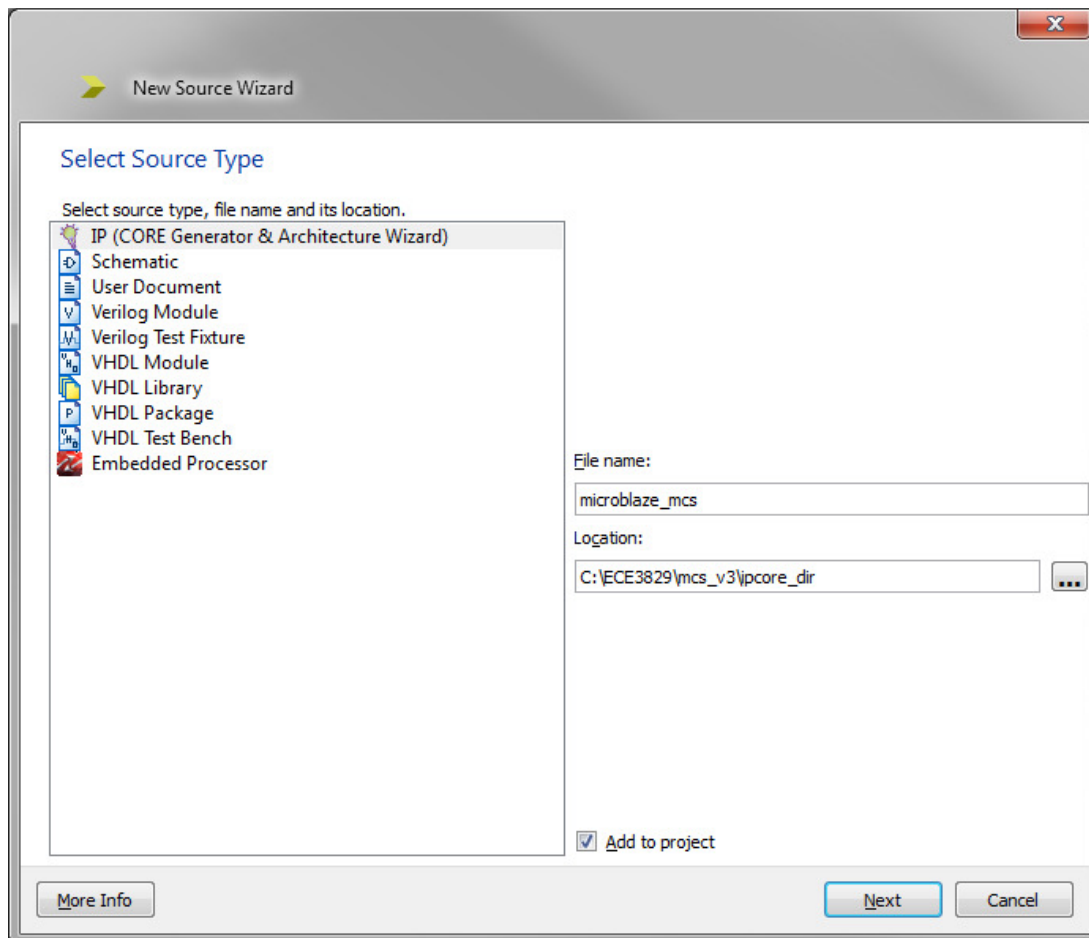


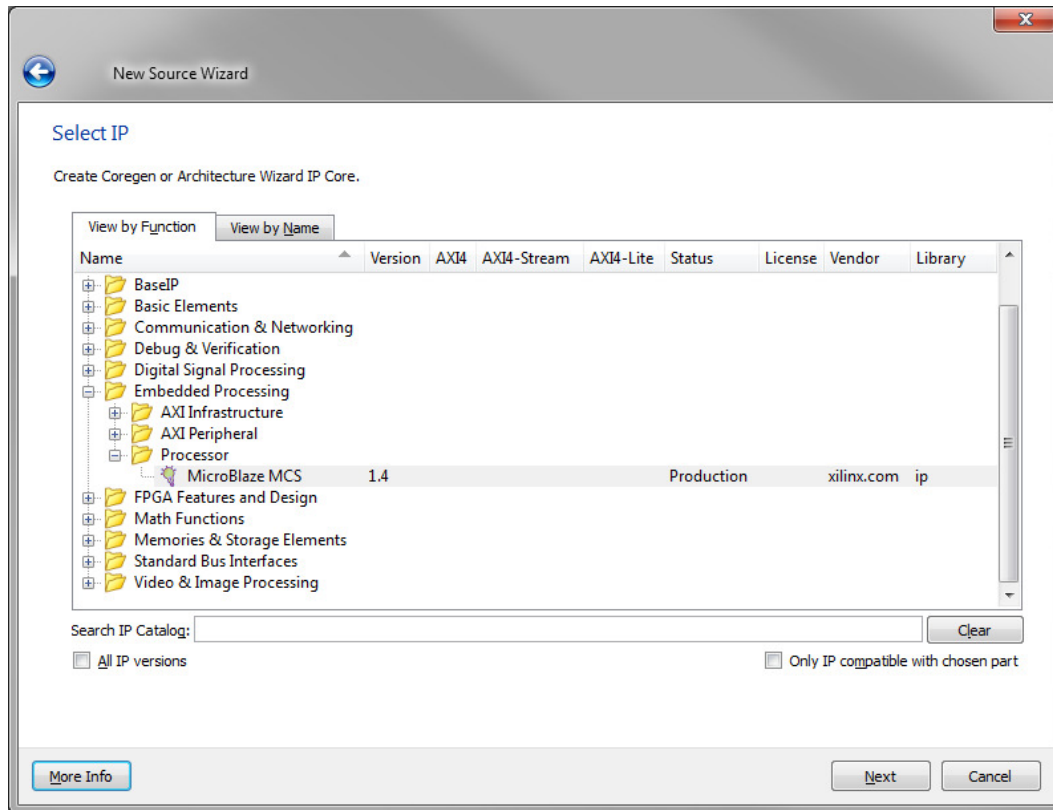
**Microblaze MCS Tutorial for Xilinx ISE 14.5****Rev 5 (October 2013) – updated to ISE 14.5**

This tutorial shows how to add a Microblaze Microcontroller System (MCS) embedded processor to a project including adding a simple C program. The design was targeted to a Spartan 6 FPGA (on a Nexys3 board) but the steps should be general enough to work on other platforms.

Create a new project and then select **Project => New Source**, and select **IP** (Core Generator & Architecture Wizard) (do NOT select the Embedded Processor source type – we want the simpler MCS version) and provide a file name (I used 'microblaze\_mcs' in this example)

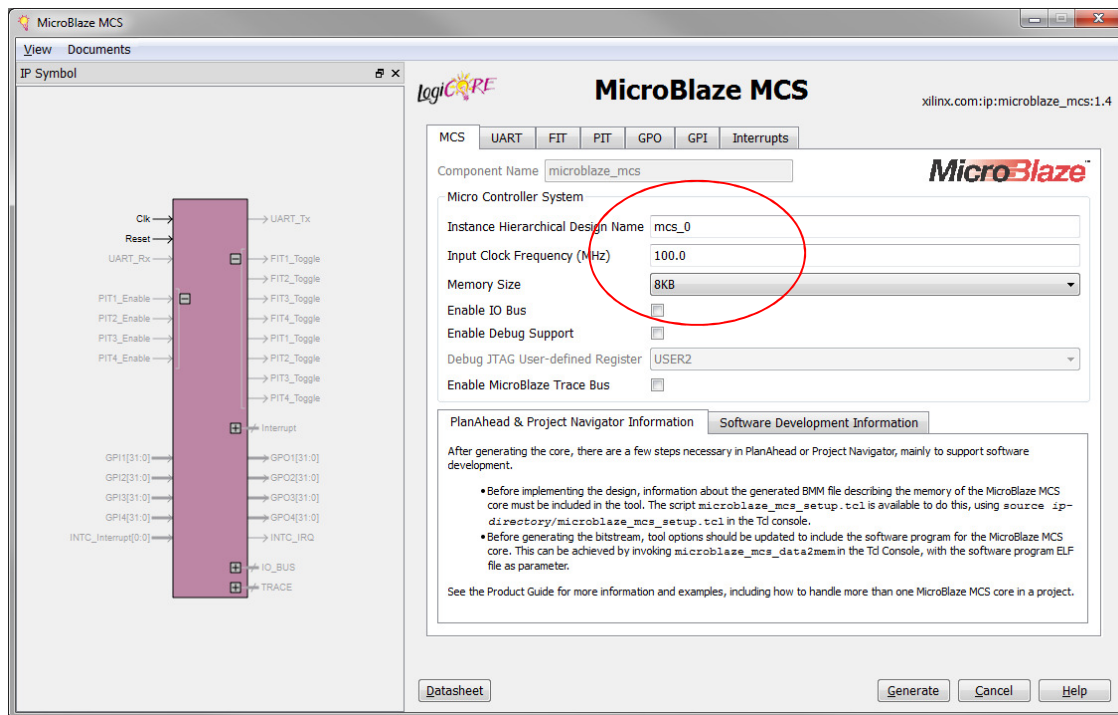


and then select Microblaze MCS under Embedded Processing:

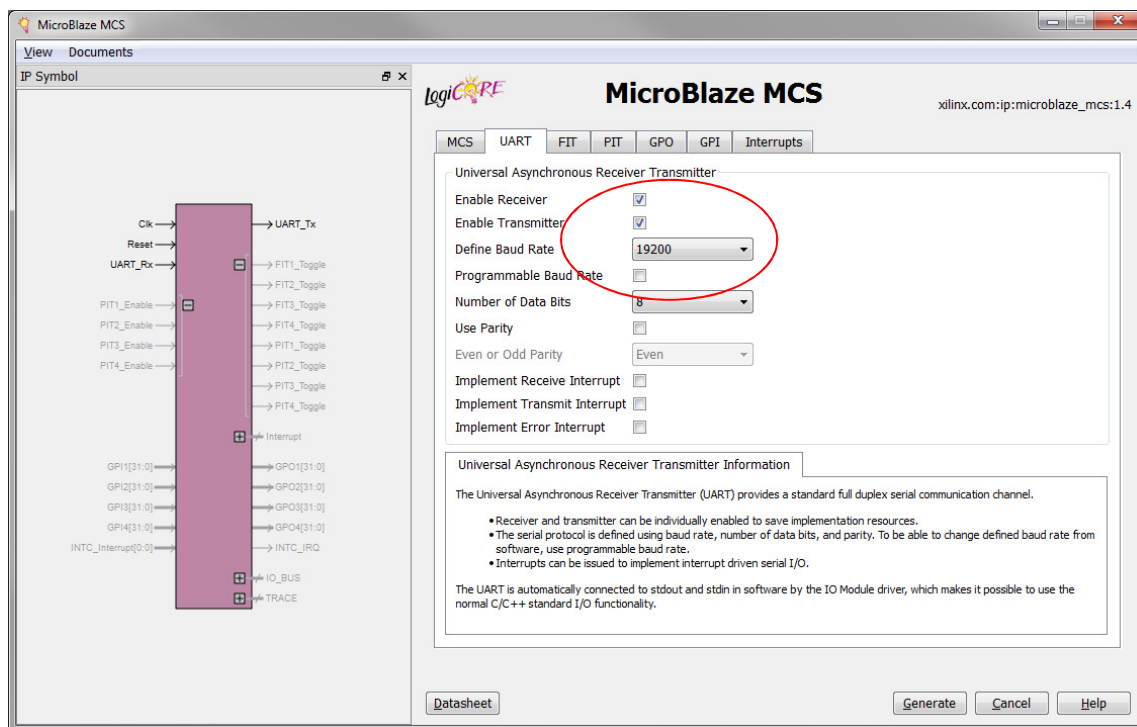


Click **Next** and then **Finish**.

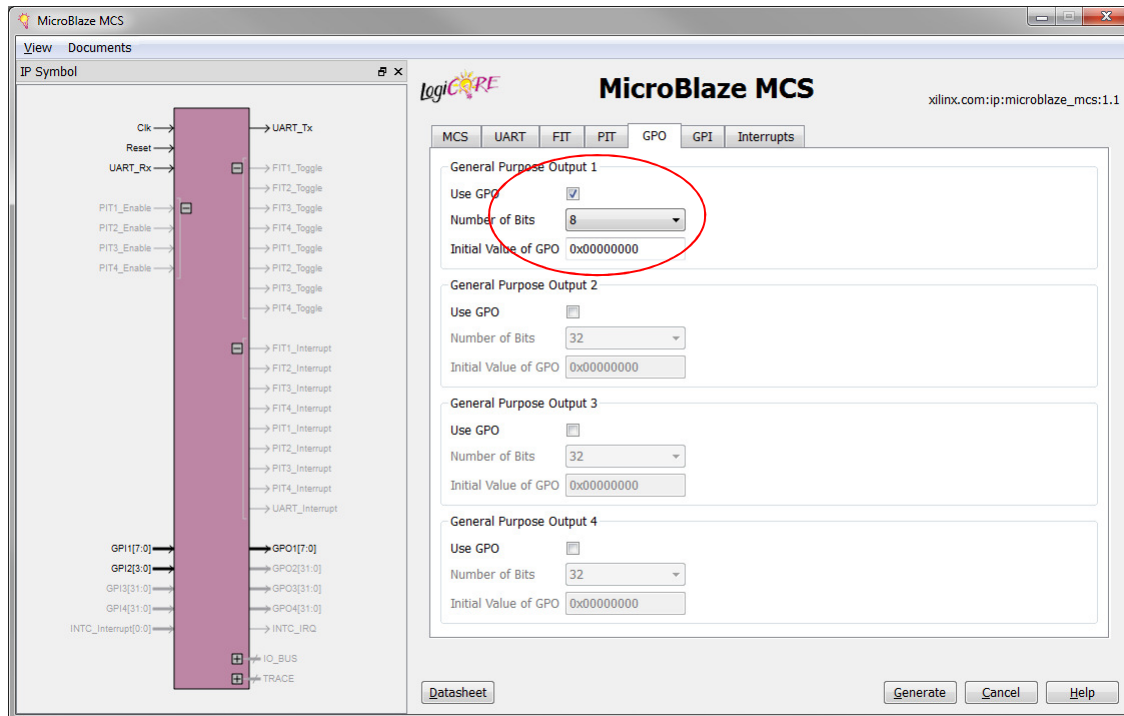
Note: the file name you provide in the *New Source File Name Dialog* box will determine the component name. In the screen shots below this is shown as 'microblaze\_mcs'. Later in this tutorial you will need the name you provided.



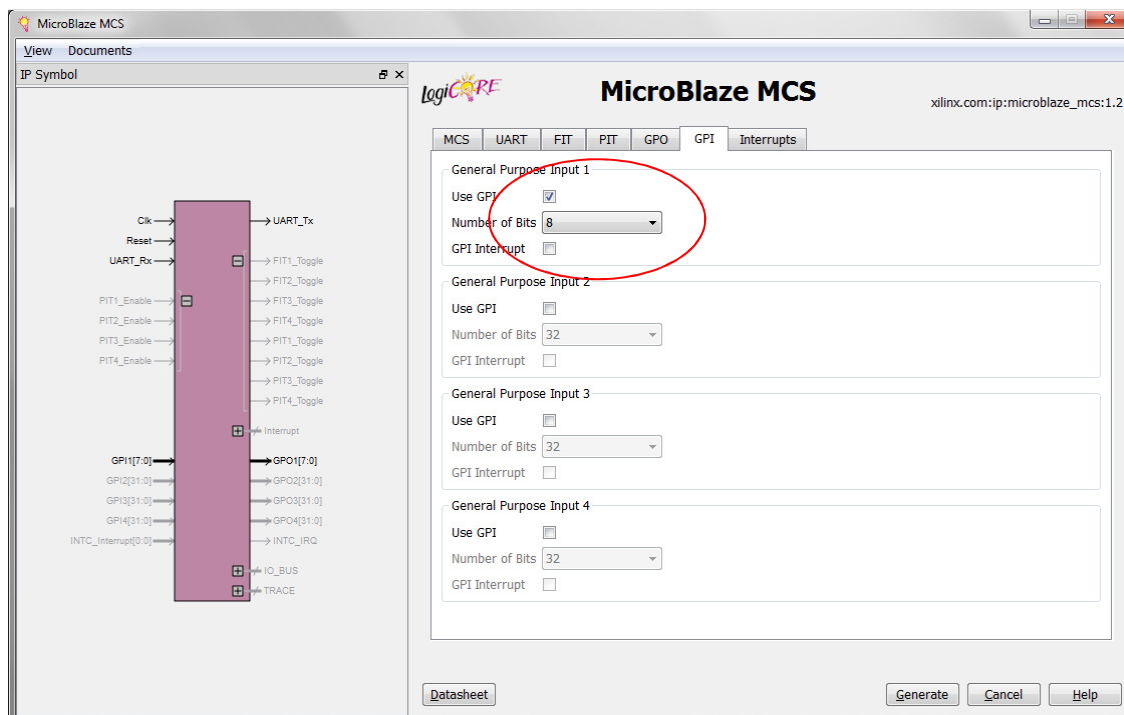
- Set the Input Clock Frequency to match your Nexys3 board (100MHz)
- Increase the memory size from 8KB to 16KB (allows for slightly larger C program)
- Note the Instance Hierarchical Design Name 'mcs\_0' (we will need this later)
- Select the UART Tab and enable the receiver and transmitter and select your baud rate:



Add an 8-bit GPO (we will connect to LEDs later):

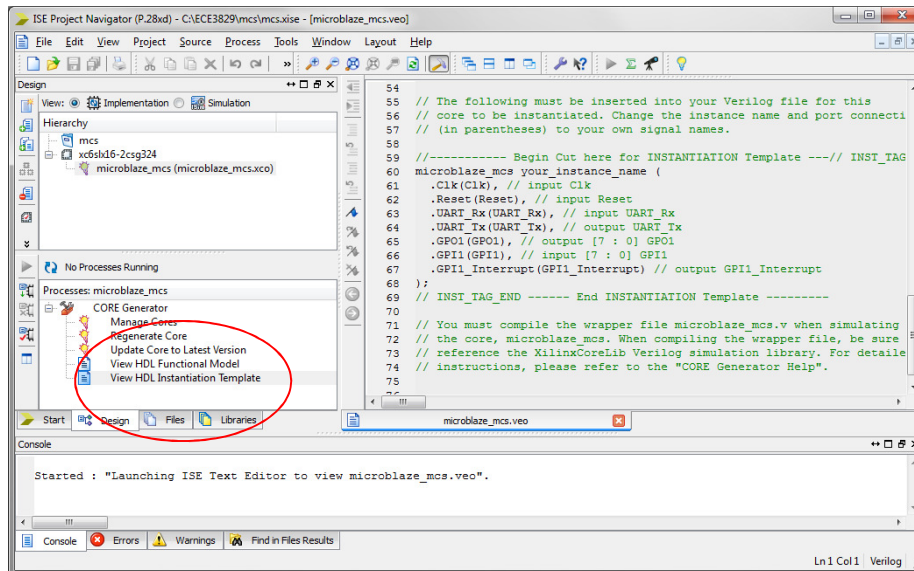


Add an 8-bit GPI (we will connect to the slider switches later):



Click on *Generate* – wait a few minutes (approx. 5 minutes) for the core to be created.

Select the microblaze-mcs core in the Hierarchy Pane then expand the *CORE Generator* in the Processes pane and select the “View HDL Instantiation Template” :

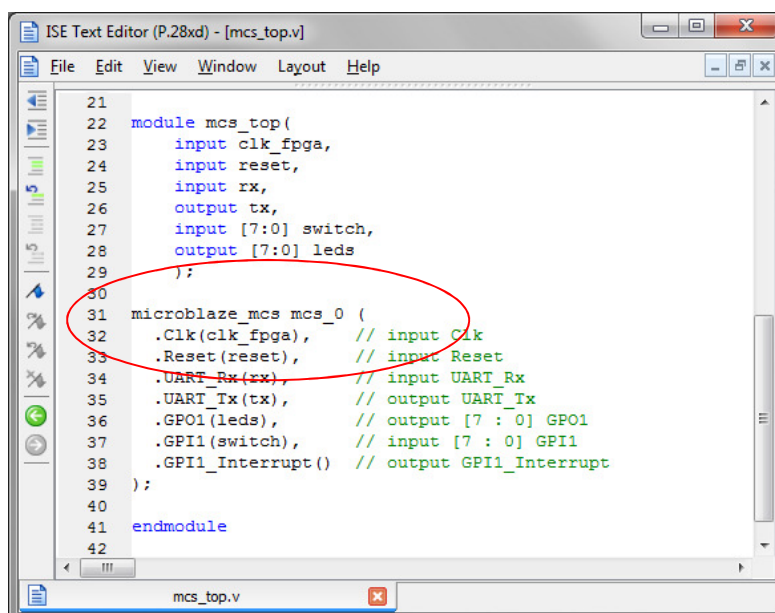


Note: we are using Verilog in this example but by changing the project settings preferred language you can create a VHDL component instead.

Create a new top level with connections to the clock and peripherals on the Nexys board and then instantiate the microblaze core by using the instantiation template provided.

Note: you may see a GPI1\_Interrupt signal (if so you can ignore this port – just leave it open)

**Important:** Use the component name you used and the instance name ‘mcs\_0’ mentioned earlier. In this example the component name is ‘microblaze\_mcs’ and instance name is ‘mcs\_0’.



**Synthesize** your project and make sure there are no warnings or errors.

**Note:** If you are working with the older Nexys2 board (with the Spartan 3E FPGA) you will see three warning messages similar to the following:

```
Analyzing top module <mcs>.
WARNING:Xst:2211 - "ipcore_dir/microblaze_mcs.v" line 36: Instantiating black
box module <microblaze_mcs>.
Module <mcs> is correct for synthesis.

WARNING:Xst:616 - Invalid property "SYN_BLACK_BOX 1": Did not attach to mcs_0.
WARNING:Xst:616 - Invalid property "SYN_NOPRUNE 1": Did not attach to mcs_0.
```

You can ignore these warnings but notice you should still manage to synthesize successfully:

```
Process "Synthesize - XST" completed successfully
```

### Create Merged BMM and Update Tool to Use BMM:

We have to do the following step by hand.

**Note:** If the Tcl Console is not visible (at the bottom of the screen), select **View -> Panels -> Tcl Console** in the menu.

In the Tcl Console type the following TCL script command:

```
source ipcore_dir/microblaze_mcs_setup.tcl
```

You should see:

```
Command>source ipcore_dir/microblaze_mcs_setup.tcl
microblaze_mcs_setup: Found 1 MicroBlaze MCS core.
microblaze_mcs_setup: Added "-bm" option for "microblaze.bmm" to ngdbuild
command line options.
microblaze_mcs_setup: Done.
```

Now **Implement** your design.

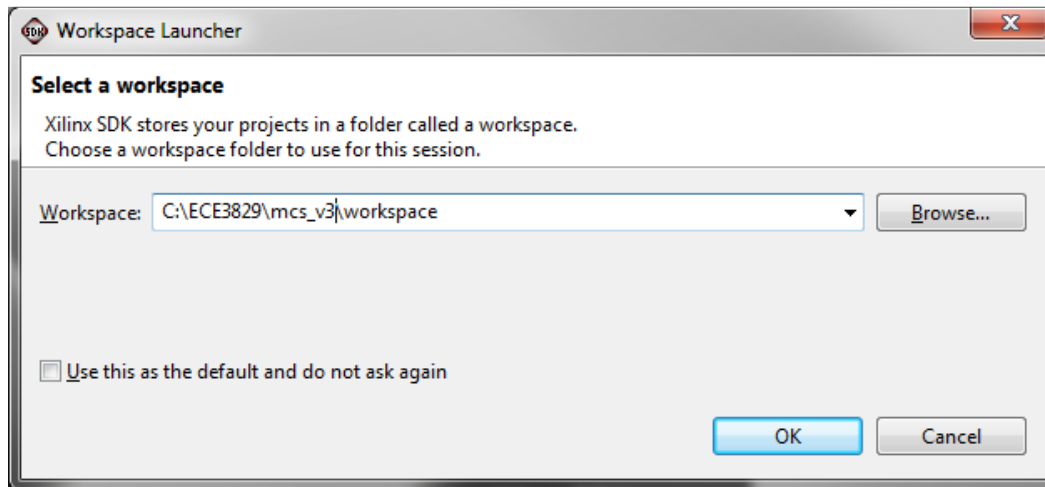
There will be 45 warnings about unconnected Microblaze output pins – ignore these:

```
WARNING:NgdBuild:440 - FF primitive
'mcs_0/U0/microblaze_I/MicroBlaze_Core_I/Area.Decode_I/Using_FPGA.Ext_NM_BRK_
FDRSE' has unconnected output pin
```

The next steps are related to the software development using SDK (Software Development Kit)

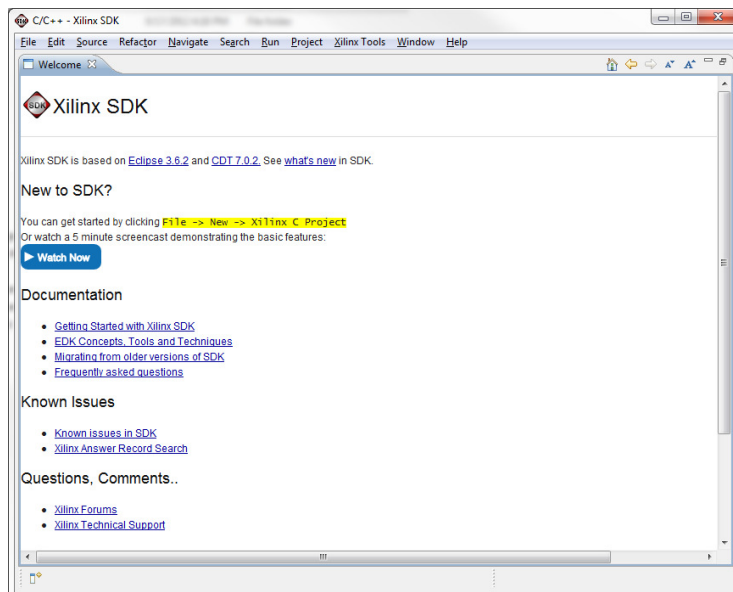
Start SDK and select the *Workspace* to match where your design is stored (for example the project is located in this example at C:\ece3829\mcs\_v3):

Note: when you add the '\workspace' to your project path this new folder will be automatically created.

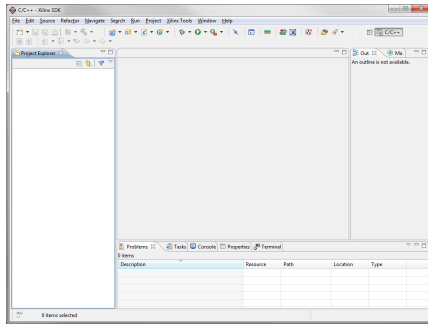


Click **OK**

SDK Starts:

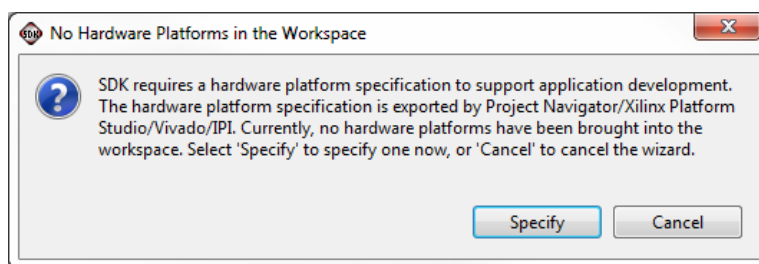


Close the Welcome screen and the Project Explorer Window will open:

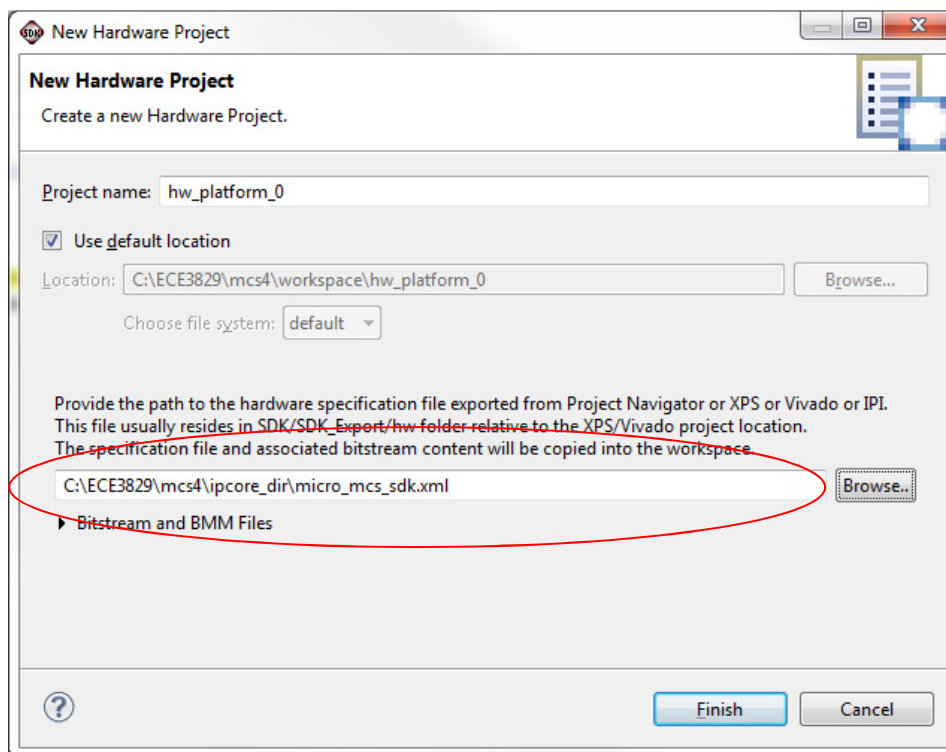


Select **New -> Board Support Package**.

The following Dialog box opens:

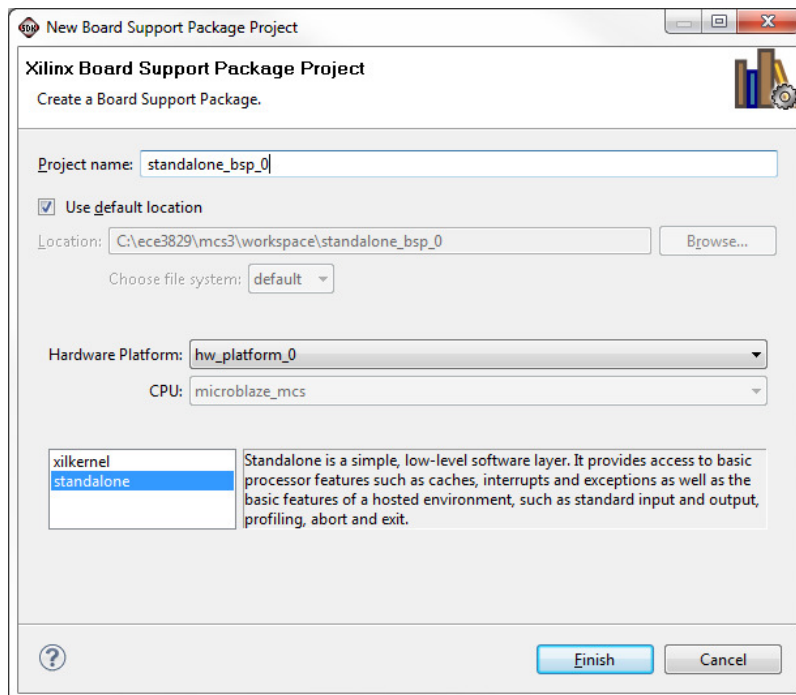


Select **Specify** and browse to select the xml project created by ISE (will be in ipcore\_dir) as shown below:

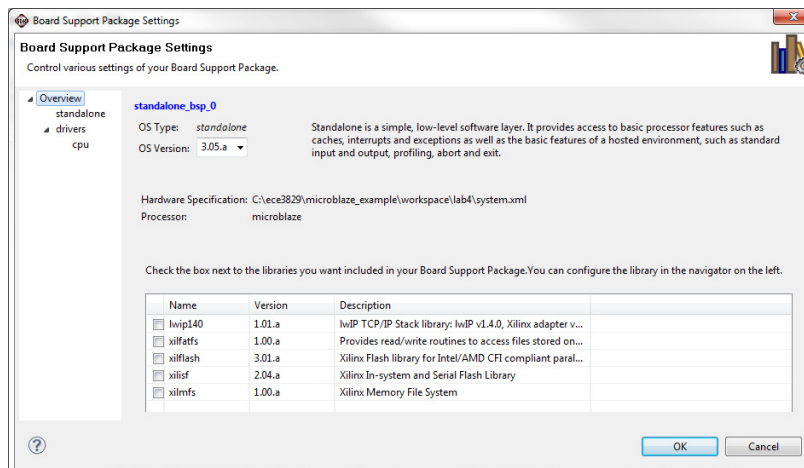


Click **Finish**





Click **Finish**



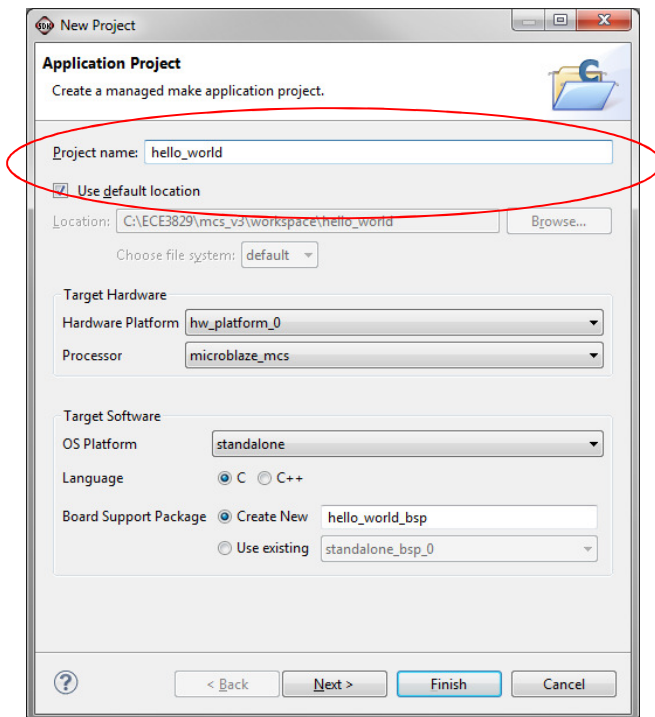
Click **OK**

You should eventually see in the SDK Console Window:

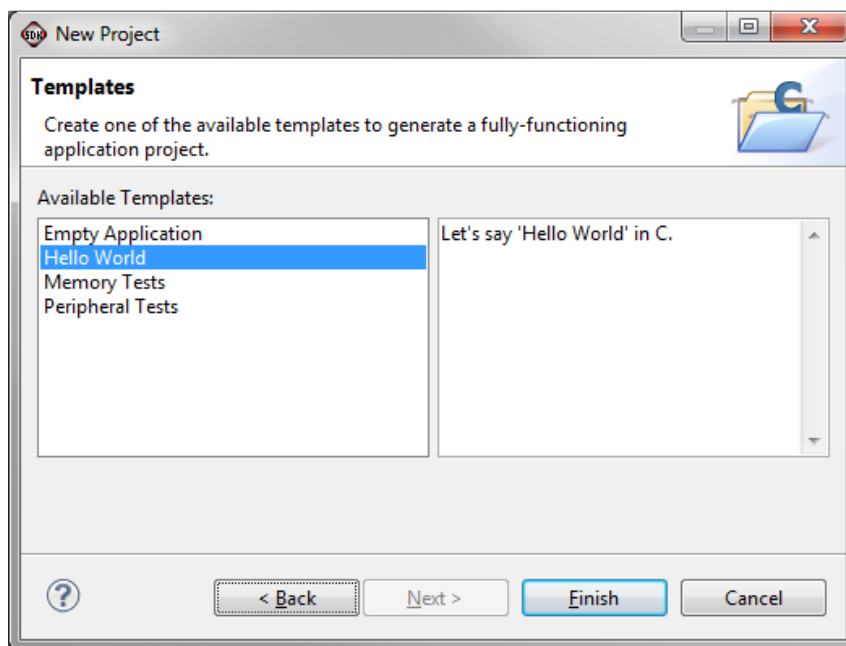
```
"Compiling iomodule"
"Compiling cpu"
Running execs_generate.
'Finished building libraries'
```

Now we will create a new C program:

Select **File => New Application Project** and Type 'hello\_world' for the project name



Click **Next**



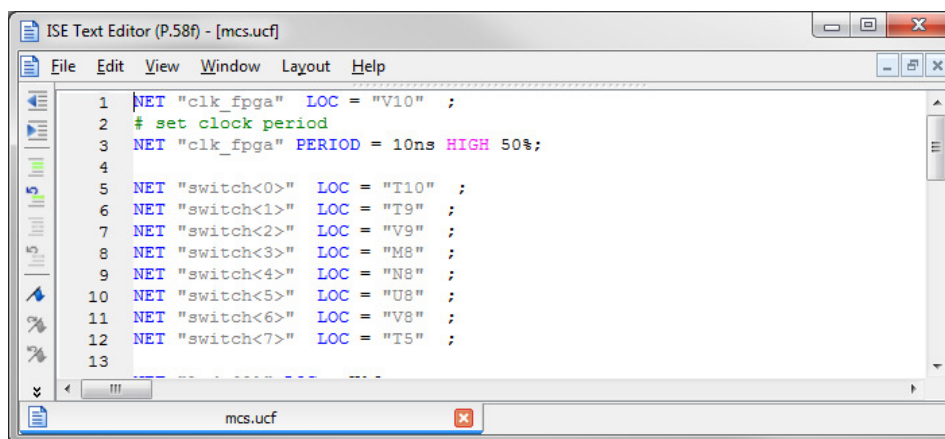
Select **Hello World** and click Finish

Hello\_world\_0.elf is produced (Executable and Linkable Format):

```
ELF file    : hello_world.elf
elfcheck passed.
Finished building: hello_world.elf.elfcheck
' '
```

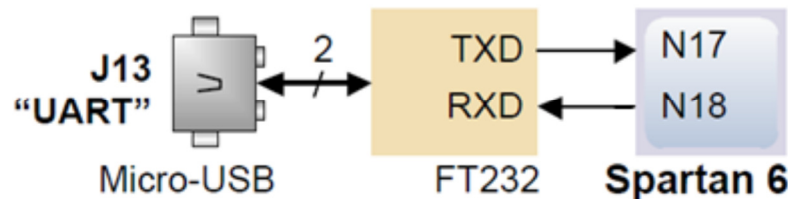
Now go back to ISE Project Navigator add a UCF file to match your Nexys board (remember to include a period timing constraint).

For example:



A comment regarding the UART connection:

In the Nexys3 board reference manual the UART TX and RX are shown as follows. This is showing the direction of transmission as seen by the UART.



So this means that the FPGA transmits on N18 (port 'tx' in the UCF file) and receives on N17 (port 'rx')

Create a bit file by running the **Generate Programming File**

Next we need to add another TCL command:

**Update Tool to Use Software, Update Bitstream with Software and Generate Simulation Files:**

Type the following command in the Tcl Console:

```
microblaze_mcs_data2mem workspace/sdk-program/Debug/sdk-program.elf
```

Where the 'sdk-program' is replaced by *hello\_world* in this example (two places).

You should see:

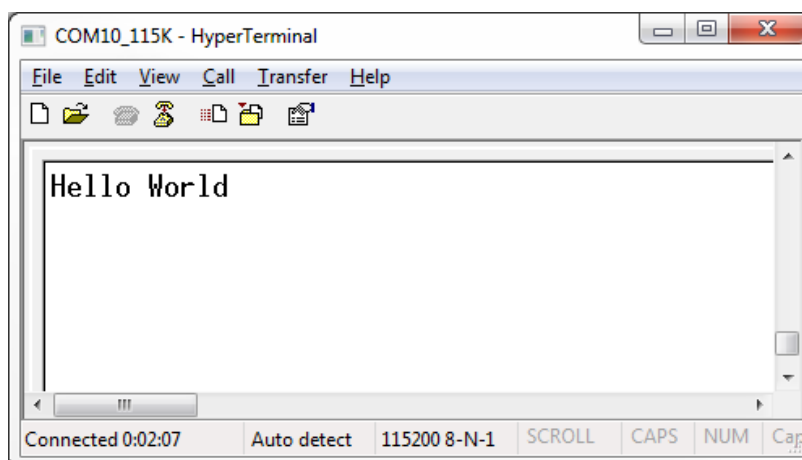
```
Command>microblaze_mcs_data2mem workspace/hello_world/debug/hello_world.elf
microblaze_mcs_data2mem: Found 1 MicroBlaze MCS core.
microblaze_mcs_data2mem: Using "hello_world_0.elf" for microblaze_mcs
microblaze_mcs_data2mem: Existing bitgen "-bd" options unchanged.
microblaze_mcs_data2mem: Running "data2mem" to create simulation files.
microblaze_mcs_data2mem: Running "data2mem" to update bitstream with software.
microblaze_mcs_data2mem: Done.
```

Note: if you see a question mark '?' next to the **Generate Programming File** in the Processes window rerun the **Generate Programming File** process to create an updated bit file with the new C program added (you do not need to redo any of the previous synthesis or implementation steps). You will need to do this anytime you modify the C program.

Connect the board to a PC using an RS232 cable (Nexys2) or a USB cable (Nexys3) .

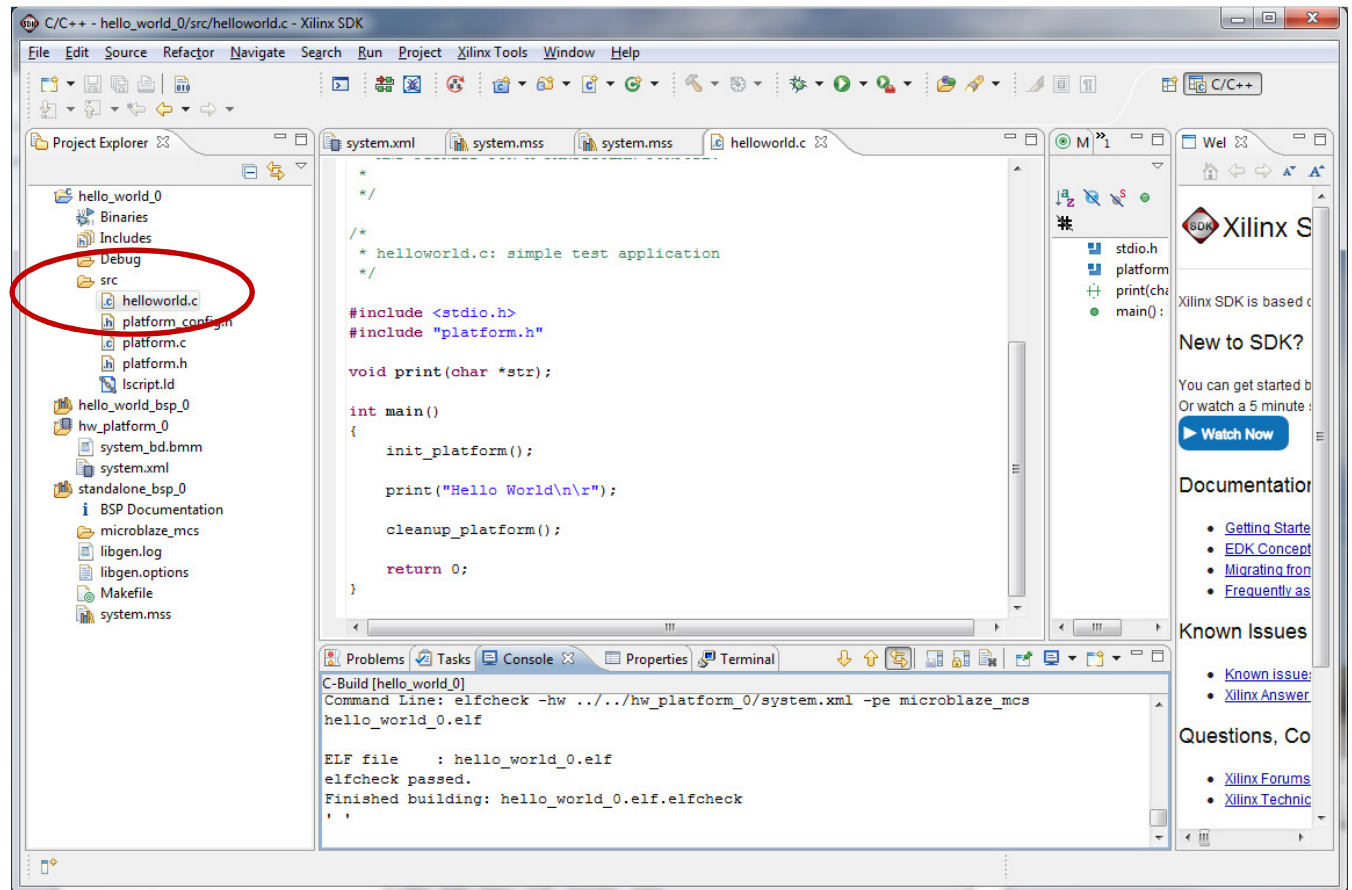
Using the Digilent Adept tool you can now download the fpga bit file from the main project directory to your board.

You should see "Hello World" appear on a serial communications link such as Putty or a Hyperterminal window:



### Extra: Modifying the C Program.

In the Xilinx XDK program, expand the src folder from the C project ,and double-click on the hello\_world.c file. You can see the C statements:



Modify the statements as required (for example change the “Hello World” to add your name) and then press **save**. A new ELF file is automatically generated.

Back in ISE, **Rerun** the **Generate Programming File** process to create an updated bit file with the new C program added (you do not need to redo any of the previous synthesis or implementation steps).

Download the new bit file to the board and verify the new changes.

**Extra: Accessing the GPIO.**

To access GPI/GPO use `XIOModule_DiscreteRead` and `XIOModule_DiscreteWrite` with channel 1-4 for GPI1-4 and GPO1-4. For example:

```
#include <stdio.h>
#include "platform.h"
#include "xparameters.h" // add
#include "xiomodule.h" // add

void print(char *str);

int main()
{
    init_platform();

    u32 data;
    XIOModule gpi;
    XIOModule gpo;

    print("Reading switches and writing to LED port\n\r");

    data = XIOModule_Initialize(&gpi, XPAR_IOMODULE_0_DEVICE_ID);
    data = XIOModule_Start(&gpi);

    data = XIOModule_Initialize(&gpo, XPAR_IOMODULE_0_DEVICE_ID);
    data = XIOModule_Start(&gpo);

    while (1)
    {
        data = XIOModule_DiscreteRead(&gpi, 1); // read switches (channel 1)
        XIOModule_DiscreteWrite(&gpo, 1, data); // turn on LEDs (channel 1)
    }

    cleanup_platform();

    return 0;
}
```

You can find the API documentation in the SDK Project Explorer, under <BSP Name>/BSP Documentation/iomodule\_v1\_00\_a. Click on "Files", "xiomodule.h" for a list of functions.

**Extra: Modifying the C Program to use xil\_printf**

The usual printf function is too large to fit into the small memory of the Microblaze but you can use the Xilinx light-weight version of printf called xil\_printf.

Here is an example of its use in my C program:

```
counter = 1234;
xil_printf("The counter value is %d in decimal and %x in hex.", counter, counter);
```

And this is what is displayed in hyperterminal:

```
The counter value is 1234 in decimal and 4D2 in hex.
```

xil\_printf is defined in 'stdio.h'.

Note: However I found out that in Xilinx version 14.1 the declaration was missing in this header file and you will see an 'implicit function declaration' warning. It did seem to link without errors and run OK. (This seems to be corrected in Version 14.2 and later so you can probably ignore this step)

But if you see the warning and want to fix it on your own system, right click on the stdio.h at the top of your C program ( #include <stdio.h> ) and select 'Open Declaration'

Add this to line 230

```
void _EXFUN(xil_printf, (const char*, ...));
```

so the nearby lines look like:

```
int _EXFUN(remove, (const char *));
int _EXFUN(rename, (const char *, const char *));
void _EXFUN(xil_printf, (const char*, ...));
#endif
```

**Assembler instructions:**

If you want to see the assembler instructions that are created from your C program look in the hello\_world => Debug => Src folder (top left pane in the Xilinx SDK application) and double-click on the hello\_world\_0.elf file.

If you scroll down this file until you find 'int main()' you will see your C instructions and the corresponding assembler and machine code values. Interesting stuff!

**Extra: Accessing the GPIO, using xil\_printf, and using the UART.**

```

#include <stdio.h>
#include "platform.h"
#include "xparameters.h" // add
#include "xiomodule.h" // add

void print(char *str);

int main()
{
    init_platform();

    u32 data;
    XIOModule iomodule; // iomodule variable for gpi, gpo, and uart

    u8 msg[15] = "This is a test"; // buffer for sending message using XIOModule_Send
    u8 rx_buf[10]; // receive buffer using XIOModule_Recv

    u32 counter;

    // example using xil_printf
    counter = 1234;
    xil_printf("The counter value is %d in decimal and %x in hex\n\r", counter,
counter);

    print("Read switches, write to LED port, and UART send and receive chars\n\r");

    // Initialize module to obtain base address
    data = XIOModule_Initialize(&iomodule, XPAR_IOMODULE_0_DEVICE_ID);
    data = XIOModule_Start(&iomodule);

    // Need to call CfgInitialize to use UART Send and Recv functions
    // int XIOModule_CfgInitialize(XIOModule *InstancePtr, XIOModule_Config *Config,
u32 EffectiveAddr);
    // note config and effective address arguments are not used
    data = XIOModule_CfgInitialize(&iomodule, NULL, 1);
    xil_printf("CFInitialize returned (0 = success) %d\n\r", data);

    // Send 12 characters using Send
    // Send is non-blocking so must be called in a loop, may return without sending a
character
    // unsigned int XIOModule_Send(XIOModule *InstancePtr, u8 *DataBufferPtr, unsigned
int NumBytes);
    const int count = 14;
    int index = 0;
    while (index < count) {
        data = XIOModule_Send(&iomodule, &msg[index], count - index);
        index += data;
    }
    xil_printf("\n\rThe number of bytes sent was %d\n\r", index);

    // Another way to send individual characters
    outbyte('X');
    outbyte(0x37); // number '7'
    outbyte('Z');
    outbyte('\n'); // line feed

    // Receive a character and store in rx_buf
    // unsigned int XIOModule_Recv(XIOModule *InstancePtr, u8 *DataBufferPtr, unsigned
int NumBytes);
    while
        ((data = XIOModule_Recv(&iomodule, rx_buf, 1)) == 0);

```



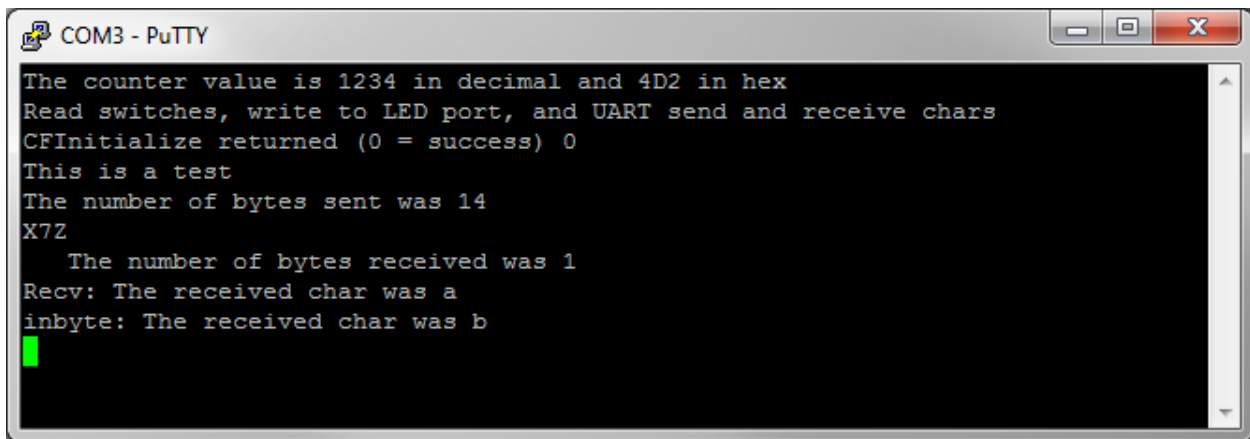
```
xil_printf("The number of bytes received was %d\n\r", data);
xil_printf("Recv: The received char was %c\n\r", rx_buf[0]);

// Another way to receive a single character
rx_buf[0] = inbyte();
xil_printf("inbyte: The received char was %c\n\r", rx_buf[0]);

while (1)
{
    //data = XIOModule_DiscreteRead(&iomodule, 1);        // read switches (channel
1)    data = XIOModule_DiscreteRead(&iomodule, 2); // read push (channel 2)
    XIOModule_DiscreteWrite(&iomodule, 1, data); // turn on LEDs (channel 1)
}

cleanup_platform();

return 0;
}
```



```
COM3 - PuTTY
The counter value is 1234 in decimal and 4D2 in hex
Read switches, write to LED port, and UART send and receive chars
CFInitialize returned (0 = success) 0
This is a test
The number of bytes sent was 14
X7Z
The number of bytes received was 1
Recv: The received char was a
inbyte: The received char was b
█
```