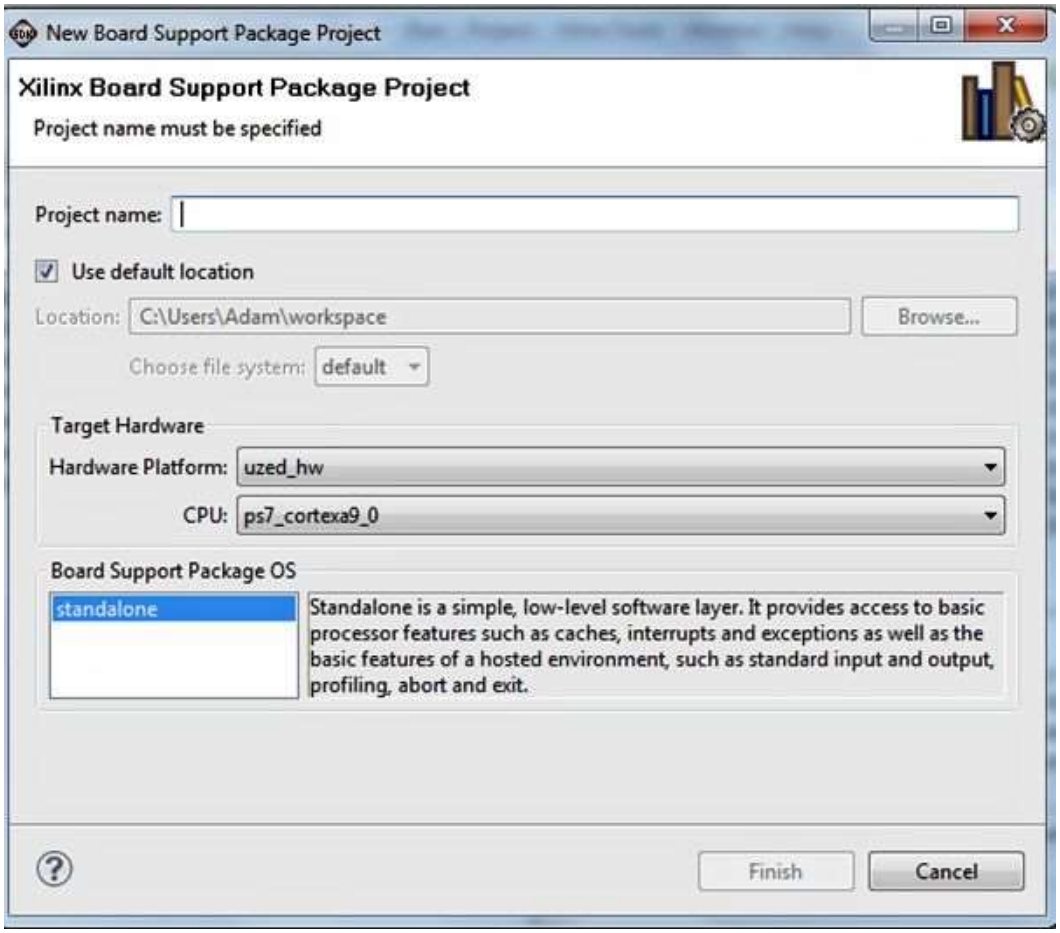


Figure 13 Creating a Board Support Package

Having created the Hardware Platform definition and the BSP, we're ready to create the main application program using File->New->Application Project.

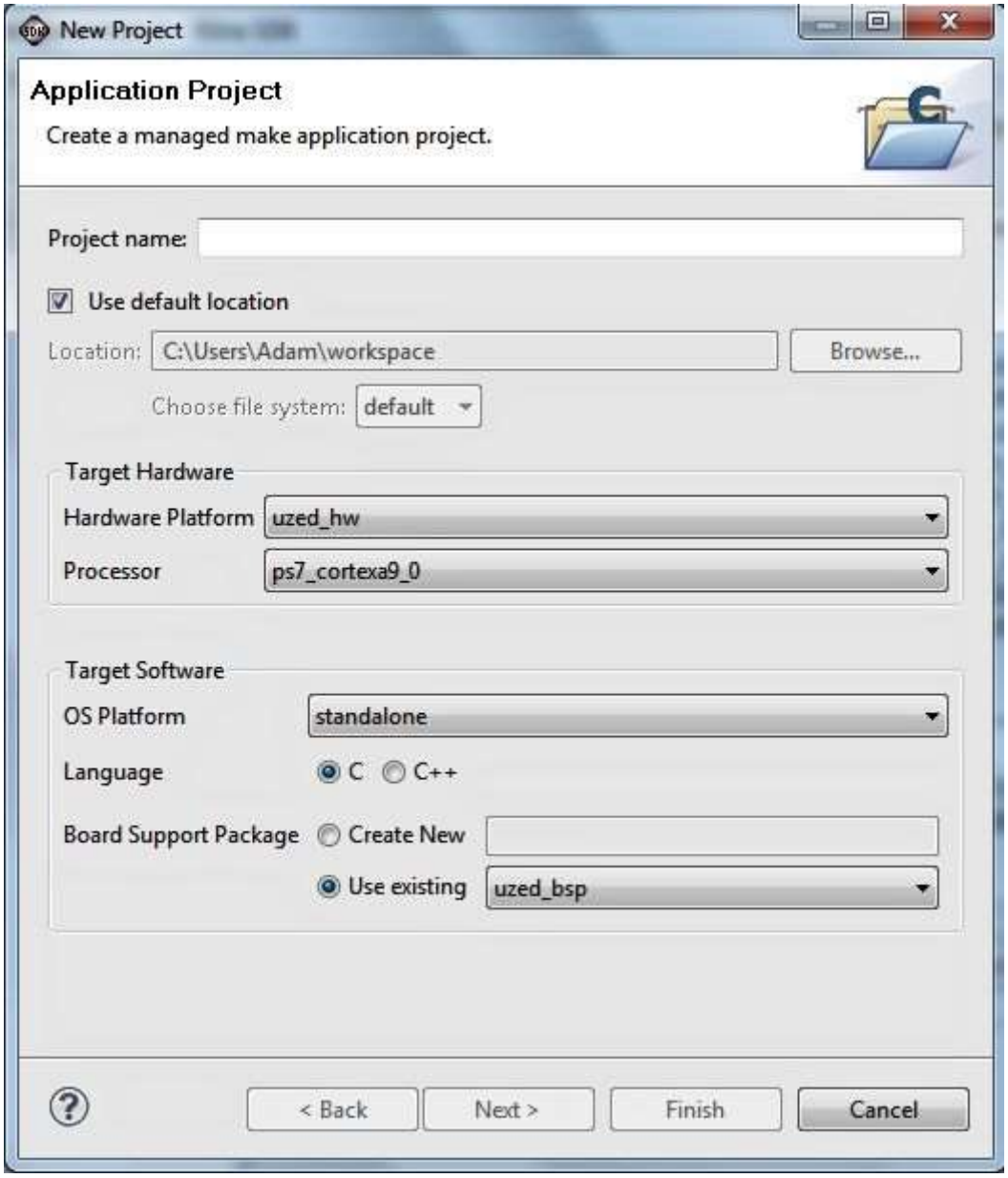


Figure 14 Creating a new application stage one

This screen lets you select the project name, the hardware platform and processor to be used, along with the board support package selection we created earlier. As this is a bare-metal implementation, we leave the OS Platform as standalone.

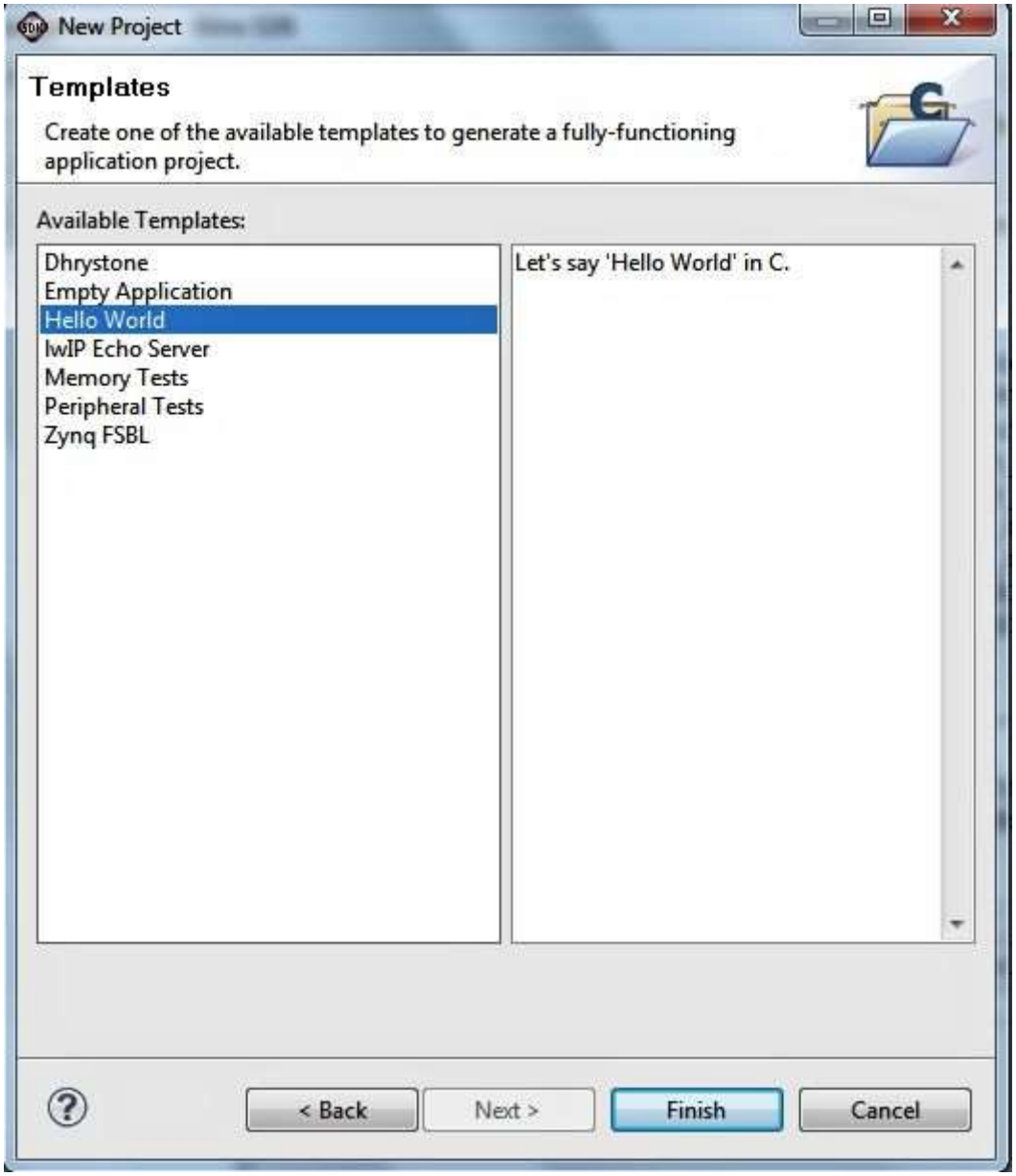


Figure 15 Creating a new application selecting a demonstration if desired

The next stage is to select the project type we want. Selecting the simple “Hello World” application will automatically create all that you need to get the Zynq up and running on the MicroZed board.

Under the application project (used_app) within the SRC directory, you will notice a number of files. The one named helloworld.c contains the program main function which initializes the platform and runs the application.

It is this file we will be modifying in the next blog so that we can finally run it on the MicroZed.

First look at the Software Environment

A quick recap of the last two blogs shows we have defined our Zynq-based MicroZed system using Vivado and established the baseline within the Software Development Kit (SDK). Now we're ready to create our first application. "Hello world" is the traditional program used to test the entire work flow for microprocessors and to get quick proof that it's working as intended (much like a flashing LED is the FPGA equivalent). Helpfully, SDK creates a hello world project for us. However, just using this application as is does not provide an understanding of what is happening or what's required to write software for the Zynq All Programmable SoC so let's look at this file in a little more detail.

If you open the helloworld.c file under the used_app src directory, you will find a pretty simple and short program after the comments at the top:

```
#include <stdio.h>
#include "platform.h"

void print(char *str);

int main()
{
    init_platform();

    print("Adam Edition MicroZed Using Vivado How To \n\r");

    return 0;
}
```

Two header files are included. The first, stdio.h, is contained within the standard C library. Header files contained within the standard C library can be found under the includes section of the software application. You can tell that this header file is within the standard C library due to the angle brackets "< >" used to enclose the call.

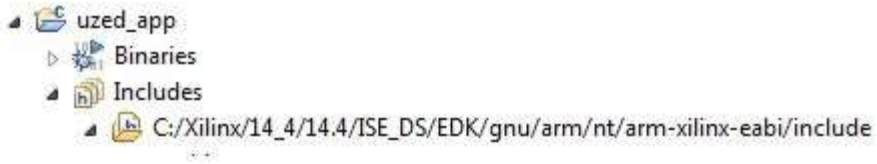


Figure 16 Location of the Standard C Library

Stdio.h contains input and output functions for the Zynq system, provided that you have correctly configured the STDIN and STDOUT within the board support package (BSP) properties.

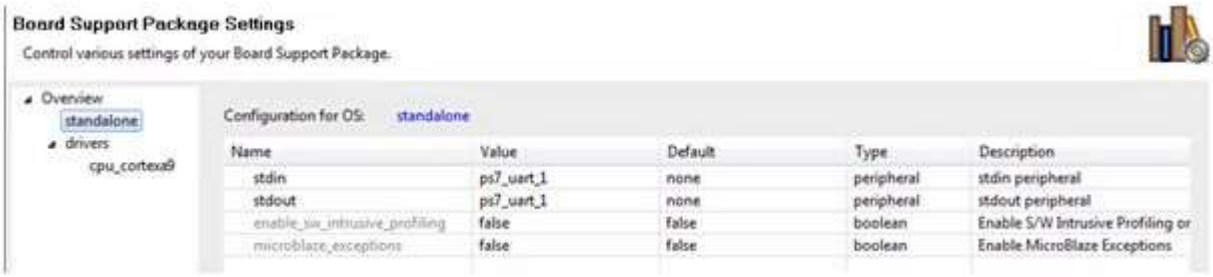


Figure 17 Setting the STDIN and STDOUT within the BSP

The second header file, platform.h, is a local file. It is located with the other application files. This file contains the function prototypes for initializing and closing down the platform. If you open this file you will notice first the include guards to prevent multiple definitions of the header file followed by the function prototypes. You will also notice there is no code defining these functions. The functionality is defined within the pre-compiled Xilinx C Library, which I defined under the BSP and is seen in libxi.a. This library also contains a number of other functions for you to use.

The source code for inclusions within this library can be seen under the libsrc directory within your BSP:

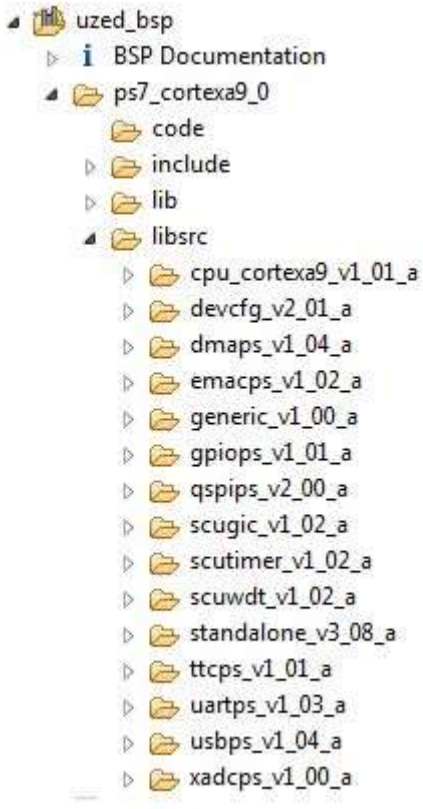


Figure 18 Include library source code

This file includes the stdio.h header, which can be used to send and receive data over the stdin and stdout ports. In this case, these ports are the Zynq All Programmable device's UART located in the Processing System.

Using stdio.h can increase the size of the application file, which is not always desirable for embedded applications. Hence the example from Xilinx uses the PRINT function, which is declared under the libsrc->standalone_v3_08_a directory in your BSP folder. This function is a much simpler output routine. Information on this function and others can be found in Xilinx user guide UG643 ("OS and Libraries Document Collection").

Because there is no header file for the PRINT function, the "Hello world" program defines the PRINT function prototype just after listing the include files. This definition allows the PRINT function within the main file to compile and link without error.

The main function follows. It calls the init_platform function, which as I explained above is defined within platform.h

The next line of code is the data we wish to output over the UART. This line uses the PRINT function to display the text entered between the quote marks. The final line of the code, "return 0," stops the program.

Building and running the file as it appears above results in the following being displayed upon your terminal.

Figure 19 Result of running the code on the Zynq

The console window will also report the size of the ELF file which can be seen below:

```
Invoking: ARM Print Size
arm-xilinx-eabi-size uzed_app.elf  |tee "uzed_app.elf.size"
   text    data     bss     dec     hex filename
  47980    1096   27736   76812   12c0c uzed_app.elf
Finished building: uzed_app.elf.size
```

Figure 20 Programme size using the PRINT function

Changing the file to use the STDIO PRINTF function in place of the PRINT function dramatically increases the size of the program. For the Zynq-based MicroZed system, program size is not too important because we have lots of DDR SDRAM capacity. However, program size can be considerable for smaller MicroBlaze-based applications.

```
Invoking: ARM Print Size
arm-xilinx-eabi-size uzed_app.elf  |tee "uzed_app.elf.size"
   text    data     bss     dec     hex filename
  79516    2264   27816  109596   1ac1c uzed_app.elf
Finished building: uzed_app.elf.size
```

Figure 21 Programme size using PRINTF function

I hope this explanation helps you to understand how the “Hello world” program works and what its dependencies and libraries are. In my next instalment, we will look at how we execute this program on the MicroZed board and the Zynq All Programmable SoC.

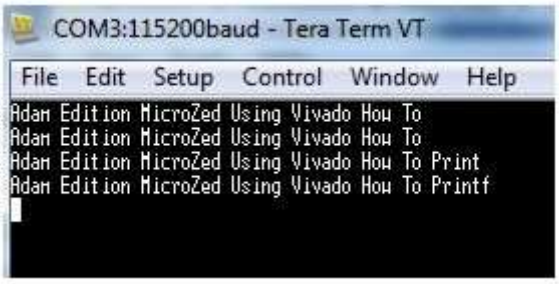


Figure 22 Output showing PRINT and PRINTF function working

Note: To ensure the best demonstration, I commented out the stdio.h header file from the project when I used the PRINT function and did a similar sort of thing when using “printf” with the print function prototype.

Configuring the Zynq the basics

So far in this series we have looked at creating the hardware system on the MicroZed board, setting up the software environment, and understanding the “hello world” program example. After three blogs it is time to pull this all together and get something running on the MicroZed so that you can see the results. We can use one of two methods to do this. We can use SDK to download and run the application in the Zynq All Programmable SoC’s on-chip RAM over a JTAG link or we can program a non-volatile memory and use a boot loader running on the Zynq SoC to load our application.

To keep things nice and simple in this blog post, we will download the application to RAM over the JTAG link. That’s the simpler of the two options and it’s of more use during debugging, at least initially.

The first step is to create a new run configuration: “project explorer -> project -> run -> run”, accessed from the right-click menu. This new configuration is created under the C/C++ ELF type on the left side of the dialog box:

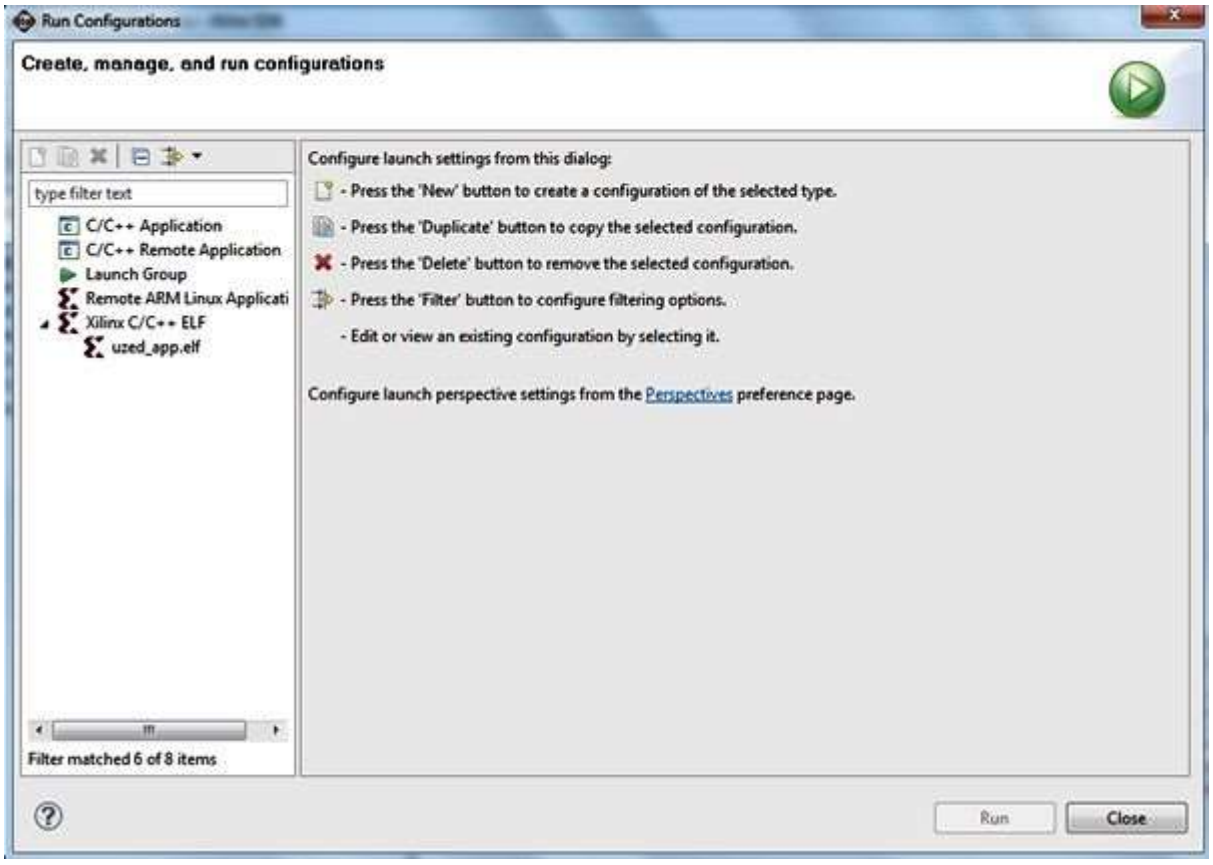


Figure 23 Creating a run configuration

Once we have created a new configuration, we can define our STDIO connection as the serial port that we will use for the STDIO—if any—along with processor reset options and debug / profiling options. (I tend to use a standalone terminal program for debugging.)

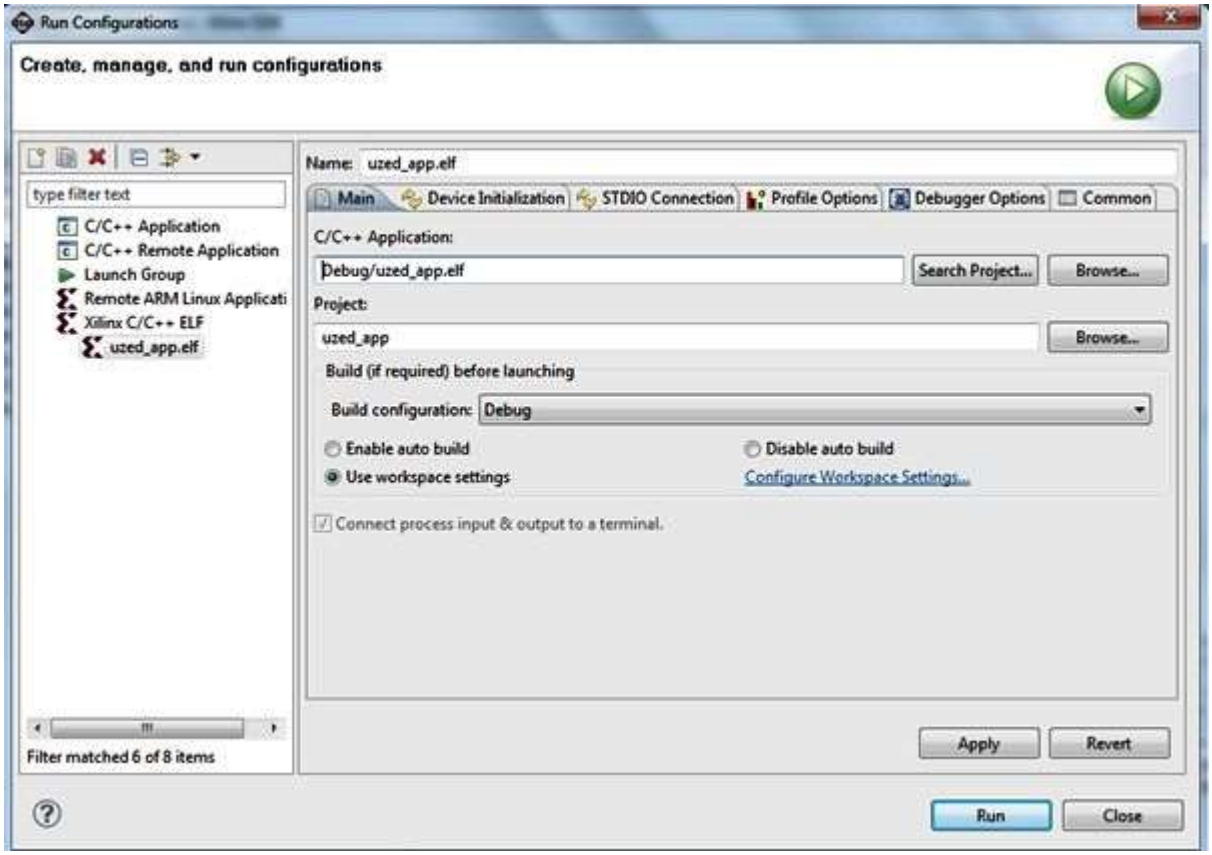


Figure 24 Build configuration

Once we are happy with the configuration, we can apply and then close it. However, at this point we're not yet ready to run anything. There's one more step.

The final step is to connect the MicroZed board to our PC and program the Zynq by downloading the previously compiled software. First, we connect both the JTAG cable and the USB (RS232) power cable to the MicroZed. Following this we can program the hardware via the JTAG connection using the "Xilinx tools -> program FPGA" option, which checks that the bit file is the correct one and then programs the device.

Before we download the software we must ensure we have a terminal program open on the PC so that we can see communications from the MicroZed board. We set the UART to 115200 baud, no parity, and one stop bit.

Having completed the configuration we can now download our ELF file and try out our program. This is as simple as selecting our project within the project explorer, right clicking, selecting "Run As", and then "Launch on Hardware." If we had not previously built or compiled our code, this procedure would automatically build our design for us, provided it is error free.

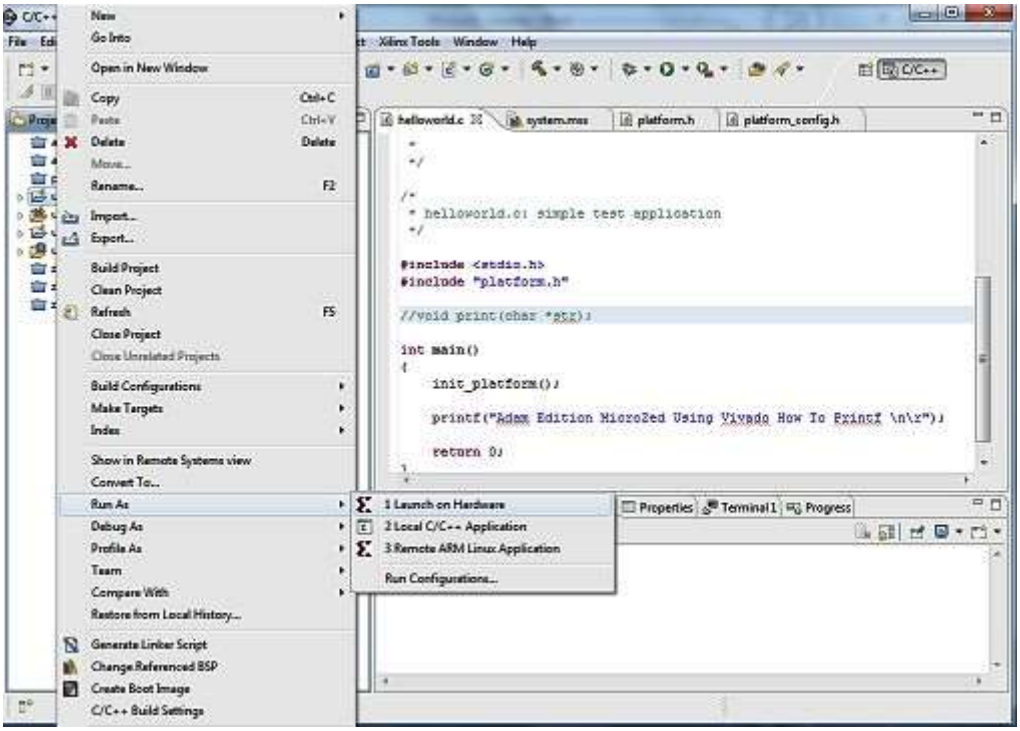


Figure 25 Lunching the software on the hardware platform

When you initiate the program launch, you will see a progress bar within SDK. Once the download completes, you should see the output from your application in the terminal window. Hopefully, it looks something like this:

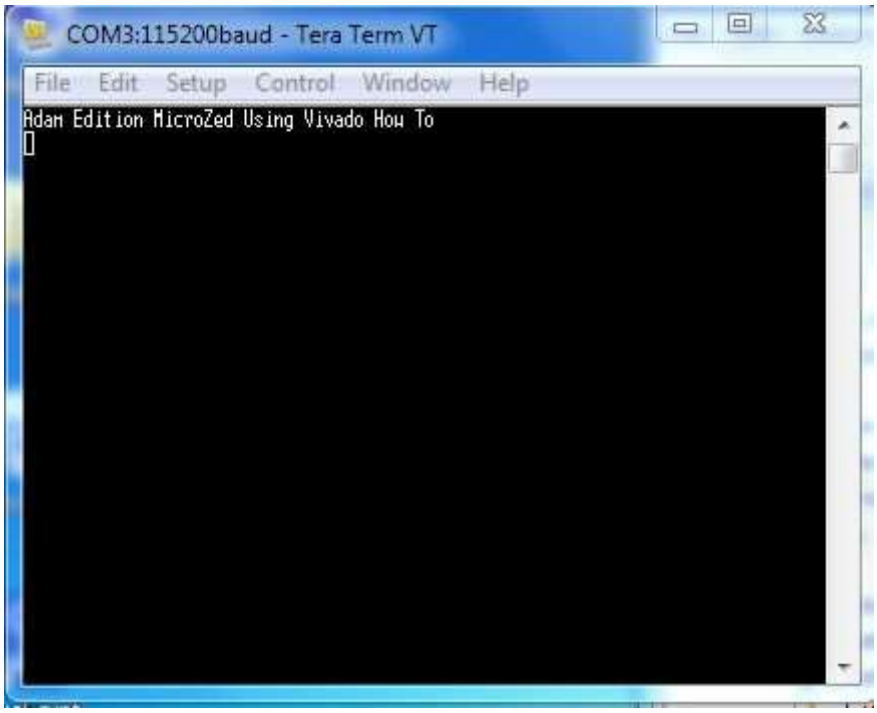


Figure 26 Result of running on the hardware platform

In my next blog we will create a First Stage Boot Loader so that we can boot the MicroZed from non-volatile memory.

First Stage Boot Loader

Following the successful testing of our software application using the Zynq All Programmable SoC's launch-on-hardware option - we now want to ensure that our software application can be stored in non-volatile memory so it will run after a power on or reset. We'll need a boot loader to do this. The boot loader loads both the FPGA configuration and the software running on the Zynq SoC's dual-core ARM Cortex-A9 MPCore processor. So to start, it's important to understand the way the Zynq SoC boots and configures itself.

The Zynq SoC requires configuration information for both the ARM-based processor system (PS) and the programmable logic (PL). To simplify the process of configuring both the PS and PL, the configuration sequence is slightly different than the sequence used for a Xilinx FPGA. This difference results from the use of two file types:

- The FPGA Bit File – Defines the behavior of the PL
- The software ELF file – The software program to be executed by the PS

Within a Zynq SoC, the PS is the master and therefore configures the PL. The only exception to this is when the JTAG interface is being used to configure the PL. The advantage of this fixed sequence (PS configures PL) is that it allows the PS to power up and operate while the PL remains unpowered. You can use this feature to reduce system power consumption. Of course, if you want to use the Zynq PL, you will need to power it up at some point during the operation of your product.

The software code and the FPGA configuration bit file can be stored within the same configuration-memory device attached to the PS. The PS supports configuration using a number of different off-chip, non-volatile memories (Quad SPI Flash, NAND Flash, NOR Flash or SD Card). The MicroZed board design allows for SD Card and Quad SPI configuration memories.

The Zynq SoC on the MicroZed board follows a typical processor boot sequence to configure both sides of the device, initially running from an on-chip, non-modifiable BootROM. The Zynq SoC's on-chip BootROM contains drivers for the supported non-volatile memories, which is rather convenient.

The Zynq SoC's BootROM is configured by a header contained within off-chip, non-volatile memory. This header defines a number of boot options and it's the first thing the BootROM code looks for. The header defines boot options such as execute in place (not possible from all memories), FSBL offset, and secure or not-secure configuration. The header ensures the BootROM operates compatibly with the way that the configuration memory has been formatted.

You have the option of using either secure (encrypted) or not-secure configuration files. Both are supported and defined by the boot ROM header. If you choose the secure configuration, the PL must be powered to activate the on-chip AES and SHA decryption hardware. These hardware security blocks are located within the Zynq SoC's PL but they are implemented as hard macros, not constructed from programmable logic.

The next configuration stage is called the First-Stage Boot Loader (FSBL) and it is created by the system design team. The FSBL can configure the DDR SDRAM memory controller and other on-chip peripherals as defined in the Xilinx Platform Studio (XPS) hardware definition. These devices should be configured before loading the software application and configuring the PL.

Overall the FSBL is responsible for

- Initializing the PS with the information provided by XPS
- Programming the Zynq SoC's PL if the appropriate bit file is provided
- Loading either a Second-Stage Boot Loader (SSBL) if an operating system is being used or loading a bare-metal application into DDR SDRAM
- Starting the execution of the SSBL or the bare-metal application

The Zynq SoC's PL is programmed via the Processor Configuration Access Port (PCAP), which allows both partial and full PL configuration. (If you'd like more detailed information about the Zynq SoC's PCAP, see "Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All Programmable SoC Devices") The PL can be configured at any time once the PS is up and running. The PS can also read back the PL configuration and check for errors, which can be very handy if you are using the Zynq SoC in an environment where it may be subject to single-event functional interrupts (SEFI).

To create a bootable image for your Zynq solution you will need at least the following:

1. Boot ROM Header – Controls Boot ROM settings such as execute in place, encryption, Quad SPI configuration, FSBL offset, and image length
2. First stage boot loader
3. PL configuration bit file
4. Software application to run on the PS

Like Xilinx FPGA's, the Xilinx Zynq All Programmable SoC uses a number of mode pins to determine which type of memory the configuration and software files are stored in, along with other crucial system settings. On the Zynq SoC, these mode pins share the multiuse I/O pins on the PS side of the device. In all, there are seven mode pins mapped to MIO[8:2]. The first four pins define the boot mode; the fifth pin determines whether the PLL is used or not; and the sixth and seventh pins define the bank voltages on MIO bank 0 and bank 1 during power up. The voltage standard defined on MIO bank 0 and 1 can later be changed by the first stage boot loader if needed.

Creating the Boot loader

When last we looked at the MicroZed board, we focused upon how the Zynq All Programmable SoC configured both the on-chip processing system (PS) and the on-chip programmable logic (PL) and how this process differs from traditional FPGA configuration procedures. Having now covered the academics behind the configuration of the Zynq SoC in sufficient detail, it is time to look at creating a First Stage Boot Loader (FSBL), which we'll use together with our software application and programmable logic design to create a configured system that boots after power on or reset.

The Xilinx Vivado Design Suite will generate an FSBL that loads your application and configures the Zynq PL. The process is remarkably simple and an example FSBL is provided. First, create a new project using "new -> application project" within the current SDK workspace (the one containing your project) as shown below

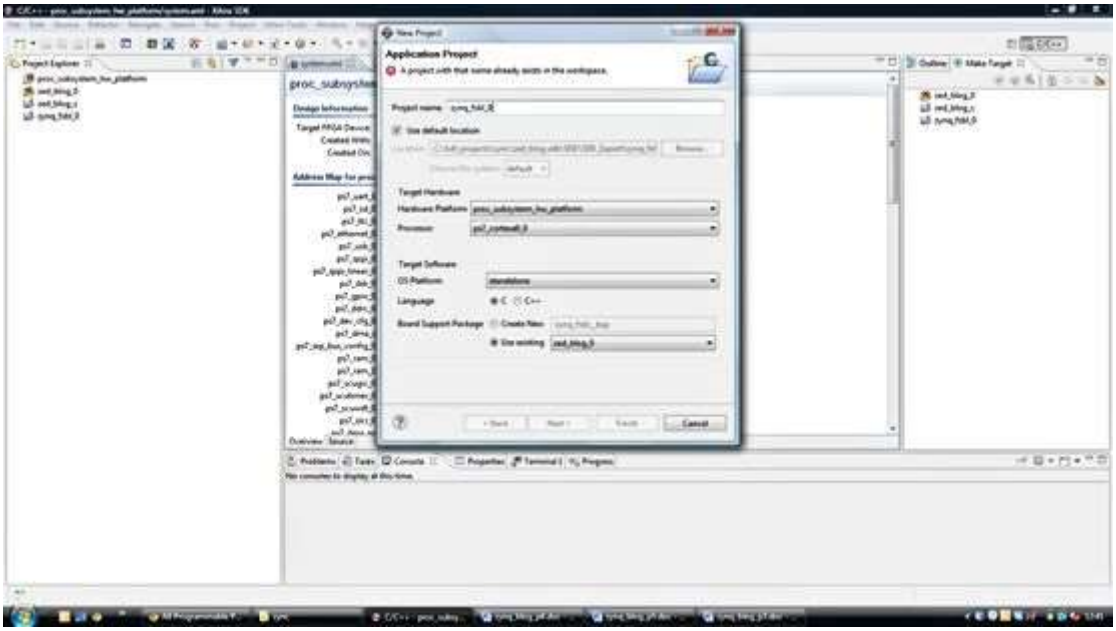


Figure 27 Creating the First Stage Boot Loader Project

Select whether you will be using C or C++ and also select the board support package (BSP) defined for your system (used_bsp). Then, name the project. In this case I used the name uzed_fsbl.

On the next tab, select the Zynq FSBL option from the available templates as shown below and your FSBL project will be created. Now, we are nearly ready to create the boot image. If you have "compile automatically" selected, the FSBL will be complied. Otherwise, it will be complied on demand later on.

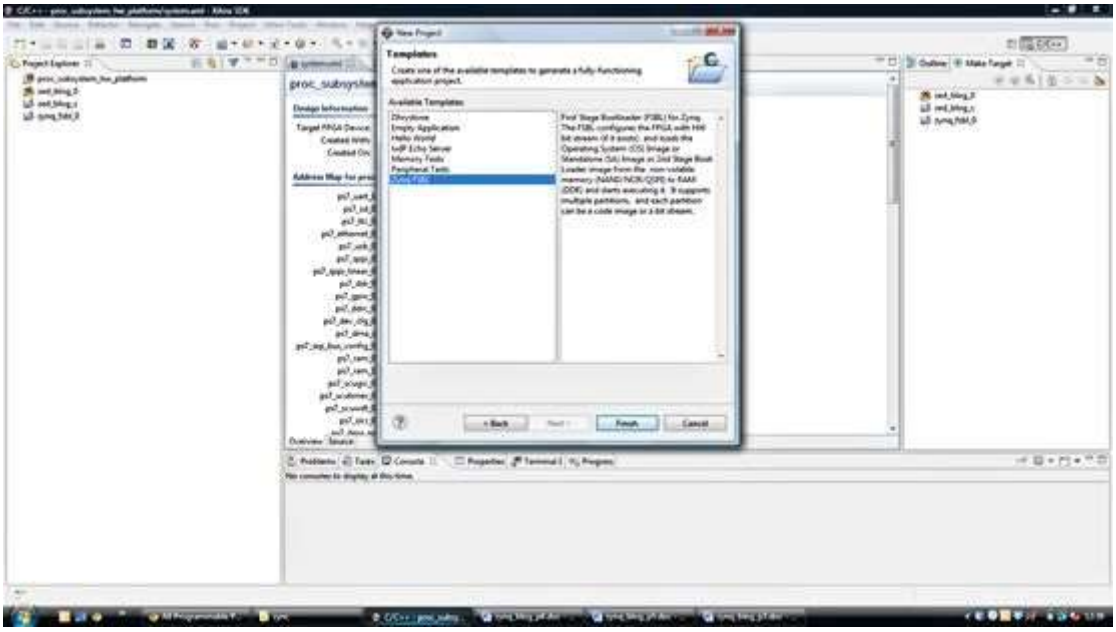


Figure 28 Creating the FSBL from the template provided.

But first, we need to make a change to the linker script provided with the FSBL because it has no idea where the DDR memory is located in the processor's address space. To do this, we need to open the lscrip.ld and specify the DDR memory location in the file. The location of the DDR SDRAM within the address space can be found in the linker script you created for your application. This information can be found in the system.xml file under the hardware platform definition.

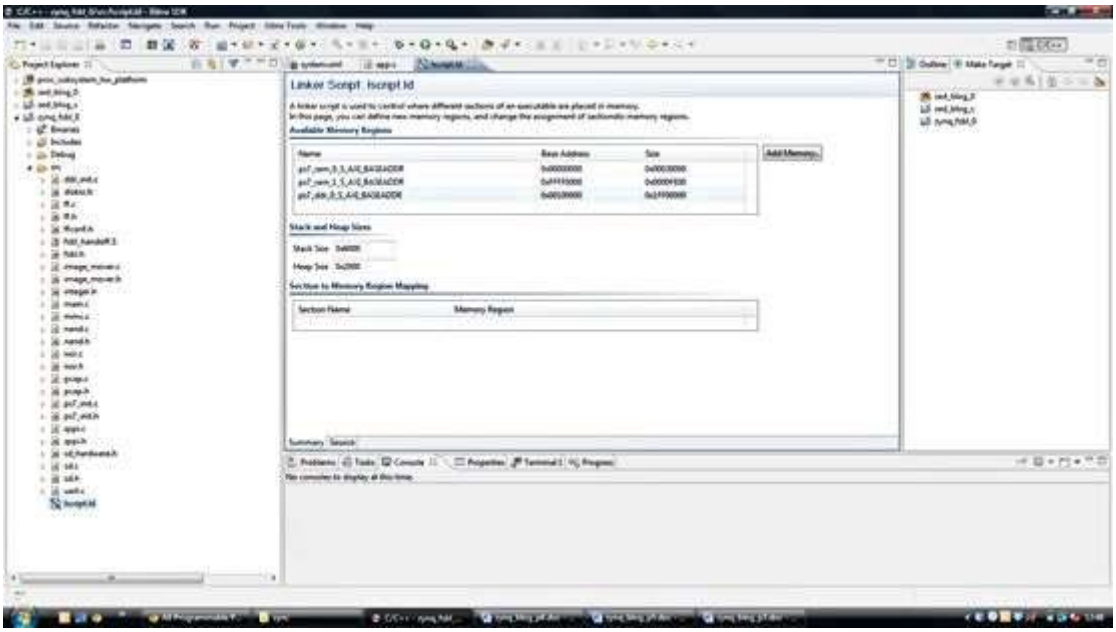


Figure 29 Defining the DDR address space

The figure above shows the FSBL lscript.ld with the system's DDR memory address added in. If you don't specify the correct DDR SDRAM address, the boot loader will run and the Zynq PL will configure, but the application will not run if it's configured to execute in DDR memory.

Generating a Configuration Image

Looking under the Vivado Design Suite's Project Explorer, you should hopefully now have the following modules:

1. uzed_hw – named after the processing subsystem you created, this is the hardware definition of your file
2. uzed_bsp – This is the board support package you created.
3. uzed_app – The application itself.
4. Used_fsbl – The first stage boot loader we have just created and modified the linker script for.

Each module should have a slightly different symbol identifying its type.

Because we are creating a bare metal application (no operating system), you need the following files—in this specific order—to create a proper boot image:

1. First Stage Boot Loader (FSBL)
2. FPGA programming bit file (created in Vivado when we ran the implementation)
3. C Application

Creating a boot image is very simple within the SDK using the "Create Zynq Boot Image" option under the Xilinx Tools Menu as shown

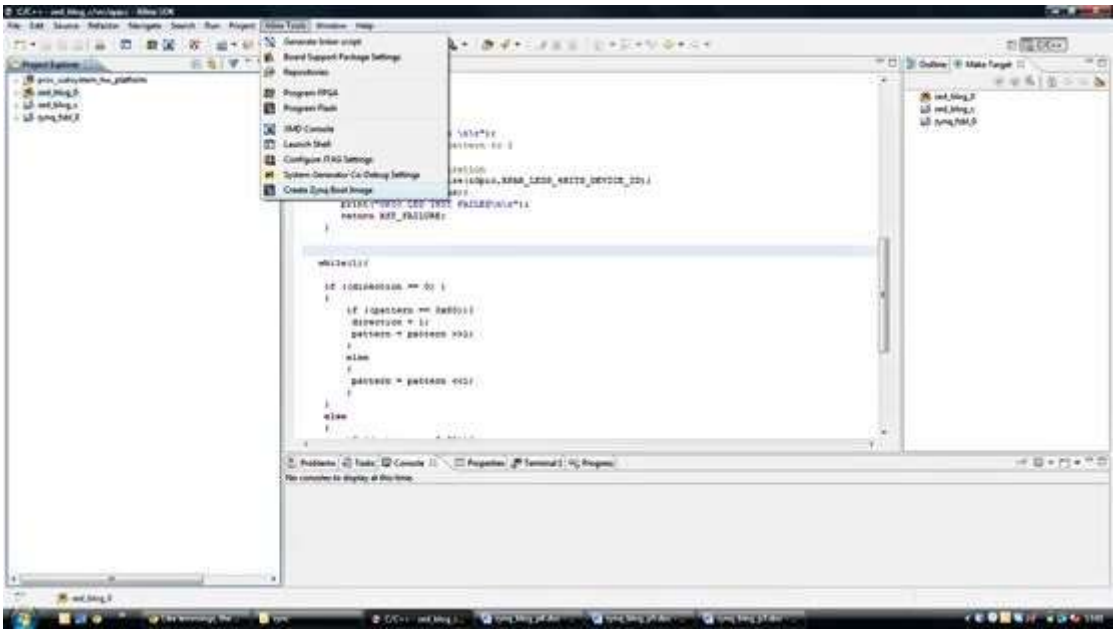


Figure 30 Options for creating the Zynq Boot Image

Once you have selected the "Create Zynq Boot Image" option, a new dialog box will open as shown below allowing you to select the required files as defined above.

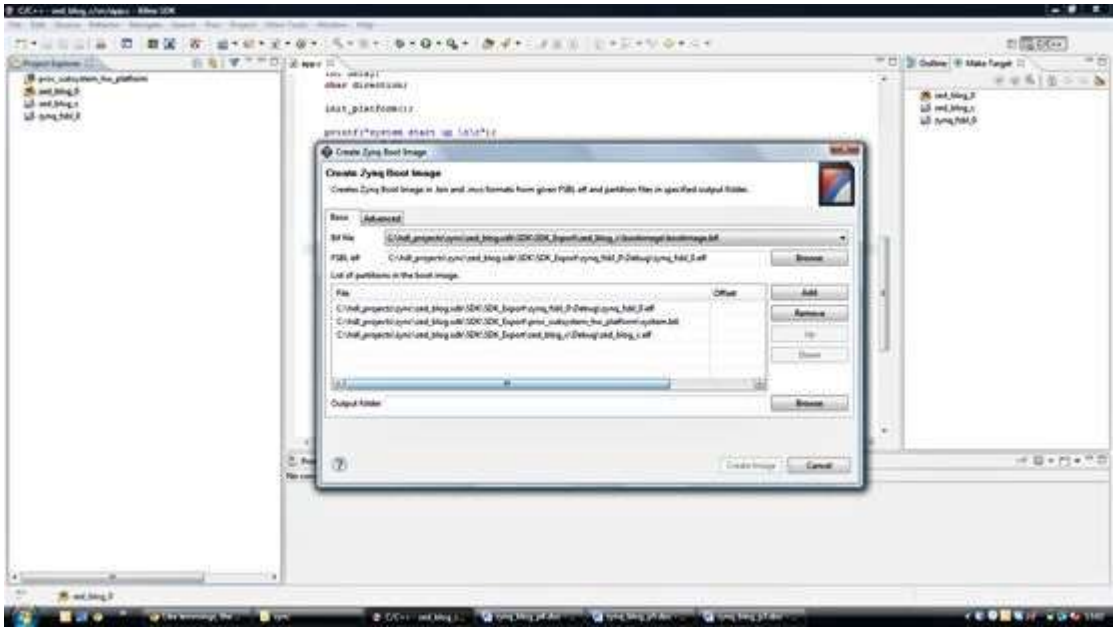


Figure 31 The complete files needed for the boot image

It is important to stress that the FPGA programming bit file must **always** follow the FSBL. Clicking on "create image" will create both a *.bin and a *.mcs file for the target device.

The MicroZed board design gives us the option of storing these generated files in either the on-board QSPI Flash memory, which requires a JTAG cable for programming, or within a microSD card plugged into the MicroZed's microSD card socket. The simpler approach is to use the microSD card. To program the microSD card, insert the SD card into the computer you're using to generate the boot file and transfer the .bin file to the microSD card. You then need to rename the file on the microSD card to boot.bin.

Insert the programmed microSD card into the MicroZed board, make sure the mode pins have been set to configure from the microSD card using the appropriate jumpers, and perform a reset. Your application should then boot.

If you wish to configure the on-board QSPI Flash, you'll need a JTAG cable. You can use the "program flash" option in the SDK or you can use Xilinx iMPACT to program the on-board QSPI Flash.

We'll use the QSPI interface in this example. We specify this option within the Vivado Design Suite as shown

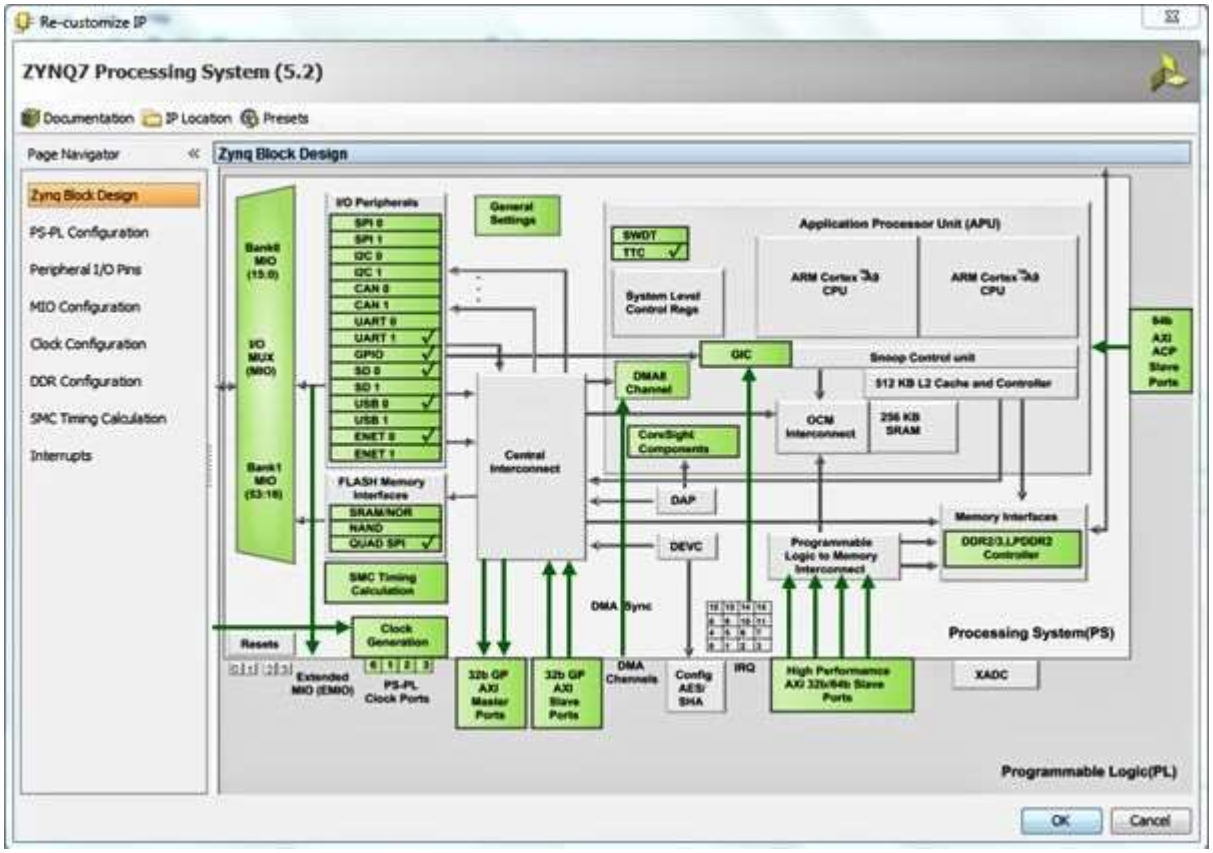


Figure 32 The Vivado view showing the Quad SPI memory option selected in the Flash memory interfaces

The Zynq hardware definition will contain the QSPI IP core and the location of the QSPI memory in the Zynq PS address map. This configuration allows the FSBL to access the QSPI memory and configure the Zynq SoC.

The next stage in programming the MicroZed board is to connect it to your PC using the JTAG programming cable. Programming the configuration memory is then as simple as selecting the “Program Flash” option from the Xilinx tools menu within the SDK as shown

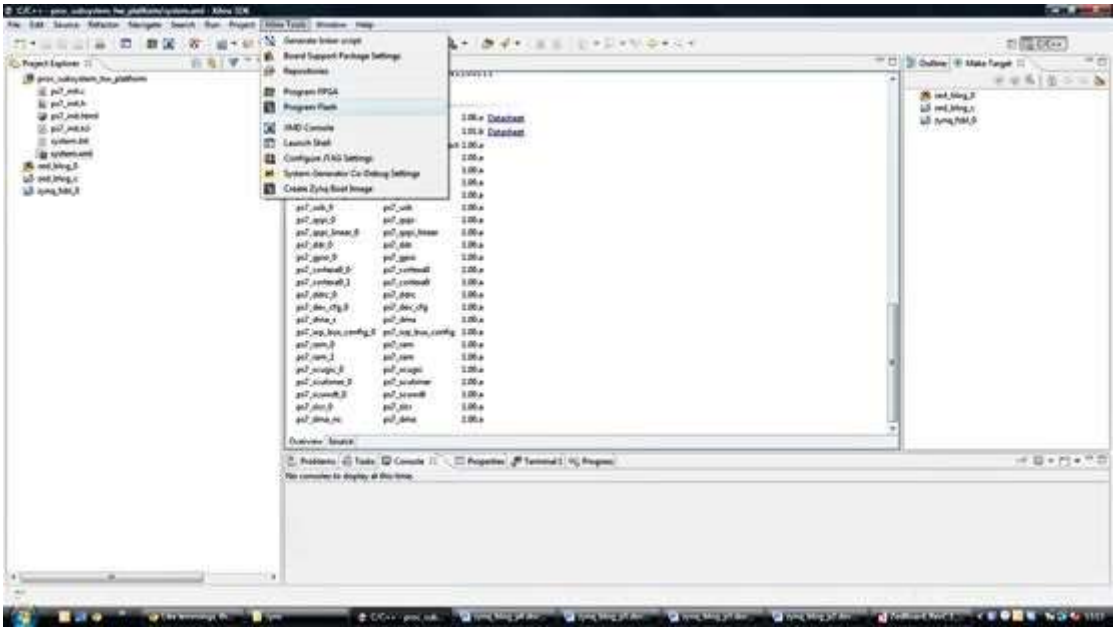


Figure 33 Programming the configuration memory

Navigate and select the MCS file you generated and enter an offset of 0x0 in the dialog box. Make sure your target hardware is powered on before you click on “program,” as shown

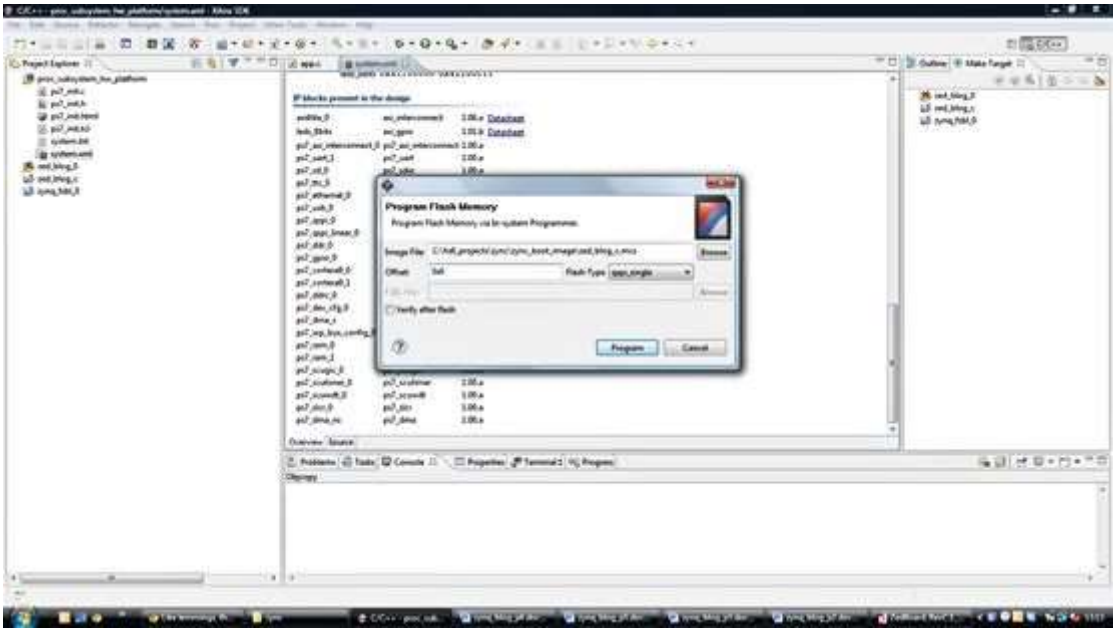


Figure 34 The Program Flash dialog

Programming the Flash memory may take a few minutes. Verification takes a little longer if you ticked the verify option. Once programming is complete, power down your hardware and the reapply the power. Alternatively, you can just reset the Zynq SoC. Assuming you have the mode pins configured correctly, your MicroZed board will first configure properly; then it will initiate the application; and then your board will spring to life.

The XADC Introduction

If you have been following the MicroZed Chronicles blogs, you will see we have run through the creation of a Zynq All Programmable SoC application, configuration, and boot loader file set using the Vivado Design Suite from concept through working prototype. However, we initially focused upon just using the processing system (PS) side of the Zynq. The real benefit of using a device like the Zynq SoC comes in the creation of a programmable system that also utilizes both the programmable logic (PL) side of the device and the dedicated, hard-IP macros in the Zynq SoC including theXADC analog subsystem, the high-speed serial (SerDes) links, and the PCIe end points.

Over the next few blogs, I will look at extending the system we have created to date by adding in the on-chip XADC block.

The Zynq SoC's XADC block contains two 12-bit analog-to-digital converters. These ADCs are capable of sampling at up to 1 Msample/sec, with an ideal effective input signal bandwidth of 500kHz (250 kHz on the auxiliary inputs). The XADC can multiplex among 17 analog inputs with additional input channels connected to on-chip voltages and temperature sensors. If your design is pin-limited in terms of available analog-capable inputs for external signals, you can configure the XADC to drive an external analog multiplexer.

The XADC is capable of unipolar or bipolar measurements—each analog input is differential. The 17 differential inputs are split between one dedicated analog input pair referred to as VP/VN and 16 auxiliary inputs that can be configured as either analog or digital I/O pins, referred to as VAuxP/VAuxN. The effective input signal bandwidth depends upon whether you are using the dedicated VP/VN differential input pair, in which case it is 500kHz, or the auxiliary inputs, in which case the maximum bandwidth is 250KHz.

The XADC's mixed-signal performance is very good, with a minimum 60-dB signal-to-noise ratio (SNR) and 70 dB of total harmonic distortion (THD) according to the data sheet. Depending upon the operating temperature range, -55 to 125°C or -40 to 100°C, the XADC's resolution is 10 bits or 12 bits respectively. This gives the XADC an equivalent number of bits of 9.67 when using the equation

$$ENOB = (SNR - 1.76 / 6.02)$$

(See Xcell Journal issue 80, "The FPGA Engineer's Guide to Using ADCs and DACs," for more detail on the theory behind this.)

The XADC supports user-selectable averaging to reduce input noise and offers 16-, 64- or 256-sample averaging. You can also program an automatic series of minimum and maximum alarm levels for each measured internal device parameter (voltage and temperature).

Designers can use the XADC for many applications ranging from simple housekeeping telemetry of on-chip parameters (voltage, current, temperature) to supporting touch sensors, motor control, or simple wireless communication protocols. The XADC can also be used in military or other critical systems to detect tampering attempts.

One great advantage is that you can use the XADC to monitor a number of internal device parameters to verify the health of your design. In addition, to ease verification during the early stages within a Zynq SoC-based system, you can use the XADC to measure the temperature as recorded by the on-chip temperature sensor, along with the following additional parameters:

- VCCInt: The internal PL core voltage

- VCCAux: The auxiliary PL voltage
- VRefP: The XADC positive reference voltage
- VRefN:The XADC negative reference voltage
- VCCBram: The PL BRAM voltage
- VCCPInt: The PS internal core voltage
- VCCPAux: The PS auxiliary voltage
- VCCDdr: The operating voltage of the DDR RAM connected to the PS

Adding in the XADC is very simple using the block-diagram editor within the Vivado Design Suite. The first thing to do is to ensure that we have enabled one of the general purpose master AXI interfaces within the Zynq PS on the PS-PL Configuration page:

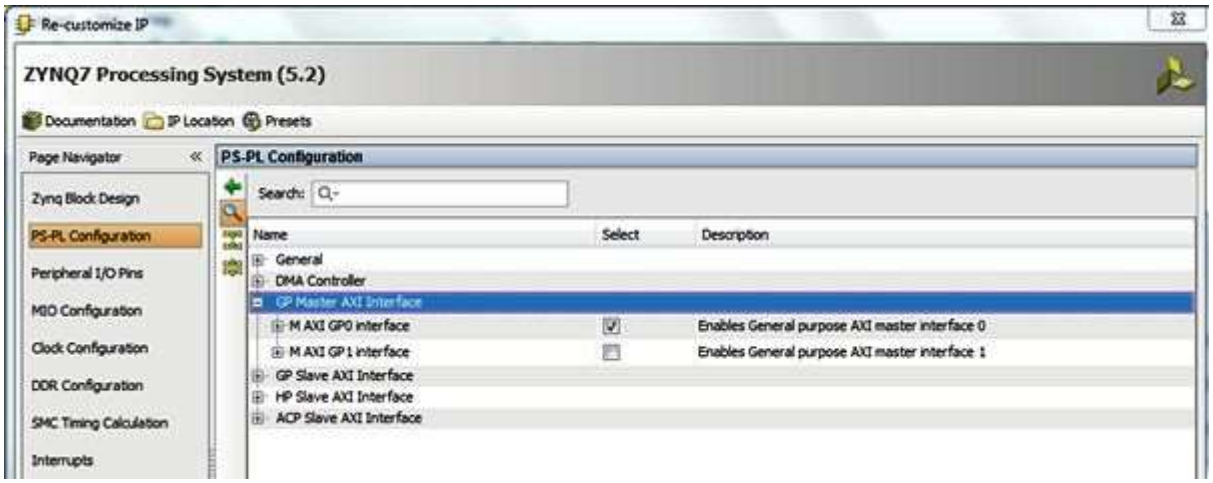


Figure 35 Enabling the General Purpose AXI Master Interface in Vivado

Once you have done this, you will see from the block diagram representation of the Zynq PS that the ports associated with the AXI interface are present (highlighted here in yellow):

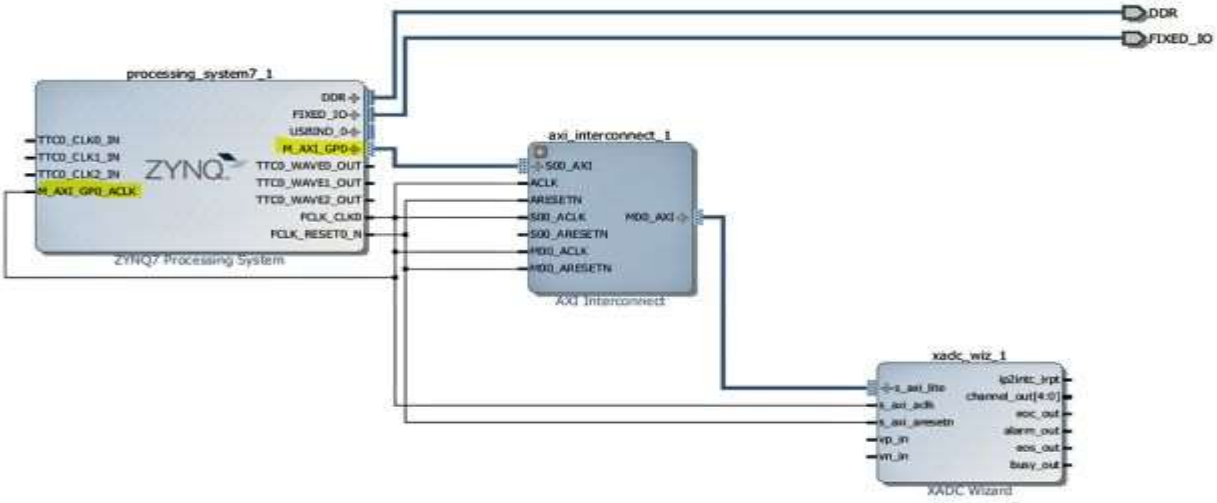


Figure 36 Adding the XADC and AXI Interconnect to the Vivado Block Diagram

The next thing to do is to connect this AXI port within the PS to an AXI Interconnect block. Doing this allows the Zynq PS to interface to the XADC's AXI 4 lite interface. You add the AXI Interconnect block to your block diagram from the Vivado IP Catalog. Once you have added it to the block diagram, connect the PS master AXI port to the AXI Interconnect Slave port. Connect FCLK_CLK0 from the PS to the AXI interconnect's S00_ACLK and master aclk. Similarly, connect the resets as shown in the diagram above.

Once the AXI Interconnect is wired to the PS we can customise the AXI Interconnect to select the number of slave and master ports. Click on the AXI interconnect module and you will be able to select the number of master and slave interfaces. In this example I selected one of each.

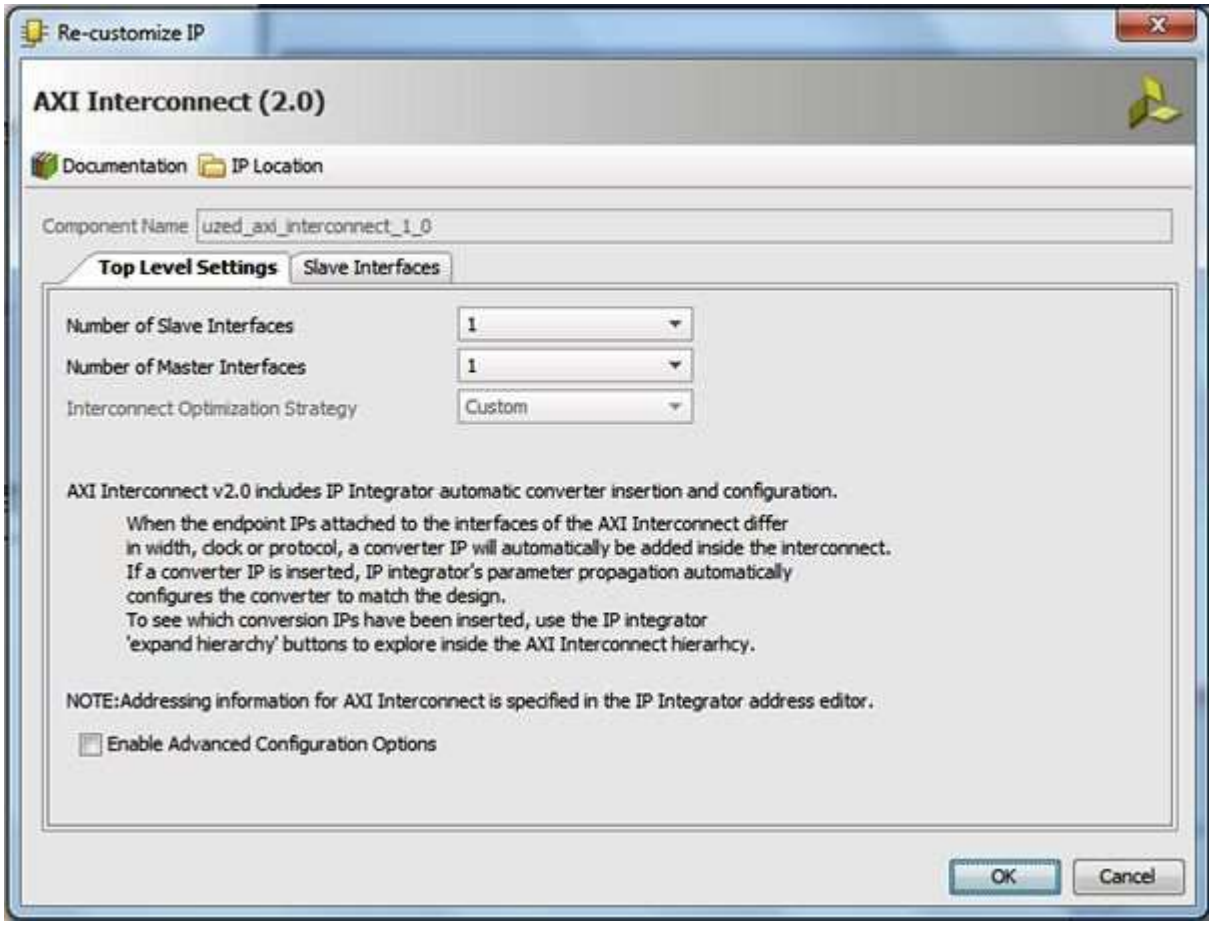


Figure 37 Customising the AXI Interconnect for the XADC connection

Once this is complete, you can include the XADC by adding it from the IP Catalog. Once you drop the XADC into your design, you can customize it to connect via the AXI4 interface using the XADC Wizard. Please note in this example we will be reading back the Zynq SoC's internal voltages and temperatures.

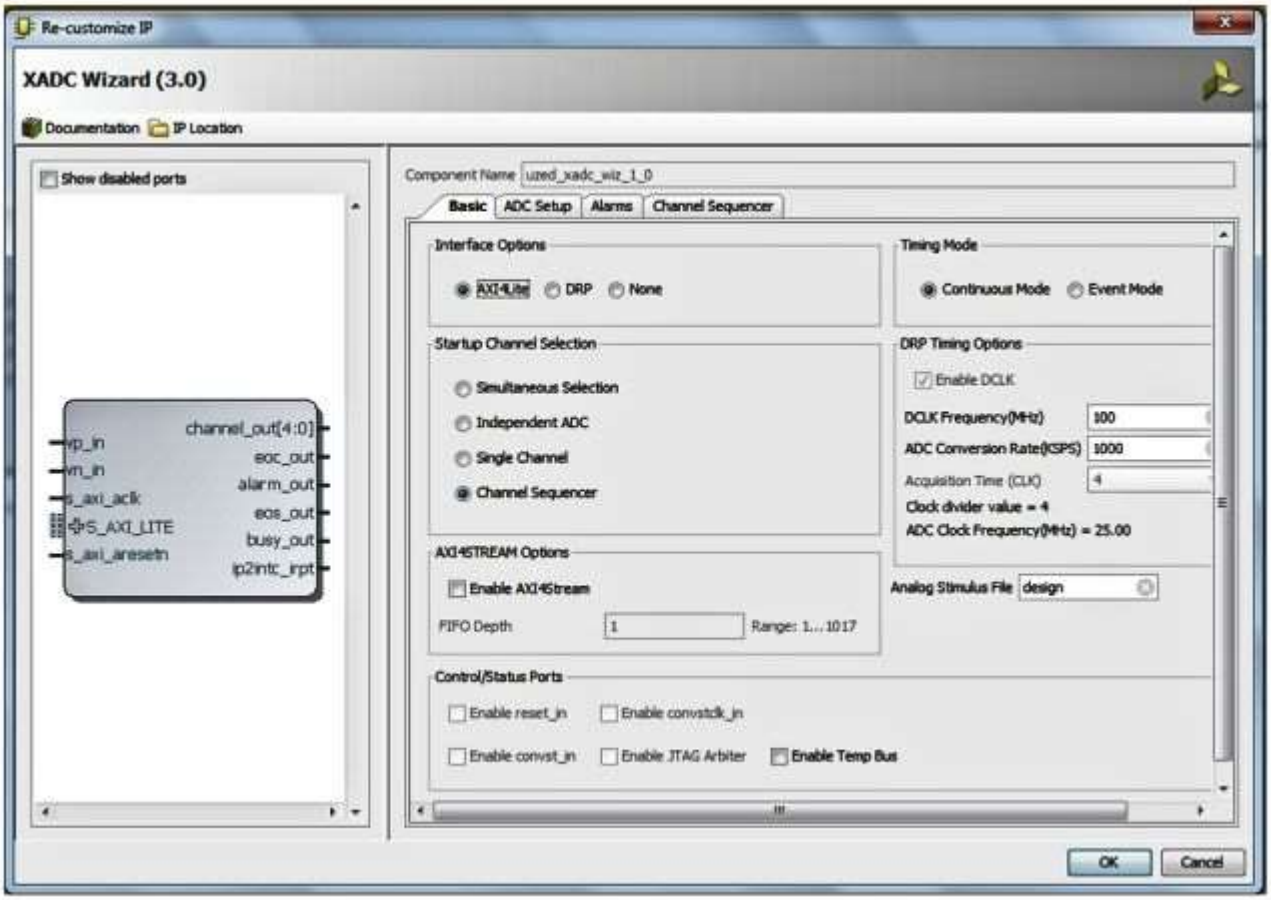


Figure 38 XADC Wizard

It is now a simple case of wiring up the XADC slave port to the AXI interconnect Master port, with the clocks and reset connected as for the AXI Interconnect.

Having connected all of the IP together, the next step is to verify that there are no errors or warnings on the system as we've designed it:

Figure 39 Successful validation following addition of the XADC

After design validation, right-click the sources window to select the block diagram and then select "generate HDL." Following this we can implement the design and generate the bit file. Once this process has finished, check the utilization report to ensure that the XADC is being used in the design:

Ref Name	Used
FDRE	666
LUT6	207
LUT4	200
BIBUF	130
LUT5	125
FDSE	98
LUT3	97
LUT2	62
SRLC32E	47
SRL16E	22
CARRY4	18
LUT1	3
XADC	1
PS7	1
BUFG	1

Figure 40 Implementation result showing XADC in use

Now the design can be exported to the SDK and then we can write the software needed to drive the XADC.

Using the XADC

When we last looked at the MicroZed we had just finished implementing the hardware and exporting it to the SDK. Once exported, you should be able to see the added XADC within the System.mss and System.xml files available under your board support package (BSP) and hardware definition respectively:

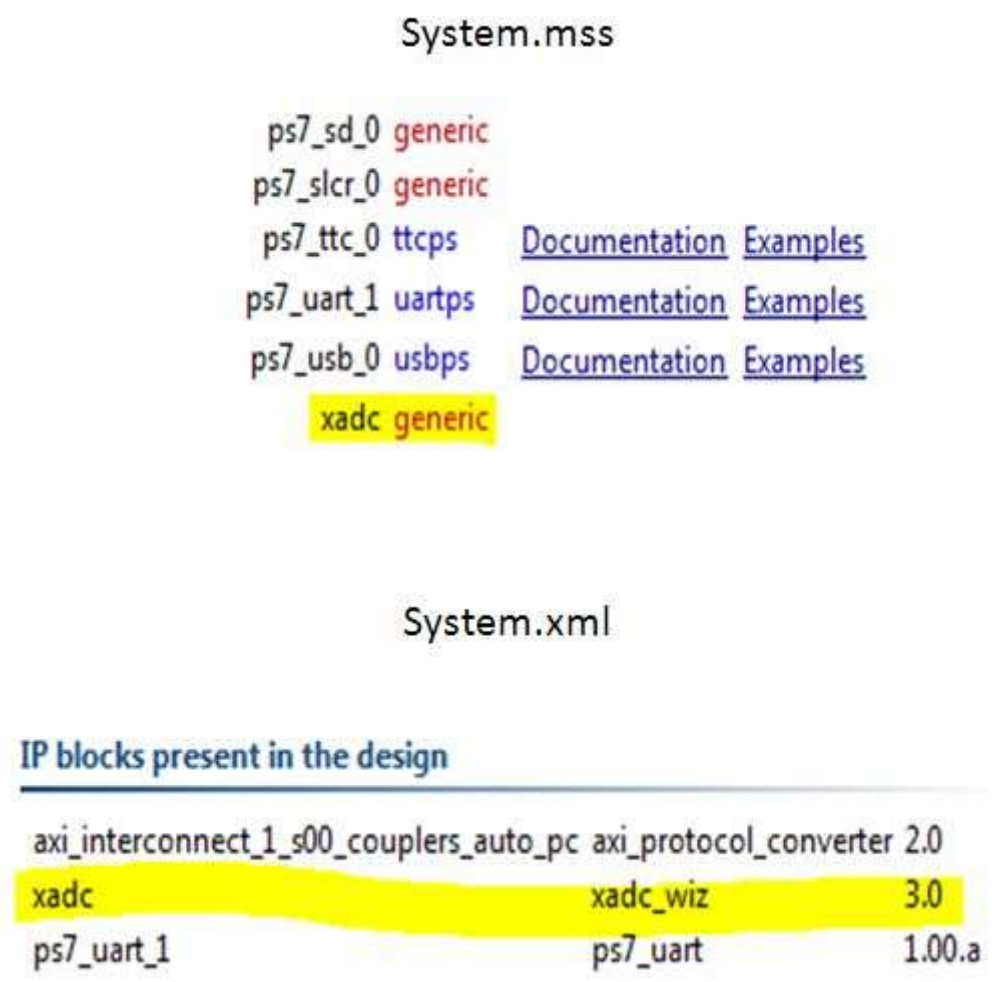


Figure 41 SDK board Support Package showing the XADC and its driver

System.mss will show the XADC and the driver type, which appears as “generic” in this import. In fact, the SDK does come with drivers for the XADC, which are defined with xadcps.h and are available under the include files within the BSP. As we will be using other driver types unique to the Zynq, I also included xil_types.h to support the u32 type.

The xil_types.h header defines a number of macros you can use to configure, control, and read from the XADC. Amongst other things, this header file contains definitions for the XADC registers, sampling averaging options, channel sequence options, and power-down modes. It also contains a series of type definitions, macros, and other functions.

For my simple example, I am going to read the Zynq SoC's internal temperature and voltage parameters and output them over an RS-232 link.

The first thing to do in the code is to look up the configuration of the XADC to be initialized; this requires a pointer of type "XAdcPs_Config." Using the function call "XAdcPs_LookupConfig()" coupled with the device ID this function will return 0 as there is only one XADC block within the Zynq. The configuration (device ID and base address of the XADC being initialized, see xparameters.h for this) will be stored in the pointer. If the configuration for the XADC with the device ID cannot be found, then "null" will be returned allowing the error to be handled.

The next step in the initialization process is to use the information previously obtained and stored within the configuration pointer, which requires a pointer of type "XAdcPs."

I named my configuration pointer "ConfigPtr" and my instantiation pointer "XADCInstPtr" (very original, as I'm sure you will agree).

Having initialized the XADC, the next few steps configure it for my example:

1. Use the "XAdcPs_SelfTest()" function to perform a self-test to check that there are no issues with the device.
2. Use "XAdcPs_SetSequencerMode()" to stop the sequencer from performing its current operation by setting it to a single channel.
3. Use "XAdcPs_SetAlarmEnables()" to disable any alarms that may be set.
4. Use "XAdcPs_SetSeqInputMode()" to restart the sequencer with the desired sequence.
5. Use "XAdcPs_SetSeqChEnables()" to configure the enables for the channels I wish to sample.

Reading a sample from the XADC can be as simple as calling the function "XAdcPs_GetAdcData()." For the internal temperature and voltage parameters, I then used two of the provided macros -- "XAdcPs_RawToTemperature()" and "XAdcPs_RawToVoltage()" -- to convert the raw XADC values into their real-world temperature or voltage equivalents.

```
VccIntRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VCCINT);
VccIntData = XAdcPs_RawToVoltage(VccIntRawData);
printf("Raw VccInt %lu Real VccInt %f \n\r", VccIntRawData, VccIntData);
```

Both these raw and real-world values are then output over an RS-232 link to be displayed in a terminal window. When I ran my code I was presented with the following results:

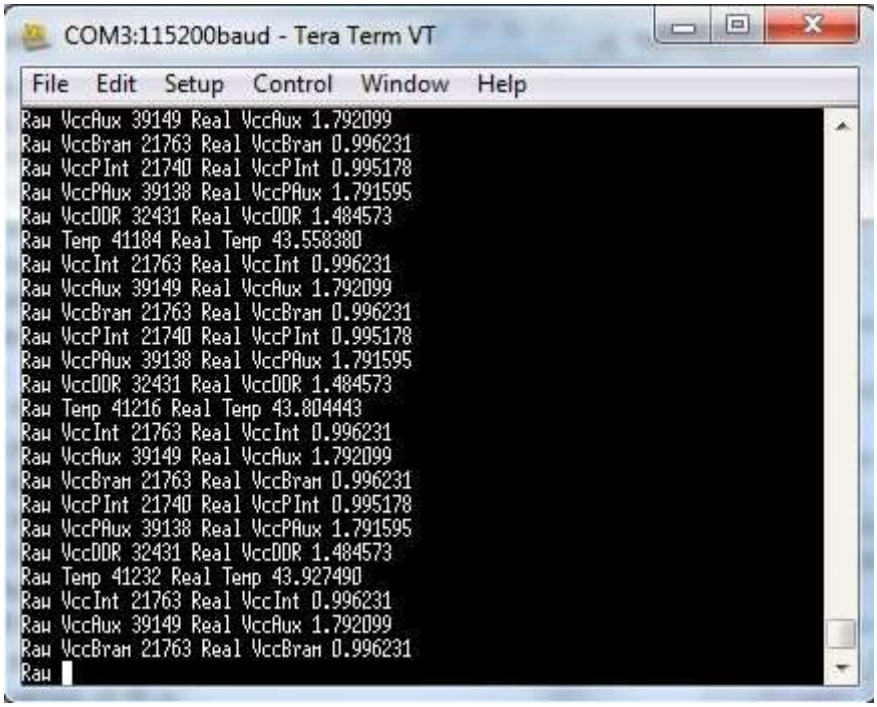


Figure 42 Results from the XADC

XADC C Source Code

```
#include <stdio.h>
#include "platform.h"
#include "xadcps.h"
#include "xil_types.h"
#define XPAR_AXI_XADC_0_DEVICE_ID 0


//void print(char *str);

static XAdcPs XADCMonInst;

int main()
{

      XAdcPs_Config *ConfigPtr;
      XAdcPs *XADCInstPtr = &XADCMonInst;

      //status of initialisation
      int Status_ADC;

      //temperature readings
      u32 TempRawData;
      float TempData;

      //Vcc Int readings
      u32 VccIntRawData;
      float VccIntData;

      //Vcc Aux readings
      u32 VccAuxRawData;
      float VccAuxData;

      //Vbram readings
      u32 VBramRawData;
      float VBramData;

      //VccPInt readings
      u32 VccPIntRawData;
      float VccPIntData;

      //VccPAux readings
      u32 VccPAuxRawData;
      float VccPAuxData;

      //Vddr readings
      u32 VDDRRawData;
      float VDDRData;

      init_platform();

    printf("Adam Edition MicroZed Using Vivado How To Printf \n\r");

    //XADC initilization

     ConfigPtr = XAdcPs_LookupConfig(XPAR_AXI_XADC_0_DEVICE_ID);
      if (ConfigPtr == NULL) {
```

```
            return XST_FAILURE;
      }

     Status_ADC = XAdcPs_CfgInitialize(XADCInstPtr,ConfigPtr,ConfigPtr-
>BaseAddress);
     if(XST_SUCCESS != Status_ADC){
         print("ADC INIT FAILED\n\r");
         return XST_FAILURE;
      }

     //self test
     Status_ADC = XAdcPs_SelfTest(XADCInstPtr);
      if (Status_ADC != XST_SUCCESS) {
            return XST_FAILURE;
      }

      //stop sequencer
      XAdcPs_SetSequencerMode(XADCInstPtr,XADCPS_SEQ_MODE_SINGCHAN);

     //disable alarms
     XAdcPs_SetAlarmEnables(XADCInstPtr, 0x0);

     //configure sequencer to just sample internal on chip parameters
     XAdcPs_SetSeqInputMode(XADCInstPtr, XADCPS_SEQ_MODE_SAFE);

     //configure the channel enables we want to monitor

XAdcPs_SetSeqChEnables(XADCInstPtr,XADCPS_CH_TEMP|XADCPS_CH_VCCINT|XADC
PS_CH_VCCAUX|XADCPS_CH_VBRAM|XADCPS_CH_VCCPINT|
            XADCPS_CH_VCCPAUX|XADCPS_CH_VCCPDRO);

     while(1){
     TempRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_TEMP);
      TempData = XAdcPs_RawToTemperature(TempRawData);
      printf("Raw Temp %lu Real Temp %f \n\r", TempRawData, TempData);

      VccIntRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VCCINT);
      VccIntData = XAdcPs_RawToVoltage(VccIntRawData);
      printf("Raw VccInt %lu Real VccInt %f \n\r", VccIntRawData,
VccIntData);

      VccAuxRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VCCAUX);
      VccAuxData = XAdcPs_RawToVoltage(VccAuxRawData);
      printf("Raw VccAux %lu Real VccAux %f \n\r", VccAuxRawData,
VccAuxData);

  //    VrefPRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VREFP);
  //    VrefPData = XAdcPs_RawToVoltage(VrefPRawData);
  //    printf("Raw VRefP %lu Real VRefP %f \n\r", VrefPRawData,
VrefPData);

  //    VrefNRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VREFN);
  //    VrefNData = XAdcPs_RawToVoltage(VrefNRawData);
   //   printf("Raw VRefN %lu Real VRefN %f \n\r", VrefNRawData,
VrefNData);

      VBramRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VBRAM);
      VBramData = XAdcPs_RawToVoltage(VBramRawData);
```

```
      printf("Raw VccBram %lu Real VccBram %f \n\r", VBramRawData,
VBramData);

      VccPIntRawData = XAdcPs_GetAdcData(XADCInstPtr,
XADCPS_CH_VCCPINT);
      VccPIntData = XAdcPs_RawToVoltage(VccPIntRawData);
      printf("Raw VccPInt %lu Real VccPInt %f \n\r", VccPIntRawData,
VccPIntData);

      VccPAuxRawData = XAdcPs_GetAdcData(XADCInstPtr,
XADCPS_CH_VCCPAUX);
      VccPAuxData = XAdcPs_RawToVoltage(VccPAuxRawData);
      printf("Raw VccPAux %lu Real VccPAux %f \n\r", VccPAuxRawData,
VccPAuxData);

      VDDRRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VCCPDRO);
      VDDRData = XAdcPs_RawToVoltage(VDDRRawData);
      printf("Raw VccDDR %lu Real VccDDR %f \n\r", VDDRRawData,
VDDRData);
     }

    return 0;
}
```

Multiplexed IO

The last few blogs in the MicroZed Chronicles have focused upon getting the MicroZed up and running and looking at the Zynq SoC's XADC. I thought it would be good in this blog to provide a little more detail on the Zynq SoC and how it works. This blog post focuses specifically on the Zynq SoC's Multipurpose IO (MIO) block. It is this interface block that provides the Zynq SoC's dual-core ARM Cortex-A9 MPCore processor with many standard interfaces. The MIO also contains the configuration settings that determine how the Zynq SoC boots.

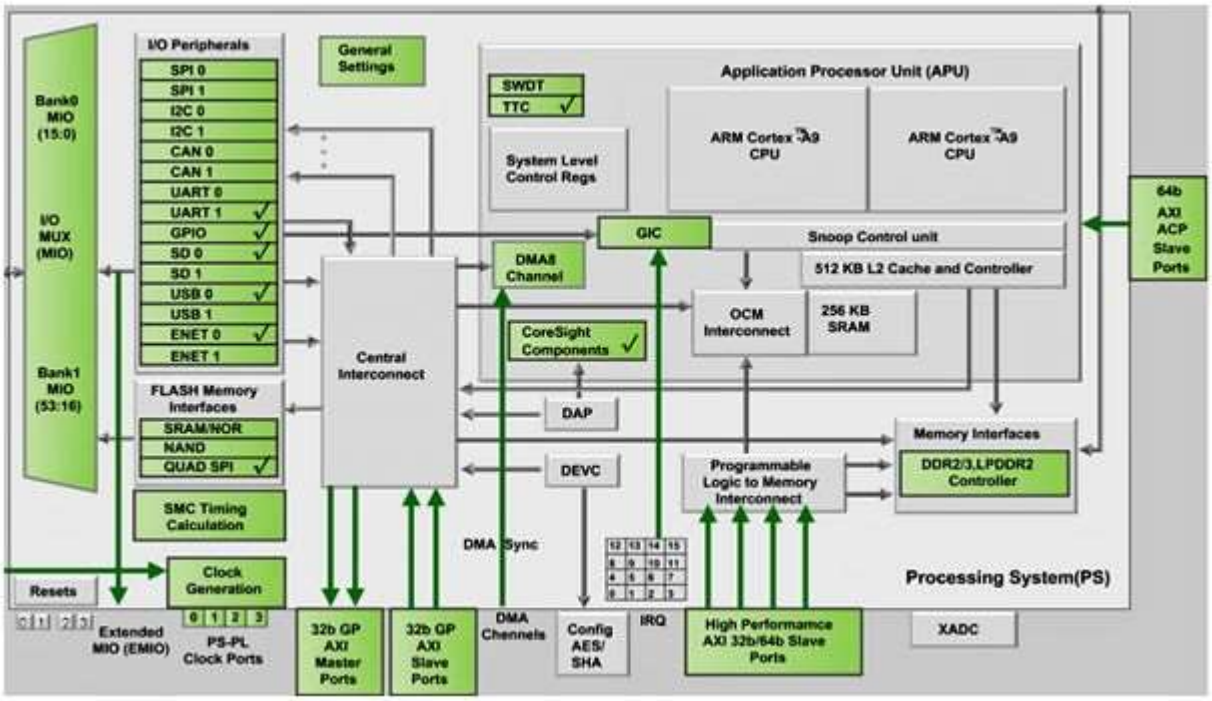


Figure 43 PS configuration view in Vivado

The MIO connects to the PS (processor system) side of the Zynq SoC. It connects to 54 pins on Zynq devices (note that the Zynq-7010 SoC in the CLG225 package has 32 MIO pins), which are used for the following:

- Defining the configuration method
- Quad SPI memory interface
- SRAM / NOR Flash memory interface
- NAND Flash memory interface
- Two 10/100/1000 Ethernet MACs
- Two USB 2.0 OTG interfaces
- Two SD Card interfaces
- Two UARTs
- Two master and slave I2C interfaces
- Two full-duplex SPI interfaces
- Two CAN 2.0B interfaces
- PJTAG and TRACE debug interfaces
- Triple timer/counter (TTC)

- System watchdog timer

I think you will agree it is a pretty impressive list standard interfaces and devices. You will of course end up in a situation at times where you need to trade off the many interfaces with the available pin count. Engineering is, after all, always an art of compromise.

Using the Vivado design flow, you assign functions within the MIO by double clicking on the processor within your block diagram, which brings up the re-customize IP window. There are two options for defining the MIO. The first option—Peripheral I/O Pins—is very graphical and allows you to see how assigning one interface standard affects others, as shown in the image below:

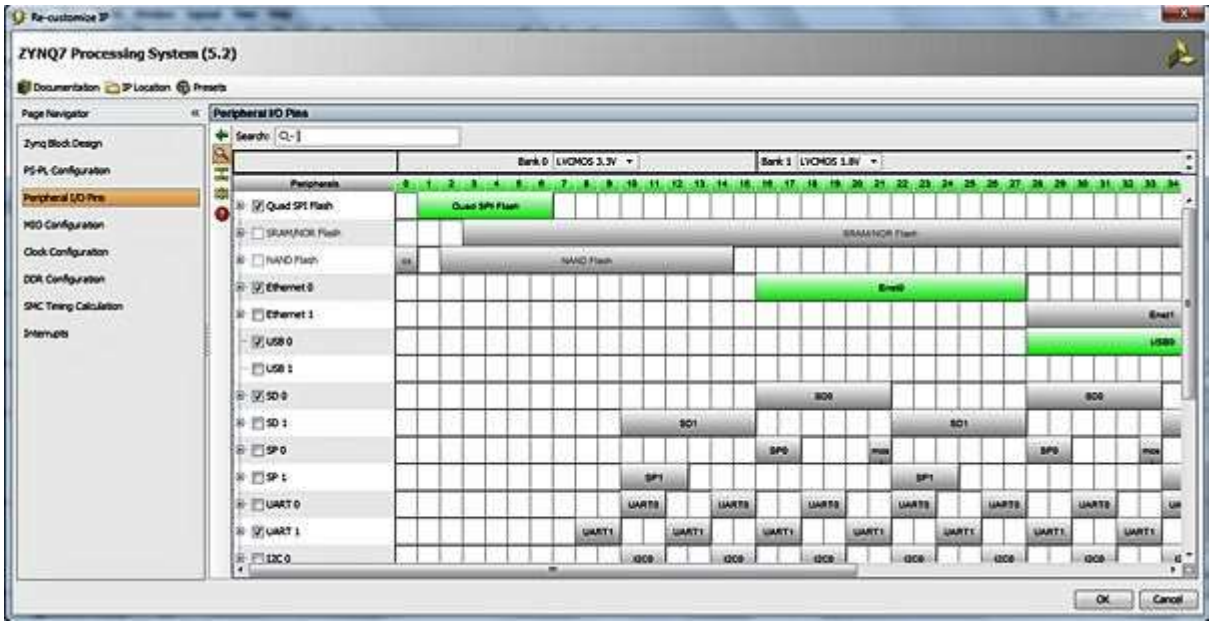


Figure 44 Configuring the MIO

You can also use this window to define the bank-voltage settings for each of the two I/O banks (green = active).

The second option is the “MIO Configuration” tab in the Zynq Processing System screen, shown below, which brings up a list of interfaces assigned to the MIO. We can also assign the EMIO pins in this view, which we will address in a little while.

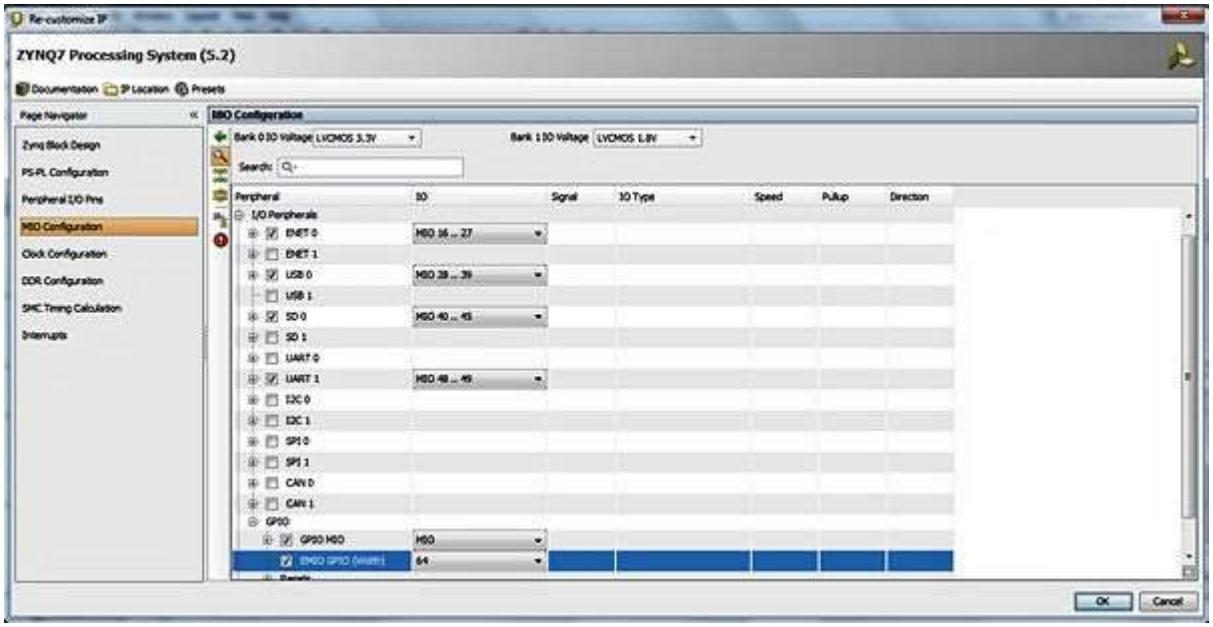


Figure 45 Configuring the MIO

The MIO is split into two voltage banks:

1. MIO0 pins 0 to 15
2. MIO1 pins 16 to 53

Bank 0 includes the configuration input pins, which are sampled following power up. These mode pins share the multiuse I/O pins on the PS side of the device. In all, there are seven mode pins mapped to MIO[8:2]. The first four mode pins define the boot mode; the fifth determines whether the PLL is used or not; and the sixth and seventh mode pins define the bank voltages on MIO bank 0 and bank 1 during power up. The voltage standard defined on MIO bank 0 and 1 can be changed from LVCMOS to HTSL following completion of the boot loader.

As mentioned above, at times there are not enough MIO pins to bring out all the interfaces you wish to have. In such cases, you can extend the MIO into the Programmable Logic (PL) side of the Zynq SoC. This is called Extended Multipurpose IO or EMIO. EMIO can provide up to 64 additional GPIO pins. Alternatively you can assign most of the MIO interfaces to the EMIO with the notable exception of the USB, SRAM/NOR memory interfaces, and the NAND Flash interface. The Zynq SoC technical reference manual provides very detailed information on the differences between MIO and EMIO capabilities. Assigning functions to the EMIO is very simple and is accomplished by clicking the EMIO button at the end of the Peripheral I/O Pins tab as shown below:

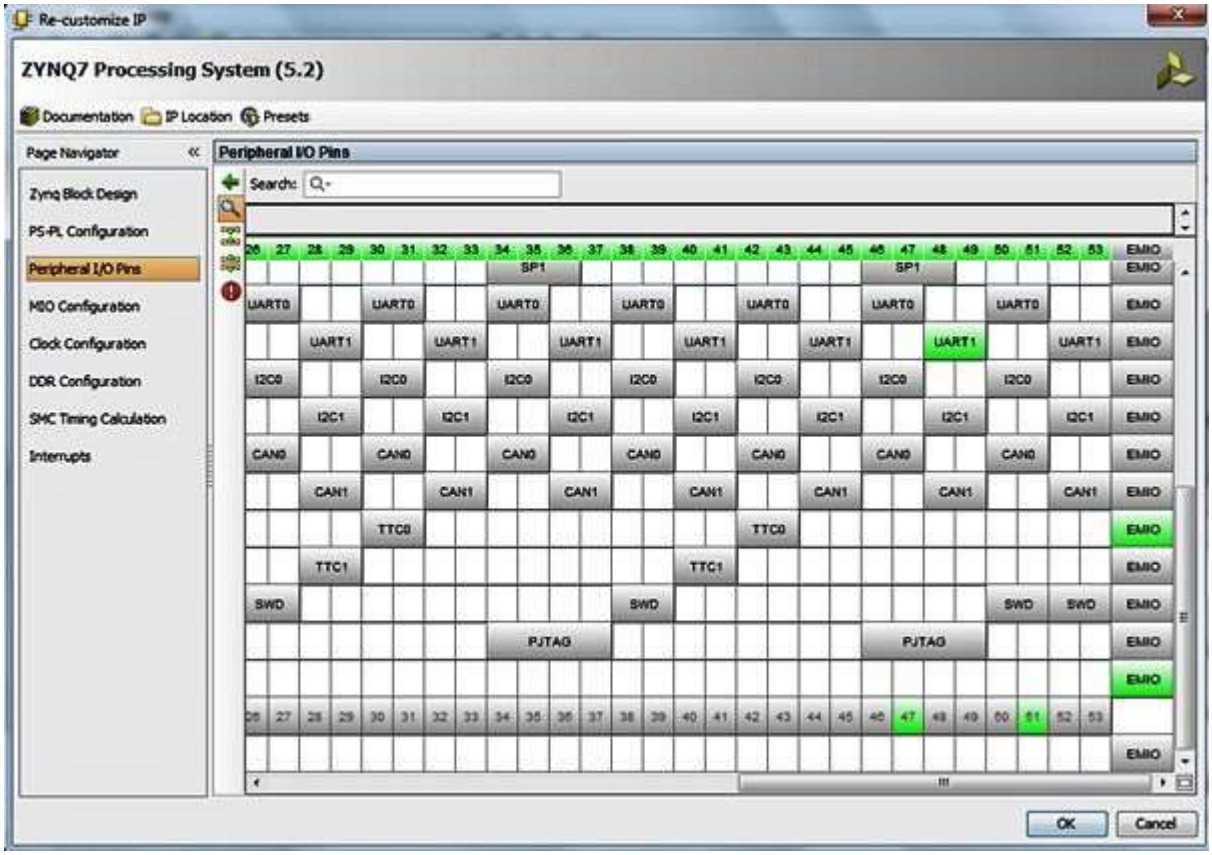


Figure 46 Using EMIO extension for MIO interfaces

The GPIO setting can be enabled and its size selected from the MIO Configuration option tab. The GPIO will be split into two banks of 32 bits each if the maximum 64-bit size is selected.

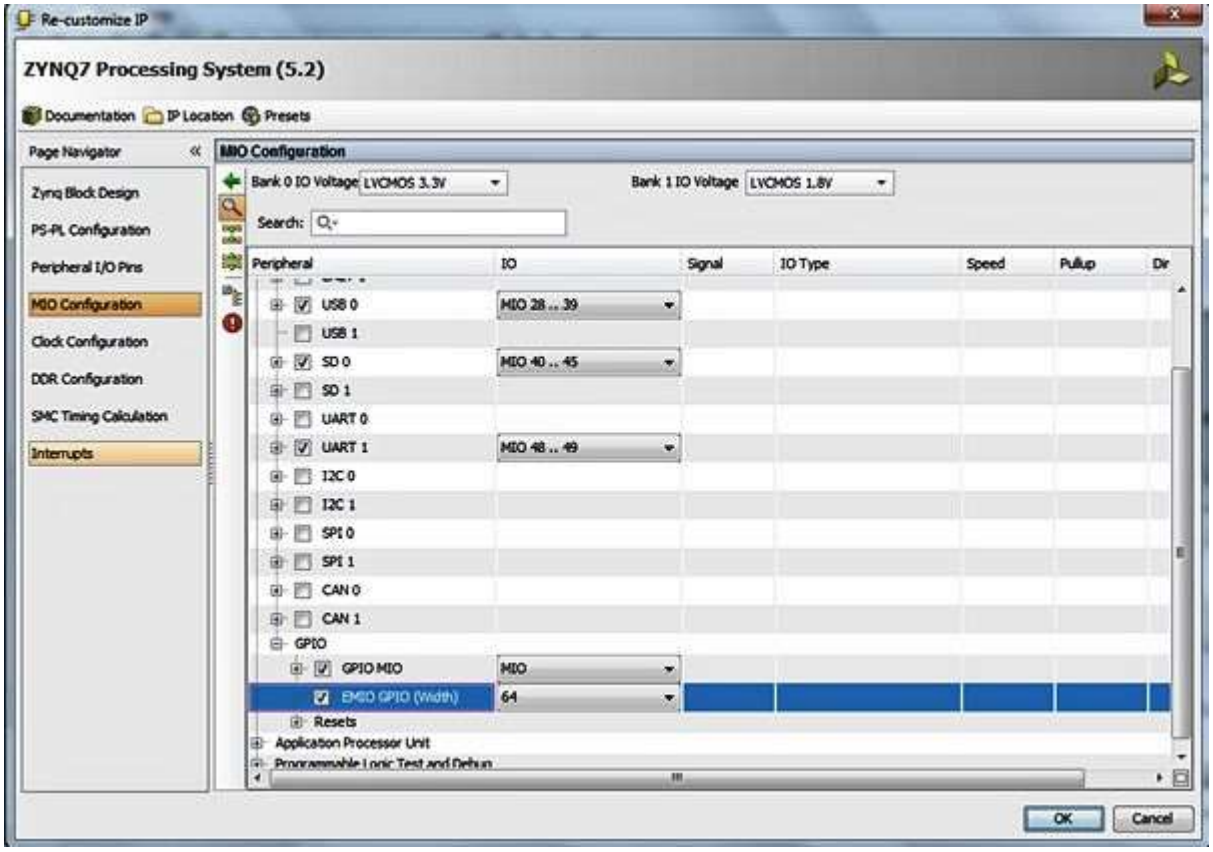


Figure 47 Enabling the EMIO

When you close the re-customize IP option, you will see that the additional ports you selected have been added to the PS within your block diagram:

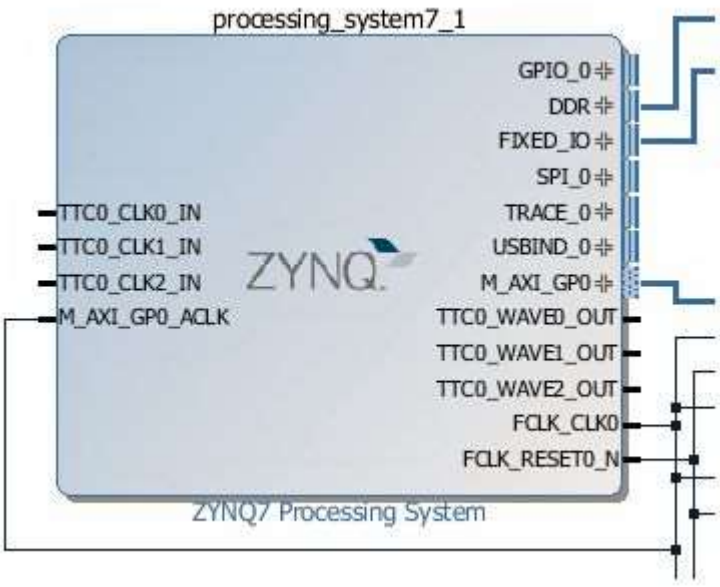


Figure 48 EMIO Interfaces on the PS in the block diagram when enabled

The example above shows the PS when GPIO_0, SPI_0 TRACE_0, and TTC_0 are assigned to the EMIO. These functions can then be assigned as external IO and will be present within the re-generated HDL netlist.

Note: Because the EMIO is within the PL side of the Zynq SoC, do not forget to the enable the level shifters between the PS and PL to ensure correct operation.

Driving the MIO

My previous blog post looked at the Zynq All Programmable SoC's MIO and EMIO, this blog looks at driving a LED connected as a GPIO. While blinking an LED may seem a very simple task, examining the steps needed to blink the LED allows us to then explore further aspects of the Zynq SoC such as its timers and interrupts. I'll be looking at those topics in later blogs.

The Zynq SoC has a number general purpose I/O pins that combine to create a 10-bit-wide, general-purpose I/O port, as can be seen below.

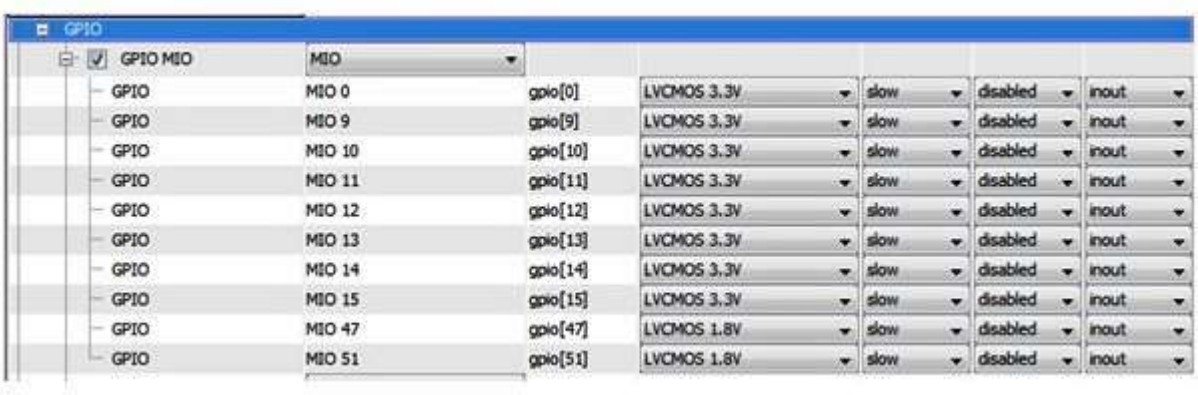


Figure 49 GPIO on the MicroZed

As you can see this GPIO bank is split across both MIO banks with a mixture of voltages. For this example, our LED will be connected to MIO 47.

Xilinx provides a number of drivers to simplify use of the Zynq SoC's GPIO. Links to supporting documentation and examples can be found linked in the system.mss file, available in your board support package:

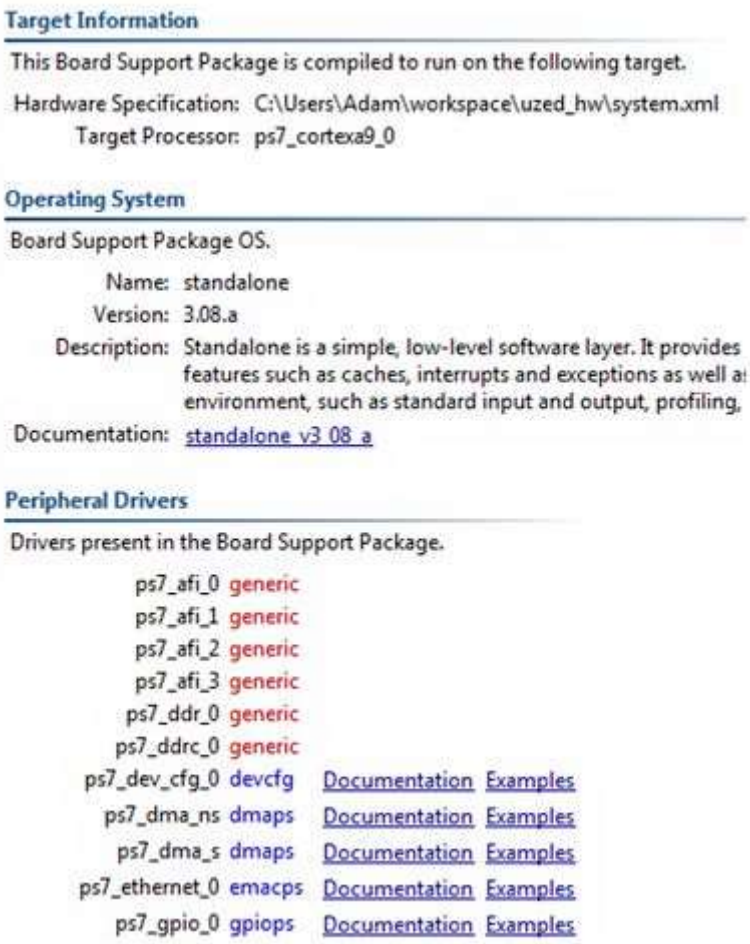


Figure 50 Board Support System.mss file provides documentation and examples modules

The code needed to drive the GPIO is very straightforward. The file xparameters.h contains the number of GPIO instances, the GPIO device id within the system, and the upper and lower address ranges:

```
/* Definitions for driver GPIOPS */
#define XPAR_XGPIOPS_NUM_INSTANCES 1

/* Definitions for peripheral PS7_GPIO_0 */
#define XPAR_PS7_GPIO_0_DEVICE_ID 0
#define XPAR_PS7_GPIO_0_BASEADDR 0xE000A000
#define XPAR_PS7_GPIO_0_HIGHADDR 0xE000AFFF
```

Macros and functions required to drive the IO are defined with the file xgpiops.h, which is available under the BSP include files. This file contains the needed configuration and initialization functions for the Zynq SoC's GPIO, along with functions to support reading from and writing to the GPIO, which is after all what we are most interested in.

To blink the LED, we need to do the following:

1. Include the xgpiops.h file within the main application code

```
#include "xgpiops.h"
```

2. Define the output pin we want to toggle. In this case it is pin 8 within the bank of 10, MIO 47:

```
#define ledpin 47
```

3. Declare the driver instance:

```
XGpioPs Gpio;
```

4. Configure the GPIO and define the status and pointer variables required for initialization within the function you wish to use:

```
int Status;
XGpioPs_Config *GPIOConfigPtr;
```

5. Initialize the GPIO Driver:

```
//GPIO Initilization
GPIOConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
Status = XGpioPs_CfgInitialize(&Gpio, GPIOConfigPtr, GPIOConfigPtr ->BaseAddr);
if (Status != XST_SUCCESS) {
return XST_FAILURE;
}
```

6. Set the direction of the GPIO pin and enable the output:

```
XGpioPs_SetDirectionPin(&Gpio, ledpin, 1);
XGpioPs_SetOutputEnablePin(&Gpio, ledpin, 1);
```

7. Write the desired output value to the GPIO pin:

```
XGpioPs_WritePin(&Gpio, ledpin, 0x0);
```

This write function can be used within a loop to get the LED flashing at a desired rate however you can also use this approach to drive anything connected to the GPIO pin. Reading the GPIO pins is achieved in a similar manner using theXGpioPs_ReadPin(&Gpio,INPUT_PIN) function.

It is well worth spending some time reading the documentation and examples provided because the Zynq SoC's GPIO is a very flexible resource.

GPIO Polled Input

We looked at driving an LED from Zynq SoC's MIO in the previous instalment of this series and I mentioned that reading the status of a pushbutton would be very similar. However, I now think this topic merits a blog of its own as it completes the description of how you can use the Zynq SoC's MIO and GPIO and it opens the door nicely on to the topic of interrupts, which I'll describe in the next blog.

The MicroZed board's pushbutton is connected to the Zynq SoC's GPIO pin 51. For this example I intend to use software to connect the pushbutton's state to the LED that we discussed in the previous blog. For this example, each press of the button will toggle the LED's state.

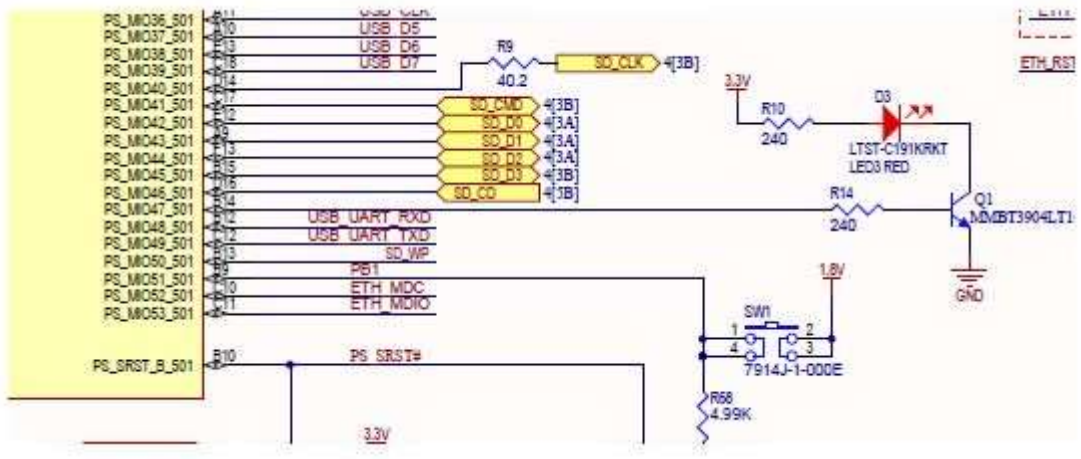


Figure 51 The MicroZed schematic showing both the LED on pin 47 and pushbutton switch on Pin 51

The first thing we need to do is declare the pin we wish to us the input within our application:

```
#define pbsw 51
```

This is going to be a polled application so the main program loop will read the switch status. There are two methods of reading inputs from the GPIO: polled or interrupt driven. We have not looked at the Zynq's interrupts to date so we will opt for the simpler polled method in this example. Interrupts will come later in this series.

All mechanical switches exhibit contact bounce, which is the result of the switch contacts physically striking together. The contacts' momentum and elasticity act together to cause them to bounce apart one or more times before settling down and making steady contact. The following scope trace shows what switch bounce typically looks like.

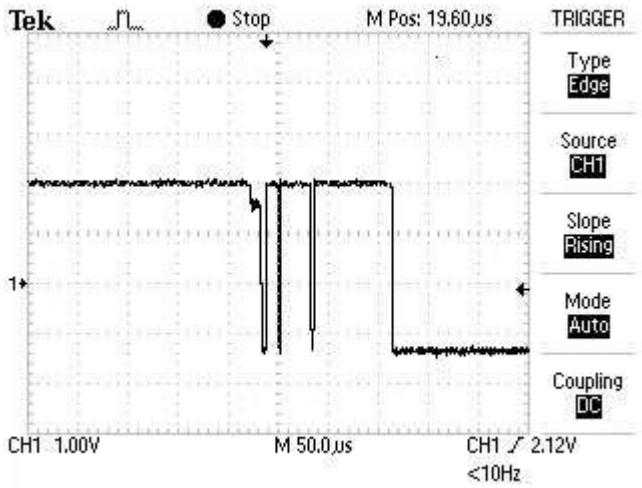


Figure 52 Typical Switch Bounce

It is therefore common to employ either hardware or software to debounce the switch. As the schematic above shows, the MicroZed board design provides no debounce circuitry for the pushbutton switch, so we must perform this task with software. Otherwise, each press of the MicroZed board's pushbutton would produce an indeterminate number of triggers to our LED control software and we would not be able to reliably control the state of the LED.

The application we are creating in this example is very simple and my approach to debouncing the switch is also going to be simple. I can debounce the switch simply by sampling the switch at a low frequency to ensure that I do not get multiple changes on the LED following a button press. I also decided that I would output the value of the LED pin over the RS232 link along with the ADC values from the Zynq SoC's XADC mixed-signal subsystem, etc.

In the previous post in this series, we had already initialized the GPIO. Therefore, all that remains to use pin 51 as an input is to configure the pin as an input using the set direction function:

```
//GPIO Initilization – Existing from the last blog
GPIOConfigPtr = XGpioPs_LookupConfig(XPAR_XGPIOPS_0_DEVICE_ID);
    Status = XGpioPs_CfgInitialize(&Gpio, GPIOConfigPtr,GPIOConfigPtr->BaseAddr);
      if (Status != XST_SUCCESS) {
             print("GPIO INIT FAILED\n\r");
            return XST_FAILURE;
      }

    //set direction and enable output- Existing from the last blog

    XGpioPs_SetDirectionPin(&Gpio, ledpin, 1);
    XGpioPs_SetOutputEnablePin(&Gpio, ledpin, 1);

    //set direction input pin – New in this blog
    XGpioPs_SetDirectionPin(&Gpio, pbsw, 0x0);
```

Reading the input pin is also very simple using the ReadPin function, which is provided by **xgpiops.h**. This value can be stored within an u32 (unsigned 32-bit integer) result as demonstrated below:

```
sw = XGpioPs_ReadPin(&Gpio, pbsw); //read pin

   if (sw == 1) { //sw=1 when switch is pushed
     toggle = !toggle; //invert value stored in toggle

   }
   printf("switch value %lu \n\r",sw); //output the sw value
   XGpioPs_WritePin(&Gpio, ledpin, toggle); //set the LED
```

When this program was run on the MicroZed, a button press toggled the LED each time it as pressed the picture below shows the results from the RS-232 output:

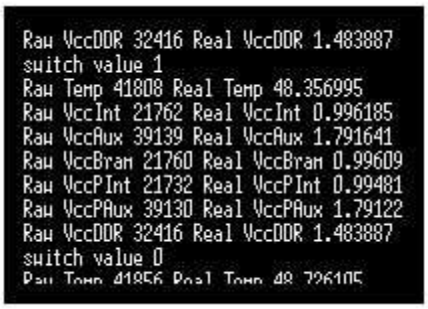


Figure 53 Results of reading the switch

GPIO Polled Source Code

```
#include <stdio.h>
#include "platform.h"
#include "xadcps.h"
#include "xgpiops.h"
#include "xil_types.h"

#define XPAR_AXI_XADC_0_DEVICE_ID 0

#define ledpin 47
#define pbsw 51
#define LED_DELAY     10000000
//void print(char *str);

static XAdcPs XADCMonInst;
XGpioPs Gpio;    /* The driver instance for GPIO Device. */

int main()
{

      XAdcPs_Config *ConfigPtr;
      XAdcPs *XADCInstPtr = &XADCMonInst;

      int Status;
      int delay;
      u32 sw;
      int toggle;
      XGpioPs_Config *GPIOConfigPtr;

      //status of initialisation
      int Status_ADC;

      //temperature readings
      u32 TempRawData;
      float TempData;

      //Vcc Int readings
      u32 VccIntRawData;
      float VccIntData;

      //Vcc Aux readings
      u32 VccAuxRawData;
      float VccAuxData;

      //Vbram readings
      u32 VBramRawData;
      float VBramData;

      //VccPInt readings
      u32 VccPIntRawData;
      float VccPIntData;

      //VccPAux readings
      u32 VccPAuxRawData;
      float VccPAuxData;
```

```
      //Vddr readings
      u32 VDDRRawData;
      float VDDRData;

      init_platform();

    printf("Adam Edition MicroZed Using Vivado How To Printf \n\r");

    //GPIO Initilization
    GPIOConfigPtr = XGpioPs_LookupConfig(XPAR_XGPIOPS_0_DEVICE_ID);
    Status = XGpioPs_CfgInitialize(&Gpio, GPIOConfigPtr,GPIOConfigPtr-
>BaseAddr);
      if (Status != XST_SUCCESS) {
             print("GPIO INIT FAILED\n\r");
            return XST_FAILURE;
      }

    //set direction and enable output
    XGpioPs_SetDirectionPin(&Gpio, ledpin, 1);
    XGpioPs_SetOutputEnablePin(&Gpio, ledpin, 1);

    //set direction input pin
    XGpioPs_SetDirectionPin(&Gpio, pbsw, 0x0);

    //XADC initilization

     ConfigPtr = XAdcPs_LookupConfig(XPAR_AXI_XADC_0_DEVICE_ID);
      if (ConfigPtr == NULL) {
            return XST_FAILURE;
      }

     Status_ADC = XAdcPs_CfgInitialize(XADCInstPtr,ConfigPtr,ConfigPtr-
>BaseAddress);
     if(XST_SUCCESS != Status_ADC){
         print("ADC INIT FAILED\n\r");
         return XST_FAILURE;
      }

     //self test
     Status_ADC = XAdcPs_SelfTest(XADCInstPtr);
      if (Status_ADC != XST_SUCCESS) {
            return XST_FAILURE;
      }

      //stop sequencer
      XAdcPs_SetSequencerMode(XADCInstPtr,XADCPS_SEQ_MODE_SINGCHAN);

     //disable alarms
     XAdcPs_SetAlarmEnables(XADCInstPtr, 0x0);

     //configure sequencer to just sample internal on chip parameters
     XAdcPs_SetSeqInputMode(XADCInstPtr, XADCPS_SEQ_MODE_SAFE);

     //configure the channel enables we want to monitor

XAdcPs_SetSeqChEnables(XADCInstPtr,XADCPS_CH_TEMP|XADCPS_CH_VCCINT|XADC
PS_CH_VCCAUX|XADCPS_CH_VBRAM|XADCPS_CH_VCCPINT|
            XADCPS_CH_VCCPAUX|XADCPS_CH_VCCPDRO);
```

```
     while(1){



     TempRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_TEMP);
      TempData = XAdcPs_RawToTemperature(TempRawData);
      printf("Raw Temp %lu Real Temp %f \n\r", TempRawData, TempData);

      VccIntRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VCCINT);
      VccIntData = XAdcPs_RawToVoltage(VccIntRawData);
      printf("Raw VccInt %lu Real VccInt %f \n\r", VccIntRawData,
VccIntData);

      VccAuxRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VCCAUX);
      VccAuxData = XAdcPs_RawToVoltage(VccAuxRawData);
      printf("Raw VccAux %lu Real VccAux %f \n\r", VccAuxRawData,
VccAuxData);

  //    VrefPRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VREFP);
  //    VrefPData = XAdcPs_RawToVoltage(VrefPRawData);
  //    printf("Raw VRefP %lu Real VRefP %f \n\r", VrefPRawData,
VrefPData);

  //    VrefNRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VREFN);
  //    VrefNData = XAdcPs_RawToVoltage(VrefNRawData);
   //   printf("Raw VRefN %lu Real VRefN %f \n\r", VrefNRawData,
VrefNData);

      VBramRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VBRAM);
      VBramData = XAdcPs_RawToVoltage(VBramRawData);
      printf("Raw VccBram %lu Real VccBram %f \n\r", VBramRawData,
VBramData);

      VccPIntRawData = XAdcPs_GetAdcData(XADCInstPtr,
XADCPS_CH_VCCPINT);
      VccPIntData = XAdcPs_RawToVoltage(VccPIntRawData);
      printf("Raw VccPInt %lu Real VccPInt %f \n\r", VccPIntRawData,
VccPIntData);

      VccPAuxRawData = XAdcPs_GetAdcData(XADCInstPtr,
XADCPS_CH_VCCPAUX);
      VccPAuxData = XAdcPs_RawToVoltage(VccPAuxRawData);
      printf("Raw VccPAux %lu Real VccPAux %f \n\r", VccPAuxRawData,
VccPAuxData);

      VDDRRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VCCPDRO);
      VDDRData = XAdcPs_RawToVoltage(VDDRRawData);
      printf("Raw VccDDR %lu Real VccDDR %f \n\r", VDDRRawData,
VDDRData);

   //   XGpioPs_WritePin(&Gpio, ledpin, 0x0);

         for( delay = 0; delay < LED_DELAY; delay++)//wait
         {}

      sw = XGpioPs_ReadPin(&Gpio, pbsw);
```

```
      if (sw == 1) {
        toggle = !toggle;

      }
      printf("switch value %lu \n\r",sw);
      XGpioPs_WritePin(&Gpio, ledpin, toggle);

     }

    return 0;
}
```

Zynq Interrupts

When we last looked at the MicroZed we had focused upon a polled example to read in the state of the push button switch. I had mentioned that it was possible to also use an interrupt driven approach which is the approach typically used in most systems. An interrupt driven approach is required as many systems will have a number of inputs keyboards, mice, push buttons, sensors etc. depending upon the system which will at times require processing. Inputs from these devices are normally asynchronous to the process or task currently executing, this is that you cannot always predict when the event will occur.

Using interrupts enables the processor to continue processing until an event occurs the processor can then address this event. Interrupts at the highest level can be split into two types maskable and non-maskable, however as processors get more advanced there is a number of sources interrupts can come from the Zynq uses a Generic Interrupt Controller (CIG) to process interrupts which can come from

- Software Generated Interrupts – 16 for each processor which can interrupt itself, the other processor or both processors.
- Shared Peripheral Interrupts – 60 in total, these can come from the I/O Peripherals or to and from the Programmable Logic side of the device, these are shared between the two CPU's
- Private Peripheral Interrupts – 5 Interrupts which are private to each CPU e.g. CPU Timer CPU watch dog timer and dedicated PL to CPU interrupt

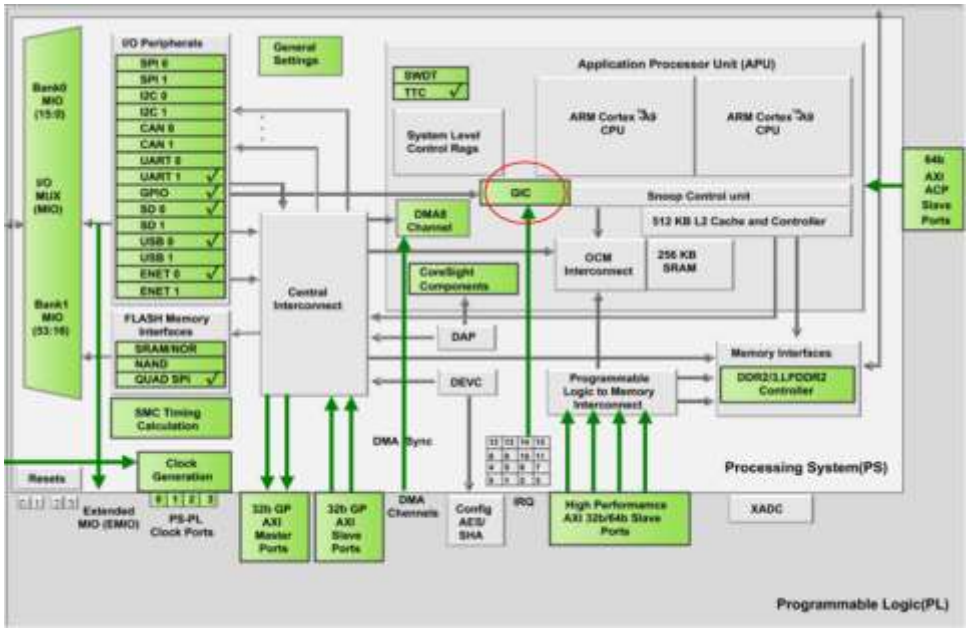


Figure 54 The Generic Interrupt Controller Circled in red

The shared peripheral interrupts are very interesting as they are very flexible, these can be routed to either CPU from the IO Peripherals (44 interrupts in total) or from the FPGA logic (16 interrupts in total) however, it is also possible to route interrupts from the IO periphery to the programme logic section of the device.

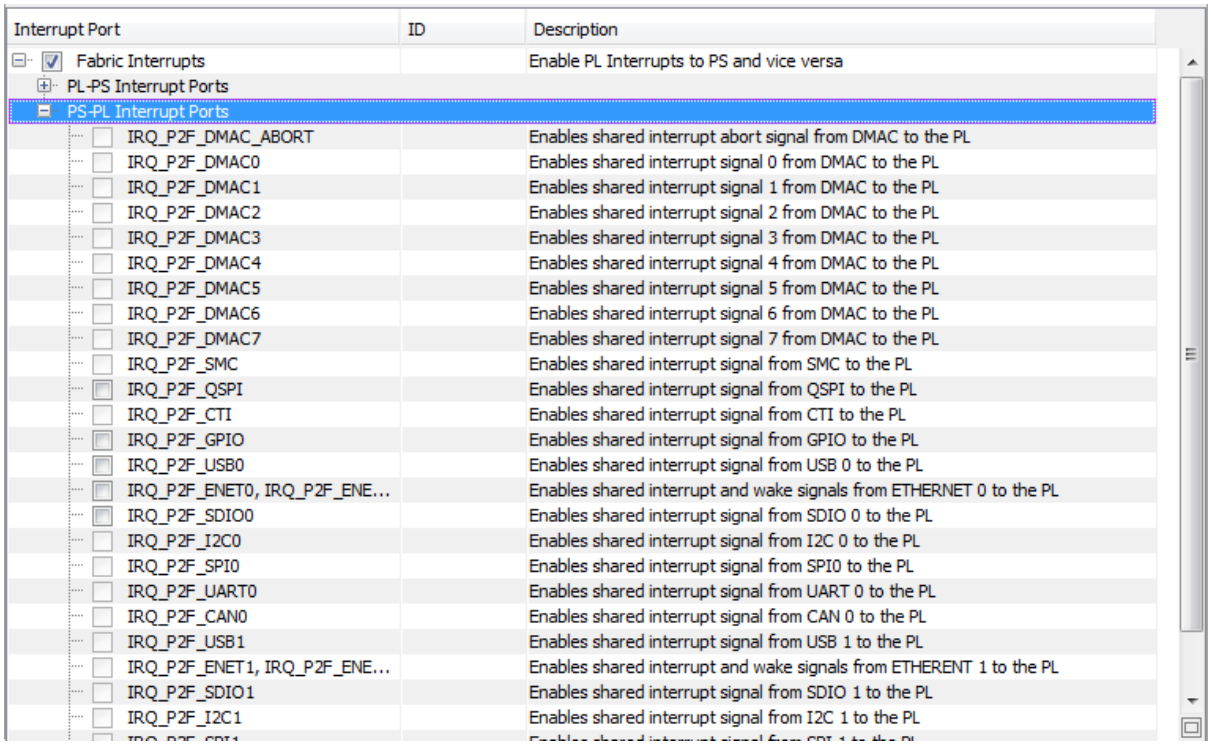


Figure 55 Interrupts from the PS IOP to the PL

Before I explain about how we set up interrupts on the Zynq, I think it is a good idea to recap how a interrupt is handled by a processor, when a interrupt occurs the following steps will occur

1. Interrupt is shown as pending
2. The processor stops executing the current thread
3. The processors saves the state of the thread in the stack to allow processing to continue once the interrupt has been handled
4. The processor executes the interrupt service routine which defines how the interrupt is to be handled
5. The processor resumes operation of the interrupted thread after restoring from the stack.

As interrupts are asynchronous events it is possible that multiple interrupts will occur at the same time, to address this interrupts are prioritised such that the highest interrupt pending can be services first.

In the next blog we will look at how we initialise and use the interrupt controller on the Zynq.

Using Interrupts

Having looked at interrupts and the different types available on the Zynq, this blog will focus upon the using a Shared Peripheral Interrupt GPIO interrupt to interrupt the current process which is reading from the XADC as before and toggle the state of the FPGA.

To implement this interrupt structure correctly we will need to write two functions

- Interrupt service routine – defines the actions that occur when the interrupt occurs
- Interrupt Set Up – Configures the interrupts, a routine allows for reuse to set up different interrupts this will be generic for all interrupts within the system and set up and enable the interrupts for the GPIO

Thankfully the standalone board support package contains a number of functions which greatly ease this task, these are provided within the following header files.

- Xparameters.h – defined processor definition device IDS
- XGPIOS.h – drivers for the configuration and use of the GPIO
- Xscugic.h – drivers for the configuration and use of the GIC
- Xil_exception.h – contains exception functions for the Cortex A9

As before when we communicated and used the XADC we need to know the hardware ID parameters for the devices we wish to use i.e. the GIC and the GPIO these are provided mostly within the BSP header file xparameters.h however the interrupt ID is provided from xparameters_ps.h (there is no need to declare this header file within your source code as it is included by the xparameters.h file)

```
#define GPIO_DEVICE_ID          XPAR_XGPIOPS_0_DEVICE_ID
#define INTC_DEVICE_ID          XPAR_SCUGIC_SINGLE_DEVICE_ID
#define GPIO_INTERRUPT_ID       XPS_GPIO_INT_ID
```

To set up the interrupt we will need two static global variables and the interrupt ID defined above

```
static XScuGic Intc; // Interrupt Controller Driver
static XGpioPs Gpio; //GPIO Device
```

Within the set up interrupt set up, function we will initialise the exceptions, configure and initialise the GIC, connect the GIC to the interrupt handling hardware. The functions needed to do this are provided within Xil_exception.h and Xscugic.h files.

When it comes to configuring the GPIO to function as an interrupt we can configure either a bank or an individual pin, this can be achieved using functions provided within xgpiops.h for example

```
void XGpioPs_IntrEnable(XGpioPs *InstancePtr, u8 Bank, u32 Mask);

void XGpioPs_IntrEnablePin(XGpioPs *InstancePtr, int Pin);
```

You will also need to configure the interrupt correctly for instance do you wish it to be edge triggered or level triggered and if so which edge and level this can be achieved using the function

```
void XGpioPs_SetIntrTypePin(XGpioPs *InstancePtr, int Pin, u8 IrqType);
```

Where the IrqType is defined by one of the five definitions within the xgpiops.h file

```
#define XGPIOPS_IRQ_TYPE_EDGE_RISING 0 /**< Interrupt on Rising edge */
#define XGPIOPS_IRQ_TYPE_EDGE_FALLING 1 /**< Interrupt Falling edge */
#define XGPIOPS_IRQ_TYPE_EDGE_BOTH 2 /**< Interrupt on both edges */
#define XGPIOPS_IRQ_TYPE_LEVEL_HIGH 3 /**< Interrupt on high level */
#define XGPIOPS_IRQ_TYPE_LEVEL_LOW 4 /**< Interrupt on low level */
```

If you decide to use the bank enable you need to know which bank the pin or pins you wish to enable interrupts are on, the Zynq supports a maximum of 118 GPIO. In this configuration all of the MIO (54 pins) are being used as GPIO along with the EMIO (64 pins), this would be broken into four banks with each bank containing 32 pins.

This set up function will also defines the interrupt service routine which is to be called when the interrupt occurs this uses the function

```
XGpioPs_SetCallbackHandler(Gpio, (void *)Gpio, IntrHandler);
```

Having written the interrupt set up the next stage is to write the interrupt service routine which will be called when an interrupt occurs. This can be as simple of as complicated as the application defines for this example it will perform the same function as that in the previous polled example in that it will toggle the status of the led on and off each time the button is pressed. The interrupt service routine will also print out to the console a message when it is run.

Interrupt Source Code

```
#include <stdio.h>
#include "platform.h"
#include "xadcps.h"
#include "xgpiops.h"
#include "xil_types.h"
#include "Xscugic.h"
#include "Xil_exception.h"

#define XPAR_AXI_XADC_0_DEVICE_ID 0

#define GPIO_DEVICE_ID                  XPAR_XGPIOPS_0_DEVICE_ID
#define INTC_DEVICE_ID                  XPAR_SCUGIC_SINGLE_DEVICE_ID
#define GPIO_INTERRUPT_ID               XPS_GPIO_INT_ID

#define ledpin 47
#define pbsw 51
#define LED_DELAY     100000000
//void print(char *str);

static XAdcPs  XADCMonInst;
static XScuGic Intc; /* The Instance of the Interrupt Controller Driver */
static XGpioPs Gpio; /* The driver instance for GPIO Device. */
int toggle;//used to toggle the LED
static void SetupInterruptSystem(XScuGic *GicInstancePtr, XGpioPs *Gpio, u16 GpioIntrId);
static void IntrHandler(void *CallBackRef, int Bank, u32 Status);

int main()
{

        XAdcPs_Config *ConfigPtr;
        XAdcPs *XADCInstPtr = &XADCMonInst;

        int Status;
//      u32 sw;

        XGpioPs_Config *GPIOConfigPtr;

        //status of initialisation
        int Status_ADC;

        //temperature readings
        u32 TempRawData;
        float TempData;

        //Vcc Int readings
```

```
	u32 VccIntRawData;
	float VccIntData;

	//Vcc Aux readings
	u32 VccAuxRawData;
	float VccAuxData;

	//Vbram readings
	u32 VBramRawData;
	float VBramData;

	//VccPInt readings
	u32 VccPIntRawData;
	float VccPIntData;

	//VccPAux readings
	u32 VccPAuxRawData;
	float VccPAuxData;

	//Vddr readings
	u32 VDDRRawData;
	float VDDRData;

	init_platform();

printf("Adam Edition MicroZed Using Vivado How To Printf \n\r");

//GPIO Initilization
GPIOConfigPtr = XGpioPs_LookupConfig(XPAR_XGPIOPS_0_DEVICE_ID);
Status = XGpioPs_CfgInitialize(&Gpio, GPIOConfigPtr,GPIOConfigPtr->BaseAddr);
	if (Status != XST_SUCCESS) {
		 print("GPIO INIT FAILED\n\r");
		return XST_FAILURE;
	}

//set direction and enable output
XGpioPs_SetDirectionPin(&Gpio, ledpin, 1);
XGpioPs_SetOutputEnablePin(&Gpio, ledpin, 1);

//set direction input pin
XGpioPs_SetDirectionPin(&Gpio, pbsw, 0x0);

//XADC initilization

 ConfigPtr = XAdcPs_LookupConfig(XPAR_AXI_XADC_0_DEVICE_ID);
	if (ConfigPtr == NULL) {
		return XST_FAILURE;
	}
```

```
    Status_ADC = XAdcPs_CfgInitialize(XADCInstPtr,ConfigPtr,ConfigPtr->BaseAddress);
    if(XST_SUCCESS != Status_ADC){
       print("ADC INIT FAILED\n\r");
       return XST_FAILURE;
     }

    //self test
    Status_ADC = XAdcPs_SelfTest(XADCInstPtr);
          if (Status_ADC != XST_SUCCESS) {
                   return XST_FAILURE;
          }

          //stop sequencer
          XAdcPs_SetSequencerMode(XADCInstPtr,XADCPS_SEQ_MODE_SINGCHAN);

    //disable alarms
    XAdcPs_SetAlarmEnables(XADCInstPtr, 0x0);

    //configure sequencer to just sample internal on chip parameters
    XAdcPs_SetSeqInputMode(XADCInstPtr, XADCPS_SEQ_MODE_SAFE);

    //configure the channel enables we want to monitor

XAdcPs_SetSeqChEnables(XADCInstPtr,XADCPS_CH_TEMP|XADCPS_CH_VCCINT|XADCPS_CH_VCCA
UX|XADCPS_CH_VBRAM|XADCPS_CH_VCCPINT|
                   XADCPS_CH_VCCPAUX|XADCPS_CH_VCCPDRO);

   SetupInterruptSystem(&Intc, &Gpio, GPIO_INTERRUPT_ID);

    while(1){

     TempRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_TEMP);
     TempData = XAdcPs_RawToTemperature(TempRawData);
     printf("Raw Temp %lu Real Temp %f \n\r", TempRawData, TempData);

     VccIntRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VCCINT);
     VccIntData = XAdcPs_RawToVoltage(VccIntRawData);
     printf("Raw VccInt %lu Real VccInt %f \n\r", VccIntRawData, VccIntData);

     VccAuxRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VCCAUX);
     VccAuxData = XAdcPs_RawToVoltage(VccAuxRawData);
     printf("Raw VccAux %lu Real VccAux %f \n\r", VccAuxRawData, VccAuxData);

     VBramRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VBRAM);
     VBramData = XAdcPs_RawToVoltage(VBramRawData);
     printf("Raw VccBram %lu Real VccBram %f \n\r", VBramRawData, VBramData);
```

```
    VccPIntRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VCCPINT);
    VccPIntData = XAdcPs_RawToVoltage(VccPIntRawData);
    printf("Raw VccPInt %lu Real VccPInt %f \n\r", VccPIntRawData, VccPIntData);

    VccPAuxRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VCCPAUX);
    VccPAuxData = XAdcPs_RawToVoltage(VccPAuxRawData);
    printf("Raw VccPAux %lu Real VccPAux %f \n\r", VccPAuxRawData, VccPAuxData);

    VDDRRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_VCCPDRO);
    VDDRData = XAdcPs_RawToVoltage(VDDRRawData);
    printf("Raw VccDDR %lu Real VccDDR %f \n\r", VDDRRawData, VDDRData);

   }

  return 0;
}

static void SetupInterruptSystem(XScuGic *GicInstancePtr, XGpioPs *Gpio, u16 GpioIntrId)
{


                XScuGic_Config *IntcConfig; /* Instance of the interrupt controller */

                Xil_ExceptionInit();

                /*
                 * Initialize the interrupt controller driver so that it is ready to
                 * use.
                 */
                IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);

                XScuGic_CfgInitialize(GicInstancePtr, IntcConfig,
                                      IntcConfig->CpuBaseAddress);

                /*
                 * Connect the interrupt controller interrupt handler to the hardware
                 * interrupt handling logic in the processor.
                 */
                Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                                (Xil_ExceptionHandler)XScuGic_InterruptHandler,
                                GicInstancePtr);

                /*
                 * Connect the device driver handler that will be called when an
                 * interrupt for the device occurs, the handler defined above performs
                 * the specific interrupt processing for the device.
                 */
                XScuGic_Connect(GicInstancePtr, GpioIntrId,
```

```
                                        (Xil_ExceptionHandler)XGpioPs_IntrHandler,
                                        (void *)Gpio);

                //Enable  interrupts for all the pins in bank 0.
                XGpioPs_SetIntrTypePin(Gpio, pbsw, XGPIOPS_IRQ_TYPE_EDGE_RISING);

                //Set the handler for gpio interrupts.
                XGpioPs_SetCallbackHandler(Gpio, (void *)Gpio, IntrHandler);

                //Enable the GPIO interrupts of Bank 0.
                XGpioPs_IntrEnablePin(Gpio, pbsw);

                //Enable the interrupt for the GPIO device.
                XScuGic_Enable(GicInstancePtr, GpioIntrId);

                // Enable interrupts in the Processor.
                Xil_ExceptionEnableMask(XIL_EXCEPTION_IRQ);
        }

static void IntrHandler(void *CallBackRef, int Bank, u32 Status)
{
        int delay;
        printf("****button pressed****\n\r");
        toggle = !toggle;
        XGpioPs_WritePin(&Gpio, ledpin, toggle);
        for( delay = 0; delay < LED_DELAY; delay++)//wait
        {}
}
```

XADC Interrupts and Alarms

Due to a few questions in the week on the XADC and the interrupts and alarms I thought it would be a good idea to quickly look at these and see how we can use them within a system.

The ability of the Zynq to monitor its voltage supplies and temperature is really interesting as it can be used during system commissioning to verify supply voltages and operating temperature remain within the targeted operating ranges throughout the testing.

This can also be used in the system when it is operating as a kind of prognostics to determine if the module the Zynq is within has an error which will result in failure e.g. slowly drifting power supply.

To enable this the Zynq has a number of maskable interrupts which can interrupt the processors should a parameter go outside of the user defined maximum or minimum parameter. In fact the XADC has a dedicated interrupt shared by both processors on IRQ 39 as the extract from the Zynq TRM shows.

Table 7-3: **PS and PL Shared Peripheral Interrupts (SPI)**

Source	Interrupt Name	IRQ ID#	Status Bits (mpcore Registers)	Required Type	PS-PL Signal Name	I/O
APU	CPU 1, 0 (L2, TLB, BTAC)	33:32	spi_status_0[1:0]	Rising edge	~	~
	L2 Cache	34	spi_status_0[2]	High level	~	~
	OCM	35	spi_status_0[3]	High level	~	~
Reserved	~	36	spi_status_0[3]	~	~	~
PMU	PMU [1,0]	38, 37	spi_status_0[6:5]	High level	~	~
XADC	XADC	39	spi_status_0[7]	High level	~	~
DVI	DVI	40	spi_status_0[8]	High level	~	~
SWDT	SWDT	41	spi_status_0[9]	Rising edge	~	~
Timer	TTC 0	43:42	spi_status_0[11:10]	High level	~	~
Reserved	~	44	spi_status_0[12]	~	~	~
DMAC	DMAC Abort	45	spi_status_0[13]	High level	IRQP2F[28]	Output
	DMAC [3:0]	49:46	spi_status_0[17:14]	High level	IRQP2F[23:20]	Output
Memory	SMC	50	spi_status_0[18]	High level	IRQP2F[19]	Output
	Quad SPI	51	spi_status_0[19]	High level	IRQP2F[18]	Output
Debug	CTI	~	~	High level	IRQP2F[17]	Output

Figure 56 PS / Pl shared Interrupts

These parameters can be set during the configuration of the XADC within Vivado as show below

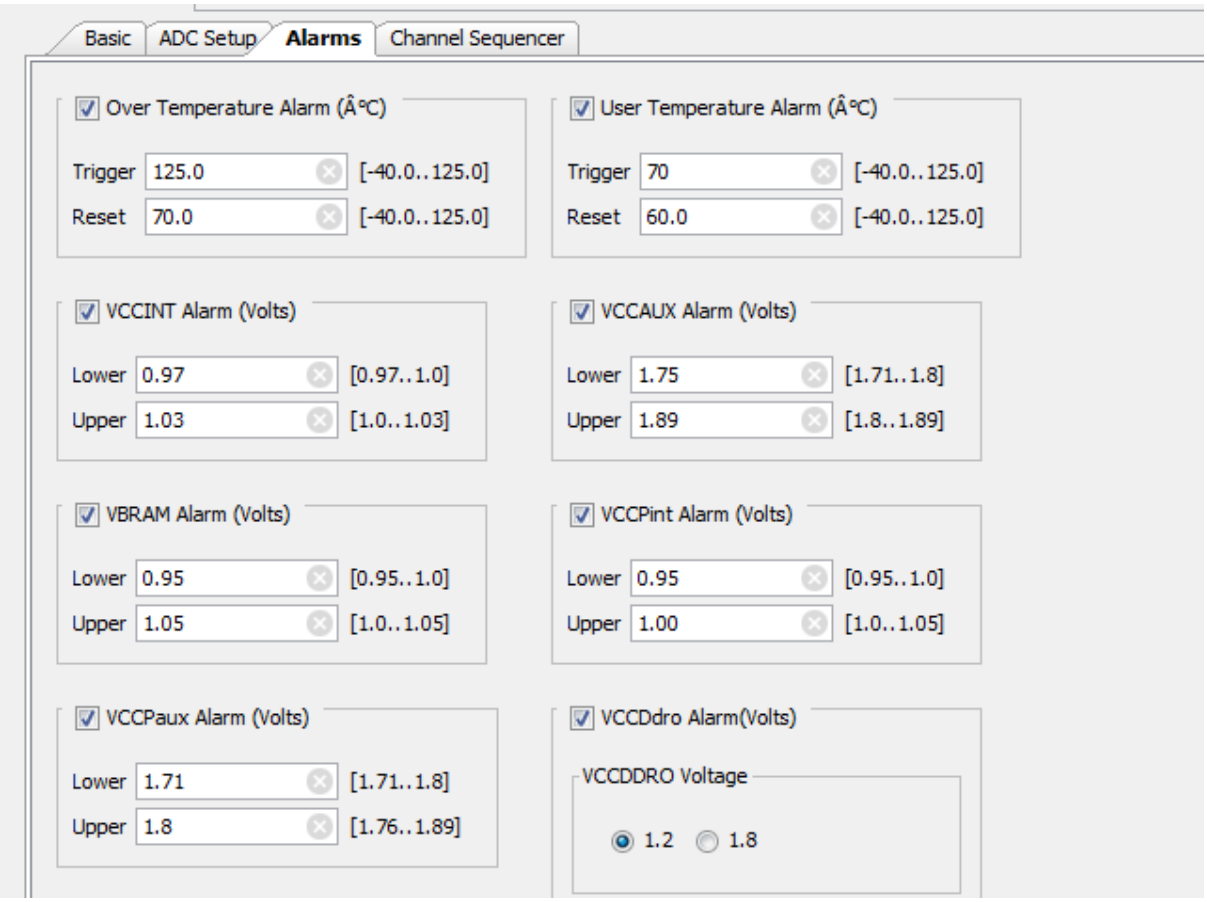


Figure 57 XADC Setting the Alarms

They can also be during execution of the software using the xadc_ps.h functions provided for instance

```
XAdcPs_SetAlarmThreshold(XADCInstPtr, XADCPS_ATR_TEMP_LOWER,(TempRawData));
```

The XADC also provides a number of alarm output signals these can be connected to other logic modules which may need to take action should an alarm occur or to external indicators for instance LEDs which could be connected on a front panel and show the status of the equipment. This is especially useful for providing the user visual warnings of temperature should the system fan fail etc.

These alarms (there are seven in total, see table from TRM below) can be enabled by using the function call with the appropriate mask, as supplied by xadcps_hw.h if more than one required then OR them together.

```
XAdcPs_SetAlarmEnables(XADCInstPtr, XADCPS_CFR1_ALM_TEMP_MASK);
```

Table 30-2: **XADC Alarm Signals**

Alarm	Description
ALM[0]	XADC temperature sensor alarm
ALM[1]	XADC V_{CCINT} sensor alarm
ALM[2]	XADC V_{CCAUX} sensor alarm
ALM[3]	XADC V_{CCBRAM} sensor alarm
ALM[4]	XADC V_{CCPINT} sensor alarm (Processor V_{CCINT})
ALM[5]	XADC V_{CCPAUX} sensor alarm (Processor V_{CCPAUX}
ALM[6]	XADC V_{CCDDRO} sensor alarm (Processor DDR controller voltage)

Figure 58 Alarm Indications

Having enabled the alarm if you want a interrupt to occur as well we can use the call, again using the masks provided by xadcps_hw.h if more than one is desired to be enabled then OR them together.

```
XAdcPs_IntrEnable(XADCInstPtr,XADCPS_INTX_ALM0_MASK);
```

With this in mind I set about creating a simple project in Vivado which consisted of the PS connected to the XADC via the AXI interface, for this example I would not be using any external analogue inputs but instead the temperature of the device.

The software would configure the XADC to issue an interrupt to the processor should the temperature go above or below the initial reading on power up plus a few degrees. Now in reality we would not want such a tight tolerance of operating temperature however for a demonstration of the interrupt it is a great application as we can use the self-heating during operation of the Zynq to trigger the interrupt.

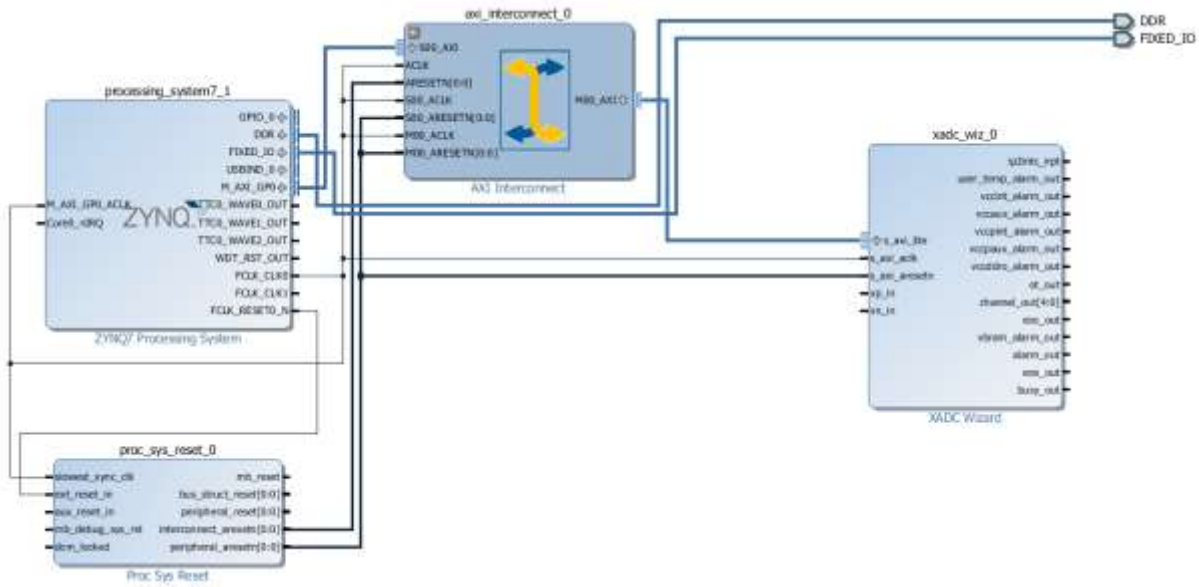


Figure 59 Configuration of the Example Block Diagram

I have attached the code and however it is split into three functions

1. The first configures the generic interrupt controller on the Zynq such that we can use the XADC interrupt as part of this the interrupt service routine to be called when the XADC interrupt is issued is defined.

2. The second configures the XADC, sets the sequencer to a safe mode and disables all alarms before reading the temperature, and then assigning an upper and lower temperature alarm based upon the value it has just read. Having set these values the function then sets the temperature alarm and enables the temperature interrupt.

3. The interrupt service routine which will be called when the temperature interrupt occurs this will clear the interrupt status register, disable more interrupts from occurring and read the temperature at which it occurred. This temperature might not be above the interrupt temperature as it will fluctuate and we do not have the averaging turned on it should however be near.

When I built the example code and generated the boot image this I achieved the results below on the MicroZed board after it had been running for a few minutes.

```
Xcell Daily blog XADC part 43

temp high alarm 41491 Real Temp 45.919220

temp low alarm 39431 Real Temp 30.077667

Temperature Interrupt 513

Raw Temp 41288 Real Temp 44.358154
```

Figure 60 Results of the Example

As you can see the interrupt triggered the 513 after is the status (in decimal) of the XADC interrupt status register.

The XADC can be a very powerful tool in both the system and FPGA designers tools box, this simple example has shown how we could use it for prognostics, or in a more critical application as part of a anti tamper policy.

XADC Alarms and Interrupts Code

```
#include <stdio.h>
#include "platform.h"
#include "xadcps.h"
#include "Xscugic.h"
#include "Xil_exception.h"


//XADC info
#define XPAR_AXI_XADC_0_DEVICE_ID 0
//Interrupt Controller
#define INTC_DEVICE_ID        XPAR_SCUGIC_SINGLE_DEVICE_ID
#define INTR_ID                     XPAR_XADCPS_INT_ID

static XAdcPs  XADCInst; //XADC
static XScuGic InterruptController;

static int SetupInterruptSystem(XScuGic *IntcInstancePtr,XAdcPs
*XAdcPtr,u16 IntrId );
static int XAdcInterruptHandler(void *CallBackRef);
static void adc_config(XAdcPs *XADCInstPtr, u16 XAdcDeviceId);

int main()
{

      //static XAdcPs *XADCInstPtr; //xadc pointer

      //temperature readings
      //u32 TempRawData;
      //float TempData;


      int Status;

      init_platform();
      printf("Xcell Daily blog XADC part 43 \n\r");



      Status = SetupInterruptSystem(&InterruptController, &XADCInst,
INTR_ID);
      if (Status != XST_SUCCESS) {
            return XST_FAILURE;
      }


      adc_config(&XADCInst,XPAR_AXI_XADC_0_DEVICE_ID );



    return 0;
}


static void adc_config(XAdcPs *XADCInstPtr, u16 XAdcDeviceId)
{

      u32 IntrStatus;
      XAdcPs_Config *ConfigPtr;           //xadc config
      u32 TempRawData;
```

```
	float TempData;

	//XADC initilization
	ConfigPtr = XAdcPs_LookupConfig(XPAR_AXI_XADC_0_DEVICE_ID);
	//adc set up
	XAdcPs_CfgInitialize(XADCInstPtr,ConfigPtr,ConfigPtr->BaseAddress);
	//stop sequencer
	XAdcPs_SetSequencerMode(XADCInstPtr,XADCPS_SEQ_MODE_SINGCHAN);
	//disable alarms
	XAdcPs_SetAlarmEnables(XADCInstPtr, 0x0);

	TempRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_TEMP);
	TempRawData = TempRawData+0x03FF;
	XAdcPs_SetAlarmThreshold(XADCInstPtr,
XADCPS_ATR_TEMP_UPPER,(TempRawData));
	TempData = XAdcPs_RawToTemperature(TempRawData);
	printf("temp high alarm %lu Real Temp %f \n\r", TempRawData,
TempData);

	TempRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_TEMP);
	TempRawData = TempRawData-0x03FF;
	XAdcPs_SetAlarmThreshold(XADCInstPtr,
XADCPS_ATR_TEMP_LOWER,(TempRawData));
	TempData = XAdcPs_RawToTemperature(TempRawData);
	printf("temp low alarm %lu Real Temp %f \n\r", TempRawData,
TempData);

//	printf("1\n\r");
	IntrStatus = XAdcPs_IntrGetStatus(XADCInstPtr);
//	printf("2 a %lu\n\r",IntrStatus);
	XAdcPs_IntrClear(XADCInstPtr, IntrStatus);
//	printf("2 b \n\r");
	IntrStatus = XAdcPs_IntrGetStatus(XADCInstPtr);
//	printf("2 c %lu\n\r",IntrStatus);
	XAdcPs_SetAlarmEnables(XADCInstPtr, XADCPS_CFR1_ALM_TEMP_MASK);
//	printf("3\n\r");
	XAdcPs_IntrEnable(XADCInstPtr,XADCPS_INTX_ALM0_MASK);//temp
//	printf("4\n\r");

	//configure sequencer to just sample internal on chip parameters
	XAdcPs_SetSeqInputMode(XADCInstPtr, XADCPS_SEQ_MODE_SAFE);
	//configure the channel enables we want to monitor
	XAdcPs_SetSeqChEnables(XADCInstPtr,XADCPS_CH_TEMP|XADCPS_CH_VCCINT|XA
DCPS_CH_VCCAUX|XADCPS_CH_VBRAM|XADCPS_CH_VCCPINT|
			XADCPS_CH_VCCPAUX|XADCPS_CH_VCCPDRO);

	while(1){

	TempRawData = XAdcPs_GetAdcData(XADCInstPtr, XADCPS_CH_TEMP);
	TempData = XAdcPs_RawToTemperature(TempRawData);
    //  printf("Raw Temp %lu Real Temp %f \n\r", TempRawData, TempData);

     }
}

static int SetupInterruptSystem(XScuGic *IntcInstancePtr, XAdcPs
*XAdcPtr,u16 IntrId )
{
	XScuGic_Config *IntcConfig; /* Instance of the interrupt controller
*/
	int Status;
```

```
	IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
	if (NULL == IntcConfig) {
		return XST_FAILURE;
	}

	Status = XScuGic_CfgInitialize(IntcInstancePtr, IntcConfig,
IntcConfig->CpuBaseAddress);
	if (Status != XST_SUCCESS) {
		return XST_FAILURE;
	}

	Status = XScuGic_Connect(IntcInstancePtr,
IntrId,(Xil_InterruptHandler)XAdcInterruptHandler,(void *)XAdcPtr);
	if (Status != XST_SUCCESS) {
		return XST_FAILURE;
	}

	XScuGic_Enable(IntcInstancePtr, IntrId);

	Xil_ExceptionInit();

	Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
			(Xil_ExceptionHandler) XScuGic_InterruptHandler,
			IntcInstancePtr);

	Xil_ExceptionEnable();
	return XST_SUCCESS;
}

static int XAdcInterruptHandler(void *CallBackRef)
{
	u32 IntrStatusValue;
	u32 TempRawData;
	float TempData;
	XAdcPs *XAdcPtr = (XAdcPs *)CallBackRef;

	IntrStatusValue = XAdcPs_IntrGetStatus(XAdcPtr);

	if (IntrStatusValue & XADCPS_INTX_ALM0_MASK) {
		/*
		 * Set Temperature interrupt flag so the code
		 * in application context can be aware of this interrupt.
		 */

		/* Disable Temperature interrupt */
		XAdcPs_IntrDisable(XAdcPtr,XADCPS_INTX_ALM0_MASK);
		printf("Temperature Interrupt %lu \n\r",IntrStatusValue);
		TempRawData = XAdcPs_GetAdcData(XAdcPtr, XADCPS_CH_TEMP);
		TempData = XAdcPs_RawToTemperature(TempRawData);
		printf("Raw Temp %lu Real Temp %f \n\r", TempRawData,
TempData);

	}

	/*
	 * Clear all bits in Interrupt Status Register.
	 */
	XAdcPs_IntrClear(XAdcPtr, IntrStatusValue);

 }
```

Timers, Clocks and Watchdogs

The Zynq has a number of timers and watchdogs available these are either private to a CPU or a shared resource available to both CPU's

- CPU 32 bit timer (SCUTIMER) clocked at half the CPU frequency
- CPU 32 bit watch dog (SCUWDT) clocked at half the CPU frequency
- Shared 64 bit Global Timer (GT) clocked at half the CU frequency, each CPU has its own 64 bit comparator which drives a private interrupt for each CPU
- System Watchdog Timer (WDT) which can be clocked from the CPU clock or an external source.
- Two Triple Timer Counter (TTC) each TTC contains three independent timers, these can be clocked by the CPU clock, or external source from the MIO or EMIO (PL).

The private timers, watchdog and global timer do not require any changes to the Vivado project to use, however to use the TTC or the system watchdog these must be enabled within the processing system definition within Vivado

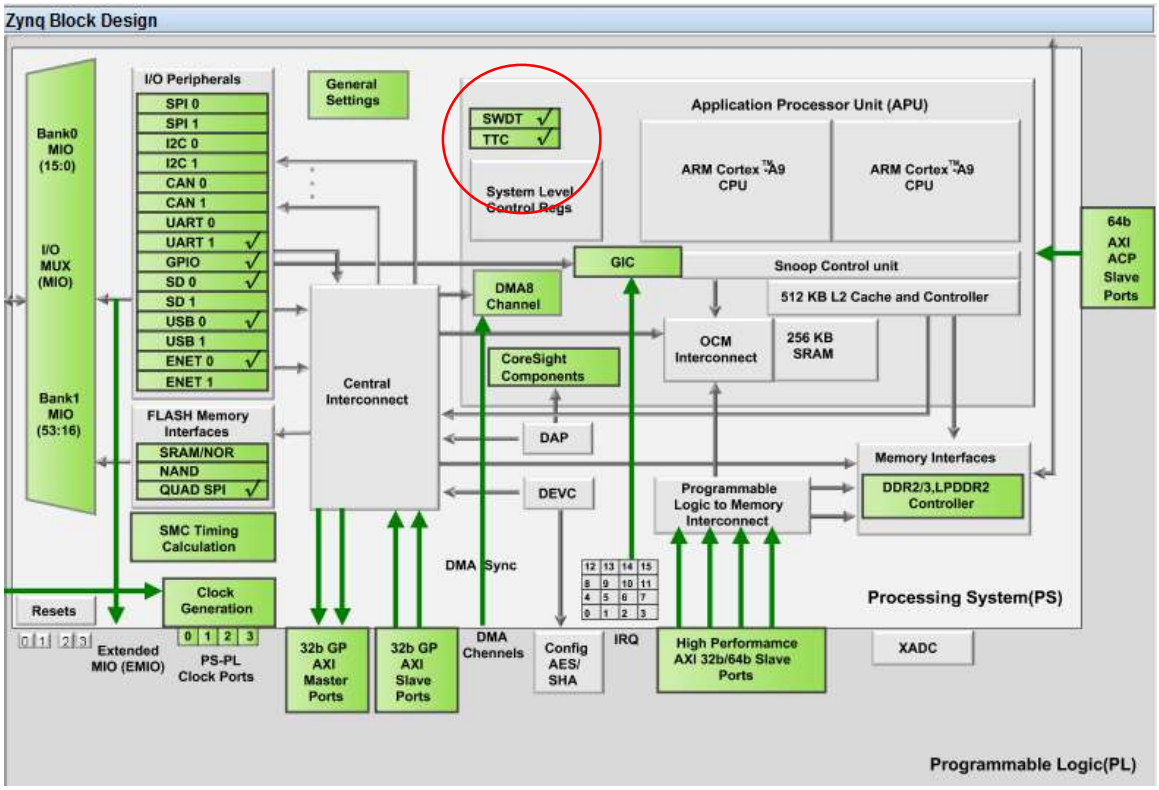


Figure 61 Location of the system watchdog and Triple Timer Counters

Once these have been enabled within the Zynq block design the drive clock can be selected via the clock configuration page.

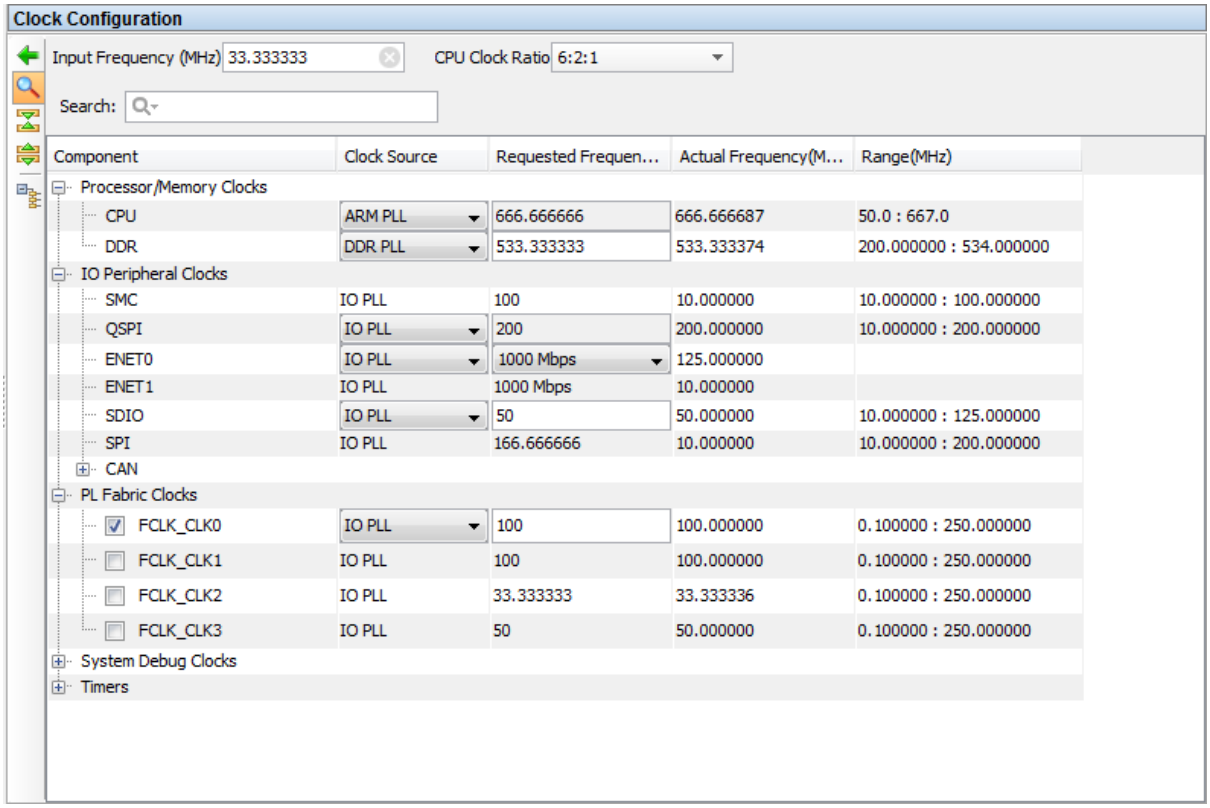


Figure 62 Setting the clocks on the Zynq

In the clock configuration above you can see that both the WDT and TTC0 are enabled and configured to use the CPU clock, the drop down list allows the selection of an external source this can come from either the MIO or the EMIO.

Once these have been configured as you desire the system can be re implemented and exported to SDK which we will look at in the next blog. Where we can explore how we drive and use the timers and watchdogs.

Before we complete this blog it is however, I think important to look at the clocking architecture of the Zynq.

The PS clock provided by the hardware drives three Phase Locked Loops which are used to multiply the frequency up. Each of these PLLs has a different function within the PS

- ARM PLL - Usually used to clock the CPU, SCU, OCM and AXI GP Interconnect
- IO PLL – Used to clock the IO Peripherals
- DDR PLL – used to clock the DDR memory and the AXI HP Interconnect

The outputs from these PLLs are then used in combination with programmable dividers and clock ratio dividers to generate the clocks used within the PS.

The CPU clock domain as generated by the ARM PLL and used by the timers has four possible clocks CPU_6x4x, CPU_3x2x, CPU_2x and CPU_1x. There are two clocking schemes for the Zynq which define the division factors and hence output frequencies of these clocks 6:3:2:1 and 4:2:2:1. This equates to the multiplication factor based from CPU_1x for example using 6:3:2:1 scheme the CPU is clocked by the CPU_6x4x which is 6 times CPU_1x. Hence the private timers and watchdog are clocked at half the CPU frequency from CPU_3x2x.

While the system watchdog timer and TTC if clocked from the CPU_1x or an external source.

These settings for the clocking of the PS is exported to SDK (ps7_init.c and ps7_init.h files within the SDK hardware definition) and used by the first stage boot loader to correctly configure the processor.

Implementing the Private Timer

In the last blog we looked at the timers available to the Zynq, in this one we will start looking at how we can use these resources starting with the private timer.

Alike most of the Zynq peripherals this comes with a number of predefined functions and macros to help the engineer use the resource efficiently. These are contained within

```
#include "xscutimer.h"
```

This contains functions (macros) which will provide the following capabilities

- Initialising the timer
- Self-test the timer
- Start and stopping the timer
- Manage the timer – restart, check if expired, load the timer, enable/disable auto loading
- Setting the pre-scaler
- Getting the pre-scaler
- Setup, enable, disable, clear and manage timer interrupts

The timer itself is controlled via four registers

- Private Timer Load Register – used in auto reload mode, this contains the value which is reloaded into the Private Timer Counter Register when auto reload is enabled

- Private Timer Counter Register – This is the actual counter itself, when it reaches zero when enabled the interrupt event flag is set

- Private Timer Control Register – The control register enables or disables the timer, auto reload mode and interrupt generation, it also contains the pre-scaler for the timer.

- Private Timer Interrupt Status Register – Contains the Private timer interrupt status event flag

As for using the GPIO and the XADC the timer device ID and timer interrupt ID which are needed to set up the timer are contained within the XParameters.h. The example for this blog will use the push button interrupt which we have developed previously, when the button is pressed the timer will be loaded and start to run (not in auto reload mode). Upon expiration of the timer a interrupt will be generated which will write a message out over the STDOUT, the interrupt will then be cleared to wait until the next time the button is pressed. This example will always load the same value into the counter hence with the declarations at the top of the file the timer count value is declared

```
#define TIMER_LOAD_VALUE      0xFFFFFFFF
```

The next stage is to configure and initialise the private timer , perform a self-test and load the timer count value into the timer.

```
//timer initialisation
TMRConfigPtr = XScuTimer_LookupConfig(TIMER_DEVICE_ID);
XScuTimer_CfgInitialize(&Timer, TMRConfigPtr,TMRConfigPtr->BaseAddr);
XScuTimer_SelfTest(&Timer);
//load the timer
```

```
XScuTimer_LoadTimer(&Timer, TIMER_LOAD_VALUE);
```

We also need to update the interrupt setup sub routine to connect the timer interrupts to the GIC and enable the timer interrupt

```
//set up the timer interrupt
XScuGic_Connect(GicInstancePtr, TimerIntrId,
                                (Xil_ExceptionHandler)TimerIntrHandler,
                                (void *)TimerInstancePtr);

//enable the interrupt for the Timer at GIC
XScuGic_Enable(GicInstancePtr, TimerIntrId);
//enable interrupt on the timer
XScuTimer_EnableInterrupt(TimerInstancePtr);
```

Where TimerIntrHandler is the name of the function which is called when the interrupt occurs, next the timer interrupt must be enabled on the GIC and within the timer itself.

The timer interrupt service routine is very simple it just clears the pending interrupt and writes out a message over the STDOUT

```
static void TimerIntrHandler(void *CallBackRef)
{

      XScuTimer *TimerInstancePtr = (XScuTimer *) CallBackRef;
      XScuTimer_ClearInterruptStatus(TimerInstancePtr);
      printf("****Timer Event!!!!!!!!!!!!!****\n\r");

}
```

With this complete the final thing to do is modify the GPIO interrupt service routine to start the timer each time the button is pushed

```
//load timer
XScuTimer_LoadTimer(&Timer, TIMER_LOAD_VALUE);
//start timer
XScuTimer_Start(&Timer);
```

To do this we first load the timer value into the timer and then call the timer start function, before again clearing the push button interrupt and resume processing.

To reduce the number of messages being output over STDOUT the reading of XADC within the main loop of the programme has been removed, you can see the example code here.

Figure 63 Results of the timer demonstration

Timer Example Source Code

```
#include <stdio.h>
#include "platform.h"
#include "xadcps.h"
#include "xgpiops.h"
#include "xil_types.h"
#include "Xscugic.h"
#include "Xil_exception.h"
#include "xscutimer.h"
//XADC info
#define XPAR_AXI_XADC_0_DEVICE_ID 0

//GPIO info
#define GPIO_DEVICE_ID          XPAR_XGPIOPS_0_DEVICE_ID
#define INTC_DEVICE_ID          XPAR_SCUGIC_SINGLE_DEVICE_ID
#define GPIO_INTERRUPT_ID       XPS_GPIO_INT_ID

//timer info
#define TIMER_DEVICE_ID         XPAR_XSCUTIMER_0_DEVICE_ID
#define INTC_DEVICE_ID          XPAR_SCUGIC_SINGLE_DEVICE_ID
#define TIMER_IRPT_INTR         XPAR_SCUTIMER_INTR

#define ledpin 47
#define pbsw 51
#define LED_DELAY     100000000
#define TIMER_LOAD_VALUE      0xFFFFFFFF
//void print(char *str);

static XAdcPs  XADCMonInst; //XADC
static XScuGic Intc; //GIC
static XGpioPs Gpio; //GPIO
static XScuTimer Timer;//timer

static int toggle = 0;//used to toggle the LED

static void SetupInterruptSystem(XScuGic *GicInstancePtr, XGpioPs *Gpio,
u16 GpioIntrId,
            XScuTimer *TimerInstancePtr, u16 TimerIntrId);

static void GPIOIntrHandler(void *CallBackRef, int Bank, u32 Status);
static void TimerIntrHandler(void *CallBackRef);

int main()
{

       XAdcPs_Config *ConfigPtr;           //xadc config
       XScuTimer_Config *TMRConfigPtr;     //timer config
       XGpioPs_Config *GPIOConfigPtr;      //gpio config
       XAdcPs *XADCInstPtr = &XADCMonInst; //xadc pointer


       init_platform();

     printf("\n\rAdam Edition MicroZed Using Vivado How To Printf \n\r");

     //GPIO Initilization
     GPIOConfigPtr = XGpioPs_LookupConfig(XPAR_XGPIOPS_0_DEVICE_ID);
     XGpioPs_CfgInitialize(&Gpio, GPIOConfigPtr,GPIOConfigPtr->BaseAddr);
     //set direction and enable output
```

```
     XGpioPs_SetDirectionPin(&Gpio, ledpin, 1);
     XGpioPs_SetOutputEnablePin(&Gpio, ledpin, 1);
    //set direction input pin
     XGpioPs_SetDirectionPin(&Gpio, pbsw, 0x0);

     //timer initialisation
     TMRConfigPtr = XScuTimer_LookupConfig(TIMER_DEVICE_ID);
     XScuTimer_CfgInitialize(&Timer, TMRConfigPtr,TMRConfigPtr->BaseAddr);
     XScuTimer_SelfTest(&Timer);
       //load the timer
       XScuTimer_LoadTimer(&Timer, TIMER_LOAD_VALUE);
       //XScuTimer_EnableAutoReload(&Timer);
       //start timer
       //XScuTimer_Start(&Timer);

       //XADC initilization
       ConfigPtr = XAdcPs_LookupConfig(XPAR_AXI_XADC_0_DEVICE_ID);
       //adc set up
     XAdcPs_CfgInitialize(XADCInstPtr,ConfigPtr,ConfigPtr->BaseAddress);
     //self test adc
     XAdcPs_SelfTest(XADCInstPtr);
       //stop sequencer
       XAdcPs_SetSequencerMode(XADCInstPtr,XADCPS_SEQ_MODE_SINGCHAN);
     //disable alarms
     XAdcPs_SetAlarmEnables(XADCInstPtr, 0x0);
     //configure sequencer to just sample internal on chip parameters
     XAdcPs_SetSeqInputMode(XADCInstPtr, XADCPS_SEQ_MODE_SAFE);
     //configure the channel enables we want to monitor

XAdcPs_SetSeqChEnables(XADCInstPtr,XADCPS_CH_TEMP|XADCPS_CH_VCCINT|XADCPS_C
H_VCCAUX|XADCPS_CH_VBRAM|XADCPS_CH_VCCPINT|
            XADCPS_CH_VCCPAUX|XADCPS_CH_VCCPDRO);

     //set up the interrupts
     SetupInterruptSystem(&Intc, &Gpio,
GPIO_INTERRUPT_ID,&Timer,TIMER_IRPT_INTR);

     while(1){
     }

     return 0;
}

static void SetupInterruptSystem(XScuGic *GicInstancePtr, XGpioPs *Gpio,
u16 GpioIntrId,
            XScuTimer *TimerInstancePtr, u16 TimerIntrId)
{


            XScuGic_Config *IntcConfig; //GIC config
            Xil_ExceptionInit();
            //initialise the GIC
            IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
            XScuGic_CfgInitialize(GicInstancePtr, IntcConfig,
                                  IntcConfig->CpuBaseAddress);
          //connect to the hardware
            Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,

      (Xil_ExceptionHandler)XScuGic_InterruptHandler,
                           GicInstancePtr);
          //set up the GPIO interrupt
```

```
            XScuGic_Connect(GicInstancePtr, GpioIntrId,
                              (Xil_ExceptionHandler)XGpioPs_IntrHandler,
                              (void *)Gpio);
            //set up the timer interrupt
            XScuGic_Connect(GicInstancePtr, TimerIntrId,
                                    (Xil_ExceptionHandler)TimerIntrHandler,
                                    (void *)TimerInstancePtr);
            //Enable  interrupts for all the pins in bank 0.
            XGpioPs_SetIntrTypePin(Gpio, pbsw,
XGPIOPS_IRQ_TYPE_EDGE_RISING);
            //Set the handler for gpio interrupts.
            XGpioPs_SetCallbackHandler(Gpio, (void *)Gpio,
GPIOIntrHandler);
            //Enable the GPIO interrupts of Bank 0.
            XGpioPs_IntrEnablePin(Gpio, pbsw);
            //Enable the interrupt for the GPIO device.
            XScuGic_Enable(GicInstancePtr, GpioIntrId);
            //enable the interrupt for the Timer at GIC
            XScuGic_Enable(GicInstancePtr, TimerIntrId);
            //enable interrupt on the timer
            XScuTimer_EnableInterrupt(TimerInstancePtr);
            // Enable interrupts in the Processor.
            Xil_ExceptionEnableMask(XIL_EXCEPTION_IRQ);
      }

static void TimerIntrHandler(void *CallBackRef)
{

      XScuTimer *TimerInstancePtr = (XScuTimer *) CallBackRef;
      XScuTimer_ClearInterruptStatus(TimerInstancePtr);
      printf("****Timer Event!!!!!!!!!!!!!****\n\r");

}


static void GPIOIntrHandler(void *CallBackRef, int Bank, u32 Status)
{
      int delay;
      XGpioPs *Gpioint = (XGpioPs *)CallBackRef;

      printf("****button pressed****\n\r");

      if (toggle == 0 )
            toggle = 1;
      else
            toggle = 0;

      //load timer
      XScuTimer_LoadTimer(&Timer, TIMER_LOAD_VALUE);
      //start timer
      XScuTimer_Start(&Timer);
      XGpioPs_WritePin(Gpioint, ledpin, toggle);

      for( delay = 0; delay < LED_DELAY; delay++)//wait
      {}
      XGpioPs_IntrClearPin(Gpioint, pbsw);
}
```

Implementing the Private Timer

In the last blog we looked at the private timers provided with each CPU in this blog we will look at the private watchdog timer how it can be used and an example of its use.

However, before we look at how we can configure and use the watchdog up, I think it would be a good idea to explore why we may need a watchdog timer and how one works. Watchdogs enable unresponsive software to be addressed and provide a reliable solution, as no matter what the end application, all engineers want to provide a reliable solution

To ensure this the engineering team will undertake a number of steps the first of which will be establishing a requirements baseline which defines the behaviour of the system and software. The engineering team will then implement these requirement following a software life cycle which includes, generation of the design documentation, software design and source code along with a validation strategy which verifies the requirements have been achieved.

Most designs will include a methods to ensure the software can respond to faults within the system. These faults can have one of two effects either allowing the software to continue operation while maintaining either a full or reduced service or they can result in the software failing to respond. It is for when the software fails to respond that systems include a watchdog timer which can be used to restart the system or ensure the system fails in a safe manner (The issue of safety critical systems and software is a complex one and would take more room than I have here to cover in detail here).

In the very simplest sense a watch dog is a timer which counts down from a value pre-loaded by the application software, as the software application executes it resets the watchdog. Should the watchdog not be reset it will reach zero and reset the processor. Many systems have a register which indicates the watchdog has triggered and come up back from reset caused by the watchdog allowing the fact to be reported.

Within the Zynq each processor has a private watchdog these private watchdogs can be used as either a timer like the private timer or as a watchdog. It is controlled via six registers

Watchdog Load Register This is the value which the watchdog timer counts down from, it is used in auto reload mode as the value which the watchdog counter is reset to. Writes to this register will result in the watchdog counter register being reset to this value.

Watchdog Counter Register This is the watchdog counter itself it is decrementing counter, depending upon the mode it can be written too to reload the counter. In watchdog mode this register can only be updated by writing to the watchdog load register.

Watchdog Control Register This register controls the configuration of the watchdog (timer or watchdog), pre-scaler, interrupt enable, auto-reload and the enabling of the watchdog in its currently configured mode.

Watchdog Interrupt Status Register Contains a single event flag which shows when the counter has reached zero, this can be reset by writing to it

Watchdog Reset Status Register This register contains a single bit which is only cleared by power on reset (not a watchdog triggered reset) or by being cleared by the software application. This allows the software to know the reason for a reboot was caused by a watchdog timeout.

Watchdog Disable Register This register requires two specific patterns to be written to it to enable the watchdog mode bit within the Watchdog control register is set to timer mode.

As with the private timer the software development environment provides a number of functions and macros which can be used to configure and drive the watchdog. These are included within

```
#include "xscuwdt.h"
```

This file enables the facilities to

- Test if the watchdog is expired
- Load the watchdog
- Start, stop and restart the watchdog
- Set the watchdog mode
- Configure and initialise the watchdog

The example will configure the watchdog to operate as a traditional watchdog with no refresh such that it times out and rests the Zynq. It then checks following watchdog reset, what was the cause of the reset e.g. power on reset or watchdog and reports this over the STDOUT. The watch dog will be started by the push button being pressed which will start the private timer to illuminate the LED and start the watchdog.

To do this we follow a standard approach of configuring the watchdog as we have done for all of the peripherals to date using the date from xparameters.h and using the configuration routines.

```
//define private watchdog
#define WDT_DEVICE_ID          XPAR_SCUWDT_0_DEVICE_ID
#define INTC_DEVICE_ID         XPAR_SCUGIC_SINGLE_DEVICE_ID
#define WDT_IRPT_INTR          XPAR_SCUWDT_INTR
#define WDT_LOAD_VALUE         0xFF


//watchdog configuration
 WCHConfigPtr = XScuWdt_LookupConfig(WDT_DEVICE_ID);
 XScuWdt_CfgInitialize(&WdtInstance, WCHConfigPtr,WCHConfigPtr->BaseAddr);
 XScuWdt_LoadWdt(&WdtInstance, WDT_LOAD_VALUE);
 XScuWdt_Start(&WdtInstance);
```

Following initialisation and loading of the watchdog the next step is to enable the interrupt (within the interrupt configuration function) and set the watchdog to the watchdog function as opposed to the timer function using the XScuWdt_SetWdMode() function.

```
//set up the watchdog
XScuGic_Connect(GicInstancePtr,
WdtIntrId,(Xil_ExceptionHandler)WdtIntrHandler,
(void *)WdtInstancePtr);

//setup the watchdog
XScuWdt_SetWdMode(WdtInstancePtr);
```

Should we wish to use the watch dog in a timer mode instead we can call the function

```
XScuWdt_SetTimerMode()
```

Which is why I set up the interrupt to trigger on the watchdog for when it is running within its timer mode and declared an empty interrupt service routine for the watchdog which would be called in this instance.

Within our system if we want to check if the last reset was due to watchdog event we can use the function which reads the watchdog reset status register

```
XScuWdt_IsWdtExpired(InstancePtr)
```

The picture below show the results output over STDOUT when the processor is powered on from a power on reset and the reset which occurs when the button is pressed enabling the watchdog timer

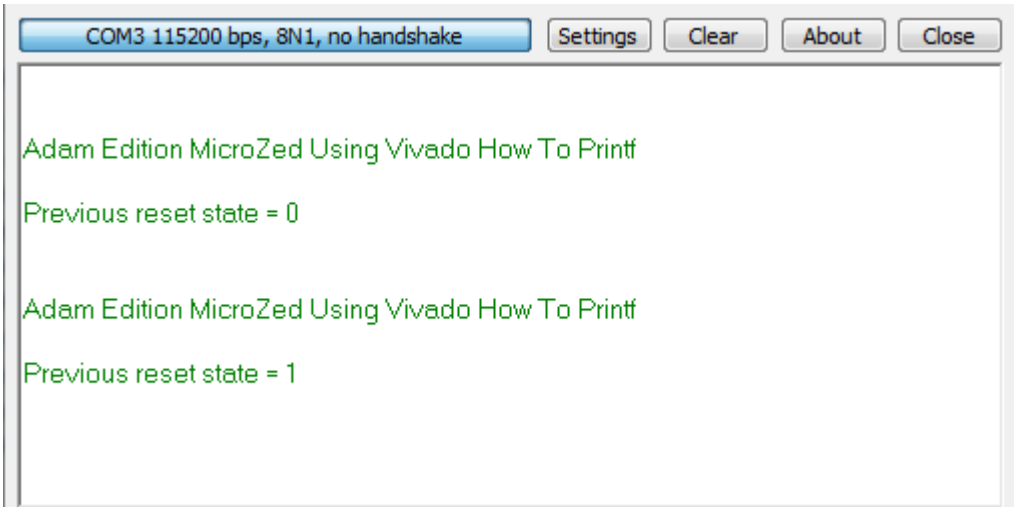


Figure 64 Private Watchdog

The previous reset state holds the state of the watchdog event stored within the watchdog reset status register accessed via XScuWdt_IsWdtExpired()

the Triple Timer Counter Part One

Over the last few blogs I have focused upon the private timers and watchdogs available within each of the Zynq ARM Cores. The Zynq also contains two triple timer counters (TTC) which provides a much more flexible timing resource these can be used as a timer or can be used to generate a waveform output on either the EMIO or MIO. The diagram below which is contained within the Technical reference manual shows the architecture of each TTC and how each one is independent of another.

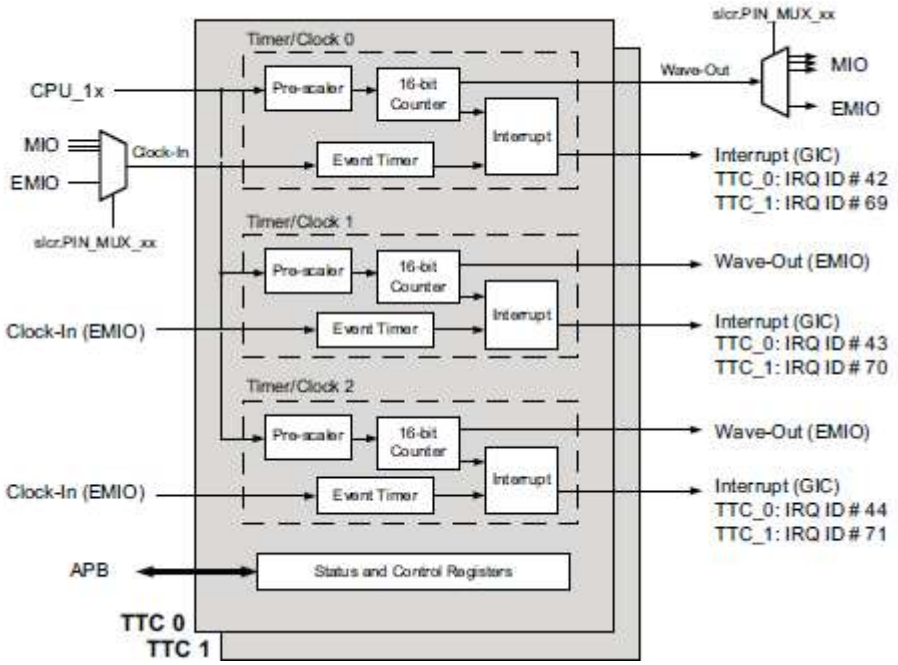


Figure 65 TTC Structure

It is not clearly shown in the diagram above however, each of the pre-scalers can be clocked by either the processor clock, PL via EMIO or MIO and is selectable via the clock control register.

So what can we use the TTC for the most obvious implementation is to use it as more detailed timer or scheduler which generates different interrupts at specified count values. The TTC can also be used to generate a waveform with a set duty cycle for instance in the most basic instance toggling a LED to show the processor is up and running the application. But what makes the TTC very flexible it is ability to generate a Pulse Width Modulated output, in embedded systems PWM can be used for a number of applications especially in industrial motor control and has a number of advantages not least of which is its noise immunity.

As many industrial systems use PWM for communication (e.g. in older smart meters between sensors or instruments) the TTC also provides the ability to receive a PWM signal provided the duty cycle is known (in processor clock cycles) as the event timer can be configured to count the number of processor clock cycles an external clock signal is high or low.

While there are three TTC in each of the two instantiations each TTC has the following registers:-

Clock Control : Defines the clock source, pre-scale value and edge of the clock to be used

Counter Control : Defines the generated waveform settings, defines the timer mode, count direction and enabling of the match and interval interrupts. Along with counter reset and disable controls.

Counter Value : A read only register defining the current value of the timer.

Interval Counter : A intermediate value which is used in the interval mode as the value that is counted to or from depending upon the direction of the count.

Match Counter * 3 : When the match registers are enabled a separate interrupts are generated when the counter value is equal to that defined in these registers

Interrupt Register : Defines the status of the six interrupts available from a TTC, the permissible interrupts are Match 1, Match 2, Match 3, Internal, Overflow and Event.

Interrupt Enable : Used to enable the interrupts available in the TTC

Event Control Timer : Configures the event control timer e.g. does the counter, count the high or low signal duration.

Event Register : The number of processor clock cycles counted by the event control register.

To use the above registers each TTC has two basic modes of operation interval or overflow mode, plus the event timer.

Interval mode : The counter counts to a value contained within the interval register, counting either up or down and generates an interval interrupt when enabled whenever zero is reached.

Overflow mode : The counter increments or decrements between 0 and full scale as the counter wraps around the overflow interrupt is generated

In both of these modes the match interrupts will be generated when the counter equals the values within the match registers if enabled.

Generation of the output waveform uses the count value contained within the match count 1 register to generate the required duty cycle, this works in both the interval and overflow modes. When the counter value equals that defined within the Match Counter 1 register the waveform being output will change polarity from either 1 to 0 or 0 to 1 depending upon the setting of the waveform polarity bit within the counter control register which the waveform defaults to. The waveform again inverts its state upon the generation of either the interval or overflow interrupt depending upon the mode of the timer.

The event timer can be used only with an external source and is how an engineer would measure the duration of an event or decode a PWM IP signal, in the next blog we will look at an examples of using the TTC.

the Triple Timer Counter Part Two

In the last blog I introduced the very versatile triple timer counter, in this blog I will be looking at how we can use the TTC to generate an output waveform. To do this we will need to use both Vivado and SDK in this blog we will focus upon what is needed to be done in Vivado.

The first thing to do is ensure within the block diagram we created within Vivado that the TTC is enabled.

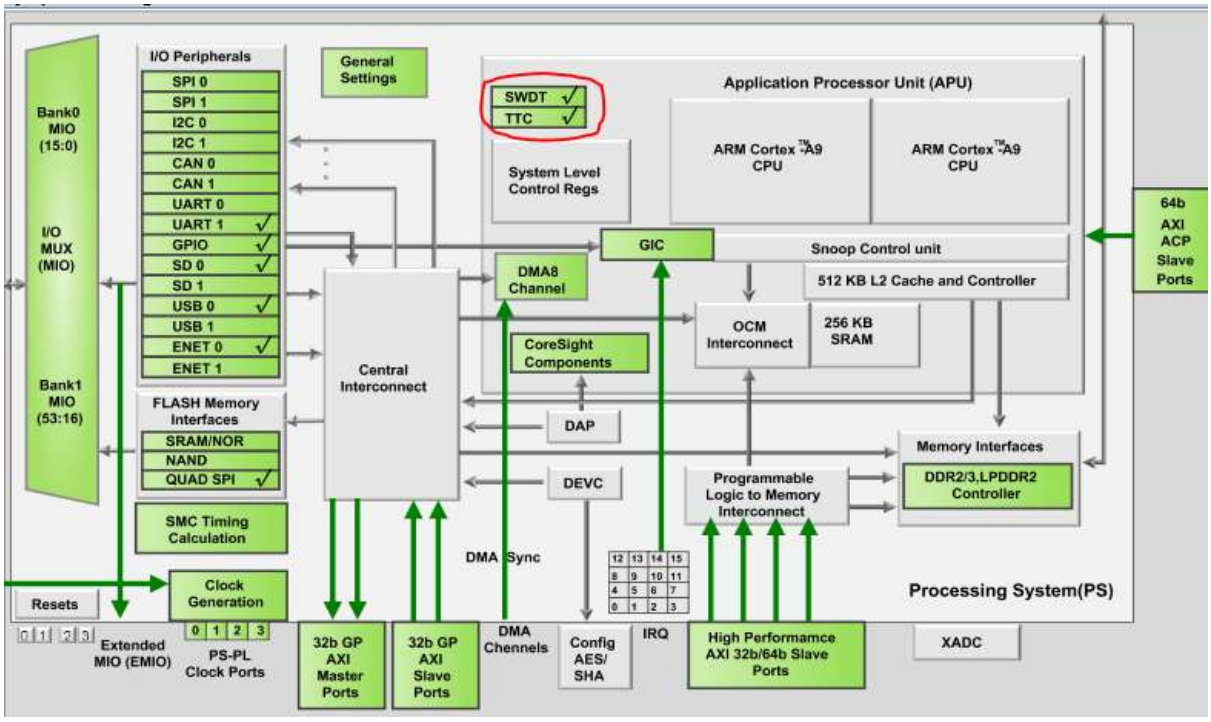


Figure 66 Ensuring the TTC is Enabled

Once this is enabled the next step is to ensure the output is selected to be on the Extended MIO within the programmable logic side of the device. This is required as the MIO locations which are available for the both TTCs are being used by the Ethernet, USB and SD Card interfaces.

To use the EMIO we need to select the EMIO option within the MIO configuration of the Zynq re-customisation screen

Figure 67 Selecting the IO for the watchdog and TTC

While we are within the re-customisation dialog it is also required to use the clock configuration page and select that the TTC will be clocked by the internal clock.

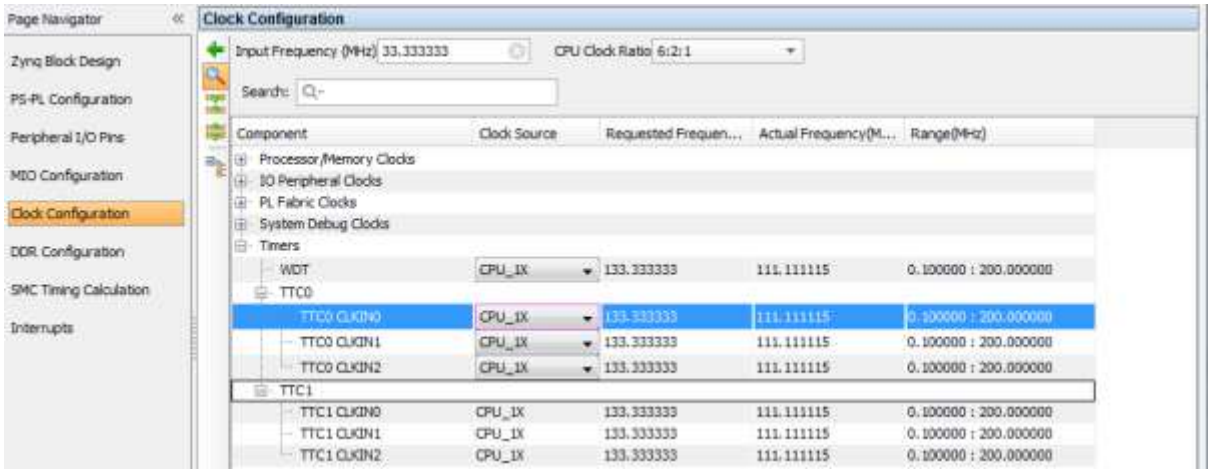


Figure 68 Selecting the TTC drive clock

Having done both of these you can close the re-customisation window and you will notice that a number of new ports have appeared on the Zynq block within the block diagram these are the TTC clock in and TTC waveform out. As we are only interested in using one of the three waveform outputs in this example we will only use the waveform output TTC0_WAVE0_OUT, the next step is to create an output port and connect it to the Zynq. This is as simple is right clicking within the block diagram and selecting the create port option, once you have named this port connect it to the TTC_WAVE0_OUT output on the Zynq block.

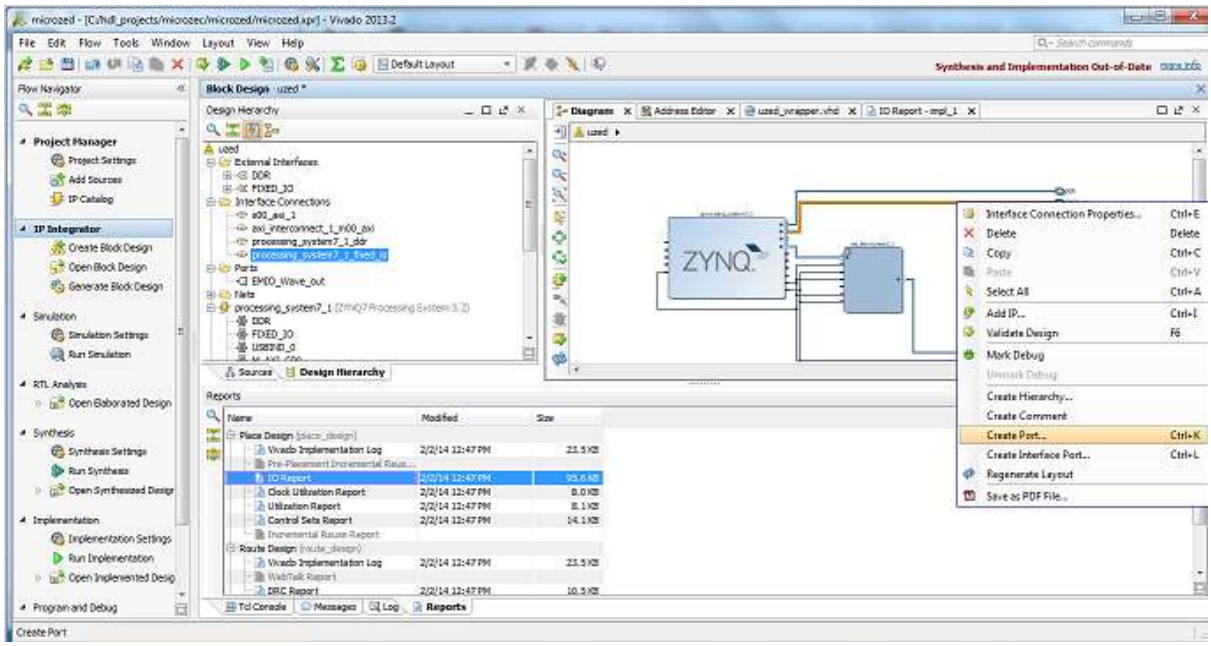


Figure 69 Creating the Output Port

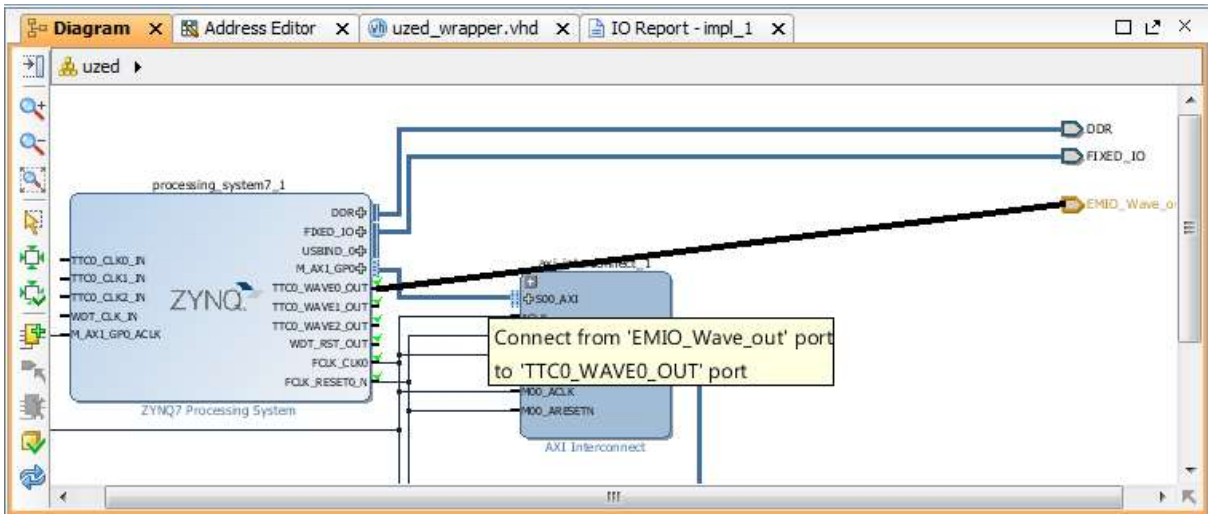


Figure 70 Connecting the output port to the Zynq TTC port

As the port we are using is not within the PS side of the device for the TTC waveform output we need to create a constraint file to define which pin on the PL side of the device is to be used to output the generated waveform. A constraints file can be created by selecting within the sources window the constraints option and right clicking upon constrs_1 and selecting edit constraints sets

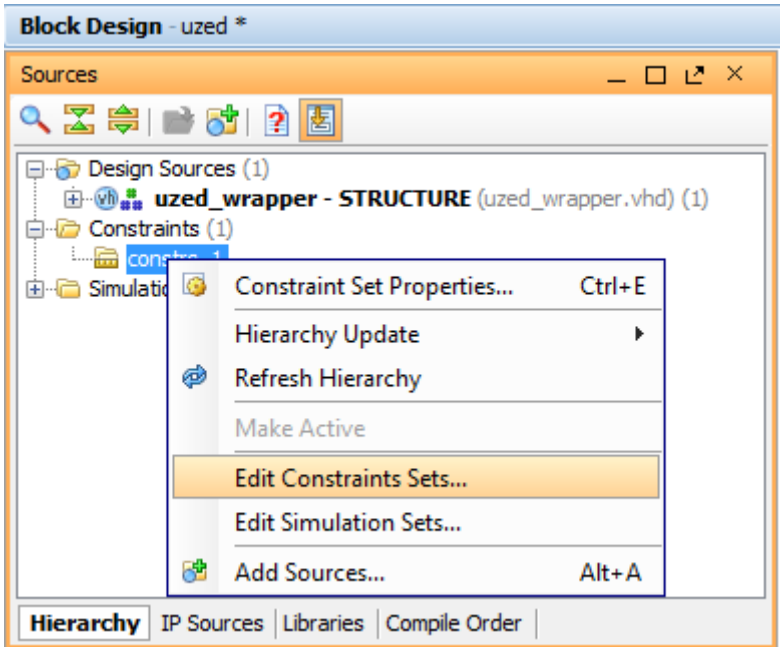


Figure 71 First step in creating the constraints file

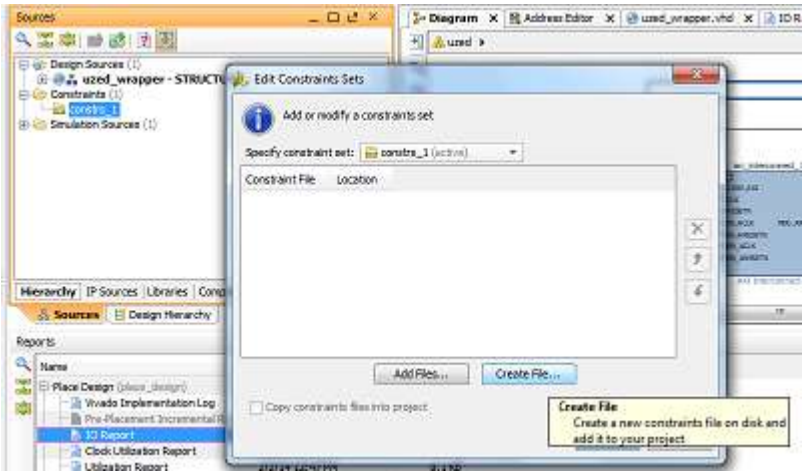
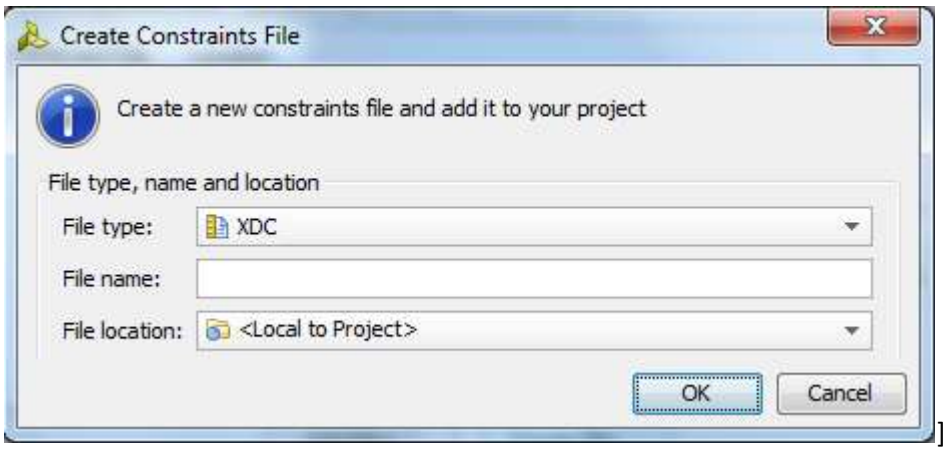


Figure 72 Creating a new constraints file.

As we do not currently have a constraints file at the next dialog we must select create new file as opposed to the add files option this will open file creation dialog.



]

Figure 73 Selecting the format and name

Enter a file name desired for your constraints file and it will open a blank constraints file ready for editing. Vivado uses XDC formatting for constraints as opposed to the UCF formatting used with ISE if you are unsure as to the new syntax there is helpfully a language template which can be used to explore and find the syntax needed for your XDC file. You can also find more help on XDC here at http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_1/ug903-vivado-using-constraints.pdf

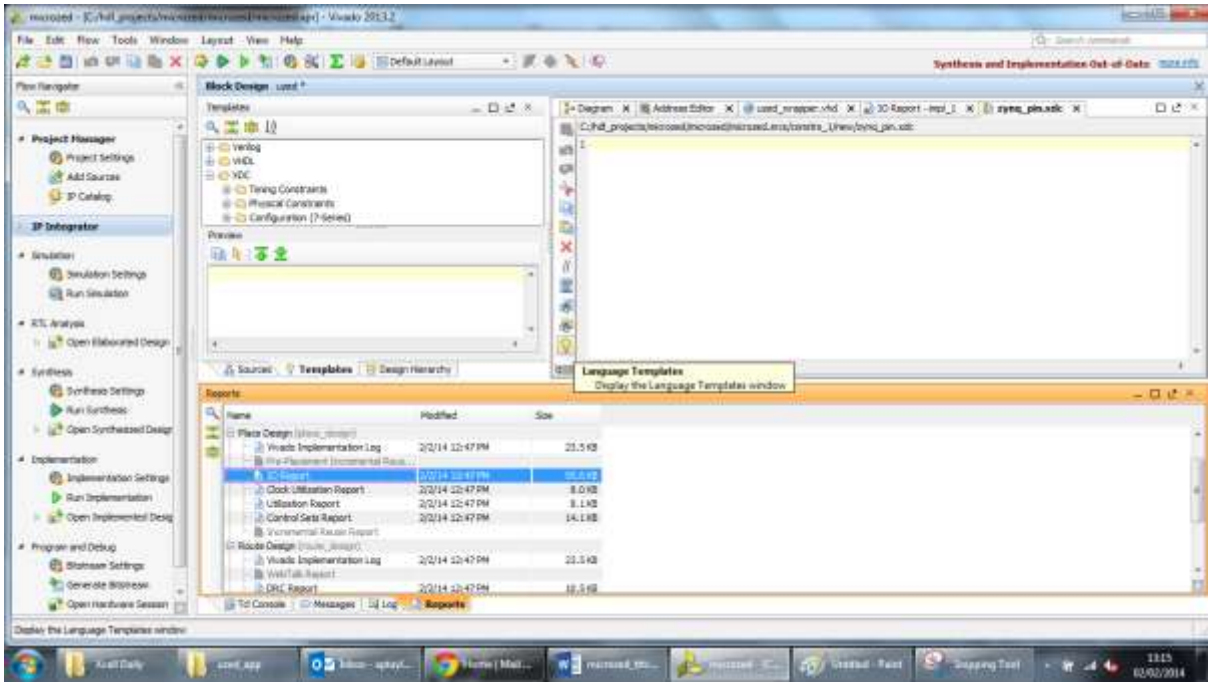


Figure 74 The language template to help writing XDC

As we are using only one output this will be a very simple XDC file, which will output the wave output to a pin on the micro header. The pin in use if R19 which is located upon back three four of the Zynq this is a high range (HR) bank and as such supports 3v3 standards. High performance (HP) banks do not support voltage standards above 1v8.

```
1 set_property PACKAGE_PIN R19 [get_ports {EMIO_Wave_out}]
2 set_property IOSTANDARD LVCMOS33 [get_ports {EMIO_Wave_out}]
3 set_property PULLUP true [get_ports {EMIO_Wave_out}]
```

Figure 75 XDC pin constraints

Once we have created this constraints file we can implement the design and export it to SDK allowing us to write software to drive the timer. Before we export it to SDK however, it is first prudent to check the EMIO pin is correctly placed on the pin desired using the IO Report.

333	R19	EMIO_Wave_out	IOB33	IO_0_34	OUTPUT	LVCMOS33

Figure 76 IO Report confirming pin placement.

Triple Timer Counter Part Three

The last blogged we looked at how we implemented the TTC within the hardware definition using Vivado. In this blog we will look at how we can use SDK to drive the TTC, obviously the first thing we need to do is export the hardware to SDK to ensure the software environment has the most up to date definition.

Having exported the hardware to SDK as always the first thing to do is pull in the functions and macros provided by the operating system these can be accessed by including

```
#include "xttcps.h"
```

You will also need to include the `Xscugic.h` and `Xil_exception.h` to ensure you can use the interrupt controller.

The next step as it always is, is to use the xparamters.h file to obtain the TTC device ID, TTC Interrupt ID and Interrupt controller device ID.

Unlike with the private timer we need to declare a structure to contain the output frequency, interval, pre-scaler and TTC options.

```
typedef struct {
	u32 OutputHz;		/* Output frequency */
	u16 Interval;		/* Interval value */
	u8 Prescaler;		/* Prescaler value */
	u16 Options;		/* Option settings */
} TmrCntrSetup;
```

This structure makes driving the TTC very easy, I also used this structure to define a pre-configured settings table.

```
static TmrCntrSetup SettingsTable[1] = {
	{10, 0, 0, 0},		/* Ticker timer counter initial setup, only output
freq */
};
```

This settings table sets initially the output frequency to be 10Hz while leaving everything else initialised as zero.

After the initialisation and configuration of the device the next step is to define the options mode of operation for the TTC. Helpfully defined within xttcps.h are all of the options for the timer control register we can therefore configure it by ORing the options together. For this example I will be running the timer in interval mode with the external wave disabled.

```
TimerSetup->Options |= (XTTCPS_OPTION_INTERVAL_MODE |
                                        XTTCPS_OPTION_WAVE_DISABLE);

XTtcPs_SetOptions(&Timer, TimerSetup->Options);

XTtcPs_CalcIntervalFromFreq(&Timer, TimerSetup->OutputHz,&(TimerSetup-
>Interval), &(TimerSetup->Prescaler));

XTtcPs_SetInterval(&Timer, TimerSetup->Interval);
XTtcPs_SetPrescaler(&Timer, TimerSetup->Prescaler);
```

Having defined the options and set them into the control register, we can call a function which will calculate and set the interval for the desired output frequency as defined within the timer set up structure.

The next stage of using the TTC is to connect this to the interrupt controller, and enable interrupts as there are several interrupts which can be used for each TTC I have enabled only the interval interrupt in this instance.

```
XTtcPs_EnableInterrupts(TtcPsInt, XTTCPS_IXR_INTERVAL_MASK);
```

The file xttcps_hw.h which is called up by xttcps.h includes definitions for all possible TTC interrupts allowing them to be enabled as desired using the above function.

Having set up the interrupt I then started the timer and connected the MicroZed to my laptop such that I could see a message printed out each time the interrupt occurred.

Within the interrupt service routine, I have simply read back the interrupt status register to determine which interrupt occurred and then cleared it down. The reading of the interrupt status register within the ISR is important as there are several different possible interrupts and in more complex use of the TTC we want to ensure we take the correct action depending upon the interrupt.

From this basic example (see attached code) you can add in the use of the match registers or more complex functions which I shall look at next time.

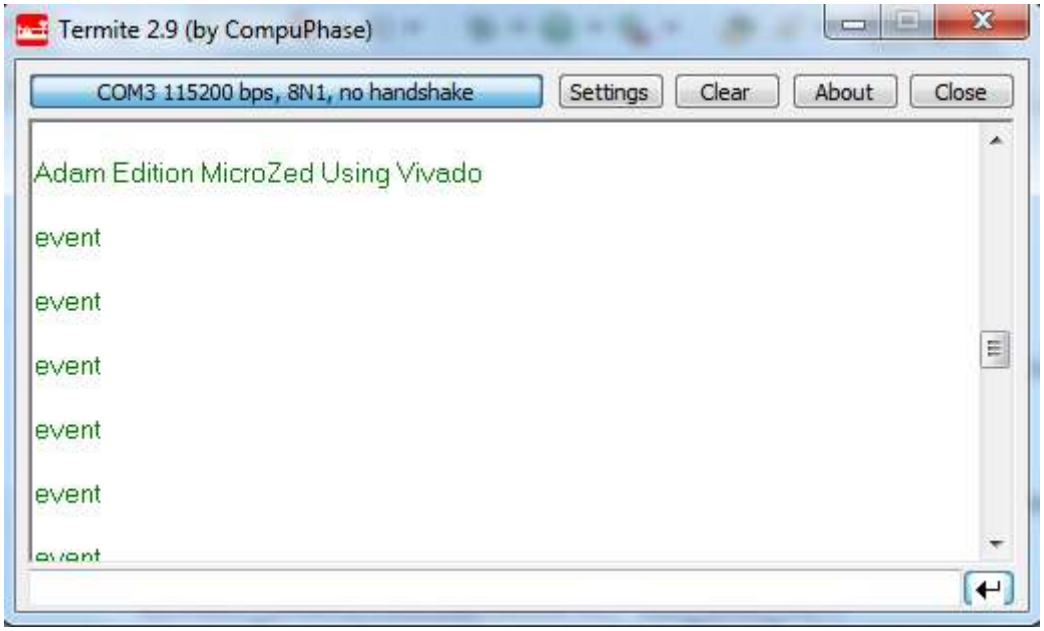


Figure 77 Timer Events

Triple Timer Counter Part Four

When we last looked at the TTC we had configured one of the three timers within TTC to operate in simple interval mode generating an interrupt at the desired frequency. However we can use the TTC to do much more than that, so in the final part of looking at the TTC I thought I would explore looking at more complex uses of it.

The first example we will look at is using the match registers to issue interrupts at different points of the counter, this will then enable us to easily generate an output waveform as when the waveform output is enable it's status is inverted when the match 1 value is reached.

The first step in this is to enable the match mode within the timer set up options, we can OR the XTTCPS_OPTION_MATCH_MODE as defined within the xttcps.h file to enable the match mode along with the disabled waveform and interval mode.

Having set the match mode the next steps are very simple configure the match register with value we wish to trigger the interrupt at using the function below.

```
XTtcPs_SetMatchValue(&Timer, 0, (interval/3));
```

In the example above the match value of match register one is defined to trigger at one third of the value defined within the interval counter. I have used the function `XTtcPs_GetInterval()` to access the value of the interval counter.

The next stage is to enable the match interrupt, in this example we only need to enable match interrupt one but can use the definitions provided within xttcps.h again

```
XTtcPs_EnableInterrupts(TtcPsInt, XTTCPS_IXR_MATCH_0_MASK);
```

Within the interrupt service routine we then need to determine the cause of it being called i.e. as it called for the interval interrupt as well. We can determine what event caused this pretty easily by performing a AND operation between the interrupt status register and an interrupt definitions.

```
if (0 != (XTTCPS_IXR_MATCH_0_MASK & StatusEvent)) {
        printf("match interrupt event\n\r");
                                }
```

This allows the ISR to take different action depending upon the source of the interrupt, in this case it a different message is printed out over STDOUT.

Figure 78 match and interval interrupts

If we decided to output the waveform it is as simple as enabling the waveform output within the option registers and double checking this is output correctly on the selected output. The polarity can also be selected using the `XTTCPS_OPTION_WAVE_POLARITY` option.

Other advanced uses of the TTC is a create a Real Time Clock, with the TTC configured to produce an interrupt at the required time resolution and then increase the count each time the interrupt occurs. RTC can be very useful in embedded systems with one example being the time stamping of when events within the system have occurred.

PS / PL Part One

So far in this epic series of blogs, we have looked at

- Creating the Zynq system in Vivado
- Configuration and boot loading
- Using the XADC
- Using the MIO and EMIO
- Interrupt structure on the Zynq
- Zynq private timers and watchdogs
- Triple Timer Counter

All of these functions are primarily focused upon the processing system (PS) side of the Zynq however, the exciting aspect of the Zynq is creating an application which uses the programmable logic (PL). The use of the PL allows off-loading from the PS to the PL side or allows the PS side to control the operations performed by the PL side in a classic system on chip application.

Using the PL side of the device can result in increased system performance, reduced power and a predictable latency for real time events.

The PS and PL are interconnected via the following interfaces

- Two 32 bit Master AXI ports (PS master)
- Two 32 bit Slave AXI ports (PL Master)
- Four 32 / 64 bit Slave High Performance Ports (PL Master)
- One 64 bit Slave Accelerator Coherency Port (PL Master)
- Four Clocks from the PS to the PL
- PS to PL Interrupts
- PL to PS Interrupts
- DMA peripheral request interfaces

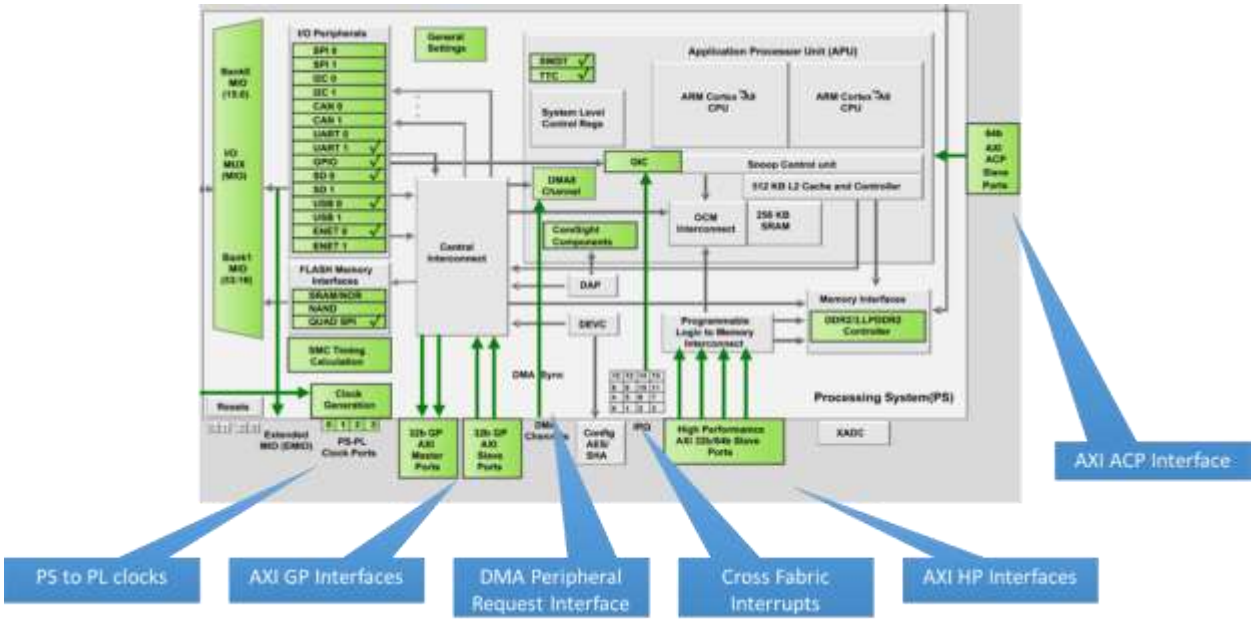


Figure 79 PS/PL interconnection

The AXI ports each contain independent read and write channels, AXI is a burst orientated protocol intended for high bandwidth while providing a low latency. One adoption of the AXI protocol which is used by less demanding interfaces is AXI-Lite this is a simpler protocol which can be used for register style control interfaces. Indeed in the XADC this was connected to the PS system using an AXI-lite interface. More information on the AXI protocol can be found here http://www.arm.com/products/system-ip/amba/amba-open-specifications.php

The Zynq supports three different types of AXI interconnects which can be used to interface to PL side of the device.

- AXI Burst transfers
- AXI-lite for simple control interfaces
- AXI Streaming for unidirectional data transfers

As engineers we are responsible for selecting the optimal interface to ensure the system performance can be achieved in the most cost effective solution. Therefore the theoretical bandwidths of each of the interfaces are defined in the table below.

Interface	Width	IF Clock	Read BW	Write BW	Combined	No Ports	Total BW
AXI GPIO	32	150 MHz	600 MBps	600 MBps	1200 MBps	2	2400 MBps
AXI HP	64	150 MHz	1200 MBps	1200 MBps	2400 MBps	4	9600 MBps
AXI ACP	64	150 MHz	1200 MBps	1200 MBps	2400 MBps	1	2400 MBps

To achieve the speeds listed within the table above and reduce the load on the A9 processors when the PS is the master the DMA controller is required. If the DMA controller is not used then the maximum date rate which can be transferred from the PS to the PL side 25 MBps maximum.

All told there is a phenomenal theoretical 14.4 GBps (115.2Gbps) bandwidth between the PS and the PL, over the next few blogs we will look more in detail at how we create and use our own peripherals within the PL side of the device to increase the system performance.

PS / PL Part Two

Having introduced the AXI interfaces in my last blog, the first step in creating a peripheral we can use within the Programmable logic side of the design is to open the Vivado design and select the “create and package IP” option from under the tools option. Note I am using Vivado 2013.4 for this blog.

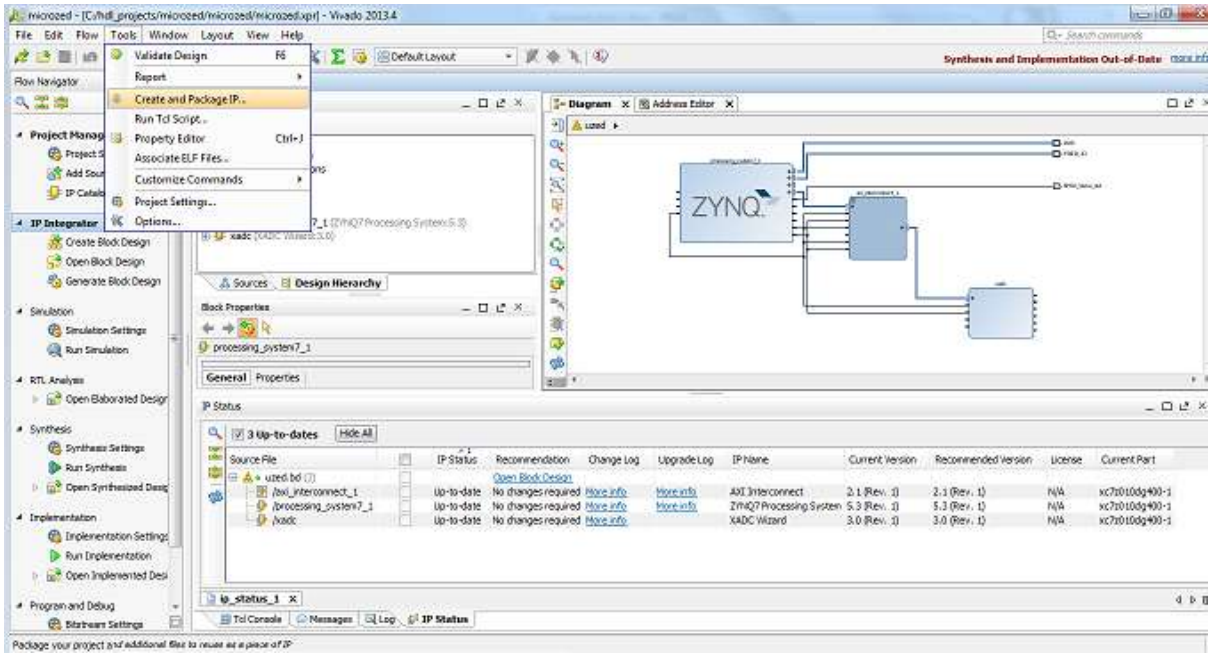


Figure 80 creating IP

This will open a dialog box which will allow you to create AXI4 peripherals, the first real page of the dialog presents a number of options to either create a new IP block or turn your current design or a directory into an IP module.

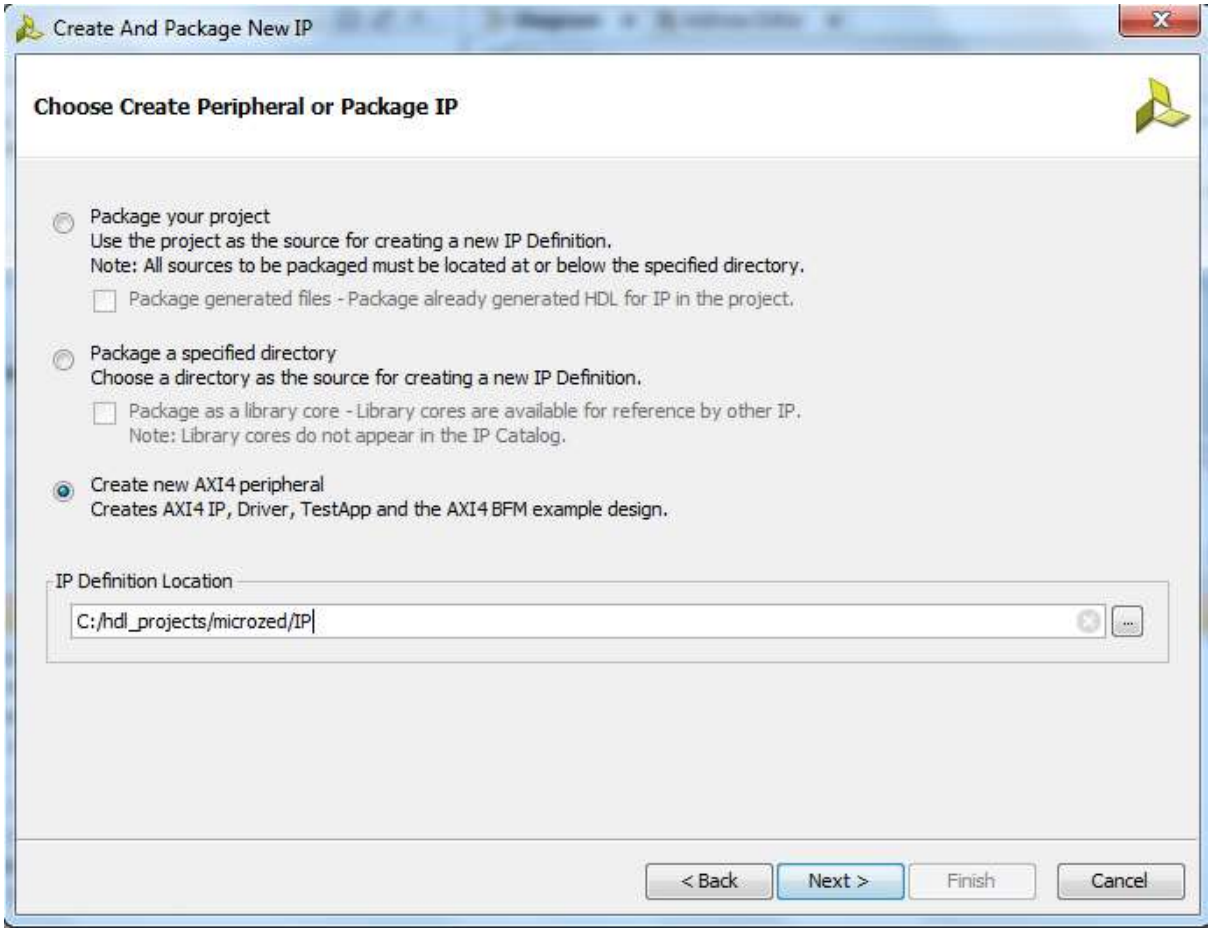


Figure 81 AXI4 Peripheral

Select create new AXI peripheral option and point it to an IP defined location, a new IP location can be created by using the Manage IP section on the Vivado home page.

Figure 82 New IP Location

The next stage of the dialog allows you to enter the library, name, description and company URL that you wish to use for the peripheral. For this very simple example which I will expand upon later, I have named it Adams_peripheral and pointed the URL back to xcell daily blog.

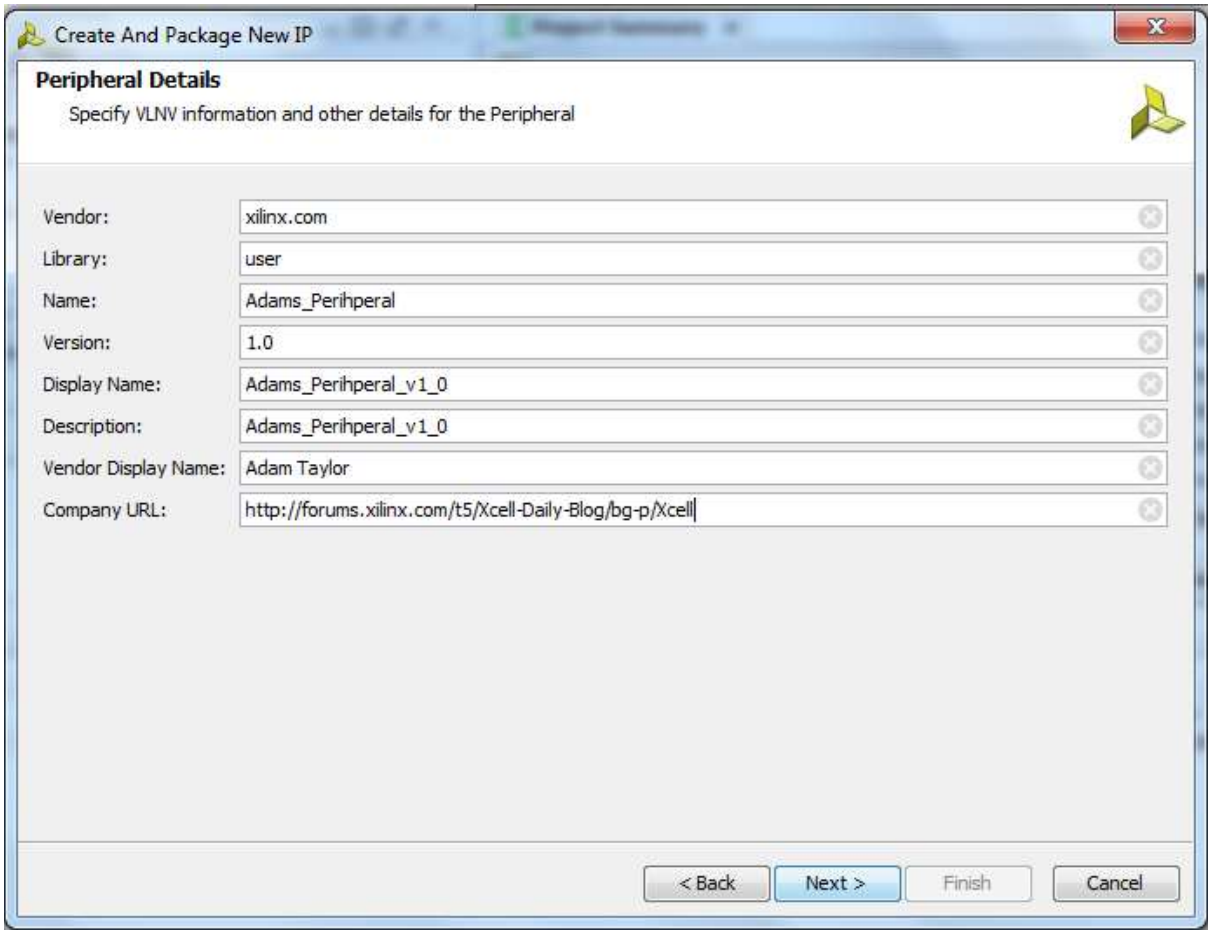


Figure 83 Peripheral Information

The next dialog box after this is the powerful one where we can define the type of interface we wish to specify

- Master or Slave
- Interface type – Lite, Streaming or Burst
- Bus width 32 or 64 bit
- Memory size
- Number of Registers

This initial example is going to be very simple such that I can demonstrate the flow needed to create the peripheral implement it within Vivado and then export it to the SDK. For this reason I shall use an AXI lite interface that contains just four registers which we can then address using software. These registers could be used to control the operation of functions within the Programmable logic side of the design.

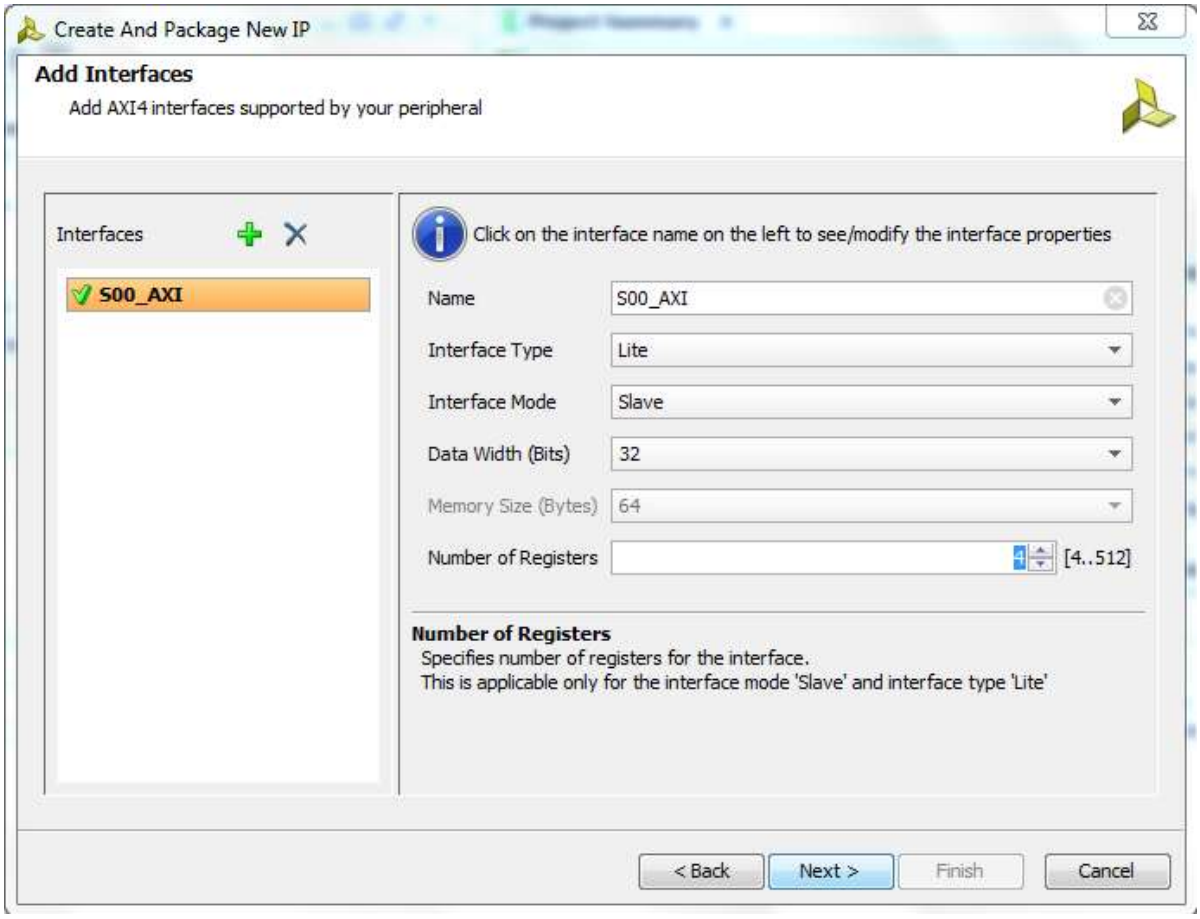


Figure 84 AXI settings

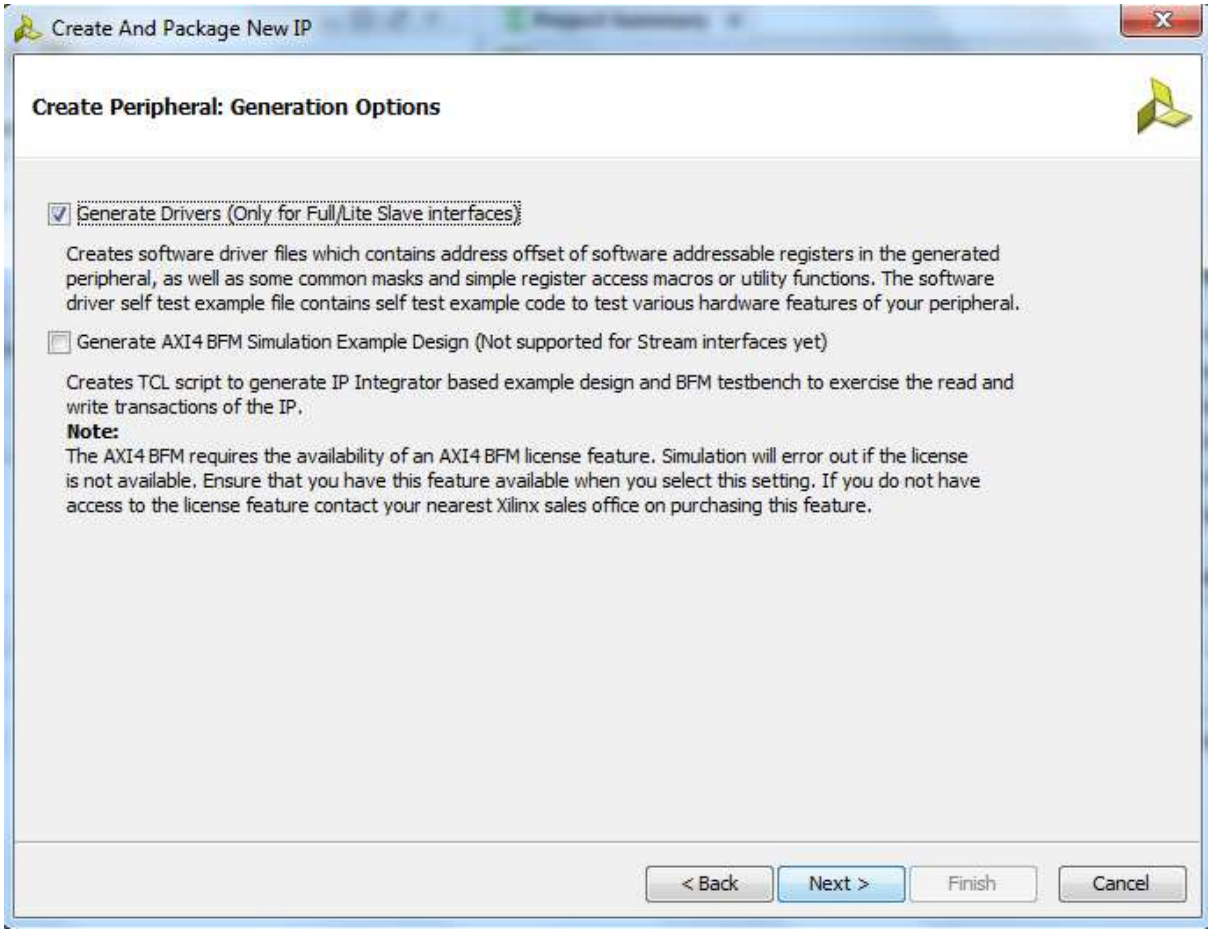


Figure 85 Generating the drivers

On the final dialog it is important that you select for the generation of the driver files, this will make the use of the peripheral with SDK much simpler.

Once the wizard is closed you can if you want open the created VHDL file and add in your user design to perform the function you desire in the PL. For this simple example initially I will just be using the four registers we created and therefore can leave the files unedited.

Having created the perihperal we will want to connect this and use it within the vivado design, doing this is very simple. Opening up the block diagram of the system we can select the Add IP option from the left hand menu, you should be able to find the peripheral created as they are listed alphabetically.

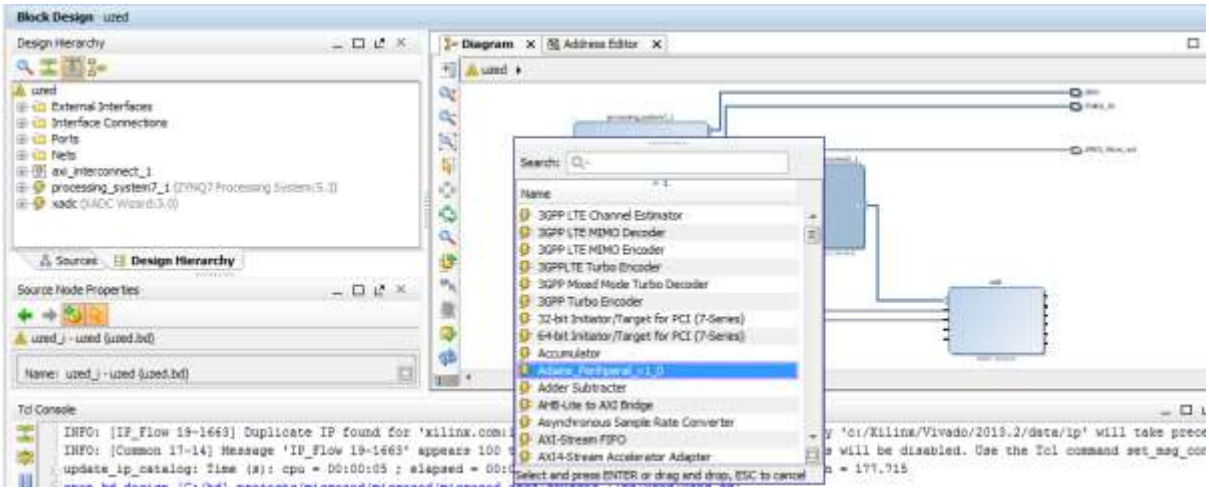


Figure 86 Adding in the Peripheral

Dragging this IP block into the design you can then connect it to the AXI GP bus, Vivado helpfully offers designer assistance to connect the new peripheral automatically which can be seen in the green bar across the top of the image below. Running the connection automation resulted quickly results in a design we can implement.

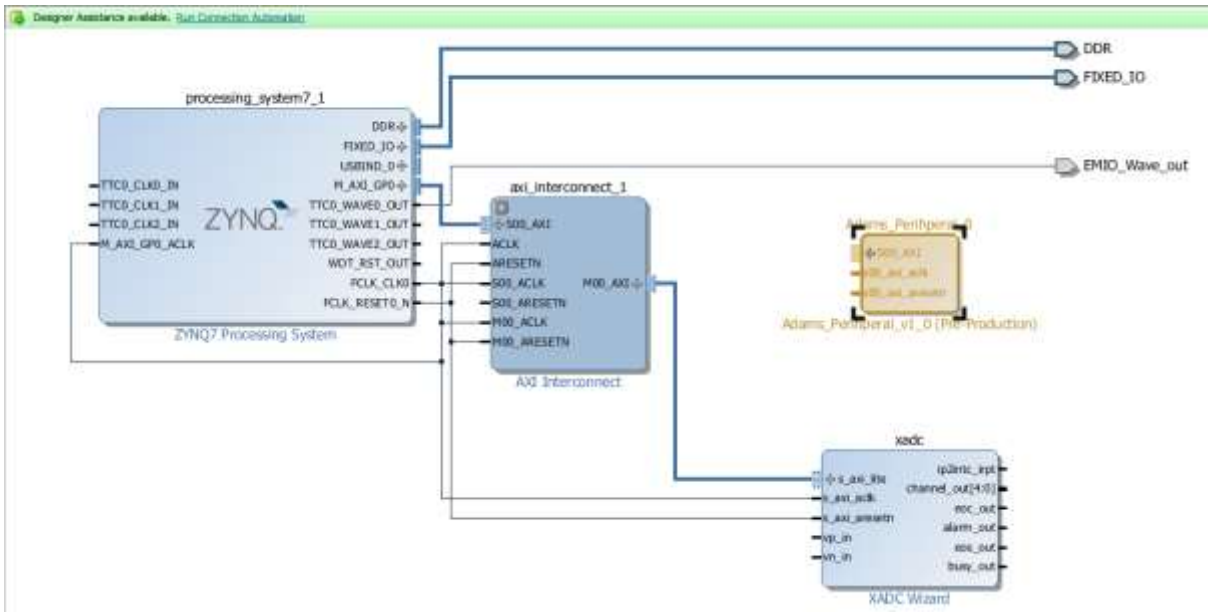
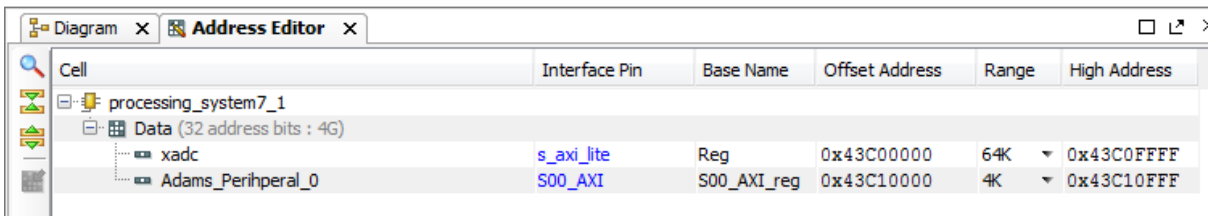


Figure 87 Connecting in the Block Diagram

The address range of the peripheral can be modified by clicking on the address editor tab, note the 4k address space is the lowest allowable.



Cell	Interface Pin	Base Name	Offset Address	Range	High Address
processing_system7_1					
Data (32 address bits : 4G)					
xadc	s_axi_lite	Reg	0x43C00000	64K	0x43C0FFFF
Adams_Perihperal_0	S00_AXI	S00_AXI_reg	0x43C10000	4K	0x43C10FFF

Figure 88 Address Allocation

Once the connections have been automatically inserted by Vivado (as shown in the diagram below) we are then in a position to implement the design and export it to SDK such that we can start to make use of our peripheral.

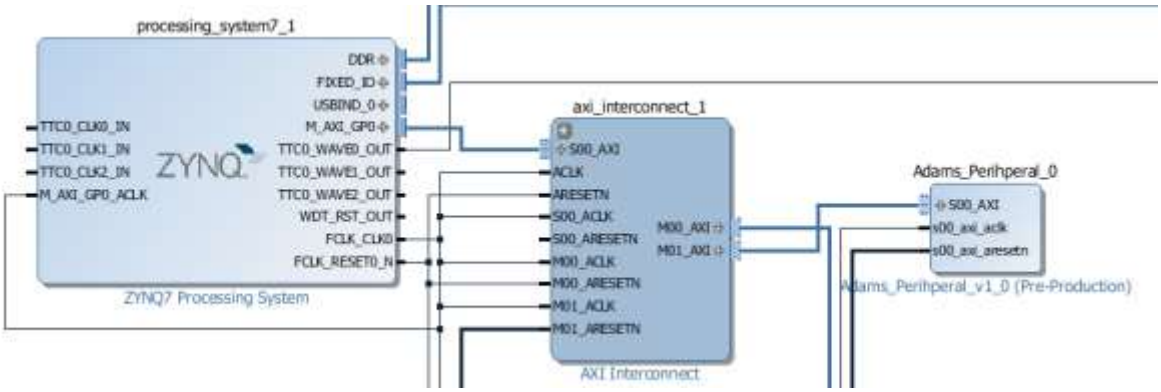


Figure 89 Connected Peripheral

Once this has been implemented you can check the implementation reports to ensure the inclusion of the peripheral created.

In the next instalment we will look at how we can use this within SDK before we explore more complicated interfaces and use of this ability.

PS / PL Part Three

In the previous blog we had used Vivado to create the peripheral and generate a bit file. Having created the hardware components of the design we need to export it to our SDK design such that we can write the software to drive it.

The first stage of this is to open the current implementation within Vivado and then export the hardware to SDK. Once this is completed SDK will issue a warning if it is currently in use or the next time you open SDK as the hardware definition and the board support package will need to be updated.

It is also necessary to update the repositories defined within the design to include the IP repository which contains the peripheral. To do this we selecting the Xilinx tools options and then repositories and add them as either local or global repositories. For this example I have chosen local.

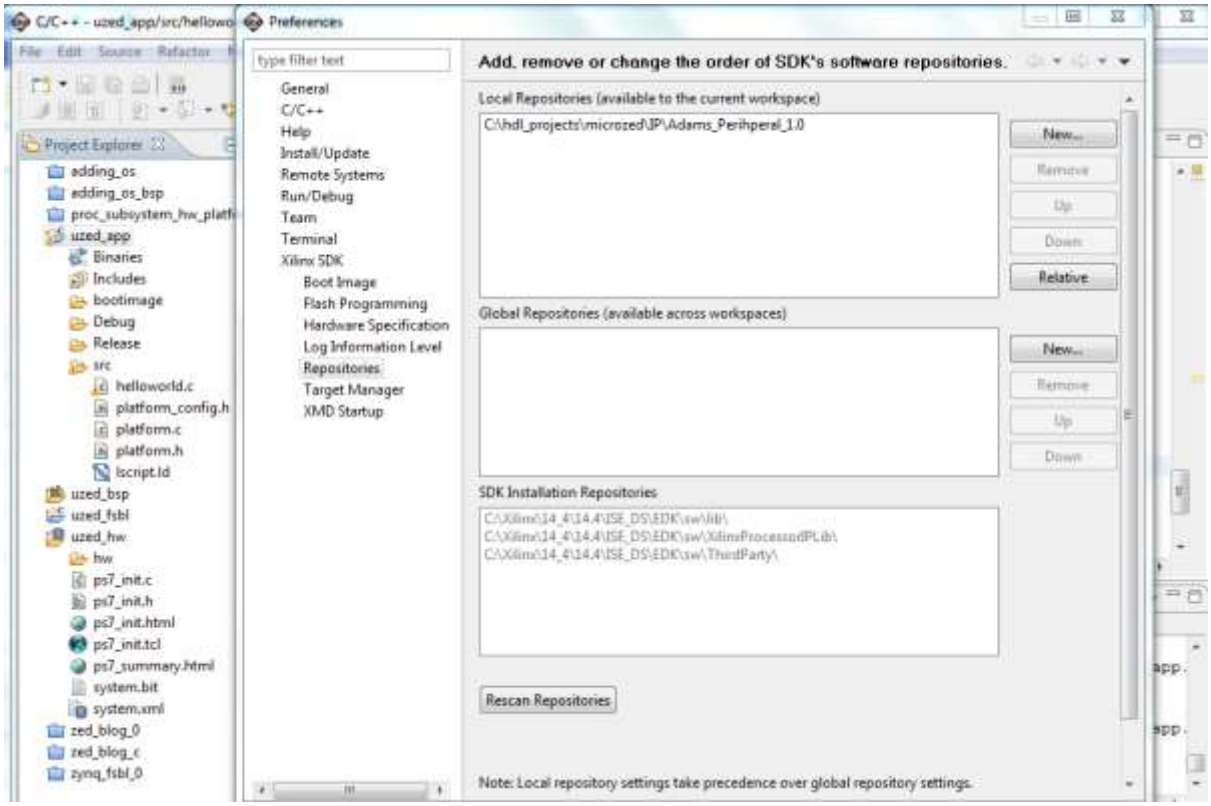


Figure 90 Adding Software Repositories for the New Peripheral

Add in the peripheral directory created by Vivado and rescan the repositories, having completed this it then allows us to then re build the project which will recreate the files needed in the BSP to support the software development.

Once the re-build has been completed, opening up the xparameters.h file (within the BSP include files) will show the address space dedicated to the peripheral that was created.

```
/******************************************************************/

/* Definitions for driver ADAMS_PERIHPERAL */
#define XPAR_ADAMS_PERIHPERAL_NUM_INSTANCES 1

/* Definitions for peripheral ADAMS_PERIHPERAL_0 */
#define XPAR_ADAMS_PERIHPERAL_0_DEVICE_ID 0
#define XPAR_ADAMS_PERIHPERAL_0_S00_AXI_BASEADDR 0x43C00000
#define XPAR_ADAMS_PERIHPERAL_0_S00_AXI_HIGHADDR 0x43C00FFF


/******************************************************************/
```

The next stage is to open the System.MSS file and customise the BSP to use the drivers generated in the peripheral creation process and not the generic drivers.

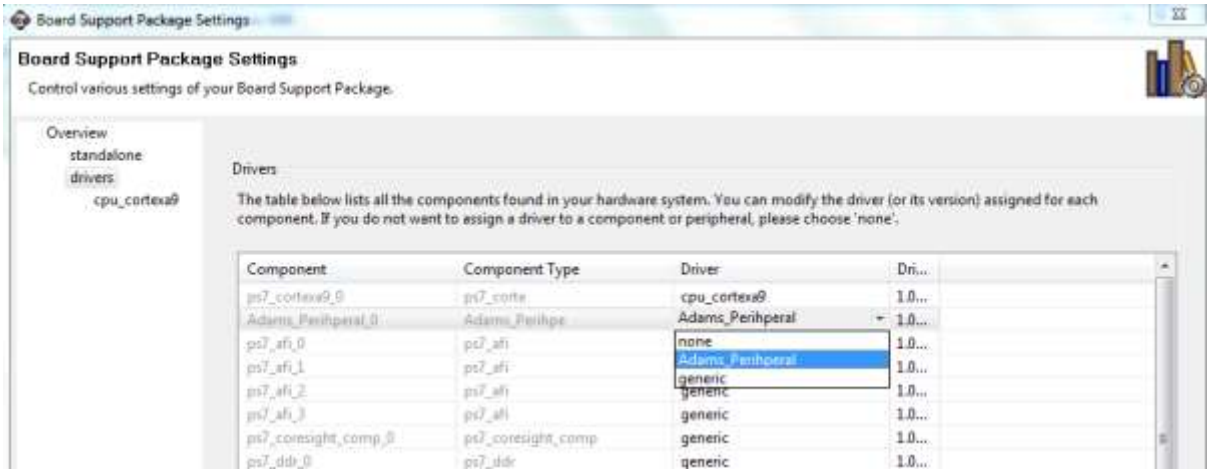


Figure 91 Selecting the Peripheral Driver

Having selected this and re building the project will ensure the driver files are loaded into the BSP, this is very helpful as these also include a simple self-test programme that can be used to test the interface is correct before we start using it for advanced things. Using this can be of great benefit as it demonstrates we have correctly instantiated the Hardware in Vivado.

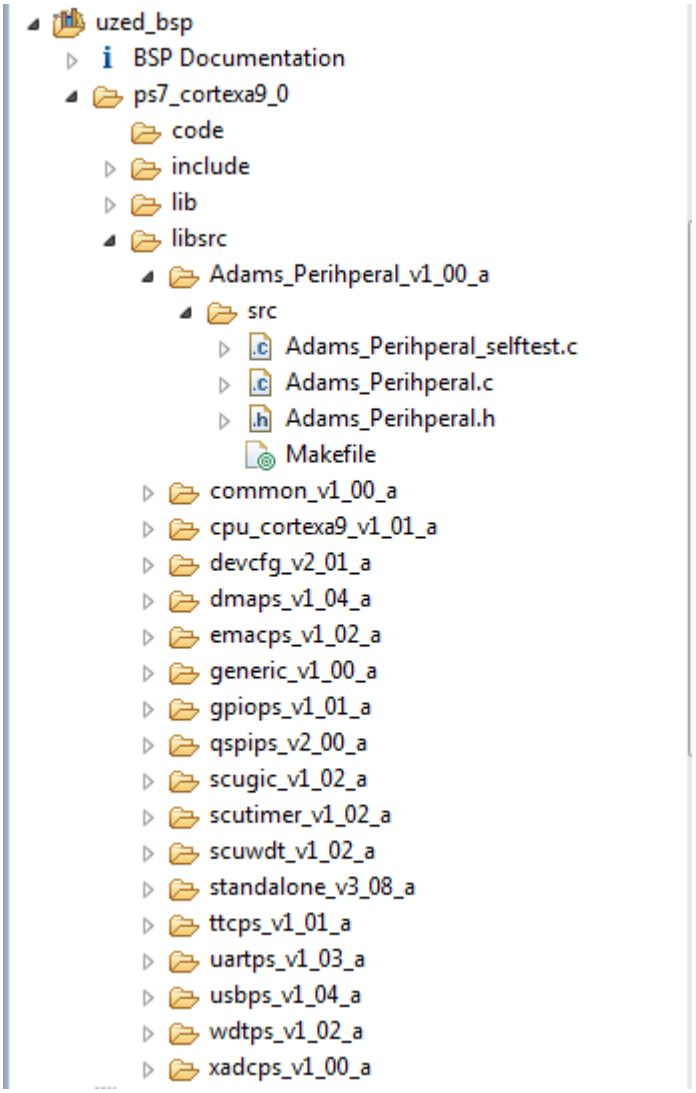


Figure 92 Inclusion of the peripheral drivers

Under the BSP included within the libsrc you will see a number of files for the peripheral, these files enable the engineer to read and write to the peripheral just as you do with those such as the XADC, GPIO which we have been using previously in other blog instalments.

The file adams_perihperal.h will contain for this simple example three functions we can use to drive this. (bonus points if you can spot my spelling mistake)

```
ADAMS_PERIHPERAL_mReadReg(BaseAddress, RegOffset)

ADAMS_PERIHPERAL_mWriteReg(BaseAddress, RegOffset, Data)

XStatus ADAMS_PERIHPERAL_Reg_SelfTest(void * baseaddr_p);
```

In actual fact with the exception of the self-test function the read and write functions are mapped to generic functions Xil_In32 and Xil_Out32 which are defined within Xil_io.h. However using the created functions enables more readable code as the peripheral being addressed is very clear.

For this example as we only have four registers within the peripheral, we will just use the self-test which will write and read to all of the registers and report a pass or fail. This gives us confidence we have got the hardware and software environments correct and we can do more advanced functions once we define them in the peripheral module.

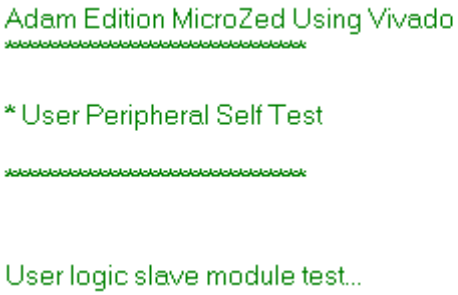


Figure 93 Results from t the MicroZed

In the next blog we will be looking at how we can add in our own VHDL to add some functionality to the peripheral such that we can offload functions from the Processing System.

PS / PL Part Four

Recapping on the Ps/Pl interface to date we have examined the interfaces between the Ps and Pl side of the device, created a simple peripheral using Vivado and then used SDK to communicate with it and performed a self-test. However, the peripheral created contained no functionality beyond the four registers which we could read and write to obviously in reality engineers will want to add functionality to the peripheral.

To achieve this we will be again using Vivado, the first thing to do is open the Vivado project and the block diagram which contains the peripheral which I created. Right clicking on the peripheral and selecting the “Edit in IP Packager option”.

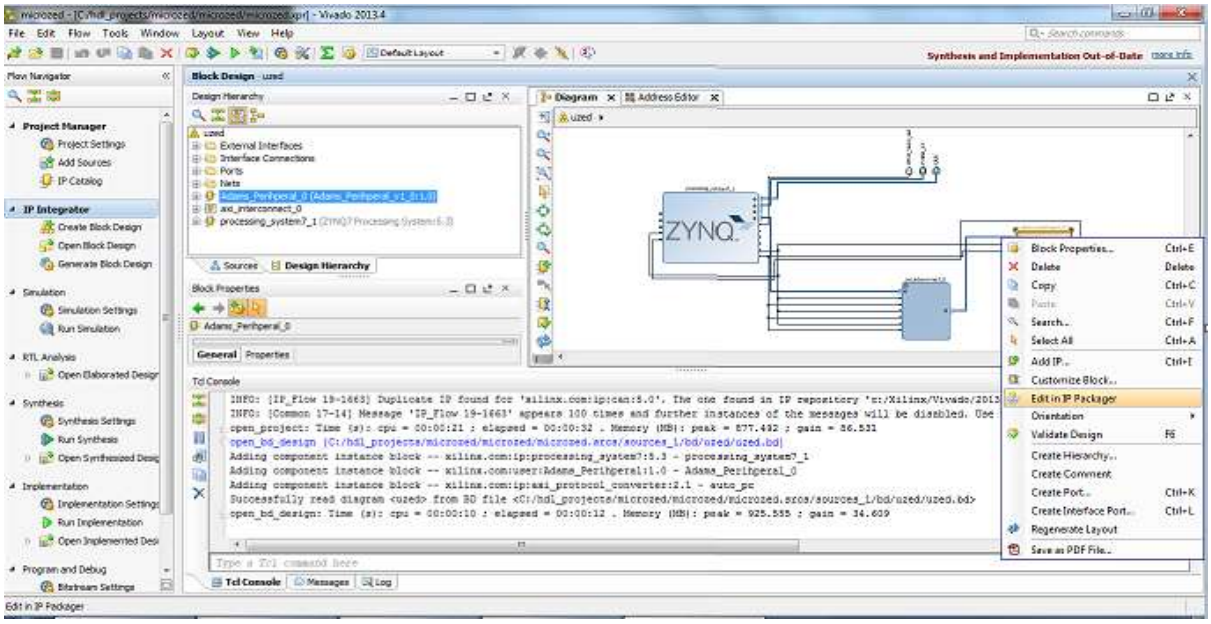


Figure 94 Editing the IP

This will open the IP Packager view in a new window which enables you to edit and update the peripheral this looks nearly to the standard project flow with the exception of the Package IP peripheral window. Beneath the design sources window you will find two files which were created during the peripheral creation process.

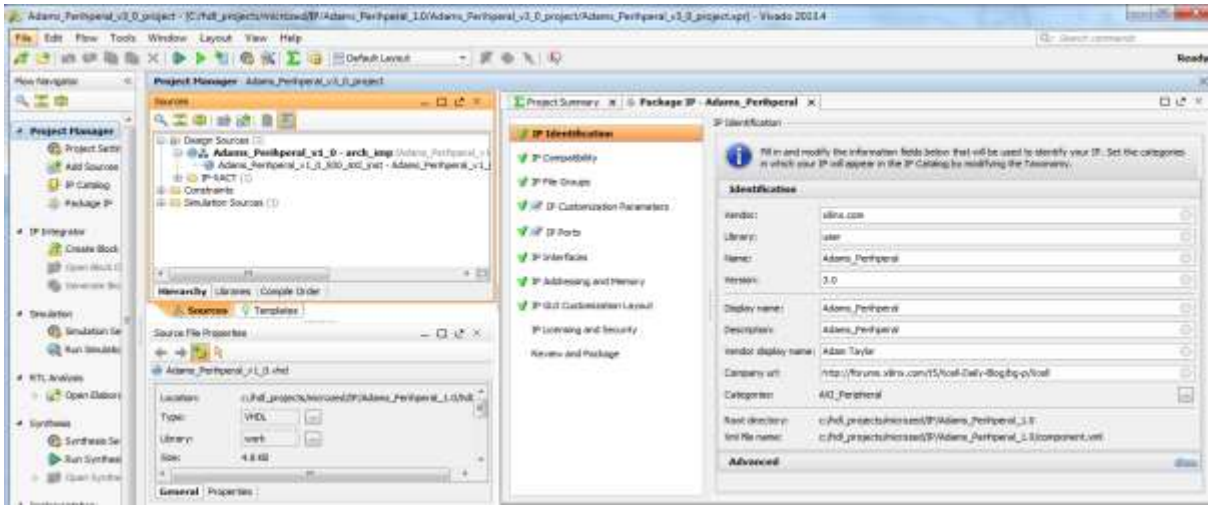


Figure 95 IP open for editing

These files are named:-

Adams_Peripheral_v1_0.vhd – Top Level architectural files, here you would define user IO which leave the module.

Adams_Peripheral_V1_0_S00_AXI.vhd – RTL file which contains the functional AXI interface including the four registers initially created.

Both of these files include comments as to where the user code is to be inserted

```
entity Adams_Perihperal_v1_0 is
    generic (
        -- Users to add parameters here

        -- User parameters ends
        -- Do not modify the parameters beyond this line


        -- Parameters of Axi Slave Bus Interface S00_AXI
        C_S00_AXI_DATA_WIDTH    : integer   := 32;
        C_S00_AXI_ADDR_WIDTH    : integer   := 4
    );
    port (
        -- Users to add ports here
        interrupt : out std_logic;
        -- User ports ends
        -- Do not modify the ports beyond this line
```

For this example I am going to introduce use the first register to define if the contents of registers 2 and 3 are added subtracted or multiplied together. What controls this is setting specific bits within register one being set, the result of the operation will be stored within the fourth register. To ensure the fourth register is not corrupted it will be made read only for the processor. The peripheral will also be capable of generating an interrupt if enabled by the control register.

The first step in this is within the functional file to declare in the entity the four registers, the first three as outputs and the final one as an input. (If we wanted to we could implement this function here however I am doing it at the top level to demonstrate what will be required on more complex functions). I also edited this file to prevent the fourth register from being written to by the processor making it read only.

Within the top level file I created an interrupt output and added the simple functional code within the architecture to perform the operations we desire.

```
PROCESS(s00_axi_aclk)
BEGIN
    IF rising_edge(s00_axi_aclk) THEN
        IF reg0(1 DOWNTO 0) = add THEN
           result(31 DOWNTO 16) <= (OTHERS =>'0');
           result(15 DOWNTO 0)  <= unsigned(reg1(15 DOWNTO 0)) + unsigned(reg2(15 DOWNTO 0));
        ELSIF reg0(1 DOWNTO 0) = sub THEN
           result(31 DOWNTO 16) <= (OTHERS =>'0');
           result(15 DOWNTO 0)  <= unsigned(reg1(15 DOWNTO 0)) - unsigned(reg2(15 DOWNTO 0));
        ELSIF reg0(1 DOWNTO 0) = mul THEN
            result <= unsigned(reg1(15 DOWNTO 0)) * unsigned(reg2(15 DOWNTO 0));
        ELSE
            null;
        END IF;
     END IF;
END PROCESS;
```

Having added in all user VHDL I synthesised the project to ensure I had not made any errors before I packaged IP and returned to my project within Vivado. However, before I packaged it on the packager page I increased the version to reflect the changes to the code. Clicking on re-package will run the packager and close the project returning you to the original Vivado project.

Once back within the project which uses the peripheral running the IP status report (under tools -> Reports -> Report IP status) will show the updated version being used within the design.

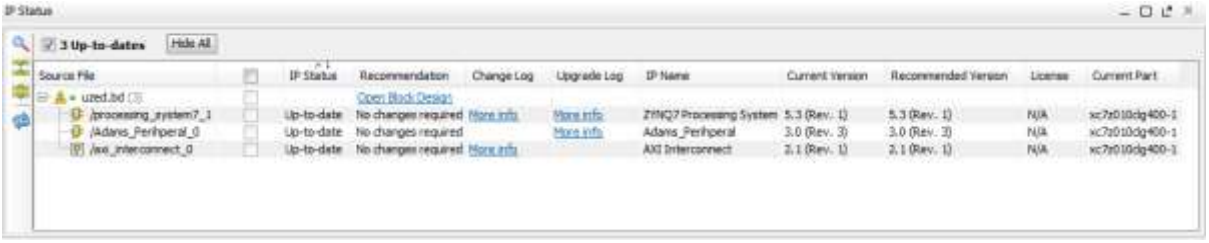


Figure 96 Updated peripheral

The project needs then to be re built prior to exporting to SDK, within SDK the same functions as before can be used to write and read into the peripheral, however this time the self-test should fail as the final register cannot be written during the test.

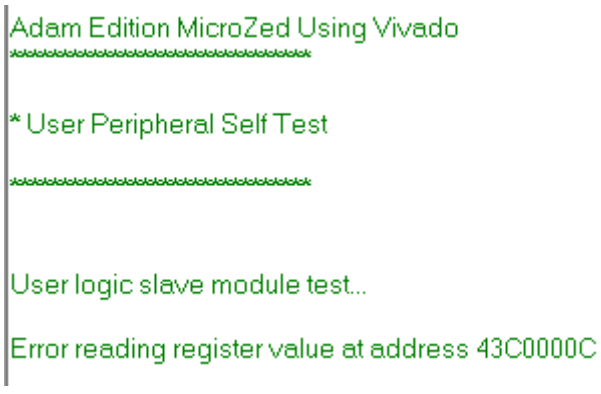


Figure 97 Results from the MicroZed of the Example

The first test is to add together the two contents stored within register 2 and 3, command 1 in register 0

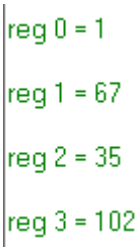


Figure 98 Results of the addition

The second test was to multiply together the contents of register 2 and 3, command 2 in register 0

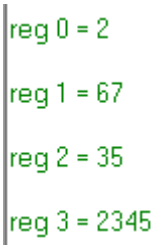


Figure 99 Results of the multiply

The final test was to subtract register 3 from register 2, command 3 in register 0

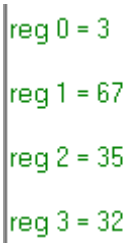


Figure 100 Results of the subtract

All of these tests have used a polled approach as the simple add subtract and multiply is achieved within one clock cycle however more complicated functions require the use of an interrupt which we will look at next time as we look at a more complex function and peripheral.

PS / PL Part Five

When we last looked that the Ps/PL interface, I had created a very simple peripheral which used a DSP48E1 to perform multiplication, addition or subtraction depending upon a control word in the first register. However, suppose that we want to perform a more complex calculation within the Zynq for instance if the device is used within an industrial control system. Typically these systems will have a number of analogue inputs (via ADC's) which come from sensors such as thermistors, thermocouples, pressure transducers, Platinum Resistance Thermometers (PRT) and so on.

Many times these require a transfer function to convert the raw value from the ADC to one which can be used in further processing. A good example of this is with the Zynq XADC which contains a number of functions / macros within XADCPS.h to convert the raw XADC values into voltage or temperature. However, these are pretty simple conversions the more complex they become the more Zynq processing time is required. The speed of calculation can be speeded up considerably if the Programmable logic side of the Zynq is used to perform these calculations, as the processor is also freed up to carry on with other software tasks you get a significant saving.

The more complex a transfer function the more processor time will be taken to calculate the result, using the example of converting an atmospheric pressure measured in milibars to an altitude in meters (see Xcell Journal Issue 80 – the basics of FPGA Mathematics) we can demonstrate this impact. The transfer function below will give the altitude in meters for a pressure between 0 and 10 millibars

$$-0.0088x2 + 1.7673x + 131.29$$

Implementing this within the processing system side of the Zynq is very simple using the following line of code below, where result is a floating number, a, b, and c are the constants defined above in the transfer function and i is the input value.

```
result = ((float)a*(i*i)) + ((float)b*(i)) + (float)c;
```

For this example I shall use the above code nested within a "for" loop to simulate steps in the input value, and output the result over the STDOUT such that the result can be seen. As I want to calculate the time it takes to perform this calculation I shall be using the private timer, to determine this, starting and stopping the timer before and after as below

```
for(i=0.0; i<10.0; i = i +0.1 ){
 XScuTimer_LoadTimer(&Timer, TIMER_LOAD_VALUE);
 timer_start = XScuTimer_GetCounterValue(&Timer);
 XScuTimer_Start(&Timer);
 result = ((float)a*(i*i)) + ((float)b*(i)) + (float)c;
 XScuTimer_Stop(&Timer);
 timer_value = XScuTimer_GetCounterValue(&Timer);
 printf("%f,%f,%lu,%lu, \n\r",i,result,timer_start, timer_value);
}
```

While this may not provide the most accurate timing reference it should be sufficient to demonstrate the principal we will be investigating over the next few blogs. Running the above code on the MicroZed I obtained the results below in the terminal window. I extracted this into excel as a comma separated variable file due to how I had formatted the output over STDOUT

```
Adam Edition MicroZed Using Vivado
Prescale = 0
 Input, Result, TMR Start, TMR Stop
0.000000,131.289993,4294967295,4294967209,
0.100000,131.466629,4294967295,4294967263,
0.200000,131.643097,4294967295,4294967270,
0.300000,131.819397,4294967295,4294967268,
0.400000,131.995499,4294967295,4294967269,
0.500000,132.171448,4294967295,4294967268,
0.600000,132.347198,4294967295,4294967268,
0.700000,132.522797,4294967295,4294967266,
0.800000,132.698196,4294967295,4294967270,
0.900000,132.873428,4294967295,4294967268,
```

Figure 101 Results of the example, showing input, result, Calculation Start Time and Calculation Stop Time

Performing some simple analysis on this shows that on average it takes 25 CPU_3x2x clock cycles to calculate the result which is not bad, with a 666 MHz processor clock this takes 76 ns. I am sure many people will have spotted a ADC output will not be floating point number but it will instead be a fixed point number, re-writing the code to function based of integer mathematics resulted in a very similar average number of clock cycles. However I thought for this example floating point numbers would be easier without having to explain the principals behind the fixed point number systems.

Having established a bench mark for how long it takes the PS side of the Zynq to perform a medium complexity transfer function we can look next time at just how fast we can do this when we off load to the PL side of the device.

PS / PL Part Six

In the last instalment of this blog (which has now been going weekly for six months now) we implemented a transfer function within the PS side of the device and crudely measured its calculation time. In this blog we will start looking at what we need to understand and do to add VHDL (or Verilog if you so desire) to the peripheral we created previously to implement the transfer function. To do this we will need to use the fixed point number system therefore in this blog we shall look at how fixed point maths works and how we can implement it correctly within our FPGA.

There are two methods of representing numbers within a design, fixed or floating-point number systems that we can use. Fixed-point representation maintains the decimal point within a fixed position allowing for straight forward arithmetic operations. The major drawback of fixed-point representation is that to represent larger numbers or to achieve a more accurate result with fractional numbers, a larger number of bits are required. A fixed point number consists of two parts called the integer and fractional parts. Floating point representation allows the decimal point to float to different places within the number depending upon the magnitude, floating point numbers are divided into two parts the exponent and the mantissa.

Although FPGA's can support both fixed and floating point numbers, most applications utilise fixed point number systems as they are simpler to implement than floating point ones.

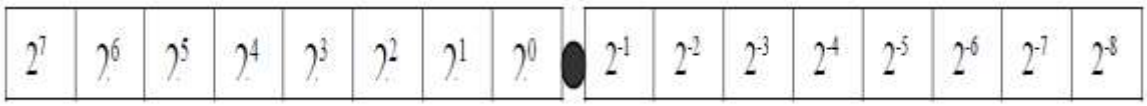


Figure 102 Fixed point number system

The above fixed-point number is capable of representing an unsigned number of between 0.0 and 255.9906375 or a signed number of between –128.9906375 and 127.9906375 using twos complement representation. Within a design we have the choice to use either unsigned or signed numbers typically this will be constrained by the algorithm being implemented. Unsigned numbers are capable of representing a range of 0 to 2n – 1, and always represent positive numbers. Signed numbers use the twos complement number system to represent both positive and negative numbers. The twos complement number system allows subtraction of one number form another by performing an addition of the two numbers. The range a twos complement number can represent is given by

$$-(2n-1) \text{ to } +(2n-1 - 1)$$

The normal way of representing the split between integer, fractional bits within a fixed-point number is x,y where x represents the number of integer bits and y the number of fractional bits. For example 8,8 represents 8 integer bits and 8 fractional bits while 16,0 represents 16 integer and 0 fractional. In many cases the correct choice of the number of integer and fractional bits required will be undertaken at design time normally following conversion from a floating point algorithm. Thanks to the flexibility of FPGA's we can represent a fixed-point number of any bit length; the number of integer bits required depends upon the maximum integer value the number is required to store, while the number of fractional bits will depend upon the accuracy of the final result. To determine the number of integer bits required we can use the following equation

$$\text{Integer Bits Required} = Ceil\left(\frac{LOG_{10}\text{Integer\_Maximum}}{LOG_{10}2}\right)$$

For example the number of integer bits required to represent a value between 0.0 and 423.0 is given by

$$9 = Ceil\left(\frac{\text{LOG}_{10}423}{\text{LOG}_{10}2}\right)$$

Meaning we would need 9 integer bits, allowing a range of 0 to 511 to be represented.

To perform addition, subtraction or division the decimal points of both numbers must be aligned. That is a x,8 number can only added to, subtracted from or divided by a number which is also in a x,8 representation. To perform arithmetic operations on numbers of different x,y format we must first ensure the decimal points are aligned. Note it is not strictly necessary to align the decimal points for division however, it needs careful consideration to ensure the result is scaled correctly in this case and not negative.

When multiplying two numbers together the decimal points do not need to be aligned as the multiplication will provide a result which is X1 + X2, Y1 + Y2 wide. Multiplying two numbers, which are formatted 14,2 and 10,6, will produce a result, which is formatted 24 integer bits and 8 fractional bits

For division by fixed constants we can of course simplify the design and calculate the reciprocal and then multiply by that as a constant to implement it more efficiently.

Having understood the above in the next blog we can progress to looking at the implementation within the FPGA using a fixed point number system.

PS / PL Part Seven

Having looked at how we can implement fixed point mathematics within the PL side of the Zync in this blog we will focus upon implementing this within the system and the rather surprising results of doing so.

Before we get to cutting code the first thing to do is to determine the scaling factors (location of the decimal point). As the input signal will be between 0 and 10 we can use to use 4 decimal bits and 12 fractional bits if we using a 16 bit input vector.

$$-0.0088x2 + 1.7673x + 131.29$$

The equation we are implementing (see above) has three constants A, B and C which require scaling to implement. The beauty of doing this in a FPGA means we can use different scaling for each to optimise the performance as in the table below

Constant	Unscaled	Decimal Point Location	Resultant Format	Scaled
A	-0.0088	16	0,16	-577
B	1.7673	15	1,15	57910
C	131.29	8	8,8	131

As we implement the above we will need to consider the expansion of the resultant vectors, which for the terms Ax^2 and Bx are defined below.

	x Format	x * x Format	A*x*x Format
Ax^2	4,12	8,24	8,40

	x Format	B * x Format
Bx	4,12	5, 27

To perform the final addition with constant C we need to have the decimal point aligned therefore, we need to divide the results and Ax^2 and Bx by a power of two to align the decimal points with C. the result will also be formatted in this value which is 8,8.

	A Format	Bx Format	Cx^2 Format
	8,8	5,27	8,40
Divider to align decimal points for addition		2^19	2^32

Having calculated the above we are ready to implement the design within the Vivado peripheral we created. The first step in this is to open up the block diagram view within Vivado, right click on the peripheral and select “Edit in IP Packager”. Once this open within the top level file we can easily implement a simple process which will perform the calculation over a number of clock cycles (in this case five clocks, although you could optimise this further).

```
PROCESS(s00_axi_aclk)
BEGIN
    IF rising_edge(s00_axi_aclk) THEN
      squared <= signed( '0'& reg1(15 DOWNTO 0)) * signed('0'& reg1(15 DOWNTO 0));
      cx2 <= (squared * c);
      bx <= (signed('0'& reg1(15 DOWNTO 0))* b);
      res_int <= a + cx2(48 DOWNTO 32) + ("000"& bx(32 DOWNTO 19));
      result(15 downto 0) <= std_logic_vector(res_int(res_int'high -1 DOWNTO 0));
    END IF;
END PROCESS;
```

Having completed this we can re-package and rebuild the project within Vivado (remember to update the version number), before exporting the updated hardware to SDK.

Once we are within SDK we can use the same approach as before with the exception we are using a fixed point number system now in place of the floating point earlier

```
  for(i=0; i<2560; i = i+25 ){
       XScuTimer_LoadTimer(&Timer, TIMER_LOAD_VALUE);
       timer_start = XScuTimer_GetCounterValue(&Timer);
       XScuTimer_Start(&Timer);
       ADAMS_PERIHPERAL_mWriteReg(Adam_Low, 4, i);
       result = ADAMS_PERIHPERAL_mReadReg(Adam_Low, 12);
       XScuTimer_Stop(&Timer);
       timer_value = XScuTimer_GetCounterValue(&Timer);
       printf("%d,%lu,%lu,%lu, \n\r",i,result,timer_start, timer_value);
     }
```

When the above code was built and run this on the Microzed we get the following result output over the serial link, the result of 33610 equals 131.289 when divided by 2^8 which is correct and in line with the floating point calculation (see part 5 of this blog).

```
Adam Edition MicroZed Using Vivado
Prescale = 0
Input, Result, TMR Start, TMR Stop
0,33610,4294967295,4294967155,
```

Figure 103 Results of the Complex Equation Implementation

However the big difference is the time it takes to perform the calculations which is 140 clocks or 420ns which is much more significant than using the PS side of the device to calculate this. However, when using the PL side we must take into account the bus latency over the AXI bus and the AXI bus frequency which in this application is 142.8MHz (the requested was 150 MHz). Which accounts for the longer than expected time however, all is not lost as offloading to the PL is not intended to be used in this manner demonstrated. Instead a block of inputs requiring calculation would be passed to the PL side at once using DMA as I explained in part 1 of the Ps/Pl interfacing. Having established why the DMA is so important it now enables me to explore how we use this in the next blog.

DMA Part One

In my last blog we had just arrived at the point that the benefit of Direct Memory Access (DMA) became obvious, although I had previously alluded to the benefit of using DMA coupled with the AXI interfaces in part 21.

Having reached this point it leaves us with the question which man has long pondered what exactly is DMA? At its most basic level DMA allows transfers of data to or from memory without the intervention of the processor once it has set up the transfer, of course this can increase significantly the performance of the system depending upon the approach undertaken.

Before we look at the DMA as provided by the Zynq in more detail, I would first like to explain a few generic principals of a DMA controllers. Typically these devices operate in one of three modes as explained below

- Burst Mode - This transfers the entire data block in one continuous operation, in many applications this will result in the processor not being able to access the system bus to continue operation.
- Cycle Stealing – This interleaves transfer of DMA bytes with system bus access as required by the processor
- Transparent mode – The most efficient mode in that data is only transferred when the processor is performing tasks which do not need access to the system bus.

One very useful feature of DMA Controllers is the ability to support scatter gathering, this enables multiple inputs to be transferred to a single source, or a single source to supply multiple output sources (buffers).

The Processing System on the Zynq has a DMA Controller (DMAC) which is connected to the AXI central interconnect and uses the AXI bus to perform transfers. The DMAC uses 64 bit AXI transfers between system memories and the programmable logic. To support multiple transfers at once the DMAC provides eight channels which allows the DMAC to execute eight threads concurrently with flow control achieved via the AXI interconnect.

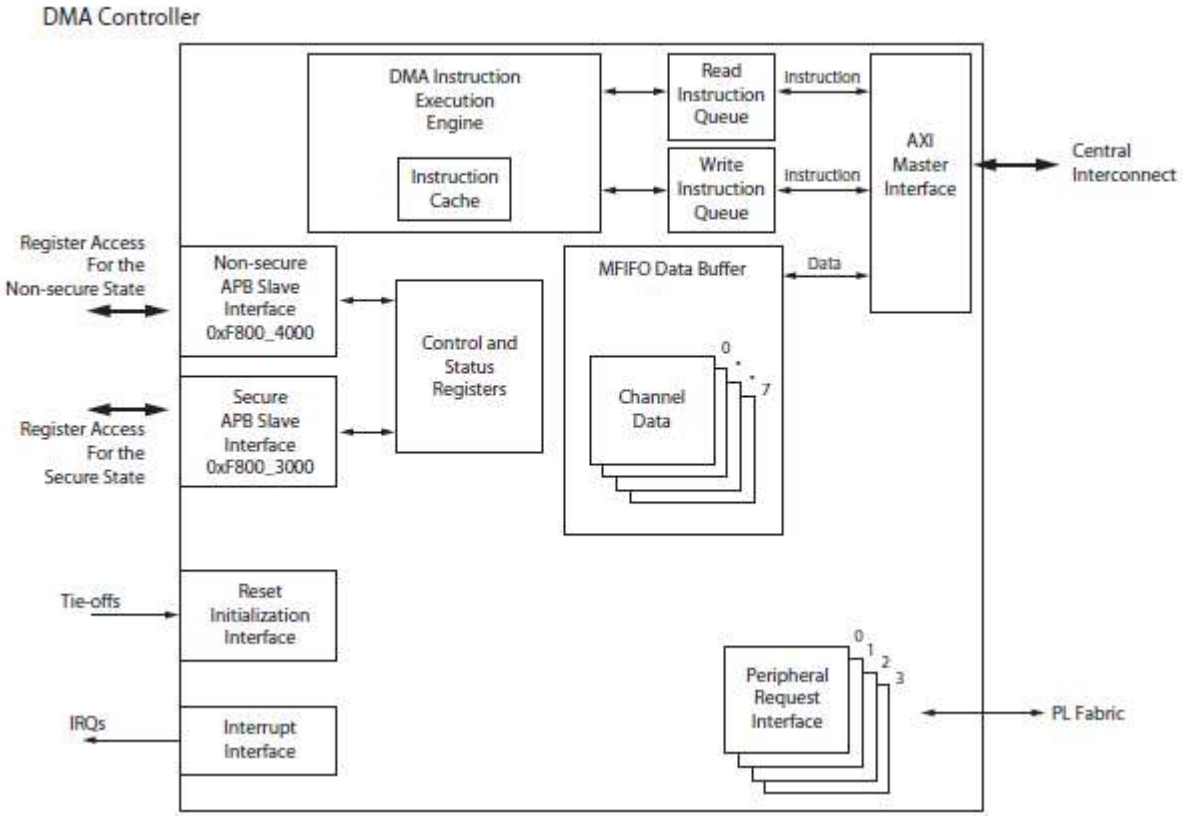


Figure 104 DMA Controller Architecture

While the DMAC allows transfer to or from system memories and PL (including its peripherals) it does not support DMA for the PS Input Output Peripherals as these have no flow control signals. However other DMA Controllers are provided within the IOP to support high data rate transfers to or from the IOP and system memory.

- GigE Controller
- SDIO Controller
- USB Controller
- DEV C Controller

The Zynq also provides support for secure register access if the device is utilising the ARM TrustZone.

Xilinx rather helpfully provide a simple driver file (xdmaps.h) which we can use within the standalone BSP to configure and initiate DMA transfers. In my next blog we will look at how we can create a simple DMA transfer using this file.

DMA Part Two

Having introduced Direct Memory Access in my previous blog this blog will focus upon creating a very simple example which we demonstrates how to set up and use DMA. To demonstrate this I will transfer one memory location to another using one channel of the DMA controller.

The starting point for this is one which we have commonly used in this blog series, the inclusion of header files generated as part of the BSP which provide macros and functions we can use to drive the DMA for this example we will need to include

```
#include "xscugic.h"
#include "xdmaps.h"
#include "xil_exception.h"
```

Xscugic.h and xil_exceptions.h enable us to use the interrupt controller while xdmaps.h is the file which enables us to configure and use the DMA.

Using parameters provided with xparamters.h we can define the device identifications for the DMA and interrupt controller, the interrupts which will be used and the length of the data we will be transferring.

```
#define DMA_DEVICE_ID               XPAR_XDMAPS_1_DEVICE_ID
#define INTC_DEVICE_ID              XPAR_SCUGIC_SINGLE_DEVICE_ID
#define DMA_FAULT_INTR              XPAR_XDMAPS_0_FAULT_INTR
#define DMA_DONE_INTR_0             XPAR_XDMAPS_0_DONE_INTR_0
#define DMA_LENGTH      1024
```

The next stage of the development is to write the three functions which will configure the DMA, configure the interrupt controller and act as the interrupt service routine at the completion of the DMA transfer.

Within the DM configuration function we must first create a DMA command using the command structure provided by xdmaps.h a DMA command consists of configuring the channel control, block descriptor, a user defined program, a pointer to the generated program and the result of the transfer. As this is a simple example we will not be requiring all of these however will be configuring the DMA controller as below.

```
DmaCmd.ChanCtrl.SrcBurstSize = 4;
DmaCmd.ChanCtrl.SrcBurstLen = 4;
DmaCmd.ChanCtrl.SrcInc = 1;
DmaCmd.ChanCtrl.DstBurstSize = 4;
DmaCmd.ChanCtrl.DstBurstLen = 4;
DmaCmd.ChanCtrl.DstInc = 1;
DmaCmd.BD.SrcAddr = (u32) Src;
DmaCmd.BD.DstAddr = (u32) Dst;
DmaCmd.BD.Length = DMA_LENGTH * sizeof(int);
```

The next step is to initialise and configure the DMA controller before running the interrupt set up function to connect the DMA interrupts to the interrupt controller

```
DmaCfg = XDmaPs_LookupConfig(DeviceId);
XDmaPs_CfgInitialize(DmaInst,DmaCfg,DmaCfg->BaseAddress);
SetupInterrupt(&GicInstance, DmaInst);
```

Following this the source memory location is seeded with data and the destination location is cleared before we connect the done handler and start the transfer, to track progress we also make one call to the DMA progress function

```
DmaCfg = XDmaPs_LookupConfig(DeviceId);
XDmaPs_CfgInitialize(DmaInst,DmaCfg,DmaCfg->BaseAddress);
SetupInterrupt(&GicInstance, DmaInst);
XDmaPs_Print_DmaProg(&DmaCmd);
```

When the attached source code file was run on the MicroZed the following results were shown on the rs232 output I was using to report the status.

```
Adam Edition MicroZed Using Vivado
Generated DMA program (27)
DMA passed
:
```

Figure 105 Results of the DMA Example

Having looked at the DMA control and basic example, in my next blog I will be moving on to look at the MicroZed carrier card and how we can use this with the MicroZed as a system of modules.

System of Modules Example Part One

I recently took delivery of a MicroZed IO Carrier card, which complements the MicroZed System of Modules approach by enabling breaking out the IO which are contained on the two micro connectors on the MicroZed.

Figure 106 The compact size of the MicroZed

The concept behind the MicroZed is that it allows you to create a system of modules where it plugs on an application specific carrier card. When you stop and think about it this approach has a number of benefits.

- Reduces Non-Recurring Effort in the development of the product allowing focus to remain upon the areas of added value. e.g. application specific carrier card
- Use of the MicroZed as a core, provides a rapid hardware platform for the development team to begin testing the SoC design upon.
- With no need to design the SoC and perform the subsequent verification this allows for a reduction in the time to market.
- Allows the hardware design effort to be targeted at the application specific card carrier, this allows for a reduced risk profile.

The MicroZed system of module concept is very flexible as the core MicroZed breaks out just not the IO from the programmable logic side of the Zynq but also from the PS side. The really interesting aspect is the IO bank voltages are also accessible and driven via the micro connectors. This enables the carrier card to set the Bank voltages for bank 34, 35 and 13 if you are using the Z7020 version of the MicroZed. This enables the carrier card to supply the IO voltage that is required for the application at hand.

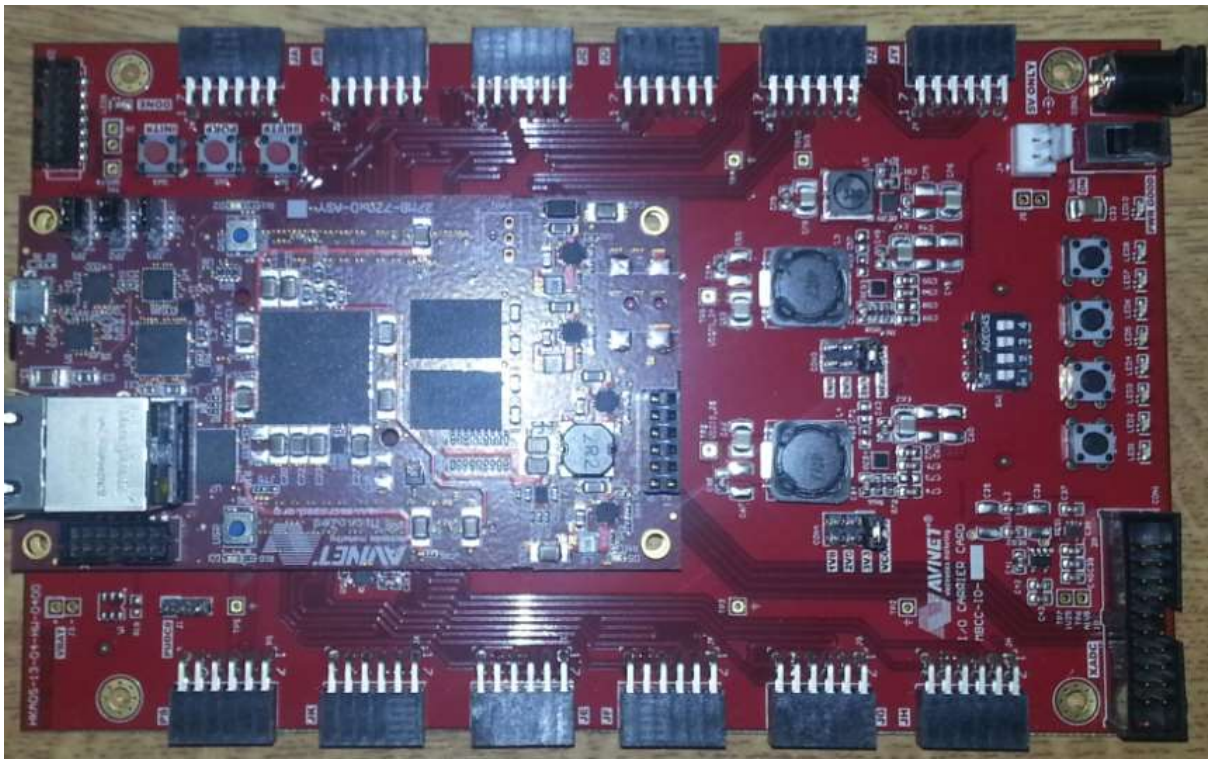


Figure 107 MicroZed and Carrier Card

The MicroZed is also designed such that the five volts power supply can be supplied from either the USB or via the carrier card this is achieved by diode OR'ing the power supply from USB and the Micro header on the MicroZed Core.

There are three off the shelf carrier cards which can be purchased off the shelf from zedboard.org

- The MicroZed IO Carrier card as shown above and whose use we will be exploring over the next few blogs.
- The Breakout Carrier card which provides a bread boarding area for developing prototypes and one off solutions.
- The FMC carrier card, which allows the MicroZed to be connected to a low pin count FMC interface.

The Carrier Card I will be experimenting with over the next few blogs comes with 12 PMOD interfaces, LEDS, Switches and a 100 MHZ oscillator for use clocking the PL side of the Zynq. I would like to use the IO carrier card to test out a few concepts I have been demonstrating over the last few blogs hopefully my orders will arrive before I sit down to write the next blog.

System of Modules Example Part Two

Since I started writing this now epic blog over 6 months ago I have, I hope explained a number of detailed aspects of both the PS and PL along with providing nice simple easy to use examples. Having introduced the IO carrier card we have reached a point where we can pull together a number of the aspects we have looked at previously so we can see how they all come together easily.

This example will look at driving an Adafruit neo pixel array from the PL under the control of the PS this will pull together the following aspects

1. Developing a PL peripheral to drive the neo pixels
2. Communication between the PS to the PL
3. Use the IO Carrier Card and the PMOD expansion ports
4. Developing a PS Application which uses

I do not expect that we will wrap this all up in one blog, it will take a few however do not worry I have it already implemented and working so there should be no issues along the way, but I would like to explain it all in detail.

At this point a few people might be asking what a neo pixel is, these are individual addressable LED's which contain Red Green and Blue LEDS under the control of a digital controller.

This makes for a very simple interface which consists of three wires Power, Return and Din. Each of the LED (Red, Green or Blue) pixels requires eight bits providing 256 levels brightness or when combined with the other pixels the possibility for 16777216 colours.

The range of colours combined with the simple interface makes these very interesting for a number of applications.

What makes these even better is that pixels can be daisy chained together as each pixel has a Dout output, out of which it will pass the data it received on Din if another command word arrives immediately after the first. In this manner it is possible to build a large strip or other arrangement of neo pixels provided you can power them each of which can be individually addressed.

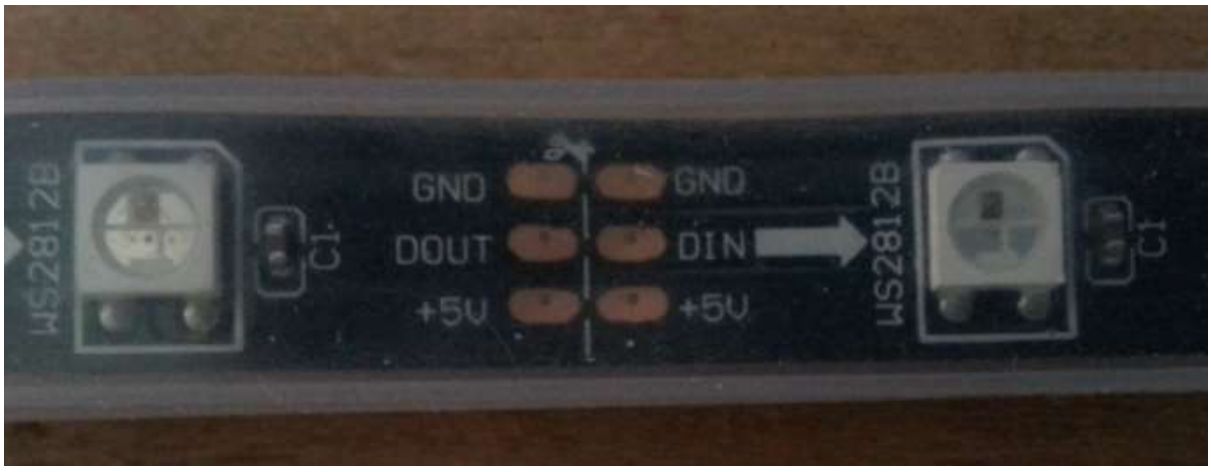


Figure 108 Two neo pixels arranged in a daisy chain

Neo pixels use the worldsemi WS2812 intelligent control LED integrated light sources (datasheet here http://www.adafruit.com/datasheets/WS2812.pdf). These devices are generally operated from

a 5 volt supply, however reading the datasheet it is possible to operate these from a 3.3v supply. The driving factor in this is the need for a voltage in the range of 3.2V to 3.4V to drive the blue LED.

Therefore it is possible to drive these using the 3v3 supply from the IO card carrier, I will explain more on this in a later blog.

To ensure I could connect the neo pixels I have to the IO Carrier card I needed a PMOD expansion module which breaks out the four signal and power / return connections on the PMOD interface to six screw terminals (https://www.digilentinc.com/Products/Detail.cfm?NavPath=2,401,539&Prod=PMOD-CON1)

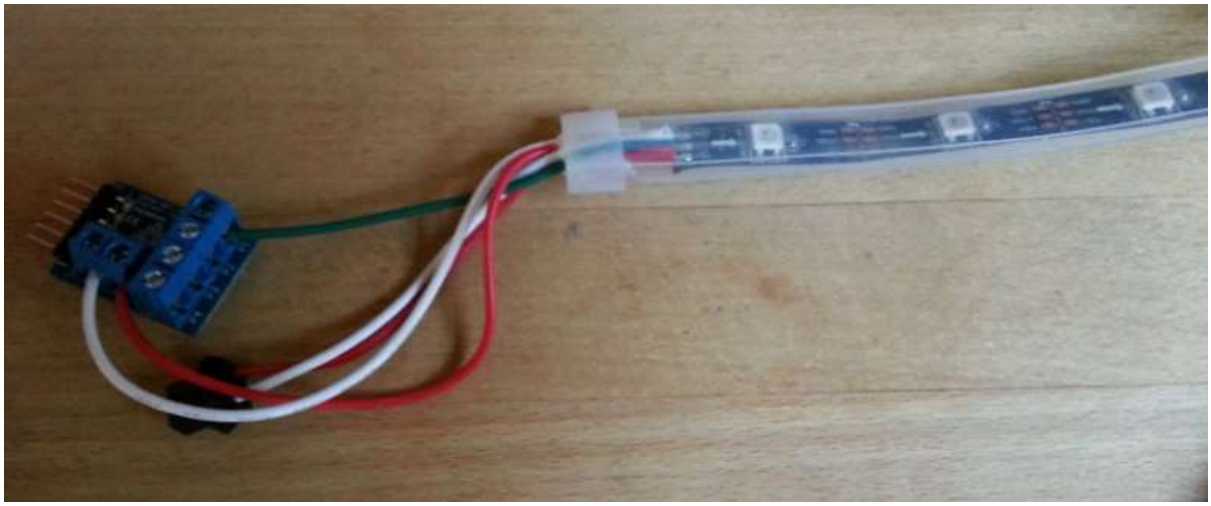


Figure 109 PMOD-Con1 connected neo pixel strip

As I have explained a little about what I will be using it is good engineering practice to define some requirements before we rush in and develop the solution therefore the key requirements for this application are

1. It shall be capable of driving a variable length of Neo Pixels
2. The number of neo pixels shall be variable and user updateable during operation
3. Each pixel shall be capable of being individually addressed
4. Each pixel shall be capable of being updated during operation

These requirements are pretty simple but will allow the finished application to be used in a number of ways.

In the next blog we will look at the architecture and approach to be implemented to achieve these requirements in the meantime the picture below shows the initial test of the solution with just one LED being controlled.

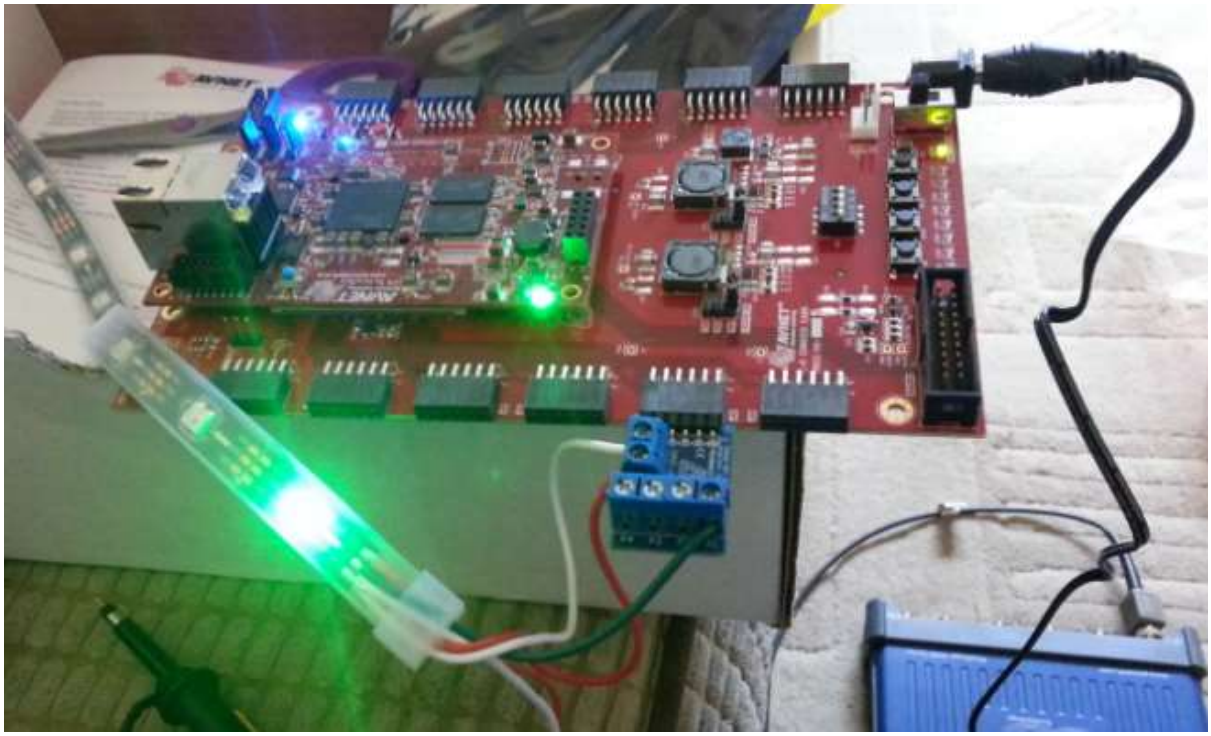


Figure 110 The first neo pixel controlled by the Zynq

System of Modules Example Part Three

In my last blog I introduced the neo pixel and outlined the cardinal points specification of the neo pixel drivers. In this blog I am going to address the hardware aspects of the solution before looking in my next blog at the architecture we will use within the PS and PL of the Zynq.

I have chosen for this example to drive a 1 meter length of neo pixels with a density of 30 pixels per meter. Reading the data sheet shows that while the Neo Pixel array is designed to work with a 5V supply. However, reading the datasheet further states the highest voltage required is the Blue LED which needs a voltage between 3.2 and 3.4 volts right in the range of a 3.3V supply voltage.

The IO Carrier card has two voltage regulators which can be used to supply banks 34 and 35 (and 13 if you are using the Z7020) on the Zynq. The output voltages of these regulators are selectable between 1.8V, 2.5V and 3.3V, of the two regulators bank 35 has a higher current rating 2.8 A compared to the 2.3 A for bank 34. As we will be driving a large number of LED's I will be using bank 35 to drive the neo pixel array.

Each neo pixel requires a maximum current of 20 mA however there are three LED's in each pixel hence each pixel requires 60 mA which when we have 30 pixels in a line requires a maximum current of 1.8 Amps from the 3.3V supply which is well within the range of both supplies but maximum de rating is supplied by using the higher rated regulator.

This means that the PMOD connector will be connected to one of PMOD connectors numbered JE, JF, JG or JH as these are connected to the bank 35 regulator. The figure below shows the pin out for a PMOD interface

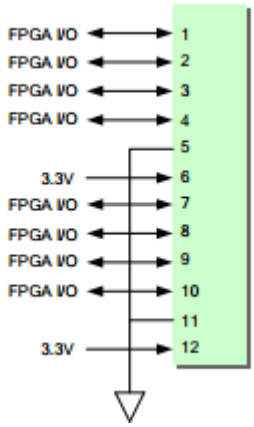


Figure 111 PMOD Pin Out

The PMOD Con1 I am using breaks out only the power, ground and four I/O which is sufficient for driving the neo pixel array as that requires, power, return and Din.

Din is how we programme a pixels required colour, each pixel requires a 24 bit word consisting of eight green, red and blue bits in that order as shown below.

Composition of 24bit data:

G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	B7	B6	B5	B4	B3	B2	B1	B0

Figure 112 24 bit Word Format

As there is no clock line each of the pixels is uses a self-clocking waveform which changes if the bit is currently a 1 or 0. This introduces a transition point between high and low at different points of the waveform as shown in the image below

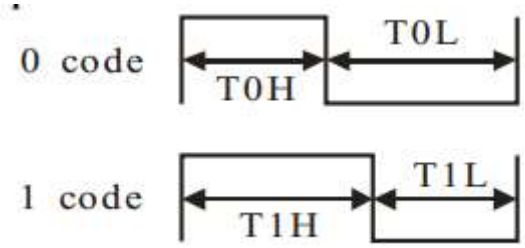


Figure 113 Neo Pixel bit timing

Where T0H = 0.35us, T0L = 0.8us and T1H = 0.7us and T1L = 0.6 us, this gives slightly different durations for a low bit (1.15us) as to a high bit (1.3us) we will be analysing this in much more detail in a future blog when we write the driver. If after receiving all 24 bits in the data stream the receiving neo pixel does not start receiving another word within 50 us it loads the word into its pixel and displays that value. If it does receive a word within the 50 us period it clocks out the previous word on it's Dout port allowing a large number of pixels to be addressed in a very simple manner.

Now I have explained a little more on the IO Carrier Card and how Neo Pixels work in the next blog I will introduce the architecture within the PS and PL to drive these pixels through the IO carrier card, in the meantime below is an example of the waveform output from the finished solution when it is outputting a High and Low bit in line with the neo pixel drive waveform.

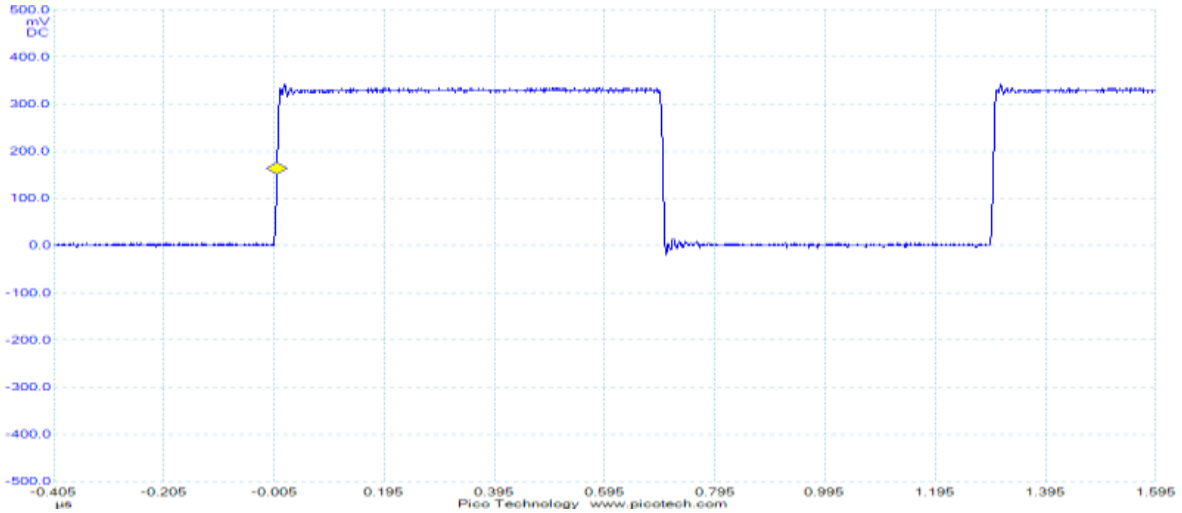


Figure 114 Neo Pixel High bit output

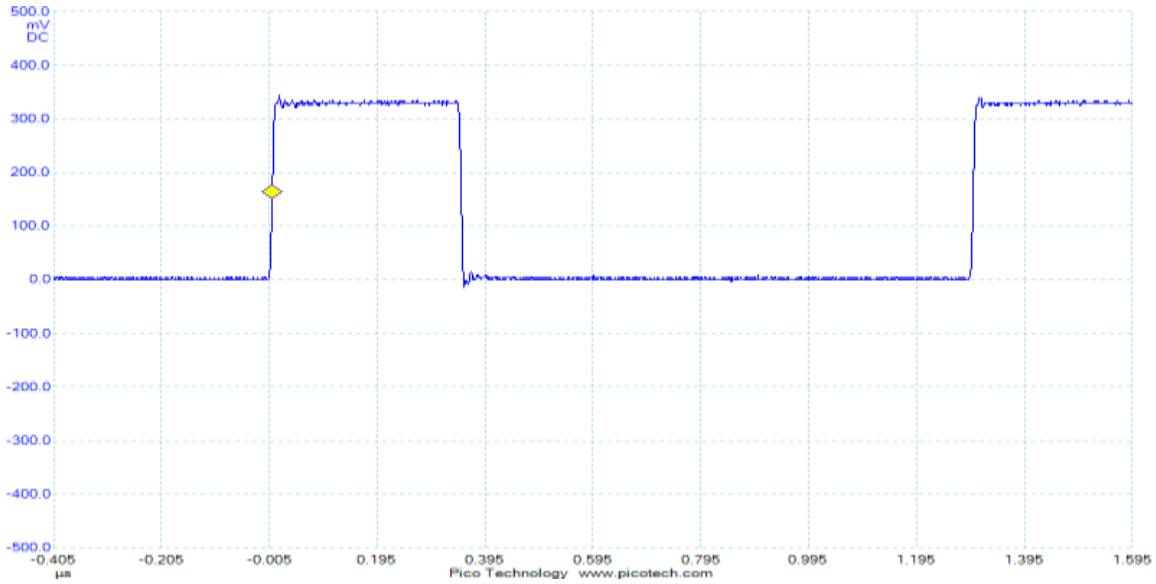


Figure 115 Neo Pixel Low bit output

System of Modules Example Part Four

Having explained more on the Neo Pixel drive waveforms, hardware required to interface with them and the power requirements in the previous instalment. It is time therefore to look at the architecture of the solution required within the PS and PL side of the Zynq.

Thinking back to the requirements, we want to be able to drive a variable number of Neo Pixels, with each pixel can be addressed individually and updated while the programme is executing.

Expanding a little more on this, in my mind when I first thought of this demonstration I pictured a GUI running on my lap top with a button for each LED in the array I want to drive from which when I click on a particular LED's button I can select a 24 bit colour. Once I have selected the colours for all of the LED's, these commands can be downloaded to the Zynq and used to update the Neo Pixels.

Figure 116 Jumping ahead a little the Neo Pixel GUI – to give some context

Obviously the PS side will handle all of the communication with the GUI programme, storing the pixel values and passing these to the custom Neo Pixel driver within the PL. To ensure reliable communication I will need to define a simple data transfer protocol for use over this interface.

This therefore means we need to pass data between the PS and PL sides of the device, the solution to this I have in mind is a dual port RAM which allows the PS to write in the pixel values via one side and the PL to read out via the other port.

Within this RAM each pixel will be allocated one address which will store the 8 bit green red and blue value. The first address of the BRAM will contain the number of pixels to be configured. Setting this to zero will prevent the neo pixel array from being updated and as such acts as an enable.

This approach therefore provides the following context diagram of the system solution

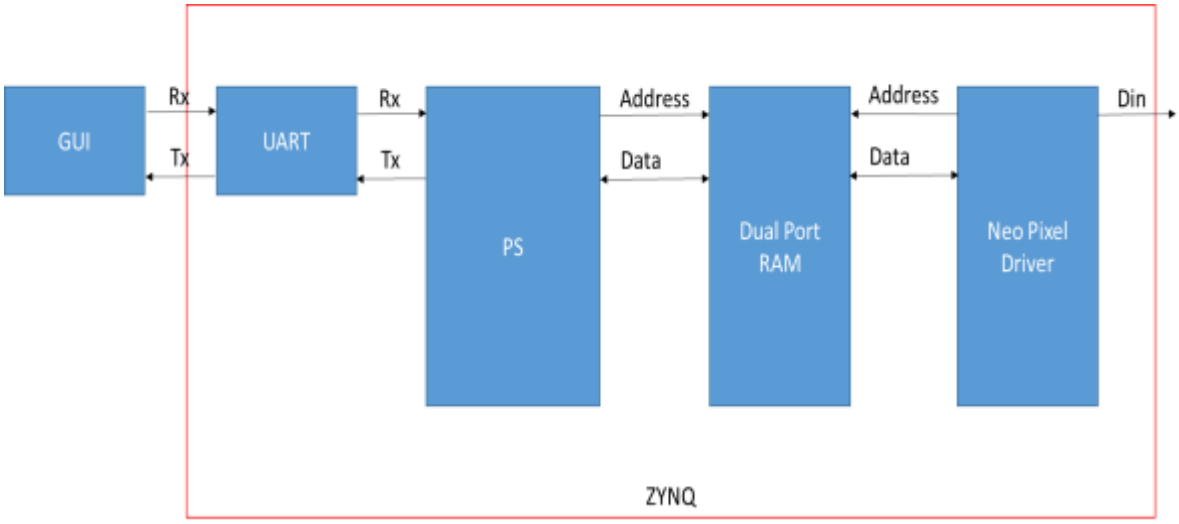


Figure 117 System Context Diagram

Rather helpfully within Vivado IP Integrator there already exists a number of IP blocks provided in the IP library which provide the function I require. This provides two benefits it not only decreases the time it takes for me to get my solution up and running but it also enables me to focus my design effort in areas where I can add the best value (should this be for a commercial application).

Therefore this solution will be using the following IP cores in addition to the PS and PS reset cores.

- AXI Interconnect – provides flexibility of AXI connections within the PL side, for this solution only one AXI master is needed.
- AXI Bram controller – This allows a for AXI interconnection to the interface to a BRAM, allowing the PS to write into BRAM attached to this controller.
- Dual Port BRAM – With one port connected to the AXI BRAM controller and the other to the custom Neo Pixel driver.

The block diagram of the completed solution is presented below

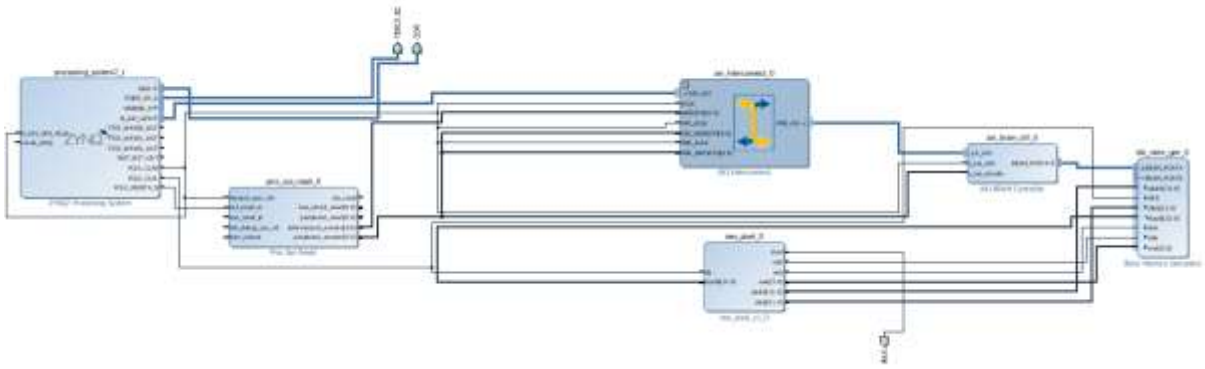


Figure 118 Block Diagram

This means I only really need to write the Neo Pixel driver which we will be looking at in the next blog.

System of Modules Example Part Five

When we last looked at the neo pixel example we had created the architecture of the solution within the Vivado block diagram. By careful consideration of the solution we had maximised the use of existing IP blocks such that we only needed to create one new module to drive the neo pixels themselves.

The focus of this blog will be that neopixel driver, which will exhibit the following behaviour:-

1. Read address 0 within the block RAM to determine the number of pixels in the array
2. If the number of pixels is not 0, read the next address to obtain the red, green and blue bytes for the first pixel
3. Determine which waveform (one or zero) to output for the most significant bit of the 24 bit word read from memory
4. Output that waveform to the neo pixel
5. Increment the word position counter e.g. msb-1, msb-2….
6. Determine which waveform to output for that next bit
7. Repeat steps 5 and 6 above until the all 24 bits have been output
8. Check if the correct total number of pixels have been output
9. If there are more pixels to output repeat steps 3 to 8 above after incrementing memory address for the next pixel.
10. If the correct number of pixels have been output, wait for the reset time for the pixels to latch in the pixel values.
11. Repeat steps 1 to 10.

This therefore requires two elements which most FPGA engineers should be familiar with a statemachine (see Xcell Journal issue 81How to Implement State Machines in Your FPGA) to perform the interfacing with the dual port RAM, sequencing of the bit to be output and the number of pixels required to be output

A shift register is used to output the waveform for each of the Neopixel bits, remember these are self-clocking waveforms so will they have transitions in different places depending if this is a one or a zero to be output. To ease this I used a two predefined constants which are used to pre-load the shift register to the one or zero pattern depending upon the bit to be output. (See attached code and test bench).

```
CONSTANT zero  : std_logic_vector(24 DOWNTO 0) := "1111111000000000000000000"; --waveform for a zero bit
CONSTANT one   : std_logic_vector(24 DOWNTO 0) := "1111111111111100000000000"; --waveform for a one bit
CONSTANT numb_pixels : integer := 24; --number of bits in a pixel
CONSTANT reset_duration : integer := 1000;--number cclocks in the reset period
```

The length of the constant and shift register depends upon the clock rate which the module is driven by, in this case it uses FCLK_CLK1 from the PS side of the device, this is set to 20 MHz which gives a period of 50 ns. This is important as the timing requires resolution down to 50 ns, when outputting a logic zero, in line with the timing diagram and table below.

Data transfer time(TH+TL=1.25μs±600ns)

T0H	0 code ,high voltage time	0.35us	±150ns
T1H	1 code ,high voltage time	0.7us	±150ns
T0L	0 code , low voltage time	0.8us	±150ns
T1L	1 code ,low voltage time	0.6us	±150ns
RES	low voltage time	Above 50μs	

Sequence chart:

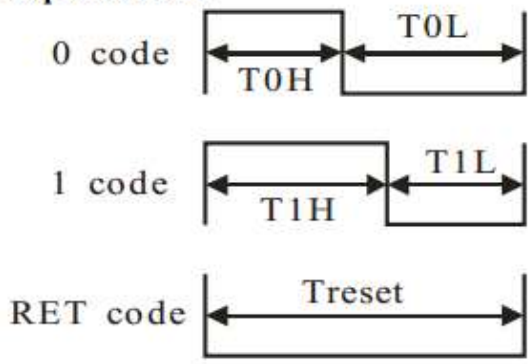


Figure 119 Timing Information for NEO Pixels

The timing overall period is different as well depending if a one or a zero is being 1.15us for zero output and 1.3 us for a one output. However as the tolerance is +/- 600 us this enables us to use the same period for both one and zero waveforms. This means a duration of 1.25 us or when clocked out at 50 ns period a 25 element shift register which can be pre-loaded and provides the 50ns required.

Once I had completed the RTL the next step was to verify the design, all engineers know that verification can take longer the original design. In this case using modelsim I created a simple test bench which stimulated the RTL and allowed me to confirm the behaviour of the module was as required. In the next blog we will look at the verification strategy I will be using to test this demonstration.

Vivado Flow Neo Pixel Driver Example Source Code

```
LIBRARY IEEE;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY neo_pixel IS PORT(
  clk   : IN std_logic;
  dout  : OUT std_logic;
  rstb  : OUT STD_LOGIC;
  enb   : OUT STD_LOGIC;
  web   : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
  addrb : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
  dinb  : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
  doutb : IN STD_LOGIC_VECTOR(31 DOWNTO 0)
);
END ENTITY;

ARCHITECTURE rtl OF neo_pixel IS

TYPE FSM IS (idle,wait1,led,count,reset,addr_out,wait2,grab,wait_done,done_addr);

CONSTANT done  : std_logic_vector(25 DOWNTO 0) := "00000000000000000000000001";  --shows when shift reg empty
CONSTANT zero  : std_logic_vector(24 DOWNTO 0) := "1111111000000000000000000"; --waveform for a zero bit
CONSTANT one   : std_logic_vector(24 DOWNTO 0) := "1111111111111100000000000"; --waveform for a one bit
CONSTANT numb_pixels : integer := 24; --number of bits in a pixel
CONSTANT reset_duration : integer := 1000;--number cclocks in the reset period

SIGNAL shift_reg : std_logic_vector(24 DOWNTO 0) := (OTHERS=>'0'); -- shift reg containing the output pixel waveform
```

```
SIGNAL shift_dne : std_logic_vector(25 DOWNTO 0) := (OTHERS=>'0'); -- shift reg for timing the output shift reg for next load

SIGNAL current_state : fsm := idle; --fsm to control the pixel beign output

SIGNAL prev_state : fsm := idle; --previous state


SIGNAL load_shr : std_logic :='0'; --loads the shr with the next pixel

SIGNAL pix_cnt  : integer RANGE 0 TO 31 := 0; --counts the position in the pixel to op

SIGNAL rst_cnt  : integer RANGE 0 TO 1023 := 0; --counts number of clocks in the reset period 50 us @ 20 MHz

SIGNAL led_numb : integer RANGE 0 TO 1023; --number of LED in the string

SIGNAL ram_addr : integer RANGE 0 TO 1023:=0; --address to read from RAM

SIGNAL led_cnt  : integer RANGE 0 TO 1023;--counts leds it has addressed

SIGNAL pixel    : std_logic_vector(23 DOWNTO 0); --holds led value to be output


BEGIN


web  <= (OTHERS => '0');

rstb <=  '0';

dinb <= (OTHERS =>'0');


pixel_cntrl :  PROCESS(clk)

BEGIN

 IF rising_edge(CLK) THEN

   load_shr <= '0';

   enb <= '0';

   CASE current_state IS

   WHEN idle =>

    current_state <= wait1;

    rst_cnt  <= 0;

    addrb <= std_logic_vector(to_unsigned(ram_addr,32));

    enb<='1';

   WHEN wait1 =>
```

```
  current_state <= led;
WHEN led =>
  led_numb <= to_integer(unsigned(doutb));
  IF to_integer(unsigned(doutb)) = 0 THEN
    current_state <= idle;
  ELSE
    current_state <= addr_out;
    ram_addr <= ram_addr +4;
  END IF;
WHEN count =>
  IF pix_cnt = (numb_pixels-1) THEN
    IF led_cnt = (led_numb-1) THEN
     current_state <= reset;
     pix_cnt <= 0;
     ram_addr <= 0;
    ELSE
     ram_addr <= ram_addr+4;
     current_state <= done_addr;
     led_cnt <= led_cnt + 1;
    END IF;
  ELSE
  current_state <= wait_done;
  END IF;
WHEN done_addr =>
  IF (shift_dne(shift_dne'high-1) = '1') THEN
    current_state <= addr_out;
  END IF;
WHEN wait_done =>
  IF (shift_dne(shift_dne'high-1) = '1') THEN
     load_shr <='1';
     pix_cnt <= pix_cnt + 1;
```

```
        current_state <= count;
       END IF;
    WHEN addr_out =>
     addrb <= std_logic_vector(to_unsigned(ram_addr,32));
     enb<='1';
     current_state <= wait2;
    WHEN wait2 =>
     current_state <= grab;
     prev_state <= wait2;
    WHEN grab =>
     pixel <= doutb(doutb'high-8 DOWNTO doutb'low);
     load_shr <= '1';
     current_state <= wait_done;
     pix_cnt <=0;
    WHEN reset =>
     pix_cnt <= 0;
     led_cnt <= 0;
     IF rst_cnt = (reset_duration-1) THEN
       current_state <= idle;
       ram_addr <= 0;
     ELSE
       rst_cnt <= rst_cnt + 1;
     END IF;
    END CASE;
  END IF;
END PROCESS;

shr_op : PROCESS(clk)
BEGIN
  IF rising_edge(clk) THEN
    IF load_shr ='1' THEN
```

```
    shift_dne <= done;
    IF pixel((numb_pixels-1)-pix_cnt) = '1' THEN
      shift_reg <= one;
    ELSE
      shift_reg <= zero;
    END IF;
  ELSE
    shift_reg <= shift_reg(shift_reg'high-1 DOWNTO shift_reg'low) & '0';
    shift_dne <= shift_dne(shift_dne'high-1 DOWNTO shift_reg'low) & '0';
  END IF;
 END IF;
END PROCESS;

dout <= shift_reg(shift_reg'high);

END ARCHITECTURE;
LIBRARY  ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY tb IS
END ENTITY;

ARCHITECTURE behavioural OF tb IS

COMPONENT neo_pixel IS PORT(
 clk   : IN std_logic;
 dout  : OUT std_logic;
 rstb  : OUT STD_LOGIC;
 enb   : OUT STD_LOGIC;
 web   : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
```

```
  addrb : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
  dinb  : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
  doutb : IN STD_LOGIC_VECTOR(31 DOWNTO 0)
);
END COMPONENT;

COMPONENT uzed_blk_mem_gen_0_0 IS
  PORT (
    clka : IN STD_LOGIC;
    rsta : IN STD_LOGIC;
    ena : IN STD_LOGIC;
    wea : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
    addra : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    dina : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    clkb : IN STD_LOGIC;
    rstb : IN STD_LOGIC;
    enb : IN STD_LOGIC;
    web : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
    addrb : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    dinb : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    doutb : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
  );
END COMPONENT uzed_blk_mem_gen_0_0;

SUBTYPE vector_size IS std_logic_vector(31 DOWNTO 0);
TYPE neo IS ARRAY (0 to 30) OF vector_size;

CONSTANT mem_data : neo := (
x"0000001e",
x"00DDDAC3",
```

```
x"006A4D03",
x"0027C3F1",
x"0046CE04",
x"000FB9ED",
x"00977D04",
x"00C1FDB2",
x"007A9E52",
x"00B34C36",
x"0056640F",
x"00F6D97A",
x"00F88BBB",
x"0012A820",
x"00BB1DF2",
x"00034EC3",
x"0054846C",
x"00EF2B7F",
x"000D685F",
x"005F0F59",
x"006BF532",
x"00F5A778",
x"00C5E351",
x"00BA06D8",
x"00F9C0F1",
x"004BE2D5",
x"0055E0F0",
x"000A44BA",
x"00B9DCB6",
x"00B26C94",
x"00689745");

CONSTANT clk_period : time := 50 ns;
```

```
SIGNAL    clk   :  STD_LOGIC:='0';
SIGNAL    rst   :  STD_LOGIC:='0';
SIGNAL    ena   :  STD_LOGIC:='0';
SIGNAL    wea   :  STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL    addra :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL    dina  :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL    douta :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL    enb   :  STD_LOGIC;
SIGNAL    web   :  STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL    addrb :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL    dinb  :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL    doutb :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL    dout  :  STD_LOGIC;

BEGIN

clk_gen :  PROCESS
BEGIN
  LOOP
    clk <= NOT(CLK);
    WAIT FOR(clk_period/2);
    clk <= NOT(CLK);
    WAIT FOR(clk_period/2);
  END LOOP;
END PROCESS;

ram_uut : uzed_blk_mem_gen_0_0 PORT MAP(
    clka  => clk,
    rsta  => rst,
    ena   => ena,
```

```
    wea   => wea,
    addra => addra,
    dina  => dina,
    douta => douta,
    clkb  => clk,
    rstb  => rst,
    enb   => enb,
    web   => web,
    addrb => addrb,
    dinb  => dinb,
    doutb => doutb);

uut : neo_pixel PORT MAP(
  clk   => clk,
  dout  => dout,
  rstb  => OPEN,
  enb   => enb,
  web   => web,
  addrb => addrb,
  dinb  => dinb,
  doutb => doutb);

ram_load : PROCESS
BEGIN
  FOR i IN 0 TO 30 LOOP
    WAIT FOR clk_period;
    WAIT UNTIL rising_edge(clk);
    wea <= (OTHERS=>'1');
    ena <='1';
    addra <= std_logic_vector(to_unsigned(i*4,32));--needs to be on boundaires of four
    dina <= mem_data(i);
```

```
    WAIT UNTIL rising_edge(clk);
    wea <= (OTHERS=>'0');
    ena <='0';
    WAIT FOR 10 ns;
  END LOOP;
  WAIT;
--  REPORT "simulation Complete" SEVERITY failure;
END PROCESS;

END ARCHITECTURE
```

System of Modules Example Part Six

Having got to the point that in the design where we have designed the Neo pixel driver integrated it into the Vivado block diagram and connected it to the remainder of the system. We need to think a little about the verification strategy typically this can be more involved than the design itself.

The strategy I will be undertaking for this example is

1. Verify the hardware interfaces
2. Simulate the Neo Pixel Driver using a VHDL simulator
3. Build the System for testing on the Zynq
4. Develop test software to read and write from all addresses BRAM to ensure the PS can correctly address the memory
5. Develop test software to drive the first Neo Pixel on the array to verify the functioning of the Neo pixel driver and verify the timing waveforms sent to the neo pixel using a oscilloscope
6. Develop test software to drive a number of pixels located at different points of the array e.g. second pixel, mid pixel and final pixel in the array
7. Use a terminal programme to test the final software can respond to in incorrect commands – out of sequence commands etc.
8. Final functional verification using a developed GUI to prove the final development works as specified originally

The above steps may seem complicated however in many instances as this is a simple application, they can be performed with relative ease.

Generally when thinking about verifying a system we need to demonstrate the system requirements have been achieved. Often before the direct functionality can be verified e.g. driving a neo pixel we need to think about verifying the hardware and interface requirements first. In many systems these hardware and interface requirements would be flowed down to the sub system requirements to allow verification against them.

Normally the first element of any testing is ensuring the hardware is operating correctly see Xcell issue 82 Nuts and Bolts of Designing an FPGA into Your Hardware and issue 85 A Pain-Free Way to Bring Up Your Hardware Design. However one benefit of the system of modules approach is that the under laying hardware on the Microzed itself already comes provided as known good hardware and functioning. This reduces the time it will take to commission our system and start verifying our functional requirements (it does not however, mean we will correctly implement our SoC). However we still need to ensure that we have the following at the hardware level before we can progress.

1. The correct pin out from the IO carrier card PMOD to the Neo Pixel connections, this is best done using a multi meter when the power is not applied to the MicroZed to ensure the VCC and GND supplies to the Neo Pixel array are correct.

2. Correct IO voltage upon the IO bank driving and powering the Neo Pixel array, this can be as simple as using a multi meter to record the voltage is 3.3 volts.

3. Once we have proved the physical interconnect and power it is a good idea to power up the MicroZed and check with the neo pixel array attached that the power supply to it is stable using the multi meter.

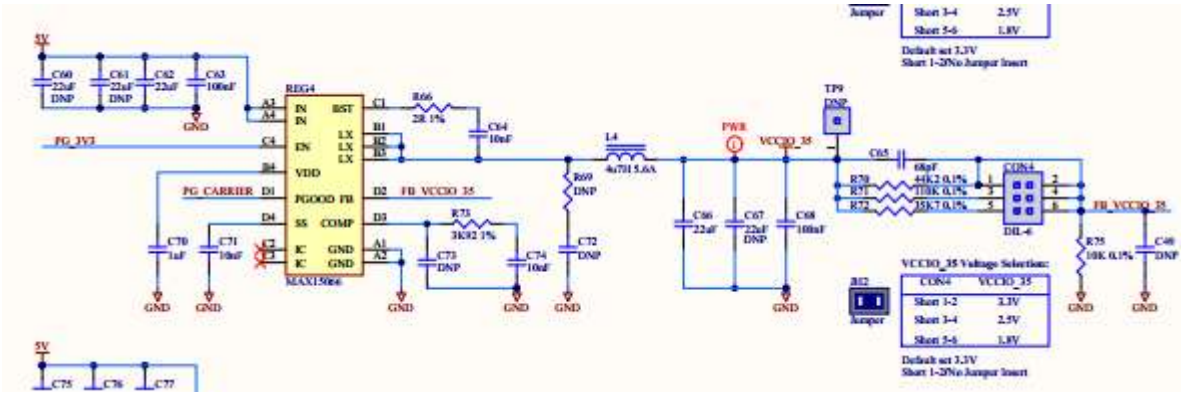


Figure 120 Test Point for the 3v3 Bank 35 supply

In my previous blog, we looked at the design and simulation of the NeoPixel driver using ModelSim, which addresses point 2 in the verification strategy and helpfully allows me to move on to point 3: verifying the functional performance of the Zynq SoC's PS and PL side of the design and the interaction between the two of them.

As we have been using the MicroZed design in the PS for a number of weeks—35 weeks, as I make it—I am pretty confident that the PS is correctly configured to boot from the SD card and execute the program from the DDR RAM. However this is the first time in this series of blogs that I have used the BRAM controller and the BRAM, which is why I developed test software to ensure I could read and write the memory from the PS side correctly.

The testing of the system's ability to drive the NeoPixel array begins to take shape during points 4 and 5 in the verification strategy. Both are closely linked and are a natural evolution from one test to the next. The first test software enabled me to write to just one NeoPixel and check the timing. What I see on the oscilloscope correlates with that from the VHDL simulation and the NeoPixel lights in the correct color. Test successful.

I then modified this test code to drive the one pixel to full scale red, green, and blue, ensuring that the driver can correctly set the pixel color. Once one pixel has been driven, the program is easily adapted to drive specific pixels to different colors. This verifies that the controller can correctly drive a specified pixel in the string. The final change to the test software is to drive all pixels in the string at the same time to the same color. For this test, I chose white because this color turns on all of the red, green, and blue LEDs in the NeoPixel array and therefore allows me to ensure that the maximum current can be driven from the I/O Carrier Card to support this string.

Once this series of tests is complete, we are confident we can drive the attached NeoPixel array. However, we have not yet verified the command interface that we will use to control the setting of the NeoPixel colors. I will address the final two points in the verification strategy regarding the serial interface and GUI in the next blog, where I will introduce the simple serial communications protocol I have implemented to control the Zynq SoC's PS and thus in turn the PL NeoPixel driver

System of Modules Example Part Seven

The example I have been explaining over these last few blogs requires communication over RS232 (in reality a USB to Serial device) to control the settings of each neo pixel in the array. This requires that I had to implement a basic communications protocol between the PC (DTE) and the MicroZed (DCE).

I will be using the UART within the PS of the Zynq to send and receive data, this is also declared with the BSP to be the STDIO

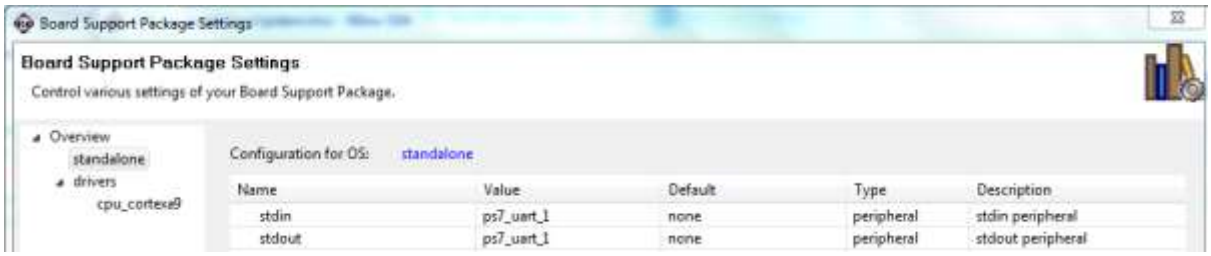


Figure 121 STDIO Definition

This allows me to use the getchar() function to implement my communications protocol. Now because I do not want to re-invent the wheel (why would any engineer) my protocol is going to use ASCII characters to exchange data (an ASCII table is here http://www.ascii-code.com/) each pixel be can therefore be addressed by sending the following data format

<STX> <Pixel Number> <Green><Red><Blue><ETX>

To update a number of pixels each pixel to be updated sends in a packet formatted as above these are transmitted sequentially at a speed of 115200 bps with no parity.

This therefore allows the length of the Neo Pixel string to be determined by the internal memory of the FPGA which is very large (it can store 4096 pixels and can easily be increased in a few minutes for adventurous designs) and therefore effectively allows the PC to determine the size of the Neo Pixel string being addressed.

Within the software all I had to do then was to implement a simple state machine which looped through the states receiving the pixel values before storing the data received as required at the correct block RAM memory location.

Having received all of the bytes over the serial link the red, green and blue bytes are formatted into a 24 bit word and written to the correct address within the Block RAM to enable the PL side of the device to set the colour if the pixels are enabled.

All told this is a very simple communication protocol however, as it is transmitting data there will at times be issues with the communication channel (noise, glitched for example) which is why the protocol must be able to handle errors which may have undesirable effect on the performance.

The state machine needed to implement this uses the only seven states and can be seen in the attached file below. Now this code is not quite the final version as it still has code within it to echo back over the serial link what was received so I can verify the interface using a simple terminal programme.

The terminal programme I used is termite (http://www.compuphase.com/software_termite.htm) with the hex view plug in to allow me to write direct hex codes to be transmitted. This allows me to transmit hex codes much easier from my lap top to corrupt the data and test the protocol is working as expected.

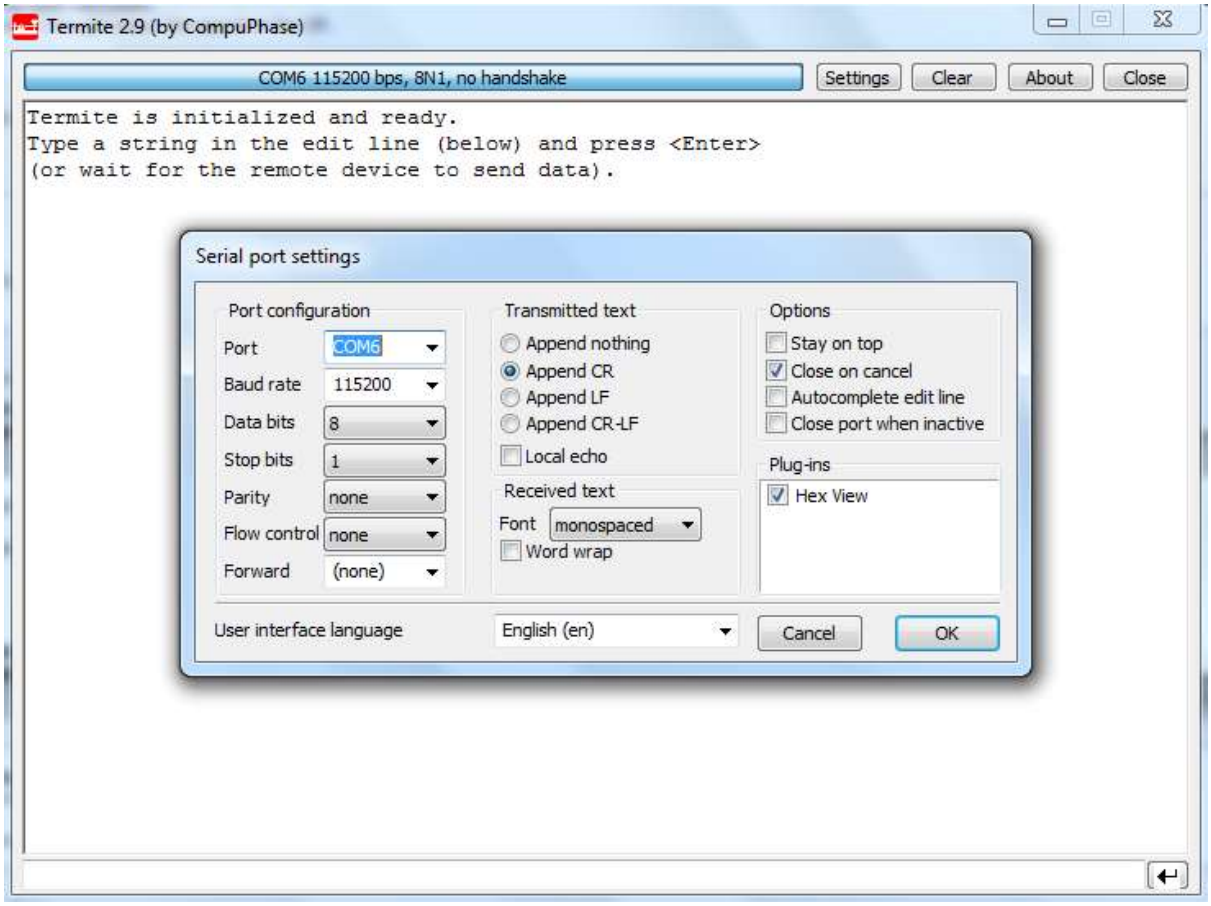


Figure 122 Serial Port Confiuration

Having looked at the verification in the last blog and the serial interface in this blog in my next blog I will explain the development of the GUI using and the final testing of the example allowing me then to move onto more exciting aspects of learning to use the Zynq.

System of Modules Example Part Eight

As we come to wrap up this example which has taken 8 weeks and pulled together a number of concepts we have been looking at with the Zynq. I think it this is a good point to recap what this example has covered

1) The system of modules approach
2) Communication between the PS and PL side
3) Using the IP catalogue to reduce the number of modules we need to implement
4) Creation of a Neo Pixel driver within the PL side of the Zynq
5) The verification approach to be undertaken
6) Definition of a serial protocol to communicate with the Zynq

This final blog on the neo pixels before we move on to look at other aspects of the zynq is to look briefly at the development of the TCL/TK gui which allows the neo pixels to be controlled from a laptop or other remote device.

This was developed to enable the user to select any colour from the 16 million possibilities for the 24 bit colour depth.

The GUI is very simple with a button for each pixel in the string which when clicked will open a colour selection window allowing the selection of colour for that pixel. Following the selection of the pixel colour it will be downloaded to the Zynq and the pixel will update its colour.

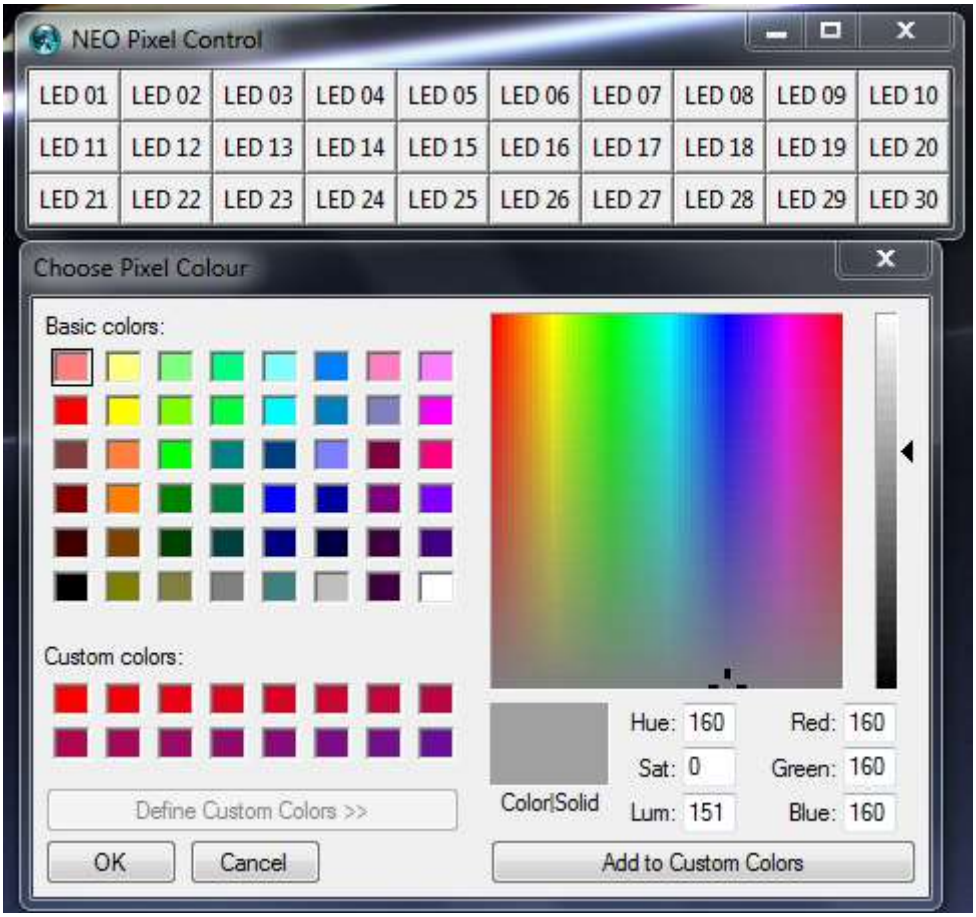


Figure 123 Neo Pixel Final GUI

The major challenge of the generating the GUI was ensuring the data being sent down was correctly formatted as binary and correlated with the colour being selected. For this reason the first version of the GUI also displayed the hex word for each colour selected and the software in the Zynq has been designed to echo back the data it receives. This is received by the GUI and saved into a text file so I could correlate what was requested with what was received by the Zynq an example of the echoed response is below

<STX>pixel number = 32 green = 0 red = ff blue = 0<ETX>

Having got the GUI working correctly the layout was changed to just show the Led numbers and pixel selection.

Validation of the system could then occur using the GUI driver to ensure each pixel can be addressed from the GUI and can be set to Red, Green, Blue and White along with a number of different colours selected at random.

Now having reached the end of this example and as we move on to look at other aspects of the Zynq we can consider how else this problem of driving the Neo Pixels could have been addressed. Remembering back to the part 17 of this blog which introduced the Triple Timer Counter and its ability to generate a PWM waveform this output could have been used to drive the Neo Pixel array however the load on the processor would have been higher.

Adding a OS Part One

To date all of the examples we have looked at in this epic blog have been performed without using an operating system within the Zynq. Such an implementation is called a bare metal system, this can be used for the very simple examples we have been using to date but if we are to use a more advanced processing system and maximise the benefits of the dual core processor we need use an operating system.

Figure 124 Operating System Ecosystem

As you can see above the Zynq is support by a very large ecosystem of operating systems and kernels which can be implemented on the Zynq many of these are ones most software engineers will be familiar with. Therefore going forward with this blog we will be looking at how we can implement the following operating systems on the Zynq such that we have demonstrated a mixture of operating systems.

- uC/OS iiii – a commercial real time system which comes from the uC/OSii family which has been is certified for MISRA-C, DO178B level A, SIL3/4 and IEC61508.

- Free RTOS – a free real time system which is provided by real time engineering limited and has a SIL3 certified version known as SafeRTOS.

- Linux – the Petra Linux distribution, we will be looking at using the standard distribution via GitHub and how we can re compile the kernel to customise it for our design.

- Android – Gingerbread distribution for the Zynq SoC

Implementing these will take a number of weeks and allow the creation of more in depth examples and allows us to use some of the Zynq resources we have not really touched on yet such as USB and Ethernet. As the Zynq has two processor cores we will also look at the Asymmetric Multi Processing, commonly known as AMP were we run different operating systems on each of the processor cores within the Zynq.

Operating systems are designed for a number of different applications from user to embedded to safety critical applications. When most people think of operating systems they tend to think Windows or Linux that are run on personal computers however, operating systems are used many

applications from Satellites to Satellite desk top boxes which receive the signal. This stems from the fact that complex software applications requires management of processor resources, memory and scheduling of tasks (sometimes called processes) along with many more aspects. While this can be written by the engineer from scratch in a bare metal design this will take considerable time to not only develop but verify. Which makes this not the most efficient use of an engineer's time when operating systems exist for the application of course they must select the correct operating system for the end application.

In the next blog we will look at the different types of operating systems and how we can select the one best suited for the application being developed.

Adding a OS Part Two

Having introduced the operating systems I intend to demonstrate on the Zynq, the first of these I intend to implement on the MicroZed is the Micrium uC/OSiii which is a hard real time operating system which is available for download here.

This operating system has been used on number of very interesting systems and is currently in progress for certification for MISRA-C, DO178B level A, SIL3/4 and IEC61508 thus it should have a wide appeal to many users on the Zynq.

I chose this one as I am most interested in using the Zynq for industrial, military, aerospace and other challenging environments. These are areas I think the Zynq can really demonstrate the benefits of a SoC implementation as performance and SWAP-C (Size, Weight and Power -Cost) are driving factors in these environments.

However, before I jump off and start talking about how we implement the operating system I thought I would provide a little back ground information on real time operating systems.

So what makes a real time operating system from an operating system, a real time operating system is deterministic that means the response of the system will achieved within a defined deadline.

But does the system have to always meet these deadlines to be classed a real time system? Actually no there are three categories of RTOS which address the deadline differently

- Hard RTOS – Missing a deadline is seen as a system failure.
- Firm RTOS – Occasionally missing a deadline is acceptable and not seen a failure.
- Soft RTOS – Missing a deadline reduces the usefulness of the results.

Real Time Systems operate around the concept of running tasks (sometimes called processes) each of these tasks performs a required function of the system for example it might read data in over an interface or perform a calculation. A simple system may use just one task but it is more likely for multiple tasks to be running on the processor at any one time. Switching between these is referred to as context switching and requires that the state of the processor for each task is stored and added to the task stack. Determining which task is to be run next is controlled by the kernel and can be complicated (especially if we want to avoid deadlock were tasks lock each other out) but the two basic methods are

- Time sharing – Each task gets a dedicated time slot on the processor; higher priority tasks can have multiple time slots, this time slicing is controlled via a regular interrupt or timer. This is often called round robin scheduling.
- Event Driven – Tasks are only switched when one with a higher priority is required to be run. This is often called pre-emptive scheduling

µC/OSiii is a pre-emptive RTOS therefore it will always run the task with the highest priority which is ready be executed.

In the next blog we will look at how we can communicate between tasks (often called inter process communication) and events like deadlock and starvation in more detail. While for many this may be a recap it is important I think that I explain all of the fundamental concepts that we will be using as we look more into operating systems and the Zynq.

Adding a OS Part Three

Having looked at the different types of a real time operating system, in my previously I think it is a good idea to look at how tasks communicate and how they share the hardware resources available and what some of the pitfalls maybe.

When two or more tasks want to share a resource for example the XADC it is possible for them to request the resource at the same time hence access needs to be controlled to prevent contention. How this is managed is very important as without the correct management deadlock or starvation might occur.

Deadlock – Occurs when a task is holding a resource and cannot release it as it is currently unable to complete as it requires another resource which is currently held by another task. As this will indefinitely stay in this state the application is said to be deadlocked, deadlock is a bad situation for a real time operating system to find itself in.

Starvation – Occurs when a task cannot run as the resources it needs are always allocated to another task.

As you can imagine there has been lots written on the subject over the years and there are many proposed solutions for instance the Decker algorithm. The most commonly used method to handle this situation are semaphores, which commonly come into two types binary semaphores and counting semaphores.

Typically each resource has a binary semaphore allocated to it, a requesting task will wait for the resource to become available before executing and once completed it will release the resource these are commonly known as WAIT and SIGNAL operations. A task will WAIT on a semaphore, if the resource is free it will then be given control of the resource and run until completed at which point it will SIGNAL completion. However, if the resource is currently occupied when the task WAITs's on the semaphore it will be suspended until it is free, this could mean once the currently executing task is finished or require a longer wait if it is pre empted by a task of higher priority. There is a special class of binary semaphores which are called mutex's these are often used to prevent priority inversion.

Counting semaphores work in the same way as binary semaphores however they are used when more than one resource is available for instance data stores. As each of the resources are allocated to tasks the count is reduced to show the number of free resources remaining, when the count gets to zero there are no more resources available and the requesting process will be suspended until one is released.

It is often required also to communicate between tasks of which there are a number of methods the simplest of which is to use a data store and semaphores as described above, more complex methods include message queues.

With message queues when a task wishes to send information to another it POSTs a message to the queue, when a task wishes to receive a message from a queue it PENDs on the queue. Message queues therefore work like a FIFO, however within uC/OSiii it is possible to configure the message queue to act in a LIFO manner. Message queues and their organisation can be a very complicated and in depth subject which could take up many more blogs.

Having explained a little on the subject of resource sharing and how tasks communicate in my next blog I will look at getting the uC/OSiii demo up and running

Micruim OSiii Operating System Demo

Over the last few blogs we have looked at the concepts of a RTOS having introduced these it is time to implement the first OS which is uC/OSiii this blog will show how to get the demo up and running.

Obviously the first thing we need to do is to download uC/OSiii from the Micrium website, it is available here http://micrium.com/downloadcenter/download-results/?searchterm=hm-xilinx&supported=true having downloaded this installation is pretty simple you just need to extract a few zip files into the correct directories under your Xilinx installation on your computer.

Ensure you extract the zip file marked Zynq-7000-ucosiii-bsp.zip into your directory <Xilinx>\14.X\ISE_DS\EDK\sw\lib\bsp\ you will notice a number of the other operating systems under this directory for example the standalone and xilkernel.

While the zip marked Zynq-7000-ucosiii-demo.zip should be extracted into the directory <Xilinx>\14.X\ISE_DS\EDK\sw\lib\sw_apps\ again you will see a number of other application demos within this directory.

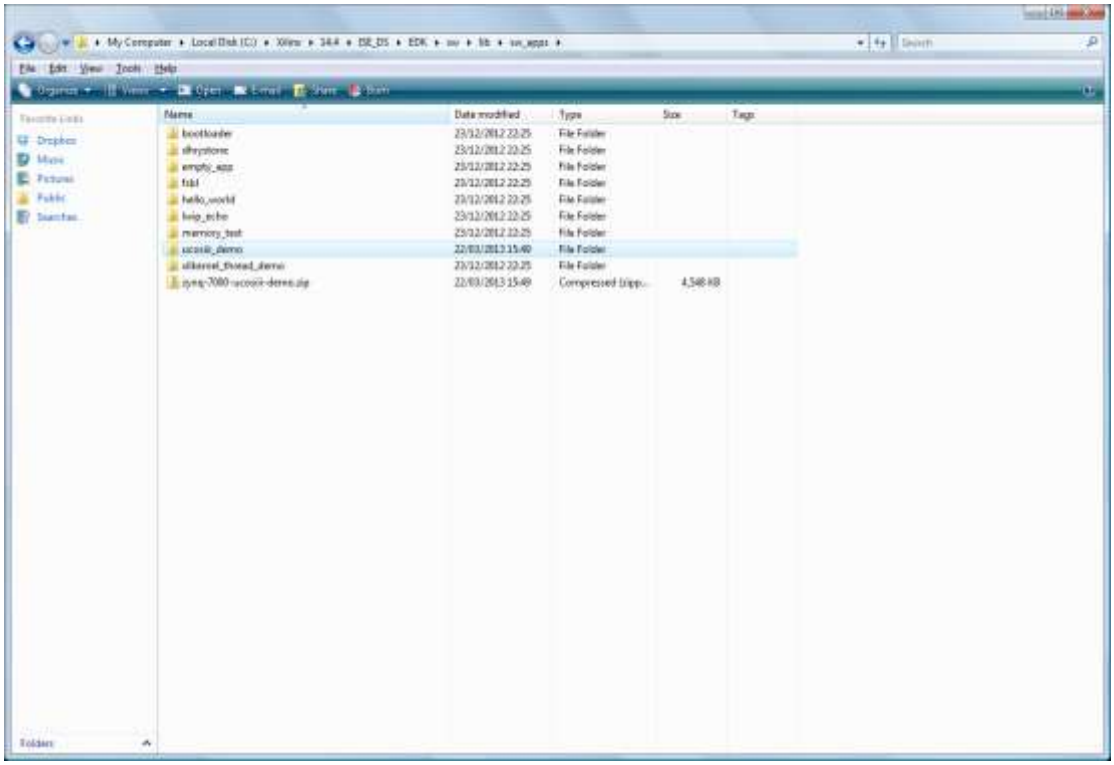


Figure 125 Location of the demo application

Having installed the two sets of files you are then ready to begin creating your new project within SDK, to do this I will be using the same base hardware which was created before however, we will require a new application and a board support package as this time we wish to include operating system. Within SDK close all open projects except your base hardware design and select the option file->new->application project give the new project a name and select operating system you require in this case uC/Osiii.

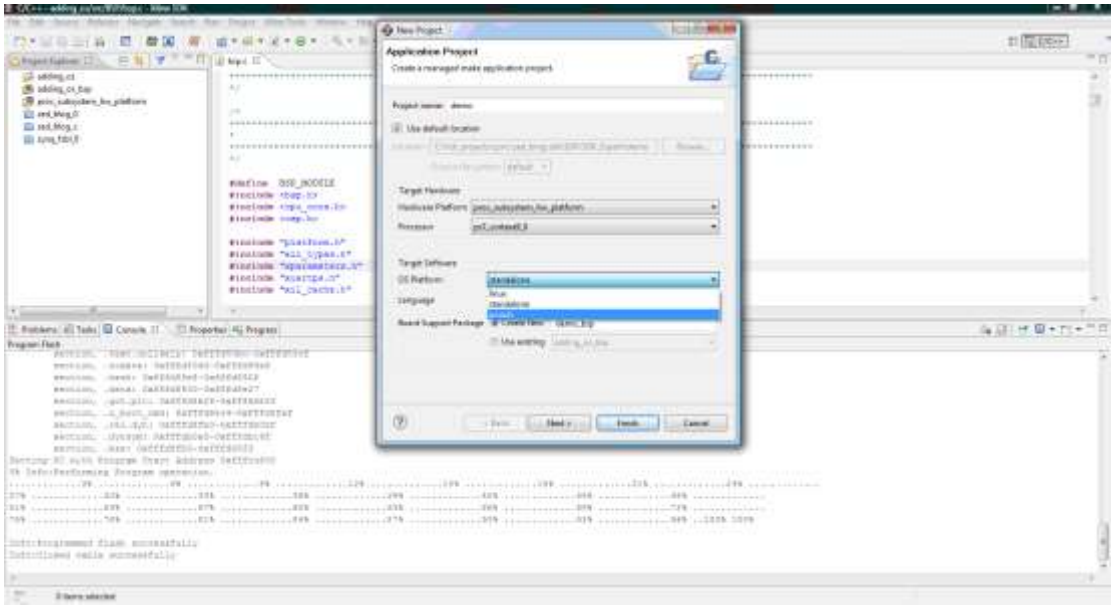


Figure 126 Selecting the OS

Click on next and select the ucosiii demo application this will import all the files needed to use this operating system.

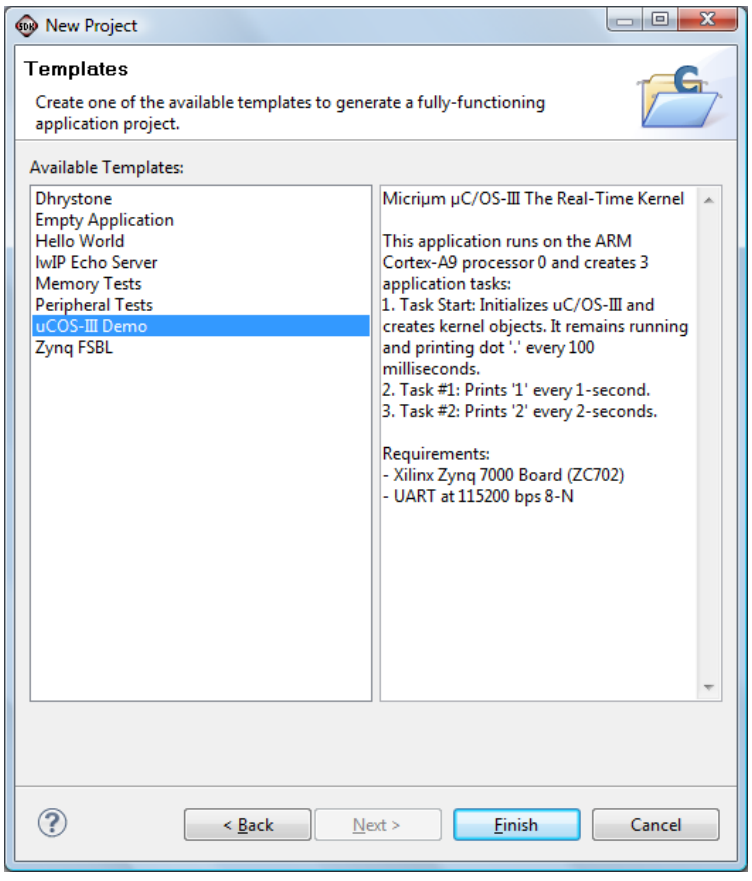


Figure 127 Selecting the Demo

Once you are happy click on finish and the application and board support package will be created within SDK for you. If you have the auto build option selected you may find a few errors are reported this is because not all of the references are correct yet so, to set these we need to import demo settings. These can be found under the project ->Src-> settings right click on this XML file and view the properties allowing and copy the location of this file

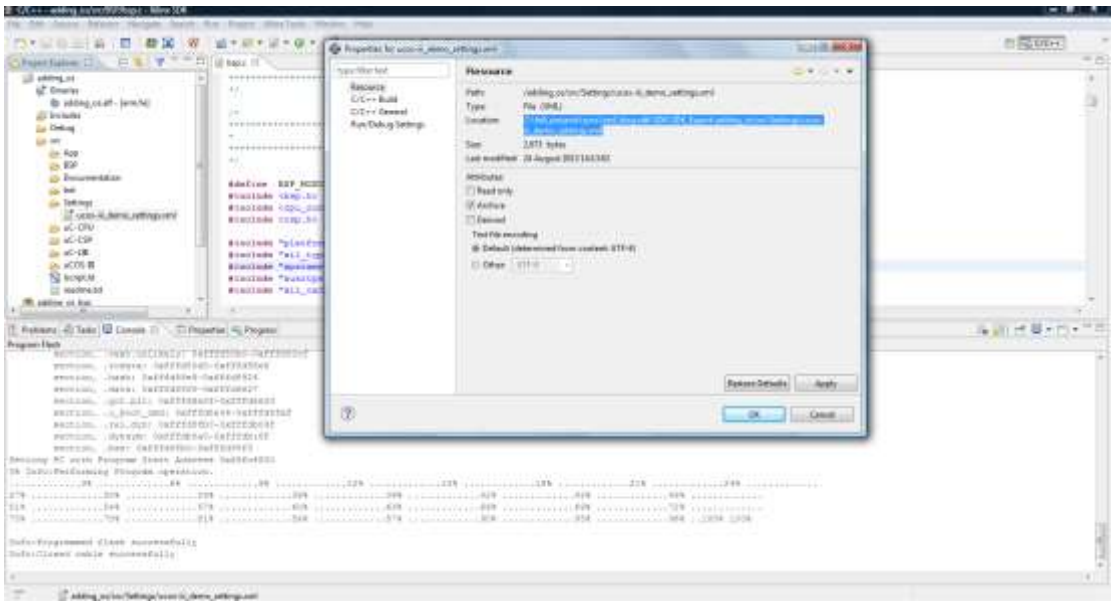


Figure 128 Location of the references requried

Once you have copied this location, right click on the project and select properties and under C/C++ General select the paths and symbol options select import settings and paste in the location of the settings file.

It is also important to ensure that the repositories are correctly pointing to the new libraries you added earlier; you can check these by setting Xilinx Tools ->Repositories this should show the location where you installed the uC/Osiii BSP previously.

As we wish to use the UART to output the status of the demo, showing the initialisation being complete and the tasks running, you may need to set the stdin and stdout to the UART under the BSP settings.

Having performed these actions you will see that the project is able to be built however there will still be a few warnings and if you tried to run this on your Microzed however it would not perform as the demo states it should this is because of a warning over undeclared functions, including `#include "xil_cache.h"` within bsp.c should correct this issue.

Once I added this include header file the project built and ran as expected on the microzed board producing the following video.

http://www.youtube.com/watch?feature=player_embedded&v=uRB4La5ijrA

Adding FreeRTOS

FreeRTOS was developed by Real Time Engineering and provides a very real benefit to embedded systems in its small footprint and very fast execution.

FreeRTOS over the years has increased in popularity to become incredibly popular and has for the last four years been in the top off class RTOS in the EE Times embedded systems marketing survey. This is not surprising as it is totally free even for commercial applications and comes with a certified safety critical version SafeRTOS (which is available for purchase). FreeRTOS benefits from a considerable ecosystem which includes CLI, TCP/IP, UDP/IP and file systems to reduce the time to market for many applications.

The FreeRTOS website also includes a number of discussion forums for developers and engineers to ask questions and learn about how to best use the OS (http://www.freertos.org/FreeRTOS_Support_Forum_Archive/freertos_support_forum_archive_index.html)

To get the demo up and running the first thing we need to do is download the FreeRTOS application which can be obtained from http://www.freertos.org/ the current version is V8.0.1. This downloads with all the architecture ports and demos which demonstrates the small size of the RTOS as the entire download is just over 175 MB when extracted.

Once you have downloaded the zip file, it is self-extracting and contains links which need to be maintained so please ensure you extract it within the directory structure you want.

The next stage is to import the Zynq demo project into SDK (File -> Import) this demo has been developed for the ZC702 development board and comes with a BSP and HW definition for that particular board. However, as I am using the MicroZed board and the BSP and HW definitions included in the demo are those defined by Vivado and SDK with no modification I will be using my own hardware definition and BSP to run this demo hence I will be only importing the demo application.

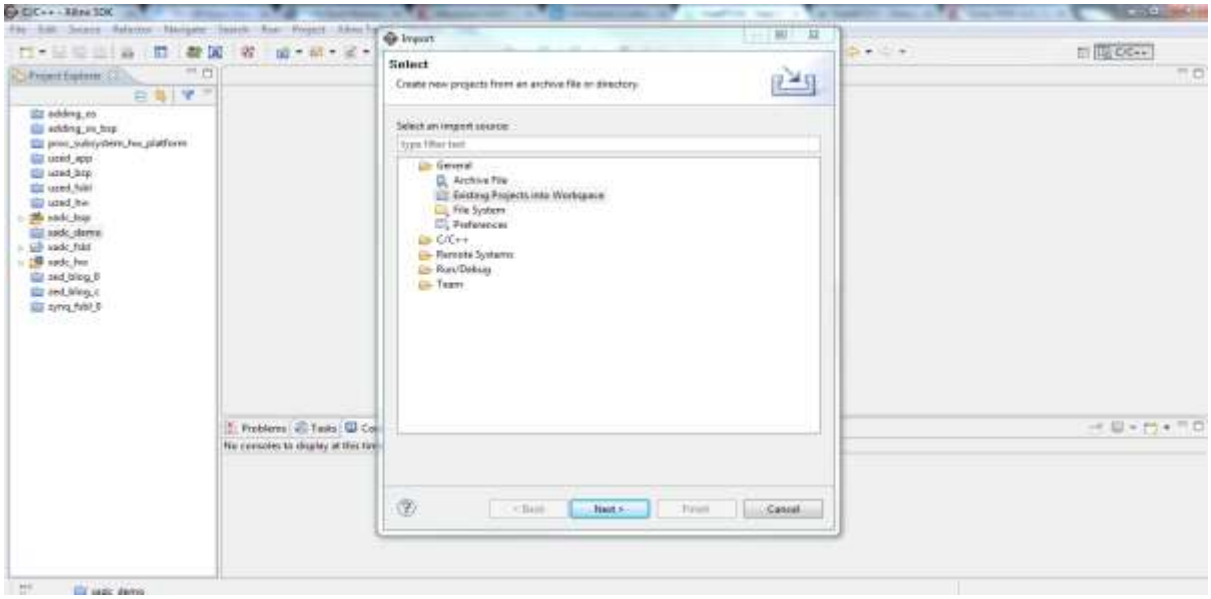


Figure 129 Importing a project into FreeRTOS

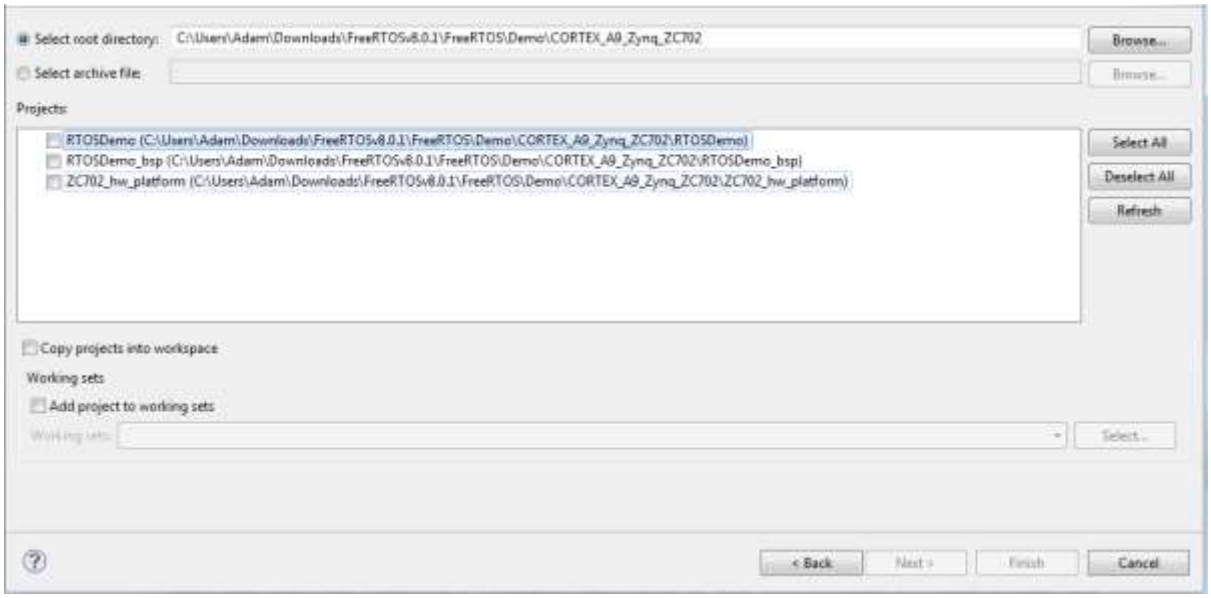


Figure 130 Selecting the projects to import

I therefore only checked the top box to import the actual project, this requires that the referenced BSP is changed for the demo application to do this we select the demo application and choose the change BSP option, selecting the BSP of choice will of course select the reference the HW design.

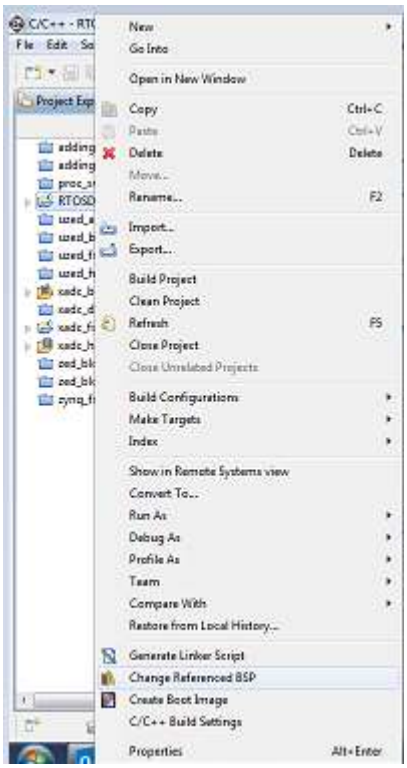


Figure 131 Changing the referenced BSP

Having referenced the desired BSP we are now in a position to build the demo application and try it on the hardware of the MicroZed. This is the same as for any of the other developments we have built over the course of this blog.

However, as we are using the MicroZed and not the ZC702 we need to make a slight change to the code as the demo should flash a LED on the development board. However the ZC702 uses a LED

connected MIO10 while the MicroZed has a LED connected to MIO47. Making this change is very simple under the RTOSDemo application SRC folder select the file ParTest.c which defines the GPIO interfaces for the processor open the file and change the line as below

```
#define partstLED_OUTPUT            (10) //before
#define partstLED_OUTPUT            (47) //after
```

This has changed the pin mapping from MIO10 to MIO47

The demo when running successfully has a very nice command line interface (CLI) available over the rs232 interface as well as the flashing LED. This is a very interesting interface as it allows us to see the run time stats of all the tasks currently running a snap shot of this can be seen below

```
 Lists all the registered commands


task-stats:
 Displays a table showing the state of each FreeRTOS task

run-time-stats:
 Displays a table showing how much processing time each FreeRTOS task has used

echo-3-parameters <param1> <param2> <param3>:
 Expects three parameters, echos each in turn

echo-parameters <...>:
 Take variable number of parameters, echos each in turn

[Press ENTER to execute the previous command again]
>run-time-stats

Task            Abs Time      % Time
****************************************
CLI             41            <1%
Math3           1796170       8%
Math4           1739625       8%
QOver           1737590       8%
SetB            36991         <1%
Reg1            12111         <1%
L1QRx           49333         <1%
Reg2            1777004       8%
IDLE            10471         <1%
SUSP_RX         1836852       8%
QProdB2         21955         <1%
QConsB3         20084         <1%
QProdB5         40378         <1%
L2QRx           48161         <1%
MuLow           102172        <1%
CNT1            1741025       8%
CNT2            1672208       7%
CNT_INC         32916         <1%
Rec3            1752878       8%
```

Figure 132 FreeRTOS Command Line Interface

FreeRTOS Creating Tasks

Having successfully got the demo up and running, we obviously want to be able to write our own application, this first example will be simple to use to configure the XADC and output results over the serial link, it will also checked the received value and if above predefined level it will set a LED. This will enable us in future blogs to look at how we can communicate this information between tasks and deliver a more complex application.

The first stage of the development is to create a function which can be called before we start the scheduler to initialise the XADC this will enable us to use the XADC in one of the two tasks we will be using for this example, this is a standard function like we have created many times before.

This example will use two tasks once which reads the XADC temperature and adds the value to the queue while the second reads the value from the queue and performs the temperature check and for this application outputs data over the serial link so we can see what is occurring. Thus demonstrating how we can communicate with hardware peripherals within the Zynq in one task and use its values in a second task. Inter task communication being one of the key aspects of task communication in RTOS and embedded systems.

Rather helpfully the demo comes with two examples the full demo and a much simpler blinking demo, this is controlled by setting the pre-processor declaration within main.c

```
#define mainCREATE_SIMPLE_BLINKY_DEMO_ONLY	1
```

Setting this and rebuilding the application results in a simple example which blinks the LED only on the MicroZed, it does this passing data between two tasks at a predefined rate using a queue. Nearly exactly what we require and hence this provides a useful starting point for the application we wish to develop.

As mentioned above the first thing we need to create is a XADC configuration routine to initialise it for use, remember to add in the required include files and the declarations needed to access the xadc. We have done this several times over the course of this blog so it should be second nature to you now. The next task is to modify the transmitting task to read the XADC temperature and send the value retrieved using the queue to the receiving process.

```
static void prvQueueSendTask( void *pvParameters )
{
TickType_t xNextWakeTime;
unsigned long ulValueToSend = 100UL;
	( void ) pvParameters;
	xNextWakeTime = xTaskGetTickCount();
	printf("task tx");
	for( ;; )
	{
		vTaskDelayUntil( &xNextWakeTime, mainQUEUE_SEND_FREQUENCY_MS );
		ulValueToSend = XAdcPs_GetAdcData(&XADCInst, XADCPS_CH_TEMP);
		xQueueSend( xQueue, &ulValueToSend, 0U );
	}
}
```

While within the receiving process we can quickly make this output the resultant value received from the queue, we can then change the point at which the LED comes on my comparing the expected value against the received value and toggling the LED.

```
static void prvQueueReceiveTask( void *pvParameters )
{
unsigned long ulReceivedValue;
unsigned long ulExpectedValue = 43000;
	( void ) pvParameters;
	printf("task rx");
	for( ;; )
	{
		xQueueReceive( xQueue, &ulReceivedValue, portMAX_DELAY );
		printf("Raw Value = %lu", ulReceivedValue);
		if( ulReceivedValue == ulExpectedValue )
		{
			vParTestToggleLED( mainTASK_LED );
			ulReceivedValue = 0U;
		}
	}
}
```

As can be seen above this very simple approach will be able to set the LED should the temperature go above an alarm value. However, in many cases this will result in chattering of the LED as values close to it will lead to it quickly turning on and off if the condition is borderline.

A much better approach is to introduce hysteresis which only turns the LED back off again once the temperature has dropped by a sufficient value to stop oscillation at the borderline values.

FreeRTOS Creating Tasks Example Source Code

```
/* Kernel includes. */
#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include "xadcps.h"
/* Standard demo includes. */
#include "partest.h"

/* Priorities at which the tasks are created. */
#define mainQUEUE_RECEIVE_TASK_PRIORITY		( tskIDLE_PRIORITY + 2 )
#define mainQUEUE_SEND_TASK_PRIORITY		( tskIDLE_PRIORITY + 1 )

/* The rate at which data is sent to the queue.  The 200ms value is converted
to ticks using the portTICK_PERIOD_MS constant. */
#define mainQUEUE_SEND_FREQUENCY_MS			( 200 / portTICK_PERIOD_MS )

/* The number of items the queue can hold.  This is 1 as the receive task
will remove items as they are added, meaning the send task should always find
the queue empty. */
#define mainQUEUE_LENGTH					( 1 )

/* The LED toggled by the Rx task. */
#define mainTASK_LED						( 0 )

/*-----------------------------------------------------------*/

/*
 * The tasks as described in the comments at the top of this file.
 */
static void prvQueueReceiveTask( void *pvParameters );
static void prvQueueSendTask( void *pvParameters );
void adc_config(XAdcPs *XADCInstPtr, u16 XAdcDeviceId);


/*
 * Called by main() to create the simply blinky style application if
 * mainCREATE_SIMPLE_BLINKY_DEMO_ONLY is set to 1.
 */
void main_blinky( void );

/*-----------------------------------------------------------*/

/* The queue used by both tasks. */
static QueueHandle_t xQueue = NULL;
```

```
#include "xadcps.h"
//XADC info
#define XPAR_AXI_XADC_0_DEVICE_ID 0
static XAdcPs  XADCInst; //XADC
/*-----------------------------------------------------------*/

void main_blinky( void )
{
	/* Create the queue. */
	xQueue = xQueueCreate( mainQUEUE_LENGTH, sizeof( uint32_t ) );

	adc_config(&XADCInst,XPAR_AXI_XADC_0_DEVICE_ID );

	if( xQueue != NULL )
	{
		/* Start the two tasks as described in the comments at the top of this
		file. */
		xTaskCreate( prvQueueReceiveTask,				/* The function that
implements the task. */
					"Rx",
	/* The text name assigned to the task - for debug only as it is not used by the kernel. */
					configMINIMAL_STACK_SIZE,		/* The size
of the stack to allocate to the task. */
					NULL,
	/* The parameter passed to the task - not used in this case. */
					mainQUEUE_RECEIVE_TASK_PRIORITY,  /* The priority
assigned to the task. */
					NULL );
	/* The task handle is not required, so NULL is passed. */

		xTaskCreate( prvQueueSendTask, "TX", configMINIMAL_STACK_SIZE, NULL,
mainQUEUE_SEND_TASK_PRIORITY, NULL );

		/* Start the tasks and timer running. */
		vTaskStartScheduler();
	}

	/* If all is well, the scheduler will now be running, and the following
	line will never be reached.  If the following line does execute, then
	there was either insufficient FreeRTOS heap memory available for the idle
	and/or timer tasks to be created, or vTaskStartScheduler() was called from
	User mode.  See the memory management section on the FreeRTOS web site for
	more details on the FreeRTOS heap http://www.freertos.org/a00111.html.  The
	mode from which main() is called is set in the C start up code and must be
	a privileged mode (not user mode). */
	for( ;; );
}
```

```
/*-----------------------------------------------------------*/

static void prvQueueSendTask( void *pvParameters )
{
TickType_t xNextWakeTime;
unsigned long ulValueToSend = 100UL;

	/* Remove compiler warning about unused parameter. */
	( void ) pvParameters;

	/* Initialise xNextWakeTime - this only needs to be done once. */
	xNextWakeTime = xTaskGetTickCount();
	printf("task tx");
	for( ;; )
	{
		/* Place this task in the blocked state until it is time to run again. */
		vTaskDelayUntil( &xNextWakeTime, mainQUEUE_SEND_FREQUENCY_MS );
		ulValueToSend = XAdcPs_GetAdcData(&XADCInst, XADCPS_CH_TEMP);
		/* Send to the queue - causing the queue receive task to unblock and
		toggle the LED.  0 is used as the block time so the sending operation
		will not block - it shouldn't need to block as the queue should always
		be empty at this point in the code. */
		xQueueSend( xQueue, &ulValueToSend, 0U );
	}
}
/*-----------------------------------------------------------*/

static void prvQueueReceiveTask( void *pvParameters )
{
unsigned long ulReceivedValue;
unsigned long ulExpectedValue = 43000;

	/* Remove compiler warning about unused parameter. */
	( void ) pvParameters;
	printf("task rx");
	for( ;; )
	{
		/* Wait until something arrives in the queue - this task will block
		indefinitely provided INCLUDE_vTaskSuspend is set to 1 in
		FreeRTOSConfig.h. */
		xQueueReceive( xQueue, &ulReceivedValue, portMAX_DELAY );
		printf("Raw Value = %lu", ulReceivedValue);
		/*  To get here something must have been received from the queue, but
		is it the expected value?  If it is, toggle the LED. */
		if( ulReceivedValue == ulExpectedValue  )
		{
			//vParTestToggleLED( mainTASK_LED );
			vParTestSetLED(mainTASK_LED, 1);
```

```
                        ulReceivedValue = 0U;
                }
        }
}

void adc_config(XAdcPs *XADCInstPtr, u16 XAdcDeviceId)
{

//      u32 IntrStatus;
        XAdcPs_Config *ConfigPtr;       //xadc config
//      u32 TempRawData;
//      float TempData;
        printf("task xadc");
        //XADC initilization
        ConfigPtr = XAdcPs_LookupConfig(XAdcDeviceId);
        //adc set up
        XAdcPs_CfgInitialize(XADCInstPtr,ConfigPtr,ConfigPtr->BaseAddress);
        //stop sequencer
        XAdcPs_SetSequencerMode(XADCInstPtr,XADCPS_SEQ_MODE_SINGCHAN);
        //disable alarms
        XAdcPs_SetAlarmEnables(XADCInstPtr, 0x0);

        //configure sequencer to just sample internal on chip parameters
        XAdcPs_SetSeqInputMode(XADCInstPtr, XADCPS_SEQ_MODE_SAFE);
        //configure the channel enables we want to monitor
        XAdcPs_SetSeqChEnables(XADCInstPtr,XADCPS_CH_TEMP|XADCPS_CH_VCCINT|XADCPS_C
H_VCCAUX|XADCPS_CH_VBRAM|XADCPS_CH_VCCPINT|
                        XADCPS_CH_VCCPAUX|XADCPS_CH_VCCPDRO);

}
```

Asymmetric MultiProcessing

So far everything we have looked at on the PS side of the Zynq has used just the one core (Core 0) however the PS side contains two cores and for many applications to get the maximum performance of the Zynq we will want to use both these cores. Using both of the cores on the Zynq is referred to as Asymmetric Multiprocessing (AMP) this can involve any of the following combinations

- Different Operating Systems on Core 0 and Core 1
- Operating System on Core 0, Bare Metal on Core 1 (or vice versa)
- Bare Metal on both cores executing different programmes

There are two kinds of multicore processing known as symmetric and asymmetric however before we define the difference between symmetric and asymmetric multiprocessing I think we first have to define what multiprocessing is

“Multiprocessing is the use of more than one processor within a system, this can allow the execution of more than one instruction at the same time however, it does not necessarily have to”

The difference between symmetric and asymmetric multiprocessing is

- Symmetric Multiprocessing, uses a number of software tasks run concurrently by distributing the load across a number of cores
- Asymmetric Multiprocessing, uses specialised processors or applications execution on identical processors for specific applications or tasks

Over the next few blogs we are going to be looking at AMP on the Zynq, this will be based around initially two bare metal applications each running a different application on each core.

When one runs AMP on the Zynq one must consider the Zynq processor cores have a mixture of both private and shared resources. Both processors have private L1 instruction and data caches, timers and watchdogs along with share interrupt controllers (with both shared and private interrupts). However interrupts on the Zynq is not as straight forward as each core in the PS is capable of interrupting either just itself, the other processor or both processors using software interrupts which are distributed via the Interrupt Controller Distribution.

However the Zynq also has a large number of shared resources of which common examples include IO peripherals, On Chip Memory, the Interrupt Controller Distributor, the L2 Cache and system memory located within the DDR memory.

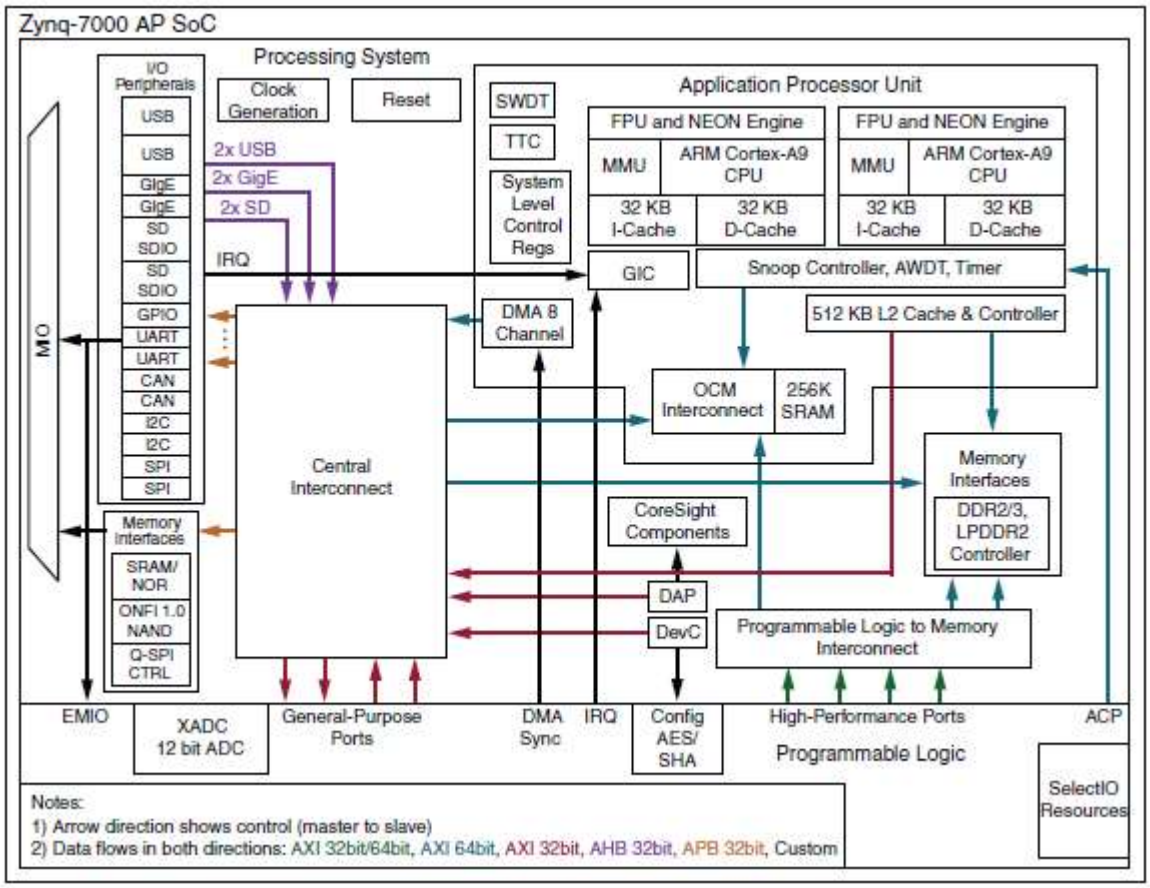


Figure 133 The Zynq Architecture

As we will be running the two cores from DDR memory we must take great care to segment the address regions used by each processor this can be determined via the linker scripts for each application if we fail to do this applications could interfere with each other's operation.

We will also have to make some modifications to auto generated files by SDK to get the system up and running the first step will be to modify the first stage boot loader in line with XAPP1079 which examines bare metal / bare metal AMP.

It is my intention to make a system which is initially very simple and which we can expand upon once up and running. The first application will have the Core 0 communicating with the user over RS232 while core 1 runs a pattern upon the LEDS connected to the MicroZed IO Carrier Card. Two application running without interaction.

This will enable us to progress to looking at how we can communicate between processors using the on chip memory along with how we can share resources between processors.

Asymmetric MultiProcessing First Steps

In my last blog I introduced the concept of bare metal Asymmetric Multiprocessing using both of the processors to execute bare metal programmes on both cores, I hope you are sitting very comfortably as this blog may be a little long but by the end we will have our AMP system up and running. However while there are a few steps involved to get AMP up and running it is actually very simple and straight forward and certainly nothing to be afraid of.

The key aspect required to get AMP running on the Zynq is a boot loader that will look for a second executable file after loading the first into the memory. Unfortunately the Vivado design suite I am using 2014.1 (I recently moved house and have no internet connection so have not been able to download the latest yet) does not support AMP when it generates a first stage boot loader. Therefore to get this example up and running I will be using the modified FSBL and modified standalone OS provided as part of Xilinx Application note XAPP1079 (The source files are available here http://www.xilinx.com/support/documentation/application_notes/xapp1079-amp-bare-metal-cortex-a9.pdf)

Having downloaded the zip file the first stage is to extract these in to your desired working directory and to rename the folder called SRC to design. These files contain both a modified FSBL and a Modified Standalone OS which we need to us, as such we need the SDK to be aware of them therefore the next step is to update your SDK repository such that it is aware of their existence. This is straight forward within SDK under the Xilinx tools menu select repositories and then new, navigating to the directory location <your working directory>\app1079\design\work\sdk_repo as shown below

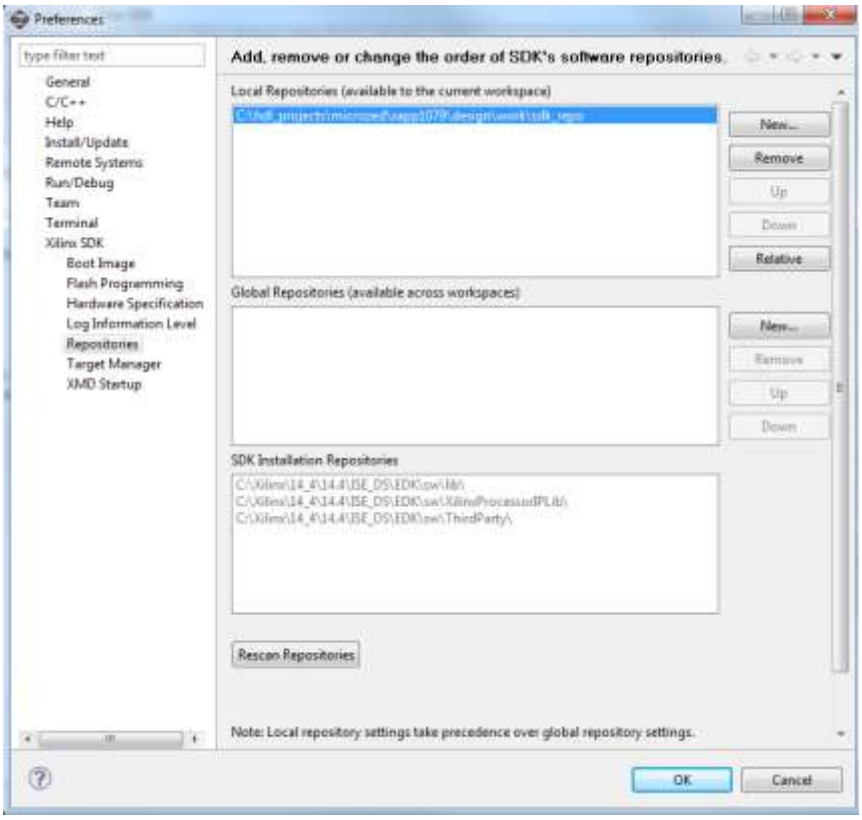


Figure 134 Adding the AMP repository

Having added in the repositories the next stage is to generate the following

- AMP first stage boot loader
- Core 0 application
- Core 1 application

For each of these we are going to generate a Board Support Package.

The first thing to do is create a new FSBL, select file new application project enables us to create a FSBL project which supports AMP, this is no different to what we have done before however we will be selecting the Zynq FSBL for AMP template in place of the of the Zynq FSBL template.

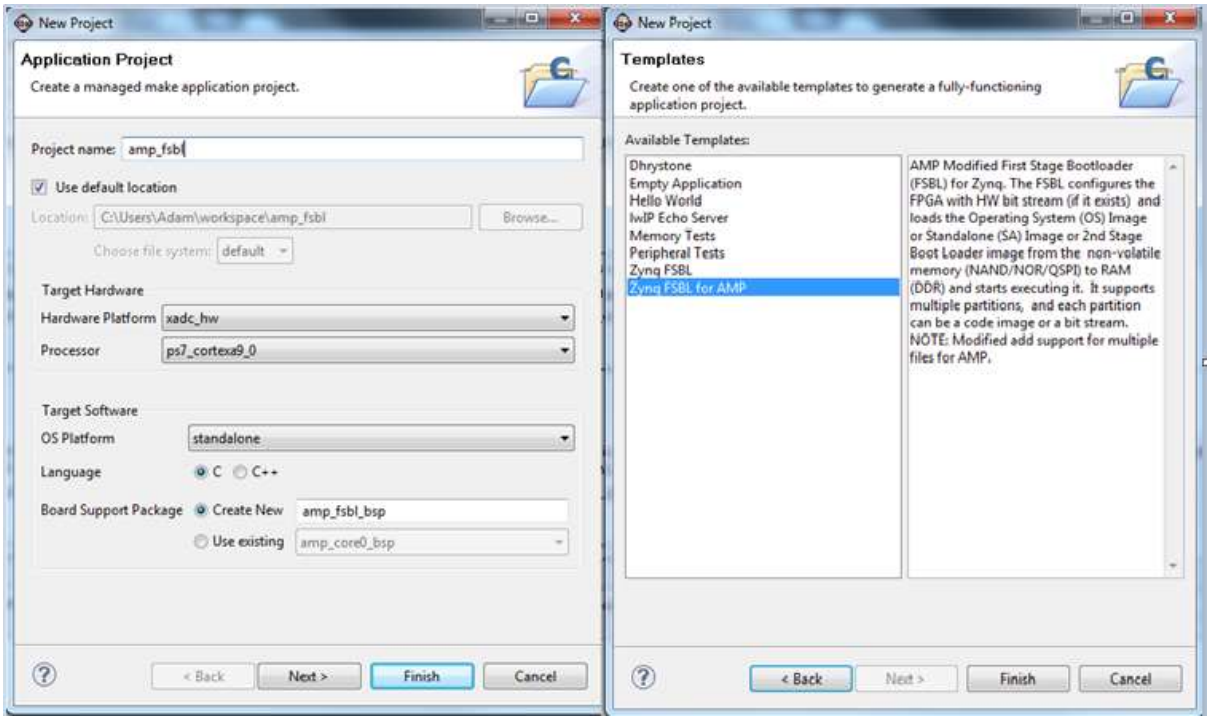


Figure 135 Creating the Zynq FSBL for AMP

Following the creation of the AMP FSBL we need to create the application for the first core, this again is very simple to do as we have done this many times before, be sure to select core 0 and the standalone OS and allow it to create its own BSP.

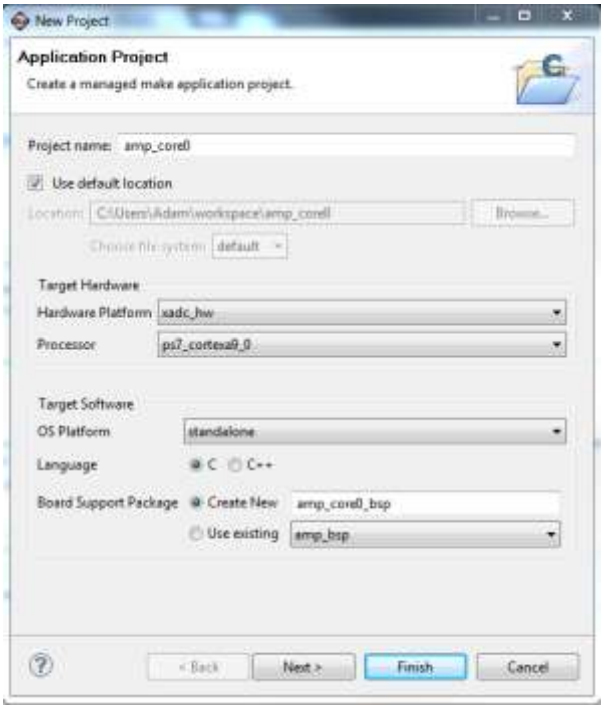


Figure 136 The creating the first application project

Once we have created this application we need to correctly define the location with DDR memory the application will execute, to do this we edit the linker script as below to show the DDR base address and size. This is important as if we do not correctly segment the DDR memory for core 0 and core 1 applications we risk one in advertently corrupting the other.

Available Memory Regions

Name	Base Address	Size
ps7_ddr_0_S_AXI_BASEADDR	0x00100000	0x00100000
ps7_ram_0_S_AXI_BASEADDR	0x00000000	0x00030000
ps7_ram_1_S_AXI_BASEADDR	0xFFFF0000	0x0000FE00

Figure 137 Updating the DDR Memory size

Having done this we can write the application we wish to execute on core 0, as this is the core which is in charge within the AMP system and must start the execution of core 1 we need to include the following section of code within the application. This disables the Cache on the On Chip Memory and writes the start address of the core 1 programme to an address core 1 will access once core 0 executes the Set Event (SEV) command this will cause core 1 to start executing its programme.

```c
#include <stdio.h>
#include "xil_io.h"
#include "xil_mmu.h"
#include "xil_exception.h"
#include "xpseudo_asm.h"
#include "xscugic.h"

#define sev() __asm__("sev")
#define CPU1STARTADR 0xffffffff0
#define COMM_VAL  (*(volatile unsigned long *)(0xFFFF0000))


int main()
{

    //Disable cache on OCM
    Xil_SetTlbAttributes(0xFFFF0000,0x14de2);           // S=b1 TEX=b100 AP=b11, Domain=b1111, C=b0, B=b0
    Xil_Out32(CPU1STARTADR, 0x00200000);
    dmb(); //waits until write has finished
    sev();
```

Figure 138 CPU0 starting CPU 1

The next step is to create a BSP for core1 as we want to use the modified standalone OS (standalone_amp) which prevents re initialisation of the PS Snoop Control Unit (SCU) we cannot allow automatic generation of the BSP as we create the project like we did for core 0. Be sure to select core 1 in the CPU selection options.

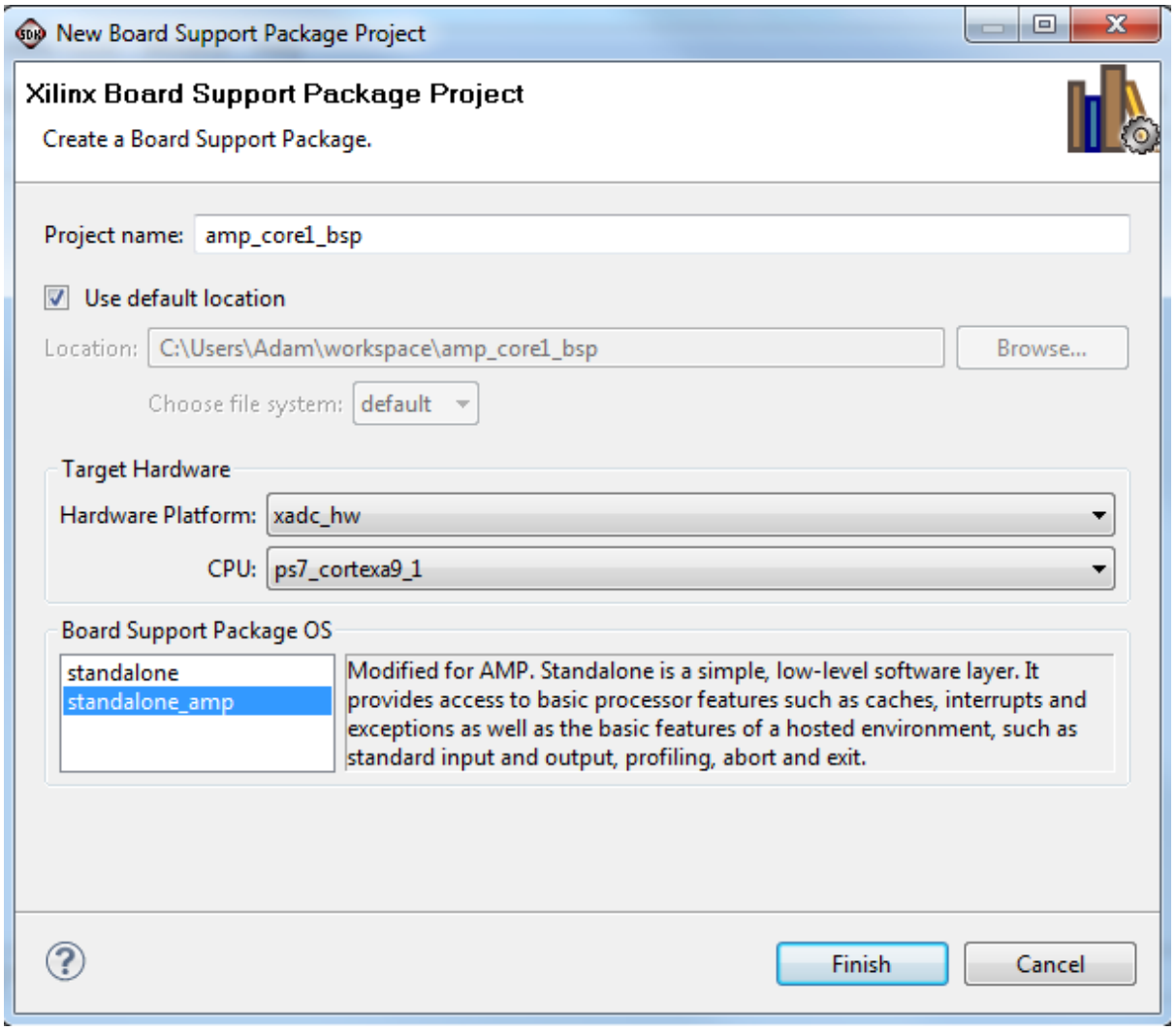


Figure 139 BSP for CPU1

Now we have created the BSP for core 1 we need to modify the settings of the BSP before we can progress to creating the application programme we want to run upon core 1. This is very simple and requires the addition of an extra complier flag of –DUSE_AMP=1 to the configuration for drivers section of the BSP.

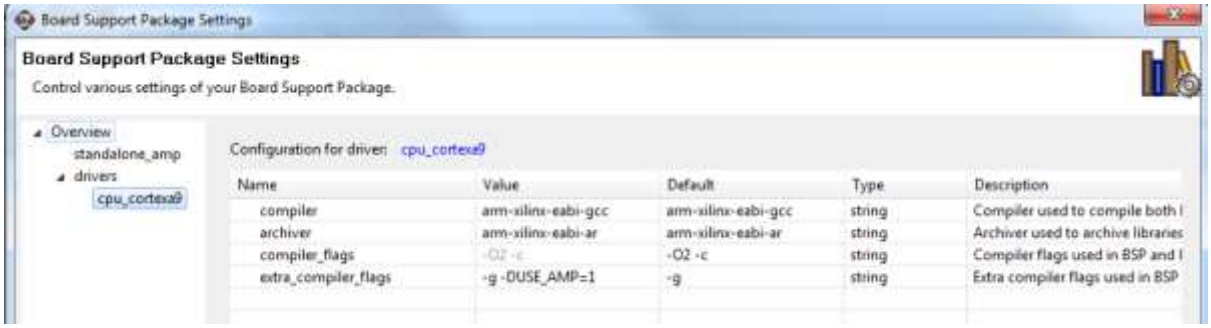


Figure 140 Setting the complier flags

With this completed we are free to create the application for core 1 be sure to select core 1 as the processor and use the BSP we just created.

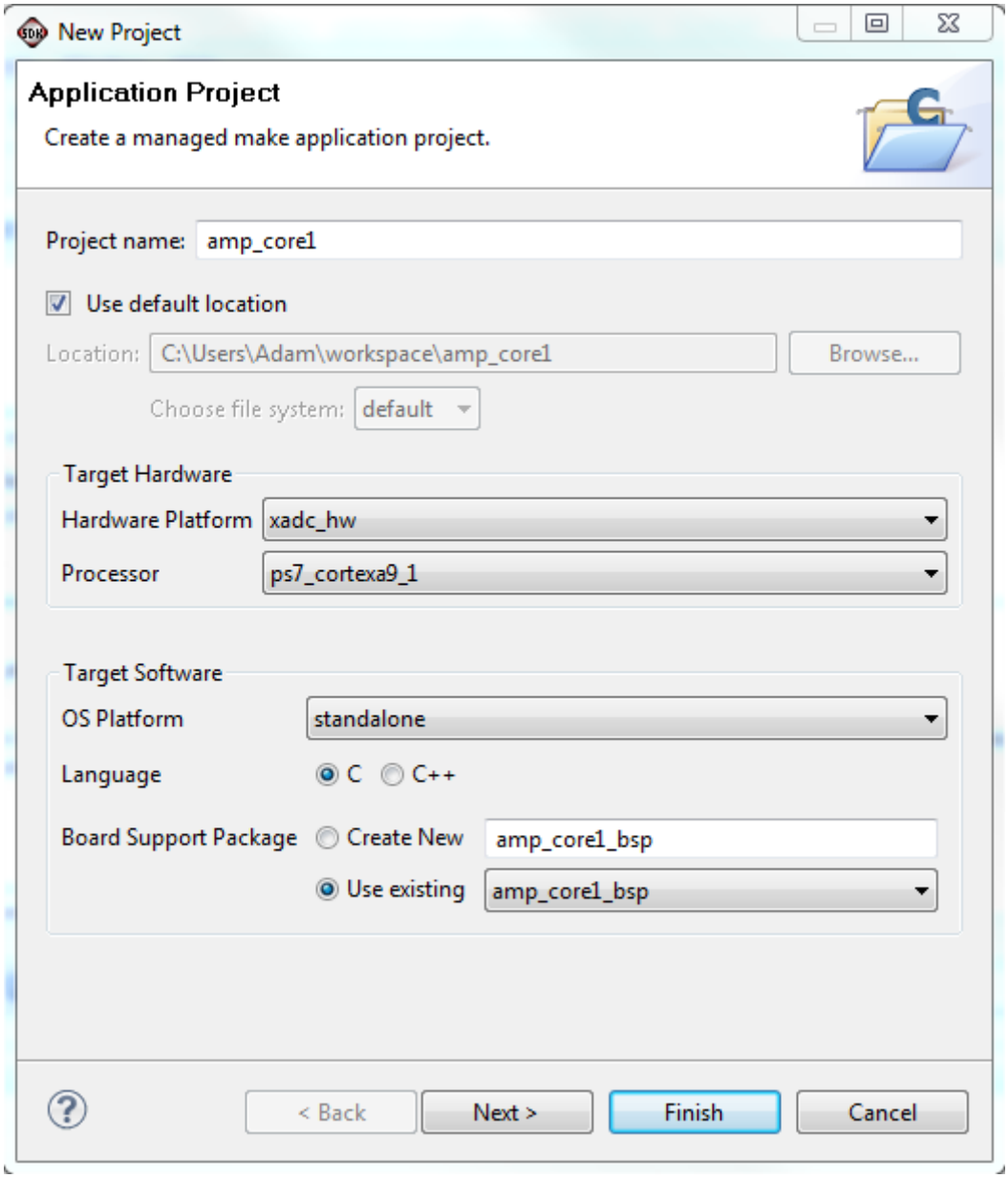


Figure 141 Creating CPU1 application

Again having created the new application we need to again define the correct memory locations within the DDR memory from which the core 1 programme will execute this is achieved by editing the linker script for the application for core 1.

Available Memory Regions

Name	Base Address	Size
ps7_ddr_0_S_AXI_BASEADDR	0x00200000	0x00100000
ps7_ram_0_S_AXI_BASEADDR	0x00000000	0x00030000
ps7_ram_1_S_AXI_BASEADDR	0xFFFF0000	0x0000FE00

Figure 142 Allocating CPU1 Memory regions

As with the first core within this application we must also disable the cache on the On Chip Memory as we will be using this in later blogs to communicate between the two processors.

Once we have completed our applications and built the projects we should now be in possession of the following

- AMP FSBL ELF
- Core 0 ELF
- CORE 1 ELF
- Bit file defining the configuration of the device.

To enable the Zynq to boot from your selected configuration memory and we need a .bin file, we also need a bif file which defines the files to be used to create this bin file and the order in which they go. Rather than use the create Zynq boot image within SDK we will be using a ISE command prompt and bat file provided as part of XAPP 1079 under the directory\design\work\bootgen this directory contains a bif file and a cpu1_bootvec.bin which is used as part of the modified FSBL to stop it looking for more applications to load.

To generate the bin file we need to copy the three generated ELF files to the bootgen directory and edit the BIF file to ensure the elf names within the bif file are correct.

```
the_ROM_image:
{

    [bootloader] amp_fsbl.elf
                 download.bit
                   amp_cpu0.elf
                   app_cpu1.elf

  //write start vector address 0xFFFFFFF0 with 0xFFFFFF00
  //This load address triggers fsbl to continue
  [load = 0xFFFFFFF0] cpu1_bootvec.bin
}
```

Figure 143 Editing the BIF File

Once this has been achieved we can open an ISE command prompt navigate to the bootgen directory and run the createboot.bat which will create the boot.bin file.

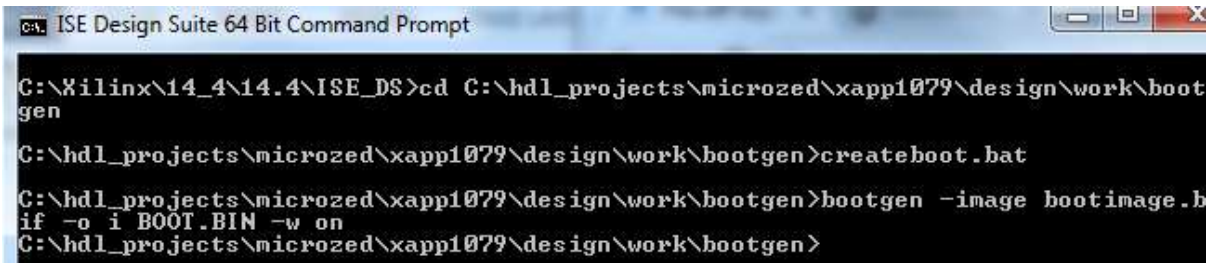


Figure 144 Generating the boot image

This file can then be downloaded into the non-volatile memory on your Zynq and booting the device will result in both cores starting and executing their respective programmes.

Asymmetric MultiProcessing The Software

The software I have up and running on both cores is very simple as I hope that it will allow me to show you how we can get the two cores communicating via the OCM. However, at the moment the software is doing simple things which deserve some discussion such that we have a baseline to move on from.

The software currently running in the example from last week performs the following tasks

- Core 0, is the master and starts the execution of core 1 it also uses the UART to print out a message to a terminal programme at a fixed delay. This delay does not use the any timers etc. although it could easily use the private timer for this and I shall do so in the future so we can show the two private timers in use at the same time.
- Core 1, once it is started by core 1 it initialises it resources and outputs a toggling pattern on the eight LED's on the Microzed Carrier IO card I have connected the MicroZed too. To achieve this we need to use CPU1 private timer and enable interrupts via the GIC.

These applications are in no way linked or share the same resource however as we progress we will want these applications to be able to do just that.

The application running upon core 0 is very simple following the start-up of core 1 it prints out a simple message in a loop forever as such this uses UART 0

```
int main()
{
    int delay;

    //Disable cache on OCM
    Xil_SetTlbAttributes(0xFFFF0000,0x14de2);           // S=b1 TEX=b100 AP=b11, Domain=b1111, C=b0, B=b0
    SetupIntrSystem(&IntcInstancePtr);
    print("CPU0: writing startaddress for cpu1\n\r");
    Xil_Out32(CPU1STARTADR, 0x00200000);
    dmb(); //waits until write has finished

    print("CPU0: sending the SEV to wake up CPU1\n\r");
    sev();

    while(1){

        for( delay = 0; delay < OP_DELAY; delay++)//wait
        {}
        print("CPU 0\n\r");
    }

    return 0;
}
```

Figure 145 Simple Core 0 Programme

However, as we intend to use the interrupt controller on core 1 we must first configure it on core 0 using the code below as part of the core 0 application.

```
static int SetupIntrSystem(INTC *IntcInstancePtr)
{
    int Status;

    XScuGic_Config *IntcConfig;
    IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
    if (NULL == IntcConfig) {
        return XST_FAILURE;
    }
    Status = XScuGic_CfgInitialize(IntcInstancePtr, IntcConfig,
                    IntcConfig->CpuBaseAddress);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
            (Xil_ExceptionHandler)INTC_HANDLER,
            IntcInstancePtr);
    Xil_ExceptionEnable();

    return XST_SUCCESS;
}
```

Figure 146 Configuring the GIC

The code for the second core is a little more complex as we are using the GPIO block within the PL side of the device to drive the LEDs on the MicroZed Carrier Card. As with all other interfaces Xilinx standalone OS provides a simple set of drivers for this using **`#include`** `"xgpio.h"` This differs slightly from the xgpio_ps.h file we have used previously which is used to drive MIO / EMIO GPIO connected to the PS side of the device. However in this instance I wanted to show we could use GPIO in the PL side of the device as well

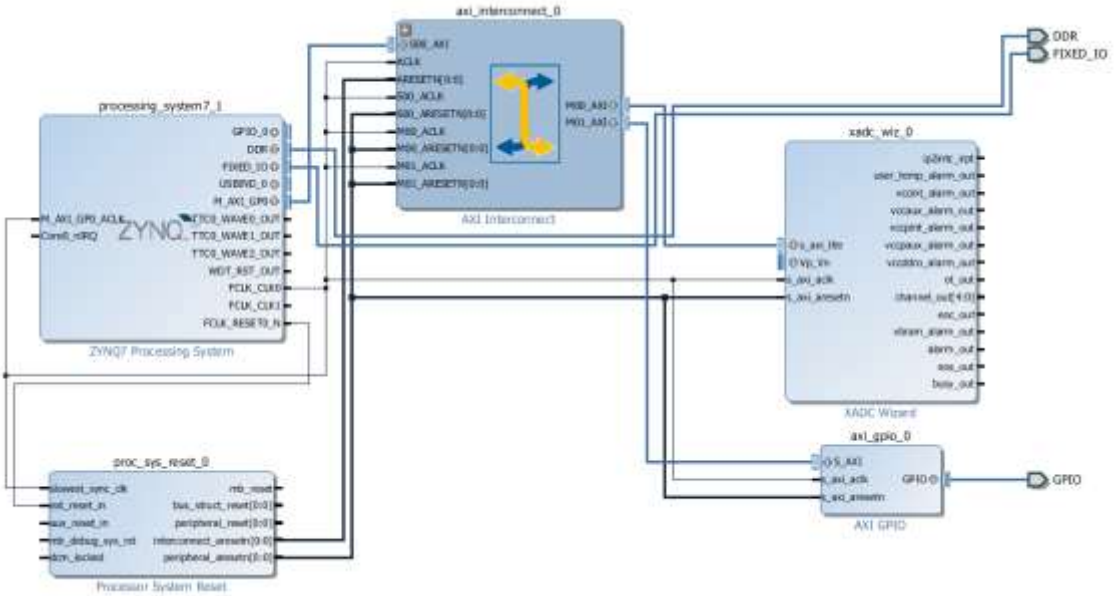


Figure 147 The Zynq system we are working with

To ensure that we can see the toggling of the LED's we will be using core1 private timer, this is identical to using the private timer on core 0 as we have done in the past.

Before the programme begins to execute its main application we need to disable the cache on the On Chip Memory (OCM), initialise the GPIO, Initialise the Private timer and configure the interrupt controller such that the private timer interrupts can be used to toggle the LEDS just as we would with any Zynq application (with the exception of disabling the cache).

```
static void TimerIntrHandler(void *CallBackRef)
{


    XScuTimer *TimerInstancePtr = (XScuTimer *) CallBackRef;
    XScuTimer_ClearInterruptStatus(TimerInstancePtr);
    //load timer
    XScuTimer_LoadTimer(&Timer, TIMER_LOAD_VALUE);
    //start timer
    if (toggle == 0 ){
        toggle = 1;
        XGpio_DiscreteWrite(&Gpio,CHANNEL,0x55);
    }
    else
    {
        toggle = 0;
        XGpio_DiscreteWrite(&Gpio,CHANNEL,0xAA);
    }
    XScuTimer_Start(&Timer);



}
```

Figure 148 The ISR on CPU1

Once we have done this we can then begin to write the rather simple interrupt service routine which will toggle the LED when the private timer elapses and restarts it such that the effect will continue for ever. I have selected alternate patterns of AA and 55for the LEDS as this exercises them all and is slightly different to all on all off approach.

```
CPU0: writing startaddress for cpu1
CPU0: sending the SEV to wake up CPU1
CPU1: starting
CPU1: configured
CPU 0
CPU 0
CPU 0
CPU 0
CPU 0
```

Figure 149 Resultant Output

Asymmetric MultiProcessing On Chip Memory

Having got the AMP up and running and looked at the basic software we have running on the processors, I want to begin exploring how we can use the On Chip Memory to communicate between the cores, however in the previous 48 instalments of this blog we have not discussed the OCM except in passing. As we plan to use the OCM going forward it is important we understand more about what it is and how it functions etc. as like most things on the Zynq it is much more powerful than its simple name suggests.

The Zynq has 256 KB of on chip SRAM which can be accessed from one of four sources as can be seen in the figure of the Zynq PS below

1. From either Core via the Snoop Control Unit
2. From the Programmable Logic using the AXI Accelerator Coherency Port via the Snoop Control Unit
3. From the Programmable Logic using the AXI High Performance port via the On Chip Memory Interconnect
4. From the Central Interconnect again via the On Chip Memory

With these different sources which can read and write the OCM is especially important we understand its operation in detail before we use it.

As there are multiple sources of accessing the OCM it is only sensible that a form of arbitration and priority is defined. As the lowest latency is required by the Snoop Control unit as that is either a processor core or the AXI ACP a SCU read has the highest priority followed by the SCU write and then OCM interconnect read and write. The user can invert the priority between the SCU write and the OCM interconnect access by setting the SCU Write Priority Low in the On Chip Memory Control Register.

The OCM itself is organised as 128 bit words, split into four 64 KB regions which are at different locations within the PS address space. The initial configuration has the first three 64KB blocks arranged at the start of the address space and the last 64KB block towards the end of the address space this can be seen in the linker files below for both applications (Core 0 top, Core 1 bottom)

Available Memory Regions

Name	Base Address	Size
ps7_ddr_0_S_AXI_BASEADDR	0x00100000	0x00100000
ps7_ram_0_S_AXI_BASEADDR	0x00000000	0x00030000
ps7_ram_1_S_AXI_BASEADDR	0xFFFF0000	0x0000FE00

Available Memory Regions

Name	Base Address	Size
ps7_ddr_0_S_AXI_BASEADDR	0x00200000	0x00100000
ps7_ram_0_S_AXI_BASEADDR	0x00000000	0x00030000
ps7_ram_1_S_AXI_BASEADDR	0xFFFF0000	0x0000FE00

Figure 150 OCM Locations for each CPU Core

Note the ps7_ram__XXX and ps7_ram_1 relate to the memory regions 0 and 1 and not the cores.

The OCM can be re organised to be mapped to be completely contiguous at the end of the address space is required by using the System Level Control Registers, On Chip Memory Configuration register and setting the appropriate RAM Hi bits.

While the OCM is single port memory it is possible for it to emulate a dual port memory provided the accesses are 128 bit aligned and consist of even burst multiples of AXI commands. To achieve this level of throughput one must use the DMA engine to move efficiently large data volumes.

We can also error protect the on chip memory if we are using it for a particularly critical application, this is enabled using the On Chip Memory Control Parity control register, this enables you to set odd or even parity individually on each of the 16 bytes which make up the 128 bit word stored at each On Chip Memory address. Through this register we can also configure how we handle parity errors e.g. by issuing the OCM shared interrupt (number 35) or sending an AXI read error (SLVERR) when an error is detected on a read.

Table 7-3: **PS and PL Shared Peripheral Interrupts (SPI)**

Source	Interrupt Name	IRQ ID#	Status Bits (mpcore Registers)	Required Type	PS-PL Signal Name	I/O
APU	CPU 1, 0 (L2, TLB, BTAC)	33:32	spi_status_0[1:0]	Rising edge	~	~
	L2 Cache	34	spi_status_0[2]	High level	~	~
	OCM	35	spi_status_0[3]	High level	~	~

Figure 151 PS and PL Shared Interrupts

Asymmetric MultiProcessing OCM Communication

The demo for this will use CPU0 to communicate over the UART link to a laptop which will receive an eight bit ASCII value over the UART. Once received this eight bit value will be transferred into the selected on chip memory address which is shared between the two processors. CPU1 will then upon each times its private timer expires read this memory address and set its GPIO and hence the LED's connected to it to the display the received ASCII pattern. This visual confirmation displayed upon the LED's ensures the correct value has been passed between the CPU's.

Obviously the first thing to do is determine the memory address to be used as we are going to be passing a simple 8 bit unsinged integer we will only need one address location. I have chosen to use the highest 64K of OCM and as such the cache has been disabled for this memory location we can do this in both processors applications using the command

```
Xil_SetTlbAttributes(0xFFFF0000,0x14de2);
```

I will be using the memory address 0XFFFF0000 to transfer my data byte from CPU Zero to CPU One. There are a number of ways that we can do this however the two most common are the following

The first is to use the generic Xilinx IO functions to read and write from the selected memory address these functions are contained within Xil_IO.h and allow for storing and accessing 8, 16 or 32 bit char, short or int within the CPU address space. Using these functions just requires the address you wish to access and the value you wish to store there if it is a write.

```
Xil_Out8(0xFFFF0000,0x55);

read_char = Xil_In8(0xFFFF0000);
```

A better way to use this to ensure the addresses are both targeting the same location within the on chip memory, especially if different people are working on the different CPU programme is to have a common header file which contains macro definitions of the address of interest for that particular transfer for instance.

```
#define LED_PAT 0xFFFF0000
```

The second approach is for both programmes to use access the memory location using a pointer instead. We can do this by defining the pointer which points to a constant address normally in C we do this using a macro

```
#define LED_OP (*(volatile unsigned int *)(0xFFFF0000))
```

Again we can use another macro definition if we want for the address to ensure that the address is common between both application programmes. This approach does not then require the use of the XIL IO libraries and instead allows simple access via the pointer.

Using both of these approaches results in the communication from my laptop via the UART to CPU0 and then into CPU1 to be displayed on the LEDS being successful as can be seen on the images below (tastefully captured in black and white to stop the glare from the LEDs LSB is far right)

```
CPU0: writing startaddress for cpu1

CPU0: sending the SEV to wake up CPU1

CPU1: starting

CPU1: configured

value 61
```

Figure 152 LED and UART values in agreement

Of course when we are communicating as we are across the OCM we must remember that the register can be updated more often than we can currently see due to the use of the timer within CPU1 to trigger the read of the memory.

Interrupts and AMP

The Zynq has 16 software generated interrupts for each core, these can be used to interrupt itself, the other core or both cores. For this example we will be using core zero to generate an interrupt to inform core one that there has been an updated LED pattern received.

Using software interrupts is not to different from using hardware interrupts except of course in how we trigger them.

Having decided upon which of the software interrupts to use, we have 16 to choose from we can define the software interrupt number

```
#define SW_INT_ID                    0
```

We need to declare this within the code for both of our cores or within a shared header file,it would be embarrassing to be issuing interrupts on one core only to be looking for a different one on the other core.

The next things we need to do within core 1 which is the core that will be receiving the interrupt is to connect the interrupt to the controller and then enable it. We can do this within the existing function SetupInterruptSystem.

Connecting the interrupt is achieved by using the function XScuGIC_Connect this connects the defined interrupt to the GIC, defines the interrupt service routine and the callback. As in this instance we have no peripheral instance pointer to act as the callback like we would with a timer for example we can use the GIC Instance as call back.

```
Status = XScuGic_Connect(GicInstancePtr, SW_INT_ID,
                         (Xil_ExceptionHandler)sgi_handler,(void *)GicInstancePtr);
if (Status != XST_SUCCESS) {
            print("error setting SGI");
            return XST_FAILURE;
}
XScuGic_Enable(GicInstancePtr, SW_INT_ID);
```

Finally having connected the selected software interrupt to the GIC we need to enable it such that we react to events which occur upon it.

With the interrupts connected and enabled the next stage is to write the interrupt service routine (ISR) this will simply read the on chip memory and clear the interrupt such that the next is not masked.

```
static void sgi_handler(void *CallBackRef)
{
    u32 INT;
    INT = Xil_In32(0xF8F0010C);
    cpu0 = Xil_In8(LED_PAT);
}
```

The read of address 0xF8F0010C reads the interrupt acknowledge register ICCIAR which clears the interrupt.

The data read from the OCM is then displayed upon the LEDS attached to the GPIO within the PL. This prevents us from missing a value like the approach in the previous blog.

Having completed all we needed to do within core 1 we must correctly configure core 0 as well to issue the interrupt.

This is achieved using the XScuGic_SoftwareIntr function provided within xscugic.h this will issue a software interrupt to the identified core

XScuGic_SoftwareIntr(<GIC Instance Ptr>, <SW Interrupt ID>, <CPU Mask>)

The GIC Instance Ptr is the same as we have used to configure the interrupt controller, while the SW Interrupt ID is the one we previously declared within the macros. The CPU mask in this case is CPU 2 which is corresponds to Core 1.

When all this was built, and the bin file generated as previously explained for the AMP Zynq I achieved a system which worked as I had wanted with the interrupt issued by core 0 and received by core 1.

Interrupts and AMP Source Coode

Core 0 Application

```
#include <stdio.h>
#include "xil_io.h"
#include "xil_mmu.h"
#include "xil_exception.h"
#include "xpseudo_asm.h"
#include "xscugic.h"


#define sev() __asm__("sev")
#define CPU1STARTADR 0xffffffF0
#define LED_PAT 0xFFFF0000

#define INTC                XScuGic
#define INTC_DEVICE_ID  XPAR_PS7_SCUGIC_0_DEVICE_ID
#define INTC_HANDLER    XScuGic_InterruptHandler
#define OP_DELAY        100000000
#define SW_INT_ID             0

static void sgi_handler(void *CallBackRef);
static int  SetupIntrSystem(INTC *IntcInstancePtr);
INTC   IntcInstancePtr;
int main()
{
      //int delay;
      unsigned int inchar;
      unsigned int newlr;
      int Status;

    //Disable cache on OCM
    Xil_SetTlbAttributes(0xFFFF0000,0x14de2);           // S=b1 TEX=b100
AP=b11, Domain=b1111, C=b0, B=b0
    SetupIntrSystem(&IntcInstancePtr);
    print("CPU0: writing startaddress for cpu1\n\r");
    Xil_Out32(CPU1STARTADR, 0x00200000);
    dmb(); //waits until write has finished

    print("CPU0: sending the SEV to wake up CPU1\n\r");
    sev();


    while(1){

      inchar = getchar();
      newlr = getchar();
      //for( delay = 0; delay < OP_DELAY; delay++)//wait
      //{}
      xil_printf("value %x",inchar);
      Xil_Out8(LED_PAT,inchar);

      //Xil_Out32(0xF8F01f00,0x02008000);
      Status = XScuGic_SoftwareIntr(&IntcInstancePtr,SW_INT_ID,
XSCUGIC_SPI_CPU1_MASK);
      if (Status != XST_SUCCESS) {
                  print("error triggering SGI");
                  return XST_FAILURE;
      }
```

```
      print("\n\r setting up SW Interrupts\n\r");
    }

    return 0;
}

static void sgi_handler(void *CallBackRef)
{
      u32 INT;

      //BaseAddress = (u32)CallbackRef;
      INT = Xil_In32(0xF8F0010C);
      print("SGI****");
}

static int SetupIntrSystem(INTC *IntcInstancePtr)
{
      int Status;
    int callback=0;
      XScuGic_Config *IntcConfig;
      IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);

      Status = XScuGic_CfgInitialize(IntcInstancePtr, IntcConfig,
                                     IntcConfig->CpuBaseAddress);
      if (Status != XST_SUCCESS) {
            return XST_FAILURE;
      }
      Xil_ExceptionInit();
      Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                  (Xil_ExceptionHandler)XScuGic_InterruptHandler,
                  IntcInstancePtr);

      Status = XScuGic_Connect(IntcInstancePtr, SW_INT_ID,

      (Xil_ExceptionHandler)sgi_handler,(void *)IntcInstancePtr);
      if (Status != XST_SUCCESS) {
                  print("error setting SGI");
                  return XST_FAILURE;
            }
      XScuGic_Enable(IntcInstancePtr, SW_INT_ID);
      Xil_ExceptionEnableMask(XIL_EXCEPTION_IRQ);

      return XST_SUCCESS;
}
```

Core 1 Application

```
#include "xparameters.h"
#include <stdio.h>
#include "xil_io.h"
#include "xil_mmu.h"
#include "xil_cache.h"
#include "xil_exception.h"
#include "xscugic.h"
#include "sleep.h"
#include "xscutimer.h"
#include "xgpio.h"


#define LED_PAT 0xFFFF0000
void        Xil_L1DCacheFlush(void);
```

```
extern u32 MMUTable;

#define GPIO_DEVICE_ID          XPAR_AXI_GPIO_0_DEVICE_ID
#define TIMER_DEVICE_ID         XPAR_PS7_SCUTIMER_0_DEVICE_ID
#define INTC_DEVICE_ID          XPAR_PS7_SCUGIC_0_DEVICE_ID
#define TIMER_IRPT_INTR         XPS_SCU_TMR_INT_ID
#define TIMER_LOAD_VALUE        0xFFFFFFFF
#define CHANNEL                       1
#define OP                            0xff
#define ADDR                          XPAR_AXI_GPIO_0_BASEADDR
#define SW_INT_ID                     0
static XGpio Gpio; //GPIO
static XScuGic Intc; //GIC
static XScuTimer Timer;//timer

volatile unsigned int cpu0;

static void TimerIntrHandler(void *CallBackRef);
static int SetupInterruptSystem(XScuGic *GicInstancePtr, XScuTimer
*TimerInstancePtr, u16 TimerIntrId);
static void sgi_handler(void *CallBackRef);
int main()
{


      XScuTimer_Config *TMRConfigPtr;     //timer config

    //Disable cache on OCM
      Xil_SetTlbAttributes(0xFFFF0000,0x14de2);           // S=b1 TEX=b100
AP=b11, Domain=b1111, C=b0, B=b0
      print("CPU1: starting\n\r");

      //GPIO Initilization
      XGpio_Initialize(&Gpio, GPIO_DEVICE_ID);
      XGpio_SetDataDirection(&Gpio, CHANNEL, 0x00);
      XGpio_DiscreteWrite(&Gpio,CHANNEL, 0xff);

    //timer initialisation
    TMRConfigPtr = XScuTimer_LookupConfig(TIMER_DEVICE_ID);
    XScuTimer_CfgInitialize(&Timer, TMRConfigPtr,TMRConfigPtr->BaseAddr);
    XScuTimer_SelfTest(&Timer);

      //load the timer
    XScuTimer_LoadTimer(&Timer, TIMER_LOAD_VALUE);

    SetupInterruptSystem(&Intc, &Timer,TIMER_IRPT_INTR);

    XScuTimer_Start(&Timer);
    print("CPU1: configured\n\r");
    while(1){


    }

    return 0;
}

static void sgi_handler(void *CallBackRef)
{
      u32 INT;
      INT = Xil_In32(0xF8F0010C);
```

```
	cpu0 = Xil_In8(LED_PAT);
}


static int SetupInterruptSystem(XScuGic *GicInstancePtr, XScuTimer
*TimerInstancePtr, u16 TimerIntrId)
{
int Status;

		XScuGic_Config *IntcConfig; //GIC config
		Xil_ExceptionInit();
		//initialise the GIC
		IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
		XScuGic_CfgInitialize(GicInstancePtr, IntcConfig,
					IntcConfig->CpuBaseAddress);
	    //connect to the hardware
		Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,

	(Xil_ExceptionHandler)XScuGic_InterruptHandler,
				GicInstancePtr);

		//set up the timer interrupt
		XScuGic_Connect(GicInstancePtr, TimerIntrId,
					(Xil_ExceptionHandler)TimerIntrHandler,
					(void *)TimerInstancePtr);

		//enable the interrupt for the Timer at GIC
		XScuGic_Enable(GicInstancePtr, TimerIntrId);

		Status = XScuGic_Connect(GicInstancePtr, SW_INT_ID,

	(Xil_ExceptionHandler)sgi_handler,(void *)GicInstancePtr);
		if (Status != XST_SUCCESS) {
				print("error setting SGI");
				return XST_FAILURE;
		}
		XScuGic_Enable(GicInstancePtr, SW_INT_ID);

		//enable interrupt on the timer
		XScuTimer_EnableInterrupt(TimerInstancePtr);
		// Enable interrupts in the Processor.
		Xil_ExceptionEnableMask(XIL_EXCEPTION_IRQ);

		return XST_SUCCESS;
	}

static void TimerIntrHandler(void *CallBackRef)
{
	//unsigned int cpu0;

	XScuTimer *TimerInstancePtr = (XScuTimer *) CallBackRef;
	XScuTimer_ClearInterruptStatus(TimerInstancePtr);
	//load timer
	XScuTimer_LoadTimer(&Timer, TIMER_LOAD_VALUE);
	//start timer

	XGpio_DiscreteWrite(&Gpio,CHANNEL,cpu0);

	XScuTimer_Start(&Timer);
```

```
}
```