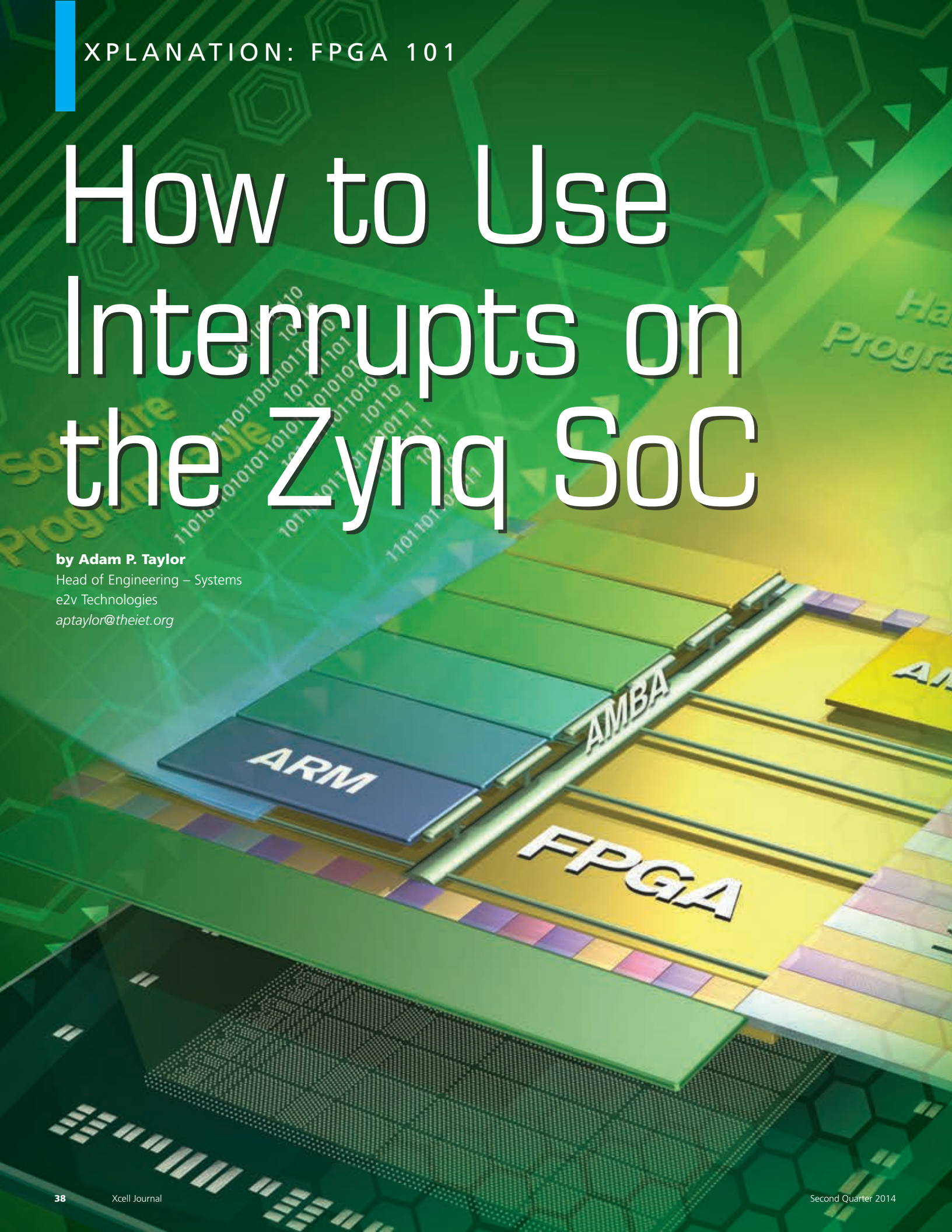
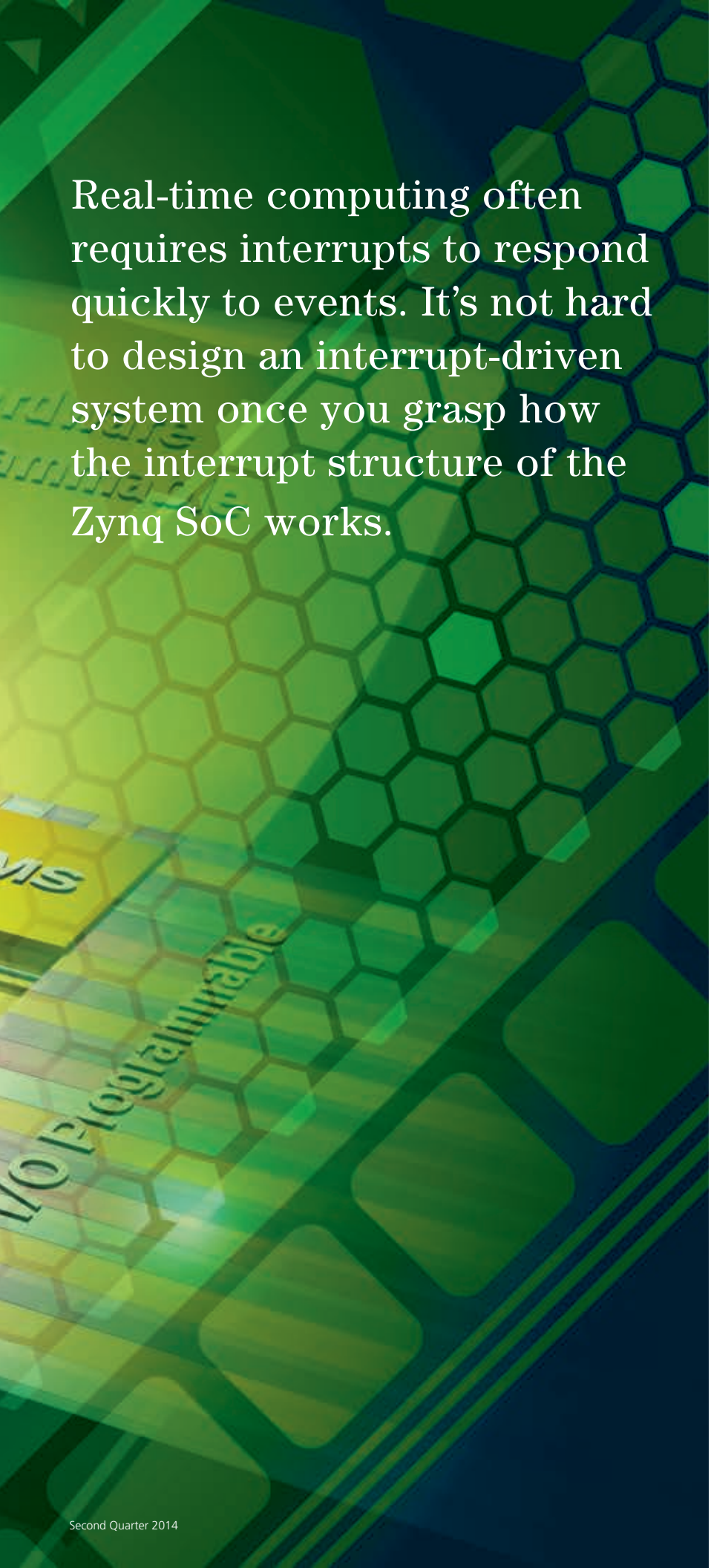
How to Use Interrupts on the Zynq SoC

by Adam P. Taylor
Head of Engineering – Systems
e2v Technologies
aptaylor@theiet.org





Real-time computing often requires interrupts to respond quickly to events. It's not hard to design an interrupt-driven system once you grasp how the interrupt structure of the Zynq SoC works.

In embedded processing, an interrupt is a signal that temporarily halts the processor's current activities. The processor saves its current state and executes an interrupt service routine to address the reason for the interrupt. An interrupt can come from one of the three following places:

- Hardware – An electronic signal connected directly to the processor
- Software – A software instruction loaded by the processor
- Exception – An exception generated by the processor when an error or exceptional event occurs

Regardless of the source, interrupts can also be classified as either maskable or non-maskable. You can safely ignore a maskable interrupt by setting the appropriate bit in an interrupt mask register. But you cannot ignore a non-maskable interrupt, because these are the types typically used for timers and watchdogs.

Interrupts can be either edge triggered or level triggered. The Xilinx® Zynq®-7000 All Programmable SoC supports configuration of the interrupt either way, as we will see later.

WHY USE AN INTERRUPT-DRIVEN APPROACH?

Real-time designs often require an interrupt-driven approach simply because many systems will have a number of inputs (for example keyboards, mice, pushbuttons, sensors and the like) that will at times require processing. Inputs from these devices are generally asynchronous to the process or task currently executing, so you cannot always predict when the event will occur.

Using interrupts enables the processor to continue processing until an event occurs, at which time the processor can address the event. This interrupt-driven approach also enables a faster response time to events than a polled approach, in which a program actively samples the status of an external device in a synchronous manner.

THE ZYNQ SOC'S INTERRUPT STRUCTURE

As processors get more advanced, there are a number of sources interrupts can come from. The Zynq SoC uses a Generic Interrupt Con-

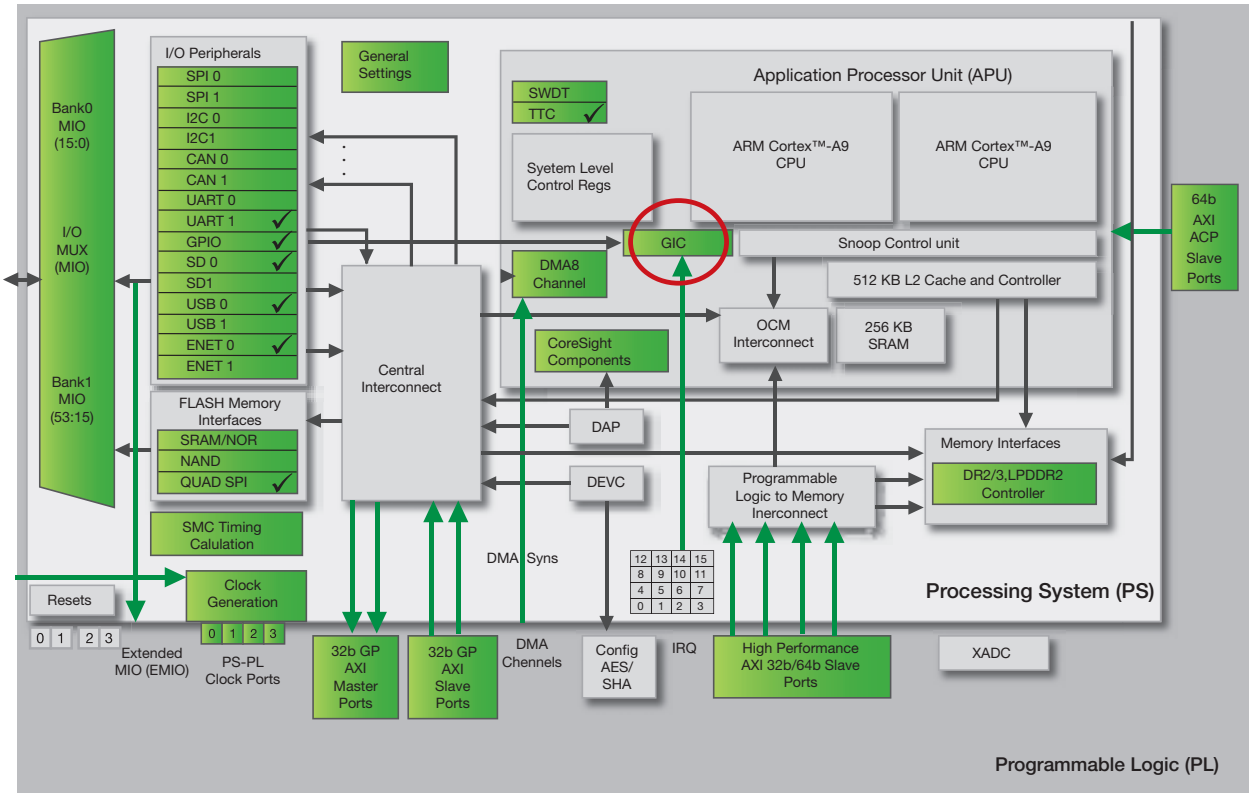


Figure 1 – The Generic Interrupt Controller is circled in red.

troller (GIC), as shown in Figure 1, to process interrupts. The GIC handles interrupts from the following sources:

- Software-generated interrupts – There are 16 such interrupts for each processor. They can interrupt one or both of the Zynq SoC's ARM® Cortex™-A9 processor cores.
- Shared peripheral interrupts – Numbering 60 in total, these interrupts can come from the I/O peripherals, or to and from the programmable logic (PL) side of the device. They are shared between the Zynq SoC's two CPUs.
- Private peripheral interrupts – The five interrupts in this category are private to each CPU—for example CPU timer, CPU watchdog timer and dedicated PL-to-CPU interrupt.

The shared peripheral interrupts are very interesting, as they are very flexible. They can be routed to either CPU from the I/O peripherals (44 interrupts in total) or from the FPGA logic (16 interrupts in total). However, it is also possible to route interrupts from the I/O peripherals to the programmable logic side of the device, as shown in Figure 2.

PROCESSING THE INTERRUPTS ON THE ZYNQ SOC

When an interrupt occurs within the Zynq SoC, the processor will take the following actions:

1. The interrupt is shown as pending.
2. The processor stops executing the current thread.
3. The processor saves the state of the thread in the stack to allow processing to continue once it has handled the interrupt.
4. The processor executes the interrupt service routine, which defines how the interrupt is to be handled.
5. The processor resumes operation of the interrupted thread after restoring it from the stack.

Because interrupts are asynchronous events, it is possible for multiple interrupts to occur at the same time. To address this issue, the processor prioritizes interrupts such that it can service the highest-priority interrupt pending first.

To implement this interrupt structure correctly, we will need to write two functions: an interrupt service routine to define the actions that will take place when the interrupt occurs, and an interrupt setup to configure the interrupt. The interrupt setup is a reusable routine that allows for constructing different interrupts. Generic for all interrupts within a system, the routine will set up and enable the interrupts for the general-purpose I/O (GPIO).

USING INTERRUPTS IN SDK

Interrupts are supported and can be implemented on a bare-metal system using the standalone board support package (BSP) within the Xilinx Software Development Kit

(SDK). The BSP contains a number of functions that greatly ease this task of creating an interrupt-driven system. They are provided within the following header files:

- Xparameters.h – This file contains the processor's address space and the device IDs.
- Xscugic.h – This file holds the drivers for the configuration and use of the GIC.
- Xil_exception.h – This file contains exception functions for the Cortex-A9.

To address a hardware peripheral, we need to know the address range and the device ID for the devices we wish to use—in other words, the GIC, which is provided mostly within the BSP header file xparameters. However, the interrupt ID is provided from xparameters_ps.h (there is no need to declare this header file within your source code as it is included in the xparameters.h file). We can use this interrupt labeled "ID" (it's the GPIO_Interrupt_ID) within our source file as shown below:

```
#define GPIO_DEVICE_ID    XPAR_XGPIOPS_0_DEVICE_ID
#define INTC_DEVICE_ID  XPAR_SCUGIC_SINGLE_DEVICE_ID
#define GPIO_INTERRUPT_ID    XPS_GPIO_INT_ID
```

For this simple example, we will be configuring the Zynq SoC's GPIO to generate an interrupt following a button push. To set up the interrupt, we will need two static global variables and the interrupt ID defined above to make the following:

```
static XScuGic Intc; // Interrupt Controller Driver
static XGpioPs Gpio; //GPIO Device
```

Within the interrupt setup function, we will need to initialize the Zynq SoC's exceptions; configure and initialize the GIC; and connect the GIC to the interrupt-handling hardware. The Xil_exception.h and Xscugic.h files provide the functions we need to accomplish this task. The result is the following code:

```
//GIC config
XScuGic_Config *IntcConfig;
Xil_ExceptionInit();

//initialize the GIC
IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);

XScuGic_CfgInitialize(GicInstancePtr, IntcConfig,
IntcConfig->CpuBaseAddress);

//connect to the hardware
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IN-
T,(Xil_ExceptionHandler)XScuGic_InterruptHandler,
      GicInstancePtr);
```

When it comes to configuring the GPIO to function as an interrupt within the same interrupt configuration routine,

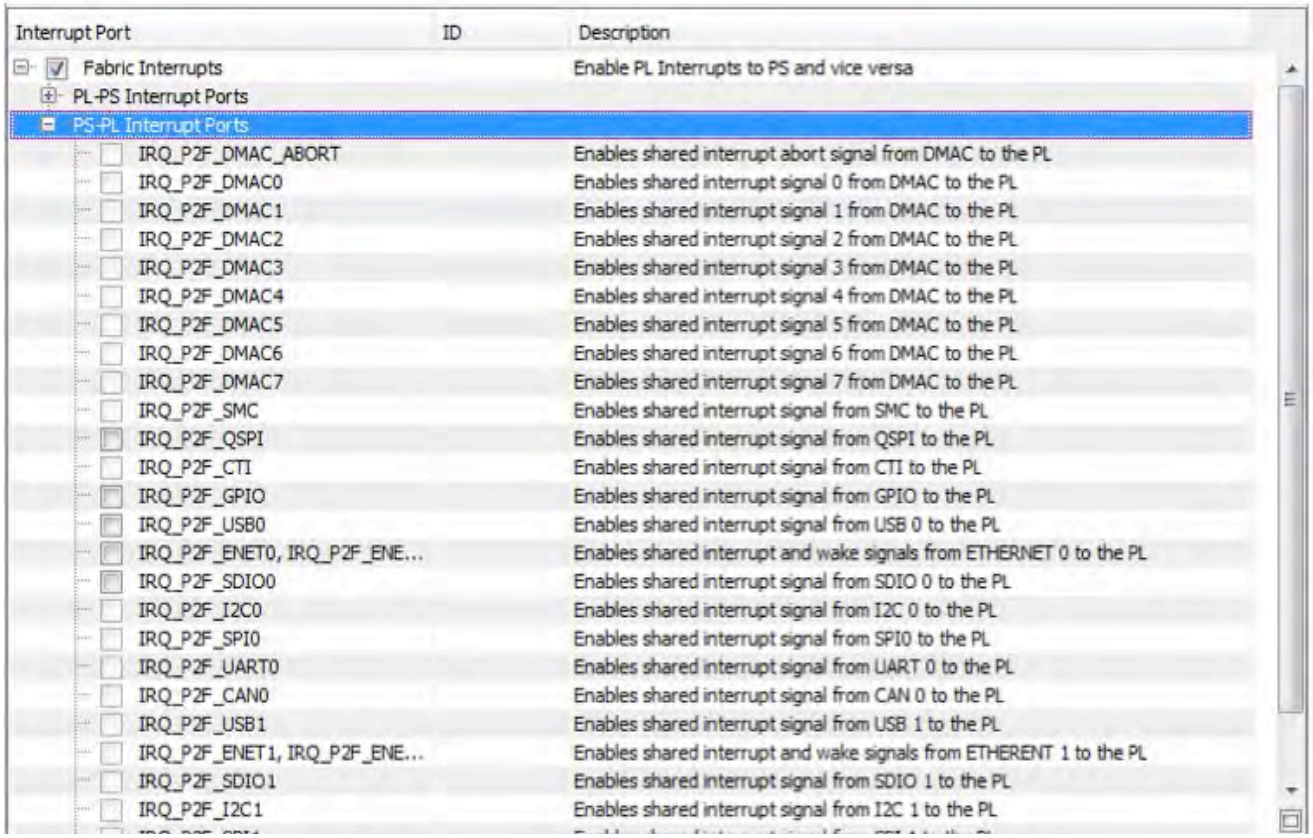


Figure 2 – These are the interrupts available between the processing system and the programmable logic.

we can configure either a bank or an individual pin. This task can be achieved using functions provided within xgpiops.h, for example:

```
void XGpioPs_IntrEnable(XGpioPs *InstancePtr, u8
   Bank, u32 Mask);
void XGpioPs_IntrEnablePin(XGpioPs *InstancePtr,
   int Pin);
```

Naturally, you will also need to configure the interrupt correctly. For instance, do you wish it to be edge triggered or level triggered? If so, which edge and level can be achieved using the function?

```
void XGpioPs_SetIntrTypePin(XGpioPs *InstancePtr,
int Pin, u8 IrqType);
```

where the IrqType is defined by one of the five definitions within xgpiops.h. They are:

```
#define XGPIOPS_IRQ_TYPE_EDGE_RISING 0 /**<
    Interrupt on Rising edge */
#define XGPIOPS_IRQ_TYPE_EDGE_FALLING 1 /**<
    Interrupt Falling edge */
#define XGPIOPS_IRQ_TYPE_EDGE_BOTH 2 /**<
    Interrupt on both edges */
#define XGPIOPS_IRQ_TYPE_LEVEL_HIGH 3 /**<
    Interrupt on high level */
#define XGPIOPS_IRQ_TYPE_LEVEL_LOW 4 /**<
    Interrupt on low level */
```

If you decide to use the bank enable, you need to know which bank the pin or pins you wish to enable interrupts are on. The Zynq SoC supports a maximum of 118 GPIOs. In this configuration, all of the MIOs (54 pins) are being used as GPIO along with the EMIOs (64 pins). We can break this configuration into four banks, with each bank containing up to 32 pins.

This setup function will also define the interrupt service routine, which is to be called when the interrupt occurs that uses the function:

```
XGpioPs_SetCallbackHandler(Gpio,
   (void *)Gpio, IntrHandler);
```

The interrupt service routine can be as simple or as complicated as the application defines. For this example, it will toggle the status of an LED on and off each time a button is pressed. The interrupt service routine will also print out a message to the console each time the button is pressed.

```
static void IntrHandler(void *CallBackRef, int
Bank, u32 Status)
{
       int delay;
         XGpioPs *Gpioint = (XGpioPs *)
         CallBackRef;
         XGpioPs_IntrClearPin(Gpioint, pbsw);
       printf("****button pressed****\n\r");
       toggle = !toggle;
       XGpioPs_WritePin(Gpioint, ledpin, toggle);
       for( delay = 0; delay < LED_DELAY; delay++)
       //wait
       {}
}
```

PRIVATE TIMER EXAMPLE

The Zynq SoC has a number of timers and watchdogs available. These are either private to a CPU or a shared resource available to both CPUs. Interrupts are required if you are to use these components efficiently in your design. The timers and watchdogs include the following:

- CPU 32-bit timer (SCUTIMER), clocked at half the CPU frequency
- CPU 32-bit watchdog (SCUWDT), clocked at half the CPU frequency
- Shared 64-bit global timer (GT), clocked at half the CPU frequency (each CPU has its own 64-bit comparator; it is used with the GT, which drives a private interrupt for each CPU)
- System watchdog timer (WDT), which can be clocked from the CPU clock or an external source
- A pair of triple timer counters (TTCs), each containing three independent timers. The TTCs can be clocked by the CPU clock or by means of an external source from the MIO or EMIO in the programmable logic.

To gain the maximum benefit from the available timers and watchdogs, we need to be able to make use of the Zynq SoC's interrupts. The simplest of these to configure is the private timer. Like most of the Zynq SoC's peripherals, this timer comes with a number of predefined functions and macros to help you use the resource efficiently. They are contained within the following:

```
#include "xscutimer.h"
```

This file contains functions (macros) that will provide a number of capabilities, including initialization and self-test. The functions within this file will also start and stop the timer, and manage the timer (restart it; check to see if it has expired; load the timer; enable/disable auto loading). Another of their jobs is to set up, enable, disable, clear and manage the timer interrupts. Finally, these functions also get and then set the prescaler.

The timer itself is controlled via the following four registers:

- Private Timer Load Register – This register is used in auto reload mode. It contains the value that is reloaded into the Private Timer Counter Register when auto reload is enabled.
- Private Timer Counter Register – This is the actual counter itself. When enabled, once this register reaches zero the interrupt event flag is set.
- Private Timer Control Register – The control register enables or disables the timer, auto reload mode and interrupt generation. It also contains the prescaler for the timer.

- Private Timer Interrupt Status Register – This register contains the private timer interrupt status event flag.

As for using the GPIO, the timer device ID and timer interrupt ID that are needed to set up the timer are contained within the XParameters.h file. Our example will use the pushbutton interrupt that we developed previously. When the button is pressed, the timer will load and start to run (not in auto reload mode). Upon expiration of the timer, an interrupt will be generated that will write a message out over the STDOUT. The interrupt will then be cleared to wait until the next time the button is pressed. This example will always load the same value into the counter; hence with the declarations at the top of the file, the timer count value is declared, as follows:

```
#define TIMER_LOAD_VALUE   0xFFFFFFFF
```

The next stage is to configure and initialize the private timer and load the timer count value into it.

```
 //timer initialisation
 TMRConfigPtr = XScuTimer_LookupConfig
   (TIMER_DEVICE_ID);
 XScuTimer_CfgInitialize(&Timer,
   TMRConfigPtr,TMRConfigPtr->BaseAddr);
//load the timer
 XScuTimer_LoadTimer(&Timer, TIMER_LOAD_VALUE);
```

We also need to update the interrupt setup subroutine to connect the timer interrupts to the GIC and enable the timer interrupt.

```
 //set up the timer interrupt
  XScuGic_Connect(GicInstancePtr, TimerIntrId,
(Xil_ExceptionHandler)TimerIntrHandler,
    (void *)TimerInstancePtr);
  //enable the interrupt for the Timer at GIC
  XScuGic_Enable(GicInstancePtr, TimerIntrId);
  //enable interrupt on the timer
  XScuTimer_EnableInterrupt(TimerInstancePtr);
```

Where TimerIntrHandler is the name of the function that is called when the interrupt occurs, the timer interrupt must be enabled on the GIC and within the timer itself.

The timer interrupt service routine is very simple. All it does is to clear the pending interrupt and write out a message over the STDOUT, as follows:

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

  XScuTimer *TimerInstancePtr =
     (XScuTimer *) CallBackRef;
  XScuTimer_ClearInterruptStatus(TimerInstancePtr);
  printf("****Timer Event!!!!!!!!!!!!!!****\n\r");
```

With this action complete, the final thing to do is to modify the GPIO interrupt service routine to start the timer each time the button is pushed, as such:

```
 //load timer
 XScuTimer_LoadTimer(&Timer, TIMER_LOAD_VALUE);
 //start timer
 XScuTimer_Start(&Timer);
```

To do this we first load the timer value into the timer and then call the timer start function. Now we can again clear the pushbutton interrupt and resume processing, as seen in Figure 3.

Many engineers initially approach an interrupt-driven system design with trepidation. However, the Zynq SoC's architecture, with the Generic Interrupt Controller coupled with the drivers provided with the SDK, enables you to get an interrupt-driven system up and running very quickly and efficiently.

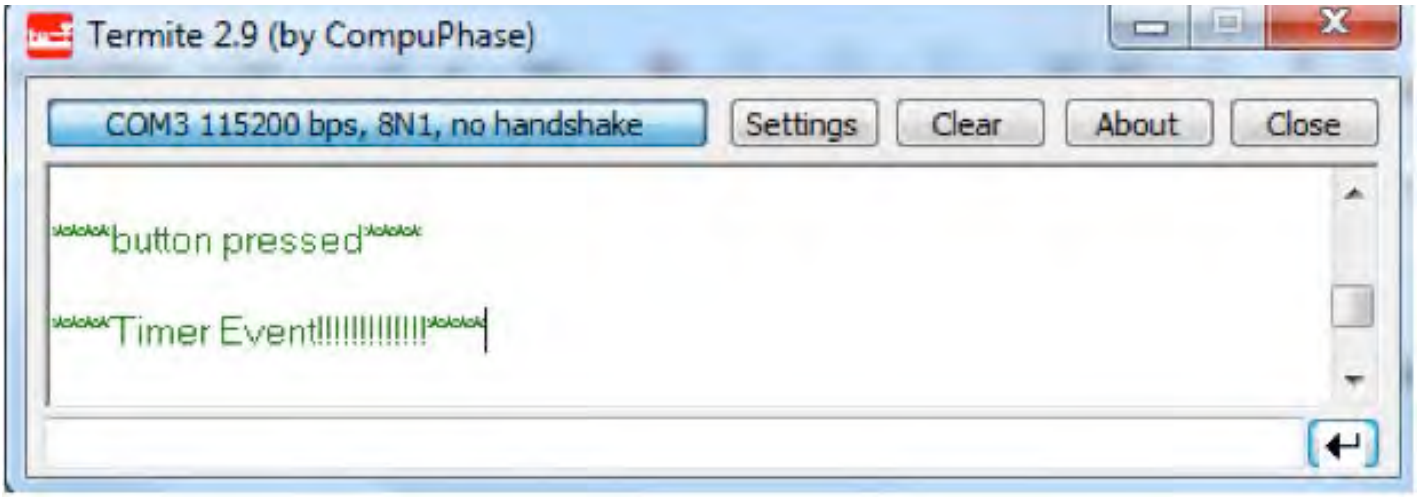


Figure 3 – This screen shows an example of the GPIO and timer interrupt event outputs.