Introduction

Most modern FPGAs are large enough to contain a design including a microcontroller. A microcontroller is somewhat different than a microprocessor than you might find in a desktop PC or laptop. Microprocessors are general purpose computing devices while microcontrollers share most of the same architecture but are targeted to control other devices which interact with the “real world”.

The purpose of including the microcontroller is to provide a way for the FPGA to more easily interact with the real world. While the time critical aspects of a design are handled by custom logic, the microcontroller can act as a supervisor or state-machine, controlling and monitoring the custom logic.

The ZedBoard uses an FPGA from the Xilinx Zynq family and has what is known as a “hard core” microcontroller (technically two of them – i.e. dual-core). This microcontroller is based on the ARM (Advanced RISC Machine or Acorn RISC Machine) Cortex architecture. Xilinx FPGAs such as the Kintex-7 family found on the KC705 evaluation board do not contain silicon dedicated to a processor and so implement a “soft core” processor. Typically this is a Microblaze architecture which shares many attributes with the ARM architecture but is not strictly a RISC (Reduced Instruction Set Computer) architecture. For the purposes of this module you don’t need to now details of how the processor works. But there are nearly 40 years of documents to read on the development of the ARM processor family.

Inside an FPGA containing a processor, soft core or hard core, you further divide the FPGA into the PL for Programmable Logic portion and the PS for Processor (Processing) System. Inside the Xilinx Vivado environment you define how the PS will be constructed and how all aspects of the PL interact with the PS. Then inside the Xilinx Vitis environment you create the necessary procedural code to run on the processor, i.e. C or C++. This includes writing low-level drivers to interact with custom PL which in turn interacts with the “outside world”.

We will design a system that interacts with the operator via the Zync processor core and a USB serial connection while at the same time controlling LEDs and generating test signals on one of the ZedBoard’s connectors.

Design

Two LEDs will blink. One is controlled entirely by the PL and blinks at a rate of 2Hz. The second blinks under the control of the PS and ultimately the operator via the USB serial connection.

The PS will also generate a regular pattern of 0’s and 1’s to be sent out on JA4 to be returned on JA1 to demonstrate how the PS and PL can be used to test an external signal path. This signal will have a minimum operating speed of 100kbit/sec as controlled by a timer in the PS.

Create Vivado Project

Create a Vivado project in the usual manner, selecting a reasonable project name such as “module2_blinky”. Other than specifying the ZedBoard as the target it is not necessary to add any other design files at this time. This example will be using VHDL and C for all coding examples.

This module introduces the concept of a block design for the top-most level with custom VHDL as needed for aspects of the PL and the PS implemented using Xilinx IP (Intellectual Property) blocks.

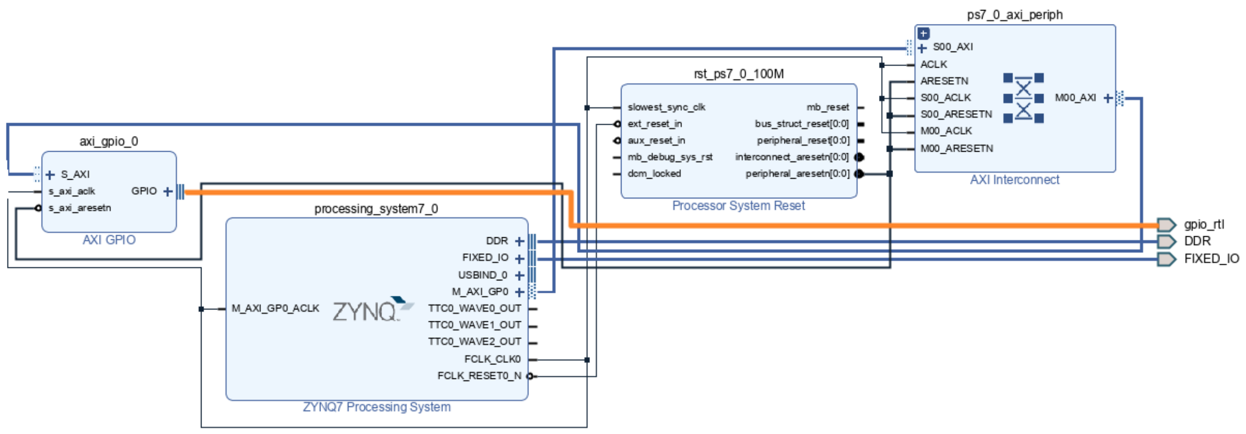
Once Xilinx has created the project, click on "Create Block Diagram" under the IP Integrator. The design name can be changed to "module2_blinky" if desired.

A blank diagram canvas appears with "This design is empty. Press the + button to add IP" in the center. We will place the following items on the canvas using the "+" button:

- ZYNQ7 Processing System
- AXI GPIO

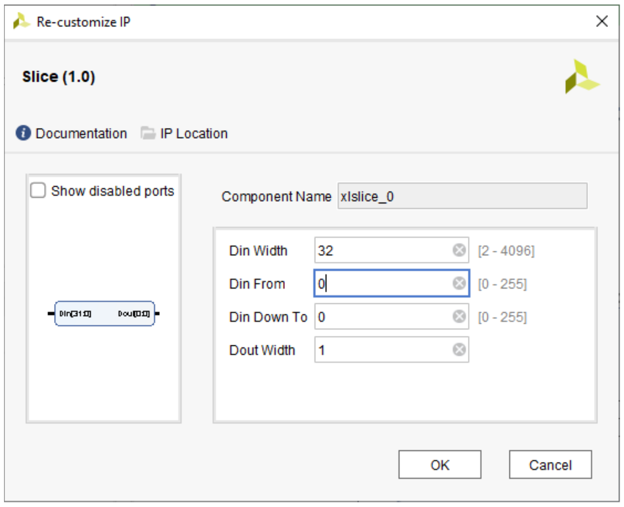
A green bar will appear, this is the Designer Assistance bar and it will allow the Block Automation and Connection Automation features of Vivado to run. When presented with both options, run the Block Automation first. Accept the defaults in this case as we have board presets already present for the ZedBoard.

The Connection Automation will require that you select what is to be automatically connected/configured. Select all items. Then click once on the GPIO and under its options pick "Custom". The options for "S_AXI" should be left to their defaults.



The final diagram should look similar to this. Highlighted in orange is a trace to delete along with the output port called "gpio_rtl". Notice that when deleted, the Designer Assistance bar returns. But this time we are going to define our own connections.

Using the "+" again, add a block called "Slice". Double clicking on the new block one can change its attributes before completing the connection. We are going to use this to "slice" off one bit of a 32-bit register provided by the GPIO block and connect it to an LED on the board.



Slice (1.0)

Show disabled ports

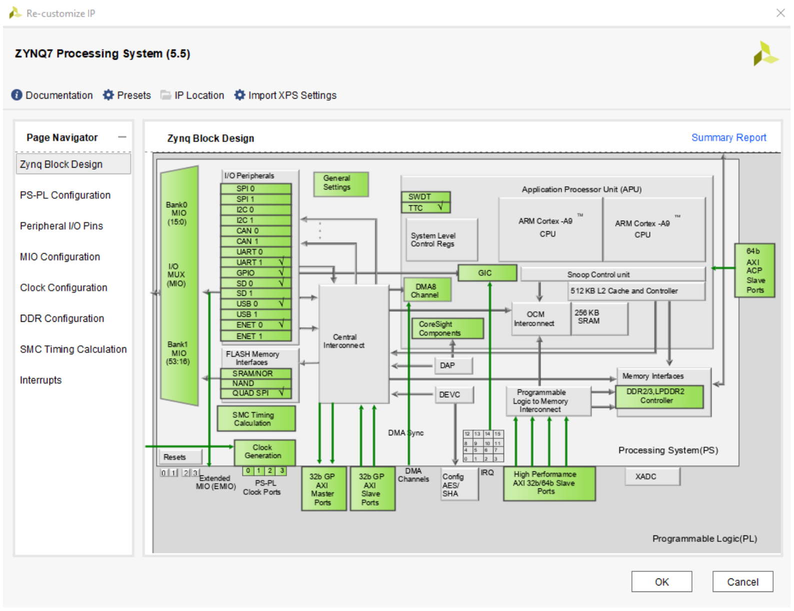
Component Name xlslice_0

Din Width	32	[2 - 4096]
Din From	0	[0 - 255]
Din Down To	0	[0 - 255]
Dout Width	1	

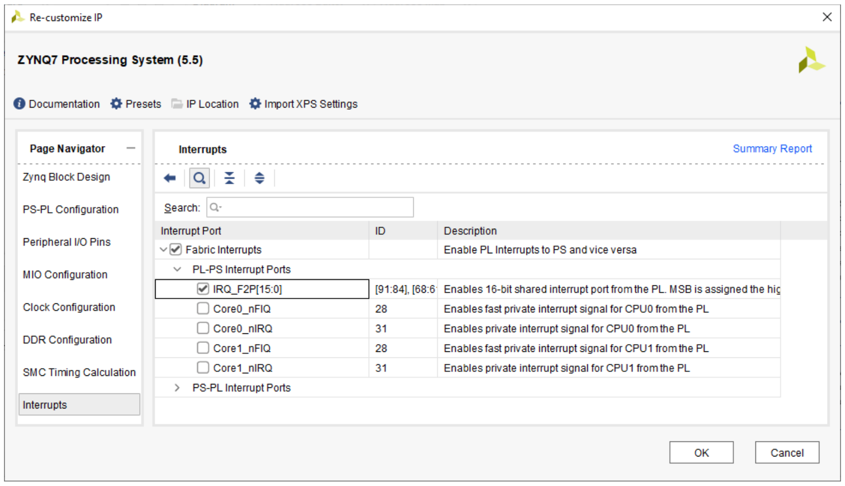
By using the defaults, we will be selecting the least significant bit of the 32-bit register. Click OK to accept and then expand the port labeled GPIO on the GPIO block by clicking its "+" sign.

Use the mouse to drag a connection from the _O port to the Din port of the Slice. Finally right-click on the Dout port of the slice and "Create a port" with the name "LD0", direction "OUTPUT" and type "OTHER". LD0 will be the LED we blink with the PS.

The final step is to add timer hardware to the design and tell it to signal the Zynq via an interrupt. This will be programmed to create our blinking LED under control of the PS. Use the "+" button to add an AXI Timer to the design, use the connection automation Wizard to complete the connection to the AXI bus. Now double click on the ZYNQ7 Processing System box to see all the options for the processor.



We are interested in the list on the far left to configure the Interupts. In particular we want the interrupt to travel from the PL to the PS so select the appropriate option.


Re-customize IP

ZYNQ7 Processing System (5.5)

Documentation Presets IP Location Import XPS Settings

Page Navigator: Zynq Block Design, PS-PL Configuration, Peripheral I/O Pins, MIO Configuration, Clock Configuration, DDR Configuration, SMC Timing Calculation, Interrupts

Interrupts — Summary Report

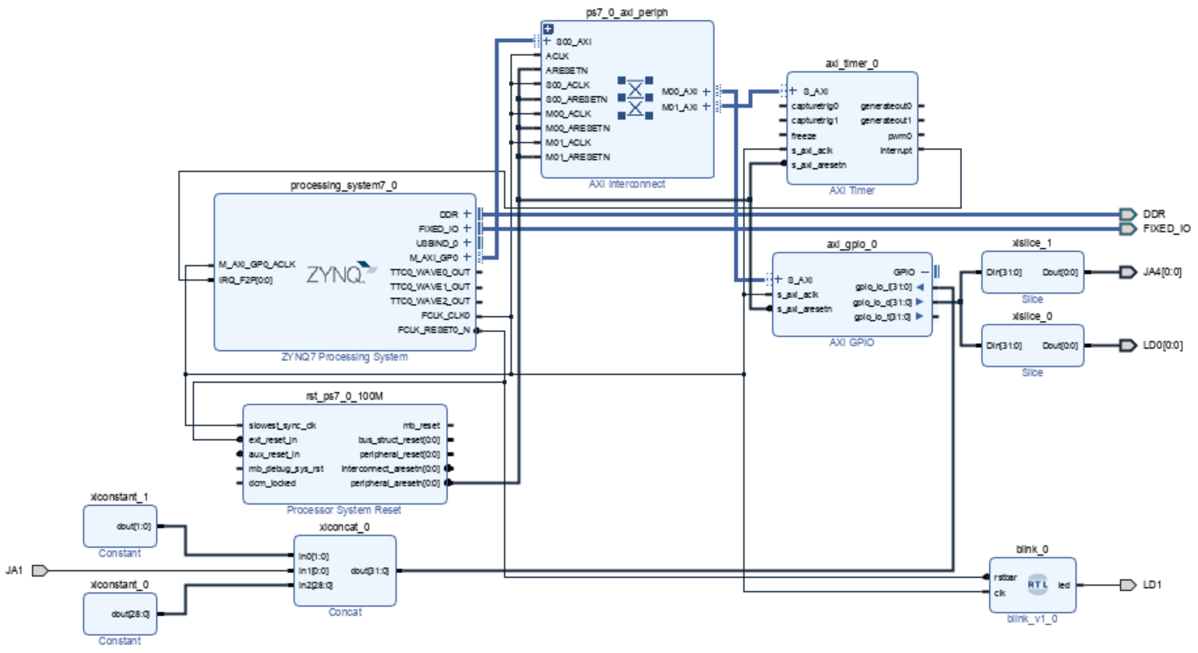
Search:

Interrupt Port	ID	Description
☑ Fabric Interrupts		Enable PL Interrupts to PS and vice versa
PL-PS Interrupt Ports		
☑ IRQ_F2P[15:0]	[91:84], [68:6	Enables 16-bit shared interrupt port from the PL. MSB is assigned the hig
☐ Core0_nFIQ	28	Enables fast private interrupt signal for CPU0 from the PL
☐ Core0_nIRQ	31	Enables private interrupt signal for CPU0 from the PL
☐ Core1_nFIQ	28	Enables fast private interrupt signal for CPU1 from the PL
☐ Core1_nIRQ	31	Enables private interrupt signal for CPU1 from the PL
PS-PL Interrupt Ports		

OK Cancel

A new port will appear on the left side of the ZYNQ block, IRQ_F2P[0:0]. Use the mouse to drag a connection from there to the interrupt output of the AXI Timer block.

The final pieces of pre-defined IP block to add are another slice to pick off D1 and route it to an output pin (JA4), a concatenation block and two constant blocks. Using their customization options (double click on the block) the final configuration should resemble this

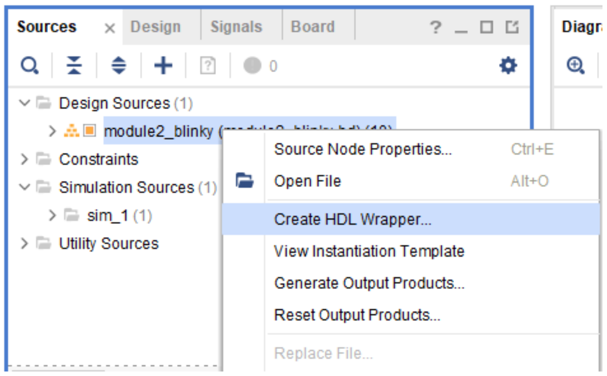


The JA1 pin is being placed onto the input side of the GPIO and is mapped to the D(2) position to not interfere with the location of the LED or JA4.

Adding Custom IP

The missing part of our design is a way to blink an LED entirely under the control of the PL. For this we need to add our own VHDL code to the diagram.

But before adding the code or attempting a compile, we need to tell Vivado that our block diagram is the top most file of the hierarchy. To do this, locate the block design (.bd) file in the sources list and right click to select "Create HDL Wrapper" from the context menu.



Accept the default to let Vivado manage wrapper and auto-update. After a short time Vivado will return and the design sources list will indicate the top most file is now a VHDL file called "module2_blinky_wrapper.vhd".
Using the "+" button of the Sources tab, create a new source called "blink" and add it to the project. When prompted for the signals to place in "blink", do not add any and just accept the defaults with OK.

The port list will be blank but we will manually populate it as:

```
entity blink is
    port (
        rstbar : in std_logic;
         clk : in std_logic;
         led : out std_logic
    );
end blink;
```

And for the architecture portion the code will be a counter that counts from 0 to 24,999,999 and at each roll-over toggles the state of the LED. 24,999,999 is selected as the magic number because the clock rate (clk) will be 100MHz as supplied by FCLK_CLK0 of the Zynq processor. Technically, this isn't making the blinking of this LED completely independent of the PS but it is a reliable way to get a clock as well as the reset signal.

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

entity blink is
    port (
         rstbar : in std_logic;
         clk : in std_logic;
         led : out std_logic
    );
end blink;

architecture behavioral of blink is

    signal iled : std_logic;
    signal counter : integer;

begin
```

```
    process(clk)
    begin
        if rising_edge(clk) then
            if rstbar = '0' then
                iled <= '0';
                counter <= 0;
            else
                counter <= counter + 1;
                if counter = 24999999 then
                    iled <= not iled;
                    counter <= 0;
                end if;
            end if;
        end if;
    end process;

    led <= iled; -- map signal to output

end behavioral;
```

This design is simple enough we could skip simulation. However to illustrate the steps, the next stage is simulating our VHDL. Generally avoid trying to simulate the operation of a processor as it will be very slow. The PS side of a project is often easier to debug in place.

Simulating PL Code

To simulate the operation of our counter we need to create a testbed. In the Sources tab, either right click on Simulation Sources and Add Sources or use the "+" button and select "Add or create simulation sources". Create a file and give it the name "blinky_tb" assuming the VHDL file to be simulated is called "blinky.vhd". The appended "_tb" stands for "TestBench". Do not bother to add any port names. When the file is created, open it in the editor.

There are several good on-line tutorials discussing the creation of testbenches as well as on-line resources for the creation of basic testbenches. One good one is maintained by Doulos Traning and can be found at https://www.doulos.com/knowhow/perl/vhdl-testbench-creation-using-perl/

Use this on-line generator and replace the contents of test_tb.vhd with its output.

The only portion of code we need to edit or create in this test is the clock speed, confirming that clock_period is set to 10ns (100MHz clock). Notice the data type is "time" and the units are "ns" for nano-seconds. VHDL supports the use of this datatype only in simulation. The data type "time" is unsynthesizable.

Next we need to add code for the initialization to control the state of the reset (rstbar) line. The reset signal coming from the ZYNQ processor is active low we will tell the test bench to hold the line low for several periods of the clock then release it high and allow the code to run freely. The portion of code to add is highlighted below.

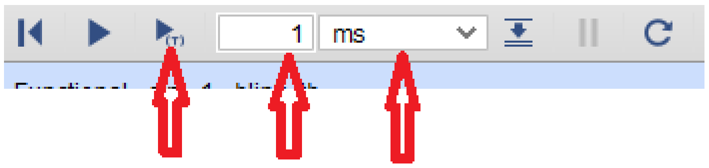
```
-- testbench created online at:
--   https://www.doulos.com/knowhow/perl/vhdl-testbench-creation-using-perl/
-- copyright doulos ltd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity blink_tb is
end;
```

```
architecture bench of blink_tb is
  component blink
      port (
          rstbar : in std_logic;
          clk : in std_logic;
          led : out std_logic
      );
  end component;
  signal rstbar: std_logic;
  signal clk: std_logic;
  signal led: std_logic ;
  constant clock_period: time := 10 ns;
  signal stop_the_clock: boolean;
begin
  uut: blink port map ( rstbar => rstbar,
                        clk    => clk,
                        led    => led );
  stimulus: process
  begin
    -- put initialisation code here
    stop_the_clock <= false;
    rstbar <= '0';
    wait for clock_period * 10;
    rstbar <= '1';
    -- put test bench stimulus code here
    wait; -- forever
  end process;

  clocking: process
  begin
    while not stop_the_clock loop
      clk <= '0', '1' after clock_period / 2;
      wait for clock_period;
    end loop;
    wait;
  end process;

end;
```

Next the simulation is started. First locate “blink_tb.vhd” in the Simulation Sources list, right click and make it the top module by clicking “Set as Top” in the context menu. By default, simulations run for 1000ns unless we change an entry by right clicking on “Run Simulation”, “Simulation Settings” and “xsim.simulate.runtime” on the Simulation tab. Typically this is left alone and if the simulation needs to run longer it can be handled from within the simulation interface.

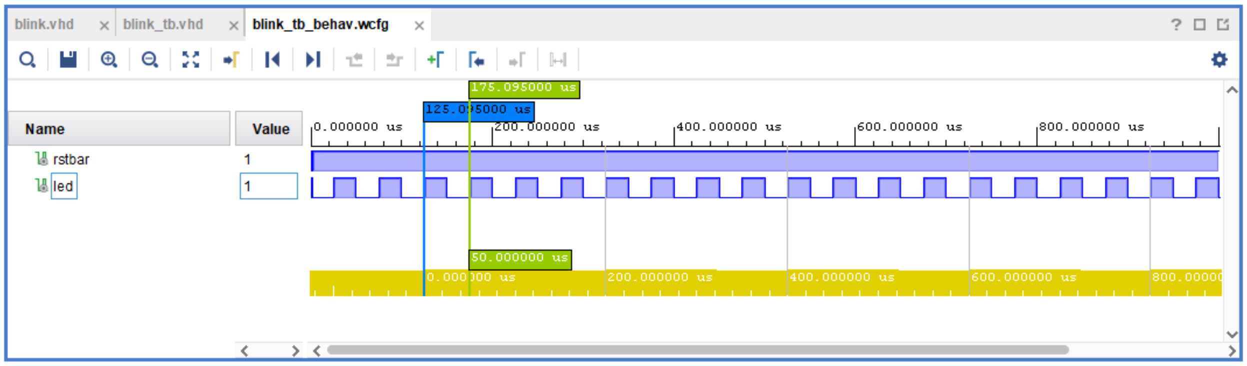
Right click “Run Simulation” and select “Behavior Simulation”. When the screen appears with the various signal and busses of the test bench you can change what is displayed. To reduce clutter, remove the signals “stop_the_clock” and “clock_period” as they do not change. Also, the clock itself can sometimes be removed if it is distracting.

This simulation needs to run for several seconds of simulation time to show that the LED is blinking at a 2Hz rate. But even with this simple code, simulations can take a long time. To run the simulation longer without restarting, use the “run for” button and the time and unit dialog box.



Because we aren't worried about simulating the real-world timing, just the behavior we can speed up the simulation by changing the 24,999,999 to a smaller number like 2,499 for a blink rate of 20kHz.

If the "blink_tb.vhd" tab isn't open, it can be re-opened from the sources tab and the file edited and saved. To restart the simulation use the "Relaunch Simulation" button, the circled arrow. The simulation will re-compile and run for 1000ns. Then additional time can be added and the results checked using various means such as the cursor and markers.

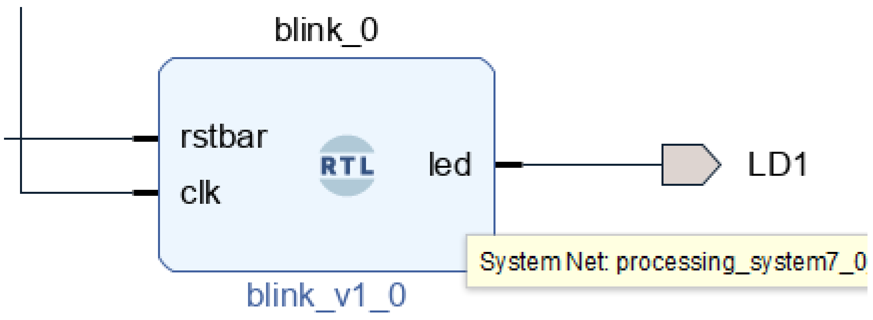


<u>As an exercise, how could the VHDL code be modified to allow the magic number to be entered as a parameter? Search VHDL documentation for the use of generic ports.</u>

Adding VHDL Modules to Block Diagram

To add the custom VHDL to the block diagram, the file name can be selected from the Design Sources list and dragged onto the block design canvas. Or by right-clicking on the block design canvas and selecting "Add Module", the code can be selected from a list of available candidates.

Once added, connect the clock to FCLK_CLK0 and rstbar to FCLK_RESET0_N using the mouse. Create an output port called LD1 and connect it to the led port. More than one copy of the VHDL can be added to a block diagram so long as there are no resource conflicts.



Because a new output, LD1, has been added the constraints file must be edited. Either by manually adding the pin and any necessary bank configuration or using the I/O Ports tab of the synthesized or implemented designs.

If entering the constraints manually, it would look like:

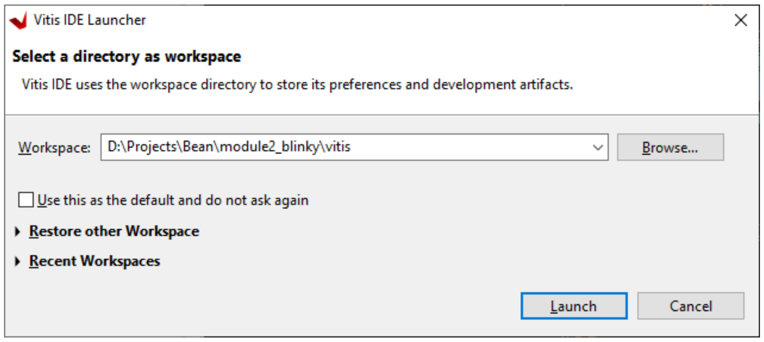
```
set_property PACKAGE_PIN T21 [get_ports LD1]
set_property IOSTANDARD LVCMOS33 [get_ports LD1]
```

Generating and Export Bitstream, Starting Vitis

To create executable code for a PS, first the entire PS+PL project must be compiled and a bitstream generated. Then the hardware is exported to create an XSA (Xilinx Support Archive) hardware design file. This file contains everything the Vitis software will need to know about exactly HOW the PS is configured and how it interacts with the PL.

Once a bitstream has been generated, from the File menu, select Export | Export Hardware and when prompted, include the bitstream. A pre-synthesis export is useful only for large team development. Allow the defaults for the name and location, “module2_blinky_wrapper” for example. If repeated compile and export cycles take place, Vivado will warn about overwriting the file. Usually this is acceptable.

Start Vitis either from the shortcut created when the software was installed or from within Vivado under the Tools menu.


Vitis IDE Launcher

Select a directory as workspace

Vitis IDE uses the workspace directory to store its preferences and development artifacts.

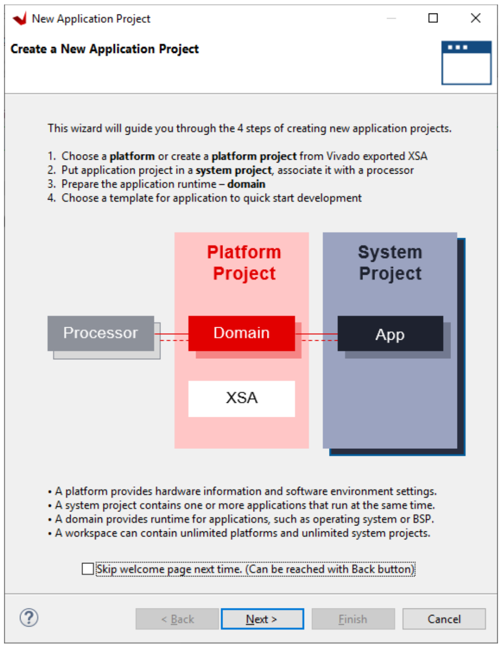
Workspace: D:\Projects\Bean\module2_blinky\vitis

The “Workspace” should be a location within your project’s directory. In this case I’ve added a subdirectory “Vitis” to contain everything associated with code development of the PS.

Creating a Vitis Project

Once launched, the next step is to “Create an Application Project”. You could alternately create a “Platform Project” first but you would still need to work through the steps of creating an application project. An application project is the code you specifically write but will always be paired with a platform project, the code generated from the exported bitstream and XSA file. If a platform project doesn’t exist, it is automatically generated during the creation of an application project.

The Vitis wizard will walk through the basics,


New Application Project

Create a New Application Project

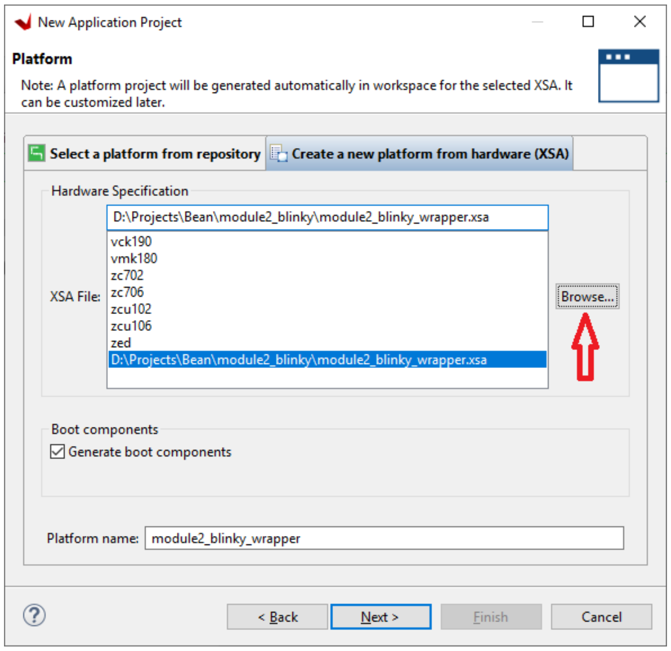
This wizard will guide you through the 4 steps of creating new application projects.

1. Choose a **platform** or create a **platform project** from Vivado exported XSA
2. Put application project in a **system project**, associate it with a processor
3. Prepare the application runtime – **domain**
4. Choose a template for application to quick start development

• A platform provides hardware information and software environment settings.
• A system project contains one or more applications that run at the same time.
• A domain provides runtime for applications, such as operating system or BSP.
• A workspace can contain unlimited platforms and unlimited system projects.

☐ Skip welcome page next time. (Can be reached with Back button)

To create the platform, you need to guide the Wizard to the location of the exported XSA file. There are a handful of pre-generated platforms installed but none of them is specifically for the ZedBoard. Use the “Browse” button on the “Generate a new platform from hardware (XSA)” tab. Also leave the Generate Boot Components option checked.


New Application Project

Platform

Note: A platform project will be generated automatically in workspace for the selected XSA. It can be customized later.

Select a platform from repository
Create a new platform from hardware (XSA)

Hardware Specification

XSA File: D:\Projects\Bean\module2_blinky\module2_blinky_wrapper.xsa

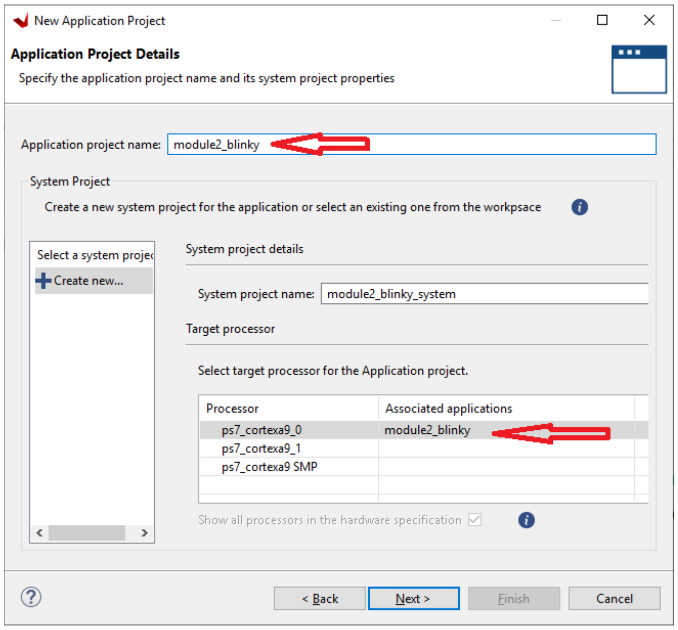
vck190
vmk180
zc702
zc706
zcu102
zcu106
zed
D:\Projects\Bean\module2_blinky\module2_blinky_wrapper.xsa

Boot components

☑ Generate boot components

Platform name: module2_blinky_wrapper

In the next step, give the application project a reasonable name. Because the ZYNQ contains a dual core ARM Cortex processor, you have the choice to assign it to processor 0 or 1. Unless you have a good reason or absolutely need dual-processing, choose processor 0 as it makes any later boot operations a bit easier being the default boot processor.


New Application Project

Application Project Details

Specify the application project name and its system project properties

Application project name: module2_blinky

System Project

Create a new system project for the application or select an existing one from the workspace

Select a system project: Create new...

System project details

System project name: module2_blinky_system

Target processor

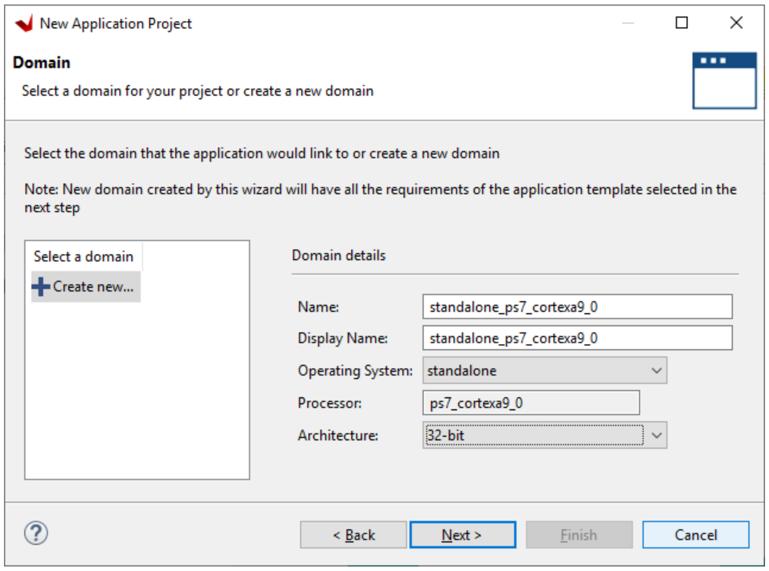
Select target processor for the Application project.

Processor	Associated applications
ps7_cortexa9_0	module2_blinky
ps7_cortexa9_1	
ps7_cortexa9 SMP	

Show all processors in the hardware specification ✓

< Back | Next > | Finish | Cancel

The "domain" for the processor is another way of thinking about how the processor should be implemented and what sort of operating system is desired. In our case we have only a 32-bit processor (ARM Cortex) and we don't want the additional overhead of an operating system such as Linux. So we select "standalone" (default) and "32-bit" for Operating System and Architecture. You could change the domain names but the auto generated names are fine.


New Application Project

Domain

Select a domain for your project or create a new domain

Select the domain that the application would link to or create a new domain

Note: New domain created by this wizard will have all the requirements of the application template selected in the next step

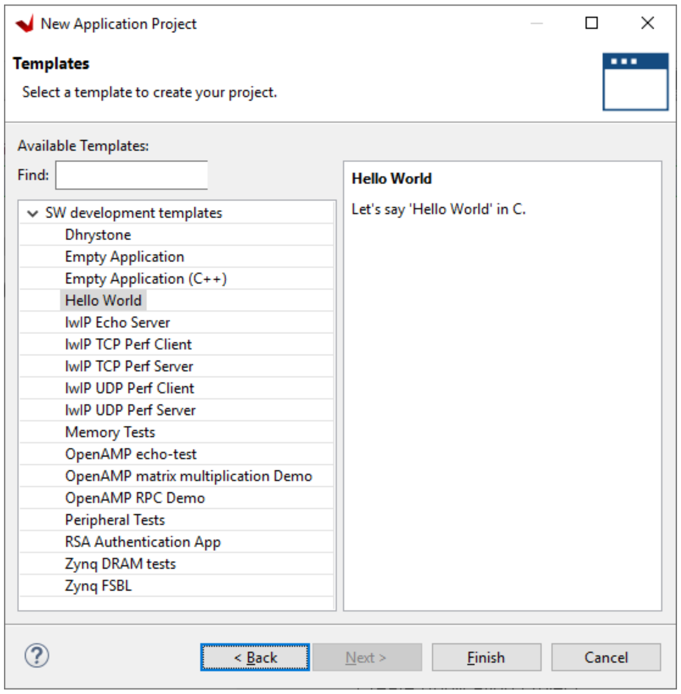
Select a domain: Create new...

Domain details

Name:	standalone_ps7_cortexa9_0
Display Name:	standalone_ps7_cortexa9_0
Operating System:	standalone
Processor:	ps7_cortexa9_0
Architecture:	32-bit

< Back | Next > | Finish | Cancel

The final choice is the basic type of application to create. We wish to write most of our own code so an Empty Application (C or C++) or the Hello World options are easiest. Usually the Hello World is the best choice because it contains a very small bit of executable code pre-generated as a very simple first test of the design. So choose Hello World in this case.

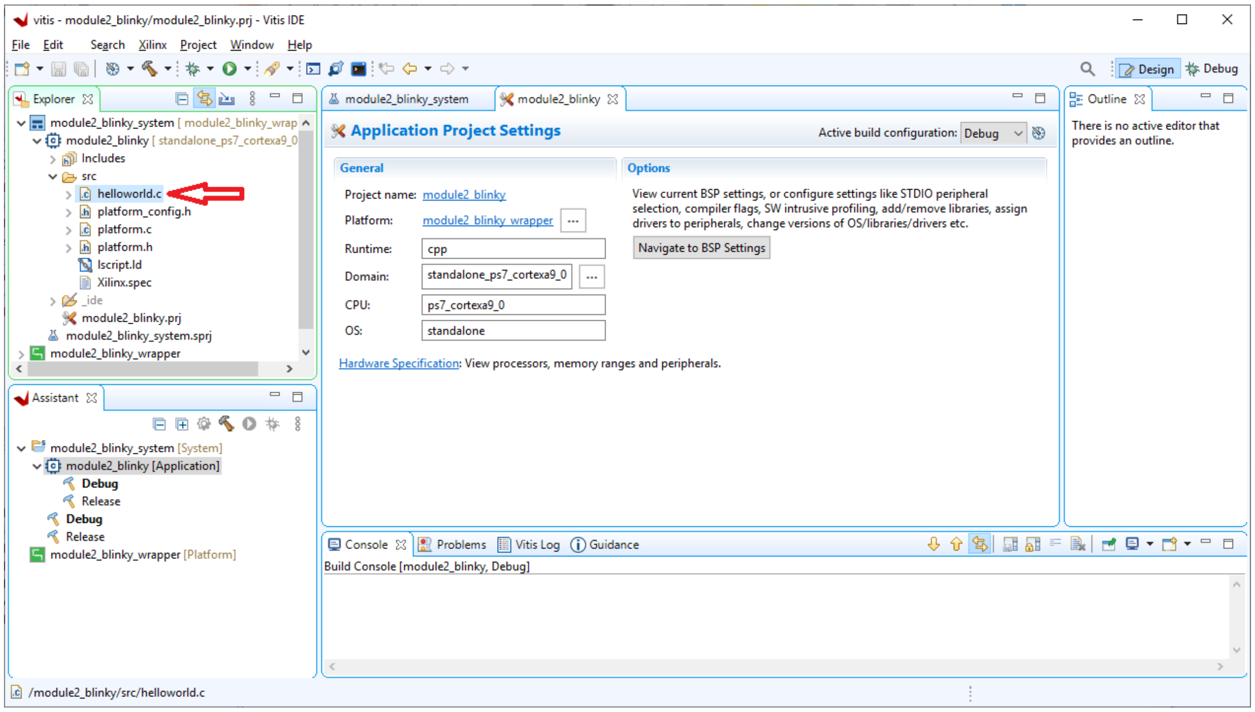


After a few moments, Vitis will return with its main editor environment.

Writing Code for ZYNC

Within the main editor environment you can do more than just write code to execute on the ARM Cortex or Microblaze processor. You have control of the entire compile process including adding or removing hardware libraries, modifying hardware libraries and running debug tests either in simulation mode or directly on the hardware via the FPGA's JTAG programming port.

Expand the SRC directory in the file explorer to find the pre-generated C and header files.


vitis - module2_blinky/module2_blinky.prj - Vitis IDE
File Edit Search Xilinx Project Window Help
Design Debug
Explorer
module2_blinky_system [module2_blinky_wrap
module2_blinky [standalone_ps7_cortexa9_0
Includes
src
helloworld.c
platform_config.h
platform.c
platform.h
lscript.ld
Xilinx.spec
_ide
module2_blinky.prj
module2_blinky_system.sprj
module2_blinky_wrapper
Assistant
module2_blinky_system [System]
module2_blinky [Application]
Debug
Release
Debug
Release
module2_blinky_wrapper [Platform]
module2_blinky_system
module2_blinky
Outline
Console Problems Vitis Log Guidance
/module2_blinky/src/helloworld.c

Application Project Settings

Active build configuration: Debug

General

Project name: module2_blinky
Platform: module2_blinky_wrapper
Runtime: cpp
Domain: standalone_ps7_cortexa9_0
CPU: ps7_cortexa9_0
OS: standalone

Options

View current BSP settings, or configure settings like STDIO peripheral selection, compiler flags, SW intrusive profiling, add/remove libraries, assign drivers to peripherals, change versions of OS/libraries/drivers etc.

Navigate to BSP Settings

Hardware Specification: View processors, memory ranges and peripherals.

There is no active editor that provides an outline.

Build Console [module2_blinky, Debug]

Because this example is relatively simple and wouldn't greatly benefit from the extra features of C++, it will remain a C language project.

Open "helloworld.c" (or change the filename by right-clicking and using the context menu) to examine the code. The other files can also be examined. The "ldscript.ld" linker file will first open in a special window which decodes the linker script or you can directly view the script as it is seen by the GCC compiler tools. Feel free to explore any of the files but refrain from changing them.

Note that the editor supports "code collapsing" so large blocks of comments such as those at the start of helloworld.c can be reduced to a single display line (or deleted).

The body of helloworld.c looks like this:

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

int main()
{
    init_platform();

    print("Hello World\n\r");
    print("Successfully ran Hello World application");
    cleanup_platform();
    return 0;
}
```

If you do not already have an understanding of C and its syntax, now would be a good time to find tutorials. Of particular note is that we will be writing code for a system with somewhat limited RAM. While technically the ZedBoard has plenty of off-FPGA DRAM available, it is not expressly used in this project. Furthermore, code run from that external RAM suffers a large reduction in execution speed. So we are going to limit most of the code to RAM found inside the FPGA. A good example of how this is handled is to look at the "print()" function. C has no build-in functions for printing (in this case sending data to the UART). It does have a function called "printf()" for print file. The function print() is a Xilinx specific greatly simplified wrapper function that has very low program and memory overhead associated. printf() is notorious in most implementations of the C language for carrying an excessive burden of memory overhead because it tries to do so many things. print() strips away all but the ability to send pre-formatted characters. A small step up is "xil_printf()" which allows some string generation before sending the characters. Both are defined in "xil_printf.h" and their source can be examined.

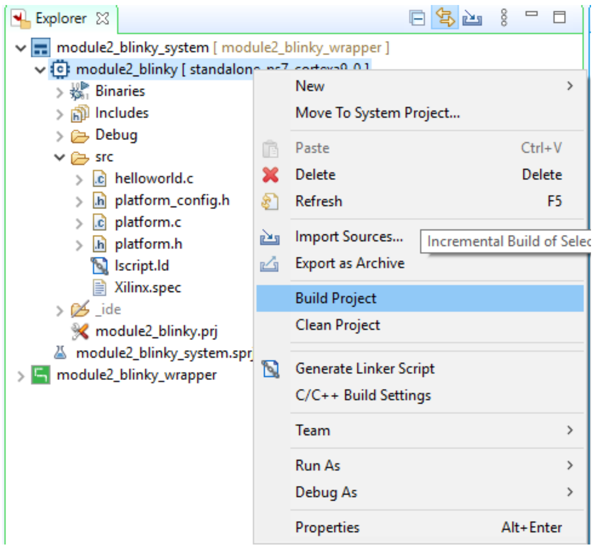
Change the code as follows:

```c
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "sleep.h"

int main()
{
    int i=0;
    init_platform();

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

    while(1)
    {
      xil_printf("The count is %d     \r", i++);
      usleep(500000);
    }

    cleanup_platform();
    return 0;
}
```

If you are used to writing C (or C++) code to run on operating systems like Linux, having a main() that does not return may be unusual. But in a standalone (bare-metal) application, there is nothing for the main() to return to. In a complete application, one would create hardware in PL that should the PS main() try to end, the PL would reboot the system and re-enter the main(). This is known as a "watchdog". But for our simple example we can skip that additional overhead.

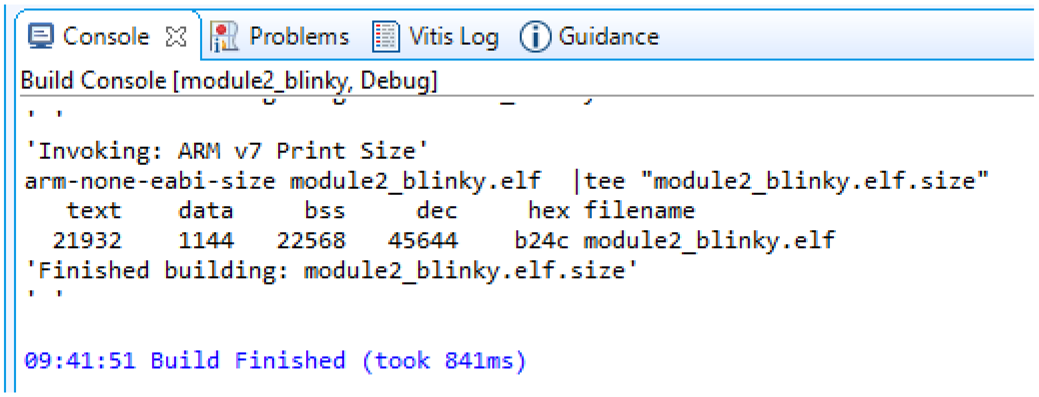
This code will just continue to count, displaying a new value approximately every 500,000 microseconds (1/2 second). The reason for the change in this test code is to allow us time to configure our terminal software (putty, Tera Term, etc.) to observe the program output.

To build the code you need to have the module highlighted in the Explorer and can either use the Project menu or a right-click context menu and select Build Project.



The initial build can take several seconds because the compiler is also chewing on all the parts of the platform project brought in. Unless the platform (FPGA design) is changed, this should only be required once because the GCC compiler supports incremental builds.

After a short time and a flurry of messages in the Console tab near the bottom of the editor, the Console tab should display something like this:



Build Console [module2_blinky, Debug]

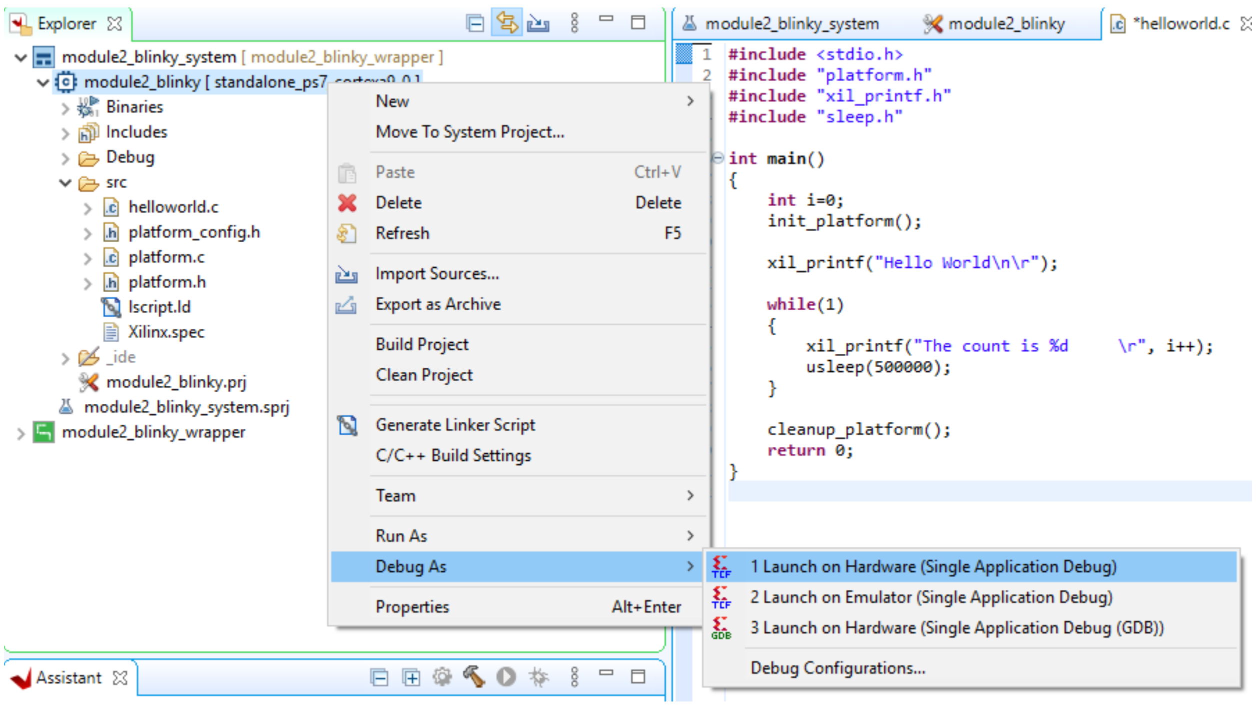
```
' '
'Invoking: ARM v7 Print Size'
arm-none-eabi-size module2_blinky.elf  |tee "module2_blinky.elf.size"
   text    data     bss     dec     hex filename
  21932    1144   22568   45644    b24c module2_blinky.elf
'Finished building: module2_blinky.elf.size'
' '

09:41:51 Build Finished (took 841ms)
```

If you click on the “Problems” tab you can see there may be one or two informational warnings. Any errors can also be located here and use the editor to jump to the location and make corrections.

The next step is to program the ZedBoard with the bitfile and our new ELF (Extensible Linking Format) file to test our PL LED blinker and our new executable code in the PS.

We will need a second USB cable attached to the port labeled UART on the left edge of the ZedBoard as well as software such as Tera Term. The Vitis editor does contain a serial terminal but it is a poor substitute for a full featured piece of terminal software.

The best way to first enter the hardware debugging mode is to again have the module highlighted in the Explorer and use the right-click context menu. We are going to “Launch on Hardware (Single Application Debug). The editor may ask you to save any altered files. This is good practice.

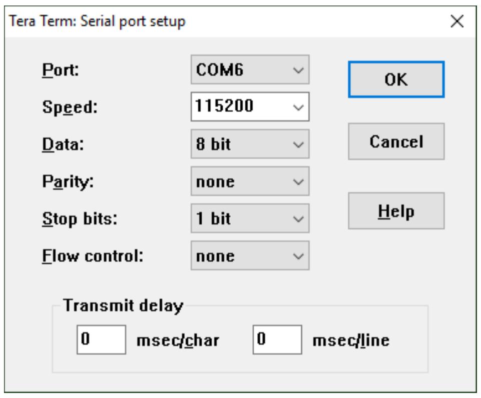


```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "sleep.h"

int main()
{
    int i=0;
    init_platform();

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

    while(1)
    {
        xil_printf("The count is %d     \r", i++);
        usleep(500000);
    }

    cleanup_platform();
    return 0;
}
```

If the connection to the ZedBoard is properly configured, after a short flurry of activity there should be proof the files were loaded into the FPGA by examining the XSCT Console in the lower right corner. Also the general layout of Vitis has changed to its Debug view. The startup-code will have executed to initialize the ARM Cortex processor and the debugger will be paused at the first executable line in main(), “init_platform()”.

At this time, start Tera Term or other terminal software and point it to the emulated serial port of the ZedBoard. The default communication speed will be 115,200 if using the Zynq or 9600 if using a MicroBlaze processor. Both can be changed if we return to Vivado and re-generate the bitstream. Leave them as defaults for now.

Assuming Tera Term, the settings would look like this (the port may vary)

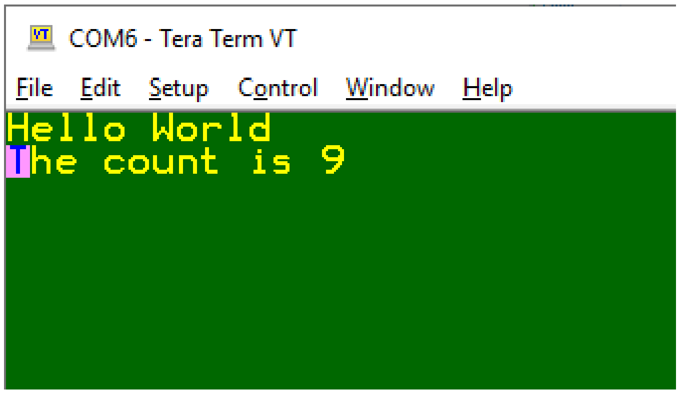


Port:	COM6
Speed:	115200
Data:	8 bit
Parity:	none
Stop bits:	1 bit
Flow control:	none

Transmit delay
0 msec/char 0 msec/line

And a reasonable terminal setup has local echo enabled so that one can see what is typed at the keyboard and the transmit can append either a carriage return (CR) or both carriage return and line feed. (CR+LF). Receive should have only an automatic CR to maintain compatibility with terminals like putty.

Use the "Resume" button on the button bar or under the "Run" menu to resume code execution. If the terminal is properly configured it should display something like this:

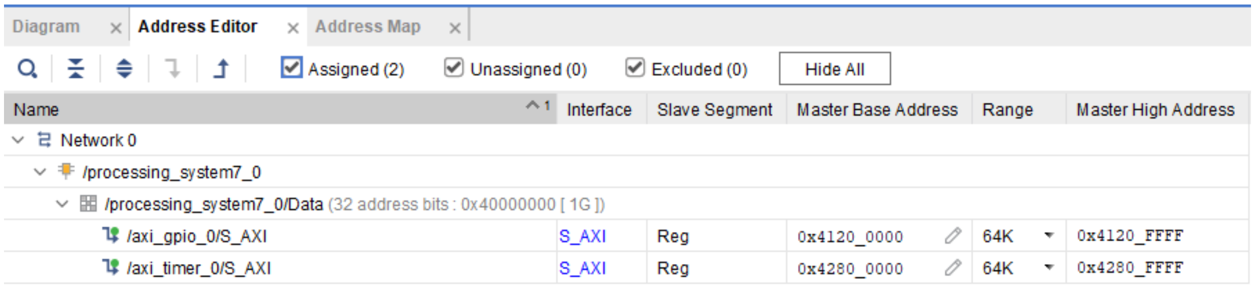


```
Hello World
The count is 9
```

Also, look at the ZedBoard and LD1 should be flashing at a 2Hz rate. You may have also noticed that prior to programming the ZedBoard it was not flashing. It started flashing as soon as the bitstream had completed loading followed by the load of the ELF file and finally the very short time the setup code was executed prior to pausing at init_platform(). This is because the clock and reset signals are taken from the ZYNC itself and until it has been programmed and starts, the clock and reset are idle.

Controlling PL from the PS

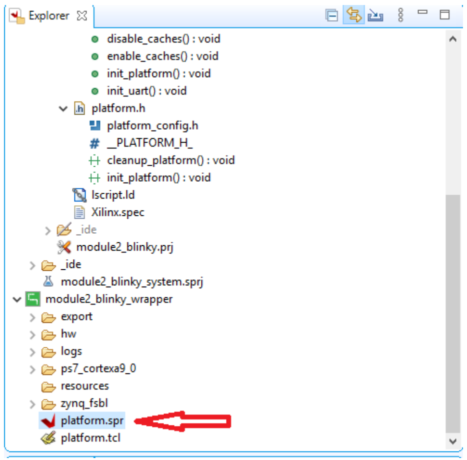
The PS sees everything in the PL as if it was memory. So every piece of hardware added to the PL lives in a "memory map". This map is automatically generated in Vivado but can be manipulated to re-arrange locations. Unless you have a good understanding of why memory-mapped I/O works, do not change the settings. The tabs "Address Editor" and "Address Map" in Vivado show the details.



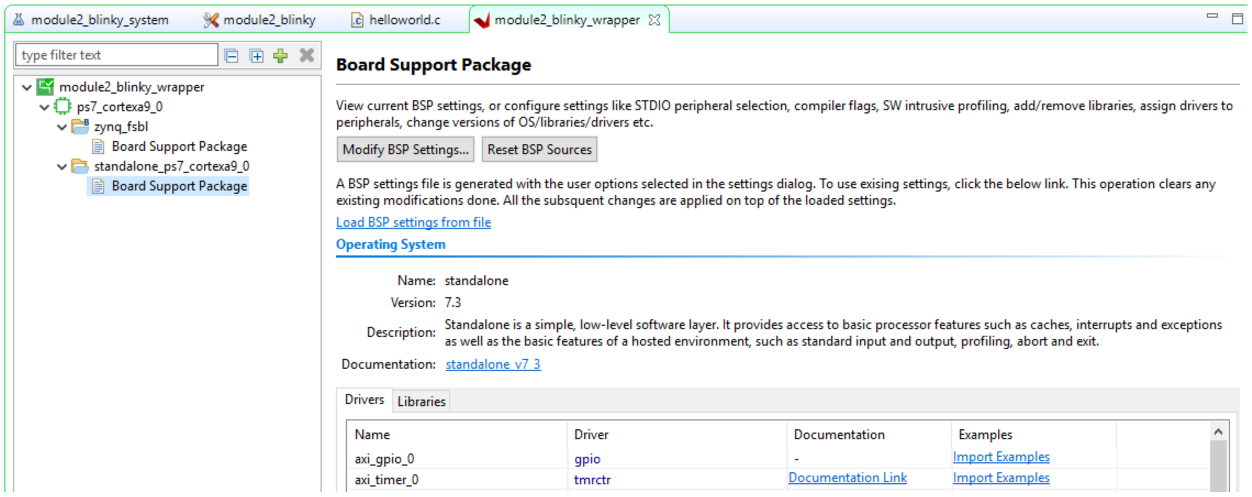
Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/axi_gpio_0/S_AXI	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
/axi_timer_0/S_AXI	S_AXI	Reg	0x4280_0000	64K	0x4280_FFFF

Here we see the GPIO (General Purpose I/O) and Timer we added to the block diagram. Notice that a range of 64K (64 x 1024) address have been assigned. Typically, devices like GPIO and timers only need a few memory locations to contain all of their programming and operating registers but by default, Vivado uses 64K memory boundaries. The control registers are all 32-bits wide (4 bytes) and so appear at 4-byte boundaries in memory. While useful to know, in practice this is not typically something that one deals with in the programming. Vivado and Vitis provide pre-generated.

Returning to the Design view in Vitis (there are two buttons in the upper right corner, Design and Debug), we need to look at the BSP (Board Support Package) to learn how to access the registers of the GPIO and Timer. Locate this file by double-clicking on the "platform.spr" file in the module wrapper.



This will open a tab in the editor to look at the wrapper. There, pick the "Board Support Package" to view the current settings.



Board Support Package

View current BSP settings, or configure settings like STDIO peripheral selection, compiler flags, SW intrusive profiling, add/remove libraries, assign drivers to peripherals, change versions of OS/libraries/drivers etc.

A BSP settings file is generated with the user options selected in the settings dialog. To use exising settings, click the below link. This operation clears any existing modifications done. All the subsquent changes are applied on top of the loaded settings.

Operating System

Name: standalone
Version: 7.3
Description: Standalone is a simple, low-level software layer. It provides access to basic processor features such as caches, interrupts and exceptions as well as the basic features of a hosted environment, such as standard input and output, profiling, abort and exit.
Documentation: standalone_v7_3

Name	Driver	Documentation	Examples
axi_gpio_0	gpio	-	Import Examples
axi_timer_0	tmrctr	Documentation Link	Import Examples

We are interested in the GPIO and Timer which happen to be listed at the top in this example. Note there are examples which can be imported into the Vitis project and examined. These source files will be populated automatically and can be browsed with the Explorer tab.

Our next step is to control LD0 from the ZYNC. For this we need to configure the GPIO block.

Our new code will be:

```
#include "stdio.h"
#include "platform.h"
#include "xparameters.h"
#include "xil_printf.h"
#include "sleep.h"
#include "xgpio.h"

#define GPIO XPAR_GPIO_0_DEVICE_ID    // less typing
#define LED_CHANNEL 1                 // LED resides in channel 1 of GPIO
#define LED_BIT 0x00000001            // LED associated with LSB of GPIO
XGpio Gpio;                           // Create instance of GPIO structure

int InitGPIO(void);

int main() {
        int i = 0;
        init_platform();

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

        if (InitGPIO() != XST_SUCCESS)
           xil_printf("\r\nThere was a problem with gpio.");

        while (1) {
           xil_printf("The count is %d     \r", i++);
           XGpio_DiscreteWrite(&Gpio, LED_CHANNEL, LED_BIT); // on
           usleep(200000);
           XGpio_DiscreteClear(&Gpio, LED_CHANNEL, LED_BIT); // off
           usleep(200000);
        }

        cleanup_platform();
        return 0;
}

int InitGPIO(void) {
        int Status;

        Status = XGpio_Initialize(&Gpio, GPIO); // initialize control of GPIO
        if (Status != XST_SUCCESS) {
           xil_printf("\r\nGPIO init failed\r\n");
           return (XST_FAILURE); // early exit
        }
        XGpio_SetDataDirection(&Gpio, LED_CHANNEL, ~LED_BIT); // set just bit 0 as output
        return (Status);
}
```

This is just one way to organize the code so that LD0 blinks, at approximately 2.5Hz. The GPIO port must be initialized then bits within it must be configured as INPUT or OUTPU (the LED is an OUTPUT) by writing a "0" into the appropriate register of the GPIO. This is handled by the XGpio helper functions. One can do this with direct memory writes but the helper functions are usually easier to understand as they are self-documenting. By right-clicking on any of these functions, the editor can jump to their definition so you can see how they work. Often times there are multiple layers to the functions. In this example case, we don't care about a little bit of processing overhead associated with these extra layers.

Once compiled and run on the hardware LD1 should be blinking at 2Hz and LD0 will blink at 2.5Hz.

Note to the addition of of "xparameters.h" and "xgpio.h" header files. These can be explored to see how the various memory mapped peripheral addresses are used and what features of the hardware are exposed to the processor.

Adding Timer and Interrupt Support

In microcontrollers and microprocessors, an "interrupt" is a signal that occurs asynchronously to main code execution. For example, typing at keyboard is completely asynchronous to anything the processor may be doing. And so if the processor needs to know about the typing or deal with its results, it should be interrupted. Interrupts can have priorities (very low to very high), be nested within other interrupts and be ignored. For the purpose of this tutorial, we have one interrupt. It comes from a timer we will configure to "tick" at a 1Hz rate. This will be used to blink our LD0 LED.

The first step is to understand how the timer works. The timer will be configured to count up from zero until it reaches the magic number. When the magic number is reached, an interrupt is generated and the timer automatically resets to zero and resumes counting.

write code to allow us to configure the timer. It uses a 100MHz master clock from the ZYNC and will produce an interrupt to the ZYNC every 100,000,000 clocks. The timer will automatically reset after each interrupt and continue to generate interrupts independently of any other hardware on the FPGA.

This calculation can be automated so that the C-preprocessor will always calculate the correct magic numbers even if we make small changes to clock speeds in the Vivado project.

```
#define TIMER_FREQUENCY XPAR_AXI_TIMER_0_CLOCK_FREQ_HZ
#define PBLINK 1.0
#define RESET_VALUE     (u32)(0xffffffff - PBLINK * TIMER_FREQUENCY)
```

The constant XPAR_AXI_TIMER_0_CLOCK_FREQ_HZ is defined in the xparameters.h include file which is automatically generated by Vitis when the boad support module is created. It contains a long list of #defines to make accessing the various memory address, offsets and register control words easier than remembering individual numbers.

Our RESET_VALUE is computed from 2^{32}-1 (the maximum count possible in the 32-bit timer) minus the desired blink period times the clock frequency driving the timer.

The complete code for programming the timer and interrupt system can be found in Appendix A. It is a combination of the previous example C code and that used in the Timer interrupt example that can be imported into the project as described above.

Writing and Reading Bits Using Timer

The final part of the project is to further modify the code so that a pattern of 1's and 0's appear on JA4 to be read on JA1. This pattern should be rapid, 100kbit/sec is a good starting place. The LD0 and LD1 leds should continue to operate as before.

The timer can be reconfigured for a 100kHz interrupt rate. Every time the counter interrupts a new bit will be written to JA4 and read back on JA1. And every 100,000 interrupts the state of the LD0 will be changed. And error between the write and read bit will be noted for possible use elsewhere in the program.

The new counter RESET_VALUE would be:

```
#define TIMER_FREQUENCY XPAR_AXI_TIMER_0_CLOCK_FREQ_HZ // clock into timer
#define FBLINK 100.0e3
#define PBLINK 1.0/FBLINK     // blink period in seconds
#define RESET_VALUE     (u32)(0xffffffff - PBLINK * TIMER_FREQUENCY) // make the magic
```

And the initialization of the GPIO should now configure bit 1 as an output and bit 2 as an input per the connections made in the Vivado block diagram.

Because the code will now potentially be manipulating the GPIO register from multiple points, it should either read the contents of the register and then manipulate that value or keep a shadow copy of the contents. A shadow copy is more efficient.

Because the rate of interrupts has been increased, the main code can no longer guarantee that it would see the tic counter reach an exact value. To blink the LEDs the tic counter would be 100,000 for a 1Hz rate but it may be that other code interrupted the main loop and it misses the count of 100,000. So instead the main loop should react when the tic counter is greater-than-or-equal to 100,000. The blinking of LEDs remains a low priority, the potential jitter introduced with this method doesn't matter.

The Timer ISR (Interrupt Service Routine) will not be responsible for toggling the output bit and reading the input bit as well as making the necessary comparison. If there were many bits to toggle or check, a different method should be employed. When the main loop is managing the LED it will also check to see if an error occurred and report the error. It won't know exactly when the error occurred, just that one did occur.

Throughout the code, bit manipulation using the logical operators & (and), | (or), & (xor) and ~ (not) appear. As well as the logical left shift operation (>>). These are very useful in writing C code to check and manipulate bits in larger data types.

The complete code appears in Appendix B.

After completing and testing the code consider if this code could be modified to create a 1Mbps test data stream?

Adding Operator Control via USB (Serial)

A final change to the code could allow the operator to enable or disable the blinking of LD0. In larger operating systems, the STDIO library includes several stream functions for receiving one or more characters from a data (serial) stream. These functions while present in the bare-metal system implemented for this exercise, are not necessarily fully realized.

A reasonable solution to receive a single character from the operator, perhaps 'A' to start blinking and 'B' to stop blinking, would be to use the low-level C libraries associated with the UART. Because we are using the UART included inside the ZYNQ, the appropriate functions are available with

```
#include "xuartps_hw.h"
```

And in particular, `XUartPs_IsReceiveData`

A relatively simple menu system or method of accepting continuous commands from a PC host could be created. Likewise, there are low level functions for pushing data back to the PC or one could use the STDIO functions.

The 3rd revision of the source code appears in Appendix C. Only the #include, #define and int main() code was changed. The support functions, ISR, etc. do not change and are not repeated in Appendix C.

Conclusion

An FPGA with both a PS and PL can create and control complex systems. This example is relatively simple and slow running. Faster tests should move the bit testing into the PL with the PS acting as a supervisor, recording and reporting results.

Appendix A

```
#include "stdio.h"
#include "platform.h"
#include "xparameters.h"
#include "xil_printf.h"
#include "sleep.h"
#include "xgpio.h"
#include "xtmrctr.h"
#include "xil_exception.h"
#include "xscugic.h"

#define VT100_ESC 27 // Tera Term emulates VT100, define the escape code
#define ClearScreen xil_printf("%c[2J",VT100_ESC) // clear screen using VT100 format string
#define CursorHome xil_printf("%c[f",VT100_ESC) // cursor home

#define GPIO XPAR_GPIO_0_DEVICE_ID // less typing
#define LED_CHANNEL 1  // LED resides in channel 1 of GPIO
#define LED_BIT 0x00000001 // LED associated with LSB of GPIO
XGpio Gpio; // Create instance of GPIO structure

#define TMRCTR_DEVICE_ID      XPAR_TMRCTR_0_DEVICE_ID // aliases to make reference easier
#define TMRCTR_INTERRUPT_ID   XPAR_FABRIC_TMRCTR_0_VEC_ID
#define INTC_DEVICE_ID        XPAR_SCUGIC_SINGLE_DEVICE_ID
#define TIMER_CNTR_0 0 // the timer contains two physical timers, 0 and 1

#define TIMER_FREQUENCY XPAR_AXI_TIMER_0_CLOCK_FREQ_HZ // clock into timer
#define PBLINK 1.0     // blink period in seconds
#define RESET_VALUE     (u32)(0xffffffff - PBLINK * TIMER_FREQUENCY) // make the magic

// using the interrupt controller built into
// the ZYNC and so giving it an alias that
// resembles the name used for Microblaze code
// if this code is ported to a new processor
#define INTC XScuGic
#define INTC_HANDLER XScuGic_InterruptHandler // less typing later
INTC InterruptController; // The instance of the Interrupt Controller
XTmrCtr TimerCounterInst; // The instance of the Timer Counter

volatile int tic; // a flag indicating the timer has rolled over

int TmrCtrIntrInit(INTC *IntcInstancePtr, XTmrCtr *InstancePtr, u16 DeviceId,
               u16 IntrId, u8 TmrCtrNumber); // initialize the timer

static int TmrCtrSetupIntrSystem(INTC *IntcInstancePtr, XTmrCtr *InstancePtr,
               u16 DeviceId, u16 IntrId, u8 TmrCtrNumber); // initialize the interrupt system

static void TimerCounterHandler(void *CallBackRef, u8 TmrCtrNumber); // timer ISR

int InitGPIO(void); // initialize the GPIO

int main() {
   int i = 0;
   init_platform();

   ClearScreen;
   CursorHome;

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

   if (InitGPIO() != XST_SUCCESS)
      xil_printf("\r\nThere was a problem with gpio."); // drat

   if (TmrCtrIntrInit(&InterruptController, &TimerCounterInst,
      TMRCTR_DEVICE_ID, TMRCTR_INTERRUPT_ID, TIMER_CNTR_0) != XST_SUCCESS) {
      xil_printf("\r\nError initializing timer & interrupt."); // drat
   } else {
      XTmrCtr_Start(&TimerCounterInst, TIMER_CNTR_0); // start the timer!
   }

   // because blinking LEDs is typically not time sensitive the code will manage
   // the blink in the main loop instead of directly in the timer ISR
   while (1) {
      if (tic) {
         tic = 0; // we have recognized the interrupt & are processing
         if (i % 2)
            XGpio_DiscreteWrite(&Gpio, LED_CHANNEL, LED_BIT); // on for odd counts
         else
            XGpio_DiscreteClear(&Gpio, LED_CHANNEL, LED_BIT); // off for even counts
         xil_printf("The count is %d     \r", i++);
      }
   }

   cleanup_platform(); // never reached
   return 0;
}

void TimerCounterHandler(void *CallBackRef, u8 TmrCtrNumber) {
   tic = 1;
}

int InitGPIO(void) {
   int Status;

   Status = XGpio_Initialize(&Gpio, GPIO); // initialize control of GPIO
   if (Status != XST_SUCCESS) {
      xil_printf("\r\nGPIO init failed\r\n");
      return (XST_FAILURE); // early exit
   }
   XGpio_SetDataDirection(&Gpio, LED_CHANNEL, ~LED_BIT); // set just bit 0 as output
   return (Status);
}

int TmrCtrIntrInit(INTC *IntcInstancePtr, XTmrCtr *TmrCtrInstancePtr,
   u16 DeviceId, u16 IntrId, u8 TmrCtrNumber) {
   int Status;

   // Initialize the timer counter so that it's ready to use,
   // specify the device ID that is generated in xparameters.h
   Status = XTmrCtr_Initialize(TmrCtrInstancePtr, DeviceId);
   if (Status != XST_SUCCESS) {
      return XST_FAILURE;
   }

   // Perform a self-test to ensure that the hardware was built
   // correctly, use the 1st timer in the device (0)
   Status = XTmrCtr_SelfTest(TmrCtrInstancePtr, TmrCtrNumber);
   if (Status != XST_SUCCESS) {
      return XST_FAILURE;
   }

   // Connect the timer counter to the interrupt subsystem such that
   // interrupts can occur.  This function is application specific.
   Status = TmrCtrSetupIntrSystem(IntcInstancePtr, TmrCtrInstancePtr, DeviceId,
                IntrId, TmrCtrNumber);
   if (Status != XST_SUCCESS) {
      return XST_FAILURE;
   }

   // Setup the handler for the timer counter that will be called from the
   // interrupt context when the timer expires, specify a pointer to the
   // timer counter driver instance as the callback reference so the
   // handler is able to access the instance data
   XTmrCtr_SetHandler(TmrCtrInstancePtr, TimerCounterHandler,
      TmrCtrInstancePtr);

   // Enable the interrupt of the timer counter so interrupts will occur
   // and use auto reload mode such that the timer counter will reload
   // itself automatically and continue repeatedly, without this option
   // it would expire once only
   XTmrCtr_SetOptions(TmrCtrInstancePtr, TmrCtrNumber,
      XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION);
   // Set a reset value for the timer counter such that it will expire
   // eariler than letting it roll over from 0, the reset value is loaded
   // into the timer counter when it is started
   XTmrCtr_SetResetValue(TmrCtrInstancePtr, TmrCtrNumber, RESET_VALUE);

   return XST_SUCCESS;
}

static int TmrCtrSetupIntrSystem(INTC *IntcInstancePtr,
   XTmrCtr *TmrCtrInstancePtr, u16 DeviceId, u16 IntrId, u8 TmrCtrNumber) {
   int Status;

   XScuGic_Config *IntcConfig;

   // Initialize the interrupt controller driver so that it is ready to use.
   IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
   if (NULL == IntcConfig) {
      return XST_FAILURE;
   }

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

   XScuGic_SetPriorityTriggerType(IntcInstancePtr, IntrId, 0xA0, 0x3);

   // Connect the interrupt handler that will be called when an
   // interrupt occurs for the device.
   Status = XScuGic_Connect(IntcInstancePtr, IntrId,
            (Xil_ExceptionHandler) XTmrCtr_InterruptHandler, TmrCtrInstancePtr);
   if (Status != XST_SUCCESS) {
      return Status;
   }

   // Enable the interrupt for the Timer device.
   XScuGic_Enable(IntcInstancePtr, IntrId);

   // Initialize the exception table.
   Xil_ExceptionInit();

    //Register the interrupt controller handler with the exception table.
   Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT, (Xil_ExceptionHandler)
      INTC_HANDLER, IntcInstancePtr);

   Xil_ExceptionEnable();      // Enable non-critical exceptions.

   return XST_SUCCESS;
}
```

Appendix B

```
#include "stdio.h"
#include "platform.h"
#include "xparameters.h"
#include "xil_printf.h"
#include "sleep.h"
#include "xgpio.h"
#include "xtmrctr.h"
#include "xil_exception.h"
#include "xscugic.h"

#define VT100_ESC 27 // Tera Term emulates VT100, define the escape code
#define ClearScreen xil_printf("%c[2J",VT100_ESC) // clear screen using VT100 format string
#define CursorHome xil_printf("%c[f",VT100_ESC) // cursor home

#define GPIO XPAR_GPIO_0_DEVICE_ID    // less typing
#define GPIO0 XPAR_GPIO_0_BASEADDR    // less typing
#define LED_CHANNEL 1  // LED resides in channel 1 of GPIO
#define LED_BIT 0x00000001 // LED associated with LSB of GPIO
#define DDR (u32)0b100  // data direction magic word
XGpio Gpio;     // Create instance of GPIO structure
volatile u32 shadow_write; // GPIO can be written from multiple sources, use shadow register

#define TMRCTR_DEVICE_ID      XPAR_TMRCTR_0_DEVICE_ID // aliases to make reference easier
#define TMRCTR_INTERRUPT_ID   XPAR_FABRIC_TMRCTR_0_VEC_ID
#define INTC_DEVICE_ID        XPAR_SCUGIC_SINGLE_DEVICE_ID
#define TIMER_CNTR_0 0 // the timer contains two physical timers, 0 and 1

#define TIMER_FREQUENCY XPAR_AXI_TIMER_0_CLOCK_FREQ_HZ // clock into timer
#define FBLINK 100.0e3
#define PBLINK 1.0/FBLINK     // blink period in seconds
#define RESET_VALUE     (u32)(0xffffffff - PBLINK * TIMER_FREQUENCY) // make the magic

// using the interrupt controller built into
// the ZYNC and so giving it an alias that
// resembles the name used for Microblaze code
// if this code is ported to a new processor
#define INTC XScuGic
#define INTC_HANDLER XScuGic_InterruptHandler // less typing later
INTC InterruptController; // The instance of the Interrupt Controller
XTmrCtr TimerCounterInst; // The instance of the Timer Counter

volatile u32 tic; // a flag indicating the timer has rolled over
volatile int error;

int TmrCtrIntrInit(INTC *IntcInstancePtr, XTmrCtr *InstancePtr, u16 DeviceId,
   u16 IntrId, u8 TmrCtrNumber); // initialize the timer

static int TmrCtrSetupIntrSystem(INTC *IntcInstancePtr, XTmrCtr *InstancePtr,
   u16 DeviceId, u16 IntrId, u8 TmrCtrNumber); // initialize the interrupt system

static void TimerCounterHandler(void *CallBackRef, u8 TmrCtrNumber); // timer ISR

int InitGPIO(void); // initialize the GPIO

int main() {
   int i = 0;
   init_platform();

   ClearScreen;
   CursorHome;

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

   if (InitGPIO() != XST_SUCCESS)
      xil_printf("\r\nThere was a problem with gpio."); // drat

   if (TmrCtrIntrInit(&InterruptController, &TimerCounterInst,
         TMRCTR_DEVICE_ID, TMRCTR_INTERRUPT_ID, TIMER_CNTR_0) != XST_SUCCESS) {
      xil_printf("\r\nError initializing timer & interrupt."); // drat
   } else {
      XTmrCtr_Start(&TimerCounterInst, TIMER_CNTR_0); // start the timer!
   }

   // because blinking LEDs is typically not time sensitive the code will manage
   // the blink in the main loop instead of directly in the timer ISR
   // the tic counter is checked for exact count or greater
   error = 0;
   while (1) {
      if (tic >= (u32) FBLINK) {
         tic = 0; // we have recognized the interrupt & are processing
         if (i % 2)
            shadow_write = shadow_write | LED_BIT; // set
         else
            shadow_write = shadow_write & (~LED_BIT); // clear
         Xil_Out32(GPIO0, shadow_write);
         xil_printf("The count is %d\t", i++);
         if (error) {
            xil_printf("err \r");
            error = 0;
         } else
            xil_printf("    \r");
      }
   }
   cleanup_platform(); // never reached
   return 0;
}

void TimerCounterHandler(void *CallBackRef, u8 TmrCtrNumber) {
   static u32 din;
   static u32 dout = 0;

   shadow_write = (shadow_write & ~(u32) 0b10) | dout; // jam test bit into the shadow register
   Xil_Out32(GPIO0, shadow_write); // write the shadow register w/ least overhead
   ++tic; // burns a little time & counts interrupts This prevents read/write/modify errors
   din = (Xil_In32(GPIO0) & (u32) 0b100) >> 1; // read back, mask and shift
   if (din != dout) // bits are the same?
      error = 1; // flag error
   dout = dout ^ (u32) 0b10; // flip our test bit
}

int InitGPIO(void) {
   int Status;

   Status = XGpio_Initialize(&Gpio, GPIO); // initialize control of GPIO
   if (Status != XST_SUCCESS) {
      xil_printf("\r\nGPIO init failed\r\n");
      return (XST_FAILURE); // early exit
   }
   XGpio_SetDataDirection(&Gpio, LED_CHANNEL, DDR); // set just bit 0 as output
   shadow_write = 0;
   Xil_Out32(GPIO0, shadow_write);
   return (Status);
}

int TmrCtrIntrInit(INTC *IntcInstancePtr, XTmrCtr *TmrCtrInstancePtr,
      u16 DeviceId, u16 IntrId, u8 TmrCtrNumber) {
   int Status;

   // Initialize the timer counter so that it's ready to use,
   // specify the device ID that is generated in xparameters.h
   Status = XTmrCtr_Initialize(TmrCtrInstancePtr, DeviceId);
   if (Status != XST_SUCCESS) {
      return XST_FAILURE;
   }

   // Perform a self-test to ensure that the hardware was built
   // correctly, use the 1st timer in the device (0)
   Status = XTmrCtr_SelfTest(TmrCtrInstancePtr, TmrCtrNumber);
   if (Status != XST_SUCCESS) {
       return XST_FAILURE;
   }

   // Connect the timer counter to the interrupt subsystem such that
   // interrupts can occur.  This function is application specific.
   Status = TmrCtrSetupIntrSystem(IntcInstancePtr, TmrCtrInstancePtr, DeviceId,
               IntrId, TmrCtrNumber);
   if (Status != XST_SUCCESS) {
      return XST_FAILURE;
   }

   // Setup the handler for the timer counter that will be called from the
   // interrupt context when the timer expires, specify a pointer to the
   // timer counter driver instance as the callback reference so the
   // handler is able to access the instance data
   XTmrCtr_SetHandler(TmrCtrInstancePtr, TimerCounterHandler,
                     TmrCtrInstancePtr);

   // Enable the interrupt of the timer counter so interrupts will occur
   // and use auto reload mode such that the timer counter will reload
   // itself automatically and continue repeatedly, without this option
   // it would expire once only
   XTmrCtr_SetOptions(TmrCtrInstancePtr, TmrCtrNumber,
      XTC_INT_MODE_OPTION | XTC_AUTO_RELOAD_OPTION);

   // Set a reset value for the timer counter such that it will expire
   // eariler than letting it roll over from 0, the reset value is loaded
   // into the timer counter when it is started
   XTmrCtr_SetResetValue(TmrCtrInstancePtr, TmrCtrNumber, RESET_VALUE);

   return XST_SUCCESS;
}

static int TmrCtrSetupIntrSystem(INTC *IntcInstancePtr,
               XTmrCtr *TmrCtrInstancePtr, u16 DeviceId, u16 IntrId, u8 TmrCtrNumber) {
   int Status;
   XScuGic_Config *IntcConfig;

   // Initialize the interrupt controller driver so that it is ready to use.
   IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
   if (NULL == IntcConfig) {
      return XST_FAILURE;
   }

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

   XScuGic_SetPriorityTriggerType(IntcInstancePtr, IntrId, 0xA0, 0x3);

   // Connect the interrupt handler that will be called when an
   // interrupt occurs for the device.
   Status = XScuGic_Connect(IntcInstancePtr, IntrId,
               (Xil_ExceptionHandler) XTmrCtr_InterruptHandler, TmrCtrInstancePtr);
   if (Status != XST_SUCCESS) {
      return Status;
   }

   // Enable the interrupt for the Timer device.
   XScuGic_Enable(IntcInstancePtr, IntrId);

   // Initialize the exception table.
   Xil_ExceptionInit();

   //Register the interrupt controller handler with the exception table.
   Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT, (Xil_ExceptionHandler)
        INTC_HANDLER, IntcInstancePtr);
   Xil_ExceptionEnable();      // Enable non-critical exceptions.
   return XST_SUCCESS;
}
```

Appendix C

```
#include "stdio.h"
#include "platform.h"
#include "xparameters.h"
#include "xil_printf.h"
#include "sleep.h"
#include "xgpio.h"
#include "xtmrctr.h"
#include "xil_exception.h"
#include "xscugic.h"
#include "xuartps_hw.h"

#define VT100_ESC 27 // Tera Term emulates VT100, define the escape code
#define ClearScreen xil_printf("%c[2J",VT100_ESC) // clear screen using VT100 format string
#define CursorHome xil_printf("%c[f",VT100_ESC) // cursor home

#define UART_BASEADDR XPAR_XUARTPS_0_BASEADDR

#define GPIO XPAR_GPIO_0_DEVICE_ID    // less typing
#define GPIO0 XPAR_GPIO_0_BASEADDR    // less typing
#define LED_CHANNEL 1  // LED resides in channel 1 of GPIO
#define LED_BIT 0x00000001    // LED associated with LSB of GPIO
#define DDR (u32)0b100        // data direction magic word
XGpio Gpio;                   // Create instance of GPIO structure
volatile u32 shadow_write; // GPIO can be written from multiple sources, use shadow register

#define TMRCTR_DEVICE_ID      XPAR_TMRCTR_0_DEVICE_ID // aliases to make reference easier
#define TMRCTR_INTERRUPT_ID   XPAR_FABRIC_TMRCTR_0_VEC_ID
#define INTC_DEVICE_ID        XPAR_SCUGIC_SINGLE_DEVICE_ID
#define TIMER_CNTR_0 0 // the timer contains two physical timers, 0 and 1

#define TIMER_FREQUENCY XPAR_AXI_TIMER_0_CLOCK_FREQ_HZ // clock into timer
#define FBLINK 100.0e3
#define PBLINK 1.0/FBLINK     // blink period in seconds
#define RESET_VALUE     (u32)(0xffffffff - PBLINK * TIMER_FREQUENCY) // make the magic

// using the interrupt controller built into
// the ZYNC and so giving it an alias that
// resembles the name used for Microblaze code
// if this code is ported to a new processor
#define INTC XScuGic
#define INTC_HANDLER XScuGic_InterruptHandler // less typing later
INTC InterruptController; // The instance of the Interrupt Controller
XTmrCtr TimerCounterInst; // The instance of the Timer Counter

volatile u32 tic; // a flag indicating the timer has rolled over
volatile int error;

int TmrCtrIntrInit(INTC *IntcInstancePtr, XTmrCtr *InstancePtr, u16 DeviceId,
               u16 IntrId, u8 TmrCtrNumber); // initialize the timer

static int TmrCtrSetupIntrSystem(INTC *IntcInstancePtr, XTmrCtr *InstancePtr,
               u16 DeviceId, u16 IntrId, u8 TmrCtrNumber); // initialize the interrupt system

static void TimerCounterHandler(void *CallBackRef, u8 TmrCtrNumber); // timer ISR

int InitGPIO(void); // initialize the GPIO

int main() {
   int i = 0;
   int doblink = 1; // assume we want to blink
   u8 c;
   init_platform();

   ClearScreen;
   CursorHome;

   xil_printf("Hello World\n\r");
   if (InitGPIO() != XST_SUCCESS)
      xil_printf("\r\nThere was a problem with gpio."); // drat
   if (TmrCtrIntrInit(&InterruptController, &TimerCounterInst,
         TMRCTR_DEVICE_ID, TMRCTR_INTERRUPT_ID, TIMER_CNTR_0) != XST_SUCCESS) {
      xil_printf("\r\nError initializing timer & interrupt."); // drat
   } else {
      XTmrCtr_Start(&TimerCounterInst, TIMER_CNTR_0); // start the timer!
   }

   // because blinking LEDs is typically not time sensitive the code will manage
   // the blink in the main loop instead of directly in the timer ISR
   // the tic counter is checked for exact count or greater
   error = 0;
   while (1) {
      if (XUartPs_IsReceiveData(UART_BASEADDR)) { // is there a char waiting?
         c = toupper((int)XUartPs_ReadReg(UART_BASEADDR, XUARTPS_FIFO_OFFSET));
      switch (c) {
         case 'A':
            doblink = 1;
            break;
         case 'B':
            doblink = 0;
            break;
         default:
            break; // ignore
      }
   }
   if (tic >= (u32) FBLINK) {
      tic = 0; // we have recognized the interrupt & are processing
      if (doblink) {
         if (i % 2)
            shadow_write = shadow_write | LED_BIT; // set
         else
            shadow_write = shadow_write & (~LED_BIT); // clear
         Xil_Out32(GPIO0, shadow_write);
      }
      xil_printf("The count is %d\t", i++);
      if (error) {
         xil_printf("err \r");
         error = 0;
      } else
         xil_printf("    \r");
      }
   }
   cleanup_platform(); // never reached
   return 0;
}
```