AN54460

PSoC® 3 和 PSoC 5LP 中断

作者：Vivek Shankar Kannan

相关项目： 有

相关器件系列：所有 PSoC 3 和 PSoC 5LP 系列

软件版本: PSoC Creator™ 4.1 Update 1 以及更高版本

相关的应用笔记：AN90799、AN90833、AN60630 和 AN89610

要想获得该应用笔记的最新版本，或者相关联的项目文件，请访问 http://www.cypress.com/go/AN54460

AN54460 通过一些示例项目介绍了 PSoC® 3 和 PSoC 5LP 中的中断架构以及其在 PSoC Creator™ IDE 中的配置。本应用笔记还说明了高级的中断主题， 比如：如何处理重入函数、中断代码最优化、中断延迟以及调试技术。

目录

1 简介 1
2 PSoC 中断架构 2
2.1 PSoC 中断性能 2
3 PSoC Creator 的中断支持 4
3.1 中断组件配置 5
3.2 中断优先级配置 7
4 中断项目 7
4.1 简单的中断项目 7
4.2 PICU 中断项目 10
4.3 LVD 中断项目 15
4.4 SysTick 中断项目 19
5 高级中断主题 20
5.1 优化中断代码 20
5.2 中断组件 API 20
5.3 中断及其他组件 21
5.4 中断延迟 21
5.5 调试 22
5.6 不受支持的固定功能中断 22
5.7 强制中断向量编号 23
6 总结 24
6.1 项目汇总 24
A PSoC 中的中断源 25
B PSoC 3 中的可重入函数 27
B.1 Keil C51 编译器的可重入性 27
B.2 PSoC Creator 的可重入支持 27
B.3 确定可重入函数 29
文档修订记录 30
销售、解决方案以及法律信息 31

1 简介

中断是嵌入式应用的重要组成部分。它们使 CPU 不必连续轮询某个特定事件的发生，而是仅在该事件发生时才通知 CPU。在片上系统 (SoC) 的架构 (如 PSoC) 中，经常使用中断向 CPU 报告片上外设的状态。

AN54460 向您介绍了 PSoC 3 和 PSoC 5LP 的中断架构。它也展示了这些 PSoC 器件的开发工具，即 PSoC Creator IDE 如何支持中断处理。此外，本文档也详细解释了各个高级中断的概念。本应用笔记附带的一些示例项目用以说明各种中断的用例。

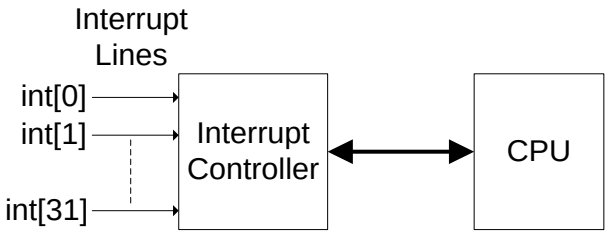
阅读本应用笔记之前，您应已了解 PSoC Creator IDE 的基本知识。如果您尚未了解 PSoC Creator，请参考 PSoC Creator 主页中的内容。为了更加了解 PSoC Creator， 您可以参考以下应用笔记 — AN54181 — PSoC 3 入门和 AN77759 — PSoC 5LP 入门 — 他们通过简单的项目分别介绍 PSoC 3 和 PSoC 5LP 器件以及 IDE 工具。

此外，还存在说明了其他 PSoC 产品系列的中断架构的应用笔记。请参阅 AN90799 — PSoC 4 中断和 AN90833 — PSoC 1 中断，它们分别是本应用笔记的 PSoC 4 和 PSoC 1 配对。

2 PSoC 中断架构

本部分提供了 PSoC 3 和 PSoC 5LP 中断架构的概述。图 1 显示的是简化的框图。

图 1. PSoC 3 和 PSoC 5LP 的中断架构



PSoC 3 和 PSoC 5LP 中包含 32 个中断信号线 (从 int[0]到 int[31])。每条中断信号线可配一个优先级 (从 0 到 7)，其中“0”表示最高的优先级。每个中断信号线都有一个中断向量地址，表示中断代码的起始地址。接收到某个中断请求后，CPU 将跳转至该地址。中断代码也被称为中断服务子程序 (ISR)。

中断控制器作为中断线与 CPU 之间的接口使用。将中断信号线的中断向量地址以及中断请求信号发送到 CPU。当进入或退出中断时，中断控制器也会接收来自 CPU 的确认信号。当多个中断信号线同时发出请求时，中断控制器会根据它们的优先级来决定先处理哪个中断信号线。

欲了解更多有关中断操作的详细信息，请参考 PSoC 3、 PSoC 5LP 的技术参考手册 (TRM)。

2.1 PSoC 中断性能

PSoC 3 和 PSoC 5LP 提供其他传统微控制器不支持的增强中断特性，包括：

- **可配置中断向量地址：**您可以使用 PSoC 来动态地配置中断向量的地址。发生中断时，CPU 可直接跳转至 ISR 代码执行中断处理。与传统的微控制器相比，这样可以降低 PSoC 的中断执行延迟。

  在传统的微控制器中，每个中断信号源的中断向量地址是固定的。通常，在此固定地址上要执行一条“JUMP”指令，以便 CPU 能够跳转至实际需要执行的实际 ISR 代码。

- **灵活的中断源：**在传统的微控制器中，中断源是硬连接到各个中断线的。在 PSoC 中，您可灵活选择每条中断信号线的中断源。这种灵活的架构允许配置任何数字信号作为中断源使用。

PSoC 器件同时支持电平触发和边沿触发的中断。通过中断源产生的中断信号可以确定中断类型是电平触发还是边沿触发。图 2 和图 3 显示的是电平触发和边沿触发的中断如何运行。

图 2. 电平触发中断

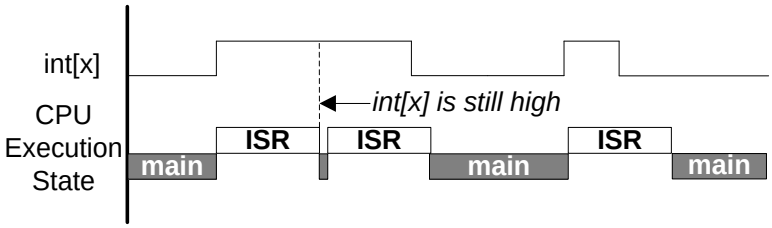
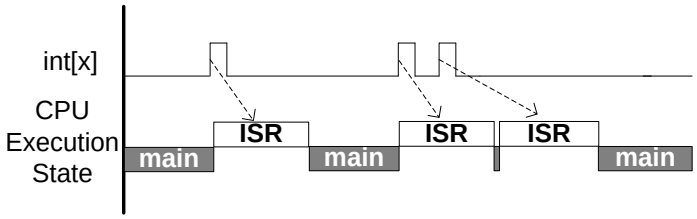


图 3. 边沿触发中断



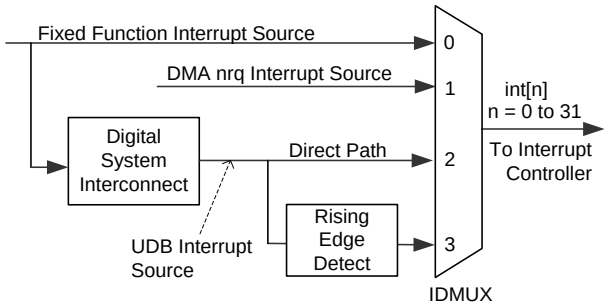
假设中断线最初无效 (逻辑“低”)，那么以下的事件序列将说明如何处理电平触发和边沿触发中断：

- 在中断线上的上升沿事件发生时，中断控制器将记录下中断请求。中断线目前处于挂起状态，它指的是其请求尚未被 CPU 处理的各中断。
- 然后，中断控制器将中断向量地址以及中断请求信号发送到 CPU。当 CPU 开始执行中断线的 ISR 时，将清除中断线的挂起状态。
- 对于电平触发的中断：如果中断信号线在执行完 ISR 后仍然是高电平，它会变成等待状态，并再次执行 ISR，如图 2 所示。只要中断线为高电平，会继续执行 ISR。
- 对于边沿触发中断：如果在执行 ISR 时，该中断线上出现了一个或多个上升沿，那么它们将被记录下为单一挂起的请求。执行完当前 ISR 后，中断控制器将继续处理挂起的中断，如图 3 所示。

边沿触发的中断都是脉冲信号，因此，一般被称为脉冲中断。每个边沿触发中断的最小脉冲宽度为一个总线时钟周期。PSoC 具有上升沿的检测逻辑，以确保该中断在上升沿上被触发一次。在后面内容中会更加详细地解释该性能。

每条中断线可由三个中断源中的一个驱动，如图 4 所示。复用器逻辑选择每条中断线的中断源。

图 4. PSoC 3 和 PSoC 5LP 的中断源



下面详细描述了各个中断源：

2.1.1 固定功能的中断源

这些是片上外设的一些预定义中断源。例如：来自固定功能模块的定时器、计数器、端口中断控制单元 (PICU) 中的各中断信号。欲了解 PSoC 3 和 PSoC 5LP 器件的中断源列表，请参阅附录 A。

几乎所有固定功能的中断源都是电平触发中断。对于这种中断，执行 ISR 时必须读取片上外设的状态寄存器，原因有两个：

1. 该状态寄存器会指出所生成中断的条件。例如，生成某个 PICU 中断时，PICU 状态寄存器的每一位将对应于一个端口的引脚。
2. 读取状态寄存器会清除各个状态位，这样会使中断线返回低电平状态。如果在执行 ISR 时未读取状态寄存器，将连续执行 ISR。

2.1.2 DMA nrq 中断源

在 PSoC 3 和 PSoC 5LP 中，每次完成直接存储器访问 (DMA) 的传输操作后，DMA 都将生成一个脉冲信号。DMA 数据传输操作完成后，该传输完成的信号 (被称为 *nrq* 信号) 可用于触发一个中断。它是一个边沿触发的中断。

2.1.3 UDB 中断源

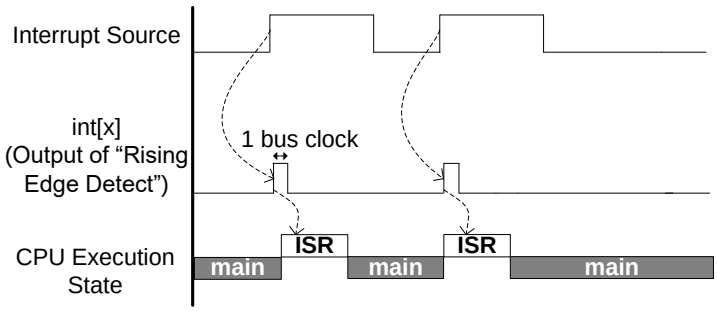
通过数字系统互连 (DSI) 将数字信号连线，可以配置任何数字信号作为中断源。这些源通常被称为 UDB 中断源，因为它们大部分来自 PSoC 中的通用数字模块 (UDB)。UDB 是组成可配置数字外设的基本模块，如 UART、SPI、I2C、定时器、计数器和 PWM。

除了专用的连接路径，各个固定功能的中断源可以通过 DSI 接口布线，如第 3 页上的图 4 所示。

UDB 中断源有两个路径。

1. 第一个路径是 UDB 中断源直接连接至中断线，请参阅第 3 页上图 4 中的直接路径。该路径用于电平触发的 UDB 中断源，例如，通信外设 (如 UART 和 SPI) 用来指示数据缓冲区存在可供读取的数据。
2. 第二个路径是将 UDB 中断源经过上升沿检测逻辑，如图 4 所示。图 5 显示的是 UDB 中断源中的上升沿信号如何转换为脉冲信号。该功能可用于边沿触发的中断源。

图 5. 边沿触发的 UDB 中断源

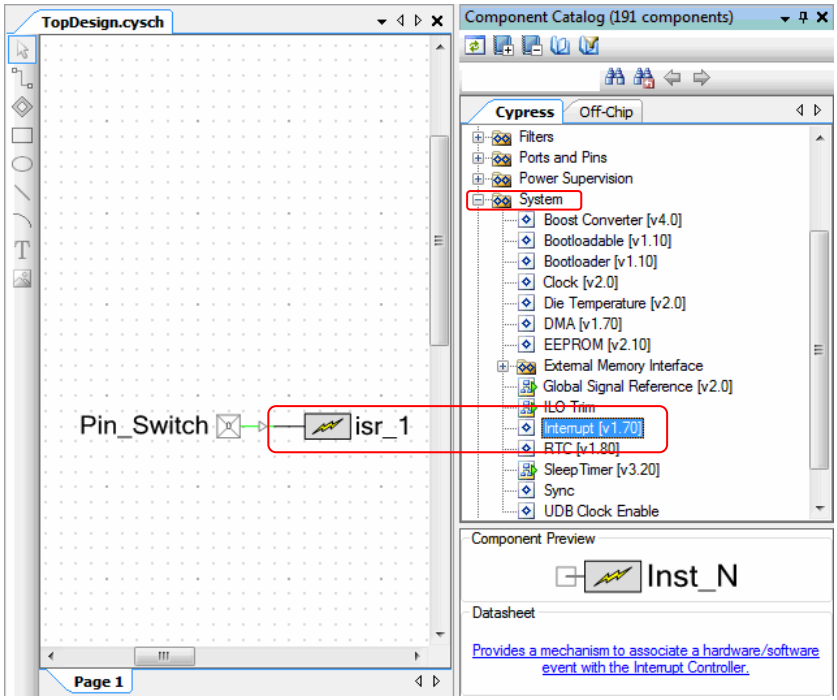


我们已经探讨了 PSoC 中断性能以及它们工作的原理。以下各节介绍了如何通过 PSoC Creator IDE 轻松处理 PSoC 中断。

3 PSoC Creator 的中断支持

通过将各中断作为一个组件提供，PSoC Creator 可支持它们。该中断组件位于 Component Catalog (组件目录) 窗口中的 System (系统) 选项卡下，如图 6 所示。每个中断组件实例均使用一个中断线。在项目原理图中，将某个中断源 (这个示例中是一个引脚组件) 连接至中断组件。

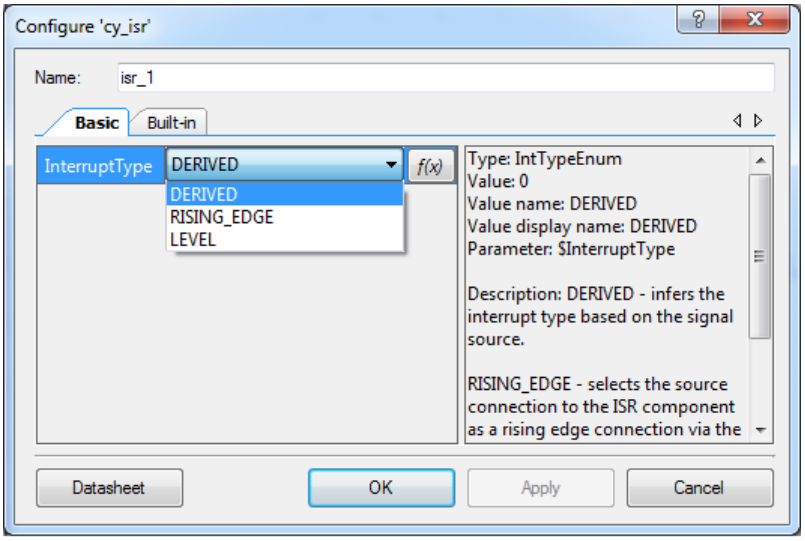
图 6. PSoC Creator 中断组件



3.1 中断组件配置

图 7 显示的是 Interrupt Component configuration (中断组件配置) 对话框。使用 InterruptType 参数配置中断源的复用器 (请参考第 3 页上的图 4)。

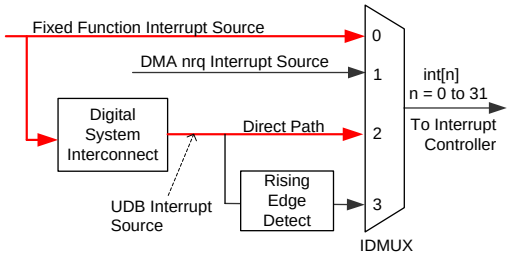
图 7. 中断组件配置



下面部分对 InterruptType 参数加以说明。

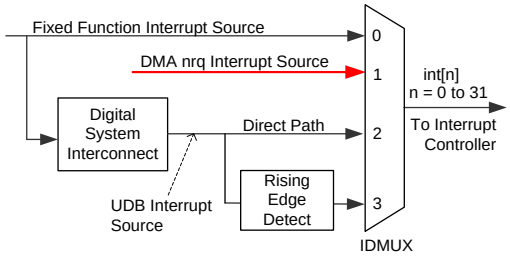
- **DERIVED**：对于该选项，PSoC Creator 将根据中断源的类型配置复用器：
  - 对于来自固定功能模块的数字输出信号 (请查看附录 A)，中断源直接连接至中断控制器。该连接可以使用中断源的专用连接路径，也可以通过 DSI 接口的直连路径，如图 8 所示 (也可以参考图 4)。

图 8. PSoC 3 和 PSoC 5LP 中固定功能模块的派生中断布线



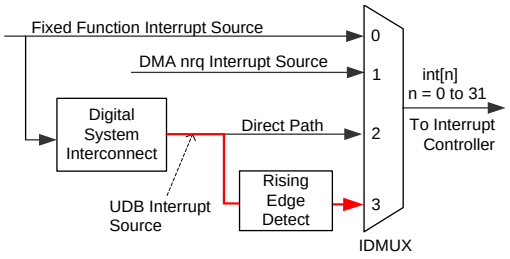
- 如果中断源是 DMA nrq 信号，将选中 DMA nrq 中断源的专用路径，如图 9 所示 (也可以查看图 4)。

图 9. DMA、PSoC 3 和 PSoC 5LP 的派生中断布线



- 对于其他中断源，如基于 UDB 的组件，或者其他数字信号，将选择 DSI 接口的上升沿选项，并且该中断源作为边沿触发的中断，如图 10 所示 (也可以参考图 4 和图 5)。

图 10. PSoC 3 和 PSoC 5LP 中 DUB 组件的中断布线



- **RISING_EDGE**：如果选中该选项，中断源信号的布线会经过上升沿检测路径，如图 10 所示。为所需要边沿触发的中断源选择该选项。
- **LEVEL**：如果选中该选项，中断源信号会经过 DSI 接口中的直接路径，如第 6 页上的图 8 所示。为所需要的电平触发中断源 (例如：基于 UDB 的组件，如 UART、SPI、定时器、计数器和 PWM) 选择该选项。

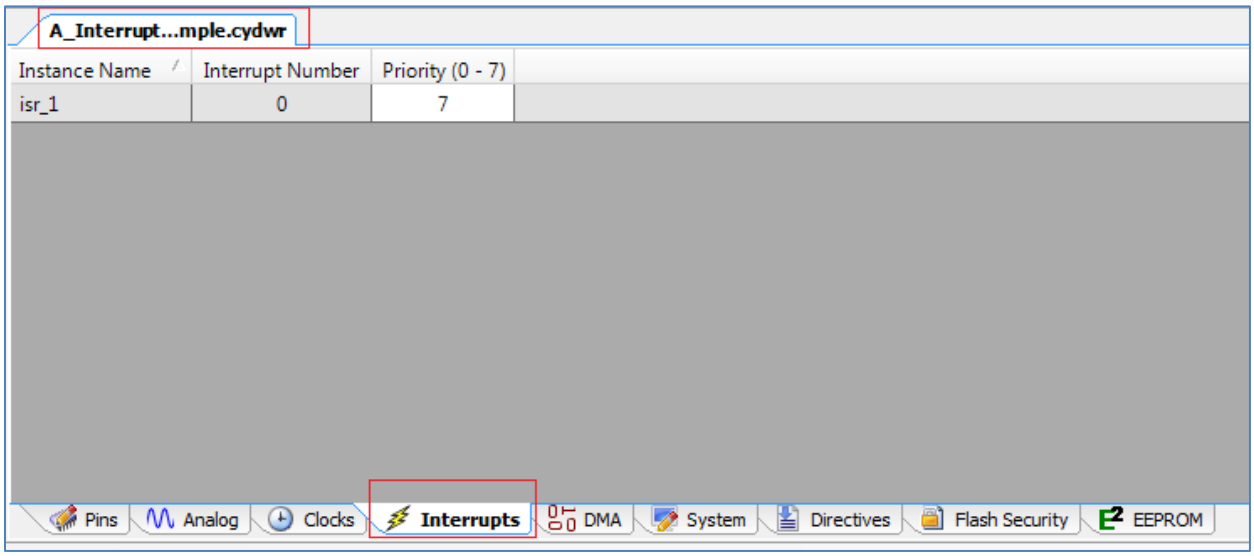
  注意： 仅为由基于 UDB 的组件 (如 UART 和 SPI) 生成的中断选择 LEVEL 选项。其原因是这些组件通过生成一个电平中断来表示 UDB FIFO 缓冲区的状态信号。深度为 4 个字节的 FIFO 缓冲区用于暂时存储所传输的 (Tx FIFO) 或所接收到的数据 (Rx FIFO)。

  对于 Rx FIFO，CPU 只要读取至少一个字节，FIFO 状态信号就为高电平。如果选中了 DERIVED 或 RISING_EDGE 选项，即使有多个字节存储在缓冲区内，则该中断只被触发一次。如果对 ISR 进行编写操作，使每次发生中断时只读取一个字节，那么缓冲区内的额外数据将被丢失。

3.2 中断优先级配置

PSoC Creator 项目的 Design Wide Resource (设计范围资源) 窗口 (*project_name.cydwr*) 中包含 Interrupts (中断) 选项卡，在该选项卡内显示了所有正在使用的中断的名称、优先级以及中断向量编号，如图 11 所示。

图 11.*cydwr* 窗口中的 Interrupt (中断) 选项卡



使用 *cydwr* 窗口更改中断的优先级。请注意：最高优先级为 0，最低优先级为 7。当构建项目时，PSoC Creator 将自动分配每个中断组件的中断矢量数。

通常，PSoC Creator 自动选择向量编号，但是您也可以手动更改它。请参考强制中断向量编号一节中的内容，了解更详细的信息。

4 中断项目

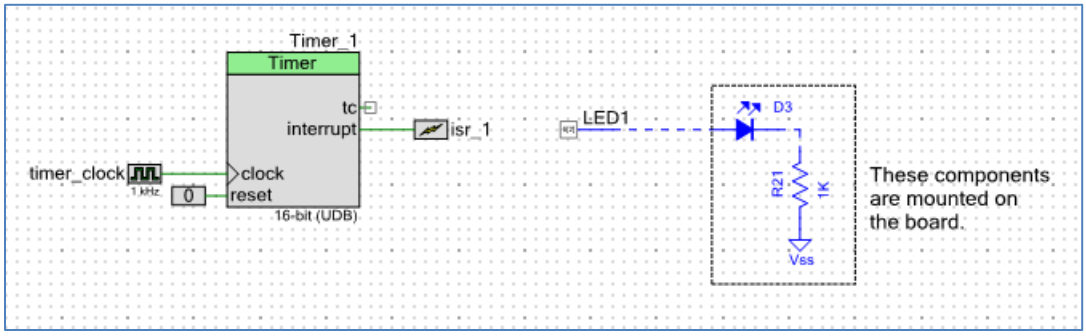
该应用笔记提供四个 PSoC Creator 项目来展示基本的中断，其为：端口中断控制器 (PICU)、低电压检测器 (LVD) 以及 PSoC 3 和 PSoC 5LP 中的 SysTick 中断。请注意，SysTick 是 Cortex-M3 CPU 的一个性能，它不适用于 PSoC 3。

4.1 简单的中断项目

该节向您介绍了创建一个基于中断的简单 PSoC Creator 中断项目。本应用笔记已随附了该项目的完整版本 (命名为 *A_InterruptExample*)。

该项目的目的是产生一个周期性中断，以所需要的频率切换引脚 (因此令一个 LED 灯闪烁)。该频率由时钟组件和定时器组件决定。图 12 是项目的原理图 (针对赛普拉斯开发套件 CY8CKIT-030 / CY8CKIT-050 而开发)。该项目可容易适应于其他套件，如知识库文章 — 将项目从 CY8CKIT-001 移植到 CY8CKIT-030 或者 CY8CKIT-050 所示。

图 12.*A_InterruptExample* 项目原理图

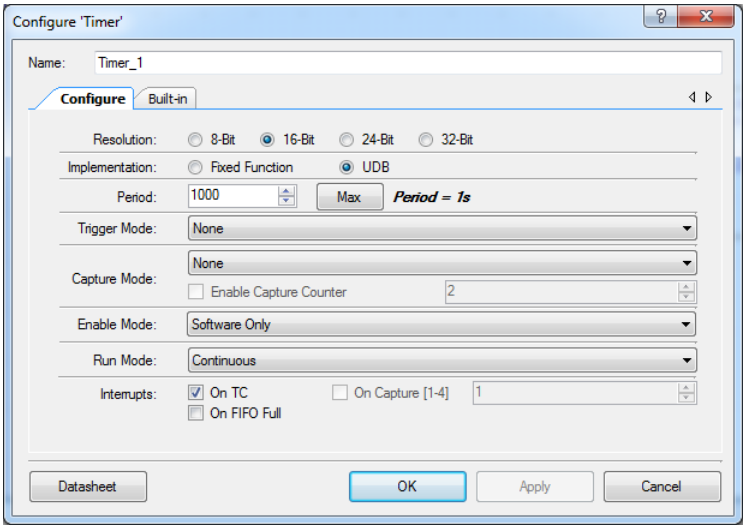


定时器组件是一个递减计数器，终端计数是指定时器已经达到其最低值 (零) 时的条件。在终端计数上，定时器重载周期值，并继续计数。终端计数条件出现时，可以随时生成中断。

中断输出将从定时器组件连接至中断组件。每次定时器的计数结束时，ISR 代码会翻转数字输出引脚 LED1 的电平。

中断频率由 timer_clock (定时器时钟) 频率和定时器周期 (Timer Period) 控制；根据图 13 所示的内容配置定时器组件。当 timer_clock 的频率为 1 KHz，定时器计数周期为 1000 时，计数结束事件每一秒发生一次。勾选“Interrupt on TC”复选框，并保证“Implementation” (实现) 为 UDB。保留其他参数为其默认设置。

图 13. 定时器组件配置



定时器组件数据手册表示中断输出仍被激活，直至读取定时器的状态寄存器为止。这意味着该中断是电平触发中断。配置中断组件，使 InterruptType (中断类型) 为 LEVEL (电平触发)，如第 5 页上的图 7 所示。

您可保留中断组件的优先级为它的默认值 (请参见第 7 页上的图 11)，因为该项目中只有一个中断。

4.1.1 编写中断服务子程序

完成原理图后，通过 PSoC Creator 菜单依次选择 Build > Generate Application。PSoC Creator 将生成各个源文件和头文件，以供给各组件在该项目中使用。它包含每个中断组件的默认中断服务子程序 (ISR)，该程序位于相应的源文件内。该 ISR 函数的名称为 `CY_ISR(isr_name)`。您必须将 ISR 代码写入到该函数内的占位符内，并添加任何相关的 `#include` 语句。

在该项目中，ISR 代码执行一下两项任务：

- 读取定时器的状态寄存器，以清除中断源。定时器组件的头文件必须包含在 ISR 源文件内。
- 切换引脚组件，使 LED 闪烁。引脚组件的头文件必须包含在 ISR 源文件内。

PSoC Creator 提供了中断源文件开头部分中的一个区域，用来包含各个头文件，如代码 1 所示：

代码 1. ISR 源文件中的头文件代码

```
/*****************************************
* Place your includes, defines and code here
*****************************************/
/* `#START isr_1_intc` */
/* Timer component header file */
#include "Timer_1.h"
/* LED1 pin component header file */
#include "LED1.h"
/* `#END` */
```

只能在#START 和#END 的两行之间写入头代码。创建项目的过程中，PSoC Creator 只保存在这两行之间所写入的代码。一旦重新创建项目，在此处之外所写入到的任何代码均被删除。

代码 2 显示的是该项目的 ISR 代码。它被在 ISR 函数中所提供的占位符区域内写入。

代码 2. ISR 源文件中的 ISR 代码

```
CY_ISR(isr_1_Interrupt)
{
    /* Place your Interrupt code here */
    /* `#START isr_1_Interrupt` */
    /* Read Timer status register to bring the interrupt line low */
    Timer_1_ReadStatusRegister();
    /* Toggle the LED */
    LED1_Write(~LED1_Read());
    /* `#END` */
}
```

只能在#START 和#END 的两行之间写入 ISR 代码。创建项目的过程中，PSoC Creator 只保存在这两行之间所写入的代码。一旦重新创建项目，在此处之外所写入到的任何代码均被删除。

4.1.2 完成 Main 函数

在主代码中，各组件被初始化并启动，如代码 3 所示：

代码 3. 简单的中断项目 *main.c*

```
void main()
void main()
{
    /* Initialize the interrupt and timer*/
    isr_1_Start();
    Timer_1_Start();

    CyGlobalIntEnable; /* enable Global interrupts */

for(;;)
    {

       /* Do nothing in the main loop; code to do something is in
the ISR. */
    }
       }
```

在代码 3 中，宏 CYGlobalIntEnable 配置中断控制器以生成发送至 CPU 的中断请求信号，同时配置 CPU 接收请求信号。该宏被定义在自动生成文件 *CyLib.h* 中。此外，根据实际所选中的 PSoC 3 或 PSoC 5LP 器件系列，它都具有与之相应的定义。使用该宏来确保代码对不同编译器工具链和器件系列具有可移植性。

isr_1_Start()函数初始化并使能 *isr_1* 中断组件。具体来说，它实现以下各项任务：

- 将中断向量地址设置为默认 ISR 函数的地址 (在中断组件源文件中已提供了该 ISR 函数)
- 根据 cydwr 窗口中所分配的优先级值来配置中断优先级
- 使能中断

4.1.3 关键字 CY_ISR 的重要性

中断源文件通过使用 CY_ISR 宏来定义 ISR 函数。每个编译器都需要一个特定函数的定义格式，以识别作为 ISR 的函数。*cytypes.h* 自动生成文件中所定义的宏为编译器提供了一个独立的 ISR 定义格式。

同样，CY_ISR_PROTO 宏声明了一个 ISR 函数原型。该声明位于中断组件的头文件中。例如，在头文件 *isr_1.h* 中，中断组件 *isr_1* 含有以下函数原型的声明：

```
CY_ISR_PROTO(isr_1_Interrupt);
```

该宏也被定义于 *cytypes.h* 文件中，并且根据所选的编译器而具有相应的定义。

这样就完成了基于简单中断的项目。创建任何其他基于中断的项目时，应当遵循相同的步骤序列。下面各节探讨了可用于 PSoC Creator 中各中断的一些高级性能。

4.2 PICU 中断项目

这是第二个示例项目；它作为 *B_PICU* 项目添加到本应用笔记。它说明了以下各主题：

1. 端口中断控制单元 (PICU) 的用途，该中断与数字输入引脚相关
2. 如何将一个用户自定义的函数变为 ISR，而不使用由 PSoC Creator 生成的默认 ISR

该项目使用机械开关控制引脚的输出，一旦*按下*开关，该项目将翻转引脚的输出电平 (令一个LED闪烁)，但开关被释放时便不再进行此操作。为了能准确地实现该性能，开关需要具有防抖动特性。监测和去抖动两个开关，每个开关具有与它相关的一个单独的LED。可采取多种方式实现这些要求；该项目中采用了基于中断的方式。

该项目仅处理PSoC中GPIO引脚的特殊中断。要想详细了解示例项目中GPIO的配置功能 ，请参阅以下应用笔记：AN72382 — 使用PSoC 3 和PSoC 5LP GPIO引脚。

该项目针对赛普拉斯开发套件CY8CKIT-001，但可容易适应于其他套件，如知识库文章 — 将项目从CY8CKIT-001 移植到CY8CKIT-030 或者CY8CKIT-050 所示。

4.2.1 PICU 中断

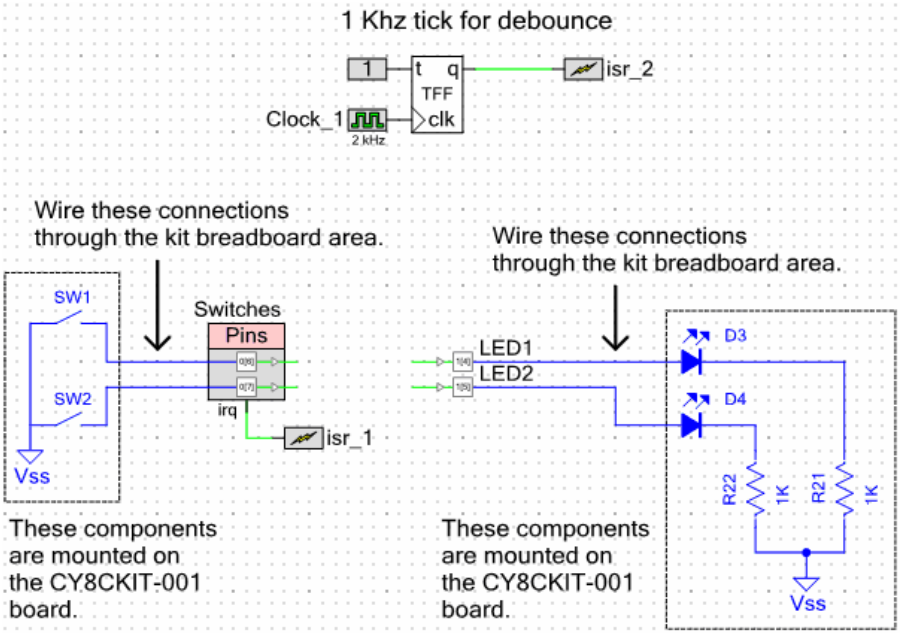
检测开关按下状态时，请使用端口中断控制单元 (PICU) 的中断。每个端口均有一个 PICU 中断，且包含八个 I/O 引脚。在上升沿、或下降沿、或两种边沿上均可单独地配置每一个 I/O 引脚，以生成 PICU 中断。

在此引脚进行配置时，将设置 8 位 PICU 状态寄存器的相应位。PICU 中断是状态寄存器的位的逻辑“OR”(或) 输出。可对 PICU 状态寄存器进行读取操作，以确定已生成中断的端口。

4.2.2 项目原理图

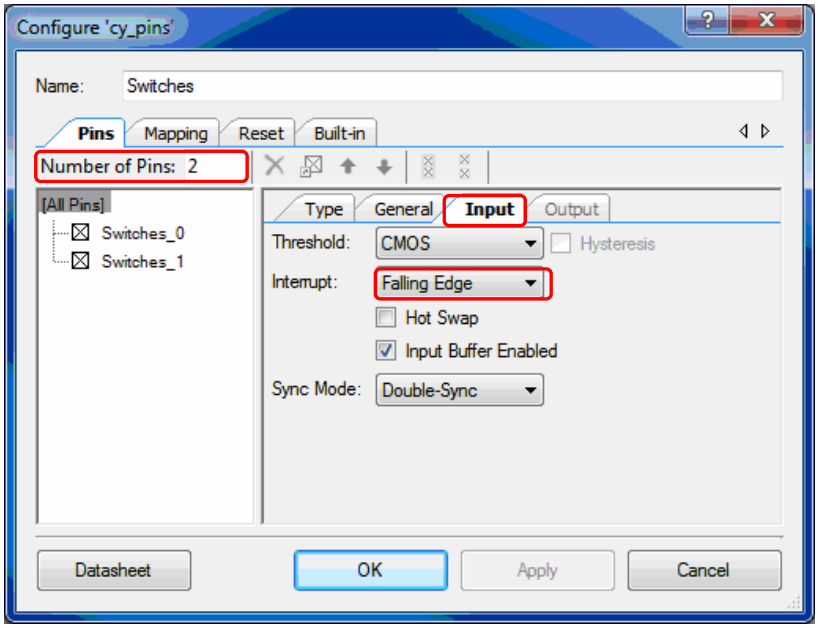
此项目原理图的详细说明显示在图 14 中。将开发套件上的开关 SW1 和 SW2 分别连接到两个输入引脚，用一个名字为“Switches”的数字输入引脚组件表示。将 PICU 中断信号从每个输入引脚连接至中断组件“isr_1”。第二个中断组件(“isr_2”)连接到时钟组件和 TFF 组件，以生成一个 1 kHz 的计时中断，用于开关去抖动。

图 14.*B_PICU* 项目的原理图



每个开关均是一个按键。被按下时，它立即接地。因此，要通过 General (通用) 选项卡配置引脚组件为上拉电阻驱动模式，以便使引脚的初始状态为逻辑高电平。此外，配置该组件，使其在引脚信号下降沿时生成 PICU 中断，如图 15 所示。

图 15. PICU 中断配置



4.2.3 项目固件

类似于简单的中断项目，*main.c* 中的代码也非常简单 。它只启动了两个中断组件并使能了全局中断。它在主循环中不做任何操作，则所有操作均在 ISR 中执行。

该项目的不同之处在于我们并不使用中断组件自动生成的 ISR。我们将编写自己的 ISR，如接下来的部分所示。请查阅附上项目 *B_PICU* 内 *InterruptRoutines.h* 和 *InterruptRoutines.c* 两个文件中的代码。这两个 ISR 分别被命名为 PICU_ISR 和 Tick_ISR。“Tick ISR”每毫秒只执行一次，不过一般都没有执行什么操作。“PICU ISR”会在任何一个开关按下时执行，并初始化用于去抖动超时的相应变量。然后“Tick ISR”将对此变量进行递减计数。递减到零时，“Tick ISR”会翻转相应引脚的输出电平。

当去抖动定时器不为零时，PICU ISR 将不会被执行，因此在去抖动周期内的产生的下降沿将被忽略。

欲了解更多有关开关去抖动信息，请参考相关的应用笔记 AN60024 — PSoC 的开关去抖动和抗干扰滤波器。

4.2.4 编写自己的 ISR

编写自己的 ISR 的主要原因是为了避免以下情况：当在 PSoC 器件系列之间移植某个工程时，必须向针对不同器件的多个中断组件源文件中添加 ISR 代码的多个副本，如图 16 所示。这可能会导致同一个代码的多个副本的维护问题。

反而，可以在您自己的源文件中编写 ISR，而不使用自动生成的代码，如图 17 所示。那时，ISR 的更改会被应用于任何 PSoC 系列中所选的器件。

图 16. 多个单独的 ISR 源文件

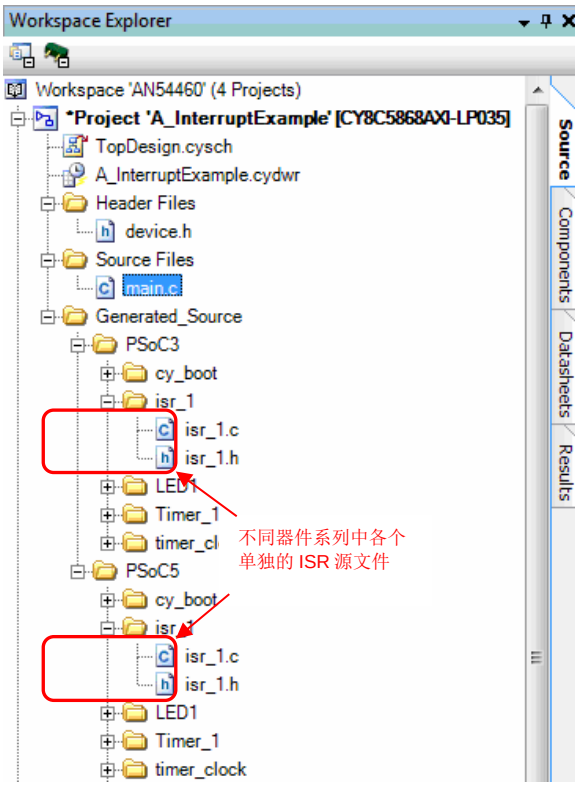
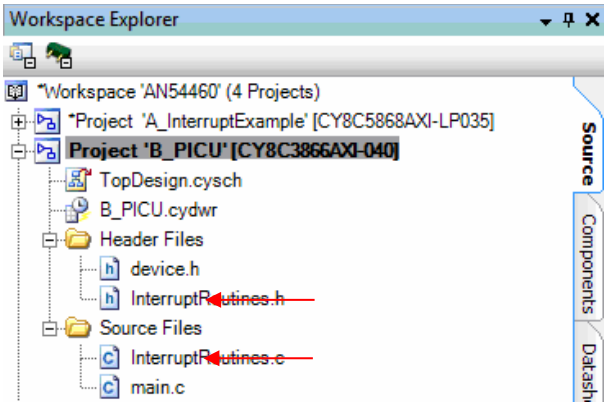


图 17. 通用的 ISR 源文件



为了能使您自己的函数 (例如：MyCustomISR) 作为某个 *isr_1* 中断组件的 ISR，请执行以下操作：

1. 通过使用 `CY_ISR_PROTO` 宏来声明该函数：

```
CY_ISR_PROTO(MyCustomISR);
```

2. 使用 `CY_ISR` 宏来定义函数：

```
CY_ISR(MyCustomISR)
{
```

```
    /* ISR code goes here */
}
```

3. 在 *main.c* 的初始化代码中，请添加对 `isr_1_StartEx()` 函数的调用。该函数类似于 `isr_1_Start()` 函数，但 `isr_1_StartEx()` 具有用于调用 ISR 函数的参数：

   `isr_1_StartEx(MyCustomISR);`

   `isr_1_StartEx()` 函数将中断向量地址设置为传入函数的地址。

请参考本应用笔记所附带的 *B_PICU* 项目，了解该技术的实现细节。为了便于参考，下面内容提供了代码段。

该项目的 `main()` 函数如下：

```
#include <device.h>

/* Header file containing the custom ISR prototypes */
#include "InterruptRoutines.h"

void main()
{
    /* Initialize the two custom defined ISRs */
    isr_1_StartEx(PICU_ISR);
    isr_2_StartEx(Tick_ISR);

    CyGlobalIntEnable; /* Enable global interrupts. */

    for(;;)
    {
        /* Do nothing in the main loop; code to do something
        is in the ISRs */
    }
}
```

该项目的 PICU_ISR 如下所示：

```
CY_ISR(PICU_ISR)
{
      /* copies of PICU registers */
      uint8 CYDATA temp_stat;

      /* read the PICU interrupt status register, with a clear on read */
      temp_stat = Switches_ClearInterrupt();

      /* Process the PICU event on SW1 only if any
         ongoing debounce period has timed out */
      if(((temp_stat & SW1_MASK) != 0) && (switch_1_timeout == TIMED_OUT))
      {
            /* reset the debounce timer for this button */
            switch_1_timeout = DEBOUNCE_TIME;
      }

      /* Process the PICU event on SW2 only if any
         ongoing debounce period has timed out */
      if(((temp_stat & SW2_MASK) != 0) && (switch_2_timeout == TIMED_OUT))
      {
            /* reset the debounce timer for this button */
            switch_2_timeout = DEBOUNCE_TIME;
      }
}
```

`Switches_ClearInterrupt()` API 用于读取 PICU 状态寄存器并确定相关引脚是否导致中断。如果在端口引脚出现一个下降沿转换并相应的去抖动定时器已经超时，去抖动定时器将重启。分别使用 `switch_1_timeout` 和 `switch_2_timeout` 变量来实现去抖动定时器。通过修改 *InterruptRoutines.h* 文件中的 `DEBOUNCE_TIME` 宏定义，可以配置去抖动周期。

命名为 Tick_ISR 的 1 ms 周期 ISR 将实现端口引脚上的去抖动逻辑过程。去抖动周期结束后，ISR 将切换输出引脚。项目的 Tick_ISR 代码如下：

```
CY_ISR(Tick_ISR)
{
      if(switch_1_timeout != 0)
      {
            /* Come here if an edge was detected by the PICU.
               Decrement and check timeout.
            */
            if(--switch_1_timeout == TIMED_OUT)
            {
                  /* Timed out. Toggle LED only on a switch press (= 0). */
                  if((Switches_Read() & SW1_MASK) == 0)
                  {
                        LED1_Write(~LED1_Read()); /* toggle the LED */
                  }
            }
      }

      /* repeat for switch 2 */
      if(switch_2_timeout != 0)
      {
            if(--switch_2_timeout == TIMED_OUT)
            {
                  if((Switches_Read() & SW2_MASK) == 0)
                  {
                        LED2_Write(~LED2_Read()); /* toggle the LED */
                  }
            }
      }
}
```

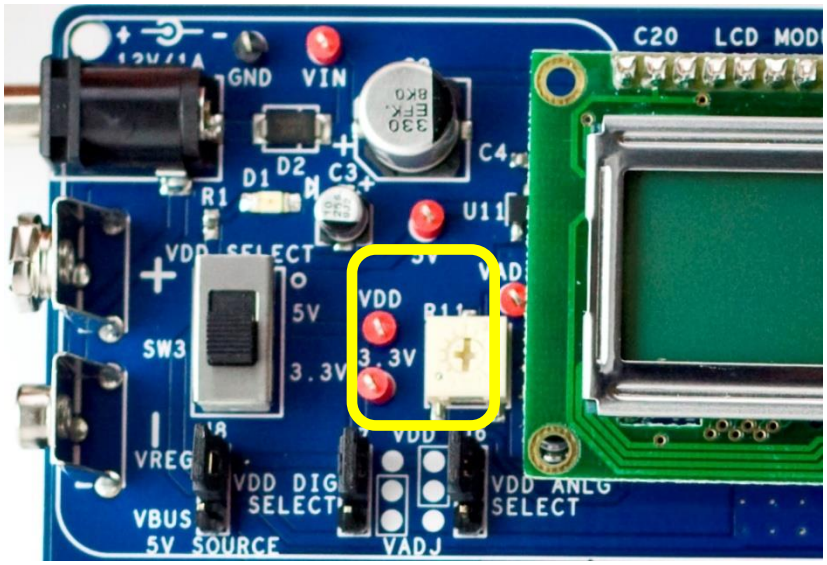
4.3 LVD 中断项目

这是本应用笔记所附带的第三个例程，名称为 *C_LVD*。该例程介绍以下主题：

1. 使用全局信号参考组件来生成中断
2. 使用某个信号量或标志、变量在主代码和 ISR 间进行通信

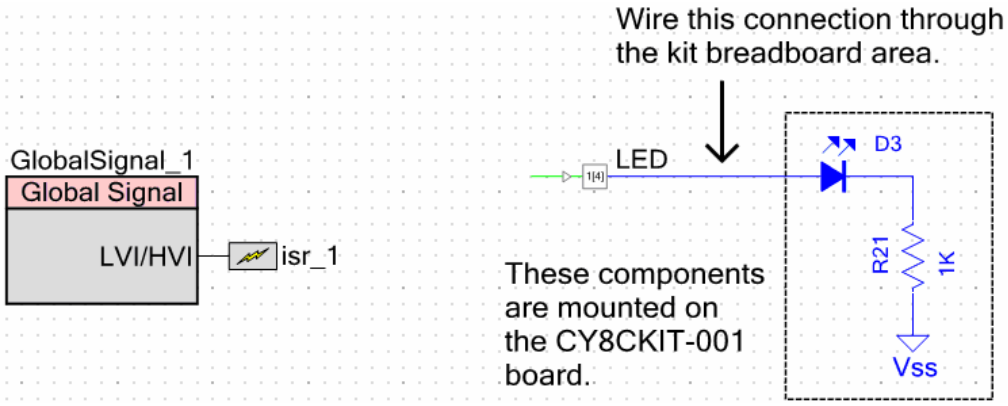
该项目监控器件 Vdd 引脚上的电压。一旦该电压下降至低于触发电平，它会点亮一个 LED；当电压回升超过该触发电平时，将关闭 LED。赛普拉斯的 CY8CKIT-001 开发套件具有一个用于测试该性能的可调电压调节器。为了使用该可调电压调节器，请将跳线器 J2、J3、J4、J5、J6 和 J7 转移到各自的 VADJ 位置。通过旋转电位器 R11 (在旧版本套件中为 R10) 调整电压调节器的输出电压，如图 18 所示。将电压计连接至 GND 和 VADJ 测试点，以监控该电压。

图 18. CY8CKIT-001 的可调电压调节器的电位器和测试点



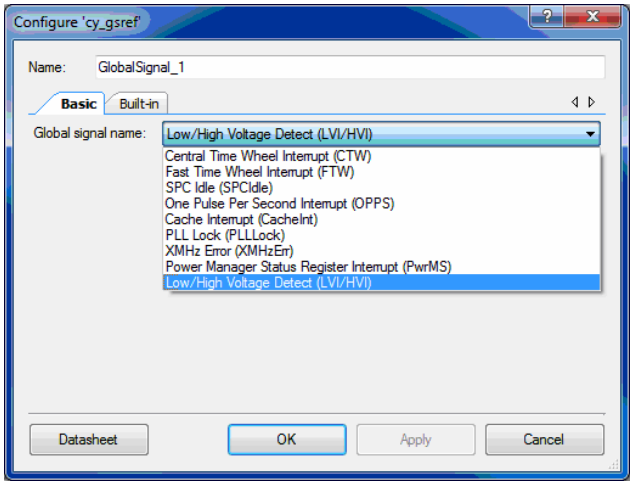
使用全局信号参考组件使能 LVD 的监控操作，如图 19 所示：

图 19.*C_LVD* 项目的原理图



通过全局信号参考组件，可以更轻松使用 PSoC 中可用的一些系统级中断，如图 20 所示：

图 20. PSoC 3 和 PSoC 5LP 的全局信号参考组件配置



该项目也演示了如何使用某个信号量、标志或变量以进行 ISR 和主代码之间的通信。在嵌入式系统设计中，这是一个常见的问题：ISR 将处理任务转移到主代码，从而减少执行 ISR 所花费的时间。如果主代码和 ISR 位于同一个文件中，此变量将被定义为一个静态类型的全局变量[1]：

```
static volatile CYBIT isr_flag = 0; /* semaphore between ISR and main() */
```

在 ISR 中设置该变量。LVD_ISR 代码具体如下：

```
CY_ISR(LVD_ISR)
{
    /* Disable the LVD interrupts. This causes this ISR to execute only once
       when the voltage is below the trigger level, otherwise the interrupt
       occurs continuously.
     */
    isr_1_Disable();

    isr_flag = 1;
}
```

当 V_{DD} 电压低于触发电平时，将不断产生中断。为了能达到更高的效率，我们希望相应的 ISR 只执行一次，因此在 ISR 中会禁止中断[2]。主代码监控该标志；当 ISR 设置它时，主代码将打开 LED，并等待电压上升超过触发电平。然后关闭 LED，并重新使能中断。

注意： 在主代码处理中断事件通常会在中断发生到中断被处理之间引入一段不定的延迟时间。如果中断处理时间有约束，则该代码应该放在 ISR 中而不是放在主程序中，那时不需要使用该技术。

[1] 因为该变量是布尔变量，所以通过 CYBIT 宏可利用 PSoC 3 中 8051 CPU 的位处理功能。更多有关信息，请参考《AN60630 — PSoC 3 8051 的代码优化》和《PSoC Creator 系统参考指南》中第二节的内容。 在 PSoC 5LP 中，该宏将实现 `uint8` 类型。

[2] 中断组件的“InterruptType” (中断类型) 被设置为“DERIVED” (请参考中断组件配置”)。这样会引起一个电平触发中断，这也是电压低于 LVD 触发电平时持续生成中断的原因。可以将“InterruptType” (中断类型) 设置为 RISING_EDGE，以便电压下降低于触发电平时仅生成一个中断。然而，这并不是理想的代替方法，因为当电压回升超过触发电平时会生成一种额外的中断。因此，仅生成唯一一个 LVD 中断的最通用和有效的方法就是先在 ISR 中禁止它，然后电压上升超过触发电平时再次使能它。

main()代码具体如下：

```
void main()
{
    /* initialize LVD for both digital and analog (Vddd and Vdda) */
    CyVdLvDigitEnable(0/*interrupt, not reset*/, 5/*~2.95 V*/);
    CyVdLvAnalogEnable(0/*interrupt, not reset*/, 5/*~2.95 V*/);

    isr_1_StartEx(LVD_ISR); /* initialize the custom defined ISR */
    isr_flag = 0; /* initialize semaphore before turning on interrupts */
    CyGlobalIntEnable; /* Enable global interrupts. */

    for(;;)
    {
        /* monitor for LVD interrupt */
        if(isr_flag != 0)
        { /* an LVD interrupt occurred */
            isr_flag = 0;
            LED_Write(1); /* turn LED on to indicate an interrupt occurred */

            /* Wait here for the LVD condition to clear: get the interrupt
               source until the response stays off.
            */
            while(CyVdStickyStatus(3/*LVID and LVIA only*/) != 0)
            {
                /* There is minimal hysteresis in the LVD trigger circuits so
                   add a delay to compensate.
                */
                CyDelay(1/*msec*/);
            };

            /* LVD condition cleared */
            isr_1_ClearPending(); /* in case the interrupt is still pending */
            isr_1_Enable(); /* re-enable LVD interrupt */
            LED_Write(0u); /* turn the LED back off */
        }

        /* Nothing else to do in main except monitor for LVD condition. */
    }
}
```

4.4 SysTick 中断项目

这是第四个兼最后的示例项目；它以*D_SysTick项目*的名字附在本应用笔记中。它说明了如何使用PSoC 5LP中Cortex-M3 CPU的SysTick功能。PSoC 3 中的 8051 CPU没有SysTick功能，但您可以使用PICU中断项目一节中所描述的技术来实现该性能。该项目针对赛普拉斯开发套件CY8CKIT-050，但也可容易移植至其他套件。

大多数实时操作系统 (RTOS) 使用系统节拍中断来控制任务切换和定时。 系统滴答速率通常为 1 毫秒。 PSoC Creator 通过在文件 *CyLib.c* / *CyLib.h* 中提供高级 API 来简化 SysTick 定时器的使用。 有关 SysTick API 的更多详细信息，请参阅 *System Reference Guide*”文档中的“System Timer (SysTick)”部分（也可在 PSoC Creator 菜单 Help > Documentation 中找到)。

在 *main.c* 中查看附加项目 *D_SysTick* 中的代码。 在 main 函数中，启动 SysTick 并设置在 Systick 中断上调用的回调函数。 此回调函数 (SysTickISRCallback) 处理一个毫秒计数器，该计数器每秒切换一次 LED。`main()`函数代码如下：

```
void main()
{
uint32 i;

/* Enable global interrupts. */
CyGlobalIntEnable;

/* Configure the SysTick timer to generate interrupt every 1 ms
    * and start its operation. Call the function before assigning the
    * callbacks as it calls CySysTickInit() during first run that
    * re-initializes the callback addresses to the NULL pointers.
    */
    CySysTickStart();

/* Find unused callback slot and assign the callback. */
for (i = 0u; i <CY_SYS_SYST_NUM_OF_CALLBACKS; ++i)
    {
if (CySysTickGetCallback(i) == NULL)
        {
/* Set callback */
            CySysTickSetCallback(i, SysTickISRCallback);
break;
        }
    }

for(;;)
    {
/* Do nothing in the main loop; code to do
            something is in the ISR callback */
    }}
```

请注意，该项目的原理图中没有包含中断组件，并且 *cydwr* 文件中的 Interrupts (中断) 选项卡下也没有显示 SysTick 中断。除了上述函数外，PSoC Creator 不支持 Cortex M3 CPU 的 SysTick 及其他内部中断。

5 高级中断主题

5.1 优化中断代码

在基于中断的应用中，ISR 代码执行时间是重要的性能要求之一。在某些应用中，当接收某个中断请求时，必须在特定的时间内执行 ISR 中的关键处理代码。此外，中断的执行时间也不能太长，否则会使主代码和其他中断的执行被停顿。要想满足这些要求，请遵循下面准则：

- **避免在 ISR 中调用函数。** 由于函数的调用和返回涉及到压栈/出栈的开销，因此调用函数将延长 ISR 代码的执行时间。执行函数可能会引入不确定的过长时间，例如字符 LCD 显示子程序。当这些调用函数在更高的优先级实现时，执行更低优先级中断中的延迟会增加。

  推荐将非重要的函数调用移到主代码内，在 ISR 中只设置一个标志变量，如 LVD 中断项目一节中所述。主代码将定期检查该标志。如果该标志被置位，主代码会清除它，并调用函数。

- **请给各个中断分配合适的优先级。**在包含了多个中断的应用中，请给时间关键的中断配置更高的优先级。

- **使用代码优化技术。** 带有 8 位 8051 CPU 的 PSoC 3 的代码执行时间比带有 32 位 ARM Cortex CPU 的 PSoC 5LP 更长。因此，您可能需要优化 PSoC 3 中的 ISR 代码，以满足代码执行时间的要求。

  8051 CPU 架构提供多种特性用来加快代码执行速度。例如将 ISR 本地变量放置在 8051 内部数据空间内，以及将仅包含了二种可能数值 (零和非零值) 的变量定义为位变量。更多有关信息，请查阅应用笔记 AN60630 — PSoC 3 8051 代码优化部分内容。有关 PSoC 5LP 代码优化的更多信息，请参阅应用笔记 AN89610 — PSoC 4 和 PSoC 5LP ARM Cortex 代码优化。

5.2 中断组件 API

中断组件自动生成的源文件和头文件提供了一些 API，用于配置中断。到目前为止，本文随附的项目只使用了常用的 `isr_Start()` 和 `isr_StartEx()` 函数。其他函数能够实现其他额外的功能：

- `Enable()`, `Disable()` 分别用于使能和禁用中断。请注意，禁用中断不会清除任何挂起的中断请求。 应使用 `ClearPending()` API 来清除待处理的请求。
- `SetVector()`, `SetPriority()` 分别用于动态修改中断向量地址和中断优先级。最佳方案是在调用这些函数前，先禁用中断。
- `SetPending()` 可以使用固件设置中断等待状态而不需要硬件的中断请求。
- `ClearPending()` 用于清除中断等待状态，使得该中断不被响应。该函数不对中断源信号产生任何影响；它只会清除中断控制器的中断信号线上的挂起状态位。要了解清除源中断信号的过程，请参见相应外设的组件数据手册。

请参见中断组件数据手册，了解中断 API 的详细介绍。

在 *CyLib.h 和 CyLib.c* 文件中，PSoC Creator 还提供了一组通用的中断函数。这些函数对 PSoC Creator 中不受支持的固定函数中断进行配置，如不受支持的固定功能中断一节所述。更多有关这些函数的信息，请参见 *系统参考指南* 文档 (也可以在 PSoC Creator 菜单依次选择 Help (帮助) > Documentation (文档)，找到该文档)。

通常的做法是在执行时间关键任务时禁用全局中断生成，或者在主线程中更新在中断例程中访问的变量时也是如此。PSoC Creator 中有两个 API 可用于实现动态全局启用 / 禁用中断：`CyEnterCriticalSection` 和 `CyExitCriticalSection`。这两个 API 用于避免固件变量和硬件寄存器的损坏。`CyEnterCriticalSection` API 禁用中断并返回一个值，指示先前是否启用了中断。`CyExitCriticalSection` API 恢复中断。

为了了解其工作原理，下面显示了写入定时器控制寄存器的示例。

```
TIMER_CONTROL_REG |= TIMER_MASK;
```

执行上述语句时会发生以下操作序列：

1. CPU 读取定时器的控制寄存器并存储在临时寄存器中。

2. CPU 使用掩码值执行临时寄存器的逻辑或。

3. 将 OR 结果加载回控制寄存器。

在步骤 1 之后，发生中断并且其处理程序在同一控制寄存器中加载新值。 在执行处理程序之后，当 CPU 恢复执行步骤 2 时，它使用临时寄存器中的陈旧控制寄存器值导致数据损坏。

为避免此问题，代码如下：

```
InterruptState = CyEnterCriticalSection();
TIMER_CONTROL_REG |= TIMER_MASK;
CyExitCriticalSection(InterruptState);
```

`CyEnterCriticalSection` 和 `CyExitCriticalSection` API 的使用，通过在写入控制寄存器时禁止发生任何中断来避免此问题。 这种情况也可以在简单的写操作中发生，因为它将导致使用临时变量。 如果在 main () 函数和任何中断处理程序中修改了任何变量或硬件寄存器，建议使用这两个 API。

有关这些 API 的详细信息，请参阅“*System Reference Guide*”文档 (也可在 PSoC Creator 菜单 Help > Documentation 中找到)。

5.3 中断及其他组件

很多 PSoC Creator 组件在实现其功能时，已经包含了一个内部中断组件。如实时时钟 (RTC)，各种通信组件 (如 UART 和 SPI)，以及 Delta Sigma ADC。

与中断组件相比相同，这些组件中的内部 ISR 提供了一个用于写入用户代码的占位符区域。

请参见 PSoC Creator 中所提供的相应组件数据手册以及相关的代码示例，了解这些组件中的中断用法。有关处理硬故障异常的信息，请参阅知识库文章 - 硬故障中断处理程序功能 - KBA86934。

5.4 中断延迟

中断延迟定义为中断断言和 ISR 中第一条指令执行之间的时间延迟。 PSoC 3 的 8051 CPU 具有 25 个周期的中断延迟，而 PSoC 5LP 的 ARM Cortex M3 CPU 具有 12 个周期的中断延迟。 这些延迟值在相应的器件数据表中指定，并且仅包括 CPU 中断架构特定的延迟值。 由于以下因素，观察到的实际中断延迟将超过此时间 - 由于闪存等待状态用于获取驻留在闪存中的中断代码的开销，ISR 中的函数调用将导致 CPU 在执行实际的中断代码之前推送所有 CPU 寄存器到堆栈。 优化中断代码部分中列出的技术可减少中断延迟。

在执行实际 ISR 代码前，进行了以下活动：

1. 处理器将当前的程序计数器 (PC)、CPU 内核专用寄存器以及一些通用的 CPU 寄存器推进到堆栈中。
2. 处理器从 NVIC 读出向量地址并将其更新到 PC 中。
3. 处理器更新 NVIC 寄存器。

要使中断服务流程更有效，PSoC 5LP 中的 Cortex M3 处理器将实现下面两个设置：

1. *尾连锁：*如果中断处于挂起状态，而处理器正在执行另一个中断处理，当执行完第一次中断并立即处理挂起中断时，可以放过拆除 (un-stacking) 这步骤。该步骤将帮您减少从堆线恢复寄存器并将相同寄存器重新放入堆栈的时间。该步骤也对减少低优先级中断的延迟起着作用。
2. *迟到：*如果在低优先级中断的堆线期间发生更高优先级中断，处理器将优先处理更高优先级中断。堆栈过程结束时，处理器将读取更高优先级中断的向量地址。一旦更高优先级中断的处理过程完成，将读取及执行被挂起的更低优先级中断处理的向量地址。该过程将排除因进入更低优先级 ISR 并将寄存器值放入堆栈中所导致的延迟，从而减少更高优先级中断的延迟。

器件从中断唤醒时，执行上电序列和相应的低功耗唤醒代码序列后，电压的稳定过程需要一段延迟时间。请参见器件数据手册，了解有关器件从不同低功耗模式中的唤醒时间的信息。

5.5 调试

PSoC 3 和 PSoC 5LP 器件通过串行线调试 (SWD) 接口以及联合测试行动组 (JTAG) 接口来支持片上调试性能。PSoC Programmer — MiniProg3 通过 SWD/JTAG 接口与主机 (PSoC Creator) 对调试数据和指令进行通信。它允许您添加多个断点，评估和编辑变量值，检查 CPU 寄存器，观察固件的汇编指令，以及读取存储器。调试模式有助于检查中，具体如下：

- 要想检查中断是否被执行，将一个断点添加到 ISR 指令之一。
- 使用调试器的 Call Stack 窗口来检查特殊中断被执行的时间。此窗口按顺序列出了被执行的函数调用指令。该窗口也可以用来检查一个高优先级中断是否发生在一个低优先级 ISR 的整个执行过程中。

使用断点触发计数来检测中断被触发的次数。它尤其有助于检查中断信号是否有干扰，若有干扰，这个中断可能被多次触发。

欲了解调试器的使用， 请参见 PSoC Creator Help 中的“使用调试器”一节。要访问本文档，请在 PSoC Creator 中按下 **[F1]** 按键或者使用 **Help** > **Topics** 菜单。

作为调试器的一个替换者，您也可以对引脚进行“bit bang”操作：一

- 检查 CPU 是否正在进入 ISR。
- 测量 ISR 执行时间。通过在 ISR 开始和结束时分别设置和重设置引脚，可以完成这个操作。

5.6 不受支持的固定功能中断

即使有全局信号参考组件，PSoC Creator 仍不支持某些 PSoC 中断源。在附录 A 中将这些中断源标记为“不受支持”。虽然不受支持，但是通过使用 *CyLib.c* 和 *CyLib.h* 文件中所提供的通用中断函数，用户仍然可以配置一定的固定功能中断源。

要想手动添加对某个固定功能中断的支持，请执行下面操作：

1. **配置固定功能中断源，以生成中断。**对特定的外设寄存器进行写操作，使能外设的中断生成 — 请参见 PSoC 3 或 PSoC 5LP 的技术参考手册 (TRM)，了解寄存器的相关信息。
2. **使能中断控制器逻辑 (仅适用于 PSoC 3)。** 在 PSoC 3 中，必须使能中断控制器模块的时钟信号，并配置中断控制器，使之为 CPU 生成 IRQ 信号。通过将“0x01”写入到 CYREG_INTC_CSR_EN 寄存器内，可以实现该操作：

   ```
   CY_SET_REG8(CYREG_INTC_CSR_EN, 0x01);
   ```

   将一个中断组件放在工程原理图内时，PSoC Creator 将自动配置该寄存器，但是不使用中断组件，手动配置中断则不会自动配置该寄存器。
3. **配置中断向量地址和中断优先级。**使用 *CyLib.h* 文件中所声明的 CyIntSetVector()和 CyIntSetPriority()函数：

   ```
   CyIntSetVector(Interrupt_Vector_Num, MyISR);
   CyIntSetPriority(Interrupt_Vector_Num, myPriorityValue);
   ```

   Interrupt_Vector_Num 是指固定功能中断源的中断向量编号 — 请参见附录 A。MyISR 是用户按照编写您自己 ISR 一节中的步骤进行定义的 ISR 函数。myPriorityValue 的有效范围是 0 到 7。
4. **使能固定功能中断和全局中断。** 使用下面代码：

   ```
   CyIntEnable(Interrupt_Vector_Num);
   CYGlobalIntEnable;
   ```

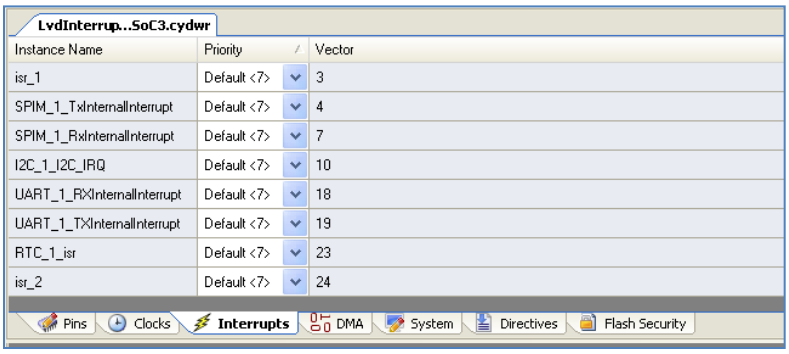
这样可以完成所需要的各个步骤，用于给 PSoC Creator 中固定函数中断源提供支持。由于大部分固定函数中断源都是电平中断，所以必须读取 ISR 中的外设状态寄存器，令中断信号线变为低电平。请参见技术参考手册 (TRM) 文档，了解不同固定函数外设所生成的中断信号的类型。

重要说明：在未使用 PSoC Creator 框架的情况下，本节所介绍的流程使用了一个固定函数中断源的中断向量编号。PSoC Creator 不知道设计中已经使用了该中断向量编号。因此，它可能将相同的向量编号分配给原理图中的中断组件。在这种情况下，您可以手动强制修改向量编号，如下一节所述。

5.7 强制中断向量编号

PSoC Creator 自动给项目中的中断组件分配向量编号。编译工程后，您可以在 *cydwr* 窗口中的 Interrupts (中断) 选项卡下查看所分配的向量编号，如图 21 所示。

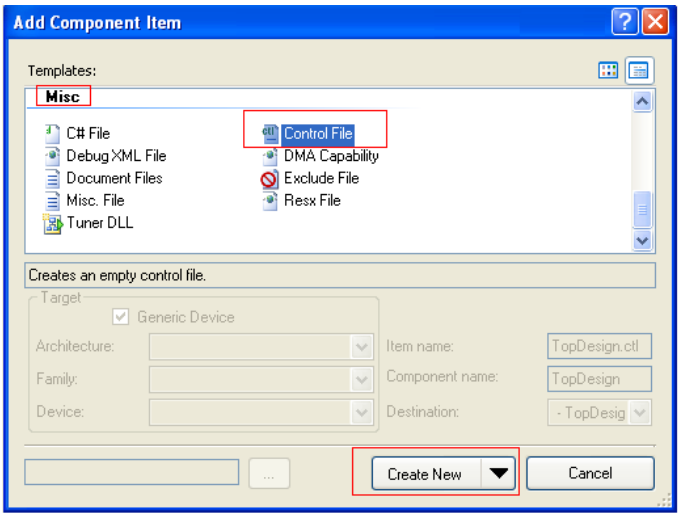
图 21.*cydwr* 窗口中的中断向量编号



如上一节所述，您可能需要覆盖 PSoC Creator 所分配的向量编号，并手动分配向量编号。通过使用一个 *Control File* (控制文件)，可以实现该操作。具体如下：

1. 点击‘Workspace Explorer’窗口中的 **Components** 选项卡。
2. 右键点击 **TopDesign** 组件，然后点击 **Add Component Item...**项。打开“Add Component Item”对话框。
3. 向下滑动到 **Misc** 组，选择 **Control File** 项，然后点击 **Create New** 按键，如图 22 所示。

图 22. 添加控制文件



一个 *TopDesign.ctl* 文件将被创建并添加到‘Workspace Explorer’窗口内。

4. 双击打开 TopDesign.ctl 文件，进行编辑。在控制文件中通过使用 attribute 关键词，可以指定每个中断组件的中断向量编号。指定中断向量编号的方法取决于下面两种情况：用户将中断组件放在示例原理图内；或在原理图中，PSoC Creator 组件内部使用了中断组件。两种方法如下介绍：

   a) 用户将中断组件放在原理图内时，句法将为：

   ```
   attribute placement_force of instance_name : label is "Intr(0,
   DesiredVectorNumber)";
   ```

   这里，instance_name 是指原理图中中断组件的名称，DesiredVectorNumber 为向量编号 (0 至 31)。例如，要想将向量 17 分配给中断组件 *isr_1*，该句法为：

```
attribute placement_force of isr_1 : label is "Intr(0, 17)";
```

b) 对于内部使用中断的组件，如 RTC、UART、SPI 以及 Delta Sigma ADC，句法将为：

```
attribute placement_force of \top_instance_name : InternalInterruptName\: label
is "Intr(0, DesiredVectorNumber)";
```

这里，top_instance_name 是指使用内部中断的组件名称。第 23 页上的图 21 的示例分别为 SPIM_1、UART_1 以及 RTC_1。

InternalInterruptName 为组件中分配给内部中断的名称。可以在 *cydwr* 窗口中的 Interrupts 选项卡 (图 21) 下查找该名称。其中，中断名称是组件实例名称的前缀。在图 21 中，*isr* 是 RTC 组件 (即为 RTC_1) 的内部中断名称。I2C_IRQ 是 I2C 组件 (即为 I2C_1) 的内部中断名称。因此，基于图 21 中的中断，中断向量编号分配如下：

```
attribute placement_force of \I2C_1:I2C_IRQ\ : label is "Intr(0,19)";
attribute placement_force of \RTC_1:isr\ : label is "Intr(0,3)";
attribute placement_force of \SPIM_1:RxInternalInterrupt\ : label is "Intr(0,4)";
```

5. 分配中断向量编号后，请点击 **Save** 按键，以保存控制文件中所修改的内容。
6. **Clean and Build** (清除并重新编译) 该例程，以使新的中断向量编号分配生效。*cydwr* 窗口中的“Interrupts”选项卡下显示了修改后的中断向量编号的分配情况。

6 总结

中断被广泛应用于嵌入式应用中。对于 PSoC 3 和 PSoC 5LP 等片上系统架构，将片上外设的状态传输给 CPU 时，中断起着关键作用。通过本应用笔记所提供的信息，可以快速容易地创建基于中断的 PSoC Creator 项目。此外，本文还介绍了各几个高级中断特性。

6.1 项目汇总

AN54460.cywrk：

该工作区包括四个代码示例，用于演示本应用笔记所介绍的不同主题。

- A_InterruptExample：*使用一个定时器中断演示了一个简单的应用，以翻转一个引脚上的输出电平。*
- B_PICU：*演示了端口中断控制单元 (PICU) 中断以及多个中断的用法。*
- C_LVD：*以低压检测 (LVD) 中断为例，演示了如何使用全局信号参考组件。*
- D_SysTick：*(仅适用于 PSoC 5LP) 演示了如何使用 Cortex-M3 CPU 中的 SysTick 中断。*

关于作者

姓名： Vivek Shankar Kannan

职务： 高级应用工程师

A PSoC 中的中断源

表 1 介绍了 PSoC 3 和 PSoC 5LP 中用于 32 个中断向量的中断源列表。此外，这个表格还提供了生成固定功能中断信号的 PSoC Creator 组件的相关信息。PSoC Creator 中不受支持的固定函数中断被标记为“不受支持”。

请注意，在表格中，UDB 中断源和 DMA nrq 中断源都没有相对应 PSoC Creator 组件名称。这是因为 PSoC Creator 基于一套复杂的规则 (如数字信号的走线) 将中断向量动态分配给 DMA 和 UDB 中断源。这些中断需要通过其内部信号名称了解它们。由于 PSoC Creator 内部控制这些内部信号的分配情况，所以您不必要了解它们的详细信息。

对于 PSoC 5LP，中断向量 0 到 31 与 Cortex-M3 的异常编号 16 到 47 相应。更多有关信息，请参见 PSoC 5LP 技术参考手册 (TRM)。

表 1. PSoC 3 和 PSoC 5LP 的中断源

中断向量编号	固定功能的中断源		DMA nrq 中断源	UDB 中断源
	中断源	PSoC Creator 组件		
0	低压检测 (LVD)	全局信号参考源	phub_termout0[0]	udb_intr[0]
1	缓存	全局信号参考源	phub_termout0[1]	udb_intr[1]
2	保留	不适用	phub_termout0[2]	udb_intr[2]
3	电源管理程序	RTC、睡眠定时器、全局信号参考源	phub_termout0[3]	udb_intr[3]
4	PICU[0]	数字输入引脚、数字双向引脚	phub_termout0[4]	udb_intr[4]
5	PICU[1]		phub_termout0[5]	udb_intr[5]
6	PICU[2]		phub_termout0[6]	udb_intr[6]
7	PICU[3]		phub_termout0[7]	udb_intr[7]
8	PICU[4]		phub_termout0[8]	udb_intr[8]
9	PICU[5]		phub_termout0[9]	udb_intr[9]
10	PICU[6]		phub_termout0[10]	udb_intr[10]
11	PICU[12]		phub_termout0[11]	udb_intr[11]
12	PICU[15]		phub_termout0[12]	udb_intr[12]
13	复合的比较器	不受支持	phub_termout0[13]	udb_intr[13]
14	复合的开关电容	不受支持	phub_termout0[14]	udb_intr[14]
15	I2C	I2C	phub_termout0[15]	udb_intr[15]
16	CAN	CAN	phub_termout1[0]	udb_intr[16]
17	定时器/计数器 0	定时器、计数器、PWM (脉冲宽度调制器)	phub_termout1[1]	udb_intr[17]
18	定时器/计数器 1	定时器、计数器、PWM (脉冲宽度调制器)	phub_termout1[2]	udb_intr[18]
19	定时器/计数器 2	定时器、计数器、PWM (脉冲宽度调制器)	phub_termout1[3]	udb_intr[19]
20	定时器/计数器 3	定时器、计数器、PWM (脉冲宽度调制器)	phub_termout1[4]	udb_intr[20]
21	USB SOF 中断	USBFS	phub_termout1[5]	udb_intr[21]
22	USB 仲裁器中断		phub_termout1[6]	udb_intr[22]
23	USB 总线中断		phub_termout1[7]	udb_intr[23]
24	USB Endpoint[0]		phub_termout1[8]	udb_intr[24]
25	USB 端点数据		phub_termout1[9]	udb_intr[25]
26	保留	不适用	phub_termout1[10]	udb_intr[26]

中断向量编号	固定功能的中断源		DMA nrq 中断源	UDB 中断源
	中断源	PSoC Creator 组件		
27	LCD	段式 LCD	phub_termout1[11]	udb_intr[27]
28	DFB	滤波器	phub_termout1[12]	udb_intr[28]
29	抽取滤波器	Delta Sigma ADC	phub_termout1[13]	udb_intr[29]
30	PHUB 错误	不受支持	phub_termout1[14]	udb_intr[30]
31	EEPROM 故障	不受支持	phub_termout1[15]	udb_intr[31]

B PSoC 3 中的可重入函数

可重入函数是指能够由多个单独流程 (如“main”函数和 ISR) 同时执行的函数。例如，当“main”函数正在执行某个可重入函数时，如果发生中断，ISR 仍可调用该函数。

函数可重入性是 C 编程语言的标准特性，并受 PSoC 5LP 编译器的支持。但是，由于 PSoC 3 8051 CPU 的限制，Keil C51 编译器一般没有实现函数的可重入性。本节介绍了通过 Keil C51 编译器处理可重入函数的流程。

B.1 Keil C51 编译器的可重入性

由于 PSoC 3 中 8051 CPU 的堆栈空间和可用 RAM 空间有限制，Keil C51 编译器默认把各函数视为不可重入。Keil 编译器将函数参数和本地变量存储到固定的内存位置内。同时调用相同的函数会使用相同内存位置，因此彼此间可能会破坏函数参数和本地变量。

既然各函数的默认设置是不可重入的，那么，可通过编写代码使它们支持可重入性。Keil 编译器将创建一个用于存储该函数本地变量及参数的独立堆栈。调用函数时，将调整该堆栈的指针，以便准确执行该函数的多次调用。可出入栈被创建在 xdata 内存区 (即 PSoC 3 的 SRAM 存储器空间)。

如果PSoC Creator工程中出现某个重入函数，该工程中*KeilStart.a51* 文件将自动生成代码使能可重入栈并初始化栈指针：

```
XBPSTACK        EQU    1
XBPSTACKTOP     EQU    CYDEV_SRAM_SIZE
```

第一个语句指定该堆栈在 xdata 内存空间 (SRAM 存储器)。第二个语句将该堆栈顶地址分配给 SRAM 的最后一个字节。这是因为可重入堆栈的堆栈顶指针是向下递减的。由于将堆栈顶指针分配给 SRAM 的最后一个字节，因此该堆栈重写 SRAM 中的变量的风险变得最小。

B.2 PSoC Creator 的可重入支持

如何将 PSoC Creator 中的函数设为可重入的流程取决于该函数是由 PSoC Creator 生成的 API，用户定义的函数还是与用户创建的自定义组件相关联的函数。以下部分介绍了各个不同的流程。

B.2.1 将所生成的 API 函数设置为可重入函数

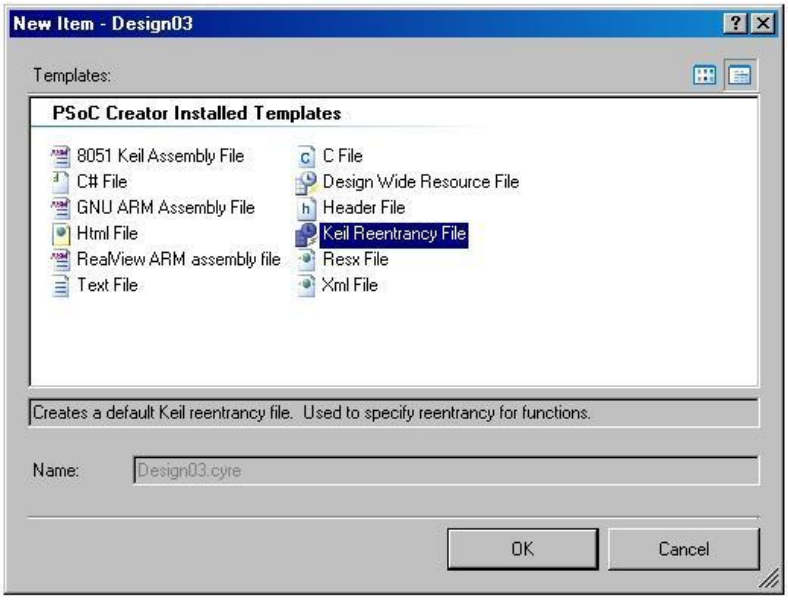
这些函数都与PSoC Creator所生成的源代码相关，如各个组件API函数。PSoC Creator允许您在“可重入文件” (即 *cyre*文件) 中指定应设为可重入的API函数。此文件指定了应设置为可重入的函数。工程编译时将自动标志可重入文件中指定的函数，以支持可重入性。

大多数API文件中的函数都可设置为可重入函数。如果不能将某个函数设置为可重入的，那么组件原文件中的注释会指示它是个不可重入的函数。这可能是由于该函数使用了全局变量或静态变量，或者调用了其他不可重入的函数。

要想将一个 *cyre* 文件添加到某个工程，请进行下面操作：

1. 在 Workspace Explorer (工作区浏览器) 中，右键点击项目，并依次选择 Add > New Item。
2. 在 New Item (新项目) 对话框中，选择‘Keil Re-entrancy File’项，然后点击 OK (请参见图 23)。*Project_Name.cyre* 文件将显示在代码编辑器中。

图 23. 将一个 *cyre* 文件添加到某个项目中



要想进入可重入函数，请进行以下操作：

3. 在 *cyre* 文件中，请在每一行上输入一个函数名称 (每一行上仅输入一个函数名称)。请注意，仅需要将函数名称输入到 *cyre* 文件中，而且函数名称区分大小写。函数的返回类型或用于表示一个函数的括号输入不需要放在 *cyre* 文件中。例如：

```
ADC_Start
PWM_Start
```

4. 完成后，请保存 cyre 文件。

B.2.2 将用户定义的函数设置为可重入函数

作为您应用代码的一部分，这些函数都是由您定义的。*cyre*文件不能用于将用户定义的函数指定为可重入函数。要想支持用户定义函数的可重入性，请将*cytypes.h*文件中的CYREENTRANT宏用于函数原型和函数定义。对于Kei工具链而言，它将参考REENTRANT关键字，而对于其他工具链，将没有关键字。这样允许在多个工具链和多个器件系列中正确地编译各函数。以下示例说明了如何将用户定义的函数Foo设置为可重入函数。

```
void Foo(void) CYREENTRANT;

void Foo(void) CYREENTRANT
{
  . . .
}
```

B.2.3 将自定义组件 API 设置为可重入函数

这些函数是用户创建的组件源文件和头文件的一部分。PSoC Creator允许您创建并使用自己的组件。

当创建一个自定义的组件时，可以使用编译表达式将该组件源文件和头文件中的任何函数设置为可重入函数。

```
`=ReentrantKeil($INSTANCE_NAME . "_FunctionName")`
```

使用实际的函数名称代替编译表达式中的“FunctionName”；所有其他字段必须保持不变。*.h* 文件中的函数声明和 *.c* 文件中的函数定义必须包含该表达式。这允许该函数的运行方式与 PSoC Creator 随附的其他组件的运行方式相同。默认情况下，它是一个标准的函数；但是，通过向**.cyre* 文件中添加函数名称，用户可以将它设置为可重入函数 (请参见将所生成的 API 函数设置为可重入函数节中介绍的内容)。

原始的函数声明和定义：

```
void `$INSTANCE_NAME`_Foo (void);

void `$INSTANCE_NAME`_Foo (void)
{
    . . .
}
```

经过修改后的可重入函数为：

```
void `$INSTANCE_NAME`_Foo(void) `=ReentrantKeil($INSTANCE_NAME . "_Foo")`;

void `$INSTANCE_NAME`_Foo(void) `=ReentrantKeil($INSTANCE_NAME . "_Foo")`
{
  . . .
}
```

B.3 确定可重入函数

仅在Keil编译器为某个函数分配RAM空间并需要并发调用该函数时，您才需要将它标志为可重入函数。Keil编译器可以帮助确定需要标志为可重入的函数。当将优化级别设置为 2 或更高级别，并且完成编译过程时，编译器将为每个被同时调用但未标记为可重入的函数产生一个编译警告。

```
Warning: L15 MULTIPLE CALL TO FUNCTION MYFUNCTION/MAIN ?C_C51STARTUP
ISR_1_INTERRUPT/ISR_1
```

在本警告信息中，`L15 MULTIPLE CALL TO FUNCTION` 是指可重入性的警告。`MYFUNCTION/MAIN` 指的是 *`main.c`* 文件中所定义的 `MyFunction` 函数可以同时调用。主代码是 `MyFunction` 函数的调用方之一。Keil 编译器使用`?C_C51STARTUP` 标记来表示起源于 `main()` 函数的主执行流。

`MyFunction` 的第二个调用程序是 *isr_1.c* 文件中的 `isr_1_Interrupt` 函数。该函数由 `ISR_1_INTERRUPT/ISR_1` 指示。它是中断服务子程序。因此，需要将 `MyFunction` 设置为可重入函数。请注意，即使在不同的情况下，Keil 编译器提示的可重入警告总是用大写字母显示函数名称和文件名称。

以下是其他警告信息示例：

WARNING: L15: MULTIPLE CALL TO FUNCTION DELAY/TIMING ISR_1/INTERRUPT_1 ISR_2/INTERRUPT_2

在本警告信息中，两个中断服务子程序 (*`Interrupt_1.c`* 文件中的 `isr_1()` 函数和 *`Interrupt_2.c`* 中的 `isr_2()` 函数) 同时调用 *`Timing.c`* 文件中的 `Delay()` 函数。因此，必须将 `Delay()` 设置为可重入函数。

警告信息的格式取决于它们如何显示在 PSoC Creator 的 Notice List (通知列表) 窗口内。Output (输出) 窗口上的格式与示例的“map”文件中的格式略有不同。在 Output (输出) 窗口和映射文件中，先前的警告使用下面的格式显示：

Warning: L15 MULTIPLE CALL TO FUNCTION NAME: MYFUNCTION/MAIN CALLER1: ?C_C51STARTUP CALLER2: ISR_1_INTERRUPT/ISR_1

即使是无需考虑可重入的情况，可重入警告也不可忽略。例如，当主代码调用函数时，可能会禁用中断。这样，主代码和 ISR 不可同时调用该函数。然而，Keil 链接器无法识别这些情况。它仍提示发现同时调用函数的可重入警告。如果忽略了可重入警告，即使是在不会产生可重入问题的情况下，可能会导致链接器在分配数据过程中为不同的执行流分配相同的存储器位置。这样会使数据损坏，并影响代码的功能。

文档修订记录

文档标题：AN54460 — PSoC® 3 和 PSoC 5LP 中断

文档编号：001-89533

版本	ECN	变更者	提交日期	变更说明
**	4145815	BOBH	10/03/2013	本文档版本号为Rev. **，译自英文版 001-54460 Rev. *F。
*A	4718356	RLJW	04/17/2015	本文档版本号为Rev. *A，译自英文版 001-54460 Rev. *G。
*B	5475297	VVSK	10/14/2016	已更新为新模板。 完成日落评论。
*C	5831951	AESATMP8	07/25/2017	更新标志和版权。
*D	6470804	XITO	02/01/2019	本文档版本号为Rev. *D，译自英文版 001-54460 Rev. *I。

销售、解决方案以及法律信息

全球销售和设计支持

赛普拉斯公司具有一个由办事处、解决方案中心、厂商代表和经销商组成的全球性网络。要想查找离您最近的办事处，请访问赛普拉斯所在地。

产品

Arm® Cortex® 微控制器	cypress.com/arm
汽车级产品	cypress.com/automotive
时钟与缓冲器	cypress.com/clocks
接口	cypress.com/interface
物联网	cypress.com/iot
存储器	cypress.com/memory
微控制器	cypress.com/mcu
PSoC	cypress.com/psoc
电源管理 IC	cypress.com/pmic
触摸感应	cypress.com/touch
USB 控制器	cypress.com/usb
无线连接	cypress.com/wireless

PSoC®解决方案

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

赛普拉斯开发者社区

社区 | 项目 | 视频 | 博客 | 培训 | 组件

技术支持

cypress.com/support

Arm and Cortex are registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

© 赛普拉斯半导体公司，2016-2019 年。本文件是赛普拉斯半导体公司及其子公司，包括 Spansion LLC（“赛普拉斯”）的财产。本文件，包括其包含或引用的任何软件或固件（“软件”），根据全球范围内的知识产权法律以及美国与其他国家签署条约由赛普拉斯所有。除非在本款中另有明确规定，赛普拉斯保留在该等法律和条约下的所有权利，且未就其专利、版权、商标或其他知识产权授予任何许可。如果软件并不附随有一份许可协议且贵方未以其他方式与赛普拉斯签署关于使用软件的书面协议，赛普拉斯特此授予贵方属人性质的、非独家且不可转让的如下许可 (无再许可权) (1) 在赛普拉斯特软件著作权项下的下列许可权 (一) 对以源代码形式提供的软件，仅出于在赛普拉斯硬件产品上使用之目的且仅在贵方集团内部修改和复制软件，和 (二) 仅限于在有关赛普拉斯硬件产品上使用之目的将软件以二进制代码形式的向外部最终用户提供 (无论直接提供或通过经销商和分销商间接提供)，和 (2) 在被软件 (由赛普拉斯公司提供，且未经修改) 侵犯的赛普拉斯专利的权利主张项下，仅出于在赛普拉斯硬件产品上使用之目的制造、使用、提供和进口软件的许可。禁止对软件的任何其他使用、复制、修改、翻译或汇编。

在适用法律允许的限度内，赛普拉斯未对本文件或任何软件作出任何明示或暗示的担保，包括但不限于关于适销性和特定用途的默示保证。没有任何电子设备是绝对安全的。因此，尽管赛普拉斯在其硬件和软件产品中采取了必要的安全措施，但是赛普拉斯并不承担任何由于使用赛普拉斯产品而引起的安全问题及安全漏洞的责任，例如未经授权的访问或使用赛普拉斯产品。此外，本材料中所介绍的赛普拉斯产品有可能存在设计缺陷或设计错误，从而导致产品的性能与公布的规 格不一致。 (如果发现此类问题，赛普拉斯会提供勘误表) 赛普拉斯保留更改本文件的权利，届时将不另行通知。在适用法律允许的限度内，赛普拉斯不对因应用或使用本文件所述任何产品或电路引起的任何后果负责。本文件，包括任何样本设计信息或程序代码信息，仅为供参考之目的提供。文件使用人应负责正确设计、计划和测试信息应用和由此生产的任何产品的功能和安全性。赛普拉斯产品不应被设计为、设定为或授权用作武器操作、武器系统、核设施、生命支持设备或系统、其他医疗设备或系统 (包括急救设备和手术植入物)、污染控制或有害物质管理系统中的关键部件，或产品植入之设备或系统故障可能导致人身伤害、死亡或财产损失其他用途（“非预期用途”）。关键部件指，若该部件发生故障，经合理预期会导致设备或系统故障或会影响设备或系统安全性和有效性的部件。针对由赛普拉斯产品非预期用途产生或相关的任何主张、费用、损失和其他责任，赛普拉斯不承担全部或部分责任且贵方不应追究赛普拉斯之责任。贵方应赔偿赛普拉斯因赛普拉斯产品任何非预期用途产生或相关的所有索赔、费用、损失和其他责任，包括因人身伤害或死亡引起的主张，并使之免受损失。

赛普拉斯、赛普拉斯徽标、Spansion、Spansion 徽标，及上述项目的组合，WICED，及 PSoC、CapSense、EZ-USB、F-RAM 和 Traveo 应视为赛普拉斯在美国和其他国家的商标或注册商标。请访问 cypress.com 获取赛普拉斯商标的完整列表。其他名称和品牌可能由其各自所有者主张为该方财产。