

7 独家秘方——Task，Event 以及 Interrupt

nRF51822 中引入了更容易理解的 Event & Task 机制。Task 是命令外设（TWI、I2C、RADIO 之类）执行任务，Event 是外设汇报上来的执行过程中的各种事件（状态）。另外，event 也与 Interrupt（中断）有关。

本章中涉及的 Interrupt 源代码，也是官方例程中没有提到的，所以这里的“独家配方”是个双关语。

7.1 Task

7.2 Event

7.3 Interrupt

中断是由 event 触发的，可以打断 CPU 的程序执行。APB 总线（详见系统整体框图）上的所有外设都支持中断。一个外设只有一个中断，中断号和外设 ID 相同。比如一个外设的 ID=4（），那么它在中断向量管理器(NVIC)中就对应 4 号中断。

INTEN 寄存器控制外设中的哪个 event 产生中断，可以设置多个 event 都产生中断。软件通过读取 event 寄存器，来查询中断产生的原因。

INTEN 寄存器中的每个 bit 对应不同的 event，地址 0x100 上的 event 对应 bit0，地址 0x104 上的 event 对应 bit1...地址 0x17C 上的 event 对应 bit31。最大只支持 32 个 event。

INTENSET 和 INTENCLR 寄存器支持“set and clear”操作。

7.4 关系表格

Register	Offset	Description
TASKS		
{TASK0}	0x000	Description of the first task
{TASK1}	0x004	Description of the second task
<>		
{TASK31}	0x07C	Description of the 32nd task (last task)
EVENTS		
{EVENT0}	0x100	Description of the first event
{EVENT1}	0x104	Description of the second event
<>		
{EVENT31}	0x17C	Description of the 32nd event (last event)
REGISTERS		
SHORTS	0x200	Shortcut register
INTENSET	0x304	Interrupt enable set register
INTENCLR	0x308	Interrupt enable clear register
{REG0}	0x400	First generic register
<>	<>	<>
{REGN}	0x7FC	Last generic register

7.5 Interrupt 代码样例

我们依旧以第三章 Hello, world 中的 UART 例程进行修改，官方例程使用轮询的工作方式，我们这里将他改为中断接收方式。

7.5.1 打开 uart_example 例程

（安装 SDK 后，源代码位于<keil path>\ARM\Device\Nordic\nRF51822\Board\PCA10001\uart_example\）

7.5.2 使能 UART 中断

在 main()函数中加入如下所示代码。

```
int main(void)
{
    simple_uart_config(RTS_PIN_NUMBER, TX_PIN_NUMBER, CTS_PIN_NUMBER,
RX_PIN_NUMBER, HWFC);

    /*-----插入的代码块-----*/
    // Enable UART interrupt
    NRF_UART0->INTENCLR = 0xffffffffUL;
    NRF_UART0->INTENSET = (UART_INTENSET_RXDRDY_Set <<
UART_INTENSET_RXDRDY_Pos) |
```

```

        (UART_INTENSET_TXDRDY_Set <<
UART_INTENSET_TXDRDY_Pos) |
        (UART_INTENSET_ERROR_Set <<
UART_INTENSET_ERROR_Pos);

    NVIC_ClearPendingIRQ(UART0_IRQn);
    NVIC_SetPriority(UART0_IRQn, 0);
    NVIC_EnableIRQ(UART0_IRQn);
    /*-----*/

    .....
}

```

7.5.3 注释掉 main 函数中轮询接收的代码

如下所示：

```

int main(void)
{
    .....
    simple_uart_putstring((const uint8_t *) " \n\Hello, world ");
    while(true)
    {
        /*-----注释的代码块-----*/
        /*uint8_t cr = simple_uart_get();
        simple_uart_put(cr);

        if(cr == 'q' || cr == 'Q')
        {
            uart_quit();
            while(1){}
        }*/
        /*-----注释的代码块-----*/
    }
    .....
}

```

7.5.4 加入 IRQ

在 main() 函数后面，加上如下中断服务代码：

```

/**@brief UART Interrupt handler.
 *
 * @details UART interrupt handler to process TX Ready when TXD is
available, RX Ready when a byte
 *         is received, or in case of error when receiving a byte.
 */
void UART0_IRQHandler(void)
{
    // Handle reception
    if (NRF_UART0->EVENTS_RXDRDY != 0)
    {
        // Clear UART RX event flag
        NRF_UART0->EVENTS_RXDRDY = 0;

        uint8_t cr = (uint8_t)NRF_UART0->RXD;
    }
}

```

```

        simple_uart_put(cr);

        if(cr == 'q' || cr == 'Q')
        {
            uart_quit();
        }
    }
}

```

7.5.5 代码缺陷

有些朋友可能会质疑这段代码，的确，这段代码有问题！nRF51822 利用串口中断接收字符后，发送回 PC 端的 simple_uart_put()函数如下：

```

void simple_uart_put(uint8_t cr)
{
    NRF_UART0->TXD = (uint8_t)cr;

    while (NRF_UART0->EVENTS_TXDRDY!=1)
    {
        // Wait for TXD data to be sent
    }

    NRF_UART0->EVENTS_TXDRDY=0;
}

```

显然，该函数内有一个循环等待的 while()函数。在中断服务程序中引入等待（或称为“阻塞”）函数是不正确的，当收到大量数据时，可能会因为中断服务函数没有退出（可能在等待），而导致丢失数据等问题。另外，如果允许中断嵌套的话，那么当前中断没有执行完毕，就可能被第二个中断所打断。

改进的办法有很多种，比如将发送函数移出中断服务，或者我们还可以采用实时（Run-Time）操作系统。在后面的章节中，我们将会讲述如何在 nRF51822 上运行实时（Run-Time）操作系统。

其他 IRQ

在 nRF51.h 中定义了其他 IRQ，请看：

```

/* ----- Interrupt Number Definition -----
----- */

typedef enum {
/* ----- Cortex-M0 Processor Exceptions Numbers -----
----- */
    Reset_IRQn            = -15,
    /*!<  1  Reset Vector, invoked on Power up and warm reset      */
    NonMaskableInt_IRQn   = -14,
    /*!<  2  Non maskable Interrupt, cannot be stopped or preempted */
    HardFault_IRQn        = -13,
    /*!<  3  Hard Fault, all classes of Fault                      */
    SVCCall_IRQn          = -5,
    /*!< 11  System Service Call via SVC instruction               */
    DebugMonitor_IRQn     = -4,
    /*!< 12  Debug Monitor                                         */
}

```

```

PendSV_IRQn          = -2,
/*!< 14 Pendable request for system service          */
SysTick_IRQn         = -1,
/*!< 15 System Tick Timer                          */
/* ----- nRF51 Specific Interrupt Numbers ----- */
----- */
POWER_CLOCK_IRQn     = 0,
/*!< 0 POWER_CLOCK
*/
RADIO_IRQn           = 1,
/*!< 1 RADIO
*/
UART0_IRQn           = 2,
/*!< 2 UART0
*/
SPI0_TWI0_IRQn       = 3,
/*!< 3 SPI0_TWI0
*/
SPI1_TWI1_IRQn       = 4,
/*!< 4 SPI1_TWI1
*/
GPIOTE_IRQn          = 6,
/*!< 6 GPIOTE
*/
ADC_IRQn             = 7,
/*!< 7 ADC
*/
TIMER0_IRQn          = 8,
/*!< 8 TIMER0
*/
TIMER1_IRQn          = 9,
/*!< 9 TIMER1
*/
TIMER2_IRQn          = 10,
/*!< 10 TIMER2
*/
RTC0_IRQn            = 11,
/*!< 11 RTC0
*/
TEMP_IRQn            = 12,
/*!< 12 TEMP
*/
RNG_IRQn             = 13,
/*!< 13 RNG
*/
ECB_IRQn             = 14,
/*!< 14 ECB
*/
CCM_AAR_IRQn         = 15,
/*!< 15 CCM_AAR
*/
WDT_IRQn             = 16,
/*!< 16 WDT
*/
RTC1_IRQn            = 17,
/*!< 17 RTC1
*/
QDEC_IRQn            = 18,

```

```
    /*!< 18 QDEC
*/
    LPCOMP_COMP_IRQn          = 19,
    /*!< 19 LPCOMP_COMP
*/
    SWI0_IRQn                 = 20,
    /*!< 20 SWI0
*/
    SWI1_IRQn                 = 21,
    /*!< 21 SWI1
*/
    SWI2_IRQn                 = 22,
    /*!< 22 SWI2
*/
    SWI3_IRQn                 = 23,
    /*!< 23 SWI3
*/
    SWI4_IRQn                 = 24,
    /*!< 24 SWI4
*/
    SWI5_IRQn                 = 25
    /*!< 25 SWI5
*/
} IRQn_Type;
```