**Lab:\_\_6\_\_ Report Evaluation Form: EE 316 Computer Engineering Junior Lab: Spring 2016**

**Report Writer \_\_\_James Bruska\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Grade \_\_\_\_\_\_\_/50**

**I. Report Content (40 points): For each area below, 0 points are given if it** is **completely missing.**

**A. Summary (Executive summary or abstract) \_\_\_\_/4**

Should summarize the design problem, the overall approach, and the results in terms of how well the specifications were met.

\_\_\_\_Excellent \_\_\_\_\_Good \_\_\_\_\_Average \_\_\_\_\_Fair \_\_\_\_\_None

**B. Design Problem Statement (Requirements and Specifications)** **\_\_\_\_/4**  
Should specify what was to be designed, what assumptions were made, what constraints were placed on your design.

\_\_\_\_Excellent \_\_\_\_\_Good \_\_\_\_\_Average \_\_\_\_\_Fair \_\_\_\_\_None

**C. Problem Decomposition** \_\_\_\_**/4**

Should describe the overall design in terms of functional blocks needed to meet the specifications. Should describe what each functional block is intended to do. There should be a block diagram that includes inputs and outputs of each functional block.

\_\_\_\_Excellent \_\_\_\_\_Good \_\_\_\_\_Average \_\_\_\_\_Fair \_\_\_\_\_None

**D. Detailed Design** **and Module level Testing (Design for Testing)** \_\_\_\_**/16**

* **(5 points)** Should describe how the design works in sufficient detail that a knowledgeable person is able to understand and verify that the design is technically sound.
* **(6 points)** There should be documentation sufficient to allow a knowledgeable person to duplicate your design. You should have schematics for all hardware and source code for all software, programmable logic, VHDL, etc. Source code is **not** needed for the “obvious” parts; you should have pseudo or actual code for the significant parts of your source code. [Include a CD, if needed].
* **(5 points)** Should describe how the design was tested at the module level and corrections made.

\_\_\_\_Excellent \_\_\_\_\_Good \_\_\_\_\_Average \_\_\_\_\_Fair \_\_\_\_\_None

**E. Alternative Designs** \_\_\_\_**/4**

Should describe alternative design approaches to meeting the specifications. You should give the rationale for selecting the design approach you used in favor of others.

\_\_\_\_Excellent \_\_\_\_\_Good \_\_\_\_\_Average \_\_\_\_\_Fair \_\_\_\_\_None

**F. Specification Testing \_\_\_\_/4**

Should describe how the completed design was tested to verify, it meets specifications or to determine deviations from the specs.

\_\_\_\_Excellent \_\_\_\_\_Good \_\_\_\_\_Average \_\_\_\_\_Fair \_\_\_\_\_None

**G. Results and Analysis \_\_\_\_/4**

Should describe how well your design met the specifications and discuss any deviations from those specifications. Give an analysis of any deviations from the specification and **explain** what should be done to correct these.

\_\_\_\_Excellent \_\_\_\_\_Good \_\_\_\_\_Average \_\_\_\_\_Fair \_\_\_\_\_None

**II. Report form. (10 points) The report should be produced in a professional manner. Basis of Evaluation:**

**2 - no deficiencies, 1- no more than 2 minor deficiencies, 0-1 major or multiple minor deficiencies**

|  |  |
| --- | --- |
| The report is well organized, with explanatory text for each section. | **\_\_\_/2** |
| The written language is clear and grammatically correct. | **\_\_\_/2** |
| All figures and tables are referenced in the text and appear shortly after the reference, or are located in an appendix with appropriate text to refer the reader there. All figures and tables have a caption | **\_\_\_/2** |
| All diagrams and schematics are drawn using CAD type software with no hand drawing or lettering | **\_\_\_/2** |
| The report includes a list of references used in completing the design and documenting your results | **\_\_\_/2** |

**Synchronous Digital System Using Xilinx MicroBlaze Processor**

**Author: James Bruska**

**Teammates: Daniel Heck and Matthew Michaels**

**Instructor: Dr. Abul Khondker**

**April 28, 2017**

**Executive Summary**

This project aimed to create a system in which a Xilinx’s NexysTM 4 board was able to display an output sequence using a Xilinx MicroBlaze processor. The system would display a given sequence on four seven-segment displays and change every second. There were also three buttons: one toggle sequence direction (forward or backward); one paused the sequence; one reset the system. The system was created and tested in Vivado 2016.3. The code was written using C. The system functions according to the provided design criteria.

**Design Problem Statement**

This system displays a repeating nine part output sequence that changes on a “effective clock” of 1 Hz that triggers via interrupt. The HEX output sequence is as follows: 0000, 0A0A, A0A0, FE45, ABCD, DCBA, FEED, DEAD, BEEF. This sequence is displayed on the four rightmost seven-segment displays. There are three buttons that control the execution of the program: sequence direction, system enable/disable, asynchronous system reset. The system runs on a Xilinx MicroBlaze Processor.

The specific design criteria:

Hardware Usage:

* Design for Xilinx’s NexysTM 4 board
* Uses the MicroBlaze soft-core processor
* All pins are connected internally via Xil kernel board support package
* Must use right most four seven segment displays to show the sequence
* Must add and use needed interrupts for buttons and timers.

Coding limitations:

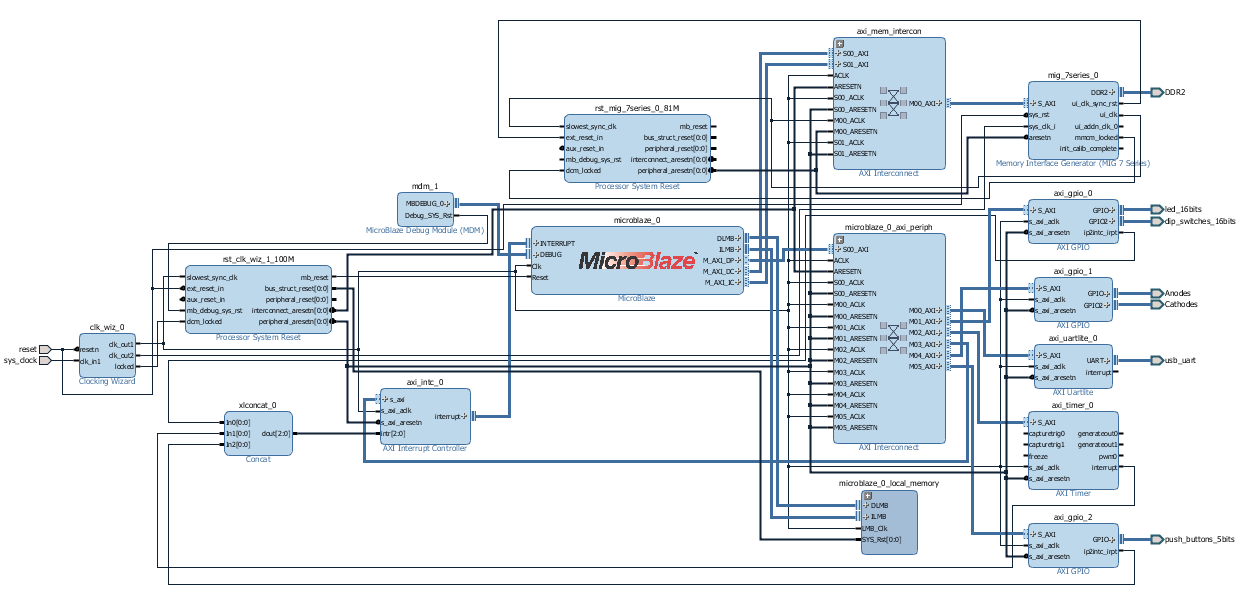
* Implement using Vivado 2016.3 and Xilinx SDK
* Debug the design using debug cores and C print statements
* Operation of states must be controlled via C and not VHDL.

System Operation Specifications:

* System starts in forward mode and counting
* Center button initializes the system asynchronously.
* System only begins counting when the center button is released
* Left button toggles the sequence direction
* Right button toggles system enable/disable

**Problem Decomposition**

Figure 1 displays an overview of the main hardware components of the system.



*Figure 1 – Hardware Diagram of MicroBlaze Processor*

All components run using the main 100MHz board clock if a clock is needed.

The first important part of the project was generating the microprocessor and loading it onto the FPGA. This was done by following the tutorial provided by Professor Khondker. This can be found on the last page of the report. This also has a link in it to digilentinc.com. This must be followed as well. The microprocessor will compile fine if these are followed.

A common and easy mistake that can be made is forgetting to set the version number in the file to the version number of Vivado that the user is compiling with. This will break it very easily. Another error is trying to load the tcl file into an already created project. This will not work correctly either.

For the purpose of the project, another interrupt and axi\_gpio component also had to be added in order to control the buttons. This was not present in the original tcl file and the project needs to be re-synthesized, re-implemented, and re-generated after the components are added.

**Detailed Design**

This is the design of the system seen below in Figure 2.



*Figure 2 – Physical Picture of the System*

For the design of the C code, the project edited some of the pre-existing functions in order to achieve a working project. The main function is actually only 50 lines because the code was aimed to be simple to read and code with in the future. This also means that the code was very in-optimized.

There were three main functions that were written/edited for ease of use: Xgpio\_SevenSegWrite(), GpioIntrExample(), and TmrCtrIntrExample(). The TimerCounterHandler function and Button\_DeviceInterruptHandler function were also implemented to allow very easy and dynamic change to how the buttons and timer acted. These functions used other functions provided within the original example functions in order to edit the correct registers.

The overall flow of the program went as follows

1.) The system is initialized and all of the interrupts were readied and enabled.

2.) The button handler was started

If the interrupt was triggered then internal variables were set to control other parts of the system.

These do not have the same race condition problem that hardware has.

3.) The timer was started to trigger every 2.5ms.

This reset the count value on its own after the interrupt went off.

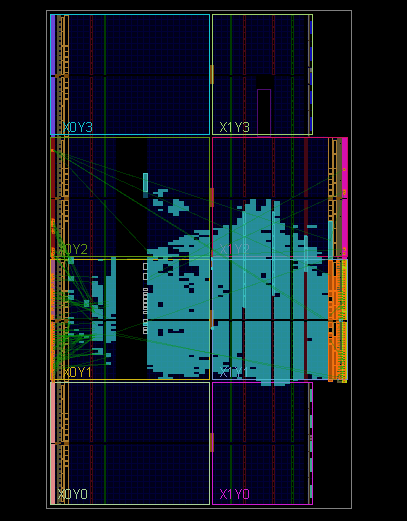
The value for seven segment sequence was displayed for a single position.

2.5ms was chosen because that allowed 10ms for the entire sequence.

If this interrupt occurred 99 times then it changed the sequence value.

The actual registers that were being edited by these functions can be found in xgpio.c, xintc.c, and xtmrctr.c. These files are shown in Appendix B through Appendix D.

Below, Figure 3 displays the implemented design of the microprocessor. All of the areas used by the microprocessor on the FPGA are highlighted in light blue.



*Figure 3 – Implementation of the MicroBlaze Processor (Highlighted)*

**Alternative Designs**

Due to the system simplicity, there was not much that could be done differently. A different kernel could have been used; but that would not have affected the programs that we generated as they used function calls that were already in place in the board support package (and consistent across multiple kernels).

There was an attempt made to multi-thread the program, but this did not provide any more benefit and only added more overhead. It also did not work based on interrupts as the scheduling algorithms needed so that race conditions could be prevented would be an excessive for a project of this size.

**Module and Specifications Testing**

Each component of the system was tested using print statements in C to the terminal and general output of the system on the seven segment displays. The running state machine of the system was incredibly basic; so all states of the state machine were tested by hand. This meant that we tested

9 sequences \* 2 directions \* 2 system enabled \* 2 reset = 72 states total.

For the sake of space, the pictures for all 72 states have not been included.

There was also an attempt to put debug cores into the program. We were successful in attaching them and compiling them into the system, but wewere not able to gather meaningful results due to poor positioning of the debug cores.

As a note, we did not test the microprocessor either as we assumed the published program operated as intended. We did test the new interrupt component when it was added by printing the interrupt register value to the terminal. This allowed us to easily gather information about how the button presses worked and verify it with what the internal hardware design showed would occur.

**Results and Analysis**

This design followed all of the instructions set forth within the design constraints. This can be seen by the Module and Specifications Testing. The C code is provided in Appendix A through Appendix D.

**References**

Xilinx. *Vivado Design Suite User Guide*. 2017. Web. 2 May 2017.

Xilinx. *MicroBlaze Processor Reference Guide*. 2017. Web. 2 May 2017.

Xilinx. *Vivado Design Suite AXI Reference Guide*. 2017. Web. 2 May 2017.

**Appendices**

1. main.c

/\*

\* main.c

\*

\* Created on: Apr 13, 2017

\* Revised on: Apr 25, 2017

\* Author: James Bruska, Daniel Heck, Matthew Michaels, aaaaand Abul Khondker

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* Copyright (C) 2009 - 2014 Xilinx, Inc. All rights reserved.

\*

\* Permission is hereby granted, free of charge, to any person obtaining a copy

\* of this software and associated documentation files (the "Software"), to deal

\* in the Software without restriction, including without limitation the rights

\* to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

\* copies of the Software, and to permit persons to whom the Software is

\* furnished to do so, subject to the following conditions:

\*

\* The above copyright notice and this permission notice shall be included in

\* all copies or substantial portions of the Software.

\*

\* Use of the Software is limited solely to applications:

\* (a) running on a Xilinx device, or

\* (b) that interact with a Xilinx device through a bus or interconnect.

\*

\* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

\* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

\* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL

\* XILINX BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,

\* WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF

\* OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

\* SOFTWARE.

\*

\* Except as contained in this notice, the name of the Xilinx shall not be used

\* in advertising or otherwise to promote the sale, use or other dealings in

\* this Software without prior written authorization from Xilinx.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\* helloworld.c: simple test application

\*

\* This application configures UART 16550 to baud rate 9600.

\* PS7 UART (Zynq) is not initialized by this application, since

\* bootrom/bsp configures it to baud rate 115200

\*

\* ------------------------------------------------

\* | UART TYPE BAUD RATE |

\* ------------------------------------------------

\* uartns550 9600

\* uartlite Configurable only in HW design

\* ps7\_uart 115200 (configured by bootrom/bsp)

\*/

#include <stdio.h>

#include "xil\_printf.h"

#include "xparameters.h"

#include "xgpio.h"

#include "xil\_io.h"

#include "xintc.h"

#include "xtmrctr.h"

#define printf xil\_printf /\* A smaller footprint printf \*/

#define LED\_POS 1

#define LED\_VAL 2

#ifdef XPAR\_INTC\_0\_DEVICE\_ID

#define INTC XIntc

#define INTC\_HANDLER XIntc\_InterruptHandler

#endif /\* XPAR\_INTC\_0\_DEVICE\_ID \*/

#ifdef XPAR\_INTC\_0\_DEVICE\_ID

#define INTC\_DEVICE\_ID XPAR\_INTC\_0\_DEVICE\_ID

#endif /\* XPAR\_INTC\_0\_DEVICE\_ID \*/

/\*

\* The following constant is used to set the reset value of the timer counter,

\* making this number larger reduces the amount of time this example consumes

\* because it is the value the timer counter is loaded with when it is started

\*/

XGpio GpioLedOutput; /\* The driver instance for GPIO Device configured as O/P \*/

/\*

\* The following variables are shared between non-interrupt processing and

\* interrupt processing such that they must be global.

\*/

unsigned char reset = 0;

unsigned char fwd\_bwd = 0xFF;

unsigned char enabled = 0xFF; // All 1s so I can just use bitwise not

int led\_outputs[9][4] = {{0,0,0,0}, {0,10,0,10}, {10,0,10,0}, {15,14,4,5}, {10,11,12,13}, {13,12,11,10}, {15,14,14,13}, {13,14,10,13}, {11,14,14,15}};

int led\_output\_addr = 0;

int led\_pos = 0;

int led\_counter = 0;

int debouncer = 0;

void XGpio\_SevenSegWrite(XGpio \* InstancePtr, u32 position, u32 value);

int IntcInterruptSetup(XIntc \*IntcInstancePtr, u16 DeviceId);

void Button\_DeviceInterruptHandler(void \*DeviceId);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Constant Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifndef TESTAPP\_GEN

/\*

\* The following constants map to the XPAR parameters created in the

\* xparameters.h file. They are defined here such that a user can easily

\* change all the needed parameters in one place.

\*/

#define GPIO\_DEVICE\_ID XPAR\_GPIO\_2\_DEVICE\_ID

#define GPIO\_CHANNEL1 1

#ifdef XPAR\_INTC\_0\_DEVICE\_ID

#define INTC\_GPIO\_INTERRUPT\_ID XPAR\_INTC\_0\_GPIO\_2\_VEC\_ID

#define INTC\_DEVICE\_ID XPAR\_INTC\_0\_DEVICE\_ID

#endif /\* XPAR\_INTC\_0\_DEVICE\_ID \*/

/\*

\* The following constants map to the XPAR parameters created in the

\* xparameters.h file. They are only defined here such that a user can easily

\* change all the needed parameters in one place.

\*/

#define TMRCTR\_DEVICE\_ID XPAR\_TMRCTR\_0\_DEVICE\_ID

#define TMRCTR\_INTERRUPT\_ID XPAR\_INTC\_0\_TMRCTR\_0\_VEC\_ID

#endif /\* TESTAPP\_GEN \*/

#ifdef XPAR\_INTC\_0\_DEVICE\_ID

#define INTC\_DEVICE\_ID XPAR\_INTC\_0\_DEVICE\_ID

#define INTC XIntc

#define INTC\_HANDLER XIntc\_InterruptHandler

#endif /\* XPAR\_INTC\_0\_DEVICE\_ID \*/

#define TIMER\_CNTR\_0 0

#ifdef XPAR\_INTC\_0\_DEVICE\_ID

#define INTC XIntc

#define INTC\_HANDLER XIntc\_InterruptHandler

#endif /\* XPAR\_INTC\_0\_DEVICE\_ID \*/

/\*

\* The following constant is used to set the reset value of the timer counter,

\* making this number larger reduces the amount of time this example consumes

\* because it is the value the timer counter is loaded with when it is started

\*/

//#define RESET\_VALUE 0xfa0a1eff /\*This exactly 1 second 100,000,000 clocks\*/

//#define RESET\_VALUE 0xffd9da5f /\*This exactly 25 milliseconds 2,500,000 clocks\*/

#define RESET\_VALUE 0xfffc2f6f /\*This exactly 2.5 milliseconds 250,000 clocks\*/

// 250ms time = 1s total due to 4 LCDs

// 2.5ms time = 10ms total due to 4 LCDs

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Function Prototypes \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void GpioHandler(void \*CallBackRef);

int GpioIntrExample(INTC \*IntcInstancePtr, XGpio \*InstancePtr,

u16 DeviceId, u16 IntrId,

u16 IntrMask, u32 \*DataRead);

int GpioSetupIntrSystem(INTC \*IntcInstancePtr, XGpio \*InstancePtr,

u16 DeviceId, u16 IntrId, u16 IntrMask);

void GpioDisableIntr(INTC \*IntcInstancePtr, XGpio \*InstancePtr,

u16 IntrId, u16 IntrMask);

int TmrCtrIntrExample(INTC\* IntcInstancePtr,

XTmrCtr\* InstancePtr,

u16 DeviceId,

u16 IntrId,

u8 TmrCtrNumber);

static int TmrCtrSetupIntrSystem(INTC\* IntcInstancePtr,

XTmrCtr\* InstancePtr,

u16 DeviceId,

u16 IntrId,

u8 TmrCtrNumber);

void TimerCounterHandler(void \*CallBackRef, u8 TmrCtrNumber);

void TmrCtrDisableIntr(INTC\* IntcInstancePtr, u16 IntrId);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Variable Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\* The following are declared globally so they are zeroed and so they are

\* easily accessible from a debugger

\*/

XGpio GpioButton; /\* The Instance of the GPIO Driver \*/

INTC Intc; /\* The Instance of the Interrupt Controller Driver \*/

static u16 GlobalIntrMask; /\* GPIO channel mask that is needed by

\* the Interrupt Handler \*/

#ifndef TESTAPP\_GEN

INTC InterruptController; /\* The instance of the Interrupt Controller \*/

XTmrCtr TimerCounterInst; /\* The instance of the Timer Counter \*/

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main()

{

u32 status;

print("BEGIN SEQUENCE\n\r");

status = XGpio\_Initialize(&GpioLedOutput, XPAR\_AXI\_GPIO\_1\_DEVICE\_ID);

if (status != XST\_SUCCESS) {

return XST\_FAILURE;

}

/\* Set the direction for all signals to be outputs \*/

XGpio\_SetDataDirection(&GpioLedOutput, LED\_POS, 0x0);

XGpio\_SevenSegWrite(&GpioLedOutput, 2, 9);

int Status;

u32 DataRead;

Status = GpioIntrExample(&Intc, &GpioButton,

GPIO\_DEVICE\_ID,

INTC\_GPIO\_INTERRUPT\_ID,

GPIO\_CHANNEL1, &DataRead);

/\*

\* Run the Timer Counter - Interrupt example.

\*/

Status = TmrCtrIntrExample(&Intc,

&TimerCounterInst,

TMRCTR\_DEVICE\_ID,

TMRCTR\_INTERRUPT\_ID,

TIMER\_CNTR\_0);

if (Status != XST\_SUCCESS) {

return XST\_FAILURE;

}

while(1){

if (debouncer > 0){

--debouncer;

}

}

return XST\_SUCCESS;

//cleanup\_platform();

return 0;

}

void XGpio\_SevenSegWrite(XGpio \* InstancePtr, u32 position, u32 value)

{

u32 convertValToSevenSeg[16] = {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E};

XGpio\_DiscreteWrite(&GpioLedOutput, LED\_POS, ~(1 << position));

XGpio\_DiscreteWrite(&GpioLedOutput, LED\_VAL, convertValToSevenSeg[value]);

}

void Button\_DeviceInterruptHandler(void \*DeviceId)

{

printf("Interrupted: %d", Xil\_In8(XPAR\_GPIO\_2\_BASEADDR));

}

int TmrCtrIntrExample(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);

/\*

\* Start the timer counter such that it's incrementing by default,

\* then wait for it to timeout a number of times

\*/

XTmrCtr\_Start(TmrCtrInstancePtr, TmrCtrNumber);

return XST\_SUCCESS;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* This function is the handler which performs processing for the timer counter.

\* It is called from an interrupt context such that the amount of processing

\* performed should be minimized. It is called when the timer counter expires

\* if interrupts are enabled.

\*

\* This handler provides an example of how to handle timer counter interrupts

\* but is application specific.

\*

\* @param CallBackRef is a pointer to the callback function

\* @param TmrCtrNumber is the number of the timer to which this

\* handler is associated with.

\*

\* @return None.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void TimerCounterHandler(void \*CallBackRef, u8 TmrCtrNumber)

{

XTmrCtr \*InstancePtr = (XTmrCtr \*)CallBackRef;

/\*

\* Check if the timer counter has expired, checking is not necessary

\* since that's the reason this function is executed, this just shows

\* how the callback reference can be used as a pointer to the instance

\* of the timer counter that expired, increment a shared variable so

\* the main thread of execution can see the timer expired

\*/

if (XTmrCtr\_IsExpired(InstancePtr, TmrCtrNumber)) {

if (led\_pos < 3){

++led\_pos;

}else{

led\_pos = 0;

}

//printf("led\_pos: %d, led\_outputs[%d][%d] = %d\r\n", led\_pos, led\_output\_addr, led\_pos, led\_outputs[led\_output\_addr][led\_pos]);

XGpio\_SevenSegWrite(&GpioLedOutput, 3-led\_pos, led\_outputs[led\_output\_addr][led\_pos]);

if (enabled){

if (led\_pos == 3){

if (led\_counter < 99){

++led\_counter;

}else{

led\_counter = 0;

if (led\_output\_addr < 8 && fwd\_bwd){

++led\_output\_addr;

}else if (led\_output\_addr > 0 && !fwd\_bwd){

--led\_output\_addr;

}else if (fwd\_bwd){

led\_output\_addr = 0;

}else{

led\_output\_addr = 8;

}

}

}

}

//printf("fwd/bwd: %d, reset: %d, enabled: %d\r\n", fwd\_bwd, reset, enabled);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* This function setups the interrupt system such that interrupts can occur

\* for the timer counter. This function is application specific since the actual

\* system may or may not have an interrupt controller. The timer counter could

\* be directly connected to a processor without an interrupt controller. The

\* user should modify this function to fit the application.

\*

\* @param IntcInstancePtr is a pointer to the Interrupt Controller

\* driver Instance.

\* @param TmrCtrInstancePtr is a pointer to the XTmrCtr driver Instance.

\* @param DeviceId is the XPAR\_<TmrCtr\_instance>\_DEVICE\_ID value from

\* xparameters.h.

\* @param IntrId is XPAR\_<INTC\_instance>\_<TmrCtr\_instance>\_VEC\_ID

\* value from xparameters.h.

\* @param TmrCtrNumber is the number of the timer to which this

\* handler is associated with.

\*

\* @return XST\_SUCCESS if the Test is successful, otherwise XST\_FAILURE.

\*

\* @note This function contains an infinite loop such that if interrupts

\* are not working it may never return.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static int TmrCtrSetupIntrSystem(INTC\* IntcInstancePtr,

XTmrCtr\* TmrCtrInstancePtr,

u16 DeviceId,

u16 IntrId,

u8 TmrCtrNumber)

{

int Status;

#ifdef XPAR\_INTC\_0\_DEVICE\_ID

#ifndef TESTAPP\_GEN

/\*

\* Initialize the interrupt controller driver so that

\* it's ready to use, specify the device ID that is generated in

\* xparameters.h

\*/

/\*

Status = XIntc\_Initialize(IntcInstancePtr, INTC\_DEVICE\_ID);

if (Status != XST\_SUCCESS) {

return XST\_FAILURE;

}

\*/

#endif

/\*

\* Connect a device driver handler that will be called when an interrupt

\* for the device occurs, the device driver handler performs the specific

\* interrupt processing for the device

\*/

Status = XIntc\_Connect(IntcInstancePtr, IntrId,

(XInterruptHandler)XTmrCtr\_InterruptHandler,

(void \*)TmrCtrInstancePtr);

if (Status != XST\_SUCCESS) {

return XST\_FAILURE;

}

#ifndef TESTAPP\_GEN

/\*

\* Start the interrupt controller such that interrupts are enabled for

\* all devices that cause interrupts, specific real mode so that

\* the timer counter can cause interrupts thru the interrupt controller.

\*/

Status = XIntc\_Start(IntcInstancePtr, XIN\_REAL\_MODE);

if (Status != XST\_SUCCESS) {

return XST\_FAILURE;

}

#endif

/\*

\* Enable the interrupt for the timer counter

\*/

XIntc\_Enable(IntcInstancePtr, IntrId);

#endif /\* XPAR\_INTC\_0\_DEVICE\_ID \*/

#ifndef TESTAPP\_GEN

/\*

\* 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);

/\*

\* Enable non-critical exceptions.

\*/

Xil\_ExceptionEnable();

#endif

return XST\_SUCCESS;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* This function disables the interrupts for the Timer.

\*

\* @param IntcInstancePtr is a reference to the Interrupt Controller

\* driver Instance.

\* @param IntrId is XPAR\_<INTC\_instance>\_<Timer\_instance>\_VEC\_ID

\* value from xparameters.h.

\*

\* @return None.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void TmrCtrDisableIntr(INTC\* IntcInstancePtr, u16 IntrId)

{

/\*

\* Disable the interrupt for the timer counter

\*/

#ifdef XPAR\_INTC\_0\_DEVICE\_ID

XIntc\_Disable(IntcInstancePtr, IntrId);

#endif

return;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* This is the entry function from the TestAppGen tool generated application

\* which tests the interrupts when enabled in the GPIO

\*

\* @param IntcInstancePtr is a reference to the Interrupt Controller

\* driver Instance

\* @param InstancePtr is a reference to the GPIO driver Instance

\* @param DeviceId is the XPAR\_<GPIO\_instance>\_DEVICE\_ID value from

\* xparameters.h

\* @param IntrId is XPAR\_<INTC\_instance>\_<GPIO\_instance>\_IP2INTC\_IRPT\_INTR

\* value from xparameters.h

\* @param IntrMask is the GPIO channel mask

\* @param DataRead is the pointer where the data read from GPIO Input is

\* returned

\*

\* @return

\* - XST\_SUCCESS if the Test is successful

\* - XST\_FAILURE if the test is not successful

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int GpioIntrExample(INTC \*IntcInstancePtr, XGpio\* InstancePtr, u16 DeviceId,

u16 IntrId, u16 IntrMask, u32 \*DataRead)

{

int Status;

/\* Initialize the GPIO driver. If an error occurs then exit \*/

Status = XGpio\_Initialize(InstancePtr, DeviceId);

if (Status != XST\_SUCCESS) {

return XST\_FAILURE;

}

Status = GpioSetupIntrSystem(IntcInstancePtr, InstancePtr, DeviceId,

IntrId, IntrMask);

if (Status != XST\_SUCCESS) {

return XST\_FAILURE;

}

return Status;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* This function performs the GPIO set up for Interrupts

\*

\* @param IntcInstancePtr is a reference to the Interrupt Controller

\* driver Instance

\* @param InstancePtr is a reference to the GPIO driver Instance

\* @param DeviceId is the XPAR\_<GPIO\_instance>\_DEVICE\_ID value from

\* xparameters.h

\* @param IntrId is XPAR\_<INTC\_instance>\_<GPIO\_instance>\_IP2INTC\_IRPT\_INTR

\* value from xparameters.h

\* @param IntrMask is the GPIO channel mask

\*

\* @return XST\_SUCCESS if the Test is successful, otherwise XST\_FAILURE

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int GpioSetupIntrSystem(INTC \*IntcInstancePtr, XGpio \*InstancePtr,

u16 DeviceId, u16 IntrId, u16 IntrMask)

{

int Result;

GlobalIntrMask = IntrMask;

#ifndef TESTAPP\_GEN

/\*

\* Initialize the interrupt controller driver so that it's ready to use.

\* specify the device ID that was generated in xparameters.h

\*/

Result = XIntc\_Initialize(IntcInstancePtr, INTC\_DEVICE\_ID);

if (Result != XST\_SUCCESS) {

return Result;

}

#endif /\* TESTAPP\_GEN \*/

/\* Hook up interrupt service routine \*/

XIntc\_Connect(IntcInstancePtr, IntrId,

(Xil\_ExceptionHandler)GpioHandler, InstancePtr);

/\* Enable the interrupt vector at the interrupt controller \*/

XIntc\_Enable(IntcInstancePtr, IntrId);

#ifndef TESTAPP\_GEN

/\*

\* Start the interrupt controller such that interrupts are recognized

\* and handled by the processor

\*/

Result = XIntc\_Start(IntcInstancePtr, XIN\_REAL\_MODE);

if (Result != XST\_SUCCESS) {

return Result;

}

#endif /\* TESTAPP\_GEN \*/

/\*

\* Enable the GPIO channel interrupts so that push button can be

\* detected and enable interrupts for the GPIO device

\*/

XGpio\_InterruptEnable(InstancePtr, IntrMask);

XGpio\_InterruptGlobalEnable(InstancePtr);

/\*

\* Initialize the exception table and register the interrupt

\* controller handler with the exception table

\*/

Xil\_ExceptionInit();

Xil\_ExceptionRegisterHandler(XIL\_EXCEPTION\_ID\_INT,

(Xil\_ExceptionHandler)INTC\_HANDLER, IntcInstancePtr);

/\* Enable non-critical exceptions \*/

Xil\_ExceptionEnable();

return XST\_SUCCESS;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* This is the interrupt handler routine for the GPIO for this example.

\*

\* @param CallbackRef is the Callback reference for the handler.

\*

\* @return None.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void GpioHandler(void \*CallbackRef)

{

XGpio \*GpioPtr = (XGpio \*)CallbackRef;

char input = Xil\_In8(XPAR\_GPIO\_2\_BASEADDR);

if (!debouncer){

if (input == 1){ // center button: reset

printf("center\r\n");

reset = 0xFF;

enabled = 0;

led\_pos = 0;

led\_output\_addr = 0;

}else if(input == 2){ // top button: nothing

printf("top\r\n");

// do nothing

}else if(input == 4){ // left button: forward/backward

printf("left\r\n");

fwd\_bwd = ~fwd\_bwd;

}else if(input == 8){ // right button: enable/disable

printf("right\r\n");

enabled = ~enabled;

}else if(input == 16){ // bottom button: nothing

printf("bottom\r\n");

// do nothing

}else if(input == 0){ // released all buttons: release reset

printf("release\r\n");

if (reset == 0xFF){

reset = 0;

enabled = 0xFF;

fwd\_bwd = 0xFF;

led\_counter = 0;

}

}else{

printf("ERROR: INVALID BUTTON INPUT!\r\n");

}

debouncer = 3000;

}

/\* Clear the Interrupt \*/

XGpio\_InterruptClear(GpioPtr, GlobalIntrMask);

}

1. xgpio.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* Copyright (C) 2002 - 2015 Xilinx, Inc. All rights reserved.

\*

\* Permission is hereby granted, free of charge, to any person obtaining a copy

\* of this software and associated documentation files (the "Software"), to deal

\* in the Software without restriction, including without limitation the rights

\* to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

\* copies of the Software, and to permit persons to whom the Software is

\* furnished to do so, subject to the following conditions:

\*

\* The above copyright notice and this permission notice shall be included in

\* all copies or substantial portions of the Software.

\*

\* Use of the Software is limited solely to applications:

\* (a) running on a Xilinx device, or

\* (b) that interact with a Xilinx device through a bus or interconnect.

\*

\* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

\* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

\* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL

\* XILINX BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,

\* WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF

\* OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

\* SOFTWARE.

\*

\* Except as contained in this notice, the name of the Xilinx shall not be used

\* in advertising or otherwise to promote the sale, use or other dealings in

\* this Software without prior written authorization from Xilinx.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* @file xgpio.c

\* @addtogroup gpio\_v4\_2

\* @{

\*

\* The implementation of the XGpio driver's basic functionality. See xgpio.h

\* for more information about the driver.

\*

\* @note

\*

\* None

\*

\* <pre>

\* MODIFICATION HISTORY:

\*

\* Ver Who Date Changes

\* ----- ---- -------- -----------------------------------------------

\* 1.00a rmm 02/04/02 First release

\* 2.00a jhl 12/16/02 Update for dual channel and interrupt support

\* 2.01a jvb 12/13/05 Changed Initialize() into CfgInitialize(), and made

\* CfgInitialize() take a pointer to a config structure

\* instead of a device id. Moved Initialize() into

\* xgpio\_sinit.c, and had Initialize() call CfgInitialize()

\* after it retrieved the config structure using the device

\* id. Removed include of xparameters.h along with any

\* dependencies on xparameters.h and the \_g.c config table.

\* 2.11a mta 03/21/07 Updated to new coding style, added GetDataDirection

\* 2.12a sv 11/21/07 Updated driver to support access through DCR bus

\* 3.00a sv 11/21/09 Updated to use HAL Processor APIs. Renamed the

\* macros to remove \_m from the name.

\* 4.1 lks 11/18/15 Clean up of the comments in the code and

\* removed support for DCR bridge

\* 4.2 sk 08/16/16 Used UINTPTR instead of u32 for Baseaddress as part of

\* adding 64 bit support. CR# 867425.

\* Changed the prototype of XGpio\_CfgInitialize API.

\* </pre>

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Include Files \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "xgpio.h"

#include "xstatus.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Constant Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Type Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Macros (Inline Functions) Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Variable Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Function Prototypes \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Initialize the XGpio instance provided by the caller based on the

\* given configuration data.

\*

\* Nothing is done except to initialize the InstancePtr.

\*

\* @param InstancePtr is a pointer to an XGpio instance. The memory the

\* pointer references must be pre-allocated by the caller. Further

\* calls to manipulate the driver through the XGpio API must be

\* made with this pointer.

\* @param Config is a reference to a structure containing information

\* about a specific GPIO device. This function initializes an

\* InstancePtr object for a specific device specified by the

\* contents of Config. This function can initialize multiple

\* instance objects with the use of multiple calls giving different

\* Config information on each call.

\* @param EffectiveAddr is the device base address in the virtual memory

\* address space. The caller is responsible for keeping the address

\* mapping from EffectiveAddr to the device physical base address

\* unchanged once this function is invoked. Unexpected errors may

\* occur if the address mapping changes after this function is

\* called. If address translation is not used, use

\* Config->BaseAddress for this parameters, passing the physical

\* address instead.

\*

\* @return

\* - XST\_SUCCESS if the initialization is successfull.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int XGpio\_CfgInitialize(XGpio \* InstancePtr, XGpio\_Config \* Config,

UINTPTR EffectiveAddr)

{

/\* Assert arguments \*/

Xil\_AssertNonvoid(InstancePtr != NULL);

/\* Set some default values. \*/

InstancePtr->BaseAddress = EffectiveAddr;

InstancePtr->InterruptPresent = Config->InterruptPresent;

InstancePtr->IsDual = Config->IsDual;

/\*

\* Indicate the instance is now ready to use, initialized without error

\*/

InstancePtr->IsReady = XIL\_COMPONENT\_IS\_READY;

return (XST\_SUCCESS);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Set the input/output direction of all discrete signals for the specified

\* GPIO channel.

\*

\* @param InstancePtr is a pointer to an XGpio instance to be worked on.

\* @param Channel contains the channel of the GPIO (1 or 2) to operate on.

\* @param DirectionMask is a bitmask specifying which discretes are input

\* and which are output. Bits set to 0 are output and bits set to 1

\* are input.

\*

\* @return None.

\*

\* @note The hardware must be built for dual channels if this function

\* is used with any channel other than 1. If it is not, this

\* function will assert.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void XGpio\_SetDataDirection(XGpio \*InstancePtr, unsigned Channel,

u32 DirectionMask)

{

Xil\_AssertVoid(InstancePtr != NULL);

Xil\_AssertVoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

Xil\_AssertVoid((Channel == 1) ||

((Channel == 2) && (InstancePtr->IsDual == TRUE)));

XGpio\_WriteReg(InstancePtr->BaseAddress,

((Channel - 1) \* XGPIO\_CHAN\_OFFSET) + XGPIO\_TRI\_OFFSET,

DirectionMask);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Get the input/output direction of all discrete signals for the specified

\* GPIO channel.

\*

\* @param InstancePtr is a pointer to an XGpio instance to be worked on.

\* @param Channel contains the channel of the GPIO (1 or 2) to operate on.

\*

\* @return Bitmask specifying which discretes are input and

\* which are output. Bits set to 0 are output and bits set to 1 are

\* input.

\*

\* @note

\*

\* The hardware must be built for dual channels if this function is used

\* with any channel other than 1. If it is not, this function will assert.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

u32 XGpio\_GetDataDirection(XGpio \*InstancePtr, unsigned Channel)

{

Xil\_AssertNonvoid(InstancePtr != NULL);

Xil\_AssertNonvoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

Xil\_AssertNonvoid((Channel == 1) ||

((Channel == 2) &&

(InstancePtr->IsDual == TRUE)));

return XGpio\_ReadReg(InstancePtr->BaseAddress,

((Channel - 1) \* XGPIO\_CHAN\_OFFSET) + XGPIO\_TRI\_OFFSET);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Read state of discretes for the specified GPIO channnel.

\*

\* @param InstancePtr is a pointer to an XGpio instance to be worked on.

\* @param Channel contains the channel of the GPIO (1 or 2) to operate on.

\*

\* @return Current copy of the discretes register.

\*

\* @note The hardware must be built for dual channels if this function

\* is used with any channel other than 1. If it is not, this

\* function will assert.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

u32 XGpio\_DiscreteRead(XGpio \* InstancePtr, unsigned Channel)

{

Xil\_AssertNonvoid(InstancePtr != NULL);

Xil\_AssertNonvoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

Xil\_AssertNonvoid((Channel == 1) ||

((Channel == 2) && (InstancePtr->IsDual == TRUE)));

return XGpio\_ReadReg(InstancePtr->BaseAddress,

((Channel - 1) \* XGPIO\_CHAN\_OFFSET) +

XGPIO\_DATA\_OFFSET);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Write to discretes register for the specified GPIO channel.

\*

\* @param InstancePtr is a pointer to an XGpio instance to be worked on.

\* @param Channel contains the channel of the GPIO (1 or 2) to operate on.

\* @param Data is the value to be written to the discretes register.

\*

\* @return None.

\*

\* @note The hardware must be built for dual channels if this function

\* is used with any channel other than 1. If it is not, this

\* function will assert. See also XGpio\_DiscreteSet() and

\* XGpio\_DiscreteClear().

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void XGpio\_DiscreteWrite(XGpio \* InstancePtr, unsigned Channel, u32 Data)

{

Xil\_AssertVoid(InstancePtr != NULL);

Xil\_AssertVoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

Xil\_AssertVoid((Channel == 1) ||

((Channel == 2) && (InstancePtr->IsDual == TRUE)));

XGpio\_WriteReg(InstancePtr->BaseAddress,

((Channel - 1) \* XGPIO\_CHAN\_OFFSET) + XGPIO\_DATA\_OFFSET,

Data);

}

/\*\* @} \*/

1. xintc.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* Copyright (C) 2002 - 2014 Xilinx, Inc. All rights reserved.

\*

\* Permission is hereby granted, free of charge, to any person obtaining a copy

\* of this software and associated documentation files (the "Software"), to deal

\* in the Software without restriction, including without limitation the rights

\* to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

\* copies of the Software, and to permit persons to whom the Software is

\* furnished to do so, subject to the following conditions:

\*

\* The above copyright notice and this permission notice shall be included in

\* all copies or substantial portions of the Software.

\*

\* Use of the Software is limited solely to applications:

\* (a) running on a Xilinx device, or

\* (b) that interact with a Xilinx device through a bus or interconnect.

\*

\* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

\* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

\* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

\* XILINX CONSORTIUM BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,

\* WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF

\* OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

\* SOFTWARE.

\*

\* Except as contained in this notice, the name of the Xilinx shall not be used

\* in advertising or otherwise to promote the sale, use or other dealings in

\* this Software without prior written authorization from Xilinx.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* @file xintc.c

\* @addtogroup intc\_v3\_4

\* @{

\*

\* Contains required functions for the XIntc driver for the Xilinx Interrupt

\* Controller. See xintc.h for a detailed description of the driver.

\*

\* <pre>

\* MODIFICATION HISTORY:

\*

\* Ver Who Date Changes

\* ----- ---- -------- --------------------------------------------------------

\* 1.00a ecm 08/16/01 First release

\* 1.00b jhl 02/21/02 Repartitioned the driver for smaller files

\* 1.00b jhl 04/24/02 Made LookupConfig global and compressed ack before table

\* in the configuration into a bit mask

\* 1.00c rpm 10/17/03 New release. Support the static vector table created

\* in the xintc\_g.c configuration table.

\* 1.00c rpm 04/23/04 Removed check in XIntc\_Connect for a previously connected

\* handler. Always overwrite the vector table handler with

\* the handler provided as an argument.

\* 1.10c mta 03/21/07 Updated to new coding style

\* 1.11a sv 11/21/07 Updated driver to support access through a DCR bridge

\* 2.00a ktn 10/20/09 Updated to use HAL Processor APIs.

\* 2.04a bss 01/13/12 Added XIntc\_ConnectFastHandler API for Fast Interrupt

\* and XIntc\_SetNormalIntrMode for setting to normal

\* interrupt mode.

\* 2.05a bss 08/16/12 Updated to support relocatable vectors in Microblaze,

\* updated XIntc\_SetNormalIntrMode to use IntVectorAddr

\* which is the interrupt vector address

\* 2.06a bss 01/28/13 To support Cascade mode:

\* Modified XIntc\_Initialize,XIntc\_Start,XIntc\_Connect

\* XIntc\_Disconnect,XIntc\_Enable,XIntc\_Disable,

\* XIntc\_Acknowledge,XIntc\_ConnectFastHandler and

\* XIntc\_SetNormalIntrMode APIs.

\* Added XIntc\_InitializeSlaves API.

\* 3.0 bss 01/28/13 Modified to initialize IVAR register with

\* XPAR\_MICROBLAZE\_BASE\_VECTORS + 0x10 to fix

\* CR#765931

\*

\* </pre>

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Include Files \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "xil\_types.h"

#include "xil\_assert.h"

#include "xintc.h"

#include "xintc\_l.h"

#include "xintc\_i.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Constant Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Type Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Macros (Inline Functions) Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Variable Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\* Array of masks associated with the bit position, improves performance

\* in the ISR and acknowledge functions, this table is shared between all

\* instances of the driver. XIN\_CONTROLLER\_MAX\_INTRS is the maximum number of

\* sources of Interrupt controller

\*/

u32 XIntc\_BitPosMask[XIN\_CONTROLLER\_MAX\_INTRS];

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Function Prototypes \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static void StubHandler(void \*CallBackRef);

static void XIntc\_InitializeSlaves(XIntc \* InstancePtr);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Initialize a specific interrupt controller instance/driver. The

\* initialization entails:

\*

\* - Initialize fields of the XIntc structure

\* - Initial vector table with stub function calls

\* - All interrupt sources are disabled

\* - Interrupt output is disabled

\*

\* @param InstancePtr is a pointer to the XIntc instance to be worked on.

\* @param DeviceId is the unique id of the device controlled by this XIntc

\* instance. Passing in a device id associates the generic XIntc

\* instance to a specific device, as chosen by the caller or

\* application developer.

\*

\* @return

\* - XST\_SUCCESS if initialization was successful

\* - XST\_DEVICE\_IS\_STARTED if the device has already been started

\* - XST\_DEVICE\_NOT\_FOUND if device configuration information was

\* not found for a device with the supplied device ID.

\*

\* @note In Cascade mode this function calls XIntc\_InitializeSlaves to

\* initialiaze Slave Interrupt controllers.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int XIntc\_Initialize(XIntc \* InstancePtr, u16 DeviceId)

{

u8 Id;

XIntc\_Config \*CfgPtr;

u32 NextBitMask = 1;

Xil\_AssertNonvoid(InstancePtr != NULL);

/\*

\* If the device is started, disallow the initialize and return a status

\* indicating it is started. This allows the user to stop the device

\* and reinitialize, but prevents a user from inadvertently initializing

\*/

if (InstancePtr->IsStarted == XIL\_COMPONENT\_IS\_STARTED) {

return XST\_DEVICE\_IS\_STARTED;

}

/\*

\* Lookup the device configuration in the CROM table. Use this

\* configuration info down below when initializing this component.

\*/

CfgPtr = XIntc\_LookupConfig(DeviceId);

if (CfgPtr == NULL) {

return XST\_DEVICE\_NOT\_FOUND;

}

/\*

\* Set some default values

\*/

InstancePtr->IsReady = 0;

InstancePtr->IsStarted = 0; /\* not started \*/

InstancePtr->CfgPtr = CfgPtr;

InstancePtr->CfgPtr->Options = XIN\_SVC\_SGL\_ISR\_OPTION;

InstancePtr->CfgPtr->IntcType = CfgPtr->IntcType;

/\*

\* Save the base address pointer such that the registers of the

\* interrupt can be accessed

\*/

#if (XPAR\_XINTC\_USE\_DCR\_BRIDGE != 0)

InstancePtr->BaseAddress = ((CfgPtr->BaseAddress >> 2)) & 0xFFF;

#else

InstancePtr->BaseAddress = CfgPtr->BaseAddress;

#endif

/\*

\* Initialize all the data needed to perform interrupt processing for

\* each interrupt ID up to the maximum used

\*/

for (Id = 0; Id < CfgPtr->NumberofIntrs; Id++) {

/\*

\* Initalize the handler to point to a stub to handle an

\* interrupt which has not been connected to a handler. Only

\* initialize it if the handler is 0 or XNullHandler, which

\* means it was not initialized statically by the tools/user.

\* Set the callback reference to this instance so that

\* unhandled interrupts can be tracked.

\*/

if ((InstancePtr->CfgPtr->HandlerTable[Id].Handler == 0) ||

(InstancePtr->CfgPtr->HandlerTable[Id].Handler ==

XNullHandler)) {

InstancePtr->CfgPtr->HandlerTable[Id].Handler =

StubHandler;

}

InstancePtr->CfgPtr->HandlerTable[Id].CallBackRef = InstancePtr;

/\*

\* Initialize the bit position mask table such that bit

\* positions are lookups only for each interrupt id, with 0

\* being a special case

\* (XIntc\_BitPosMask[] = { 1, 2, 4, 8, ... })

\*/

XIntc\_BitPosMask[Id] = NextBitMask;

NextBitMask \*= 2;

}

/\*

\* Disable IRQ output signal

\* Disable all interrupt sources

\* Acknowledge all sources

\*/

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_MER\_OFFSET, 0);

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_IER\_OFFSET, 0);

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_IAR\_OFFSET, 0xFFFFFFFF);

/\*

\* If the fast Interrupt mode is enabled then set all the

\* interrupts as normal mode.

\*/

if(InstancePtr->CfgPtr->FastIntr == TRUE) {

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_IMR\_OFFSET, 0);

#ifdef XPAR\_MICROBLAZE\_BASE\_VECTORS

for (Id = 0; Id < 32 ; Id++)

{

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_IVAR\_OFFSET

+ (Id \* 4), XPAR\_MICROBLAZE\_BASE\_VECTORS

+ 0x10);

}

#else

for (Id = 0; Id < 32 ; Id++)

{

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_IVAR\_OFFSET

+ (Id \* 4), 0x10);

}

#endif

}

/\* Initialize slaves in Cascade mode\*/

if (InstancePtr->CfgPtr->IntcType != XIN\_INTC\_NOCASCADE) {

XIntc\_InitializeSlaves(InstancePtr);

}

/\*

\* Indicate the instance is now ready to use, successfully initialized

\*/

InstancePtr->IsReady = XIL\_COMPONENT\_IS\_READY;

return XST\_SUCCESS;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Starts the interrupt controller by enabling the output from the controller

\* to the processor. Interrupts may be generated by the interrupt controller

\* after this function is called.

\*

\* It is necessary for the caller to connect the interrupt handler of this

\* component to the proper interrupt source. This function also starts Slave

\* controllers in Cascade mode.

\*

\* @param InstancePtr is a pointer to the XIntc instance to be worked on.

\* @param Mode determines if software is allowed to simulate interrupts or

\* real interrupts are allowed to occur. Note that these modes are

\* mutually exclusive. The interrupt controller hardware resets in

\* a mode that allows software to simulate interrupts until this

\* mode is exited. It cannot be reentered once it has been exited.

\*

\* One of the following values should be used for the mode.

\* - XIN\_SIMULATION\_MODE enables simulation of interrupts only

\* - XIN\_REAL\_MODE enables hardware interrupts only

\*

\* @return

\* - XST\_SUCCESS if the device was started successfully

\* - XST\_FAILURE if simulation mode was specified and it could not

\* be set because real mode has already been entered.

\*

\* @note Must be called after XIntc initialization is completed.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int XIntc\_Start(XIntc \* InstancePtr, u8 Mode)

{

u32 MasterEnable = XIN\_INT\_MASTER\_ENABLE\_MASK;

XIntc\_Config \*CfgPtr;

int Index;

/\*

\* Assert the arguments

\*/

Xil\_AssertNonvoid(InstancePtr != NULL);

Xil\_AssertNonvoid((Mode == XIN\_SIMULATION\_MODE) ||

(Mode == XIN\_REAL\_MODE))

Xil\_AssertNonvoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

/\*

\* Check for simulation mode

\*/

if (Mode == XIN\_SIMULATION\_MODE) {

if (MasterEnable & XIN\_INT\_HARDWARE\_ENABLE\_MASK) {

return XST\_FAILURE;

}

}

else {

MasterEnable |= XIN\_INT\_HARDWARE\_ENABLE\_MASK;

}

/\*

\* Indicate the instance is ready to be used and is started before we

\* enable the device.

\*/

InstancePtr->IsStarted = XIL\_COMPONENT\_IS\_STARTED;

/\* Start the Slaves for Cascade Mode \*/

if (InstancePtr->CfgPtr->IntcType != XIN\_INTC\_NOCASCADE) {

for (Index = 1; Index <= XPAR\_XINTC\_NUM\_INSTANCES - 1; Index++)

{

CfgPtr = XIntc\_LookupConfig(Index);

XIntc\_Out32(CfgPtr->BaseAddress + XIN\_MER\_OFFSET,

MasterEnable);

}

}

/\* Start the master \*/

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_MER\_OFFSET, MasterEnable);

return XST\_SUCCESS;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Stops the interrupt controller by disabling the output from the controller

\* so that no interrupts will be caused by the interrupt controller.

\*

\* @param InstancePtr is a pointer to the XIntc instance to be worked on.

\*

\* @return None.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void XIntc\_Stop(XIntc \* InstancePtr)

{

/\*

\* Assert the arguments

\*/

Xil\_AssertVoid(InstancePtr != NULL);

Xil\_AssertVoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

/\*

\* Stop all interrupts from occurring thru the interrupt controller by

\* disabling all interrupts in the MER register

\*/

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_MER\_OFFSET, 0);

InstancePtr->IsStarted = 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Makes the connection between the Id of the interrupt source and the

\* associated handler that is to run when the interrupt is recognized. The

\* argument provided in this call as the Callbackref is used as the argument

\* for the handler when it is called. In Cascade mode, connects handler to

\* Slave controller handler table depending on the interrupt Id.

\*

\* @param InstancePtr is a pointer to the XIntc instance to be worked on.

\* @param Id contains the ID of the interrupt source and should be in the

\* range of 0 to XPAR\_INTC\_MAX\_NUM\_INTR\_INPUTS - 1 with 0 being

\* the highest priority interrupt.

\* @param Handler to the handler for that interrupt.

\* @param CallBackRef is the callback reference, usually the instance

\* pointer of the connecting driver.

\*

\* @return

\*

\* - XST\_SUCCESS if the handler was connected correctly.

\*

\* @note

\*

\* WARNING: The handler provided as an argument will overwrite any handler

\* that was previously connected.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int XIntc\_Connect(XIntc \* InstancePtr, u8 Id,

XInterruptHandler Handler, void \*CallBackRef)

{

XIntc\_Config \*CfgPtr;

/\*

\* Assert the arguments

\*/

Xil\_AssertNonvoid(InstancePtr != NULL);

Xil\_AssertNonvoid(Id < XPAR\_INTC\_MAX\_NUM\_INTR\_INPUTS);

Xil\_AssertNonvoid(Handler != NULL);

Xil\_AssertNonvoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

/\* Connect Handlers for Slave controllers in Cascade Mode \*/

if (Id > 31) {

CfgPtr = XIntc\_LookupConfig(Id/32);

CfgPtr->HandlerTable[Id%32].Handler = Handler;

CfgPtr->HandlerTable[Id%32].CallBackRef = CallBackRef;

}

/\* Connect Handlers for Master/primary controller \*/

else {

/\*

\* The Id is used as an index into the table to select the

\* proper handler

\*/

InstancePtr->CfgPtr->HandlerTable[Id].Handler = Handler;

InstancePtr->CfgPtr->HandlerTable[Id].CallBackRef =

CallBackRef;

}

return XST\_SUCCESS;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Updates the interrupt table with the Null Handler and NULL arguments at the

\* location pointed at by the Id. This effectively disconnects that interrupt

\* source from any handler. The interrupt is disabled also. In Cascade mode,

\* disconnects handler from Slave controller handler table depending on the

\* interrupt Id.

\*

\* @param InstancePtr is a pointer to the XIntc instance to be worked on.

\* @param Id contains the ID of the interrupt source and should be in the

\* range of 0 to XPAR\_INTC\_MAX\_NUM\_INTR\_INPUTS - 1 with 0 being

\* the highest priority interrupt.

\*

\* @return None.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void XIntc\_Disconnect(XIntc \* InstancePtr, u8 Id)

{

u32 CurrentIER;

u32 Mask;

XIntc\_Config \*CfgPtr;

/\*

\* Assert the arguments

\*/

Xil\_AssertVoid(InstancePtr != NULL);

Xil\_AssertVoid(Id < XPAR\_INTC\_MAX\_NUM\_INTR\_INPUTS);

Xil\_AssertVoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

/\*

\* Disable the interrupt such that it won't occur while disconnecting

\* the handler, only disable the specified interrupt id without

\* modifying the other interrupt ids

\*/

/\* Disconnect Handlers for Slave controllers in Cascade Mode \*/

if (Id > 31) {

CfgPtr = XIntc\_LookupConfig(Id/32);

CurrentIER = XIntc\_In32(CfgPtr->BaseAddress + XIN\_IER\_OFFSET);

/\* Convert from integer id to bit mask \*/

Mask = XIntc\_BitPosMask[(Id%32)];

XIntc\_Out32(CfgPtr->BaseAddress + XIN\_IER\_OFFSET,

(CurrentIER & ~Mask));

/\*

\* Disconnect the handler and connect a stub, the callback

\* reference must be set to this instance to allow unhandled

\* interrupts to be tracked

\*/

CfgPtr->HandlerTable[Id%32].Handler = StubHandler;

CfgPtr->HandlerTable[Id%32].CallBackRef = InstancePtr;

}

/\* Disconnect Handlers for Master/primary controller \*/

else {

CurrentIER = XIntc\_In32(InstancePtr->BaseAddress +

XIN\_IER\_OFFSET);

/\* Convert from integer id to bit mask \*/

Mask = XIntc\_BitPosMask[Id];

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_IER\_OFFSET,

(CurrentIER & ~Mask));

InstancePtr->CfgPtr->HandlerTable[Id%32].Handler =

StubHandler;

InstancePtr->CfgPtr->HandlerTable[Id%32].CallBackRef =

InstancePtr;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Enables the interrupt source provided as the argument Id. Any pending

\* interrupt condition for the specified Id will occur after this function is

\* called. In Cascade mode, enables corresponding interrupt of Slave controllers

\* depending on the Id.

\*

\* @param InstancePtr is a pointer to the XIntc instance to be worked on.

\* @param Id contains the ID of the interrupt source and should be in the

\* range of 0 to XPAR\_INTC\_MAX\_NUM\_INTR\_INPUTS - 1 with 0 being

\* the highest priority interrupt.

\*

\* @return None.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void XIntc\_Enable(XIntc \* InstancePtr, u8 Id)

{

u32 CurrentIER;

u32 Mask;

XIntc\_Config \*CfgPtr;

/\*

\* Assert the arguments

\*/

Xil\_AssertVoid(InstancePtr != NULL);

Xil\_AssertVoid(Id < XPAR\_INTC\_MAX\_NUM\_INTR\_INPUTS);

Xil\_AssertVoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

if (Id > 31) {

/\* Enable user required Id in Slave controller \*/

CfgPtr = XIntc\_LookupConfig(Id/32);

CurrentIER = XIntc\_In32(CfgPtr->BaseAddress + XIN\_IER\_OFFSET);

/\* Convert from integer id to bit mask \*/

Mask = XIntc\_BitPosMask[(Id%32)];

XIntc\_Out32(CfgPtr->BaseAddress + XIN\_IER\_OFFSET,

(CurrentIER | Mask));

}

else {

/\*

\* The Id is used to create the appropriate mask for the

\* desired bit position.

\*/

Mask = XIntc\_BitPosMask[Id];

/\*

\* Enable the selected interrupt source by reading the

\* interrupt enable register and then modifying only the

\* specified interrupt id enable

\*/

CurrentIER = XIntc\_In32(InstancePtr->BaseAddress +

XIN\_IER\_OFFSET);

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_IER\_OFFSET,

(CurrentIER | Mask));

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Disables the interrupt source provided as the argument Id such that the

\* interrupt controller will not cause interrupts for the specified Id. The

\* interrupt controller will continue to hold an interrupt condition for the

\* Id, but will not cause an interrupt.In Cascade mode, disables corresponding

\* interrupt of Slave controllers depending on the Id.

\*

\* @param InstancePtr is a pointer to the XIntc instance to be worked on.

\* @param Id contains the ID of the interrupt source and should be in the

\* range of 0 to XPAR\_INTC\_MAX\_NUM\_INTR\_INPUTS - 1 with 0 being the

\* highest priority interrupt.

\*

\* @return None.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void XIntc\_Disable(XIntc \* InstancePtr, u8 Id)

{

u32 CurrentIER;

u32 Mask;

XIntc\_Config \*CfgPtr;

/\*

\* Assert the arguments

\*/

Xil\_AssertVoid(InstancePtr != NULL);

Xil\_AssertVoid(Id < XPAR\_INTC\_MAX\_NUM\_INTR\_INPUTS);

Xil\_AssertVoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

if (Id > 31) {

/\* Enable user required Id in Slave controller \*/

CfgPtr = XIntc\_LookupConfig(Id/32);

CurrentIER = XIntc\_In32(CfgPtr->BaseAddress + XIN\_IER\_OFFSET);

/\* Convert from integer id to bit mask \*/

Mask = XIntc\_BitPosMask[(Id%32)];

XIntc\_Out32(CfgPtr->BaseAddress + XIN\_IER\_OFFSET,

(CurrentIER & ~Mask));

} else {

/\*

\* The Id is used to create the appropriate mask for the

\* desired bit position. Id currently limited to 0 - 31

\*/

Mask = XIntc\_BitPosMask[Id];

/\*

\* Disable the selected interrupt source by reading the

\* interrupt enable register and then modifying only the

\* specified interrupt id

\*/

CurrentIER = XIntc\_In32(InstancePtr->BaseAddress +

XIN\_IER\_OFFSET);

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_IER\_OFFSET,

(CurrentIER & ~Mask));

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Acknowledges the interrupt source provided as the argument Id. When the

\* interrupt is acknowledged, it causes the interrupt controller to clear its

\* interrupt condition.In Cascade mode, acknowledges corresponding interrupt

\* source of Slave controllers depending on the Id.

\*

\* @param InstancePtr is a pointer to the XIntc instance to be worked on.

\* @param Id contains the ID of the interrupt source and should be in the

\* range of 0 to XPAR\_INTC\_MAX\_NUM\_INTR\_INPUTS - 1 with 0 being

\* the highest priority interrupt.

\*

\* @return None.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void XIntc\_Acknowledge(XIntc \* InstancePtr, u8 Id)

{

u32 Mask;

XIntc\_Config \*CfgPtr;

/\*

\* Assert the arguments

\*/

Xil\_AssertVoid(InstancePtr != NULL);

Xil\_AssertVoid(Id < XPAR\_INTC\_MAX\_NUM\_INTR\_INPUTS);

Xil\_AssertVoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

if (Id > 31) {

/\* Enable user required Id in Slave controller \*/

CfgPtr = XIntc\_LookupConfig(Id/32);

/\* Convert from integer id to bit mask \*/

Mask = XIntc\_BitPosMask[(Id%32)];

XIntc\_Out32(CfgPtr->BaseAddress + XIN\_IAR\_OFFSET, Mask);

} else {

/\*

\* The Id is used to create the appropriate mask for the

\* desired bit position.

\*/

Mask = XIntc\_BitPosMask[Id];

/\*

\* Acknowledge the selected interrupt source, no read of the

\* acknowledge register is necessary since only the bits set

\* in the mask will be affected by the write

\*/

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_IAR\_OFFSET, Mask);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* A stub for the asynchronous callback. The stub is here in case the upper

\* layers forget to set the handler.

\*

\* @param CallBackRef is a pointer to the upper layer callback reference

\*

\* @return None.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static void StubHandler(void \*CallBackRef)

{

/\*

\* Verify that the inputs are valid

\*/

Xil\_AssertVoid(CallBackRef != NULL);

/\*

\* Indicate another unhandled interrupt for stats

\*/

((XIntc \*) CallBackRef)->UnhandledInterrupts++;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Looks up the device configuration based on the unique device ID. A table

\* contains the configuration info for each device in the system.

\*

\* @param DeviceId is the unique identifier for a device.

\*

\* @return A pointer to the XIntc configuration structure for the specified

\* device, or NULL if the device was not found.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

XIntc\_Config \*XIntc\_LookupConfig(u16 DeviceId)

{

XIntc\_Config \*CfgPtr = NULL;

int Index;

for (Index = 0; Index < XPAR\_XINTC\_NUM\_INSTANCES; Index++) {

if (XIntc\_ConfigTable[Index].DeviceId == DeviceId) {

CfgPtr = &XIntc\_ConfigTable[Index];

break;

}

}

return CfgPtr;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Makes the connection between the Id of the interrupt source and the

\* associated handler that is to run when the interrupt is recognized.In Cascade

\* mode, connects handler to corresponding Slave controller IVAR register

\* depending on the Id and sets all interrupt sources of the Slave controller as

\* fast interrupts.

\*

\* @param InstancePtr is a pointer to the XIntc instance to be worked on.

\* @param Id contains the ID of the interrupt source and should be in the

\* range of 0 to XPAR\_INTC\_MAX\_NUM\_INTR\_INPUTS - 1 with 0 being

\* the highest priority interrupt.

\* @param Handler to the handler for that interrupt.

\*

\* @return

\* - XST\_SUCCESS

\*

\* @note

\* Slave controllers in Cascade Mode should have all as Fast

\* interrupts or Normal interrupts, mixed interrupts are not

\* supported

\*

\* WARNING: The handler provided as an argument will overwrite any handler

\* that was previously connected.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int XIntc\_ConnectFastHandler(XIntc \*InstancePtr, u8 Id,

XFastInterruptHandler Handler)

{

u32 Imr;

u32 CurrentIER;

u32 Mask;

XIntc\_Config \*CfgPtr;

/\*

\* Assert the arguments

\*/

Xil\_AssertNonvoid(InstancePtr != NULL);

Xil\_AssertNonvoid(Id < XPAR\_INTC\_MAX\_NUM\_INTR\_INPUTS);

Xil\_AssertNonvoid(Handler != NULL);

Xil\_AssertNonvoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

Xil\_AssertNonvoid(InstancePtr->CfgPtr->FastIntr == TRUE);

if (Id > 31) {

/\* Enable user required Id in Slave controller \*/

CfgPtr = XIntc\_LookupConfig(Id/32);

if (CfgPtr->FastIntr != TRUE) {

/\*Fast interrupts of slave controller are not enabled\*/

return XST\_FAILURE;

}

/\* Get the Enabled Interrupts \*/

CurrentIER = XIntc\_In32(CfgPtr->BaseAddress + XIN\_IER\_OFFSET);

/\* Convert from integer id to bit mask \*/

Mask = XIntc\_BitPosMask[(Id%32)];

/\* Disable the Interrupt if it was enabled before calling

\* this function

\*/

if (CurrentIER & Mask) {

XIntc\_Disable(InstancePtr, Id);

}

XIntc\_Out32(CfgPtr->BaseAddress + XIN\_IVAR\_OFFSET +

((Id%32) \* 4), (u32) Handler);

/\* Slave controllers in Cascade Mode should have all as Fast

\* interrupts or Normal interrupts, mixed interrupts are not

\* supported

\*/

XIntc\_Out32(CfgPtr->BaseAddress + XIN\_IMR\_OFFSET, 0xFFFFFFFF);

/\* Enable the Interrupt if it was enabled before calling this

\* function

\*/

if (CurrentIER & Mask) {

XIntc\_Enable(InstancePtr, Id);

}

}

else {

/\* Get the Enabled Interrupts \*/

CurrentIER = XIntc\_In32(InstancePtr->BaseAddress +

XIN\_IER\_OFFSET);

/\* Convert from integer id to bit mask \*/

Mask = XIntc\_BitPosMask[Id];

/\* Disable the Interrupt if it was enabled before calling

\* this function

\*/

if (CurrentIER & Mask) {

XIntc\_Disable(InstancePtr, Id);

}

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_IVAR\_OFFSET +

(Id \* 4), (u32) Handler);

Imr = XIntc\_In32(InstancePtr->BaseAddress + XIN\_IMR\_OFFSET);

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_IMR\_OFFSET,

Imr | Mask);

/\* Enable the Interrupt if it was enabled before

\* calling this function

\*/

if (CurrentIER & Mask) {

XIntc\_Enable(InstancePtr, Id);

}

}

return XST\_SUCCESS;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Sets the normal interrupt mode for the specified interrupt in the Interrupt

\* Mode Register. In Cascade mode disconnects handler from corresponding Slave

\* controller IVAR register depending on the Id and sets all interrupt sources

\* of the Slave controller as normal interrupts.

\*

\* @param InstancePtr is a pointer to the XIntc instance to be worked on.

\* @param Id contains the ID of the interrupt source and should be in the

\* range of 0 to XPAR\_INTC\_MAX\_NUM\_INTR\_INPUTS - 1 with 0 being the

\* highest priority interrupt.

\*

\* @return None.

\*

\* @note

\* Slave controllers in Cascade Mode should have all as Fast

\* interrupts or Normal interrupts, mixed interrupts are not

\* supported

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void XIntc\_SetNormalIntrMode(XIntc \*InstancePtr, u8 Id)

{

u32 Imr;

u32 CurrentIER;

u32 Mask;

XIntc\_Config \*CfgPtr;

/\*

\* Assert the arguments

\*/

Xil\_AssertVoid(InstancePtr != NULL);

Xil\_AssertVoid(Id < XPAR\_INTC\_MAX\_NUM\_INTR\_INPUTS);

Xil\_AssertVoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

Xil\_AssertVoid(InstancePtr->CfgPtr->FastIntr == TRUE);

if (Id > 31) {

/\* Enable user required Id in Slave controller \*/

CfgPtr = XIntc\_LookupConfig(Id/32);

/\* Get the Enabled Interrupts \*/

CurrentIER = XIntc\_In32(CfgPtr->BaseAddress + XIN\_IER\_OFFSET);

/\* Convert from integer id to bit mask \*/

Mask = XIntc\_BitPosMask[(Id%32)];

/\* Disable the Interrupt if it was enabled before calling

\* this function

\*/

if (CurrentIER & Mask) {

XIntc\_Disable(InstancePtr, Id);

}

/\* Slave controllers in Cascade Mode should have all as Fast

\* interrupts or Normal interrupts, mixed interrupts are not

\* supported

\*/

XIntc\_Out32(CfgPtr->BaseAddress + XIN\_IMR\_OFFSET, 0x0);

#ifdef XPAR\_MICROBLAZE\_BASE\_VECTORS

for (Id = 0; Id < 32 ; Id++)

{

XIntc\_Out32(CfgPtr->BaseAddress + XIN\_IVAR\_OFFSET

+ (Id \* 4), XPAR\_MICROBLAZE\_BASE\_VECTORS

+ 0x10);

}

#else

for (Id = 0; Id < 32 ; Id++)

{

XIntc\_Out32(CfgPtr->BaseAddress + XIN\_IVAR\_OFFSET

+ (Id \* 4), 0x10);

}

#endif

/\* Enable the Interrupt if it was enabled before calling this

\* function

\*/

if (CurrentIER & Mask) {

XIntc\_Enable(InstancePtr, Id);

}

}

else {

/\* Get the Enabled Interrupts \*/

CurrentIER = XIntc\_In32(InstancePtr->BaseAddress + XIN\_IER\_OFFSET);

Mask = XIntc\_BitPosMask[Id];/\* Convert from integer id to bit mask \*/

/\* Disable the Interrupt if it was enabled before

\* calling this function

\*/

if (CurrentIER & Mask) {

XIntc\_Disable(InstancePtr, Id);

}

/\*

\* Disable the selected interrupt as Fast Interrupt by reading the

\* interrupt mode register and then modifying only the

\* specified interrupt id

\*/

Imr = XIntc\_In32(InstancePtr->BaseAddress + XIN\_IMR\_OFFSET);

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_IMR\_OFFSET,

Imr & ~Mask);

#ifdef XPAR\_MICROBLAZE\_BASE\_VECTORS

for (Id = 0; Id < 32 ; Id++)

{

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_IVAR\_OFFSET

+ (Id \* 4), XPAR\_MICROBLAZE\_BASE\_VECTORS

+ 0x10);

}

#else

for (Id = 0; Id < 32 ; Id++)

{

XIntc\_Out32(InstancePtr->BaseAddress + XIN\_IVAR\_OFFSET

+ (Id \* 4), 0x10);

}

#endif

/\* Enable the Interrupt if it was enabled before

\* calling this function

\*/

if (CurrentIER & Mask) {

XIntc\_Enable(InstancePtr, Id);

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Initializes Slave controllers in Cascade mode. The initialization entails:

\* - Initial vector table with stub function calls

\* - All interrupt sources are disabled for last controller.

\* - All interrupt sources are disabled except sources to 31 pin of

\* primary and secondary controllers

\* - Interrupt outputs are disabled

\*

\* @param InstancePtr is a pointer to the XIntc instance to be worked on.

\*

\* @return None

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static void XIntc\_InitializeSlaves(XIntc \* InstancePtr)

{

int Index;

u32 Mask;

XIntc\_Config \*CfgPtr;

int Id;

Mask = XIntc\_BitPosMask[31]; /\* Convert from integer id to bit mask \*/

/\* Enable interrupt id with 31 for Master

\* interrupt controller

\*/

XIntc\_Out32(InstancePtr->CfgPtr->BaseAddress + XIN\_IER\_OFFSET, Mask);

for (Index = 1; Index <= XPAR\_XINTC\_NUM\_INSTANCES - 1; Index++) {

CfgPtr = XIntc\_LookupConfig(Index);

XIntc\_Out32(CfgPtr->BaseAddress + XIN\_IAR\_OFFSET,

0xFFFFFFFF);

if (CfgPtr->IntcType != XIN\_INTC\_LAST) {

/\* Enable interrupt ids with 31 for secondary

\* interrupt controllers

\*/

XIntc\_Out32(CfgPtr->BaseAddress + XIN\_IER\_OFFSET,

Mask);

} else {

XIntc\_Out32(CfgPtr->BaseAddress + XIN\_IER\_OFFSET, 0x0);

}

/\* Disable Interrupt output \*/

XIntc\_Out32(CfgPtr->BaseAddress + XIN\_MER\_OFFSET, 0);

/\* Set all interrupts as normal mode if Fast Interrupts

\* are enabled

\*/

if(CfgPtr->FastIntr == TRUE) {

XIntc\_Out32(CfgPtr->BaseAddress + XIN\_IMR\_OFFSET, 0);

#ifdef XPAR\_MICROBLAZE\_BASE\_VECTORS

for (Id = 0; Id < 32 ; Id++)

{

XIntc\_Out32(CfgPtr->BaseAddress +

XIN\_IVAR\_OFFSET + (Id \* 4),

XPAR\_MICROBLAZE\_BASE\_VECTORS + 0x10);

}

#else

for (Id = 0; Id < 32 ; Id++)

{

XIntc\_Out32(CfgPtr->BaseAddress +

XIN\_IVAR\_OFFSET + (Id \* 4), 0x10);

}

#endif

}

/\*

\* Initialize all the data needed to perform interrupt

\* processing for each interrupt ID up to the maximum used

\*/

for (Id = 0; Id < CfgPtr->NumberofIntrs; Id++) {

/\*

\* Initalize the handler to point to a stub to handle an

\* interrupt which has not been connected to a handler.

\* Only initialize it if the handler is 0 or

\* XNullHandler, which means it was not initialized

\* statically by the tools/user.Set the callback

\* reference to this instance so that unhandled

\* interrupts can be tracked.

\*/

if ((CfgPtr->HandlerTable[Id].Handler == 0) ||

(CfgPtr->HandlerTable[Id].Handler ==

XNullHandler)) {

CfgPtr->HandlerTable[Id].Handler = StubHandler;

}

CfgPtr->HandlerTable[Id].CallBackRef = InstancePtr;

}

}

}

/\*\* @} \*/

1. xtmrctr.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* Copyright (C) 2002 - 2015 Xilinx, Inc. All rights reserved.

\*

\* Permission is hereby granted, free of charge, to any person obtaining a copy

\* of this software and associated documentation files (the "Software"), to deal

\* in the Software without restriction, including without limitation the rights

\* to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

\* copies of the Software, and to permit persons to whom the Software is

\* furnished to do so, subject to the following conditions:

\*

\* The above copyright notice and this permission notice shall be included in

\* all copies or substantial portions of the Software.

\*

\* Use of the Software is limited solely to applications:

\* (a) running on a Xilinx device, or

\* (b) that interact with a Xilinx device through a bus or interconnect.

\*

\* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

\* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

\* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL

\* XILINX BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,

\* WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF

\* OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

\* SOFTWARE.

\*

\* Except as contained in this notice, the name of the Xilinx shall not be used

\* in advertising or otherwise to promote the sale, use or other dealings in

\* this Software without prior written authorization from Xilinx.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* @file xtmrctr.c

\* @addtogroup tmrctr\_v4\_0

\* @{

\*

\* Contains required functions for the XTmrCtr driver.

\*

\* <pre>

\* MODIFICATION HISTORY:

\*

\* Ver Who Date Changes

\* ----- ---- -------- -----------------------------------------------

\* 1.00a ecm 08/16/01 First release

\* 1.00b jhl 02/21/02 Repartitioned the driver for smaller files

\* 1.10b mta 03/21/07 Updated to new coding style

\* 2.00a ktn 10/30/09 Updated to use HAL API's. \_m is removed from all the macro

\* definitions.

\* 2.05a adk 15/05/13 Fixed the CR:693066

\* Added the IsStartedTmrCtr0/IsStartedTmrCtr1 members to the

\* XTmrCtr instance structure.

\* The IsStartedTmrCtrX will be assigned XIL\_COMPONENT\_IS\_STARTED in

\* the XTmrCtr\_Start function.

\* The IsStartedTmrCtrX will be cleared in the XTmrCtr\_Stop function.

\* There will be no Initialization done in the

\* XTmrCtr\_Initialize if both the timers have already started and

\* the XST\_DEVICE\_IS\_STARTED Status is returned.

\* Removed the logic in the XTmrCtr\_Initialize function

\* which was checking the Register Value to know whether

\* a timer has started or not.

\* 4.0 als 09/30/15 Updated initialization API.

\* 4.1 sk 11/10/15 Used UINTPTR instead of u32 for Baseaddress CR# 867425.

\* Changed the prototype of XTmrCtr\_CfgInitialize API.

\* </pre>

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Include Files \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "xstatus.h"

#include "xparameters.h"

#include "xtmrctr.h"

#include "xtmrctr\_i.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Constant Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Type Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Macros (Inline Functions) Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Function Prototypes \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static void XTmrCtr\_StubCallback(void \*CallBackRef, u8 TmrCtrNumber);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Variable Definitions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* This function populates the timer counter's configuration structure and sets

\* some configurations defaults.

\*

\* @param InstancePtr is a pointer to the XTmrCtr instance.

\* @param ConfigPtr is a pointer to the configuration structure that will

\* be used to copy the settings from.

\* @param EffectiveAddr is the device base address in the virtual memory

\* space. If the address translation is not used, then the physical

\* address is passed.

\*

\* @return None.

\*

\* @note Unexpected errors may occur if the address mapping is changed

\* after this function is invoked.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void XTmrCtr\_CfgInitialize(XTmrCtr \*InstancePtr, XTmrCtr\_Config \*ConfigPtr,

UINTPTR EffectiveAddr)

{

/\* Verify arguments. \*/

Xil\_AssertVoid(InstancePtr != NULL);

Xil\_AssertVoid(ConfigPtr != NULL);

Xil\_AssertVoid(EffectiveAddr != 0x0);

InstancePtr->IsReady = 0;

InstancePtr->Config = \*ConfigPtr;

InstancePtr->Config.BaseAddress = EffectiveAddr;

InstancePtr->BaseAddress = EffectiveAddr;

InstancePtr->Handler = XTmrCtr\_StubCallback;

InstancePtr->CallBackRef = InstancePtr;

InstancePtr->Stats.Interrupts = 0;

InstancePtr->IsReady = XIL\_COMPONENT\_IS\_READY;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* (Re-)initialzes all timer counters which aren't started already.

\*

\* @param InstancePtr is a pointer to the XTmrCtr instance.

\*

\* @return

\* - XST\_SUCCESS if at least one timer counter is stopped.

\* - XST\_DEVICE\_IS\_STARTED otherwise.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int XTmrCtr\_InitHw(XTmrCtr \*InstancePtr)

{

int Status = XST\_DEVICE\_IS\_STARTED;

u8 TmrIndex;

u32 TmrCtrStarted[XTC\_DEVICE\_TIMER\_COUNT];

/\* Verify arguments. \*/

Xil\_AssertNonvoid(InstancePtr != NULL);

Xil\_AssertNonvoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

TmrCtrStarted[0] = InstancePtr->IsStartedTmrCtr0;

TmrCtrStarted[1] = InstancePtr->IsStartedTmrCtr1;

for (TmrIndex = 0; TmrIndex < XTC\_DEVICE\_TIMER\_COUNT; TmrIndex++) {

/\* Only initialize timers counters which aren't started. \*/

if (TmrCtrStarted[TmrIndex] == XIL\_COMPONENT\_IS\_STARTED) {

continue;

}

/\* Set the compare register to 0. \*/

XTmrCtr\_WriteReg(InstancePtr->BaseAddress, TmrIndex,

XTC\_TLR\_OFFSET, 0);

/\* Reset the timer and the interrupt. \*/

XTmrCtr\_WriteReg(InstancePtr->BaseAddress, TmrIndex,

XTC\_TCSR\_OFFSET,

XTC\_CSR\_INT\_OCCURED\_MASK | XTC\_CSR\_LOAD\_MASK);

/\* Release the reset. \*/

XTmrCtr\_WriteReg(InstancePtr->BaseAddress, TmrIndex,

XTC\_TCSR\_OFFSET, 0);

/\* Indicate that at least one timer is not running and has been

\* initialized. \*/

Status = XST\_SUCCESS;

}

return Status;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Initializes a specific timer/counter instance/driver. Initialize fields of

\* the XTmrCtr structure, then reset the timer/counter.If a timer is already

\* running then it is not initialized.

\*

\*

\* @param InstancePtr is a pointer to the XTmrCtr instance.

\* @param DeviceId is the unique id of the device controlled by this

\* XTmrCtr component. Passing in a device id associates the

\* generic XTmrCtr component to a specific device, as chosen by

\* the caller or application developer.

\*

\* @return

\* - XST\_SUCCESS if initialization was successful

\* - XST\_DEVICE\_IS\_STARTED if the device has already been started

\* - XST\_DEVICE\_NOT\_FOUND if the device doesn't exist

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int XTmrCtr\_Initialize(XTmrCtr \*InstancePtr, u16 DeviceId)

{

XTmrCtr\_Config \*ConfigPtr;

Xil\_AssertNonvoid(InstancePtr != NULL);

/\* In case all timer counters are already started, don't proceed with

\* re-initialization. \*/

if ((InstancePtr->IsStartedTmrCtr0 == XIL\_COMPONENT\_IS\_STARTED) &&

(InstancePtr->IsStartedTmrCtr1 == XIL\_COMPONENT\_IS\_STARTED)) {

return XST\_DEVICE\_IS\_STARTED;

}

/\* Retrieve configuration of timer counter core with matching ID. \*/

ConfigPtr = XTmrCtr\_LookupConfig(DeviceId);

if (!ConfigPtr) {

return XST\_DEVICE\_NOT\_FOUND;

}

XTmrCtr\_CfgInitialize(InstancePtr, ConfigPtr, ConfigPtr->BaseAddress);

return XTmrCtr\_InitHw(InstancePtr);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Starts the specified timer counter of the device such that it starts running.

\* The timer counter is reset before it is started and the reset value is

\* loaded into the timer counter.

\*

\* If interrupt mode is specified in the options, it is necessary for the caller

\* to connect the interrupt handler of the timer/counter to the interrupt source,

\* typically an interrupt controller, and enable the interrupt within the

\* interrupt controller.

\*

\* @param InstancePtr is a pointer to the XTmrCtr instance.

\* @param TmrCtrNumber is the timer counter of the device to operate on.

\* Each device may contain multiple timer counters. The timer

\* number is a zero based number with a range of

\* 0 - (XTC\_DEVICE\_TIMER\_COUNT - 1).

\*

\* @return None.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void XTmrCtr\_Start(XTmrCtr \* InstancePtr, u8 TmrCtrNumber)

{

u32 ControlStatusReg;

Xil\_AssertVoid(InstancePtr != NULL);

Xil\_AssertVoid(TmrCtrNumber < XTC\_DEVICE\_TIMER\_COUNT);

Xil\_AssertVoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

/\*

\* Read the current register contents such that only the necessary bits

\* of the register are modified in the following operations

\*/

ControlStatusReg = XTmrCtr\_ReadReg(InstancePtr->BaseAddress,

TmrCtrNumber, XTC\_TCSR\_OFFSET);

/\*

\* Reset the timer counter such that it reloads from the compare

\* register and the interrupt is cleared simultaneously, the interrupt

\* can only be cleared after reset such that the interrupt condition is

\* cleared

\*/

XTmrCtr\_WriteReg(InstancePtr->BaseAddress, TmrCtrNumber,

XTC\_TCSR\_OFFSET,

XTC\_CSR\_LOAD\_MASK);

/\*

\* Indicate that the timer is started before enabling it

\*/

if (TmrCtrNumber == 0) {

InstancePtr->IsStartedTmrCtr0 = XIL\_COMPONENT\_IS\_STARTED;

} else {

InstancePtr->IsStartedTmrCtr1 = XIL\_COMPONENT\_IS\_STARTED;

}

/\*

\* Remove the reset condition such that the timer counter starts running

\* with the value loaded from the compare register

\*/

XTmrCtr\_WriteReg(InstancePtr->BaseAddress, TmrCtrNumber,

XTC\_TCSR\_OFFSET,

ControlStatusReg | XTC\_CSR\_ENABLE\_TMR\_MASK);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Stops the timer counter by disabling it.

\*

\* It is the callers' responsibility to disconnect the interrupt handler of the

\* timer\_counter from the interrupt source, typically an interrupt controller,

\* and disable the interrupt within the interrupt controller.

\*

\* @param InstancePtr is a pointer to the XTmrCtr instance.

\* @param TmrCtrNumber is the timer counter of the device to operate on.

\* Each device may contain multiple timer counters. The timer

\* number is a zero based number with a range of

\* 0 - (XTC\_DEVICE\_TIMER\_COUNT - 1).

\*

\* @return None.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void XTmrCtr\_Stop(XTmrCtr \* InstancePtr, u8 TmrCtrNumber)

{

u32 ControlStatusReg;

Xil\_AssertVoid(InstancePtr != NULL);

Xil\_AssertVoid(TmrCtrNumber < XTC\_DEVICE\_TIMER\_COUNT);

Xil\_AssertVoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

/\*

\* Read the current register contents

\*/

ControlStatusReg = XTmrCtr\_ReadReg(InstancePtr->BaseAddress,

TmrCtrNumber, XTC\_TCSR\_OFFSET);

/\*

\* Disable the timer counter such that it's not running

\*/

ControlStatusReg &= ~(XTC\_CSR\_ENABLE\_TMR\_MASK);

/\*

\* Write out the updated value to the actual register.

\*/

XTmrCtr\_WriteReg(InstancePtr->BaseAddress, TmrCtrNumber,

XTC\_TCSR\_OFFSET, ControlStatusReg);

/\*

\* Indicate that the timer is stopped

\*/

if (TmrCtrNumber == 0) {

InstancePtr->IsStartedTmrCtr0 = 0;

} else {

InstancePtr->IsStartedTmrCtr1 = 0;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Get the current value of the specified timer counter. The timer counter

\* may be either incrementing or decrementing based upon the current mode of

\* operation.

\*

\* @param InstancePtr is a pointer to the XTmrCtr instance.

\* @param TmrCtrNumber is the timer counter of the device to operate on.

\* Each device may contain multiple timer counters. The timer

\* number is a zero based number with a range of

\* 0 - (XTC\_DEVICE\_TIMER\_COUNT - 1).

\*

\* @return The current value for the timer counter.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

u32 XTmrCtr\_GetValue(XTmrCtr \* InstancePtr, u8 TmrCtrNumber)

{

Xil\_AssertNonvoid(InstancePtr != NULL);

Xil\_AssertNonvoid(TmrCtrNumber < XTC\_DEVICE\_TIMER\_COUNT);

Xil\_AssertNonvoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

return XTmrCtr\_ReadReg(InstancePtr->BaseAddress,

TmrCtrNumber, XTC\_TCR\_OFFSET);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Set the reset value for the specified timer counter. This is the value

\* that is loaded into the timer counter when it is reset. This value is also

\* loaded when the timer counter is started.

\*

\* @param InstancePtr is a pointer to the XTmrCtr instance.

\* @param TmrCtrNumber is the timer counter of the device to operate on.

\* Each device may contain multiple timer counters. The timer

\* number is a zero based number with a range of

\* 0 - (XTC\_DEVICE\_TIMER\_COUNT - 1).

\* @param ResetValue contains the value to be used to reset the timer

\* counter.

\*

\* @return None.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void XTmrCtr\_SetResetValue(XTmrCtr \* InstancePtr, u8 TmrCtrNumber,

u32 ResetValue)

{

Xil\_AssertVoid(InstancePtr != NULL);

Xil\_AssertVoid(TmrCtrNumber < XTC\_DEVICE\_TIMER\_COUNT);

Xil\_AssertVoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

XTmrCtr\_WriteReg(InstancePtr->BaseAddress, TmrCtrNumber,

XTC\_TLR\_OFFSET, ResetValue);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Returns the timer counter value that was captured the last time the external

\* capture input was asserted.

\*

\* @param InstancePtr is a pointer to the XTmrCtr instance.

\* @param TmrCtrNumber is the timer counter of the device to operate on.

\* Each device may contain multiple timer counters. The timer

\* number is a zero based number with a range of

\* 0 - (XTC\_DEVICE\_TIMER\_COUNT - 1).

\*

\* @return The current capture value for the indicated timer counter.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

u32 XTmrCtr\_GetCaptureValue(XTmrCtr \* InstancePtr, u8 TmrCtrNumber)

{

Xil\_AssertNonvoid(InstancePtr != NULL);

Xil\_AssertNonvoid(TmrCtrNumber < XTC\_DEVICE\_TIMER\_COUNT);

Xil\_AssertNonvoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

return XTmrCtr\_ReadReg(InstancePtr->BaseAddress,

TmrCtrNumber, XTC\_TLR\_OFFSET);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Resets the specified timer counter of the device. A reset causes the timer

\* counter to set it's value to the reset value.

\*

\* @param InstancePtr is a pointer to the XTmrCtr instance.

\* @param TmrCtrNumber is the timer counter of the device to operate on.

\* Each device may contain multiple timer counters. The timer

\* number is a zero based number with a range of

\* 0 - (XTC\_DEVICE\_TIMER\_COUNT - 1).

\*

\* @return None.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void XTmrCtr\_Reset(XTmrCtr \* InstancePtr, u8 TmrCtrNumber)

{

u32 CounterControlReg;

Xil\_AssertVoid(InstancePtr != NULL);

Xil\_AssertVoid(TmrCtrNumber < XTC\_DEVICE\_TIMER\_COUNT);

Xil\_AssertVoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

/\*

\* Read current contents of the register so it won't be destroyed

\*/

CounterControlReg = XTmrCtr\_ReadReg(InstancePtr->BaseAddress,

TmrCtrNumber, XTC\_TCSR\_OFFSET);

/\*

\* Reset the timer by toggling the reset bit in the register

\*/

XTmrCtr\_WriteReg(InstancePtr->BaseAddress, TmrCtrNumber,

XTC\_TCSR\_OFFSET,

CounterControlReg | XTC\_CSR\_LOAD\_MASK);

XTmrCtr\_WriteReg(InstancePtr->BaseAddress, TmrCtrNumber,

XTC\_TCSR\_OFFSET, CounterControlReg);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\*

\* Checks if the specified timer counter of the device has expired. In capture

\* mode, expired is defined as a capture occurred. In compare mode, expired is

\* defined as the timer counter rolled over/under for up/down counting.

\*

\* When interrupts are enabled, the expiration causes an interrupt. This function

\* is typically used to poll a timer counter to determine when it has expired.

\*

\* @param InstancePtr is a pointer to the XTmrCtr instance.

\* @param TmrCtrNumber is the timer counter of the device to operate on.

\* Each device may contain multiple timer counters. The timer

\* number is a zero based number with a range of

\* 0 - (XTC\_DEVICE\_TIMER\_COUNT - 1).

\*

\* @return TRUE if the timer has expired, and FALSE otherwise.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int XTmrCtr\_IsExpired(XTmrCtr \* InstancePtr, u8 TmrCtrNumber)

{

u32 CounterControlReg;

Xil\_AssertNonvoid(InstancePtr != NULL);

Xil\_AssertNonvoid(TmrCtrNumber < XTC\_DEVICE\_TIMER\_COUNT);

Xil\_AssertNonvoid(InstancePtr->IsReady == XIL\_COMPONENT\_IS\_READY);

/\*

\* Check if timer is expired

\*/

CounterControlReg = XTmrCtr\_ReadReg(InstancePtr->BaseAddress,

TmrCtrNumber, XTC\_TCSR\_OFFSET);

return ((CounterControlReg & XTC\_CSR\_INT\_OCCURED\_MASK) ==

XTC\_CSR\_INT\_OCCURED\_MASK);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Default callback for the driver does nothing. It matches the signature of the

\* XTmrCtr\_Handler type.

\*

\* @param CallBackRef is a pointer to the callback's data.

\* @param TmrCtrNumber is the ID of the timer counter which a user-defined

\* callback would normally operate on.

\*

\* @return None.

\*

\* @note None.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static void XTmrCtr\_StubCallback(void \*CallBackRef, u8 TmrCtrNumber)

{

Xil\_AssertVoid(CallBackRef != NULL);

Xil\_AssertVoid(TmrCtrNumber < XTC\_DEVICE\_TIMER\_COUNT);

}

/\*\* @} \*/

**EE 316 Computer Engineering Junior Lab**

**Project 6-Tutorial**

**HelloWorld with Xilinx Microblaze**

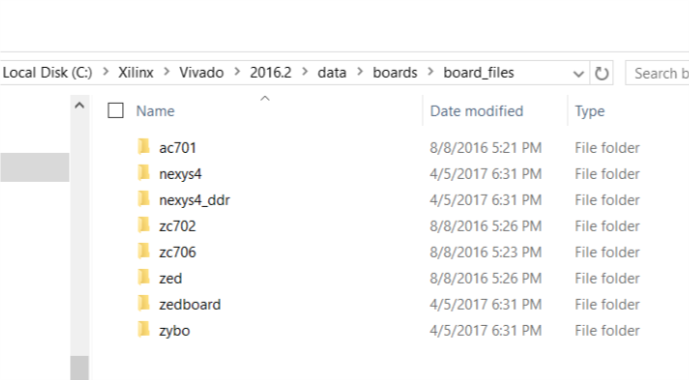
**Spring 2017**

For this project, please complete the following tutorials on the **Nexys4 DDR** board.

1. Download board files for Vivado:

<https://reference.digilentinc.com/reference/software/vivado/board-files?redirect=1>

and follow the directions and add new board files as shown below.



1. <https://reference.digilentinc.com/learn/programmable-logic/tutorials/nexys-4-ddr-getting-started-with-microblaze/start>
2. Download the following tcl-file by double clicking on it. This file was generated in Vivado version **2016.1**



Follow the following steps.

* Open the Tcl file in Notepad++. Search for “set scripts\_vivado\_version **2016.1**” in the text. Change the text to the version number of the Vivado on your computer.
* Save the Tcl file in a folder in C drive: Say **C:\MicroBlaze**
* Open Vivado. At the bottom of Vivado, in the Tcl command window type the following.

cd C:/MicroBlaze/

pwd

dir

source design\_1\_bd.tcl

* Wait until the block diagram is ready. Note this block diagram has a few new blocks and outputs. Study the new blocks by double clicking each of the blocks one at a time.
* Now go back to step 12 first tutorial from digilentinc.com and continue from here. You may need to assign the ports and save the XDC file. Generate the bit-stream file, export to the SDK and open SDK and repeat the rest of the part 2 in this project.