手把手课堂：FPGA 101

如 何 在 Zynq SoC 上使用中断

作者：**Adam P. Taylor**
e2v 技术公司工程系统部负责人
aptaylor@theiet.org

实时计算经常要求中断针对事件快速做出响应。只要掌握 Zynq SoC 中断结构的工作原理，就不难设计出中断驱动型系统。

在嵌入式处理中，中断表示暂时停止处理器的当前活动。处理器会保存当前的状态并执行中断服务例程，以便对引起中断的原因进行寻址。中断可能来自下列三个地方之一：

- 硬件 – 直接连接处理器的电子信号
- 软件 – 处理器加载的软件说明
- 异常情况 – 发生错误或异常事件时处理器出现的异常情况

无论中断的来源在何处，都可将中断的类别归为可屏蔽和不可屏蔽两种。您可通过在中断掩码寄存器中设置相应的位来安全地忽略可屏蔽中断。但不能忽略不可屏蔽中断，因为这类中断通常用于定时器和看门狗监控器。

中断的触发既可以是边缘触发也可以是水平触发。我们将在后面部分看到，赛灵思 Zynq®-7000 All Programmable SoC 支持中断的这两种配置方式。

为什么使用中断驱动方案？

实时设计通常要求采用中断驱动方案，因为众多系统都会有很多输入单元（如键盘、鼠标、按钮、传感器以及类似设备等）偶尔需要处理。这些设备的输入单元通常会被异步至当前正在执行的进程或任务，因而用户不可能始终准确预测事件的发生时间。

使用中断，处理器能继续进行处理，直到事件发生，这时处理器便可处理这一事件。此外，与轮询方案相比，中断驱动方案对事件的响应时间更短，在中断驱动方案中，程序会以同步的方式主动对外部设备的状态进行采样。

Zynq SoC 的中断结构

随着处理器技术不断进步，中断的来源也多种多样。如图 1 所示，Zynq SoC 可使用通用中断控制器（GIC）来处理中断。GIC 可处理源自以下方面的中断：

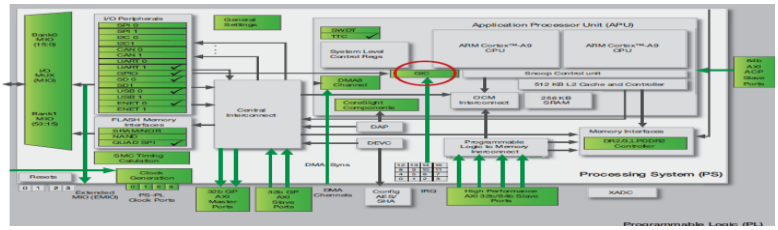

图 1 – 红圈显示的是通用中断控制器。

I/O 外设	通用设置	应用处理器单元（APU）	
系统级控制寄存器	窥探控制单元		64b AXI ACP 从系统端口
	512KB L2 高速缓存与控制器		
FLASH 存储器接口	中央互联	DMA8 通道 CoreSight 组件	存储器接口
			DR2/3，LPDDR2 控制器
	SMC 时序计算 Calulation	到存储器互联点的可编程逻辑	
复位	时钟生成		
扩展 MIO (EMIO)	PS-PL 时钟端口	32b GP AXI 主系统端口	32b GP AXI 从系统端口
DMA 同步 DMA 通道	配置 AES/ SHA	高性能 AXI 32b/64b 从系统端口	处理系统（PS） 可编程逻辑（PL）

- 软件生成的中断 – 每个处理器有 16 个此类中断，能够中断一个或两个 Zynq SoC 的 ARM®Cortex™-A9 处理器内核；
- 共享外设中断 – 共计 60 个，这些中断来自 I/O 外围设备，或往返于设备的可编程逻辑（PL）侧。Zynq SoC 的两个 CPU 共享这些中断；
- 专用外设中断 – 这种类型中包含的 5 个中断对每个 CPU 都属于专用中断，比如 CPU 定时器、CPU 看门狗监视器定时器以及专属 PL 至 CPU 中断。

共享外设中断非常有趣，因为它们非常灵活。可将它们从 I/O 外设（共 44 个中断）或 FPGA 逻辑（共 16 个中断）路由至两个 CPU 中的一个，但也可以将中断从 I/O 外设路由至设备的可编程逻辑侧，参见图 2。

在 Zynq SoC 上处理中断

在 Zynq SoC 中发生中断时，处理器会采取以下措施：

1. 将中断显示为挂起；
2. 处理器停止执行当前线程；
3. 处理器在协议栈中保存线程状态，以便在中断处理后继续进行处理；
4. 处理器执行中断服务例程，其中定义了如何处理中断；
5. 在处理器从协议栈恢复之前，被中断的线程继续运行；

中断属于异步事件，因此可能同时发生多个中断。为了解决这一问题，处理器会对中断进行优先级排序，从而首先服务于优先级别最高的中断挂起。

为了正确实现这一中断结构，需要编写两个函数：一是中断服务例程，用于定义中断发生时的应对措施；二是用于配置中断的中断设置。中断设置例程可重复使用，允许构建不同的中断。该例程适用于系统中的所有中断，将针对通用 I/O（GPIO）设置和使能中断。

如何在 SDK 中使用中断

可使用赛灵思软件开发套件（SDK）中的独立板支持包（BSP）在物理硬件上支持并实现中断。BSP 具备众多功能，可显著降低创建中断驱动系统的任务难度。它们位于带有以下报头的文件中：

- Xparameters.h – 该文件包含处理器的地址空间和设备 ID；
- Xscugic.h – 该文件包含配置驱动程序以及 GIC 的使用范围；
- Xil_exception.h – 该文件包含 Cortex-A9 的异常函数。

为了对硬件外设进行寻址，我们需要知道想要使用的设备（也就是 GIC）的地址范围和设备 ID，这些信息大多位于 BSP 报头文件 xparameters 下。但是 xparameters_ps.h（无需在您的源代码中申报该报头文件，因为它包含在 xparameters.h 文件中）提供了中断 ID。我们可在源文件中使用这个标记有中断的“ID”（GPIO_Interrupt_ID），使用方式如下：

```
#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
```

在这个简单的例子中，我们将配置 Zynq SoC 的 GPIO，以便在按下按钮后生成中断。为

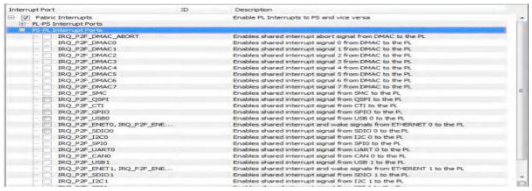

图 2 – 这些是处理系统与可编程逻辑之间可用的中断。

了设置中断，我们需要两个静态全局变量以及上述定义的中断 ID 来执行以下操作：

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

在中断设置功能中，我们需要初始化 Zynq SoC 异常；配置并初始化 GIC；并将 GIC 连接到中断处置硬件。Xil_exception.h 和 Xscugic.h 文件可提供完成这一任务所需的函数。结果生成以下代码：

```
//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);
```

当配置 GPIO 以使其在同一中断配置例程中发挥中断功能时，我们能够配置内存库或单个引脚。我们可通过使用在 xgpiops.h 中提供的函数来完成这项任务，比如：

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

当然，您还需要正确配置中断。例如，您希望采用边缘触发或水平触发吗？若答案为是，那么采用这个函数能实现何种边缘和水平呢？

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

在这里，xgpiops.h 中五个定义中的其中一个可对 IrqType 定义。这五个定义是：

```
#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 */
```

如果您决定使用 Bank 使能，那么您需要知道您希望使能中断的单个引脚或多引脚位于哪个 Bank 上。Zynq SoC 最多支持 118 个 GPIO。在这种配置下，所有 MIO（54 个引脚）都会与 EMIO（64 个引脚）一起被用作 GPIO。我们能将这个配置分为四个 Bank，每个 Bank 容纳 32 个引脚。

此外，这项设置功能还将定义中断服务例程，发生中断时，可用以下函数调用该例程：

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

中断服务例程的繁简程度由其应用定义。在该例中，每按一次按钮，它便会触发一个 LED，打开和关闭这个 LED。另外，在每次按下按钮时，中断服务例程还会向控制台打印一条信息。

```
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
        {}
}
```

专有定时器举例

Zynq SoC 拥有许多可用的定时器和看门狗监视器。它们既可作为一个 CPU 的专用资源也可作为两个 CPU 的共享资源。如需在您的设计中高效利用这些组件，则需要中断。这些定时器和看门狗监视器包括：

- CPU 32 位定时器（SCUTIMER），以 CPU 频率的一半计时
- CPU 32 位看门狗监视器（SCUWDT），以 CPU 频率的一半计时
- 共享 64 位全局定时器（GT），以 CPU 频率的一半计时（每个 CPU 都有其自己的 64 位比较器；它与 GT 配合使用，能驱动各个 CPU 的专用中断）
- 系统看门狗监视时钟（WDT），可通过 CPU 时钟或外部来源进行计时
- 一对三重定时器计数器（TTC），每个包含三个独立定时器。在可编程逻辑中，可通过 CPU 时钟或来自 MIO 或 EMIO 的外部来源对 TTC 进行计时。

为了通过可用的定时器和看门狗监视器获得最大优势，我们需要使用 Zynq SoC 中断。其中配置最简单的就是专有定时器。和 Zynq SoC 的大多数外设一样，该定时器带有很多预定义的函数和宏指令，能帮助您高效使用这一资源。这些函数和宏指令位于：

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

这个文件中的函数（宏指令）能够提供许多功能，包括初始化和自测试等。此外，文件中的函数还能启动和停止定时器并对其进行管理（重启；检查是否过期；加载定时器；使能/禁用自动加载）。它们的另一项工作就是设置、使能、禁用、清除和管理定时器中断。最后，这些函数还能获取并设置预分频器。

定时器本身通过以下四个寄存器来控制：

- 专用定时器加载寄存器 – 可将该寄存器用于自动重新加载模式，包含在使能自动重新加载时被重新加载到专用定时器计数器寄存器中的数值。
- 专用定时计数寄存器 (Private Timer Counter Register) – 这是真实计数器本身。使能后，一旦寄存器达到零，则会设置中断事件标志。

- 专用定时器控制寄存器 – 控制寄存器可使能或禁用定时器、自动重新加载模式以及中断生成，还包含定时器的预分频器。
- 专用定时器中断状态寄存器 – 该寄存器包含专用定时器中断状态事件标志。

就使用 GPIO 而言，设置定时器所需的定时器设备 ID 和定时器中断 ID 都包含在 XParameters.h 文件中。在本例中，我们将使用先前开发的按钮中断。当按下按钮时，定时器将加载并开始运行（采用非自动重新加载模式）。一旦定时器过期，将生成能通过 STDOUT 输出一条消息的中断。然后清除该中断，以便等待下一次按下按钮。在本例中，将始终向计数器加载相同的数值；因此，在文件顶部的公告中公布了定时器计数值，如下所示：

```
#define TIMER_LOAD_VALUE    0xFFFFFFFF
```

下一步是配置和初始化专用定时器并在其中加载定时器计数值。

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

此外，我们还需要更新中断设置子例程，从而将定时器中断连接至 GIC 并使能定时器中断。

```
//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);
```

发生中断时，需要调用 TimerIntrHandler 函数，这时必须在 GIC 上以及定时器本身使能定时器中断。

定时器中断服务例程非常简单。它仅需清除挂起的中断，并通过 STDOUT 输出一条消息，如下所示：

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

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

完成该操作后，最后还要修改 GPIO 中断服务例程，从而在每次按下按钮后启动定时器，如下所示：

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

首先，我们要将定时器值加载到定时器中，然后调用定时器启动函数。现在，我们能够再次清除按钮中断并恢复处理，如图 3 所示。

在开始着手设计中断驱动系统时，很多工程师都会心生畏惧。但是，Zynq SoC 架构以及通用中断控制器（与配备 SDK 的驱动器相结合）可帮助您快速、高效地启动和运行中断驱动系统。

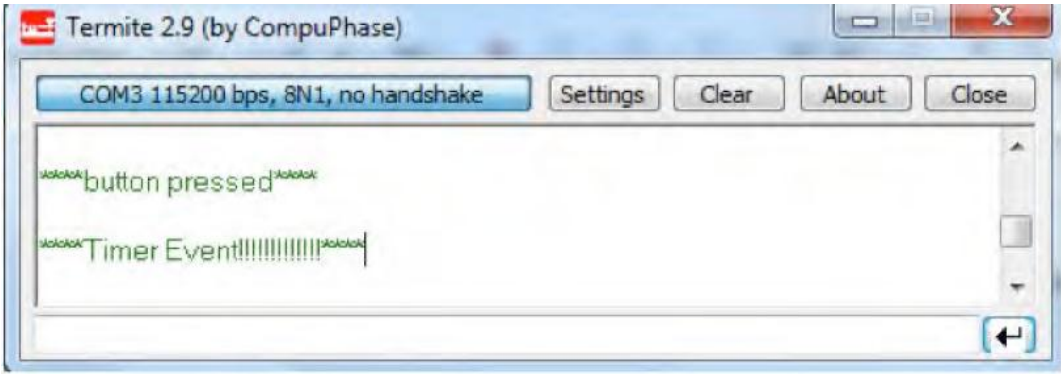

图 3 – GPIO 与定时器中断事件输出的界面示例。