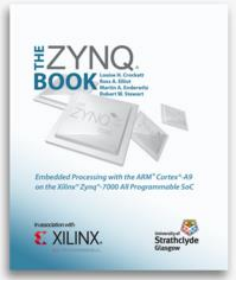
Zynq Book Tutorials II



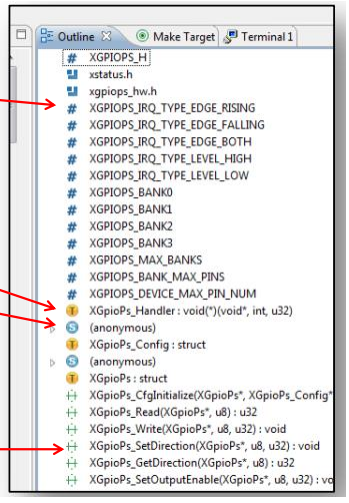
Zynq Processor System - GPIO. The *xgpiops.h* header file is shown on the *Outline* pane.

All #define statements (Macro definitions) are listed with a # icon.

All type definitions are shown with a yellow circular T icon.

All structs are shown with a circular blue S icon.

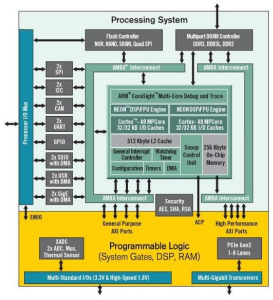
All function prototypes are shown with circular icons with a green hatch pattern.



Zynq processor System - GPIO. In order to use these functions in our application, we first need to include them in our C code.

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

The *#include "xparameters.h*" gives access to the predefined *#define* macros which are provided in the Xilinx Board Support Package (BSP) for the Digilent Zybo Board.

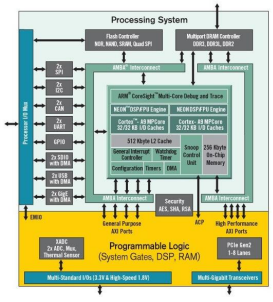


Zynq processor System - GPIO. Although we have added the *#include "xparameters.h",* we might not know where the file is located in the tree of sources or what is in it.

But the SDK will find it for us.

Hold down the *Ctrl* key on and hover your mouse over the #include line and the line of C has become a link.

Click the link and the *xparameters.h* source file will be open in the editor.



Zynq processor System - GPIO. All Xilinx drivers use the same principles of operation and require that the driver be initialized before use.

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

XGpioPs my_Gpio

```
typedef struct {
      XGpioPs_Config GpioConfig;
      u32 IsReady;
      XGpioPs_Handler Handler;
      void *CallBackRef;
} XGpioPs;
```


Zynq processor System - GPIO. In declaring an instance of a struct, values are not assigned to the variables inside it.

The *struct* represents various operating parameters and in the case of complex peripherals there may be a very large number of variables inside the *struct*.

XGpioPs my_Gpio

```
typedef struct {
      XGpioPs_Config GpioConfig;
      u32 IsReady;
      XGpioPs_Handler Handler;
      void *CallBackRef;
} XGpioPs;
```

Zynq processor System - GPIO. Fortunately in the case of the GPIO it is relatively simple and there are only a few variables. Here is the declaration of an instance of the *struct* and called it *my_Gpio*. There are four variables inside the *struct*.

The first is called “GpioConfig” and is of data type *XGpioPs_Config*.

This data type is actually another *struct* used to configure the *my_Gpio* instance.

XGpioPs my_Gpio

```
typedef struct {
      XGpioPs_Config GpioConfig;
      u32 IsReady;
      XGpioPs_Handler Handler;
      void *CallBackRef;
} XGpioPs;
```

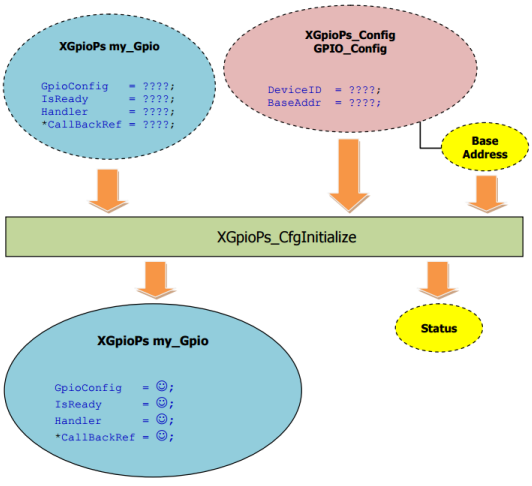

Zynq processor System - GPIO. The remaining three variables are all of different data types. Xilinx supplies a function in C which does the initialization of the variables called *XGpioPs_CfgInitialize*.

This function automatically configures everything because all of the variables inside are uninitialized when the *struct* is declared.

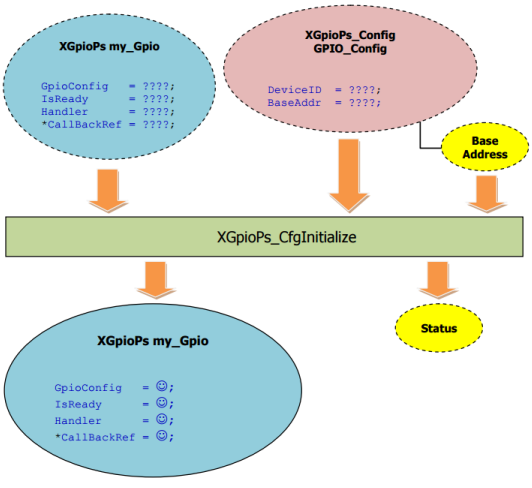
XGpioPs my_Gpio

```
typedef struct {
      XGpioPs_Config GpioConfig;
      u32 IsReady;
      XGpioPs_Handler Handler;
      void *CallBackRef;
} XGpioPs;
```

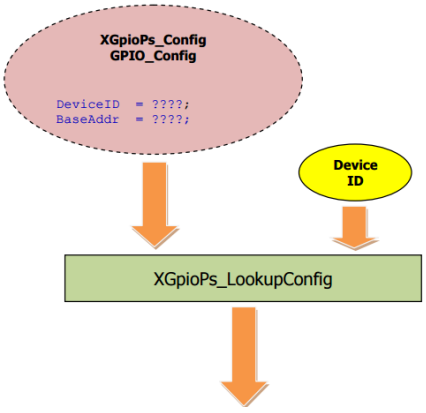
Zynq processor System - GPIO. The function requires three inputs: the instance of *my_Gpio* that was declared, the *GPIO_Config struct*, and a base address which can be extracted from the *GPIO_Config struct*.



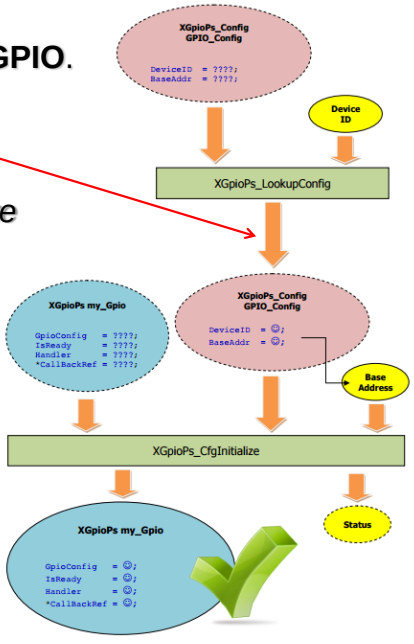
Zynq processor System - GPIO. The output of the *XGpioPs_CfgInitialize* function is a status value which Indicates whether the initialization was successful.



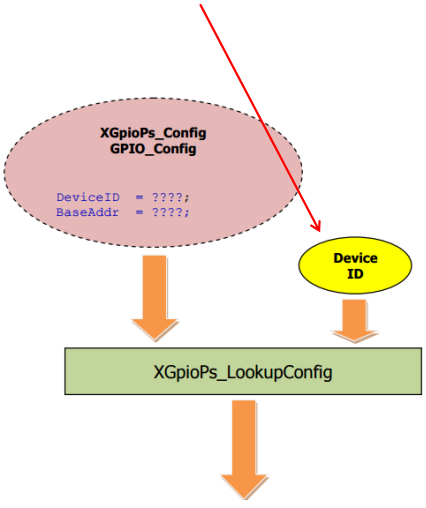
Zynq processor System - GPIO. However, there is a problem as the *GPIO_Config struct* hasn't been initialized. Another automated function called *XGpioPs_LookupConfig* provides the initialization.



Zynq processor System - GPIO. The output of the *XGpioPs_LookupConfig* function is therefore fed into the *XGpioPs_CfgInitialize* function.



Zynq processor System - GPIO. The information is required is the *Device ID*, and that comes from the *xparameters.h* file



ECE3622 Embedded Systems Design Zynq Book Tutorials II

Zynq processor System - GPIO. To employ the Xilinx GPIO driver two *structs* are declared: *XGpioPs_Config* and *XGpioPs* which is used to control the GPIO.

pointer

```
int main()
{
        XGpioPs_Config *GPIO_Config;
        XGpioPs my_Gpio;
        int Status, Temp;
        GPIO_Config =
              XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
        Status = XGpioPs_CfgInitialize(&my_Gpio, GPIO_Config,
              GPIO_Config->BaseAddr);
```

These are listed in the *Outline* view of SDK when *xgpiops.h* is selected.

Zynq processor System - GPIO. The function *XGpioPs_LookupConfig* needs the *DEVICE ID* parameter passed to it which is in the *xparameters.h* file and automatically generated for the design by Xilinx.

```
int main()
{
        XGpioPs_Config *GPIO_Config;
        XGpioPs my_Gpio;
        int Status;
        GPIO_Config =
              XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
        Status = XGpioPs_CfgInitialize(&my_Gpio, GPIO_Config,
              GPIO_Config->BaseAddr);
```

GPIO_Config is the instance name for the *XGpioPs_Config struct* data type.

Zynq processor System - GPIO. The *XGPioPs_Config struct* contains two variables: a 16-bit device ID called DeviceId, and a 32-bit base address called BaseAddr.

```
int main()
{
        XGpioPs_Config *GPIO_Config;
        XGpioPs my_Gpio;
        int Status;
        GPIO_Config =
              XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
        Status = XGpioPs_CfgInitialize(&my_Gpio, GPIO_Config,
              GPIO_Config->BaseAddr);
```

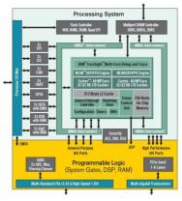
SDK can determine the definition of any function or data type by using the *Open Declaration* feature from the Right-Click menu.

Zynq processor System - GPIO. Discovering the base address for the GPIO is by using the *GPIO_Config->BaseAddr* notation. The compiler to looks inside the *GPIO_Config struct* for a variable called *BaseAddr* by using the -> syntax.

```
int main()
{
        XGpioPs_Config *GPIO_Config;
        XGpioPs my_Gpio;
        int Status;
        GPIO_Config =
            XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
        Status = XGpioPs_CfgInitialize(&my_Gpio, GPIO_Config,
            GPIO_Config->BaseAddr);
```


Zynq processor System - GPIO. The *XGpioPs_CfgInitialize* function returns a value to the variable Status. Success is a returned value of 0

```
int main()
{
        XGpioPs_Config *GPIO_Config;
        XGpioPs my_Gpio;
        int Status;
        GPIO_Config =
            XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
        Status = XGpioPs_CfgInitialize(&my_Gpio, GPIO_Config,
            GPIO_Config->BaseAddr);
```



Zynq processor System - GPIO. The *XGpioPs_SetDirectionPin* and the *XGpioPs_WritePin* functions send values to the various registers within the GPIO peripheral.

```
XGpioPs_SetDirectionPin(&my_Gpio, 7, 1);
        while(1)
        {
                XGpioPs_WritePin(&my_Gpio, 7, 0);
```

XGpioPs_SetDirectionPin sets the direction of the GPIO to be an output, but only for a specific pin 7.

XGpioPs_WritePin writes a value to pin 7 as either 0 or 1.

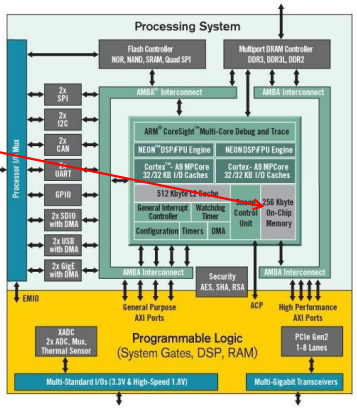
*XGpioPs_SetDirectionPin(XGpioPs *InstancePtr, int Pin, int Direction);*
*XGpioPs_WritePin(XGpioPs *InstancePtr, int Pin, int Data);*

Zynq processor System - GPIO. The C language *code snippet* excuted then is:

```
while(1)
        {
                XGpioPs_WritePin(&my_Gpio, 7, 0);          //OFF
                for (Temp=0; Temp< 20000; Temp++)
                {
                        xil_print(".");        //delay
                }
                XGpioPs_WritePin(&my_Gpio, 7, 1);          //ON
                for (Temp=0; Temp< 20000; Temp++)
                {
                        xil_print(".");        //delay
                }
        }

        return 0;
}
```

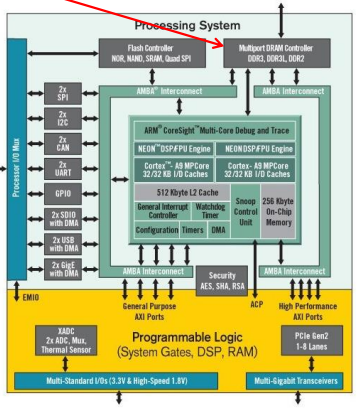
Zynq Processor System - Memory. Memory is not only used to store software in an embedded system but as the interface between the embedded processor system and some custom hardware in the FPGA design.

The On Chip Memory (OCM) in Zynq behaves like a dual port BlockRAM memory.



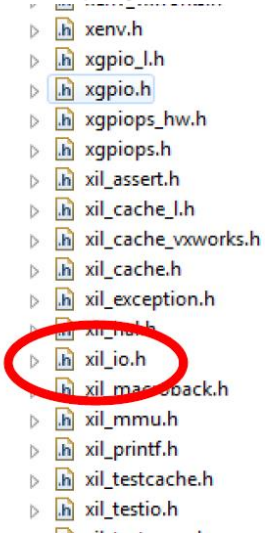
Zynq Processor System - Memory. External DDR memory works in much the same way as BlockRAMM.

The difference is that the memory controller interprets the request from the processor and generates a transaction for the external memory which uses the correct protocol of signals used by the DIMM.



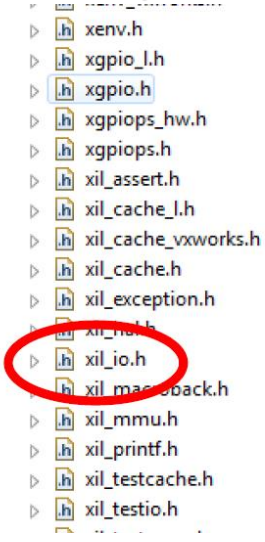
Zynq Processor System - Memory. To employ the external DDR memory, drivers that have been generated by the Library Generator are used.

The drivers are not associated with any peripheral, but are in the EDK in all processor designs in the *include* directory under the name of the processor instance.



Zynq Processor System - Memory. The Library Generator has also created other header files that do not relate to any particular peripheral.

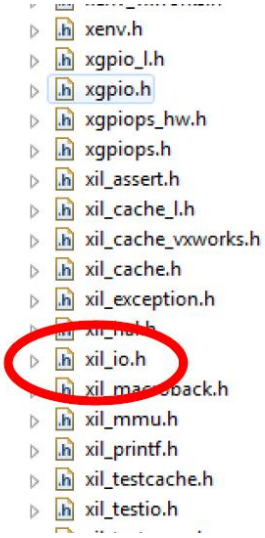
The *xil_io.h* header file has various functions designed specifically for reading from and writing to memory, in various byte widths.



Zynq Processor System - Memory. These functions include:

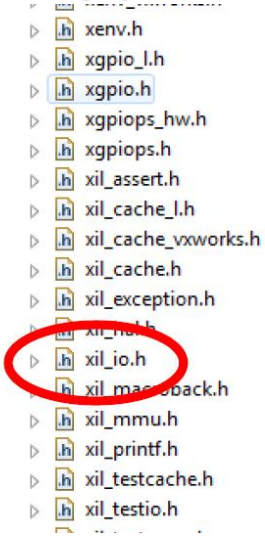
Xil_In8
Xil_In16
Xil_In32
Xil_Out8
Xil_Out16
Xil_Out32

The processor works on byte (8 bit) address boundaries. If an address in memory is specified, it refers to a number of bytes.



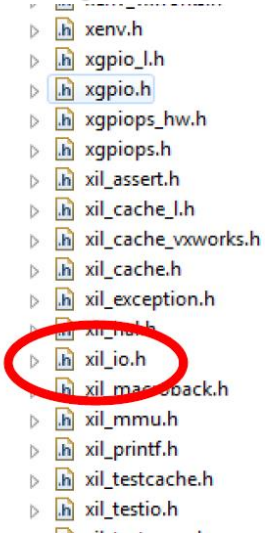
Zynq Processor System - Memory. Writing byte-wide data values into the first four consecutive locations in memory starting at *DDR_BASEADDR* is accomplished by writing to:

DDR_BASEADDR + 0
DDR_BASEADDR + 1
DDR_BASEADDR + 2
DDR_BASEADDR + 3



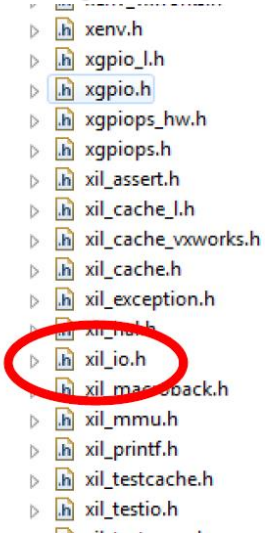
Zynq Processor System - Memory. Writing 16 bit-wide data values into the first four consecutive locations in memory starting at *DDR_BASEADDR* is accomplished by writing to:

DDR_BASEADDR + 0
DDR_BASEADDR + 2
DDR_BASEADDR + 4
DDR_BASEADDR + 6



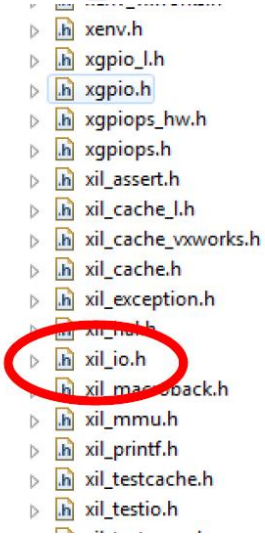
Zynq Processor System - Memory. Writing 32 bit-wide data values into the first four consecutive locations in memory starting at *DDR_BASEADDR* is accomplished by writing to:

DDR_BASEADDR + 0
DDR_BASEADDR + 4
DDR_BASEADDR + 8
DDR_BASEADDR + 12



Zynq Processor System - Memory. The xil_io.h header file has functions for reading and writing from memory in 8 bit, 16 bit and 32 bit transactions.

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

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



Zynq Processor System - Memory. An example of accessing the Zynq OCM:

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

        Xil_Out8(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 0, 0x12);
        Xil_Out8(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 1, 0x34);
        Xil_Out8(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 2, 0x56);
        Xil_Out8(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 3, 0x78);

        result1 = Xil_In32(XPAR_PS7_RAM_0_S_AXI_BASEADDR);

        Xil_Out16(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 4, 0x9876);
        Xil_Out16(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 6, 0x5432);

        result2 = Xil_In32(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 4);

        return(0);
        }
```

Zynq Processor System - Memory. Question: what is the value of *result1*?:

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

        Xil_Out8(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 0, 0x12);
        Xil_Out8(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 1, 0x34);
        Xil_Out8(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 2, 0x56);
        Xil_Out8(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 3, 0x78);

        result1 = Xil_In32(XPAR_PS7_RAM_0_S_AXI_BASEADDR);

        Xil_Out16(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 4, 0x9876);
        Xil_Out16(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 6, 0x5432);

        result2 = Xil_In32(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 4);

        return(0);
        }
```

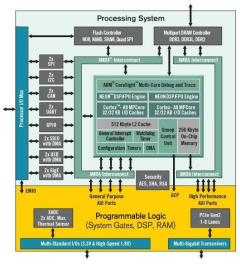

Zynq Processor System - Memory. Question: what is the value of *result2*?:

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

        Xil_Out8(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 0, 0x12);
        Xil_Out8(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 1, 0x34);
        Xil_Out8(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 2, 0x56);
        Xil_Out8(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 3, 0x78);

        result1 = Xil_In32(XPAR_PS7_RAM_0_S_AXI_BASEADDR);

        Xil_Out16(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 4, 0x9876);
        Xil_Out16(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 6, 0x5432);

        result2 = Xil_In32(XPAR_PS7_RAM_0_S_AXI_BASEADDR + 4);

        return(0);
        }
```

Zynq Processor System – Polled Timers. Loops are effective for implementing delays but they are extremely inefficient in terms of processor time.

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

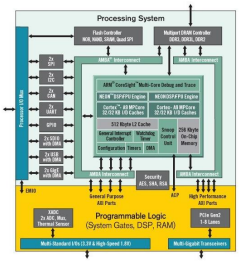
All of the time that we spent wastefully going around in the loop doing nothing could be better spent executing real code and producing useful results.



Zynq Processor System – Polled Timers. Rudimentary loops have no real control over the amount of delay time that would be produced by the delay.

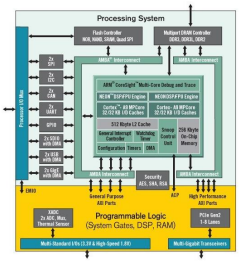
```
for (i=0; i<2000; i++)
        {
        print(".");
        }
```

The size of the loop can be adjusted, but this is a trial and error process and has no precise correlation to predictable and measurable units of time.



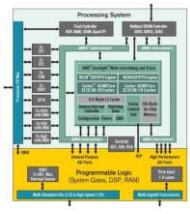
Zynq Processor System – Polled Timers. It is possible to accurately measure time using a hardware-based timer peripheral. This gives much greater flexibility in the designs, considerably more control and avoids wasting the processor's time.

The timer will decrement once per clock cycle until it reaches zero and then stop. While counting down the timer will be *polled*.



Zynq Processor System – Polled Timers.

```
#include <stdio.h>
#include "xscutimer.h"
#include "xparameters.h"

int main()
{
        // Declare variables

        int Status;
        int timer_value;
        int counter = 0;

        // Declare two structs for the timer instance and timer config
        XScuTimer my_Timer;
        XScuTimer_Config *Timer_Config;
```



Zynq Processor System – Polled Timers.

```
// Look up the the config information for the timer
Timer_Config =
  XScuTimer_LookupConfig(XPAR_PS7_SCUTIMER_0_DEVICE_ID);

// Initialize the timer using the config information
Status = XScuTimer_CfgInitialize(&my_Timer, Timer_Config,
            Timer_Config->BaseAddr);

// Load the timer with a value that represents one second
// The timer is clocked at half the frequency of the CPU (processor)
XScuTimer_LoadTimer(&my_Timer,
    XPAR_PS7_CORTEXA9_0_CPU_CLK_FREQ_HZ / 2);

// Start the timer running (it counts down)
XScuTimer_Start(&my_Timer);
```

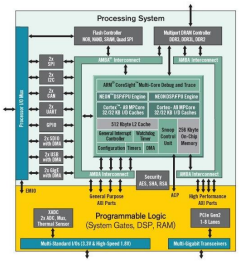
ECE3622 Embedded Systems Design Zynq Book Tutorials II

Zynq Processor System – Polled Timers.

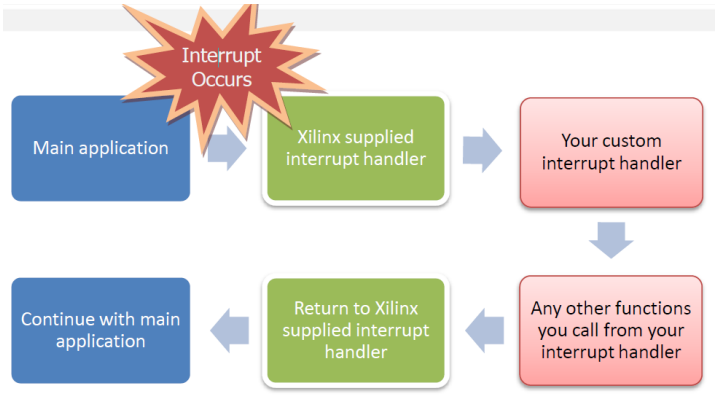
```
while(1)  // An infinite loop
{
        // Read the value of the timer
        timer_value = XScuTimer_GetCounterValue(&my_Timer);

        // If the timer has reached zero
        if (timer_value == 0)
        {
           // Re-load the original value into the timer and re-start it
                XScuTimer_RestartTimer(&my_Timer);
                // Write something to the UART (and count the
                    seconds)
                printf("Timer has reached zero %d times\n\r",
                    counter++);
        }
```

Zynq Processor System – Polled Timers.

```
                    else
                {
                    // Show the value of the timer's counter value, for
                        debugging purposes
                    //xil.print("Timer is still running (Timer value =
                        %d)\n\r", timer_value);
                }
            }
            return 0;
}
```



Zynq Processor System – Interrupt Timers. In the ARM Cortex-A9 processor of the Zynq all interrupts are managed by and connected via the general interrupt controller.



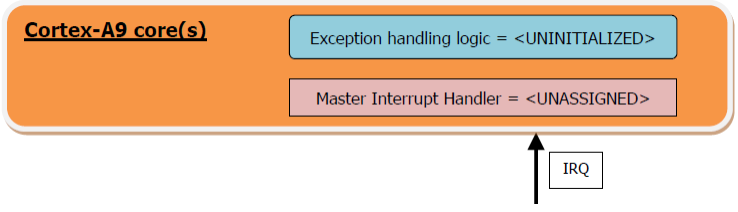
Zynq Processor System – Interrupt Timers. Instances of the general interrupt controller (GIC) and the timer are created and initialized using the *Lookup_Config* and *CfgInitialize* functions.

```
XScuTimer my_Timer;
XScuTimer_Config *Timer_Config;
XScuGic my_Gic;
XScuGic_Config *Gic_Config;
Gic_Config =
   XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);
XScuGic_CfgInitialize(&my_Gic, Gic_Config,
   Gic_Config->CpuBaseAddress);
Timer_Config =
   XScuTimer_LookupConfig(XPAR_PS7_SCUTIMER_0_DEVICE_ID);
XScuTimer_CfgInitialize(&my_Timer, Timer_Config,
   Timer_Config->BaseAddr);
```

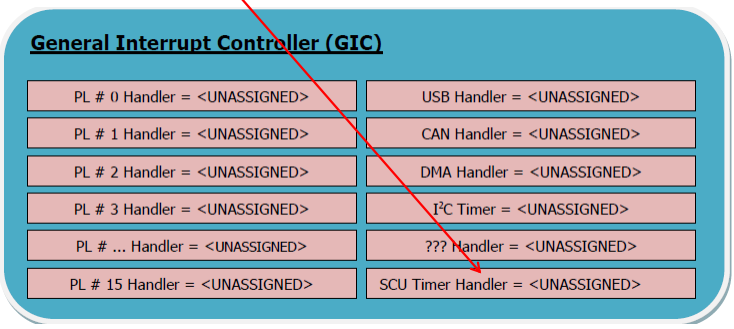
ECE3622 Embedded Systems Design Zynq Book Tutorials II

Zynq Processor System – Interrupt Timers. The Cortex-A9 CPU cores have internal exception handing logic which is disabled and initialized at power-up.

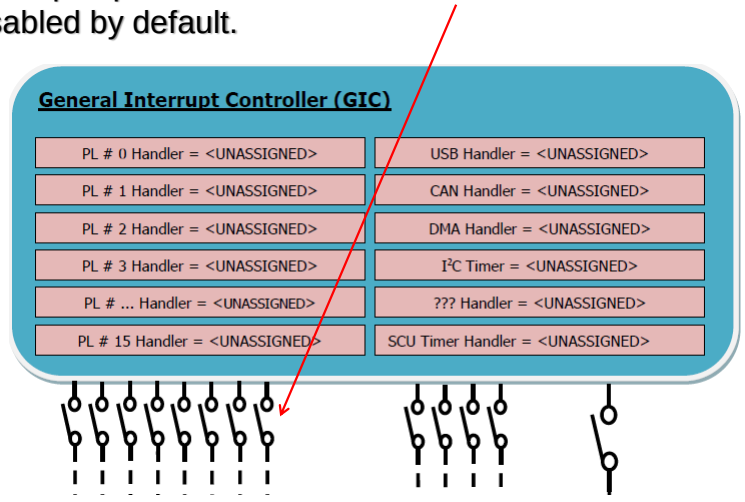
There is one standard interrupt pin on the Cortex-A9 core, and there is a master interrupt handler which the CPU executes when receiving any interrupt request (IRQ). The handler is unassigned.



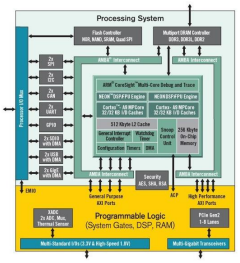
Zynq Processor System – Interrupt Timers. The General Interrupt Controller (GIC) has the ability to manage many interrupt inputs and has a table which allows interrupt handlers to be assigned to each incoming interrupt.

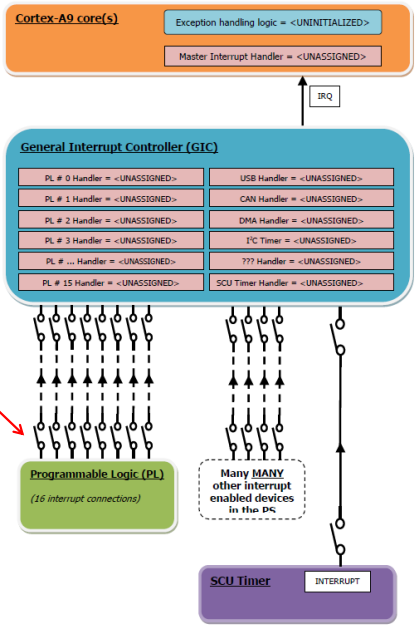


Zynq Processor System – Interrupt Timers. Each interrupt input on the GIC has an enable switch that is disabled by default.



Zynq Processor System – Interrupt Timers. Each peripheral / interrupt source has an output enable switch that is disabled by default.

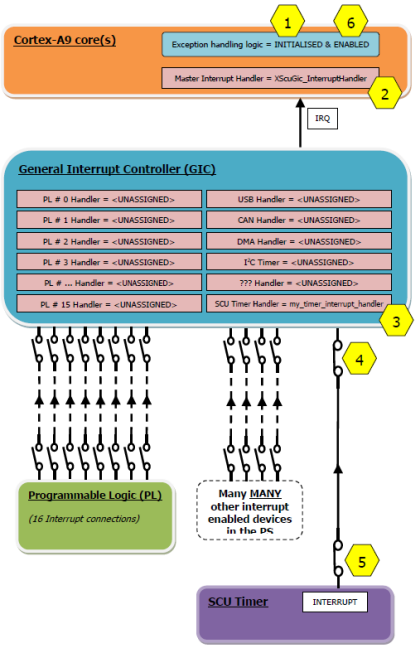




Zynq Processor System – Interrupt Timers. Initialize the exception handling features on the ARM processor using a function call from the *xil_exception.h* header file.

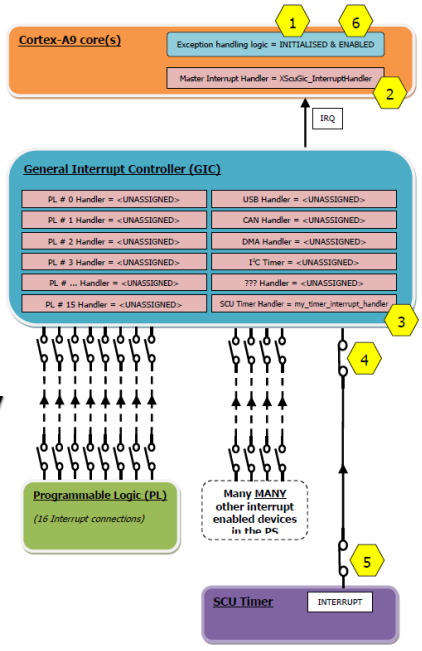
```
Xil_ExceptionInit();
```

1



Zynq Processor System – Interrupt Timers. When an interrupt occurs, the processor has to interrogate the interrupt controller to find out which peripheral generated the interrupt.

Xilinx provides an interrupt handler to do this automatically *XScuGic_InterruptHandler*

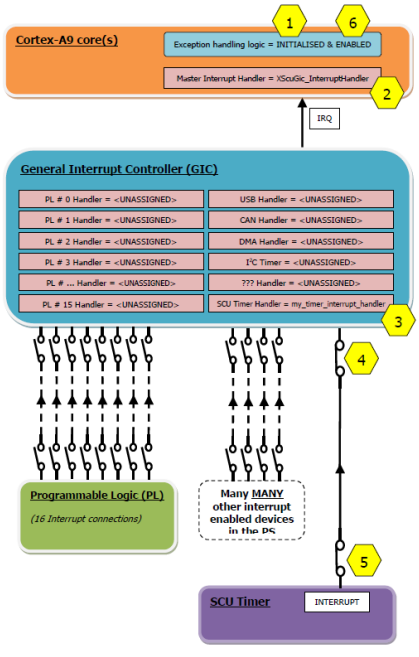


ECE3622 Embedded Systems Design Zynq Book Tutorials II

Zynq Processor System – Interrupt Timers. Initialize the exception handling features on the ARM processor using a function call from the *xil_exception.h* header file.

```
Xil_ExceptionInit();
```

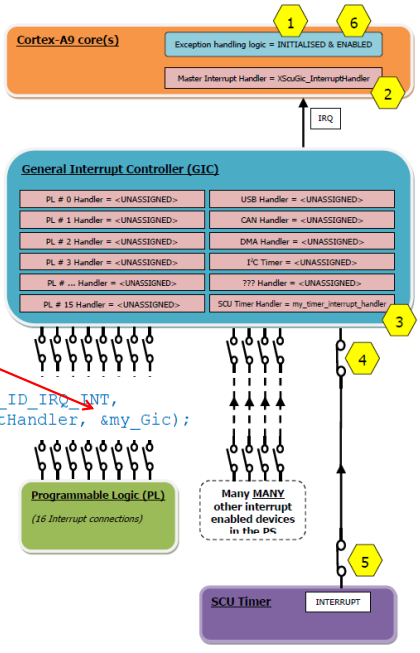
1



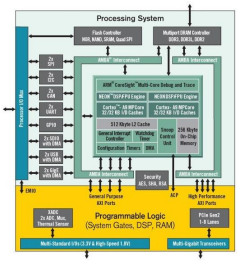
Zynq Processor System – Interrupt Timers. The supplied handler has to be assigned it to the interrupt controller using the GIC instance at the end of the function: *&my_Gic*.

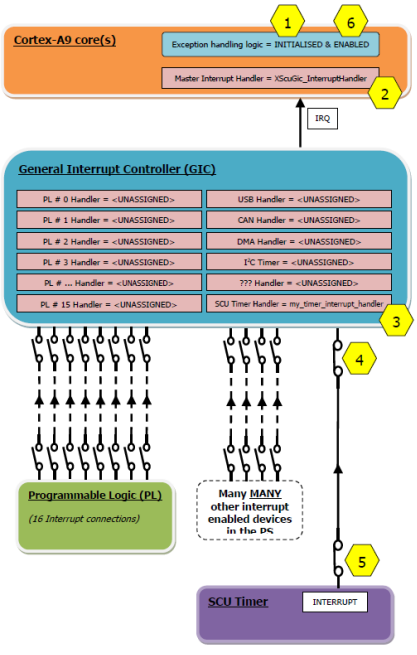
```
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT,
    (Xil_ExceptionHandler)XScuGic_InterruptHandler, &my_Gic);
```

2

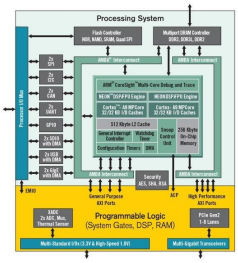


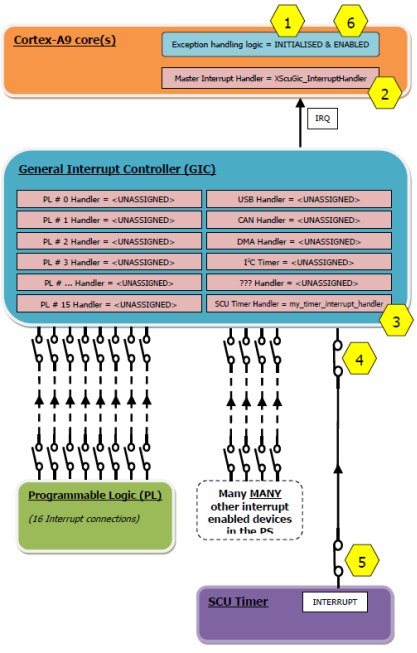
Zynq Processor System – Interrupt Timers. The interrupt handler has to be assigned to the timer peripheral and is named: *my_timer_interrupt_handler*.





Zynq Processor System – Interrupt Timers. The interrupt handler is connected to a unique interrupt ID number for the timer: *XPAR_SCUTIMER_INTR*

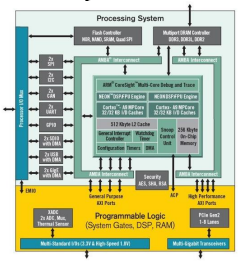


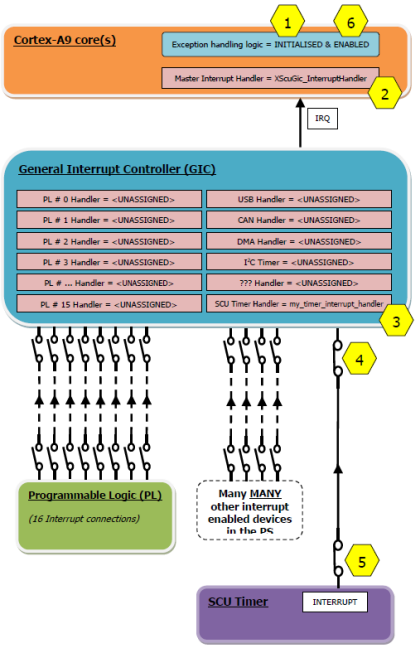


ECE3622 Embedded Systems Design

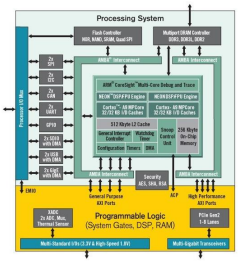
Zynq Book Tutorials II

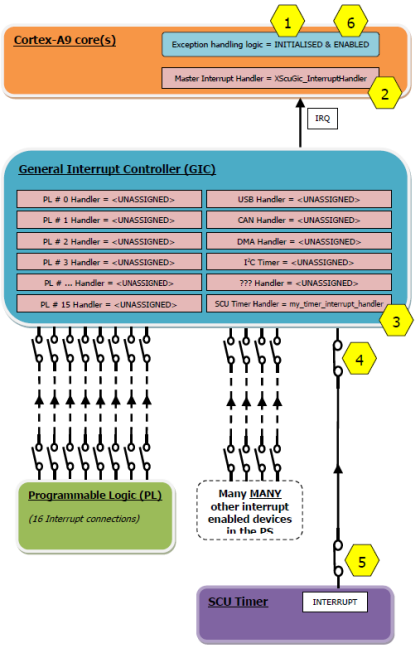
Zynq Processor System – Interrupt Timers. A list of these IDs is in the *xparameters_ps.h* header file for all of the peripherals in the PS which generate interrupts.





Zynq Processor System – Interrupt Timers. For an interrupt which comes from a peripheral in the PL, a similar list is in the *xparameters.h* header file.

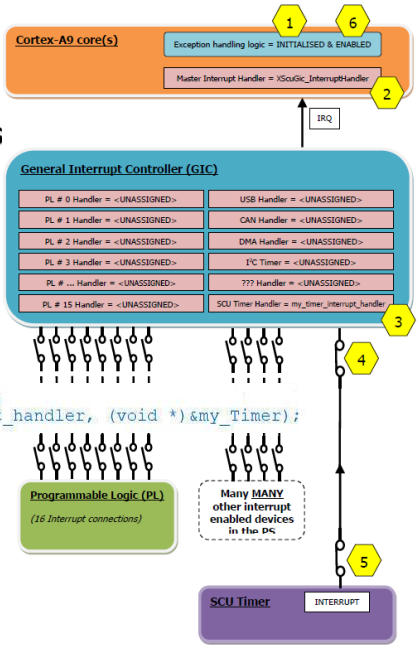




Zynq Processor System – Interrupt Timers. The interrupt handler which handles the interrupts for the timer peripheral is connected to its unique interrupt ID number.

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

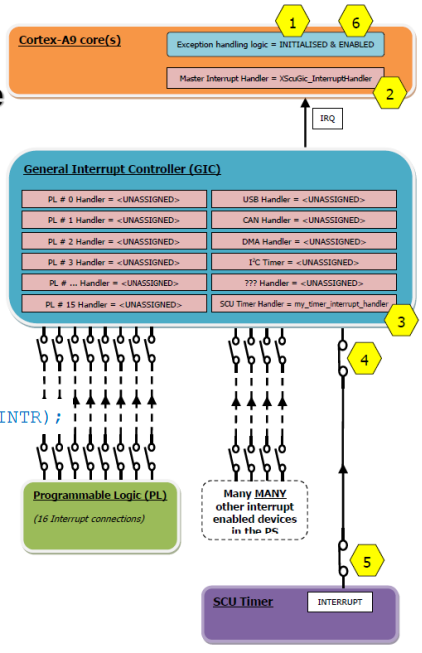
3



Zynq Processor System – Interrupt Timers. Next, enable the interrupt input for the timer on the interrupt controller. Interrupt controllers can be enabled and disabled.

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

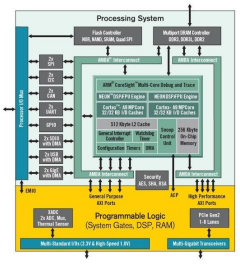
4

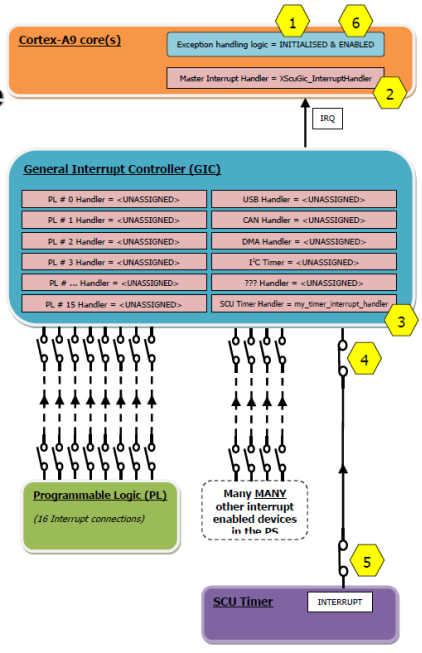


Zynq Processor System – Interrupt Timers. Next, enable the interrupt output on the timer.

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

5

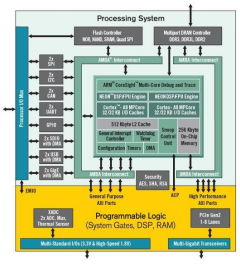


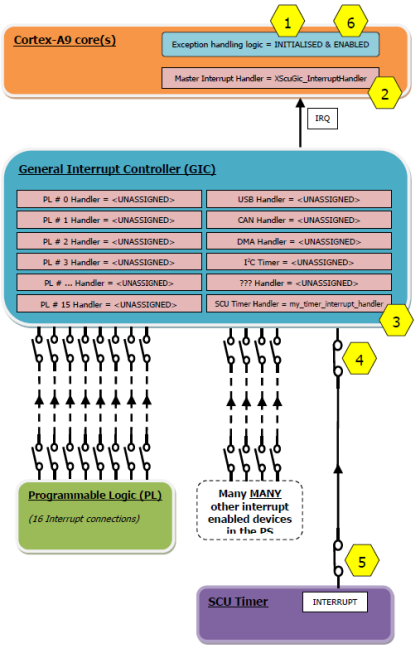


Zynq Processor System – Interrupt Timers. Finally, enable interrupt handling on the ARM processor. This function call is in the *xil_exception.h* header file.

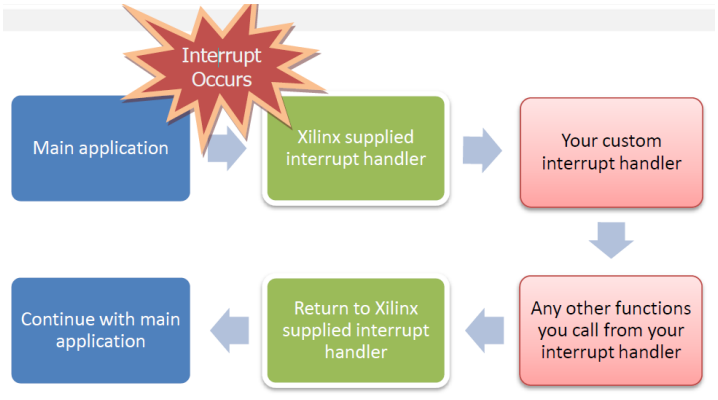
```
Xil_ExceptionEnable();
```

6

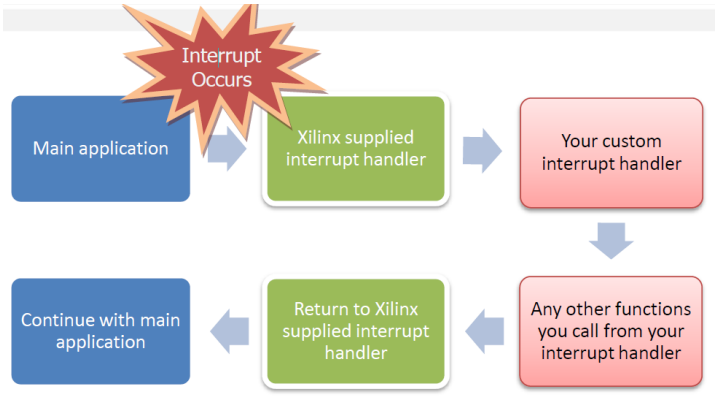




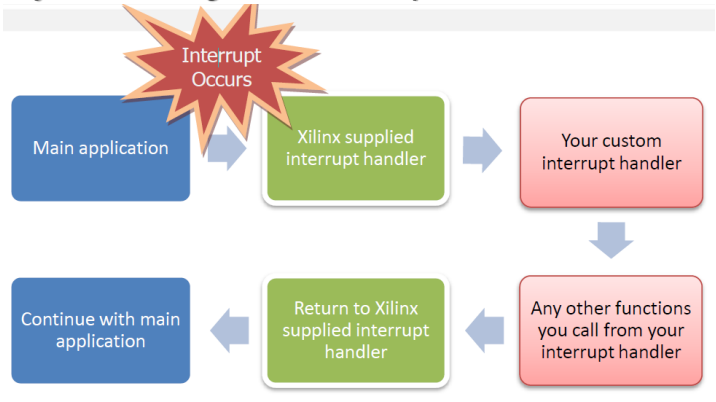
Zynq Processor System – Interrupt Timers. An interrupt handler, which is the function that will be called when the interrupt occurs, is now developed as *my_timer_interrupt_handler*.



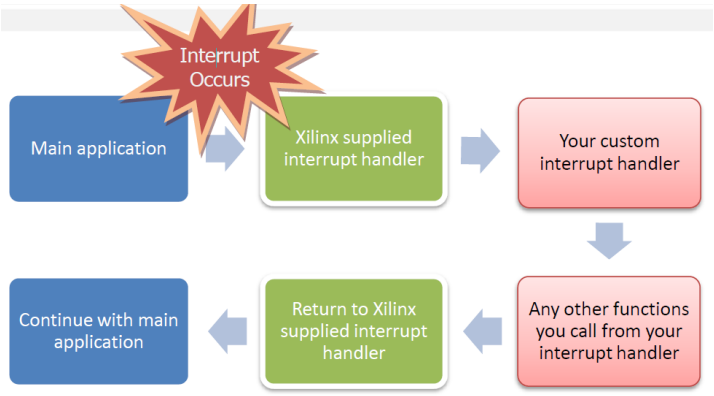
Zynq Processor System – Interrupt Timers. Good programming practice when coding an interrupt handler function is to first check to make absolutely sure that the right peripheral has raised the interrupt.



Zynq Processor System – Interrupt Timers. Problems can be created when it is believed that one interrupt has occurred, when actually it's a completely different one. This problem is hard to detect because in most cases you have no way of checking which interrupt has occurred.

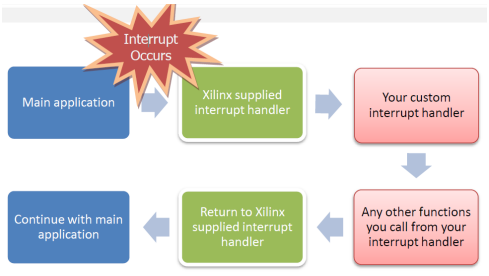


Zynq Processor System – Interrupt Timers. Xilinx has solved this problem by including some clever code in their driver, using a *CallBackRef*.



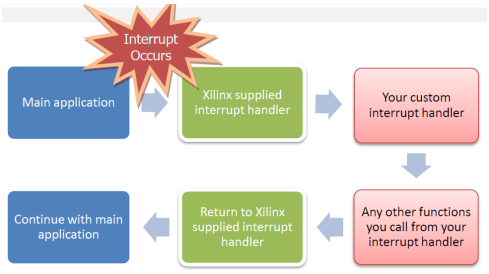
Zynq Processor System – Interrupt Timers. The Xilinx interrupt handler *XScuGic_InterruptHandler* calls the application handler, but it also passes information about the driver instance which is relevant to the peripheral that generated the interrupt or the *CallBackRef*.

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



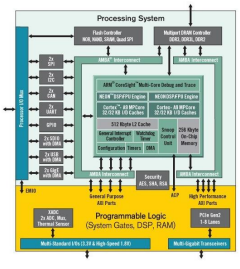
Zynq Processor System – Interrupt Timers. The Xilinx drivers have essentially already provided the details of which peripheral generated the interrupt.

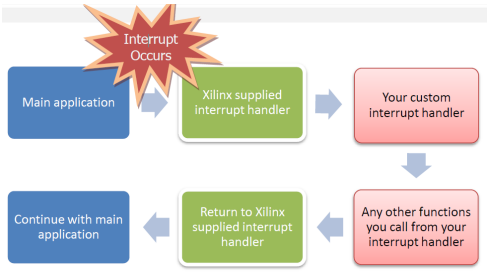
```
static void my_timer_interrupt_handler(void *CallBackRef)
{
    // Your code goes in here
}
```



Zynq Processor System – Interrupt Timers. If the timer is to be controlled as part of our interrupt handler, without the *CallBackRef* it could not be done.

This is because the original instance of *my_Timer* does not exist in the scope of this function since it was declared in *main*().

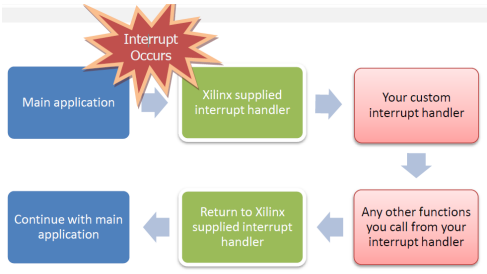




Zynq Processor System – Interrupt Timers. But a local copy of the *my_Timer* instance can be declared and assigned to the information provided by the *CallBackRef*:

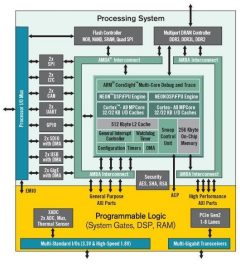
```
static void my_timer_interrupt_handler(void *CallBackRef)
{
XScuTimer *my_Timer_LOCAL = (XScuTimer *) CallBackRef;
}
```

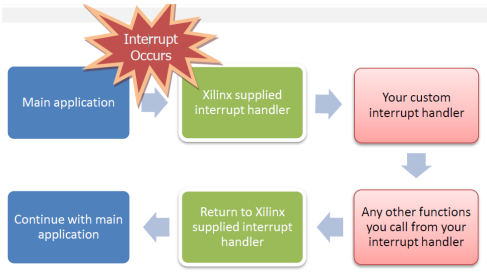
The instance *my_Timer_LOCAL* is now an exact copy of the one in the *main*() function and the timer can be controlled in the same way as before.



Zynq Processor System – Interrupt Timers. Any changes to *my_Timer_LOCAL* will also apply to the original copy of *my_Timer* because they are linked together using a pointer.

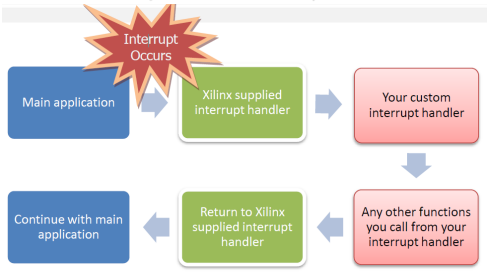
```
static void my_timer_interrupt_handler(void *CallBackRef)
{
    XScuTimer *my_Timer_LOCAL = (XScuTimer *) CallBackRef;
}
```





Zynq Processor System – Interrupt Timers. Check to insure that the timer really did generate this interrupt. To do that use a standard function from the timer header file.

```
static void my_timer_interrupt_handler(void *CallBackRef)
{
    XScuTimer *my_Timer_LOCAL = (XScuTimer *) CallBackRef;
    if (XScuTimer_IsExpired(my_Timer_LOCAL))
        {
            xil_print("We are in the interrupt handler!!\n\r");
        }
}
```

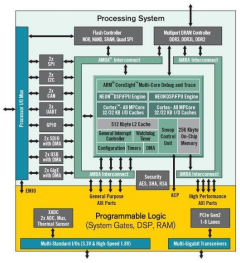


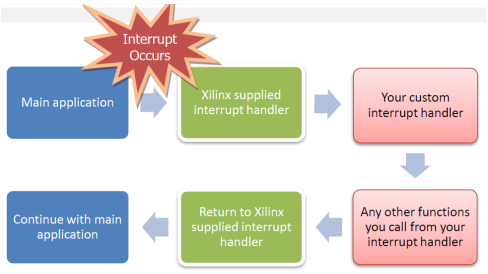
Zynq Processor System – Interrupt Timers. All of the functions work on *my_Timer_LOCAL* as in *main*() for *my_Timer* because they the same data type: *XScuTimer*:

```
static void my_timer_interrupt_handler(void *CallBackRef)
{
    XScuTimer *my_Timer_LOCAL = (XScuTimer *) CallBackRef;
    if (XScuTimer_IsExpired(my_Timer_LOCAL))
        {
            xil_print("We are in the interrupt handler!!\n\r");
        }
}
```

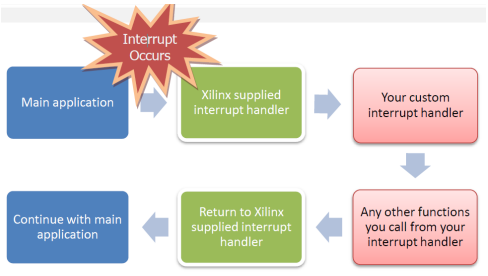
Zynq Processor System – Interrupt Timers. Finally clear the interrupt condition in the timer peripheral. If not done, the interrupt condition will still be active and upon exiting the interrupt handler and the processor executes the interrupt again. Again there is a supplied function call:

XScuTimer_ClearInterruptStatus(my_Timer_LOCAL);





Zynq Processor System – Interrupt Timers. Task: Print a message to the UART at an interval of 1 Hz which contain a counter which shows how many interrupts have been generated since the start of the application and make an LED flash also at 1 Hz using interrupt control.



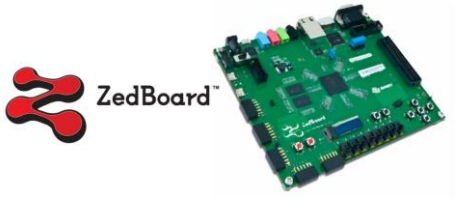
Zynq Processor System – Interrupt Timers.

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


```
#define INTERRUPT_COUNT_TIMEOUT_VALUE 50

// Function prototypes
static void my_timer_interrupt_handler(void *CallBackRef);

// Global variables
int InterruptCounter = 0;
int Flashing_LED_state = 0;
```


Zynq Processor System – Interrupt Timers.

```
int main()
{
        init_platform();

        // Declare variables
        int Status;
        unsigned int DIP_value;
        unsigned int LED_value;

        // Declare two structs.  One for the Timer instance, and
        //     the other for the timer's config information
        XScuTimer my_Timer;
        XScuTimer_Config *Timer_Config;

        // Declare two structs.  One for the General Interrupt
        //     Controller (GIC) instance, and the other for config information
        XScuGic my_Gic;
        XScuGic_Config *Gic_Config;
```



Zynq Processor System – Interrupt Timers.

```
// Initialize the GPIO using a reference to the my_Gpio struct,
//    the struct "GPIO_Config", and a base address value
Status = XGpio_CfgInitialize(&my_Gpio, GPIO_Config,
    GPIO_Config->BaseAddress);
Status = XGpioPs_CfgInitialize(&my_PS_Gpio, PS_GPIO_Config,
     PS_GPIO_Config->BaseAddr);

// Set the direction of the bits in the GPIO.
// The lower (LSB) 8 bits of the GPIO are for the DIP Switches (inputs).
// The upper (MSB) 8 bits of the GPIO are for the LEDs (outputs).
XGpio_SetDataDirection(&my_Gpio, 1, 0x00FF);
XGpioPs_SetDirectionPin(&my_PS_Gpio,7,1);
```

Zynq Processor System – Interrupt Timers.

```
// Look up the config information for the GIC
Gic_Config =
     XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);

// Initialise the GIC using the config information
Status = XScuGic_CfgInitialize(&my_Gic, Gic_Config,
    Gic_Config->CpuBaseAddress);

// Look up the the config information for the timer
Timer_Config =
     XScuTimer_LookupConfig(XPAR_PS7_SCUTIMER_0_DEVICE_ID);

// Initialise the timer using the config information
Status = XScuTimer_CfgInitialize(&my_Timer, Timer_Config,
    Timer_Config->BaseAddr);

// Initialize Exception handling on the ARM processor
Xil_ExceptionInit();
```

Zynq Processor System – Interrupt Timers.

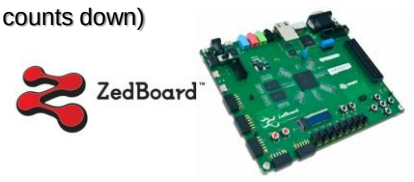
```
// Connect the supplied Xilinx general interrupt handler
// to the interrupt handling logic in the processor.
// All interrupts go through the interrupt controller, so the
// ARM processor has to first "ask" the interrupt controller
// which peripheral generated the interrupt.  The handler that
// does this is supplied by Xilinx and is called
// XScuGic_InterruptHandler

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

Zynq Processor System – Interrupt Timers.

```
// Assign (connect) our interrupt handler for our Timer
Status = XScuGic_Connect(&my_Gic, XPAR_SCUTIMER_INTR,
    (Xil_ExceptionHandler)my_timer_interrupt_handler,
    (void *)&my_Timer);

// Enable the interrupt *input* on the GIC for the timer's interrupt
XScuGic_Enable(&my_Gic, XPAR_SCUTIMER_INTR);

// Enable the interrupt *output* in the timer.
XScuTimer_EnableInterrupt(&my_Timer);

// Enable interrupts in the ARM Processor.
Xil_ExceptionEnable();
```

Zynq Processor System – Interrupt Timers.

```
// Load the timer with a value that represents one second of real time
// The SCU Timer is clocked at half the frequency of the CPU.
|
XScuTimer_LoadTimer(&my_Timer,
    XPAR_PS7_CORTEXA9_0_CPU_CLK_FREQ_HZ / 2);

// Enable Auto reload mode on the timer.  When it expires, it re-loads
// the original value automatically.  This means that the timing interval
// is never skewed by the time taken for the interrupt handler to run

XScuTimer_EnableAutoReload(&my_Timer);

// Start the SCU timer running (it counts down)
XScuTimer_Start(&my_Timer);
```



ECE3622 Embedded Systems Design Zynq Book Tutorials II

Zynq Processor System – Interrupt Timers.

```
// Create an infinite loop
while(1)
{
        // Read from the GPIO for the DIP switches
        DIP_value = XGpio_DiscreteRead(&my_Gpio, 1);

        // Mask the upper 8 bits
        DIP_value = DIP_value & 0x00FF;

        // Assign a value to LED_Value variable
        LED_value = DIP_value << 8;
```

Zynq Processor System – Interrupt Timers.

```
// Write the current status of the flashing LED to the GPIO
XGpioPs_WritePin(&my_PS_Gpio,7,Flashing_LED_state);

// Print the values of the variables to the UART
xil_print("DIP = 0x%04X, LED = 0x%04X\n\r", DIP_value, LED_value);

// Write the value back to the GPIO
XGpio_DiscreteWrite(&my_Gpio, 1, LED_value);

// Check to see if we've serviced more than 20 interrupts
if (InterruptCounter >= INTERRUPT_COUNT_TIMEOUT_VALUE)
        {
        // Break out of the while loop
            break;
        }
}
```

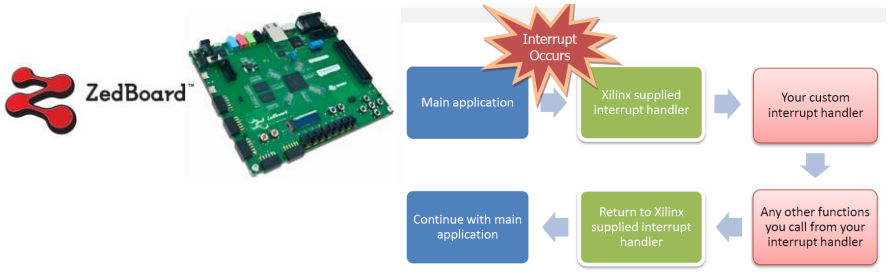
ECE3622 Embedded Systems Design Zynq Book Tutorials II

Zynq Processor System – Interrupt Timers.

```
// Print a message to the UART
xil_print("If we see this message, then we've out of the while loop\n\r");

// Disable interrupts in the Processor.
Xil_ExceptionDisable();

// Disconnect the interrupt for the Timer.
XScuGic_Disconnect(&my_Gic, XPAR_SCUTIMER_INTR);

cleanup_platform();

return 0;
}
```

Zynq Processor System – Interrupt Timers.

```
static void my_timer_interrupt_handler(void *CallBackRef)
{
        // The Xilinx drivers automatically pass an instance of
        // the peripheral which generated in the interrupt into this
        // function, using the special parameter called CallBackRef.
        // Locally declare an instance of the timer, and assign it to CallBackRef.

        XScuTimer *my_Timer_LOCAL = (XScuTimer *) CallBackRef;
```



Zynq Processor System – Interrupt Timers.

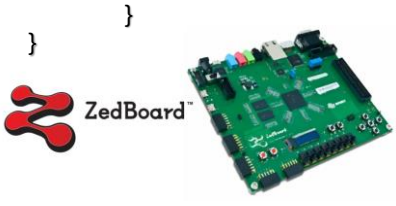
```
        // Check to see if the timer counter has expired. This is an example of
        // how a callback reference can be used as a pointer to the instance of
        // the timer that expired. Even if there were two timers the same
        // handler for both can be used and the CallBackRef would indicate
        // which generated the interrupt

        if (XScuTimer_IsExpired(my_Timer_LOCAL))
        {
                // Clear the interrupt flag in the timer, so that the same
                // interrupt is not serviced twice
                XScuTimer_ClearInterruptStatus(my_Timer_LOCAL);

                // Increment a counter global variable for
                // how many interrupts have been
                // generated.
                InterruptCounter++;
```

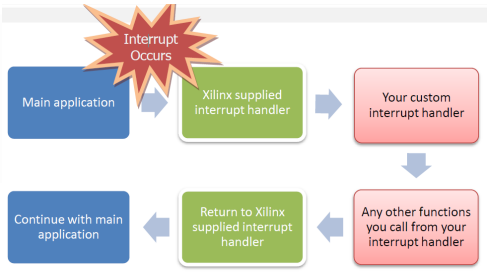
Zynq Processor System – Interrupt Timers.

```
// Update the value of the variable that stores the flashing LED state
        if (Flashing_LED_state > 0)
        {
            Flashing_LED_state = 0;
        }
        else
        {
            Flashing_LED_state++;
        }

        // Print to the UART to show that the interrupt handler is active
        Xil_print("\n\r** This message comes from the interrupt
            handler! (%d) **\n\r\n\n\r",
            InterruptCounter);
```

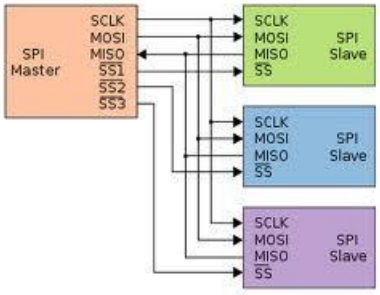
Zynq Processor System – Interrupt Timers.

```
// Check to see if we've had more than the defined number of interrupts
        if (InterruptCounter >=
            INTERRUPT_COUNT_TIMEOUT_VALUE)
        {
            // Stop the timer from automatically re-loading, so
            // that there are no more interrupts using CallBackRef
            XScuTimer_DisableAutoReload(my_Timer_LOCAL);
        }
    }
}
```



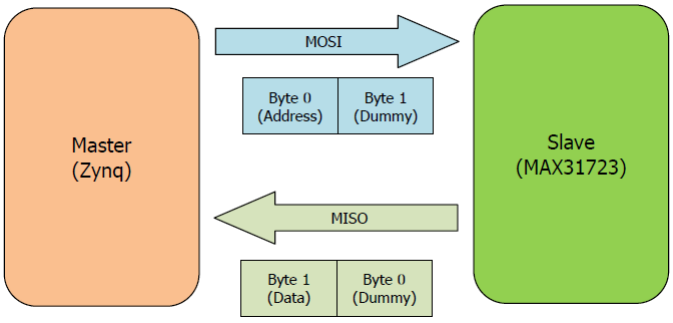


Zynq Processor System – Peripherals. A temperature sensing Pmod board from Maxim, the MAX31723PMB1 is interfaced to the Zynq. Although the MAX31723 can also be used in I2C mode, SPI is used here.

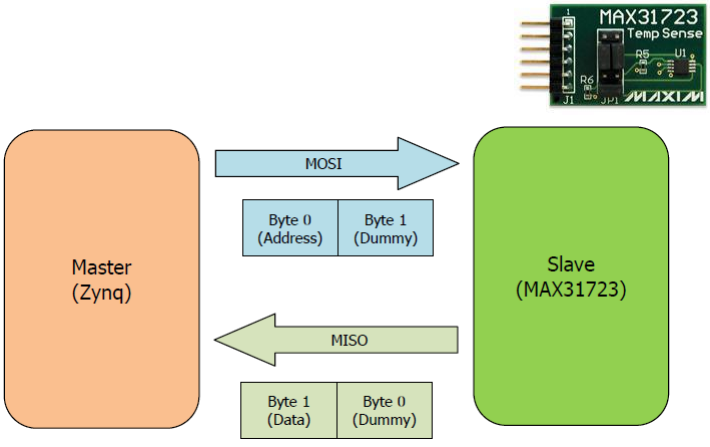


Zynq Processor System – Peripherals. The MAX31723 has a number of useful features, but only three of its internal registers are used.

Each register on the device has an address, which can be read or written by sending two bytes via SPI.

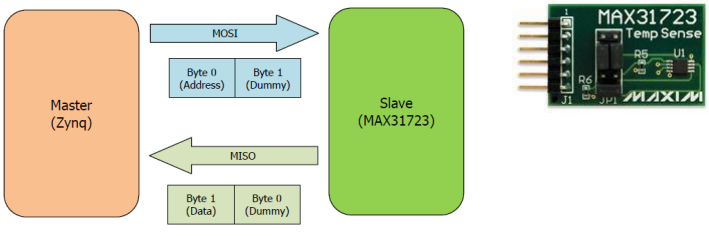


Zynq Processor System – Peripherals. During a write, the first byte contains the address of the register to be written and the second byte contains the register value.

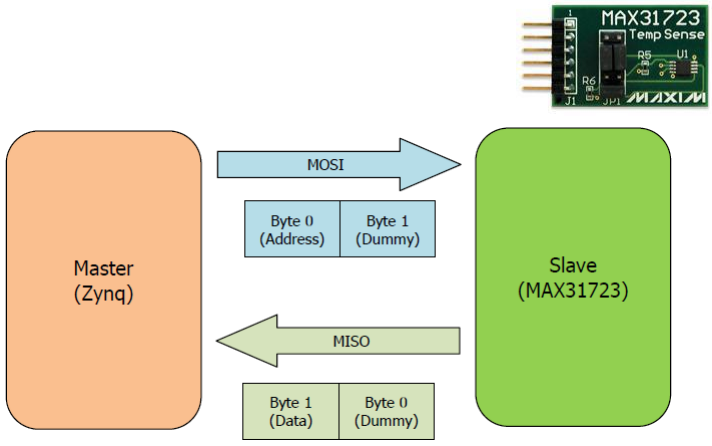


Zynq Processor System – Peripherals. During a read, the first byte contains the address of the register and the second byte is a dummy byte.

The MAX31723 decodes the address provided by the master in the first byte and ignores the second incoming byte. In return, it outputs a dummy value when it receives the first byte and sends the data from the register back during the second byte.



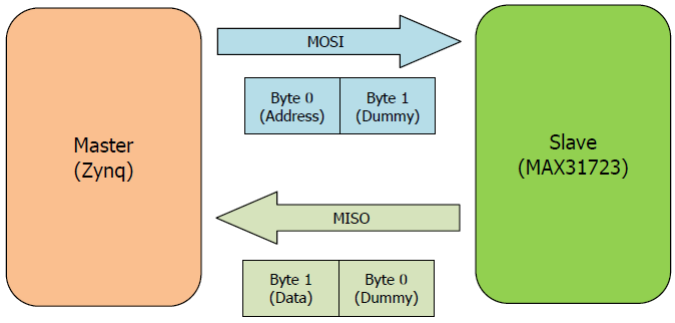
Zynq Processor System – Peripherals. Two registers from the MAX31723 requires four bytes to be transferred in total: two address bytes and two dummy bytes.



ECE3622 Embedded Systems Design Zynq Book Tutorials II

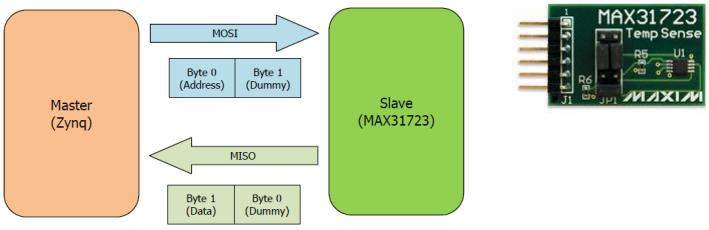
Zynq Processor System – Peripherals. The TEMPLSB register is at address 0x01 and the TEMPMSB register is at address 0x02.

Using the data from these two registers, it is possible to calculate the current temperature.



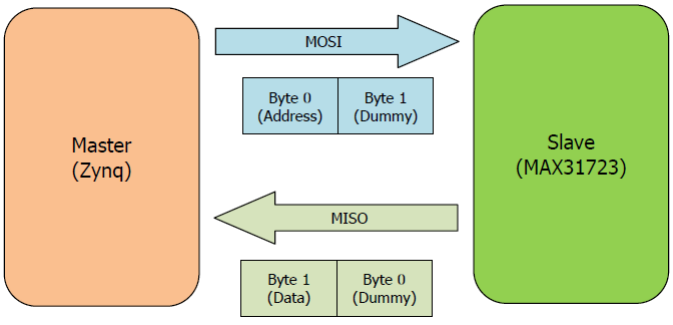
Zynq Processor System – Peripherals. The TEMPMSB byte represents the temperature in whole units (1 degree C, 2 degrees C... 100 degrees C, etc) in two's compliment binary.

The TEMPLSB represents fractions of a degree (the bits in the binary word represent ½ degree C, ¼ degree C, ⅛ degree C, etc).



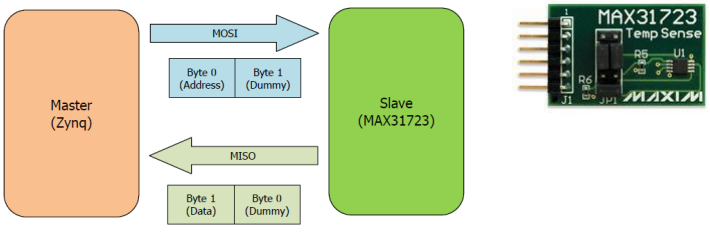
Zynq Processor System – Peripherals. Before reading the temperature registers, the device is enabled using the configuration register which is at address 0x80.

To get a basic temperature reading, clear all of the bits in the configuration register.

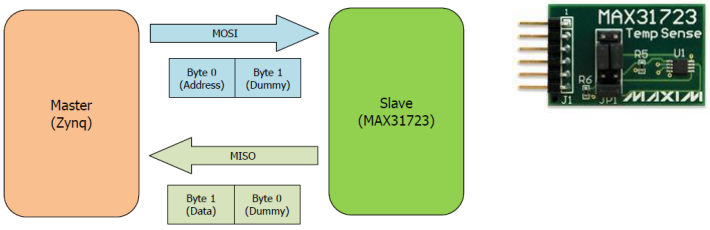


Zynq Processor System – Peripherals. The device is run in interrupt mode and sets up the driver and initializes the interrupts.

The process starts a transfer and then sets up a global variable and a while loop that halts execution until the SPI transfer has completed.



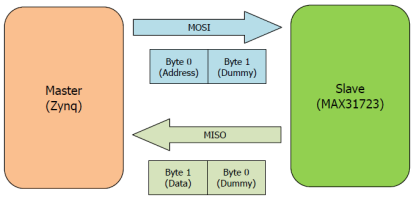
Zynq Processor System – Peripherals. The SPI peripheral can generate an interrupt for a number of reasons such as completion of a transfer and error conditions, and the interrupt handler checks to see what event caused the interrupt. The interrupt handler is thus more complicated.



Zynq Processor System – Peripherals. If the interrupt condition is the desired result or transfer complete then the handler sets the global variable back to the original value and the main section of the application continues.

Global variables are also used to implement two buffers for the SPI peripheral: a read buffer and a write buffer.

These buffers are then passed to the SPI drivers and are sent / received, byte by byte, during an SPI transfer.



Zynq Processor System – Peripherals.

```
#include "xparameters.h"
#include "xscugic.h"
#include "xil_exception.h"
#include <stdio.h>
#include "xspi.h"              /* SPI device driver */
#include "math.h"

#define MAX31723_CONFIG_REG_ADDRESS 0x80
#define MAX31723_TEMP_LSB_REG_ADDRESS 0x01
#define MAX31723_TEMP_MSB_REG_ADDRESS 0x02

#define MAX31723_THERMOMETER_RESOLUTION_9BIT_MODE 0x00
#define MAX31723_THERMOMETER_RESOLUTION_10BIT_MODE 0x02
#define MAX31723_THERMOMETER_RESOLUTION_11BIT_MODE 0x04
#define MAX31723_THERMOMETER_RESOLUTION_12BIT_MODE 0x06
```

Zynq Processor System – Peripherals.

```
#define MAX31723_CONTINUOUS_TEMPERATURE_CONVERSION_MODE
        0x00
#define MAX31723_COMPARATOR_MODE 0x00
#define MAX31723_DISABLE_ONE_SHOT_TEMPERATURE_CONVERSION
        0x0

// Define the temperature calibration offset here
#define TEMPERATURE_CALIBRATION_OFFSET -3.30

//  This is the size of the buffer to be transmitted/received
#define BUFFER_SIZE 4

//The following data type is used to send and receive data on the SPI interface.

typedef u8 DataBuffer[BUFFER_SIZE];
```

ECE3622 Embedded Systems Design Zynq Book Tutorials II

Zynq Processor System – Peripherals.

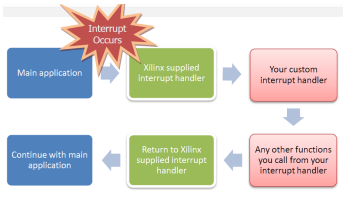
```
// Function Prototypes
void SpiIntrHandler(void *CallBackRef, u32 StatusEvent, u32 ByteCount);
void display_buffers(void);
void clear_SPI_buffers(void);
float read_current_temperature(XSpi *SpiInstance);

// Variable Definitions
// The following variables are shared between non-interrupt processing and
// interrupt processing must be global
volatile int SPI_TransferInProgress;
int SPI_Error_Count;

//The following variables are used to read and write to the SPI device, they
// are global to avoid having large buffers on the stack.
u8 ReadBuffer[BUFFER_SIZE];
u8 WriteBuffer[BUFFER_SIZE];
```

Zynq Processor System – Peripherals.

```
int main(void)
{
        XSpi_Config *SPI_ConfigPtr;
        XScuGic_Config *IntcConfig;
        XScuGic IntcInstance;       /* Interrupt Controller Instance */
        static XSpi SpiInstance;    /* The instance of the SPI device */

        int Status;
        float temperature;
        float previous_temperature;

        // Initialize the SPI driver
        SPI_ConfigPtr =
            XSpi_LookupConfig(XPAR_PS7_QSPI_0_DEVICE_ID);
        if (SPI_ConfigPtr == NULL) return XST_DEVICE_NOT_FOUND;
        Status = XSpi_CfgInitialize(&SpiInstance, SPI_ConfigPtr,
            SPI_ConfigPtr->BaseAddress);
        if (Status != XST_SUCCESS) return XST_FAILURE;
```


Zynq Processor System – Peripherals.

```
        // Reset the SPI peripheral
        XSpi_Reset(&SpiInstance);

        // Initialize the interrupt controller
        IntcConfig =
            XScuGic_LookupConfig(XPAR_SCUGIC_0_DEVICE_ID);
        if (NULL == IntcConfig) return XST_FAILURE;
        Status = XScuGic_CfgInitialize(&IntcInstance, IntcConfig,
            IntcConfig->CpuBaseAddress);
        if (Status != XST_SUCCESS) return XST_FAILURE;

        // Initialize exceptions on the ARM processor
        Xil_ExceptionInit();
```



Zynq Processor System – Peripherals.

```
// Connect the interrupt controller interrupt handler to the hardware
// interrupt handling logic in the processor.
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT,
    (Xil_ExceptionHandler)XScuGic_InterruptHandler, &IntcInstance);

// 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 = XScuGic_Connect(&IntcInstance,
    XPAR_FABRIC_AXI_QUAD_SPI_0_IP2INTC_IRPT_INTR,
    (Xil_ExceptionHandler)XSpi_InterruptHandler,
    (void *)&SpiInstance);
if (Status != XST_SUCCESS) return Status;
```

ECE3622 Embedded Systems Design Zynq Book Tutorials II

Zynq Processor System – Peripherals.

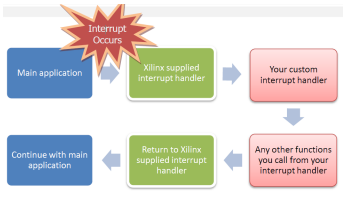
```
// Enable the interrupt for the SPI peripheral.
XScuGic_Enable(&IntcInstance,
    XPAR_FABRIC_AXI_QUAD_SPI_0_IP2INTC_IRPT_INTR);

// Enable interrupts in the Processor.
Xil_ExceptionEnable();

// Perform a self-test to ensure that the hardware was built correctly.
Status = XSpi_SelfTest(&SpiInstance);
if (Status != XST_SUCCESS) return XST_FAILURE;

xil_print("MAX31723PMB1 PMOD test\n\r\n\r");

// Run loopback test only in case of standard SPI mode.
if (SpiInstance.SpiMode != XSP_STANDARD_MODE) return
    XST_SUCCESS;
```

Zynq Processor System – Peripherals.

```
// Setup the handler for the SPI that will be called from the interrupt
// context when an SPI status occurs, specify a pointer to the SPI
// driver instance as the callback reference so the handler is able to
// access the instance data.
XSpi_SetStatusHandler(&SpiInstance, &SpiInstance,
    (XSpi_StatusHandler)SpiIntrHandler);

// Set the SPI device to the correct mode for this application
xil_print("Setting the SPI device into Master mode...");

Status = XSpi_SetOptions(&SpiInstance, XSP_MASTER_OPTION +
    XSP_MANUAL_SSELECT_OPTION +
    XSP_CLK_PHASE_1_OPTION);
if (Status != XST_SUCCESS) return XST_FAILURE;
xil_print("DONE!!\n\r");
```


Zynq Processor System – Peripherals.

```
// Select the SPI Slave.  This asserts the correct SS bit on the SPI bus
XSpi_SetSlaveSelect(&SpiInstance, 0x01);

// Start the SPI driver so that interrupts and the device are enabled.
printf("Starting the SPI driver, enabling interrupts and the device...");
XSpi_Start(&SpiInstance);
printf("DONE!!\n\r");

printf("Writing to the MAX31723 Config Register...");

// Clear the SPI read and write buffers
clear_SPI_buffers();
```



Zynq Processor System – Peripherals.

```
// Set the CONFIG register to a basic state
// Put the commands for the MAX31723 device in the write buffer
WriteBuffer[0] = MAX31723_CONFIG_REG_ADDRESS;
WriteBuffer[1] =
MAX31723_DISABLE_ONE_SHOT_TEMPERATURE_CONVERSION
    + MAX31723_COMPARATOR_MODE
    + MAX31723_THERMOMETER_RESOLUTION_12BIT_MODE +
MAX31723_CONTINUOUS_TEMPERATURE_CONVERSION_MODE;

// Transmit the data.
SPI_TransferInProgress = TRUE;
Status = XSpi_Transfer(&SpiInstance, WriteBuffer, NULL, 2);

while (SPI_TransferInProgress);  // Wait here until SPI has finished
xil_printf("DONE!\n\r\n\r\n\r");
```

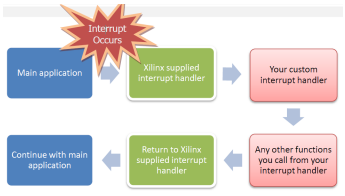
ECE3622 Embedded Systems Design Zynq Book Tutorials II

Zynq Processor System – Peripherals.

```
// Endless loop which reads and displays the current temperature
while(1)
{
        temperature = read_current_temperature(&SpiInstance);

        // Check to see if the temperature is different
        // Only update the display on the UART if it is different.
        if (previous_temperature != temperature)
        {
                xil_printf("Temperature = %3.4f  {Temperature
                    calibration offset = %3.4f}\n\r", temperature,
                    TEMPERATURE_CALIBRATION_OFFSET);
                previous_temperature = temperature;
        }
}
```

Zynq Processor System – Peripherals.

```
        // Disable and disconnect the interrupt system.
        XScuGic_Disconnect(&IntcInstance,
            XPAR_FABRIC_AXI_QUAD_SPI_0_IP2INTC_IRPT_INTR);

        return XST_SUCCESS;
}

void SpiIntrHandler(void *CallBackRef, u32 StatusEvent, u32 ByteCount)
{
        xil_print("** In the SPI Interrupt handler **\n\r");
        xil_print("Number of bytes transferred, as seen by the handler =
            %d\n\r", ByteCount);
```

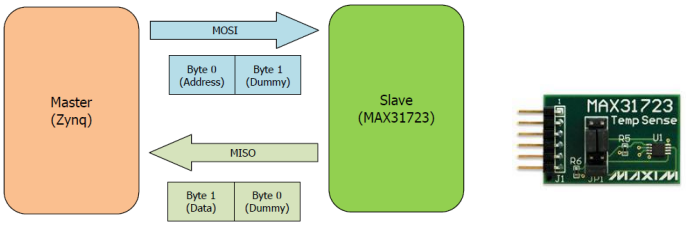


Zynq Processor System – Peripherals.

```
        // Indicate the transfer on the SPI bus is no longer in progress
        // regardless of the status event.
        if (StatusEvent == XST_SPI_TRANSFER_DONE)
        {
                SPI_TransferInProgress = FALSE;
        }
        else    // If the event was not transfer done, then track it as an error.
        {
                printf("\n\r\n\r ** SPI ERROR **\n\r\n\r");
                SPI_Error_Count++;
        }
}
```

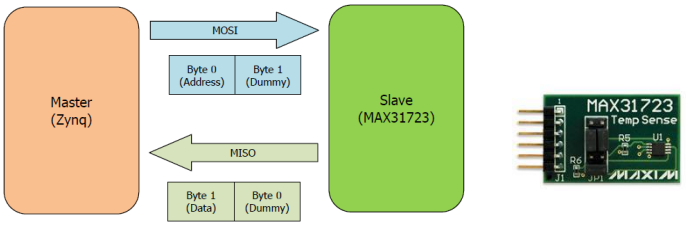
Zynq Processor System – Peripherals.

```
void display_buffers(void)
{
        int i;
        for(i=0; i<BUFFER_SIZE; i++)
        {
                xil_printf("Index 0x%02X  -->  Write = 0x%02X  |  Read =
                    0x%02X\n\r", i, WriteBuffer[i], ReadBuffer[i]);
        }
}
```



Zynq Processor System – Peripherals.

```
void clear_SPI_buffers(void)
{
        int SPI_Count;

        // Initialize the write buffer and read buffer to zero
        for (SPI_Count = 0; SPI_Count < BUFFER_SIZE; SPI_Count++)
        {
                WriteBuffer[SPI_Count] = 0;
                ReadBuffer[SPI_Count] = 0;
        }
```



Zynq Processor System – Peripherals.

```
float read_current_temperature(XSpi *SpiInstance)
{
        u8 Temperature_LSB = 0;
        u8 Temperature_MSB = 0;
        float Temperature_LSB_float = 0;
        float Temperature_MSB_float = 0;
        float Temperature_float = 0;
        int Status = 0;
        int i = 0;

        // Clear the SPI read and write buffers
        clear_SPI_buffers();

        // Put the commands for the MAX31723 device in the write buffer
        WriteBuffer[0] = MAX31723_TEMP_MSB_REG_ADDRESS;
        WriteBuffer[1] = 0x00000000;
```

ECE3622 Embedded Systems Design Zynq Book Tutorials II

Zynq Processor System – Peripherals.

```
        // Transmit the data.
        SPI_TransferInProgress = TRUE;
        Status = XSpi_Transfer(SpiInstance, WriteBuffer, ReadBuffer, 2);

        while (SPI_TransferInProgress);  // Wait here until SPI is finished

        // Fetch the byte of data from the ReadBuffer
        Temperature_MSB = ReadBuffer[1];

        // Clear the SPI read and write buffers
        clear_SPI_buffers();

        // Put the commands for the MAX31723 device in the write buffer
        WriteBuffer[0] = MAX31723_TEMP_LSB_REG_ADDRESS;
        WriteBuffer[1] = 0x00000000;
```

Zynq Processor System – Peripherals.

```
// Transmit the data
SPI_TransferInProgress = TRUE;
Status = XSpi_Transfer(SpiInstance, WriteBuffer, ReadBuffer, 2);
if (Status != XST_SUCCESS) return XST_FAILURE;

while (SPI_TransferInProgress);  // Wait here until SPI is finished

 // Fetch the byte of data from the ReadBuffer
Temperature_LSB = ReadBuffer[1];

if (Temperature_MSB & 0x80)  // If the sign bit is a '1'
{
        Temperature_LSB_float = (float)Temperature_LSB;

        Temperature_MSB = (~Temperature_MSB) + 1;
        Temperature_MSB_float = 0 - (float)Temperature_MSB;
        Temperature_LSB_float = 0;
```


Zynq Processor System – Peripherals.

```
        for (i=0; i<4; i++)
        {
                if (Temperature_LSB & (0x80 >> i))
                {
                        Temperature_LSB_float += 0.5 / pow(2, i);
                        // -lm switch must be added to the linker
                }
        }
}
else
{
        Temperature_LSB_float = (float)Temperature_LSB / 256;
        Temperature_MSB_float = (float)Temperature_MSB;
}
Temperature_float = Temperature_MSB_float +
  Temperature_LSB_float + TEMPERATURE_CALIBRATION_OFFSET;

return (Temperature_float);
}
```

End of Zynq Book Tutorials II