TP 0x01	Zynq BE		CHERIF Bilel
	PS Gpio, timer, and interrupts manipulation Guide		

PS GPIO

In order to use these functions in our application, we first need to include two header files 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.

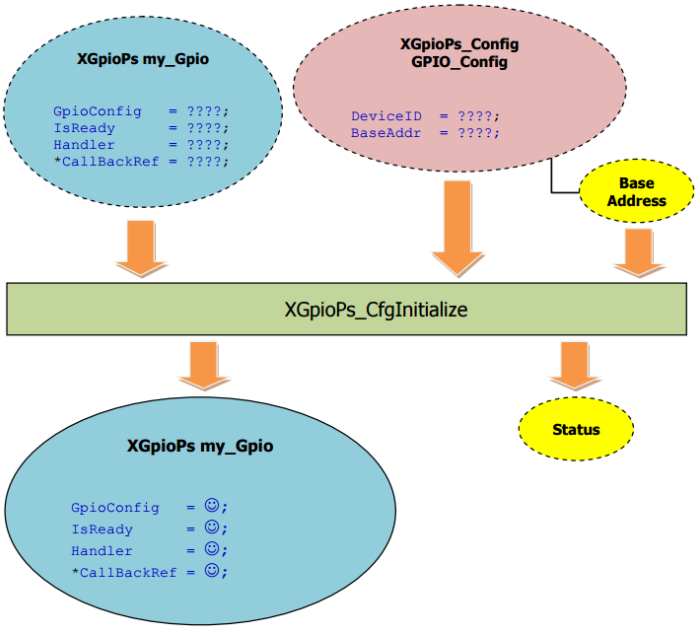
All Xilinx drivers use the same principles of operation and require that the driver be initialized before use. All Xilinx drivers have a struct (structure) which holds all of the various setup values which will be needed by the peripheral. A struct is merely a collection of variables / data types, wrapped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

XGpioPs my_Gpio

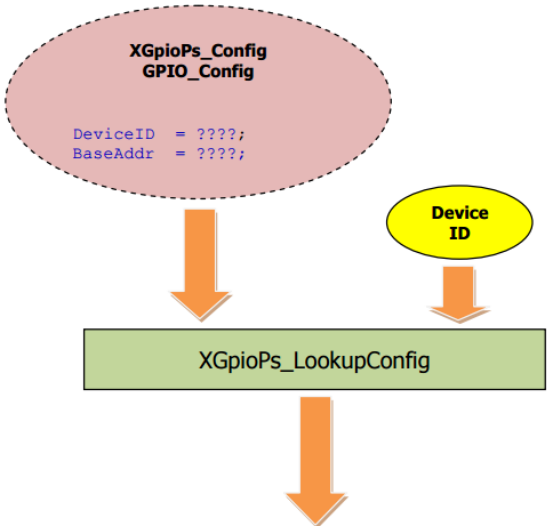
```
typedef struct {
	XGpioPs_Config GpioConfig;
	u32 IsReady;
	XGpioPs_Handler Handler;
	void *CallBackRef;
} XGpioPs;
```

When 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.
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 isactually another struct used to configure the my_Gpio instance.
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

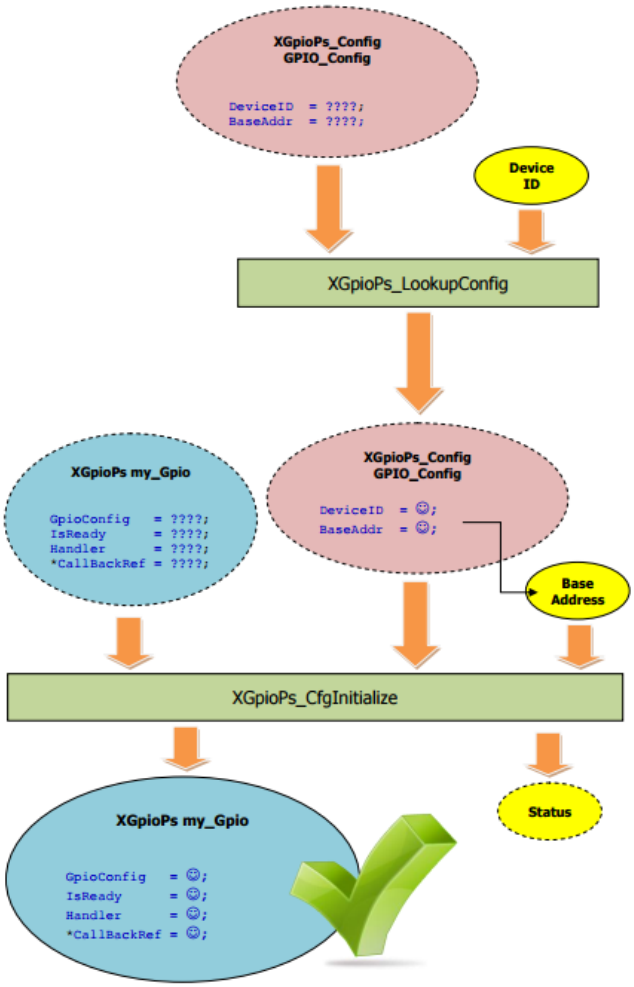
configures everything because all of the variables inside are uninitialized when the struct is declared.
The function requires three inputs: the instance of my_Gpio that was declared, the GPIO_Config struct, and a base address which can be extracted from the GPIO_Config struct.



The output of the XGpioPs_CfgInitialize function is a status value which Indicates whether the
initialization was successful.
However, there is a problem as the GPIO_Config struct hasn't been initialized. Another automated function called XGpioPs_LookupConfig provides the initialization.



The output of the XgpioPs_LookupConfig function is therefore fed into the XgpioPs_CfgInitialize function. The information required for this function is the Device ID, and that comes from the xparameters.h file.



XGpioPs_SetDirectionPin and the XGpioPs_WritePin functions send values to the various registers within the GPIO peripheral.

Use the following function to configure a pin and write a value to it:

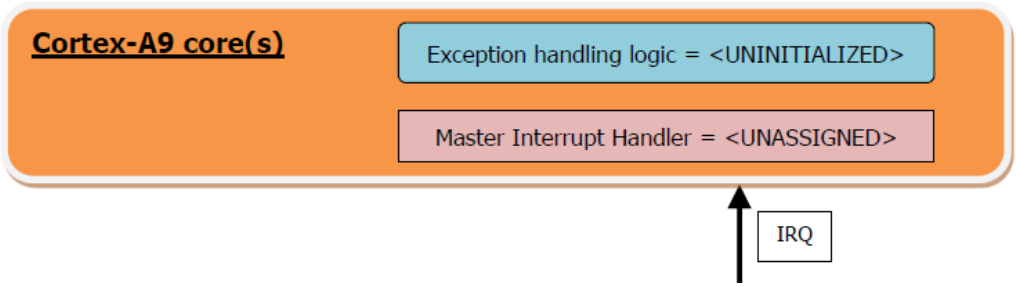
```
XGpioPs_SetDirectionPin(&Gpio, pin_Mio_Number, Direction);

XGpioPs_SetOutputEnablePin(&Gpio, Pin_Mio_Number, Enable_value);

XGpioPs_WritePin(&Gpio, pin_Mio_Number, output_value);
```

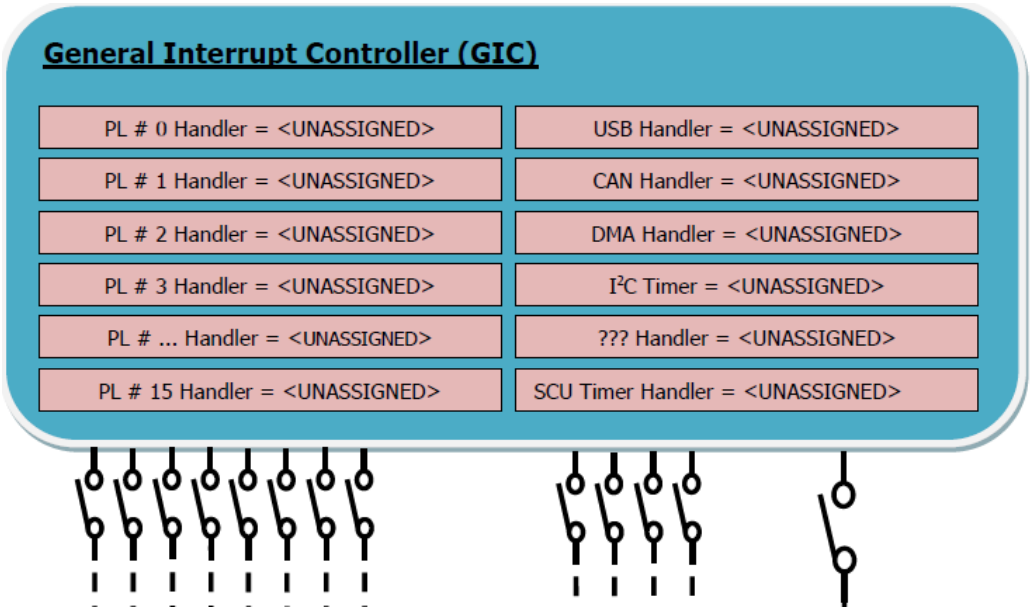
Interrupt Timers

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.

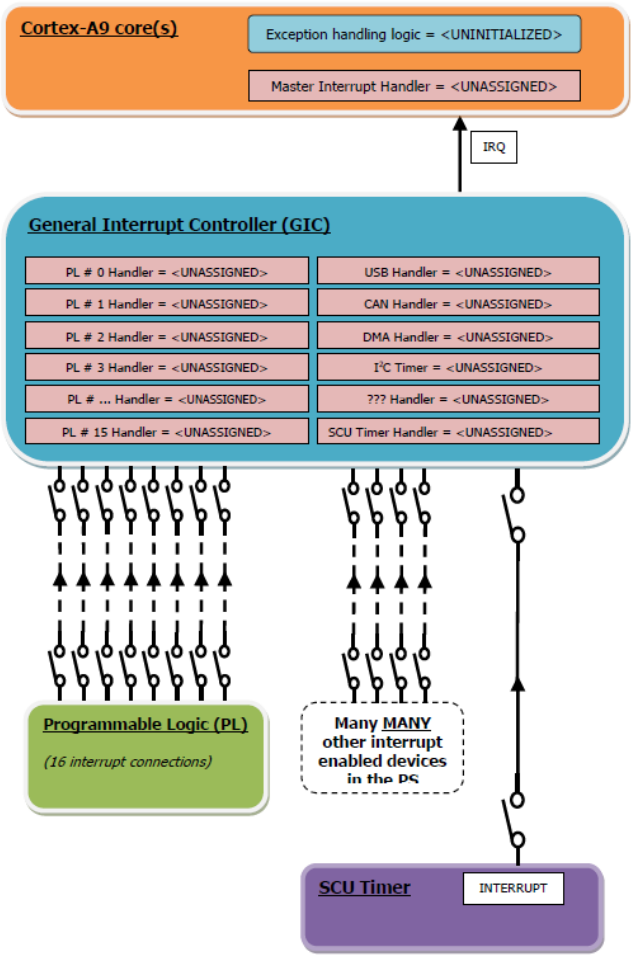


The General Interrupt Controller (GIC) has the ability to manage many interrupt inputs and has a table which allows interrupt handlers to be assigned to each incoming interrupt.

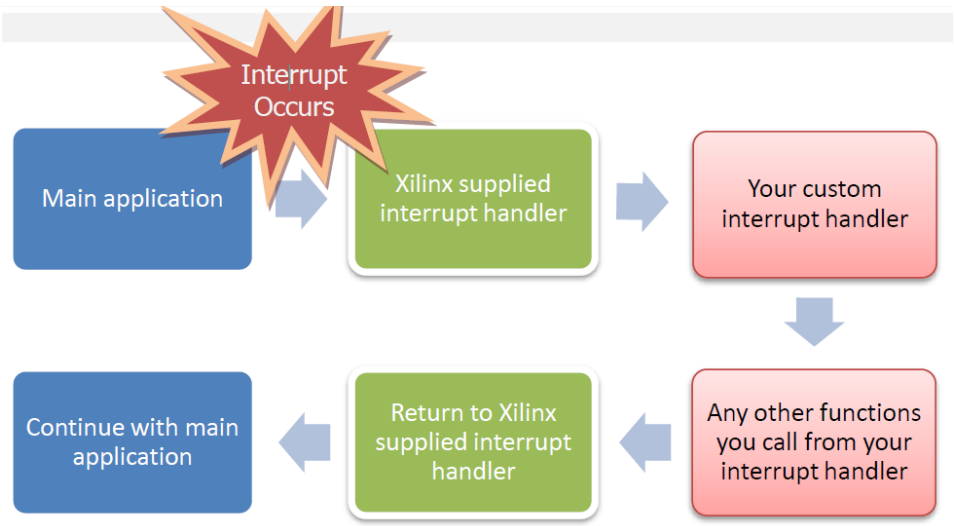
Each interrupt input on the GIC has an enable switch that is disabled by default.



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



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



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

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

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

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

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

```
Xil_ExceptionInit();
```

When an interrupt occurs, the processor first has to interrogate the interrupt controller to find out which peripheral generated the interrupt. Xilinx provide an interrupt handler to do this automatically,

and it is called "XScuGic_InterruptHandler". To use this supplied handler, we have to assign it to the interrupt controller. The syntax is pretty scary, but it's the same for all designs so it can just be copied and pasted for every design that you create. The only item that needs to be changed is the name of the GIC instance at the end of the function (in our case "&my_Gic").

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

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

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

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

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

Next, we need to enable the interrupt outputon the timer.

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

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

```
Xil_ExceptionEnable();
```

That's the setup completed for the interrupt controller and timer.

Now let's talk a little bit about our interrupt handler function that will be executed when the interrupt occurs.

The interrupt handler which handles the interrupts for the timer peripheral is connected to its unique interrupt ID number. When you setup

the timer interrupt handler using the following function:

```
  //set up the timer interrupt
            XScuGic_Connect(GicInstancePtr, TimerIntrId,
                                  (Xil_ExceptionHandler)TimerIntrHandler,
                                  (void *)TimerInstancePtr);
```

The TimerIntrHandler is the function that will be executed when the interrupt occurred. You should also define it in your C code.
Use the following prototype:

```
static void TimerIntrHandler(void *CallBackRef);
```

inside the interrupt handler function we should clear our interrupt status register to get back to our main program when interrupt handling function finishes it's task by using the CallBackRef passed to our hundler function.

```
      XScuTimer *TimerInstancePtr = (XScuTimer *) CallBackRef;
      XscuTimer_ClearInterruptStatus(TimerInstancePtr);
```

Now all you have to do is to test your c code and enjoy your application :) .