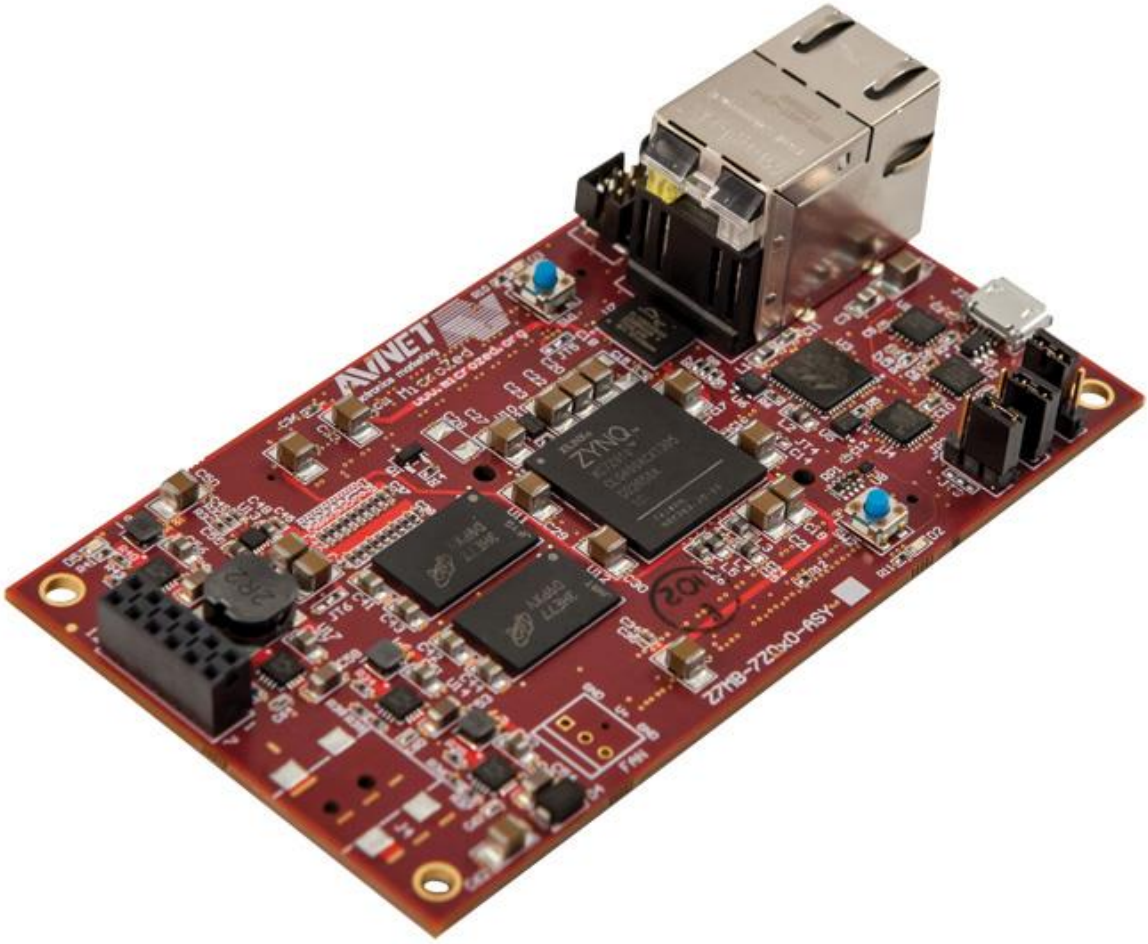


Zynq-7000 Workshop for Beginners

Version 1.1 (MicroZed Board - Vivado 2015.4)

Rich Griffin
Xilinx Embedded Specialist, Avnet Memec-Silica EMEA

Introduction

Welcome to the Zynq beginners workshop. The purpose of this document is to give you a hands-on introduction to the Zynq-7000 SoC devices, and also to the Xilinx Vivado Design Suite. Throughout the course of this guide you will learn about the Zynq SoC solution step-by-step, and gain the knowledge and experience you need to create your own designs.

How does it work?

However you want it to! But the following notes might help:

- Please feel free to experiment beyond the guidance of the exercises – it is the fastest way to learn. This is not going to be a regimented "follow the steps in the instructions" style of material. The aim is to learn by doing, getting it wrong, and then fixing it.
- Please work on your own if you can, you'll learn more by doing than by watching someone else use the tools. If you do work with someone else, please try to get equal hands-on time. You won't learn much by watching.
- Everyone will be work at their own speed when following this guide. Depending on previous knowledge and experience, some people will progress quickly and others will take longer to master the concepts. The idea is to progress through the exercises, learning how to use the Xilinx embedded processor tools simply by "getting your hands dirty". Each exercise will introduce new concepts of embedded processor design and will enable you to learn new skills. The exercises will become more challenging as the workshop progresses, and will gradually offer less step-by-step guidance.
- If you're working though this material at an organised event, there are more exercises listed here than most people will be able to cover during one day. This has been done quite intentionally so that you will have something more to explore in your own time if you so choose. If you are working through this material alone, then you can obviously set your own schedule and pace.
- Have fun!

Prerequisites

These exercises have been designed to guide you through the usage of the Xilinx tools at every step of the way, and teach the basic principles of the code that we'll be writing in C and VHDL. That said, it would be advantageous if you have a basic understanding of both languages, just so that you don't feel like you're living in a world of peculiar-looking hieroglyphics.

Agenda

- Exercise 1 - Getting something (anything!) working
  - "I just want to see the board work. How do I do it?"
  - Use of the Block Diagram tool.
  - "print" is your friend. We will learn how to use it for debugging.

- Exercise 2 - Using drivers to flash an LED
  - What is a driver?
  - Where do I find the supplied drivers?
  - Using the GPIO peripheral and the supplied drivers, control some pins to flash LEDs on the board.

- Exercise 3 - Debugging
  - Why is this important?
  - How does it help us?
  - What options do we have for debugging?
  - Use the debugger to step through your Flashing LED design.

- Exercise 4 – Expanding your design into the programmable logic (PL)
  - Add soft peripherals to the AXI interfaces.
  - Interfacing between the PS and PL.
  - Connecting interrupts and assigning IO pin locations.

- Exercise 5 - Making your design interactive
  - How do we read inputs from the user?
  - User input driver functions.
  - Using a UART connection, make your software respond to characters sent via the UART.
  - Make an LED light on the board when a certain character is sent from the UART.

- Exercise 6 - Reading from and writing to memory
  - Use the "Xil_io" functions to write to and read from memory.
  - Why is this important?

- Exercise 7 - Timers (Polled mode)
  - Why are timers useful?
  - When do we use them?
  - Using a timer peripheral, flash an LED at a frequency of 1Hz.

- Exercise 8 - Timers (Interrupt mode)
  - What is an interrupt?
  - Why are they important?
  - Make a system that will light an LED in response to a user input, but at the same time print messages to the UART at a frequency of 1Hz. Neither task is permitted to noticeably interfere with the other.

- Exercise 9 – Talking to external components
  - Using a PMOD expansion module, we will experiment with using the SPI protocol to talk to an ambient light sensor.
  - This exercise will discuss the basics of SPI, and will show how to read from and write to an external device to expand the capabilities of your design.
  - We will then implement an interrupt based design to send and receive data from the external board via SPI.

- Exercise 10 – Autonomous Boot
  - Using an SD flash memory card, we will make the MicroZed board boot automatically.

Preparation

During this workshop we shall be using an evaluation board to demonstrate some of the principles behind designing an embedded processor system on Xilinx SoC devices. The board we will use is a low-cost "MicroZed 7010" development kit, available from the http://microzed.org website. The board features a Zynq 7010 device, and also provides a few system components to enable users to experiment with various aspects of their embedded designs. These include a UART, external (DDR3) memory, a push button, an LED, an Ethernet PHY & socket, and a PMOD expansion socket.

The MicroZed board does not have its own in-built JTAG configuration, so a Xilinx or Digilent JTAG cable will be required to download the Programmable Logic (PL) configuration to the Zynq SoC. A power supply for the board is not required, because the board can be powered via the USB connection. We shall be using the RS232 UART in our experiments, which is connected to the same USB socket on the board via a Silicon Labs CP2104 USB-UART bridge. You will therefore need a USB to MicroUSB cable to connect the board to your laptop.

I'm new to Xilinx. What tools do I need?

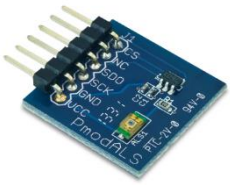
The Xilinx design tools are designed to cater for both hardware and software engineers. The Xilinx FPGA and Zynq SoC devices are extremely flexible and so there is a lot of functionality in the toolset, which is spread across different applications.

Vivado – The top level design environment for the hardware designer. Use this tool to create the contents of your Programmable Logic, and to create the embedded processor section of the design. We will use Vivado to configure our settings for the Zynq "Processing System" section of the design.

SDK – The Software Development Kit. A tool for software engineers, allowing the user to develop C code, generate BSPs, create boot images, and test their code using the debugger.

What hardware do I need?

This workshop uses the MicroZed board and the cables which are supplied in the box. For Exercise 9, you will also need a "PMOD ALS" ambient light sensor board, which contains a light sensor from Vishay® Semiconductor (TEMT6000X01) and an A-to-D converter from Texas Instruments® (ADC081S021).



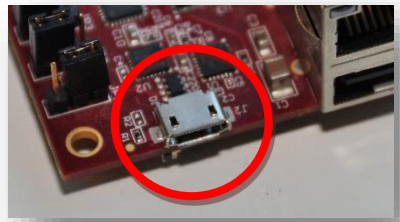
Exercise 1 - Getting something (anything!) working

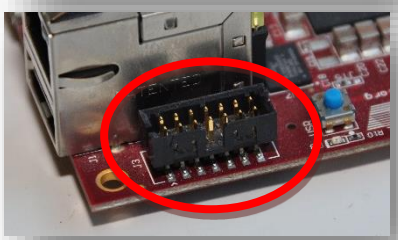
This exercise has a triple purpose. Firstly, it will check that Xilinx tools have been correctly installed. The second and main part of the exercise will be to build a very basic processor system using the Xilinx Vivado tools, to help create a design which will then be synthesised, implemented, and downloaded to the demonstration board via a programming cable. Lastly, the exercise will enable the user to test the MicroZed board to check that it is correctly powered and configured. If successful, the user will see messages displayed on the terminal connected to the UART socket on the board. As with all development boards, there are supplied reference designs which are

provided for the user to see the board in action out of the box. However, the purpose of this workshop is for you to do it, so we'll create our own design rather than use the supplied reference designs.

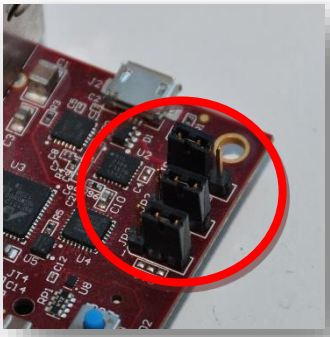
1. Connect the USB cable between your PC and the "J2" micro-USB socket on the demonstration board. You will find this socket to the left of the Ethernet socket. Later on, we will use the Terminal utility built into the Xilinx Software Development Kit to view the UART output from the board.

2. Connect the JTAG programming cable / pod to the "J3" socket next to the right of the Ethernet socket, and the other end to your PC.





3. Check the jumper settings for "JP1", "JP2", and "JP3" in the bottom-right corner of the board. The jumpers for JP2 and JP3 should be set to connect pins 2 & 3, and the jumper for JP1 should be set to connect pins 1 & 2.



4. Remove the SD card from the socket on the back of the board, if one is inserted. Check that the green "Power Good" LED (D5) on the demo board is illuminated. (A red LED (D3) may also be illuminated).

5. Depending on whether your PC has previously been used with this board, you may see an automated installation wizard run on your PC for the USB UART drivers and JTAG cable drivers. If this happens, permit the wizard to look for files on the internet and the cable should be installed automatically. (N.B. The wizard may re-run several times before the cable is correctly installed. This is normal so please persevere)

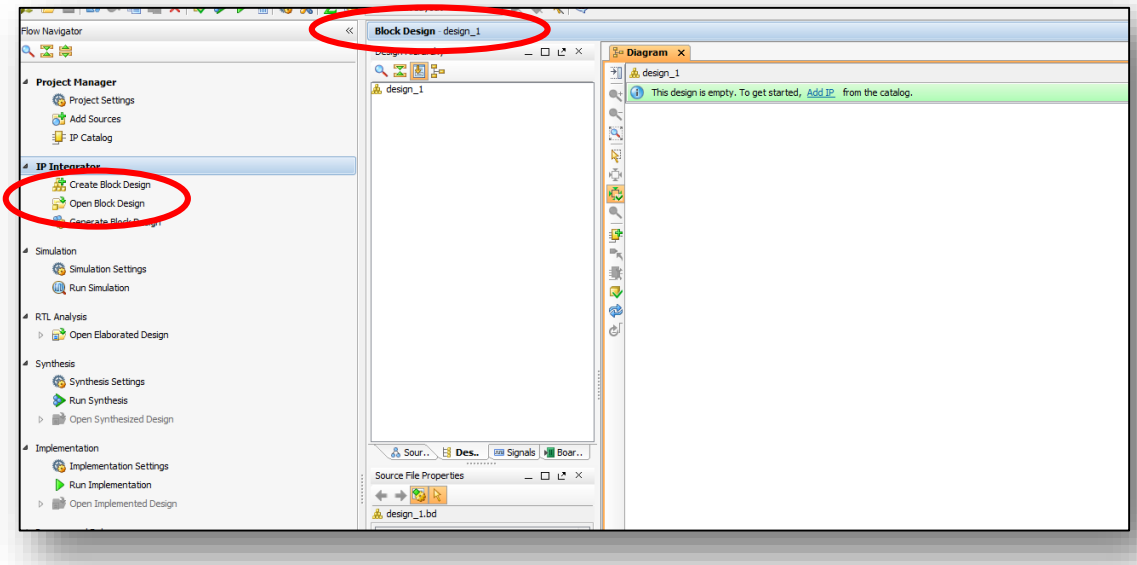
6. Run the Xilinx Vivado tool (Start → All Programs → Xilinx Design Tools → Vivado 2015.4 → Vivado 2015.4). *You can use a different version of the Vivado tools, but please be aware that some of the screens you will see may differ from the ones shown in this document.*

   6.1. From the menu, choose to "Create new project" and then click "Next".
   6.2. Give the project a name of your choice, and specify a convenient location on your PC for the project files. Leave the "Create Project Subdirectory" box ticked, and click "Next". Choose "RTL Project" from the list of project types and click "Next".
   6.3. Choose "VHDL" as your target language from the drop down list. If you have a particular preference for Verilog, you can choose it here but the guidance later on in this workshop will need to be modified using your own skills. Click "Next" several times until you see the "Default Part" screen.
   6.4. Click the "Boards" option in the "Select:" area. Choose "em.avnet.com" from the "Vendor" filter drop-down.
   6.5. Check to see which version of the MicroZed you have by examining the chip markings on the Zynq device. You should see either "XCZ7010" or "XC7Z020", denoting the size of the programmable logic region of the device.
   6.6. Choose the correct version of the "MicroZed 70x0 Board" from the list of known boards. Click "Next", and then click "Finish".

Note: If you do not see the MicroZed boards listed, then you will need to install the MicroZed board files for your Vivado tools. Please visit http://microzed.org/support/documentation/ , choose the MicroZed board, and then follow the instructions in the "Board Definition Files" section near the bottom of the page.

7. You will now see an empty Vivado project. Maximise the Vivado window if it is not already filling the screen, so that you can more easily explore this tool.
   7.1. On the left of the screen you will see the Flow Navigator pane. This is where you control the flow of the design and where you will click to run Synthesis, Implementation, and generate a configuration bitstream.
   7.2. In the middle of the screen near the top, you will see the "Sources" pane. This is where you will see design sources that you will add to your project. Currently there are no Design Sources in this project so the list will be empty. You will also see the design constraints lists here, which is also empty. Don't be confused by the "constrs_1" sub folder; this represents that you have a default "Constraint Set" present in the project, but there are no constraints files in it yet.
   7.3. On the right of the screen you will see the Project Summary pane. This is where you will see statistics about your project; how big it is, how many resources you are using in your Zynq device, and a summary from the various report files. There is little information in this pane at the moment, because we have not yet done anything.
   7.4. At the bottom of the screen you will see a series of tabs which show reports, errors, warnings, messages, a log of recent events, and the progress of any open designs in the "Design Runs" tab. There is also a TCL console in this area which allows power users to enter advanced custom commands.

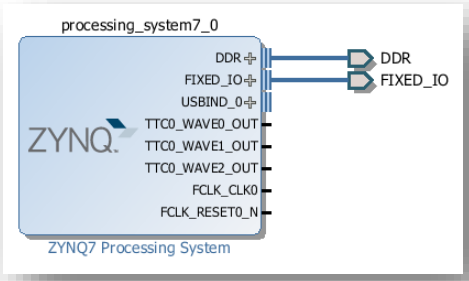
8. Xilinx Zynq-7000 devices offer users the ability to combine custom logic blocks with embedded processor systems. We are going to add a source file to the project which represents the Embedded Processor section of our design in block diagram form.
   8.1. On the left of the screen in the Flow Navigator pane, click the "Create Block Design" icon.
   8.2. Leave the name of the block diagram at the default setting, and click "OK".
   8.3. After a few seconds, the pane on the right of the screen will open into the Block Design mode (shown in the blue bar at the top of the pane), and an



empty block diagram will be shown. Note also the message in the centre of the screen, which is prompting you to start your design by adding some IP to the block diagram.

9. You will now configure the hardened Processing System section of the Zynq device.

   9.1. In the centre of the screen, click the "Add IP" button. If the message is not visible, you can open the "Add IP" window using its identical icon button on the left of the block diagram pane.
   9.2. Choose the "ZYNQ7 Processing System" entry from the list, either by scrolling down to locate it, or by typing the first few characters into the search box at the top of the box. Double-click the IP in the list to add it to the block diagram, or drag it to the block diagram with your mouse.
   9.3. Note the appearance of the green "designer assistance" bar at the top of the block diagram, which is now offering a "Run Block Automation" feature. Vivado has detected that there is an un-configured block on the diagram and is offering the user the chance to connect some required pins on the symbol. Click the "Run Block Automation" link in the designer assistance bar, and review the "processing_system7_0" instance name that appears in the list of permitted "automation" operations on the left of the window. Leave the "Apply Board Preset" box ticked, and leave the "Cross Trigger" settings disabled, and then click OK.
   9.4. Review how the DDR & Fixed IO connections are made automatically. The Vivado tools are 'board aware' and have also now made a number of changes to the internal configuration of the "Zynq7 Processing System" block. You will notice that a number of additional connections have appeared on the block including clocks, resets, timer outputs, and USB interfaces which allow the connection of additional signals in the design.



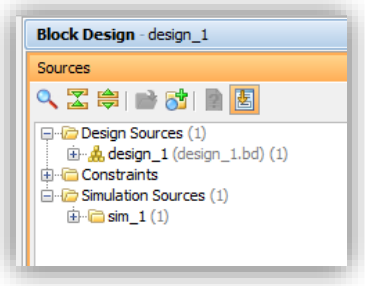
   9.5. Double click the "Zynq7 Processing System" symbol to open the customisation window. This is where changes can be made to the Processing System section of a Zynq device, and there are a large number of settings available, illustrated by a functional block diagram of the device. We are going to keep our design very simple at first, so we will not enable any AXI bus interfaces that connect to the Programmable Logic section of the device.
   9.6. The diagram of the Zynq device shown on this window is interactive; anything in bright green can be clicked to adjust the settings. Note that several of the I/O Peripherals in the list on the left of the diagram have tick marks next to them. These are the features of the MicroZed which have been automatically configured using the information in the Board Definition file. The tools have also correctly configured the DDR memory on the board, using the information from the Board Definition file. Feel free to explore the various settings in the GUI, but please resist the urge to change anything at this time. You can return

to the device Block Diagram at any time by clicking the "Zynq Block Design" page in the Page Navigator" section on the left of the window. Click "OK" to return to the block diagram.

9.7. This completes the settings for the embedded processing section of the design. Click the "Validate Design" icon to the left of the block diagram to check for any errors. You should see a "Validation successful" message.

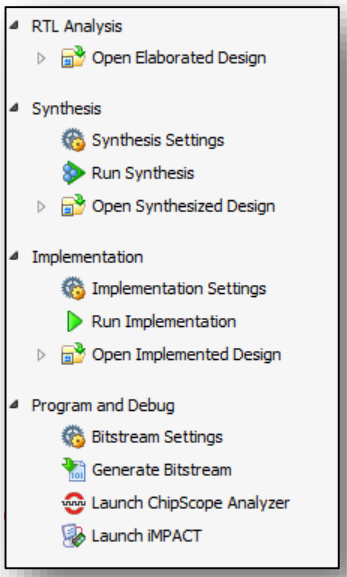
9.8. Save the block design by choose "File → Save Block Design" from the menus.

10. Click the "Sources" tab in the "Design Hierarchy" pane in the middle-left of the screen. You will now see your block design source (design_1) in the "Design Sources" folder, denoted by a pyramid stack of yellow boxes. Click the name of the embedded source that you have created (with its yellow icon and .bd file suffix) to select it. This module contains all of the settings that you configured in the block diagram editor, and you can return to it any time by double-clicking this source.



11. We must now create a top level VHDL / Verilog wrapper file to go around this block design. Normally this top level source file would be created manually by the user, but as a short-cut we can make the Vivado tools do this for us.

11.1. Right-click the block design source file (design_1) and choose "Create HDL Wrapper" from the resulting menu. Choose to "Let Vivado manage wrapper and auto-update", and click OK. The tools will automatically create a VHDL or Verilog file which will have the same name as your block design source, but with the suffix "_wrapper" appended to the name. If you wish, you can double click to open and review the HDL code for this top level file in the editing pane on the right of the screen. This wrapper will serve as the top level HDL file for our design.



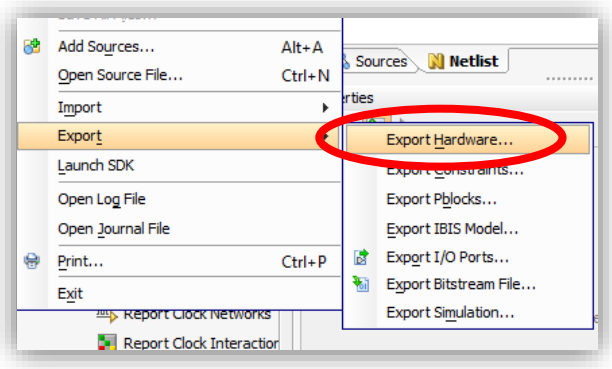
12. We will now synthesise, implement, and create a downloadable bitstream for this design.
    12.1. On the left of the screen in the "Flow Navigator" pane you will see menu items for "Run Synthesis", "Run Implementation" and "Generate Bitstream". We could run all of these steps in turn, but the Vivado tool is capable of managing the dependences automatically.
    12.2. We want to generate a downloadable bitstream to configure the Zynq device, so choose "Generate Bitstream" from the bottom of the Flow Navigator pane on the left of the screen. You may see a confirmation box asking you to confirm that you want to re-run the Synthesis and Implementation steps. You should agree to this and the flow will proceed. The Vivado tool will Synthesise, Implement, and finally generate the bitstream for your design. You can monitor the progress of this process as it moves from one stage to the next using the status indicator in the top-right corner of the screen (just below the Search box), but it may take a few minutes to complete. Grab a coffee! *(When you see the message "write_bitstream Complete" in the top right corner of the screen, you will be ready to continue)*
    12.3. Depending on whether you have changed the default settings in your Xilinx tools on a previous occasion, you may also see a pop-up box asking whether you wish to open the implemented design, view reports, or open the hardware manager. If you see this, choose to "Open Implemented Design". If you do not see this pop-up box, click "Open Implemented Design" in the "Flow Navigator" pane on the right of the screen.

13. That's it, you've now built the hardware and you have a Bitstream which can be used to configure the Programmable Logic (PL) section of the device. In this case the bitstream is just a wrapper, because all of the functionality in this first simple design is contained within the hardened Processing System (PS) section of the device. You will see a diagram of the Zynq device appear on the right of the screen. The boxes represent different design regions of the Programmable Logic, and the bright orange section towards the left of the image is the hardened Processing System which contains the memory controllers, common IO peripherals, and ARM processor cores.
    13.1. The settings you made using the block design tool were of huge importance, and all of this information has been included in our design. Almost all of these settings were applied automatically by the tool because we chose the "MicroZed Board" when we created the project. Vivado knows how to correctly configure the DDR controller, clocks, IO pins, and many other settings, so a huge amount of design time has been saved here. For custom

boards the user would need to manually configure these settings to match their board.

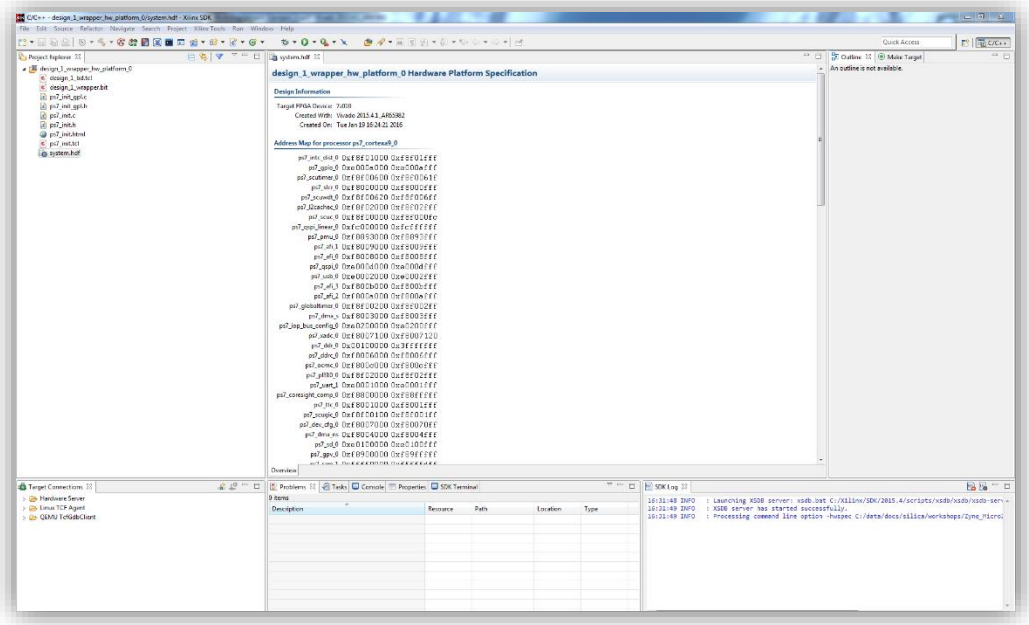
14. We now need to start writing some software to run on one of the ARM Cortex A9 processors, and to do that we must use the Software Development Kit (SDK).

15. Launch the SDK tool from within Vivado.
    15.1. Choose "File → Export → Export Hardware..." from the menus.
    15.2. Tick the "Include Bitstream" box from the pop up, leave the other settings as the defaults, and click OK.
    15.3. Choose "File → Launch SDK" from the menus, accepting the default settings.



After a few seconds you will see a new design tool appear. The Software Development Kit (SDK) is based upon the "Eclipse" tools which are fast becoming an industry standard for software development environments across the world. If you are new to Eclipse, please take a few moments to familiarise yourself with the layout of the SDK. On a Windows PC you may see a firewall warning at this stage, and you should select the default "Private networks" option and click "Allow Access".

15.4. On the left of the screen is the Project Explorer pane. This is where you will see all of your software projects, Board Support Packages, and Hardware Platforms listed, including all of the files which represent the project. At this stage you will see only a hardware platform, which is depicted by the yellow folder icon containing a microchip symbol. This hardware platform has been



automatically imported from the Vivado tool and represents the hardware settings which are applicable for the MicroZed Board that we are using today.

15.5. In the middle of the screen you will see the code and file editing pane. This is where you will make changes to source files, and edit your C code. At this stage you should see a .HDF file open (Hardware Description File), and this represents the hardware for the MicroZed board that has been imported. Feel free to explore the text content of this file; you will see that it contains all of the hardware base address and high address settings for the various peripherals on the Zynq device.

15.6. On the right of the screen is the Outline pane. This will provide an overview of your software code later on, but is blank for the moment.

15.7. At the bottom of the screen you will see a series of tabs that operate in much the same way as they did in the Vivado tool. The tabs show you a console which will provide messages, warnings, and status updates. There is a Tasks tab which shows you what the SDK is currently doing (perhaps in the background), and a Terminal which allows you to interact with serial ports on the board.

16. We will start our software development by creating a Board Support Package (BSP). BSPs contain drivers, libraries, and essentially anything else which will allow your software applications to access features on the hardware. In many cases it is the user's responsibility to create a BSP from scratch, or import one that someone else has created. The Xilinx tools differ massively from this way of working, and are capable of creating a BSP for you automatically. This is a huge benefit and will save a lot of time during your software development.

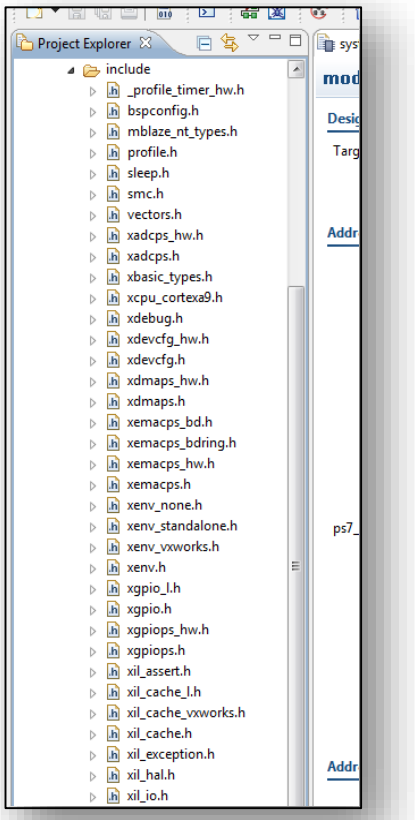
16.1. From the menus choose "File → New → Board Support Package". Accept all of the defaults and click "Finish".

16.2. A BSP settings window will appear, giving you the option to adjust what will be contained within the BSP. We are going to keep it simple, so please just click "OK".

16.3. In the Console tab at the bottom of the screen you will see a special utility called "Library Generator" calling the GNU compiler to create your BSP. You will ultimately see the message "Build Finished", which will indicate that the BSP is complete. That's it! You have just created a fully featured BSP, customised to the hardware in your design, which would normally require weeks of engineering time.

16.4. In the Project Explorer pane on the left of the screen you will see that your "standalone_bsp_0" BSP has appeared in the list of sources. Expand this project by clicking on the arrow to the left of the name and continue to expand the yellow folders until you see the "include" folder. Inside this folder you will see a list of header (.h) files which represent all of the drivers that you will need to develop your software.

16.5. As an example of the available drivers, open the "xuartps.h" header file by double-clicking it, and explore the contents of the file. You will find a selection of functions which allow you to configure and control the UART. Near the bottom of the file you will see functions that Send and Receive bytes to/from the UART. At this stage also review the "Outline" tab on the right of the screen. For large complex source files, the Outline tab can help you to navigate through the source file with ease. Try clicking a few of the items in the outline tab, and note how the view in the source code window jumps to that section of the code.



16.6. You will find that all of the function prototypes in the header file are documented and provided in C source code (in the associated "libsrc" directory of the BSP). This is a very useful resource for software developers, and will save huge amounts of time when writing your code.

16.7. Close the header file, and collapse the BSP folder to tidy up the Project Explorer pane.

17. We will now create a simple software application. Just like all good software engineers, our first application will simply write "Hello World" to the UART.

17.1. From the menus, choose "File → New → Application Project"

17.2. Give the project a name, such as "Exercise_01".

17.3. At the bottom of the window, choose the "Use existing" Board Support Package, and then select the BSP that you created a few moments ago. Click "Next".

17.4. Finally, choose the "Hello World" template from the list of examples and click "Finish".

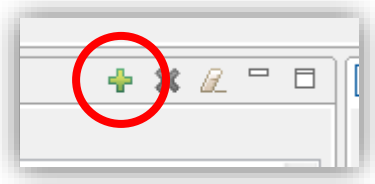
17.5. Monitor the Project Explorer pane and you will note that the new software project called "Exercise_01" appears on the left.

17.6. Expand the "Exercise_01" folder and descend into the "src" sub-folder. Double click to open the "helloworld.c" file and examine the contents. You will notice that the code is extremely simple and all of the usual complicated code that is often present to configure a processor appears to be missing. Actually it's not missing, but has been cleverly abstracted away by Xilinx to a different

location. *(You can see where the setup code is hiding later on, but for now please just believe me that it works!)*

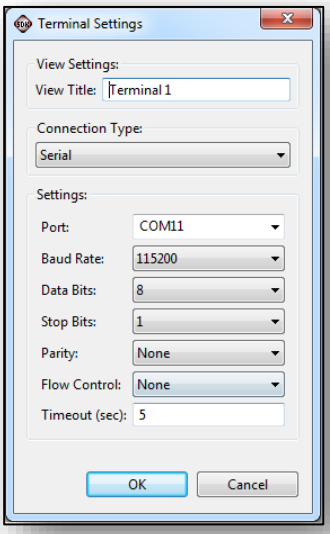
18. Check that the MicroZed board is powered, and that you can see the green "Power Good" LED illuminated.

19. We will now configure the built-in UART Terminal to listen to UART traffic coming from the board.



19.1. At the bottom of the SDK screen, click to activate the "SDK Terminal" tab.

19.2. In the SDK Terminal tab's menu bar, click the green "+" icon. The terminal settings will be shown in a small pop-up window.

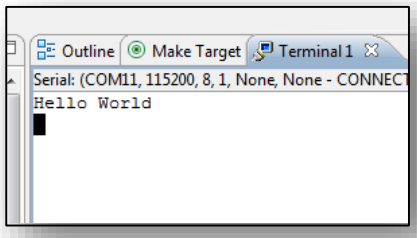


19.3. Set the COM port to the one that has been installed on your PC for the USB-UART. You may need to check the Device Manager in the Windows Control Panel to find out which one it is.

19.4. Set the baud rate to "115200", the Data Bits to "8", the Stop bits to "1", and both Parity and Flow Control to "None". Leave the rest of the settings as default and click OK. The connection to the UART will automatically be opened, and you should see a "Connected" message in the SDK Terminal tab.

20. In the Project Explorer pane, right-click on the "Exercise_01" project and choose "Run as ➔ Launch on Hardware (System Debugger)". You will see a warning that the FPGA configuration is not done on the target board. This is a very useful reminder because without configuring the programmable logic (PL) section of the device, your design may not work correctly. However in this instance we are only using the UART which is located in the fixed functionality "Processing System" (PS) region of the device. We can therefore click OK to bypass this warning. The application will be downloaded to the MicroZed board and it will execute on the first Cortex A9 processor core.

21. Depending on the precise window arrangement, SDK has a mildly irritating habit of deselecting the SDK Terminal tab when an application is downloaded, so you may need to re-select it at the bottom of the screen. Alternatively, you can click and drag the Terminal window to a different pane in SDK. The area occupied by the "Outline" tab, or the one to its left, is often a good place for it. You should see the "Hello World" message displayed in the Terminal window.



22. That's it! Although there were a lot of steps to explain in this first exercise, you have taken a Zynq device and configured it to run a simple software application, starting completely from scratch. Congratulations!

Further experiments

The "print" C code function can be used to display text to the standard output device (which in our case is the UART). The syntax for "print" is as follows:

```
print("Your line of text goes in here!\n\r");
```

The "\n\r" is a special tag which tells the STDOUT device to send a line feed (\n) to the UART, followed by a carriage return (\r). Therefore to send a blank line to the UART we would use the function call:

```
print("\n\r");
```

End of Exercise Challenge

Modify the "Exercise_01" software to send a series of your own messages to the UART, by using the "print" function. Experiment with using the "\n" "\r" tags individually and together.

When you have made your code modifications, the application can be run on the board using the "Right-click → Run as → Launch on Hardware (System Debugger)" method, the same as you did before.

Note: To add confusion here, some SDK Terminals will automatically add a "\r" when they detect a "\n". This can often lead to the assumption that there is no difference between them.

Exercise 2 - Using drivers to flash an LED

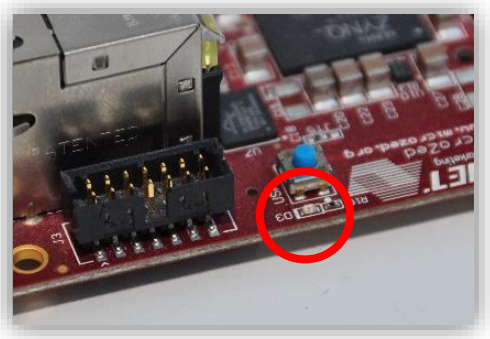
We have now verified that the board and the relevant connections are correct. In this exercise we will explore the use of software drivers to control an output pin on the Zynq-7000 device to drive an LED. As we saw at the end of Exercise 1, the Vivado tools have the ability to maintain several software applications designed for use on the same processor hardware, so we must first create a new software application in SDK, by choosing File → New → Application Project from the menus. Give the application the name of "Exercise_02", and select the existing "standalone_bsp_0" board support package in the same way that you did in the previous exercise. On the second screen, choose the "Hello World" template again. We could create an Empty Application, but it's actually easier to create a "Hello World" template and then modify it, because the template also includes some basic setup for the Zynq-7000 SoC.

Expand the new "Exercise_02" application in the Project Explorer pane, and then also expand the "src" folder. Right-click on the "hello_world.c" source file and choose "Rename" from the menu. Give the source file a new name, such as "exercise_02.c". Note that the SDK will guide you to choose an appropriate filename extension for this type of source file.

Open the "exercise_02.c" source file by double-clicking it. You will see that there are some comments at the top of the source file, and the basic code required to print a hello world message to the UART. Note also the "init_platform()" function; this is the basic setup for the UART that is useful to us. Modify the message printed to the UART by adjusting the print statement to a printf statement, and remove the declaration for the print function.

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

int main()
{
	init_platform();

	printf("Hello World\n\r");

	return 0;
}
```

This printf statement will provide us with a simple sanity check when we run the application later on, and reassure us that the application is executing correctly on the ARM processor core.

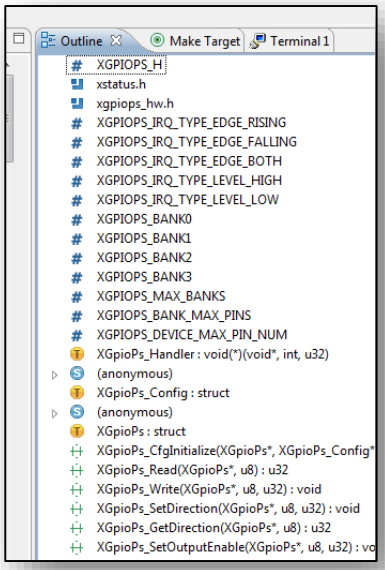


To flash an LED we first need to work out where the LEDs are on the MicroZed board, and to what they are connected. The MicroZed board Hardware User's Guide

(available from the http://microzed.org website) tells us that there is one user LED, labelled D3, and this is connected to package pin B14 (MIO 47) of the 54 MIO pins which is accessible on the "PS" (Processing System) side of the Zynq device.

We will now find the appropriate driver to use from the BSP that we created in Exercise 1. Expand the BSP in the Project Explorer, and navigate down to the contents of the "include" directory within the BSP. All Xilinx drivers begin with the letter "X" and then have a name that reflects their functionality. We're looking for the GPIO driver for pins connected to the PS, so we need to look for the "xgpiops.h" header file (X GPIO PS).

Open the "xgpiops.h" header file and review the code. You will probably become very rapidly overwhelmed by the quantity and complexity of the code. This is where the "Outline" pane becomes useful on the right of the screen. All #define statements (Macro definitions) in the code are listed with a "#" icon. All function prototypes are shown with circular icons with a green hatch pattern. All structs are shown with a circular blue "S" icon. All type definitions are shown with a yellow circular "T" icon. The Outline view is also interactive. Try clicking to select various items in the Outline pane, and you will see that the view in the editor window jumps to that section of the code.

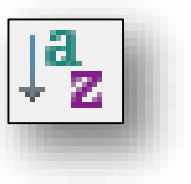


In order to use these functions in our application, we first need to tell the compiler to include them in our C code. Scroll to the top of your "exercise_02.c" file, and add the following lines near the other "#include" lines.

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

The second #include line (xparameters.h) will give you access to many of the predefined "#define" macros which are provided in the Xilinx BSP. Take a moment to view the contents of this file, and to do this we will introduce another feature of the SDK. Although we have added the #include line for xparameters.h, we might not have any idea where the file is located in the tree of sources. In fact it doesn't matter, because the SDK will find it for us. Hold down the "Ctrl" key on your keyboard and hover your mouse over the "#include" line in your source. Notice how the line of C has become a link. Click the link and the xparameters.h source file will be opened in the editor.

Scrolling through the file you'll find that there are lines which define a "Device ID" to each of the peripherals in the system. The one that interests us for this exercise is "XPAR_PS7_GPIO_0_DEVICE_ID" which you will find roughly halfway down the file. If you find it difficult to locate, you can use the "Outline" view to make this easier. If it is still too hard to find, then try sorting the Outline view alphabetically by toggling the "A-Z" sort feature in the Outline tab's header bar.



Xilinx drivers all use the same principles of operation and they require that the driver be initialised before use. To avoid confusion, we will explore the concepts of driver initialisation before we write any code.

All Xilinx drivers have a "struct" (structure) which holds all of the various setup values which will be needed by the peripheral. A "struct" is merely a collection of variables / data types, wrapped and bundled together in such a way that allows us to move many variables around using just one name.

Fruit My_Fruit_Collection

```
typedef struct {
        u8  Banana_Count;
        u16 Apple_Count;
        u32 Lemon_Count;
} Fruit;
```

Here we can see that we have a struct with an instance name of "My_Fruit_Collection", and it is of a data type called "Fruit".

The struct contains three variables, "Banana_Count", "Apple_Count", and "Lemon_Count". They are all of different data types (for no real reason whatsoever, but just to prove that it is possible!), which are unsigned integers of 8, 16, and 32 bits wide, respectively.

To declare an instance of this struct in C, we could use the following line of code.

```
Fruit My_Fruit_Collection;
```

It's important to recognise that by declaring an instance of a struct, we do not assign any values to the three variables inside it. They are only declared at this stage, but we've saved ourselves a lot of typing by grouping them together. Imagine that you were writing software to count all of the fruit in a supermarket; there are so many different types of fruit that you would need to write hundreds of lines of code just to declare the variables. Untidy, and very time consuming!

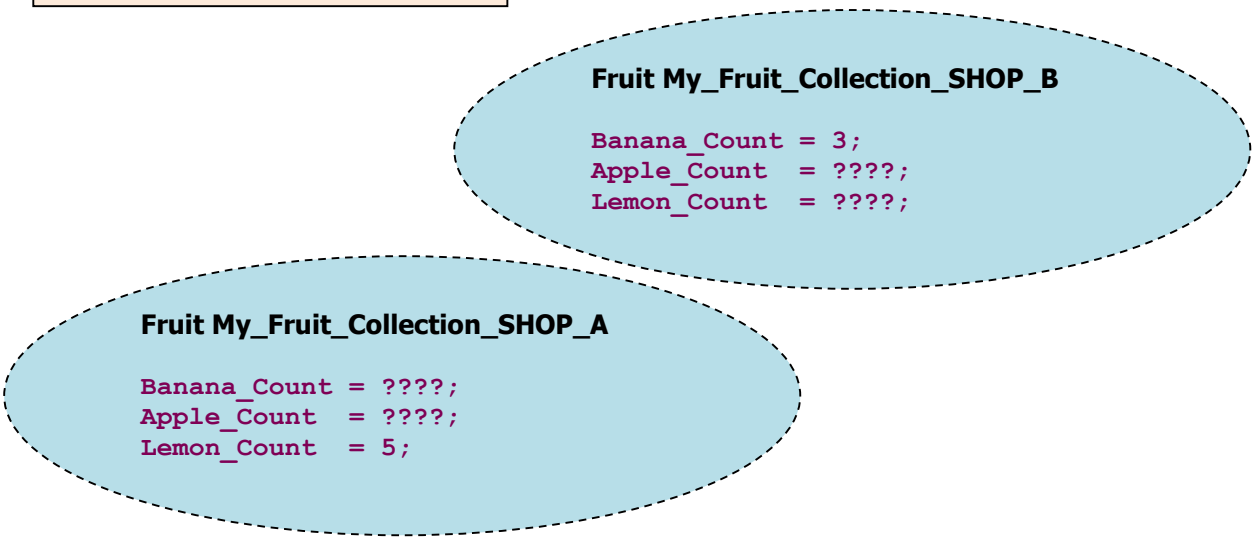
Worse still, imagine that you wanted to expand the software to count the fruit in two different shops; you'd need to double the number of lines of code just to declare the variables. Whereas with a struct, you'd simply go from one line of code to two:

```
Fruit My_Fruit_Collection_SHOP_A;
Fruit My_Fruit_Collection_SHOP_B;
```

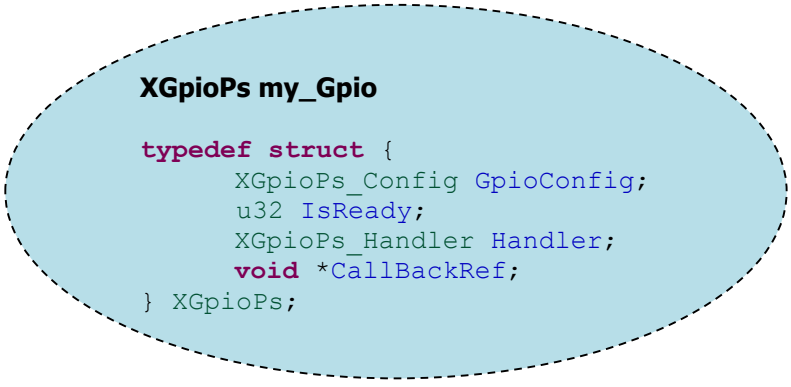
To access the individual variables inside the struct, you simply use the "->" notation in C. Here is an example showing how you'd assign two different variables inside two different structs:

```
My_Fruit_Collection_SHOP_A->Lemon_Count = 5;
My_Fruit_Collection_SHOP_B->Banana_Count = 3;
```

The result is two structs, and each of them has one initialised value.

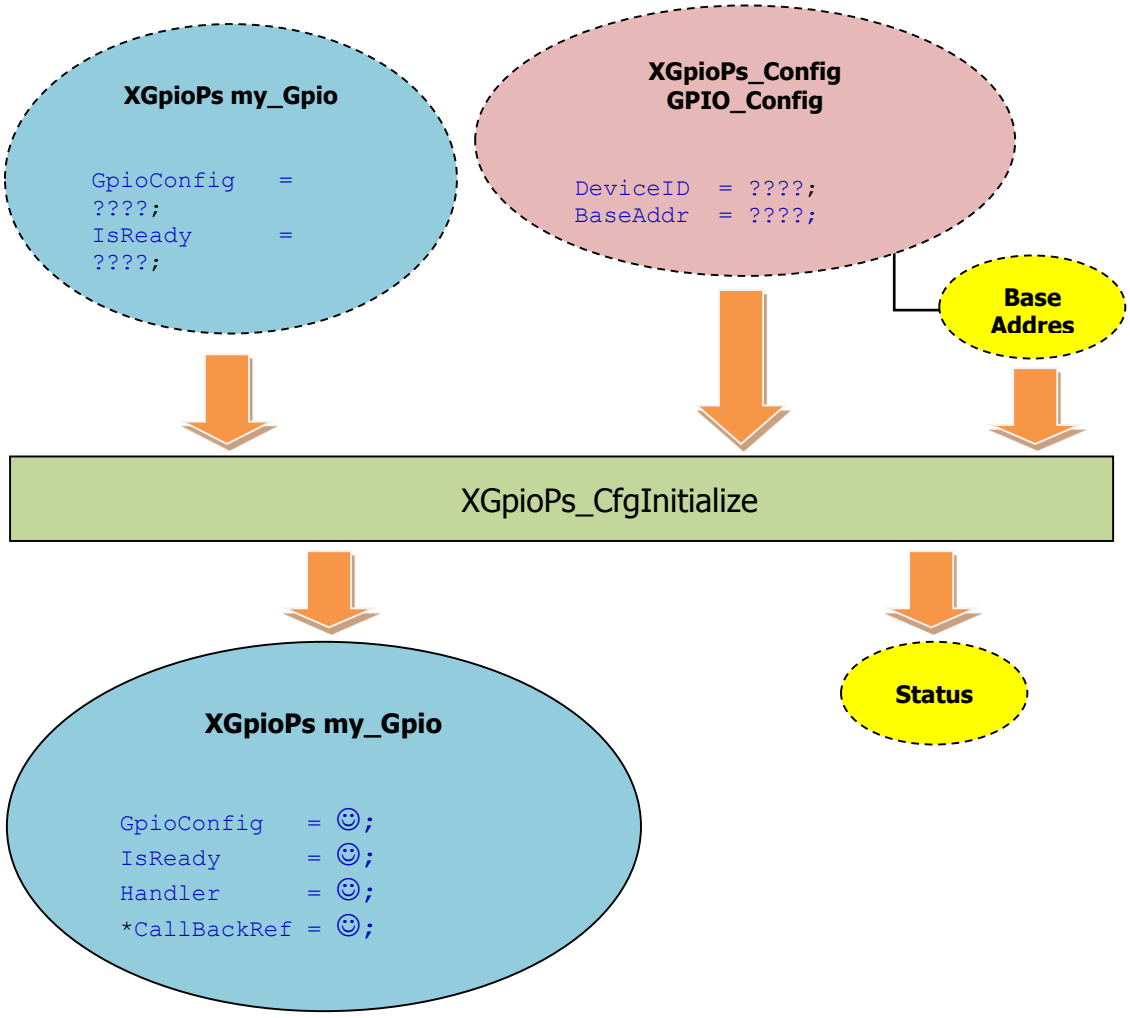


Xilinx drivers work in the same way. The struct for each peripheral represents the various operating parameters, and in the case of complex peripherals there may be a very large number of variables inside the struct. Fortunately in the case of our GPIO it is relatively simple and there are only a few variables. Here is the struct which controls the GPIO in the Processing System of a Zynq device. In this example we have declared an instance of the struct and called it "my_Gpio".



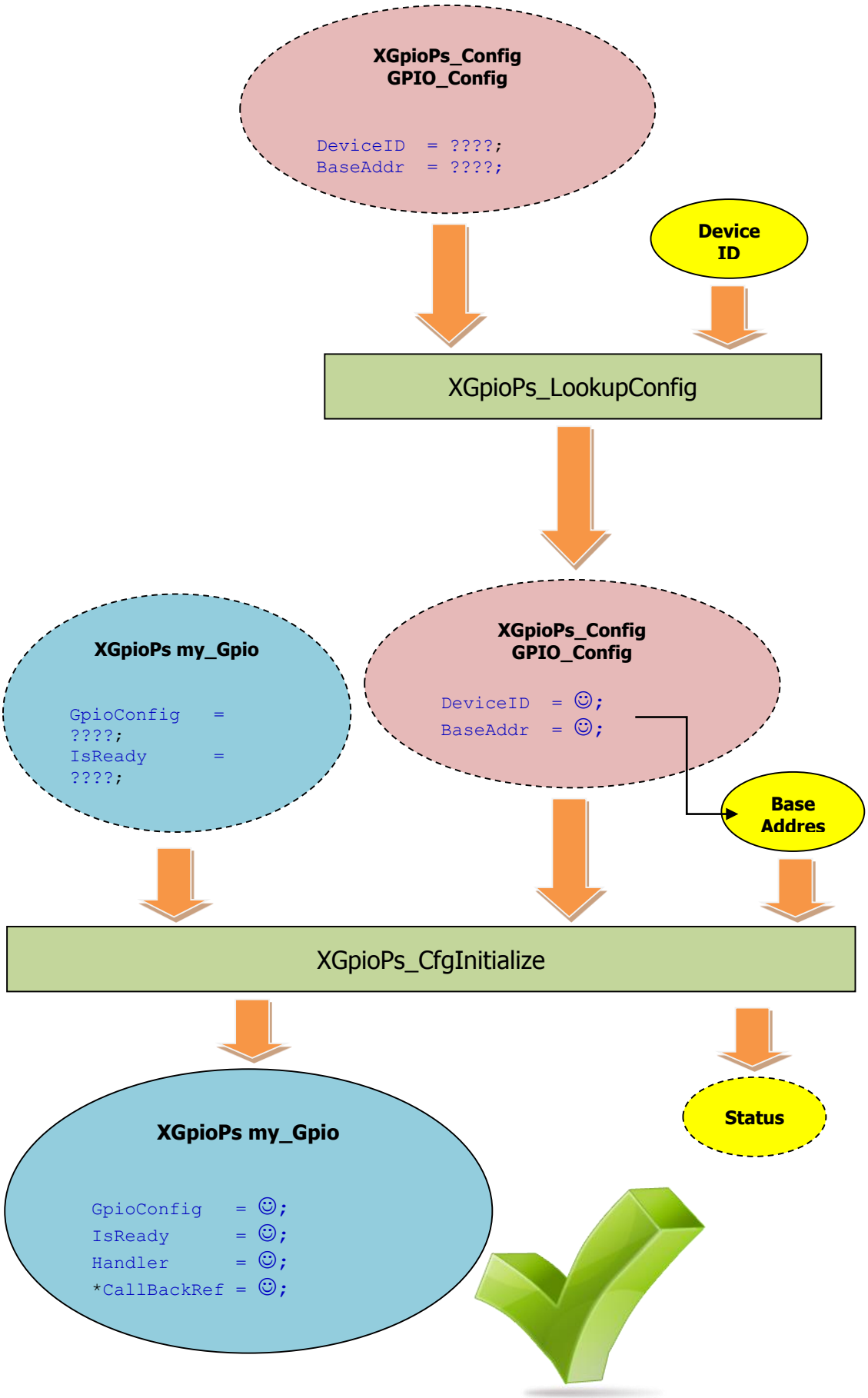
There are four variables inside the struct. The first is called "GpioConfig" and is of data type "XGpioPs_Config". This latter data type is actually another struct (yes, it's possible to have a struct inside a struct!), and we will use it to configure our "my_Gpio" instance.
The remaining three variables are all of different data types. It's not necessary to understand what everything does, but it is not difficult to believe that it would be a tricky task manually configure everything inside the struct. For this reason, Xilinx supplies a function in C which does the job for us, and it is called

"XGpioPs_CfgInitialize". The function automatically configures everything for us, because all of the variables inside are uninitialised when the struct is declared. The function requires three inputs; the instance of "my_Gpio" that we declared, the GPIO_Config struct, and a base address (which can be easily extracted from the GPIO_Config struct). The output of the function is a status value which lets us know whether the initialisation was successful. Here is a graphical description of how it works; we can see the three required items being passed into the XGpioPs_CfgInitialize function, and the initialised struct coming out (plus a status value).



But there's a problem! The "GPIO_Config" struct hasn't been initialised yet, either. So our first step is to use another automated function called "XGpioPs_LookupConfig" to achieve this task. In essence, the output of one function is simply fed into the other.

Again, let's represent this in a diagram for clarity. The important thing to note is that the only piece of information that we've had to enter manually is the "Device ID", and even that comes from the xparameters.h file:


XGpioPs_Config
GPIO_Config
DeviceID = ????;
BaseAddr = ????;
Device ID
XGpioPs_LookupConfig
XGpioPs my_Gpio
GpioConfig = ????;
IsReady = ????;
XGpioPs_Config
GPIO_Config
DeviceID = ☺;
BaseAddr = ☺;
Base Addres
XGpioPs_CfgInitialize
Status
XGpioPs my_Gpio
GpioConfig = ☺;
IsReady = ☺;
Handler = ☺;
*CallBackRef = ☺;

Let's start putting this into C code. The first task is to declare two structs; a temporary configuration struct called "XGpioPs_Config", and the second struct is the instance that will be used to control the GPIO, called "XGpioPs". You will see these listed in the "Outline" view of SDK when you select the "xgpiops.h" in your editor (yellow circular icon with a "T" logo).

Return to your "execise_2.c" source file and start adding some code above the "init_platform()" line. At this stage we will also introduce an incredibly powerful feature of the code editor called "Content Assist". This feature will save you huge amounts of typing once you learn how to use it. Start typing the first few characters of "XGpioPs_Config" and then hold down the "Ctrl" key on your keyboard and press the space-bar. A pop-up screen will appear giving you a list of possibilities, which has the same appearance as the Outline pane. Scroll down the list of possibilities (or continue typing the next letters to filter the list) and choose the "XGpioPs_Config" item from the list using either the mouse or the arrow keys on the keyboard and pressing enter. Note that the coloured icons are shown here too, to make it easier for you. Finish off the line manually by adding a name for this instance, and making the code match the following line. Notice the use of the "*" because this is a pointer, and of course the semicolon at the end of the line.

```
XGpioPs_Config *GPIO_Config;
```

Immediately below the line you've just typed, declare an instance of "XGpioPs" and also an integer called "Status" that we will use for some error checking later on.

```
XGpioPs my_Gpio;
int Status;
```

To initialize the GPIO driver, we need to provide it with three pieces of information when we use the function call in our code. The instance name of the driver which in our case is "my_Gpio", the configuration settings for the GPIO (we have created a placeholder for this in the form of "GPIO_Config" but not done anything with it yet), and the base address for the GPIO peripheral in question. The second and third parameters can be easily provided using a function call which you may have seen when you reviewed the Outline pane for the "xgpiops.h" file. As we saw in the diagrams, the function is called "XGpioPs_LookupConfig", and just needs to have one parameter passed to it (the "DEVICE ID") which we will find in the xparameters.h file (automatically generated for the design by Xilinx).

Below the "printf()" line in your existing code, add the following to your code to look up the settings for the GPIO configuration and store them in your "GPIO_Config" placeholder. Experiment using the Content Assist feature by pressing Ctrl+<Space> to help you write the code without lots of typing:

```
GPIO_Config = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
```

"GPIO_Config" is our instance name for the "XGpioPs_Config" struct data type that we declared earlier. As we saw previously, structs are essentially a series of smaller data types (variables, usually) which are grouped together under one name to make life easier for the user. In this particular case the "XGPioPs_Config" struct contains two variables; a 16-bit device ID called "DeviceId", and a 32-bit base address called "BaseAddr". SDK will allow you to find out the definition of any function or data type that you choose by using the "Open Declaration" feature from the Right-Click menu (this is the same as holding down "Ctrl" and clicking on the object). Add the following line to initialize the driver, remembering once again to use Content Assist to ease the strain your fingers:

```
Status = XGpioPs_CfgInitialize(&my_Gpio, GPIO_Config, GPIO_Config->BaseAddr);
```

You can now begin to see how structs work. We are recovering the base address for the GPIO using the "GPIO_Config->BaseAddr" notation. Essentially this is like using an ordinary variable name, but we're telling the compiler to look inside the "GPIO_Config" struct for a variable called "BaseAddr" by using the "->" syntax.

You'll notice that the initialisation function returns a value which we are capturing in our variable called "Status". We could add some code here to check that the initialisation was successful (Success = a returned value of "0"), or alternatively we could check the value in the debugger at runtime if we experience problems. To keep our code as simple as possible we'll choose the latter option, and we'll trust that the function call works as it should. In your own designs for production, you may want to check that status = 0 to ensure that everything was successful; which is good coding practice.

That's it! The GPIO driver is now initialised, and we can now use it to control the LED on our board. If we want to perform operations on the GPIO in our code, we can now do so by using the "my_Gpio" instance that we have just created and initialized.

When we configured the Zynq device using the Platform Studio tool in Exercise 1, the imported settings in the XML file for the MicroZed board included a GPIO peripheral. The GPIO is perhaps the simplest of all peripherals, in that it allows us to control a series of wires from software. Once again, review the driver header file for the GPIO and you will see that there are functions which help you to set the pins as inputs or outputs, and also functions which allow you to read and write from those pins. *(Hint: Use the "Ctrl + <click>" operation on #include line for the GPIO's header file)*

We're only interested in outputting a value to either illuminate or extinguish the LED, so we only need three functions:

```
XGpioPs_SetDirectionPin(XGpioPs *InstancePtr, int Pin, int Direction);
XGpioPs_SetOutputEnablePin(XGpioPs *InstancePtr, u32 Pin, u32 OpEnable);
XGpioPs_WritePin(XGpioPs *InstancePtr, int Pin, int Data);
```

These functions allow us to send values to the various registers within the GPIO peripheral. The first line will set the direction of the GPIO to be an output, but only for a specific pin. In our case this will be MIO 47 which is where the LED is connected. The second line sets the output enable buffer on the GPIO, allowing the value in the data register to reach the pin. A value of "1" enables the output buffer, and a value of "0" disables it. The third line allows us to write our chosen value to the data register; either "0" or "1". On the MicroZed board, the LED is connected from the Zynq-7000's output pin to GND, so a logic "1" will provide power to the LED.

Add the three lines of code to your application, filling in the blanks with the correct information. The first parameter should be a reference to your instance of the "my_Gpio" struct, so add a "&" character before the name of the struct to tell the compiler that you are using a reference (References are a more advanced concept of the C language... don't worry about it, just trust me!). The pin number is easy; 47. The data direction is unknown to us at the moment so we need to find this information. Go back to the "xgpiops.h" header file and click on the "XGpioPs_SetDirectionPin" function in the Outline pane. The editor will jump to the location of this function in the file. Now move your mouse to the editor window, and hover the mouse pointer over the highlighted function name. A helper window will pop up, showing you key information about the function. In this information you will see that a value of "0" makes the GPIO and input, and the value of "1" makes the GPIO an output. Excellent; we now have enough information to complete the code in our application. So let's add the code:

```
XGpioPs_SetDirectionPin(&my_Gpio, 47, 1);
XGpioPs_SetOutputEnablePin(&my_Gpio, 47, 1);
XGpioPs_WritePin(&my_Gpio, 47, 1);
```

That's it. We have changed the direction of MIO 47 to an output (output = 1) using the first line, the second line enables the buffer, and the third line writes the value of "1" to the pin to switch on the LED. Save the C source file, and the application will be compiled automatically. Check for any errors in the C code which will be highlighted and marked with a red cross.

Run the code on the board using the "Run as..." feature that you explored in Exercise 1. You should see the LED "D3" illuminate.

Change the code to send a "0" to the GPIO pin to switch the LED off, and then run the code on the board again using "Run as...". Be sure to change only the "XGpioPs_WritePin" line of code and not the "XGpioPs_SetDirectionPin" line. We only want to change the value we're sending to the UART, not change the direction of the port!

Now that you have established basic control over an LED, let's make it flash.

Let's create a loop in software using a "while" loop. Edit your code to match the following:

```
XGpioPs_SetDirectionPin(&my_Gpio, 47, 1);
XGpioPs_SetOutputEnablePin(&my_Gpio, 47, 1);

while(1)
{
        XGpioPs_WritePin(&my_Gpio, 47, 0);
        XGpioPs_WritePin(&my_Gpio, 47, 1);
}
```

This code makes an endless loop which repeatedly turns the LED on and off. Run the code on your board and see what happens.

- Is there any change?
- Really? Are you sure? The LED might be on, but is it as bright as it used to be?

If you're correct, then you will have realised that the LED is being switched on and off extremely quickly. So quickly, in fact, that it has the appearance of being dimmer than before. This is the way that LEDs are dimmed using digital control; and it's called Pulse Width Modulation. The brightness of an LED can be varied by changing the percentage of the time the LED is on versus off. Provided that the flash rate is fast enough, the human eye can't tell the difference.

For our purposes it would be interesting to have the software insert a delay between the "on" command and the "off" so that we can drive the LED on and off at a speed that we can realistically observe. Here is an example of a delay function:

```
for (delay=0; delay < 2000; delay++)
        {
        printf(".");
        }
```

As you can see, this is a simple loop which instructs the processor to print a full-stop to the UART two thousand times. Naturally, we are not at all interested in generating 2000 full-stops on the terminal, but doing so will take the processor a finite amount of time to complete the task, hence the delay. For any expert software engineers reading this; we know that this delay is phenomenally crude, but this is intentionally basic stuff for the moment. It gets more advanced later! ☺
The iterations are counted by using a variable called "delay" which we declared as an integer at the top of the file for convenience.

End of Exercise Challenge

Experiment with these function calls to write a piece of software that will make the LED flash at different speeds. If you really want to challenge your coding skills, try to make the LED fade up and down in brightness.

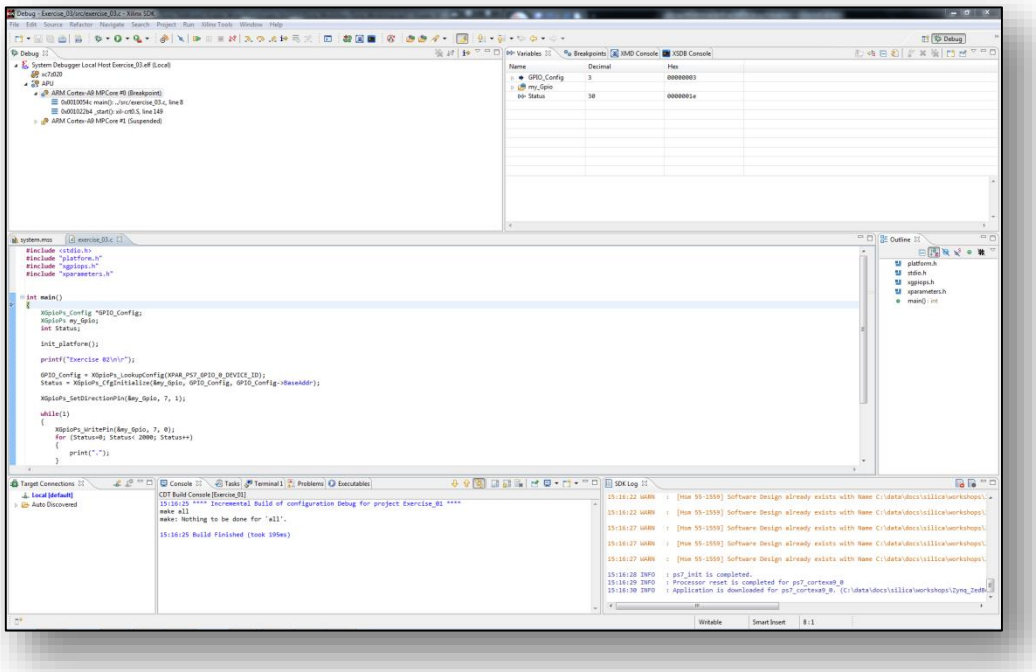
Exercise 3 - Debugging

In this exercise we shall look at the concepts behind debugging a piece of C code. This is important for all software development, because we are all human and therefore software rarely works first time. The ARM Cortex-A9 processors in the Zynq-7000 devices are supplied with a very advanced debug and trace interface, but we will simply cover some basic debugging techniques.

We shall now create a project to show the steps required for debugging. The finished version of Exercise 2 is an ideal design for this exercise. Create a new software application called "Exercise_03" using standard cut and paste operations, based on Exercise 2. The steps to do this are intentionally not explained, because it's very intuitive to do. Once you've made a new copy of the application project, experiment with changing the filename from "exercise_02.c" to "exercise_03.c" to match your new project (and you might want to change the printf message inside the file too; just to convince yourself that you're running different software!). The rename feature can be found in the right-click menu.

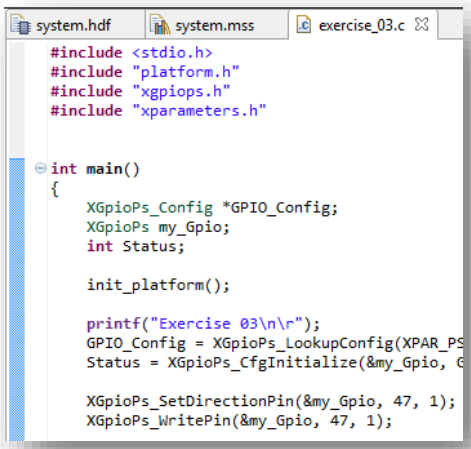
To debug the code on the processor we shall be using the system debugger which is built into the SDK, and can be accessed by Right-Clicking on the project name and choosing "Debug as" from the menu rather than "Run as".

Right click on the "Exercise_03" software project, and choose "Debug as → Launch on Hardware (System Debugger)" from the menu. After a few seconds you may see a pop-up message asking you whether you want to switch to the "debug perspective". Choose "Yes".

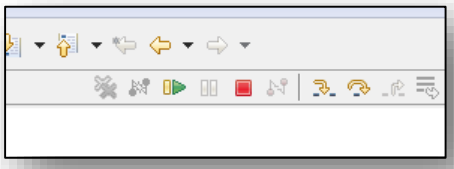


The windows will all re-arrange themselves, and you will now be in the "Debug perspective" rather than the "C/C++ Perspective". A "Perspective" in the Eclipse SDK

tool is the name given to a layout of the various panes on the screen. The layout of the screen in a SDK tool is critical to making the software engineer more productive. When writing a developing code, it is useful to have a very different layout of the screen than when running the debugger. Different information needs to be displayed, and the required size of each pane is very different. In the Eclipse SDK, the user can toggle back and forth between the different perspectives by using the buttons in the top-right of the screen. Advanced users can even design and use their own perspectives, to match their precise needs and personal preferences.



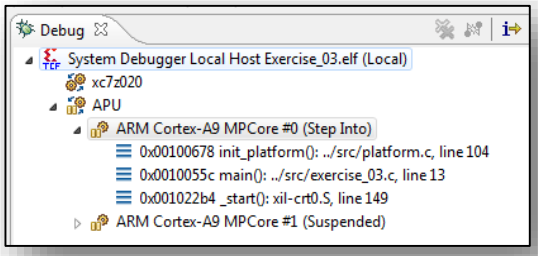
With the debug perspective open, you will notice that the code editing window has become smaller and has moved to the left of the screen. The Project Explorer window has vanished altogether; we no longer need to concentrate on different applications because we're debugging just one. In the code editing window, you will see that there is a line of code that is highlighted in green, and also marked with a blue arrow in the left margin. This is showing you that the processor has been halted at this stage of the application, and that the software execution is under control of the GDB debugger. The highlighted line of code is GDB telling you "I am about to execute this line of code, but I've not done it yet". The pane above the code editing window shows you that the Exercise_03.elf file is being debugged, and that debug control is being handled by something called "TCF". TCF is the Target Communication Framework system, and is the utility which manages the connection between the GDB debug engine, and the JTAG cable. You will be able to see that the code is halted inside the "main" function. This display will update to show when you're inside a function, and the hierarchy will be shown to demonstrate which functions called each other as you step through your code.



To see a practical example of this, we will step through our code until we descend into the "init_platform()" function. At the top of the "Debug" pane you will see a series of new icons. There's a green "Resume" arrow button which tells the code to continue to the next breakpoint (more on breakpoints in a moment), a red square "Terminate" button to halt execution completely, and then there is a series of yellow arrows which control the single-stepping of your application.

Hover your mouse over the yellow single-step arrow buttons and locate the one marked "Step Into". Click this button (perhaps a few times, depending on your code that you copied from Exercise 2) and you will "Step Into" the init_platform() function. Observe what happens in the "Debug" pane when you descend into the function. You will see the "_start()" function underneath the "main" function, which is also

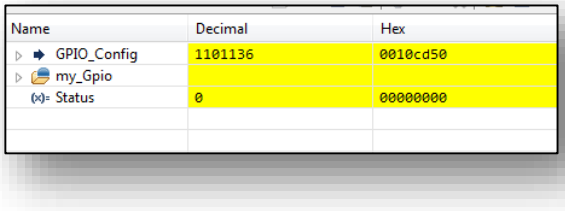
underneath the "init_platform" function. This will tell you that "_start" has called "main", which in turn has called "init_platform", and therefore tells you which functions you passed through on the way to your current point in the code. In our example it's very simple, but in a complex application this is very useful to see how you got to your breakpoint.



We will now examine the difference between the "Step Into" and "Step Over" buttons. Inside the "init_platform" there is a further function call to "enable_caches". Click "Step Into" again, and you will see that the debugger will descend once again into this function. Note the change to the "Debug" pane and the hierarchy view. We could single step through this lower level function if we wanted to, but let's imagine that we want to execute to the end of the function and return to just one level of hierarchy up. Click the "Step Return" button, and the code will execute all of the way through the current function and back up to stop at the level above. Now click the "Step Over" button, and you will see that the debugger stays at the same level of hierarchy, but executes all of the "init_uart" function in one go. Click "Step Return" to get back to the "main" function.

In the code editing pane, scroll down a few lines and find the line of code which sets the direction of the GPIO port. In the blue margin to the left of this line, double click your mouse and you will see a blue dot with a tick mark appear. This is a breakpoint, and tells GDB that you wish to always stop at this point. Now click the green arrow "resume" button, and the debugger will execute all of your code between the current position and the breakpoint that you have just set. This is a useful way to execute a lot of your application, but only stop at the section you wish to debug.

At this stage we will turn our attention to the Variables pane which you will see in the top right of the screen. You will see all of the local variables which are relative to this function. It's a simple application, so you should only see three or four listed. One of the variables is "Status" and hopefully its value will be "0". This proves that the "XGpioPs_CfgInitialize" function completed and returned no errors. This is good news. In the "Variables" pane you should also see the "GPIO_Config" placeholder which you created in Exercise 2. Recall that this is actually a struct, which is a collection of variables under one name. Expand the hierarchy of the "GPIO_Config" struct by clicking on the white arrow to the left of the name. You will now see the two variables contained within the struct, and their values. The "GPIO_Config -> BaseAddr" variable probably seems to be a meaningless number, but that's because we're looking at the decimal value rather than the hexadecimal value. A second column exists in the Variables table which shows the hexadecimal equivalent of the



number. Now the number should make more sense. This is the base address of the GPIO in the Zynq-7000 device memory map.

You can also expand the "my_Gpio" instance in the Variables pane. You will see that it's actually a struct containing another struct, and therefore there will be multiple levels of hierarchy that you can explore. You may see that some of the variables are highlighted in yellow. This occurs when the variable has been updated since the last breakpoint or during the last single-step operation. This feature of the debugger can be very useful because it draws your attention to things that change during your debugging session.

But the Variables pane is not the only place where you can explore the value of variables. In the code editing window, hover your mouse over the name of a variable or a struct. A pop up helper will be displayed that will give you the same information. Simple, but useful.

Add another breakpoint inside your "for" delay loop which prints the "." characters to the UART. Click the "Resume" button and the code will advance to inside this loop. Click "Resume" a few more times, and observe the change to the variable which counts the iterations of this loop. You will see the variable incrementing with each click of "Resume". Single-stepping our way out of this loop could be a very tedious process, and would require 2000 mouse clicks. But what if there was a fault in the code just before the last iteration of the loop? The debugger can help us in these cases, because we can dynamically change the values of variables in the middle of executing the code.

Inside the "Variables" pane, click the value for your loop variable and manually change it to "1997". Now click the "Resume" button and notice that within a few clicks we have left the loop. If you find yourself back in the loop but with the loop variable set to zero, it's because you have exited the loop and re-entered it on the next iteration of the entire application. By using this feature, we have manually updated the value of a variable to set up a specific condition in the software. This can be used to debug a known set of events which might be causing a problem in your application.

That's it – you've just mastered the basics of single step debugging. Feel free to play around in the debugger to familiarise yourself with the features that are available.

End of Exercise Challenge

It is sometimes convenient to temporarily disable a breakpoint, but still remember where you put it. The SDK allow you to do this.

Select the "Breakpoints" tab in the top right of the screen, and you will see a list of the breakpoints that are currently set. Experiment with the tick boxes to the left of the breakpoints. Set up various breakpoints in your code, and then enable and disable them to see how the debugger works.

End of Exercise Mega Challenge

Still hungry for more? Well OK then...

In this exercise we manually adjusted the value of our loop variable to a given value (1997). This was fine for our application, but in a more complex software project this method could seriously affect the correct operation of the design.

Another way to achieve the same goal is to set up a "conditional breakpoint". A conditional breakpoint is one where the execution of the code will only stop when a specified condition is met. In our case, we could set the condition to only break when the value of our loop variable was equal to "1997".

Select the "Breakpoints" tab in the top right of the screen, and you will see a list of the breakpoints that are currently set. Right-click on the breakpoint which applies to your loop, and then choose "Breakpoint Properties..."

Work out, perhaps by using the Help system built into SDK, how to set up a conditional breakpoint that only stops when the value of the loop variable reaches "1997". This might not be immediately obvious, so don't get too annoyed with it if you can't find the answer.

When you have finished debugging your code, click the C/C++ button in the top right of the screen to return to the code editing perspective.

Exercise 4 – Expanding your design into the programmable logic

In the previous exercises we have confined our efforts to just the hardened "processing system" (PS) section of the Zynq-7000 device. However, Zynq-7000 is much more than just a processor, and it provides a large quantity of FPGA "programmable logic" (PL) that can be configured by the user.

Several use models for the PL; the user can write and design custom code in a hardware description language such as VHDL or Verilog, they can use a high level language synthesis tool to convert C code into hardware (such as Vivado HLS), or they can add items from an existing catalogue of IP. To expand our design we will choose the latter option, adding some existing IP from the catalogue.

Close the SDK tool, and re-open Vivado if you have closed it. In the "Sources" pane, expand the hierarchy of sources until you can see the block design that you created in Exercise 1. Double click this source and the block diagram editor will be invoked so that you can make edits to the design.

Note that the design only contains one item, the "Zynq7 Processing System", which is the IP that represents the hardened "PS" section of the Zynq device. We will now expand this into the programmable logic "PL" section of the Zynq device. Specifically, we will add functionality to use some BlockRAM in the PL, and also control one of the PMOD expansion sockets.

Double click the "Zynq7 Processing System" block to open the customisation window. Click on the green box at the bottom of the Zynq diagram marked "32b GP AXI Master Ports", and then enable the "M AXI GP0 Interface". This will provide an AXI port which can be used to connect additional peripherals in the programmable logic part of the Zynq-7000 device. Click "OK".

Click the "Add IP" icon in the control bar to the left of the block diagram. Type "bram" into the search box, then click and drag an instance of the "AXI BRAM Controller" from the catalogue into the white space of the block design. Now search for "block" and drag an instance of the "Block Memory Generator" peripheral into the block design. Click anywhere in the white space of the block diagram to close the Add IP box.

We now need to set the parameters for these two IPs. Double click the AXI BRAM controller block to open its customisation window. In the "BRAM options" section, adjust the "Number of BRAM interfaces" setting to 1, and then click "OK"

Note that at the top of the window the "Designer Assistance" feature in the green bar has re-appeared. Click "Run Connection Automation" and tick the box next to the "All Automation" item in the list. Click "OK". Note how the two new peripherals have been automatically connected to the Zynq processing system block, and the Vivado tools have added AXI interconnects, reset controllers, and clock nets, without the user having to make any manual connections. The block diagram may now look a bit untidy, but this can be automatically fixed later.

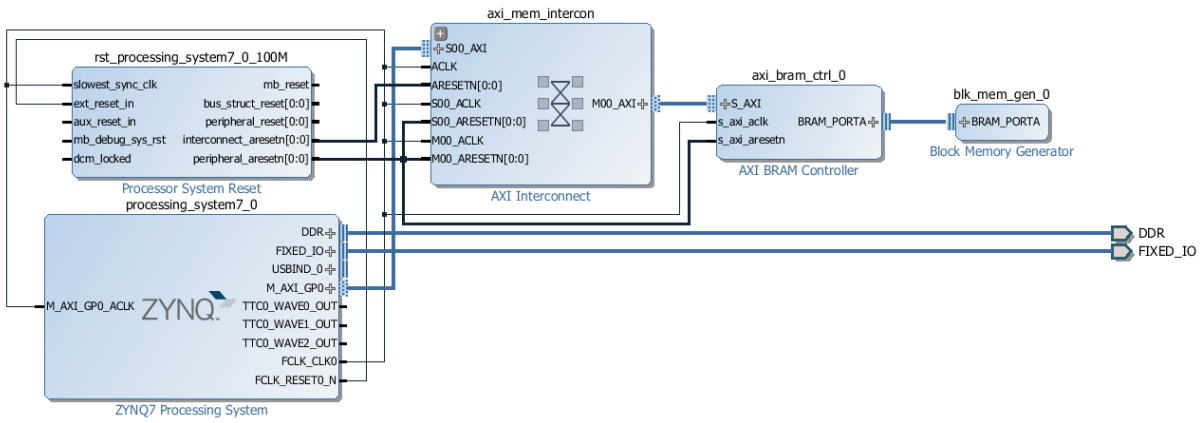
Later in this workshop, we are going to plug an expansion board into the 12 pin "PMOD" expansion socket at the end of the board. PMOD sockets have four user IOs, one Vcc, and one GND pin on each row. The PMOD connector on the MicroZed board is a dual row socket, but we only intend to use one row of pins for the expansion board. The MicroZed Hardware user guide (http://microzed.org) shows that the first four IOs on the PMOD socket are connected to MIO10 to MIO13.

Table 11 – Digilent Pmod™ Compatible Interface Connections

Pmod	Signal Name	PS MIO Pin	Zynq pin
J5 MIO Pmod	PMOD_D0	MIO 13	E8
	PMOD_D1	MIO 10	E9
	PMOD_D2	MIO 11	C6
	PMOD_D3	MIO 12	D9
	PMOD_D4	MIO 0	E6
	PMOD_D5	MIO 9	B5
	PMOD_D6	MIO 14	C5
	PMOD_D7	MIO 15	C8

We must therefore enable one of the SPI controllers in the PS and route the IOs to external pins on the MicroZed board. Double click on the "Zynq7 Processing System" block and then select the "MIO Configuration" page from the left of the dialogue box. Enable the "SPI1" controller by ticking the box, and then use the drop-down box to select "MIO 10 … 15" as its user IOs. The first slave select pin "SS[0]" will be enables on MIO13; we do not require more than one slave select pin for the expansion board, so leave SS[1] and SS[2] disabled. Click "OK" to return to the block diagram.

Spend a few moments reviewing the completed block diagram and note that the additional peripherals are now connected to the processing system block. If the block diagram is looking cluttered or untidy, click the "Regenerate Layout" button in the control panel to the left of the block diagram, and the block design will be automatically re-drawn.



Select the "Address Editor" tab of the block design, and expand the hierarchy for the "processing_system7_0" instance and also for the "Data" settings. You will see the base address and high address settings for the new soft peripheral that we've just added to the design (the block ram). The tools have automatically allocated addresses to the new peripherals, making them suitable for use with the Zynq-7000 device. If you wanted to adjust the size of the BlockRAM, you could do so by changing the value in the "Range" column. The larger the Block RAM you choose, the more resources you will consume in the PL and the tools will need to work harder during place and

route. We will only be using a small amount of block ram, so please leave this set to "8KB". Note: You will not see any settings for the SPI controller in this table, because the SPI controller has a fixed base address within the processing system.

That completes the changes to this design. Save the changes to the block diagram, and then from the Vivado Flow Navigator pane, click "Generate Bitstream". A dialogue box should appear warning you that the Synthesis results are out of date, and asking you for permission to re-run synthesis and implementation. Click "Yes" and the design will be completely re-built. Take a few minutes to stretch your legs and top up your caffeine levels. The next exercise will require your concentration.

When the Bitstream has been generated, please re-export the design into SDK by choosing "File → Export → Export Hardware..." from the menus, and tick the "Include Bitstream" option. You may see a warning that you are about to overwrite the existing design, and this is OK. If you had closed the SDK tool, please re-launch the SDK using "File → Launch SDK".

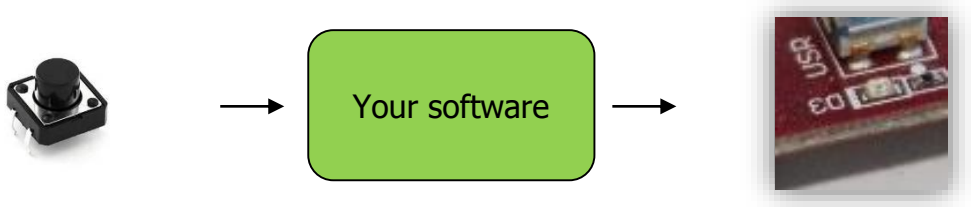
When the SDK opens, you may notice furious activity from the Library Generator and compiler tools. If the SDK was already open, you may see an alert advising you that the hardware has changed. This is because the tools will automatically detect the changes you have made to the hardware, and they will then update the BSP and all of your software applications for you without any fuss. It just works!

Exercise 5 - Making your design interactive

It is time to develop a more complex design, which will allow you to practice using the tools in preparation for your future embedded processor projects. In this exercise, we will look at how to make the processor read values from user inputs (in our case a simple push button!) and then use that information to control some outputs (in our case an single LED). Once again this will require the use of the supplied software drivers, which have been produced by the Library Generator tool when we created a BSP. All software drivers read or write to an address in memory, which in turn is mapped to a register in a peripheral. The job of a peripheral is to make hardware registers appear as if they are memory, so that the processor believes that it is simply reading from and writing to RAM. The fact that the peripheral adds extra functionality to these registers for controlling UARTs, GPIO, and other functions is completely hidden from the processor.

On the MicroZed board the push button is connected to MIO 51, and the LED is connected to MIO 47 as we saw in the earlier exercise.

For our first example, we shall make the processor read the push button and simply output its state to the LED. Obviously this could be done by wiring the switch to the LED in FPGA programmable logic, but we will use software for the purposes of our learning.



End of Exercise Challenge

Using the first few exercises as a reference, complete the following:

- Use the menus in SDK to create a new software application called "Exercise 5"
- Examine the driver created for the GPIO by Library Generator by examining the appropriate header file. The GPIO we want is the Xilinx GPIO in the PS. *The underlines are a "subtle" hint!* ☺
- Follow the guidance discussed in the previous exercises to declare an instance of the Xilinx GPIO in your software.
- Use the same method as you did in the previous exercise, configure the GPIO by identifying the struct that will be used to store the Xilinx GPIO configuration

information. Also identify a function that will lookup the details of the Configuration.

- The function to lookup the config information will require a device ID, so locate this in the xparameters.h file.

- Locate the function that will initialise the configuration, using the config struct that you found just now.

- Find a function that sets the data direction on the GPIO so that the pins for the push button and LED are inputs and outputs, as appropriate. This is slightly more complicated than last time because you need to call the function twice, once per pin.

- Identify two functions that will read and write from the GPIO pins.

- Remember that you can use the "Open Declaration" feature in the right-click menu (or use Ctrl+Click) to examine functions in greater detail, and this will help you to determine how to use them. Xilinx documents each of their driver functions in code comments above the function in question.

- Once you have researched the functions mentioned above, write a piece of software in C that will check the value of the push button, and then write the state of the button to the LED.

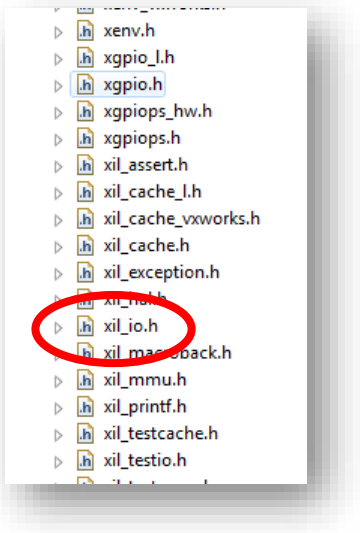
Here is a starting point for the code, you will need to fill in the gaps. This code template is also available in the supplied files:

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


int main()
{
	XGpioPs_Config *GPIO_Config;
	XGpioPs my_Gpio;
	int Status;

	init_platform();

	printf("Exercise 05\n\r");
	<ADD CODE HERE TO LOOKUP THE CONFIG OF THE GPIO PERIPHERAL>
	<ADD CODE HERE TO INITIALISE THE GPIO PERIPHERAL>

	<ADD CODE HERE TO SET THE DIRECTION OF THE GPIO PINS>
	<ADD CODE HERE TO ENABLE ANY GPIO OUTPUT BUFFERS>

	while(1)
	{
		<ADD CODE HERE TO READ THE VALUE OF THE BUTTON INTO A VARIABLE>
		<ADD CODE HERE TO WRITE THE VALUE OF THE BUTTON INTO A VARIABLE>
	}

	return 0;
}
```

Exercise 6 – Reading from and writing to memory

This exercise will look at ways to read from and write to memory. We shall begin by exploring why this is so important. Memory is not only used to store software in an embedded system; quite often the user will have a piece of memory as the interface between the embedded processor system and some custom hardware in the FPGA design. The Xilinx dual port BlockRAMs are very often used in this way, so it is important that we explore how to access memory from our processor software. The On Chip Memory (OCM) in a Zynq-device 7000 also behaves like a dual port BlockRAM memory. External (e.g. DDR) memory works in much the same way, the only difference being that the memory controller interprets the request from the processor and generates a transaction for the external memory which uses the correct protocol of signals used by the DIMM or device.

To do this, we shall once again use some drivers that have been generated by the Library Generator. The drivers are not associated with any peripheral, but are nevertheless made available to EDK users in all processor designs. Let us look again in the "include" directory under the name of the processor instance. Here we can see that Library Generator has created some other header files (.h) that do not relate to any particular peripheral. Let us take a look at the "xil_io.h" header file. In it we will see various functions designed specifically for reading from and writing to memory, in various byte widths.



These functions include:

- Xil_In8
- Xil_In16
- Xil_In32
- Xil_Out8
- Xil_Out16
- Xil_Out32

Processors work on byte (8bit) address boundaries. If we specify an address in memory, we must do so assuming that we are always referring to a number of bytes. Therefore, if we assume a 32 bit wide memory and we wish to write byte-wide data values into the first four consecutive locations in a region of memory starting at "DDR_BASEADDR", we must write the first to DDR_BASEADDR + 0, the second to DDR_BASEADDR + 1, the third to DDR_BASEADDR + 2, and the last to DDR_BASEADDR + 3.

However, if we wish to write four half-word wide (16 bit) data values to four memory addresses starting at the same location, we must write the first to DDR_BASEADDR + 0, the second to DDR_BASEADDR + 2, the third to DDR_BASEADDR + 4, and the last to DDR_BASEADDR + 6.

It therefore follows that when writing word wide (32 bit) data values, we must do so on 4 byte boundaries; 0x0, 0x4, 0x8, and 0xC.

As you can see from the xil_io.h header file, there are functions for reading from memory in 8 bit, 16 bit, and 32 bit transactions. The same applies for writing to memory in the same width variants. The exact syntax is very simple to use; here are the function templates for writing:

```
Xil_Out8(memory_address, 8_bit_value);
Xil_Out16(memory_address, 16_bit_value);
Xil_Out32(memory_address, 32_bit_value);
```

... and for reading:

```
8_bit_value = Xil_In8(memory_address);
16_bit_value = Xil_In16(memory_address);
32_bit_value = Xil_In32(memory_address);
```

Here is an example, showing how you might access the BlockRAM in the PL:

```
int main(void)
        {
        int result1;  // integers are 32 bits wide!
        int result2;  // integers are 32 bits wide!

        Xil_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR + 0, 0x12);
        Xil_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR + 1, 0x34);
        Xil_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR + 2, 0x56);
        Xil_Out8(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR + 3, 0x78);

        result1 = Xil _In32(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR);

        Xil_Out16(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR + 4, 0x9876);
        Xil_Out16(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR + 6, 0x5432);

        result2 = Xil_In32(XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR + 4);

        return(0);
        }
```

MENTAL EXERCISE QUESTION

For this code example, predict the values of "result1" and "result2" after this software has been executed? Read the code carefully and consider the width of each read and write transaction.

End of Exercise Challenge

- Write some software that will write eight data values, each one byte wide with the following values; 0xAB, 0xFF, 0x34, 0x8C, 0xEF, 0xBE, 0xAD, 0xDE. The

values should be written to consecutive addresses, starting at the base address of the Block RAM region.

- Leave a gap in the RAM memory of 2 "words" in size following the values you have just written. Then write four "half-word" data values at the next available addresses in the OCM, of values 0x1209, 0xFE31, 0x6587 and 0xAAAA.

- Add some code that will write the value of 0x00000000 (32 bits of data) to address 0xE000A244.

- Add code that will read two words of data starting from the base address of the block RAM region, placing the values into integer variables called "word1" and "word2".

- Add code that will read a word (32 bits) of data from address 0xE000A064, placing the result into an integer variable called "word3". Type this address in your code carefully – it's special, and it is NOT the same as the one we used just now. A lot of people get this wrong.

- Using the following lines of code, display the contents of word1, word2, and word3 to the UART.

```
printf("Word1 = 0x%08x\n\r", word1);
printf("Word2 = 0x%08x\n\r", word2);
printf("Word3 = 0x%08x\n\r", word3);
```

- Run the software on the board using "Run as → Launch on Hardware (System Debugger)". Check that the output on the UART terminal screen is as you'd expect for word1 and word2. Some of the data might seem to be around the wrong way. It's not, but consider why this might be the case. Think about how the addresses are arranged in the memory map for data values larger than 8 bits.

- Does the value of "word3" mean anything to you? It probably doesn't; but try running the software on the board again, but this time press and hold the push button on the board while you run the application. What happens to the value of "word3"?
  - Note: If you are reading the value of "0x00000000" for word3, then you have typed the wrong address into your code. Please go back and check carefully.

- Compare the value of word3 between when you press or don't press the push button, and see if you can determine what's going on. It might be helpful to convert the hexadecimal values to binary, and a pattern should suddenly emerge. What do you think is located at this special address 0xE000A064 ?

Big Gotcha!!

Now that we have created some soft peripherals in the programmable logic, it is vital that you program the bitstream into the Programmable Logic before you run the software. To do this, use the menus in SDK; Xilinx Tools → Program FPGA, and click "Program" to accept the defaults. The Bitstream will be loaded into the device, and the blue "DONE" LED will illuminate on the MicroZed board to indicate that the bitstream was successfully programmed. Remember to download the bitstream again if you power-cycle the board for any reason.

End of Exercise Challenge – ANSWERS

- Word 1 should be 0x8C34FFAB.
- Word 2 should be 0xDEADBEEF.
- If you've guessed correctly, you will have worked out that there is a GPIO peripheral at address 0xE000A064.
  - Specifically, this address is the register that reads the values of the pins which are attached to the push button on the board.
  - By pressing and holding the button, you are affecting bit 18 of the PS_GPIO read register at address 0xE000A064.
  - If we wanted to determine whether the button was being pushed, we would read the contents this register and then bitwise AND it with the mask 0x40000 ("$_{31}$0000 0000 0000 0100 0000 0000 0000 0000$_{0}$" in binary) to isolate bit 18. If the result was non-zero then we'd know that the button was being pushed.
  - This last part of the exercise essentially demonstrates how device drivers are written. The GPIO driver includes the "XGpioPs_ReadPin" function call that we used in our earlier experiments. The function call performs the read and the mask operation for us, but simply hides the complexity from us to make life easier.

Exercise 7 - Timers (Polled mode)

So far in this workshop, we have experimented with various driver functions to control the features on the MicroZed board. Processors are able to execute instructions extremely quickly, and in some of our examples we have used loops in software to slow down the processor to a speed which is suitable for our needs. Here is one of those (extremely crude!) delay loops from earlier:

```
for (i=0; i<2000; i++)
        {
        print(".");
        }
```

Although loops like the one shown here are effective for implementing delays, they are extremely inefficient in terms of processor time, which is often a precious resource in embedded designs. All of the time that we spent wastefully going around in the loop doing nothing, could be better spent executing real code and producing useful results. Also, when using our rudimentary loops we had no real control over the amount of delay time that would be produced by the delay loop. We could adjust the size of the loop, but this is a trial and error process and has no precise correlation to predictable and measurable units of time.

In this exercise we shall look at how it is possible to accurately measure time using a hardware-based timer peripheral. This will give us much greater flexibility in our designs, considerably more control, and will allow us to avoid wastage of the processor's time.

A timer, just like any other processor peripheral, is a block of hardware which has been added to the bus structure using memory mapped registers. Timers can work in many ways but we shall first look at using a countdown timer in "polled mode". To explain the process in its most basic form, we shall use the processor to write a value into one of the registers within the timer and then starting it running. The timer will decrement once per clock cycle until it reaches zero, and then stop. While it is counting down we shall "poll" the timer from the processor, which is simply the art of repeatedly asking the timer if it has finished. If it were a human conversation it would go something like this:

PROCESSOR: Load the value of 10,000 into your register and start running.
TIMER: OK. I'm doing that.
PROCESSOR: Have you reached zero yet?
TIMER: No.
PROCESSOR: Have you reached zero yet?
TIMER: No.
PROCESSOR: Have you reached zero yet?
TIMER: No.
PROCESSOR: Have you reached zero yet?
TIMER: Yes.

PROCESSOR: Good, I can start doing something else.

This scheme will allow us to measure time quite accurately; we know the value that was loaded into the timer, and we know the frequency of the clock connected to the timer. Therefore we know precisely how long it will be before we reach 0x00000000, and that can be checked by the processor to insert a known delay into our software. The Xilinx tools have already generated device driver functions in the BSP for writing the values into the timer's registers, enabling and disabling the timer, and for checking the value currently held within the timer. We can use these functions to control the timer from our code, in the same way that we did for the other peripherals.

There are various timers in a Zynq-7000 device, but we shall use the one built into the Snoop Control Unit (don't worry about the details of the SCU for a moment – but please just believe me that there's a timer in there!).

In the "xparameters.h" file you will find a reference to the SCU timer called "XPAR_PS7_SCUTIMER_0_DEVICE_ID". You will need this in your code.

Open the header file in the BSP "include" directory called "xscutimer.h". Browse through the available functions and identify functions which do the following:

- Lookup the config for initialising the timer.
- Initialise the timer using config information.
- Load a value into the timer.
- Start the timer.
- Read the current value of the timer's counter register
- Restart the timer from the initial value

Also identify the pre-defined struct which is used to store the settings for the timer driver, and the struct used for configuration.

End of Exercise Challenge

You should be getting the hang of using Xilinx drivers by now, so you can write this application on your own. Oh, stop blubbing, it's not that hard!

- Using the xscutimer.h header file as a reference, write some code which does the following:
  - Look up the config for the timer and the initialize it.
  - Load a suitable value into the timer which reflects a period of one second in real time. *(HINT: The SCU Timer runs at half the frequency of the processor, and in xparameters.h you will find some #define statements which define the frequency of the CPU clock)*.
  - Start the timer.
  - Inside an infinite software loop, constantly check the value of the timer's counter register. When it reaches zero:

- Print something to the UART. An incrementing counter value would be a good idea, so you can see the number of seconds which have elapsed.
- Restart the timer from the original value so that the timer keeps running, reaching zero repeatedly at a frequency of 1Hz.

Hints and Tips

- Make sure that you read the header file that is generated in the BSP. This will explain many things!
- The debugger is your friend, remember to use it! Also remember that you can insert "printf()" statements in the code to keep an eye on the progress of your software during development.

Exercise 8 - Timers (Interrupt mode)

We have seen from the last exercise that peripheral based timers can be very useful for accurately measuring time, and represent a much more elegant solution than using variable delay loops in software. Although we have cured the problems of inaccuracy, in the previous exercise we are still polling the timer which is an inefficient use of processor time. To cure this inefficiency, we can use interrupts.

An interrupt is simply an event which causes the processor to stop what it is doing, make a record of the stage it had reached and the contents of all its internal registers, and then begin work on a new task by executing a different piece of software. The alternate task usually performs a specific function which is related to handling the event which caused the interrupt, and thus we call the alternate piece of software an "interrupt handler". An interrupt handler is written in C code just like any other function, and we then assign the interrupt handler function to the peripheral using some special C functions supplied in the BSP. The Xilinx BSP automates the rest of the flow, and our software will return the processor to the original task once the interrupt handler has executed to completion.

People often get confused about interrupts, and don't always understand why they are helpful in software design. The easiest way to understand interrupts is to liken them to a simple household scenario. Imagine the front door of your house, and imagine you're sitting inside the house watching TV. How do you know if there is somebody waiting at your door with the pizza that you ordered? One method is that you could routinely get up, walk to the door, and check. If you check very regularly, then you stand a good chance of finding your pizza at the exact moment it arrives, but you'll spend a lot of wasted time going back and forth, you'll miss most of your TV show, and your neighbours will think you're odd. If you check less often then you will waste less of your time, you'll see more TV, but you might need to be quite lucky in order to get your pizza before it gets cold. This is "polling"; a constant checking process that requires a lot of manual work. All of the same principles apply in software.

The second method is to install a doorbell. When pizza boy arrives, he sends you a signal using a button by the door, and you are notified. You can then stop watching TV, and go to the door only when needed. It takes a bit more time to set up the system in the first place, but ultimately it's more efficient. This is an interrupt. Again, all of the same principles apply in software. Simple, huh?

Here is an example of a piece of code that could write a line of text to the UART when an interrupt was generated. At the moment we will not worry about how or why the

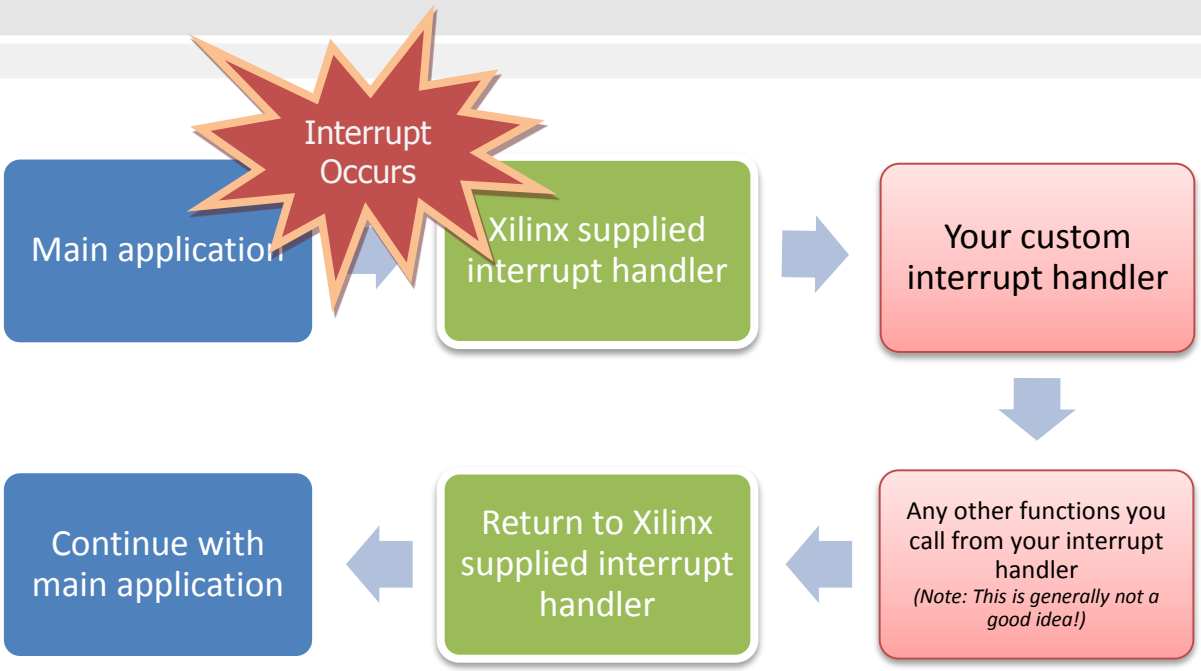
interrupt was generated, but simply study it as a textbook example of how the code might appear in C:

```
int main (void)
        {
        assign_interrupt_handler(&my_doorbell, front_door_interrupt_handler);

        while(1)
                {
                print("I'm watching TV\n\r");
                }

        return(0);
        }


void front_door_interrupt_handler (void)
        {
        print("My pizza has arrived!\n\r");
        }
```

Interrupts can be generated by many different peripherals, but for this exercise we will be generating interrupts from the Snoop Control Unit Timer peripheral that we used in the previous exercise. In the last exercise we were operating the timer in polled mode, although the timer peripheral is in fact capable of producing interrupts. The Zynq documentation states that an interrupt is generated by the timer whenever it reaches zero, so we shall use this feature to our advantage.

In order to use interrupts, we need to use a few more driver functions to enable the interrupt functionality on the timer, to enable interrupts on the ARM processor, and also to set up the interrupt controller. For the beginner, this process can get a bit scary, but I'll walk you through it. You're going to need coffee for this next bit. **No really… please get coffee!**

Let's look at the software flow for an interrupt. We start in the main part of the code, and then when an interrupt occurs we jump first into a supplied general purpose interrupt handler, and then again into your custom interrupt handler. The supplied interrupt handler does all of the important tasks such as saving the state of the processor, and tidying up any mess that is caused by the interruption. All you have to do is to write the code that you want to execute when the interrupt occurs.



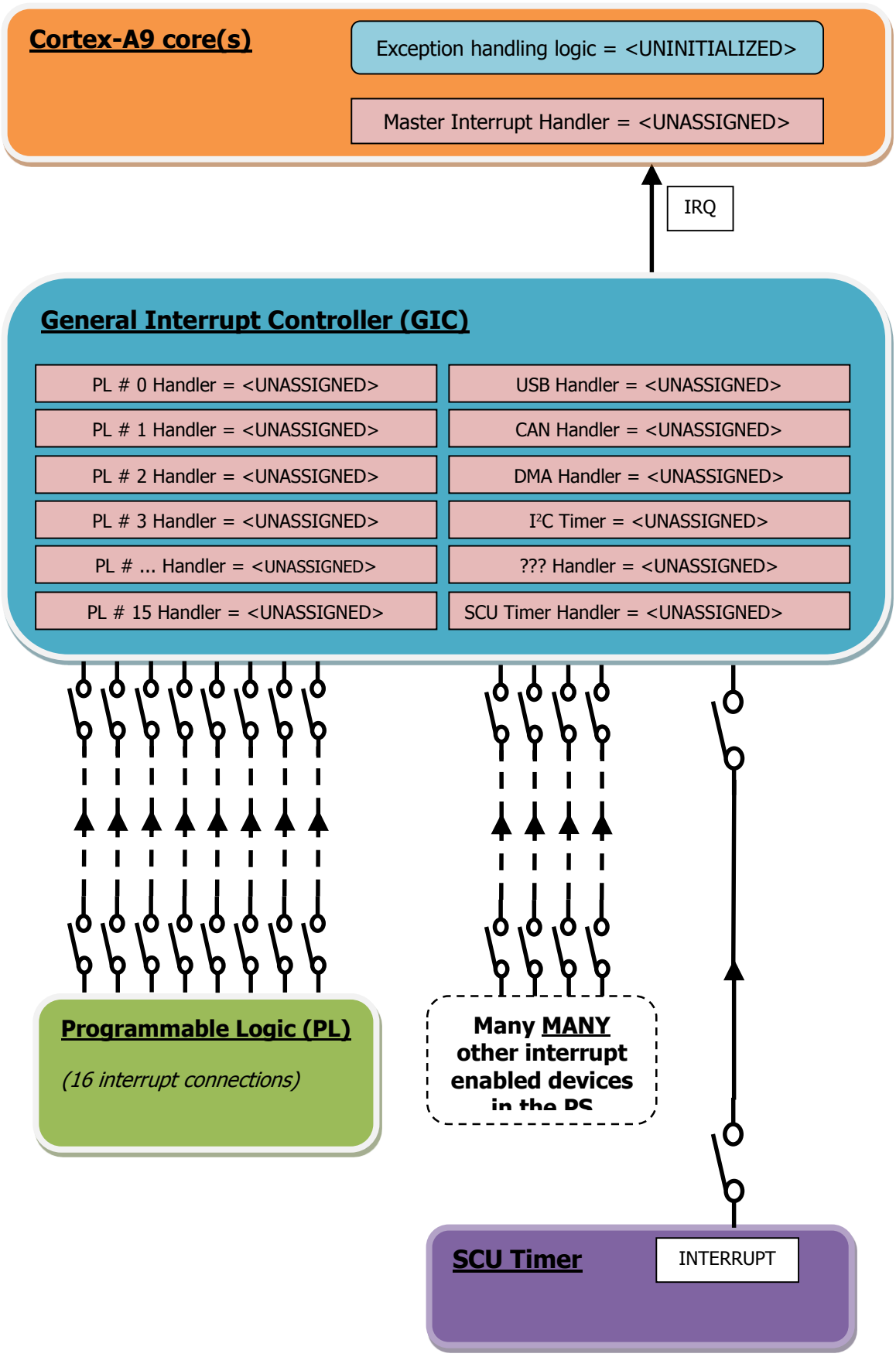
In a processor like the ARM Cortex-A9, all interrupts are managed by, and connected via, the general interrupt controller. So our first task is to create instances of the general interrupt controller (GIC) and the timer, and initialise them both. This is done in the same way as before, using the "Lookup_Config" and "CfgInitialize" functions.

```
XScuTimer my_Timer;
XScuTimer_Config *Timer_Config;

XScuGic my_Gic;
XScuGic_Config *Gic_Config;

Gic_Config = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);
XScuGic_CfgInitialize(&my_Gic, Gic_Config, Gic_Config->CpuBaseAddress);

Timer_Config = XScuTimer_LookupConfig(XPAR_PS7_SCUTIMER_0_DEVICE_ID);
XScuTimer_CfgInitialize(&my_Timer, Timer_Config, Timer_Config->BaseAddr);
```

We should now consider the layout of our processor system, and understand what needs to be done to configure it correctly. We shall start with a block diagram which shows a simplified view of a Zynq device at power-up. Please note the following things, which are all important to the success of this exercise:

- The Cortex-A9 CPU cores have internal exception handing logic, and this is disabled and initialised at power-up.
- There is one (standard) interrupt pin on the Cortex-A9 core, and there is a master interrupt handler which the CPU executes when receiving any interrupt request (IRQ). The handler is unassigned.
- The General Interrupt Controller (GIC) has the ability to manage many interrupt inputs, and has a table which allows interrupt handlers to be assigned to each incoming interrupt.
- Each interrupt input on the GIC has an enable "switch" that is disabled by default.

- Each peripheral / interrupt source has an output enable "switch", that is disabled by default.



Next we need to initialise the exception handling features on the ARM processor. This is done using a function call from the "xil_exception.h" header file.

```
Xil_ExceptionInit();
```

1

When an interrupt occurs, the processor first has to interrogate the interrupt controller to find out which peripheral generated the interrupt. Xilinx provide an interrupt handler to do this automatically, and it is called "XScuGic_InterruptHandler". To use this supplied handler, we have to assign it to the interrupt controller. The syntax is pretty scary, but it's the same for all designs so it can just be copied and pasted for every design that you create. The only item that needs to be changed is the name of the GIC instance at the end of the function (in our case "&my_Gic").

```
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT,
    (Xil_ExceptionHandler)XScuGic_InterruptHandler, &my_Gic);
```

2

We now need to assign our interrupt handler, which will handle interrupts for the timer peripheral. In our case, the handler is called "my_timer_interrupt_handler". It's connected to a unique interrupt ID number which is represented by the "XPAR_SCUTIMER_INTR". You'll find a list of these IDs in the "xparameters_ps.h" header file, and they cover all of the peripherals in the PS which generate interrupts. If you were dealing with an interrupt which came from a peripheral in the PL, you'd find a similar list in the "xparameters.h" header file.

3

```
XScuGic_Connect(&my_Gic, XPAR_SCUTIMER_INTR,
    (Xil_ExceptionHandler)my_timer_interrupt_handler, (void *)&my_Timer);
```

The next task is to enable the interrupt input for the timer on the interrupt controller. Interrupt controllers are flexible, so you can enable and disable each interrupt to decide what gets through and what doesn't.

```
XScuGic_Enable(&my_Gic, XPAR_SCUTIMER_INTR);
```

4

Next, we need to enable the interrupt output on the timer.

```
XScuTimer_EnableInterrupt(&my_Timer);
```

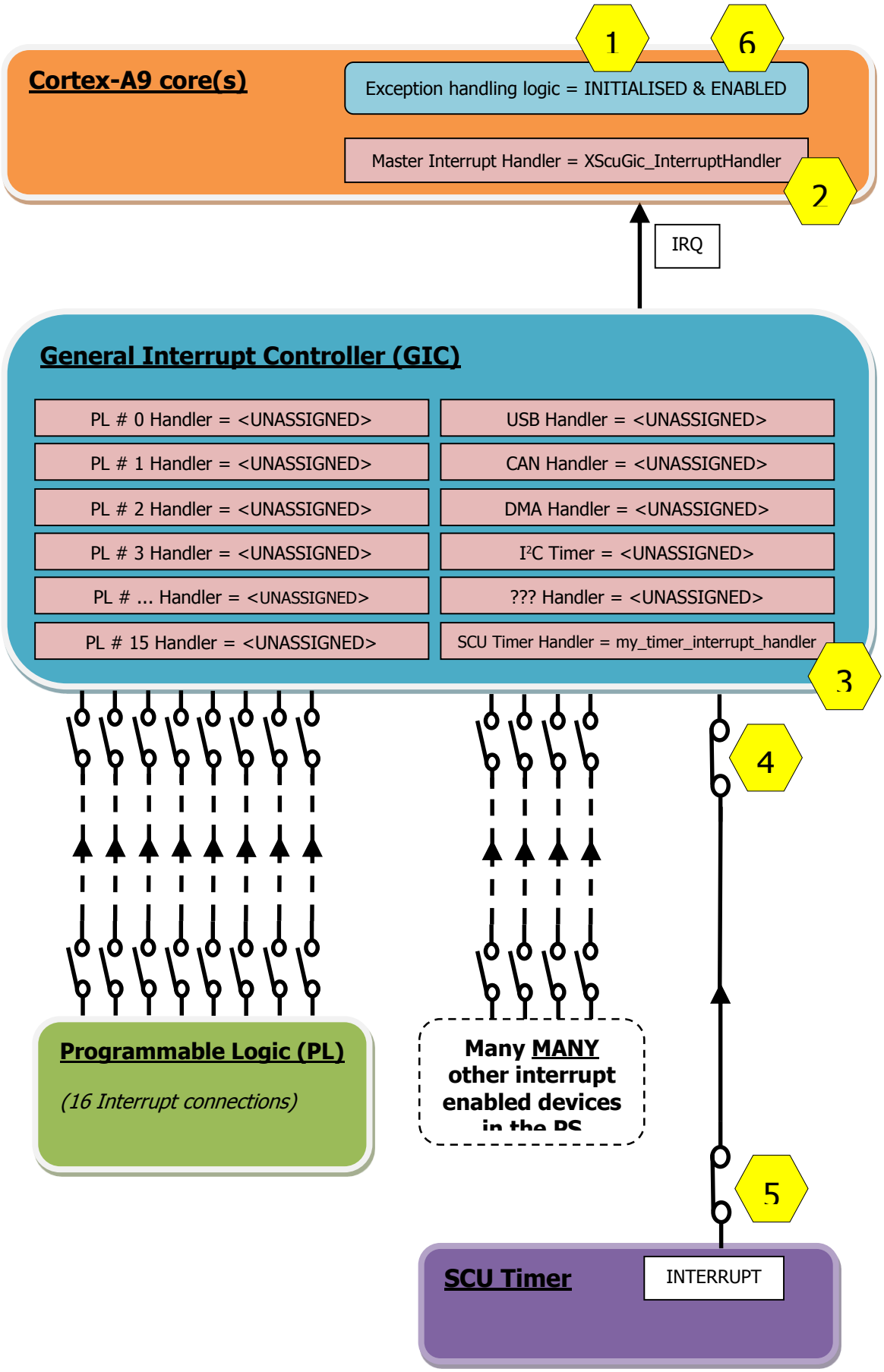
5

Finally, we need to enable interrupt handling on the ARM processor. Again, this function call can be found in the "xil_exception.h" header file.

```
Xil_ExceptionEnable();
```

6

That's the setup completed for the interrupt controller and timer. Let's review our work, and see what changes we've made to the system diagram. Use the yellow numbers to cross-reference each line of C code to the block diagram.


1
6
Cortex-A9 core(s)
Exception handling logic = INITIALISED & ENABLED
Master Interrupt Handler = XScuGic_InterruptHandler
2
IRQ
General Interrupt Controller (GIC)
PL # 0 Handler = <UNASSIGNED>
PL # 1 Handler = <UNASSIGNED>
PL # 2 Handler = <UNASSIGNED>
PL # 3 Handler = <UNASSIGNED>
PL # ... Handler = <UNASSIGNED>
PL # 15 Handler = <UNASSIGNED>
USB Handler = <UNASSIGNED>
CAN Handler = <UNASSIGNED>
DMA Handler = <UNASSIGNED>
I²C Timer = <UNASSIGNED>
??? Handler = <UNASSIGNED>
SCU Timer Handler = my_timer_interrupt_handler
3
4
Programmable Logic (PL)
(16 Interrupt connections)
Many MANY other interrupt enabled devices in the PS
5
SCU Timer
INTERRUPT

Now we need to write an interrupt handler, which is the function that will be called when the interrupt occurs. We've already decided on a name because we used it above; "my_timer_interrupt_handler".

It is considered good programming practice when coding an interrupt handler function to first check to make absolutely sure that the right peripheral has raised the interrupt. Lots of headaches can be created when a software engineer starts making decisions in their code because they believe one interrupt has occurred, when actually it's a completely different one. This problem is hard to detect because in most cases you have no way of checking which interrupt has occurred. Xilinx has solved this problem by including some clever code in their driver, using something called a "CallBackRef". We discussed before that there is a supplied interrupt handler which is provided by Xilinx for the interrupt controller, called "XScuGic_InterruptHandler". This handler is the one that calls our handler, but it also helpfully passes information about the driver instance which is relevant to the peripheral that generated the interrupt. This is the CallBackRef. In practice, this means that from the moment we enter our interrupt handler, the Xilinx drivers have already given us the details of which peripheral generated the interrupt.

This is what the basic structure of our timer interrupt handler looks like:

```
static void my_timer_interrupt_handler(void *CallBackRef)
{
	// Your code goes in here
}
```

We can clearly see the "CallBackRef" that is passed into the function, and we can use it to our advantage. Suppose we wanted to control the timer as part of our interrupt handler code? Without the CallBackRef we couldn't do it, because the original instance of "my_Timer" doesn't exist in the scope of this function. *(We declared it in main() and therefore it cannot be directly used in our interrupt handler because our handler is a different function)*

But we can declare a local copy of the "my_Timer" instance, and assign it the information provided by the "CallBackRef" :

```
static void my_timer_interrupt_handler(void *CallBackRef)
{
	XScuTimer *my_Timer_LOCAL = (XScuTimer *) CallBackRef;
}
```

The instance "my_Timer_LOCAL" is now an exact copy of the one in our "main" function, and we can control the timer in the same way as we did before. Any changes to "my_Timer_LOCAL" will also apply to the original copy of "my_Timer", because they are linked together using a pointer.

Returning to our previous comments on good coding practices, we now want to check to make sure that the timer really did generate this interrupt. To do that we can use a standard function from the timer's header file. All of the functions will work on "my_Timer_LOCAL" in the same way that they did in "main" for the original instance of "my_Timer", because they are declared as the same data type ("XScuTimer"):

```
static void my_timer_interrupt_handler(void *CallBackRef)
{
	XScuTimer *my_Timer_LOCAL = (XScuTimer *) CallBackRef;

	if (XScuTimer_IsExpired(my_Timer_LOCAL))
	{
		xil_printf("We are in the interrupt handler!!\n\r");
	}
}
```

The last task is to clear the interrupt condition in the timer peripheral. If we don't do this, then the interrupt condition will still be active. That means that the moment we exit from our interrupt handler, the processor will jump straight back into it again. The main section of our application would therefore never be executed. Again there is a supplied function call in the timer's header file to do this, so we need to add it to our interrupt handler :

```
XScuTimer_ClearInterruptStatus(my_Timer_LOCAL);
```

End of Exercise Challenge

- Create a new software application in the SDK "Project Explorer" pane.
- Using the guidance provided in this exercise, write a piece of software that will print a message to the UART at an interval of precisely 1Hz. The message should contain a counter which shows how many interrupts have been generated since the start of the application.

And when you've got that working...

- Add your code back in from the previous exercises to make the LED show the value of the push button, at the same time that the messages on the UART are getting displayed. If you do it correctly, then the messages will continue to be displayed at 1Hz no matter what you are doing with the rest of the code.

Something to think about at the end of this exercise...

I don't want to scare you, but you're already becoming quite proficient with Zynq-7000. The basics that you've covered so far are the foundations for the vast majority of Zynq embedded design projects. You understand how the Xilinx drivers work, you understand how interrupts work, and you understand how peripherals are added in the PL. You can do a lot with that! ☺

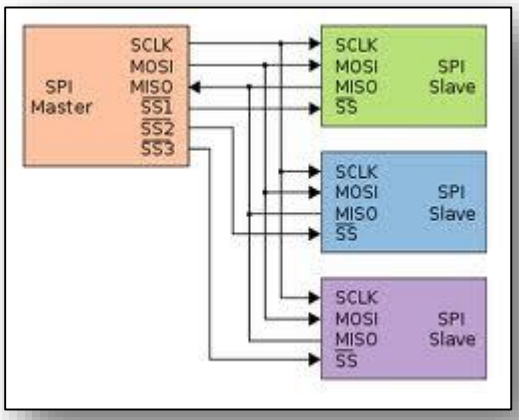
Exercise 9 - Talking to external components

You're nearly there! We have looked at all of the basic techniques that would be needed to create an embedded processor system with Zynq-7000. For this final exercise we will go beyond the pins of the Zynq-7000 device, and expand the design to communicate with an external device.

You may recall that in our previous exercises we added a peripheral (a Block RAM) to the PL section of the Zynq-7000 device, and we have already used it. At the same time we also configured the SPI controller in the PS, and we routed the IOs to be connected to MIO 10 ... MIO 13. In this exercise we will explore the use of the incredibly popular SPI protocol, and use it to control a device on an external "PMOD" daughter board.

But first a little bit of background...

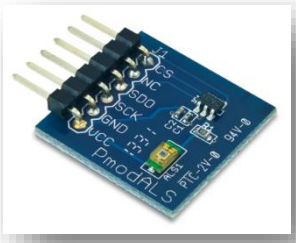
SPI (Serial Peripheral Interface) is a very well established standard interface in the digital electronics world, that allows master and slave devices to communicate using just four wires; SCLK, MOSI, MISO, and SS. SCLK is (perhaps obviously!) the clock connection. MOSI is an abbreviation for "Master Out Slave In". MISO is a similar abbreviation for "Master In Slave Out". SS is an abbreviation for "Slave Select", and is an active-low connection. Technically speaking, therefore, the signal should be named "$_n$SS" but it hardly ever is.



SPI is a simple protocol, whereby data is sent and received in a loop. Strictly speaking SPI has no concept of "read" and "write", but merely understands the concept of data "transfers". For every byte the master sends to the slave, the slave always sends a byte back again. As a result of this system, it is up to the master to "know" what data is expected back from any given slave, so that irrelevant dummy bytes (often 0x00 or 0xFF) can be ignored. For example, if the master is sending a command to the slave, but nothing is expected back, then the master must be programmed to ignore any bytes that come back. If the master sends a byte and the slave is configured to reply immediately, then the master must actually send two bytes (one for the command and one dummy byte) because the slave will reply with its data during byte number two.

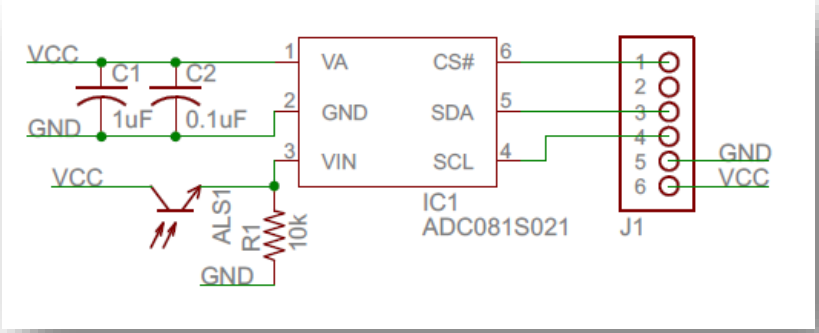
PMOD is an abbreviation for "Peripheral MODule" and was designed by Digilent Inc (http://www.digilentinc.com). Digilent hold the trademark for "PMOD", and details of

the licensing agreement to use the PMOD range of products can be found at http://www.digilentinc.com/Pmods/licensing.cfm .
PMOD is essentially a specification for connecting expansion modules using either 6 or 12 pin connectors. A very large number of PMOD expansion modules exist, and cover applications ranging from simple 7-segment displays, through to data converters, GPS receivers, SD card interfaces, joysticks, rotary encoders, real-time clocks, PS/2 connectors, and many other accessories.

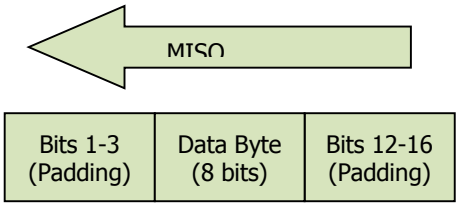


For this experiment we shall be using an ambient light level sensor PMOD board called the "PMOD ALS". The board is very simple, and features a few very simple components and one IC on the 6 pin PMOD board. Three of the pins are used for the SPI interface, there is one "no connect" pin, and the remaining two are used for Vcc and GND. *(Note: usually four pins are required for SPI, but on this PMOD there is no "MOSI" pin. More about that in a moment!)*

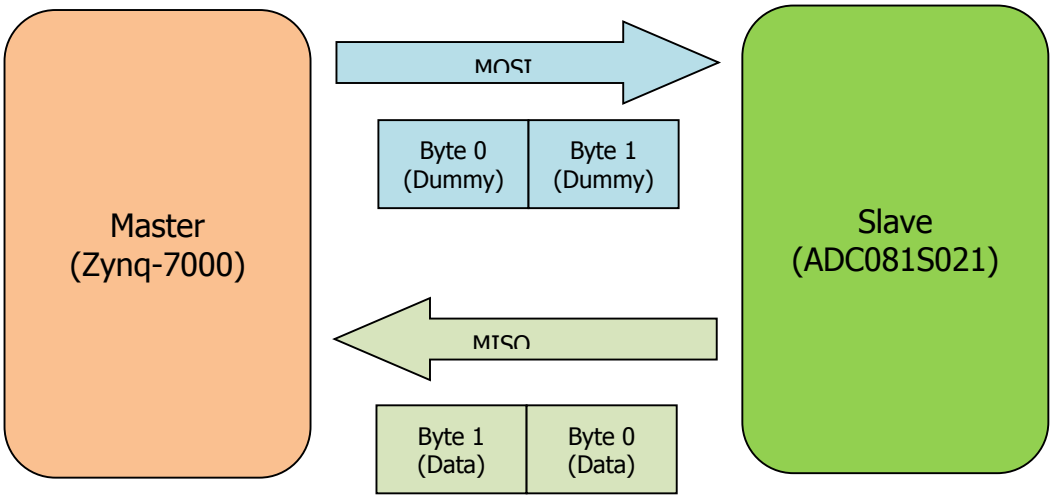
The IC is a Texas Instruments® "ADC081S021" analogue-to-digital converter, and its analogue input is connected to a simple potential divider with one of the "resistors" replaced by a Vishay® Semiconductor "TEMT6000X01" ambient light sensor.



The ADC081S021 is an 8-bit successive approximation A-to-D converter, with a serial data output in pure binary format. SPI based devices are usually implemented using addressable registers, and the user is expected to send the address of the register they wish to read from the device, and then listen to the reply. However the ADC081S021 is a much simpler device and does not have any addressable registers. The ADC only supports one operation… "read"! So in order to use the device, the user simply pulls the CS# pin low during the first cycle of the first byte, and the ADC will then send a byte of data via its SDATA pin which represents the value on the analogue input pin, "VIN". The format of the data sent by the ADC is three leading zeros, followed by eight data bits (MSB first), followed by trailing zeros.



To read the data sent by the ADC, we need to generate a two byte SPI transaction. As we discussed before, SPI communications are circular and therefore the SPI master (which in our case is the Zynq-7000 device) must send two bytes to the ADC on the MOSI pin in order to receive two bytes back on the MISO pin. It doesn't matter what data the master sends because the ADC isn't interested. In fact the MOSI pin isn't even connected to the ADC. The important point is that sending two bytes will generate the 16 clock edges that the ADC requires.



From the hardware perspective the design is ready to go, because all of the MIO pin mappings were added to the design in our previous exercises. Plug the ADC081S021 module into the upper row of pins on connector J5 of the MicroZed board. (The "CS" pin on the PMOD ALS module should be connected to pin 1 of the J5 socket on the MicroZed)

We will use the device in interrupt mode, and due to the complexity of the C code a template is supplied for you. The code follows all of the same rules that we saw from our previous exercises, using the same principles to set up the driver and initialise interrupts.

The C code starts a transfer and then sets up a global variable and a while loop that halts execution until the SPI transfer has completed. The SPI peripheral can generate an interrupt for a number of reasons (completion of a transfer, error conditions, etc.) and so the interrupt handler has to check to see what event caused the interrupt before taking any action. As a result, the interrupt handler in this exercise is slightly more advanced; because it first has to do some checking. If the interrupt condition is of the desired result (i.e. "transfer complete") then the handler sets the global variable back to the original value and the main section of the application can continue.

Global variables are also used to implement two buffers for the SPI peripheral; one read buffer and one write buffer. These buffers are then passed to the SPI drivers and are sent / received, byte by byte, during an SPI transfer.

End of Exercise Challenge

Straight to the challenge this time!

- Create a new software application project in the SDK.
- Use the supplied code for Exercise 9 as a template to implement your design. You will need to fill in the blanks using your newly found knowledge of Zynq-7000 embedded design.
    - You will need to set up the interrupt system and test it.
    - You will need to load the write buffer with data to send to the ADC081S021 device via SPI. *(Think carefully!!)*
    - Only one SPI transaction is required; to read the two bytes of data from the ADC.
    - You will need to capture the data from the read buffer, and pass that to the section of the code which will display the light level. The read buffer has been created for you in the code template.
- Use the debugger to get the code working.
- I have left some useful features in the code (unused) to help you. These include functions like "display_buffers()" which could be called to help you debug what you're doing.
- Depending on your level of C coding experience, this might be harder than it looks. Make good use of caffeine!

Further fun with the ambient light sensor

When your code is up and running, you can experiment with using the data that the ambient light sensor provides. Earlier in the workshop we saw how to control an LED; expand your code to make the LED illuminate when darker conditions are detected by the ambient light sensor.

Exercise 10 – Autonomous Boot

We now have a design that works on the board, and is advanced enough for us to call it "finished". We now want to make the MicroZed board boot automatically from power on.

To make this happen, we need to create a "First Stage Boot Loader" (FSBL), and then combine our application software, the FSBL, and the Programmable Logic hardware bitstream into one flash image file. Happily, Xilinx have made this a very easy flow and it will allow you to get your board booting without any trouble.

Creating the FSBL (First Stage Boot Loader)

From the Xilinx SDK tool, choose "File → New → Application Project" from the menus. Give the project a name, for example "FSBL", set the Board Support Package options to create a new Board Support Package for this application, and click "Next". At the bottom of the list of templates choose "Zynq FSBL" and click "Finish".

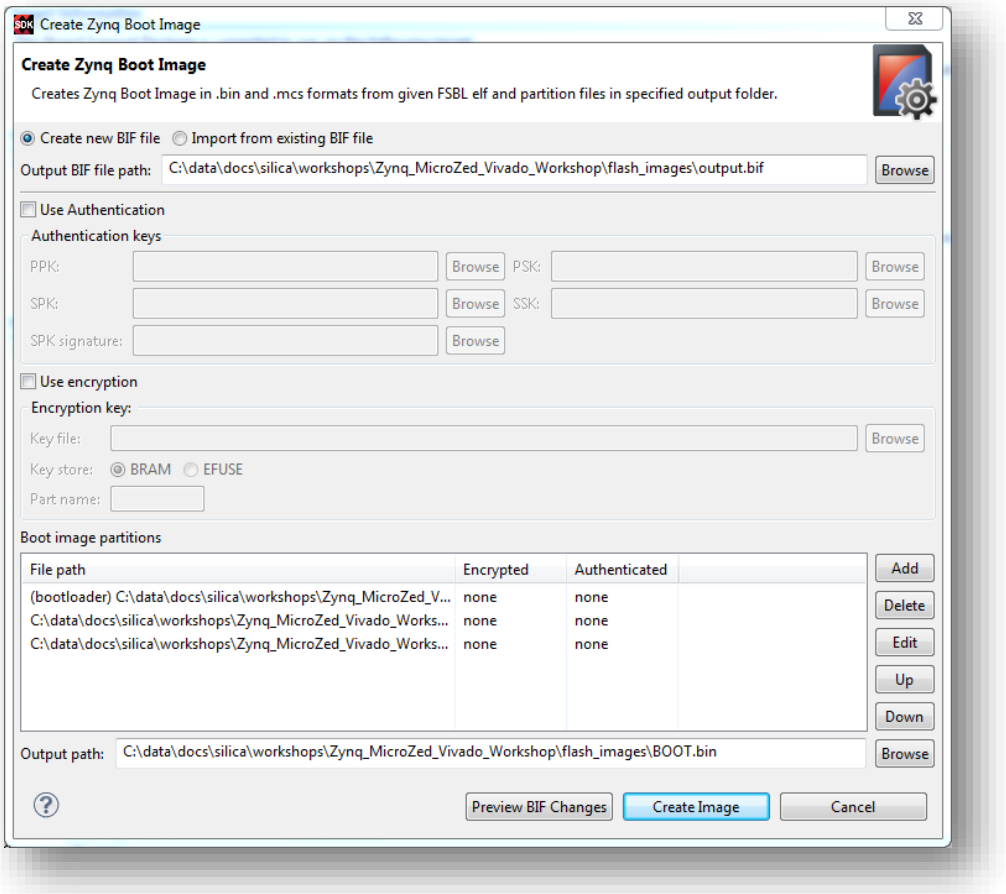
Note: The Xilinx FSBL requires the inclusion of some libraries to access the FAT file system on the SD card. This is the reason that a new Board Support Package must be generated, because the tools automatically include the required libraries.

The FSBL project will be created next to all of the other software application projects. No modifications are necessary – the code just works.

Creating the Boot Image

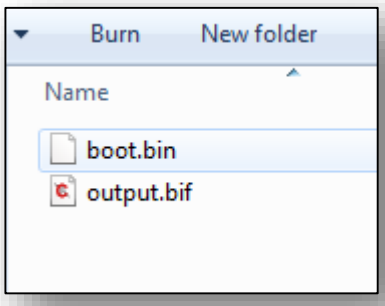
- Use a windows explorer session to create a folder called "flash_images" on your PC. It doesn't really matter where, just remember where you put it.
- From the menus, choose "Xilinx Tools → Create Boot Image".
- Choose the "Create new BIF file" radio button, and then browse to the folder you created a moment ago. Choose an output BIF filename of "output.bif".
- At the bottom of the window, click the "Add" button next to the Boot Image Partitions table. Browse to the SDK workspace, and into the "<project_name>\<project_name>.sdk\FSBL\Debug" folder. Choose the "FSBL.elf" file and click "Open". Check that the partition type is set to "bootloader" and click "OK".
- The second location in the flash image table is for the Programmable Logic bitstream. Add another partition to the table, and browse to the "<project_name>\<project_name>.sdk\<block_diagram_name>_hw_platform_0" folder, and choose the .BIT bitstream file that has been generated by your Vivado project. Check that the partition type is set to "datafile" for this partition.
- Click the "Add" button on the right of the table for a third time, and browse to the .elf file in your "Exercise 9" directory. Add this to your table of boot partitions as a "datafile" type partition.

- At the bottom of the window, check that the "output path" for the boot image is in the "flash_images" folder that you created at the beginning of this exercise. Use the default filename "boot.bin".



Open a Windows Explorer session (or an alternative file system application if you are using a different operating system on your host machine), and navigate to the location of the boot image that you have just created. You will see two files (.bif and .bin). The .bif file contains the settings for the "Boot Image Generator" wizard, and can be used if you ever want to re-create the boot image using the same settings as before. The second file (.bin) is the binary image that is used if you are configuring the Zynq-7000 device using a SD Card.

Locate an SD card and connect it to your PC via the SD card slot, or a card reader. Open another Windows Explorer session, and navigate to the SD card.



- Using a "drag and drop", or cut and paste operation, copy the .bin file to the SD card.
- When booting in SD card mode, Zynq-7000 devices look for a reserved filename called

"boot.bin". Rename the .bin file on your SD card to "boot.bin" and then safely eject the SD card.

The moment of truth!...

- Switch off the MicroZed board, and disconnect the JTAG cable.
- Insert the SD card into the slot on the MicroZed board, taking care to push it all the way in.
- Switch on the power, and wait for a few seconds. Observe the UART terminal and you will see your application running on the board.
- Celebrate! ☺