华清远见 FARSIGHT 始于2004 | 嵌入式培训专家

10年口碑积累，成功培养50000多名研发工程师，铸就专业品牌形象

华清远见的企业理念是不仅要做良心教育、做专业教育，更要做受人尊敬的职业教育。

《CORTEX-M3+UCOS-II 嵌入式系统开发入门与应用》

作者：华清远见

专业始于专注 卓识源于远见

第 5 章 STM32F103 处理器内部资源 C 编程与实例

专业始于专注 卓识源于远见

5.1 I/O 控制模块 C 编程与实例

5.1.1 实例内容与目标

- 熟悉 STM32F103 处理器 I/O 编程方法；
- 通过实验掌握 STM32F103 处理器 I/O 控制 LED 显示的方法；
- 学习 LED 驱动原理。

5.1.2 I/O 控制模块操作原理

（1）STM32F103 处理器 I/O 控制工作原理。

STM32F103 处理器有 A～E 5 组输入/输出端口，每组端口都拥有自己的 2 个 32bit 寄存器（GPIOx_CRL 和 GPIOx_CRH）、2 个 32bit 数据寄存器（GPIOx_IDR 和 GPIOx_ODR）、1 个 32bit set/reset 寄存器（GPIOx_BSRR）、1 个 16bit reset 寄存器（GPIOx_BRR）和 1 个 32bit 锁定寄存器（GPIOx_LCKR）。

每组端口都有复用的功能，例如可以作为输入/输出端口，还可以定义为中断触发功能，用户可以通过软件配置寄存器来满足不同系统和设计的需要。在运行主程序之前，必须先对每一个用到的引脚的功能进行设置。如果某些引脚的复用功能没有使用，那么可以先将该引脚设置为通用 I/O 端口。

I/O 端口的配置过程类似，因此接下来主要以 GPIO E 端口作为对象来讲解端口配置的方法。

- 端口配置低寄存器

表 5.1 所示显示了低寄存器的定义。

表 5.1　　端口配置低寄存器定义

比　特　位	定　　义
Bits：31：30，27：26，23：22，19：18，15：14，11：10，7：6，3：2	软件可写位，配置相应的 I/O 端口。 输入模式（mode[1：0] = 00）： 00：模拟输入模式 01：浮点输入 10：输入上拉/下拉 11：保留位 输出模式（mode[1：0]>00）： 00：通用功能输出上拉 01：通用功能输出 Open-drain 10：转换功能输出上拉 11：转换功能输出 Open-drain
Bits29：28，25：24，21：20，17：16，13：12，9：8，5：4，1：0	软件可写位，配置相应的 I/O 端口。 00：输入模式 01：输出模式，最大速率 10MHz 10：输出模式，最大速率 2MHz 11：输出模式，最大速率 50MHz

- 端口配置高寄存器

表 5.2 所示为端口配置高寄存器的定义。

表 5.2　　端口配置高寄存器定义

比　特　位	定　　义
Bits：31：30，27：26，23：22，19：18，15：14，11：10，7：6，3：2	软件可写位，配置相应的 I/O 端口。 00：模拟输入模式 01：浮点输入

	10：输入上拉/下拉 11：保留位
Bits29：28，25：24，21：20，17：16，13：12，9：8，5：4，1：0	软件可写位，配置相应的 I/O 端口。 输入模式（mode[1：0] = 00）： 00：模拟输入模式 01：浮点输入 10：输入上拉/下拉 11：保留位 输出模式（mode[1：0]>00）： 00：通用功能输出上拉 01：通用功能输出 Open-drain 10：转换功能输出上拉 11：转换功能输出 Open-drain

- 端口输入数据寄存器

GPIO 端口数据可以通过该寄存器读入。表 5.3 所示为端口输入数据寄存器的定义。

表 5.3　　端口输入数据寄存器定义

Bits31：16	保留位，读出数据为 0
Bits：15：0	IDRx[15：0]：端口输入数据（x = 0, …, 15） 该寄存器为只读寄存器并且只能在字模式下访问；读出数据为相应 I/O 端口的输入值

- 端口输出数据寄存器

地址偏移量：0Ch；初始值：00000000h。

GPIO 端口数据可以通过该寄存器输出。表 5.4 所示为端口输出数据寄存器的定义。

表 5.4　　端口输出寄存器定义

Bits31：16	保留位，读出数据为 0
Bits：15：0	IDRx[15：0]：端口输入数据（x = 0, …, 15） 该寄存器为可读/可写寄存器，只能在字模式下访问。 注意：该寄存器允许原子操作（设置/重启），每一位允许单独操作。通过对 GPIOx_BSRR 寄存器的操作，可以设置或重启任一比特位

- 比特位设置/重启寄存器 GPIOx_BSRR

地址偏移量：10h；初始化值：00000000h。

表 5.5 所示为比特位设置/重启寄存器的定义。

表 5.5　　比特位设置/重启寄存器定义

Bits31：16	BRx：reset bit x（x = 1, …, 15） 该寄存器为只写寄存器，而且仅能在字模式下访问。 0：对相应 ODRx 位不做任何操作。 1：重启 ODRx 相应位。 注意：如果同时设置了 BSx 和 BRx，BSx 具有较高优先级
Bits：15：0	BSx：相应的 x 位置位（x = 1, …, 15） 该寄存器为只写寄存器而且仅能在字模式下访问。 0：对相应 ODRx 位不做任何操作。 1：ODRx 相应位置位

- 端口位重启寄存器 GPIOx_BRR

地址偏移量：14h；初始化值：00000000h。

表 5.6 所示为端口位重启寄存器的定义。

表 5.6	端口位重启寄存器定义
Bits31：16	保留位
Bits：15：0	BSx：重启相应的 x 位（x = 1, …, 15） 该寄存器为只写寄存器，而且仅能在字模式下访问。 0：对相应 ODRx 位不做任何操作； 1：重启相应 ODRx 位。 注意：如果同时设置了 BSx 和 BRx，BSx 具有较高优先级

- 端口配置锁定寄存器 GPIOx_LCKR

当该寄存器的 bit16（LCKK）通过正确的序列写入时，I/O 端口位配置被锁定。Bit[15：0]用于锁定 GPIO 的配置。当对 bit16 进行写操作时，LCKR[15：0]不允许变化。

配置锁定寄存器的每一位对应控制寄存器（CRL，CRH）的 4 个 bit 位。

地址偏移量：18h；初始化值：00000000h。

（2）LED 结构及发光原理。

50 年前人们已经了解半导体材料可产生光线的基本知识，第一个商用二极管产生于 1960 年。LED（Light Emitting Diode，发光二极管）的基本结构是一块电致发光的半导体材料，置于一个有引线的架子上，然后四周用环氧树脂密封，起到保护内部芯线的作用，所以 LED 的抗震性能好。

LED 结构如图 5.1 所示。

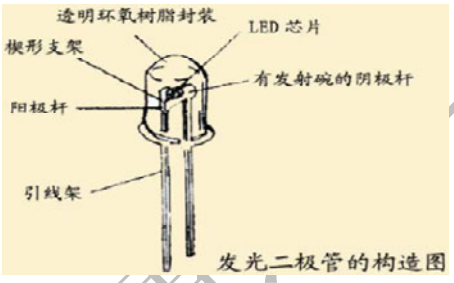

图 5.1　LED 结构图

发光二极管的核心部分是由 P 型半导体和 N 型半导体组成的晶片，在 P 型半导体和 N 型半导体之间有一个过渡层，称为 PN 结。在某些半导体材料的 PN 结中，注入的少数载流子与多数载流子复合时会把多余的能量以光的形式释放出来，从而把电能直接转换为光能。PN 结加反向电压，少数载流子难以注入，故不发光。这种利用注入式电致发光原理制作的二极管叫发光二极管，通称 LED。当它处于正向工作状态时（即两端加上正向电压），电流从 LED 阳极流向阴极时，半导体晶体就发出从紫外到红外不同颜色的光线，光的强弱与电流有关。

- LED 光源的特点

① 电压：LED 使用低压电源，供电电压在为 6V～24V，根据产品不同而异，所以它是一个比使用高压电源更安全的电源，特别适用于公共场所。

② 效能：消耗能量较同光效的白炽灯减少 80%。

③ 适用性：其体积很小，每个单元 LED 小片是 3～5mm^2 的正方形，所以可以制备成各种形状的器件，并且适合于易变的环境。

④ 稳定性：连续工作 10 万小时，光衰为初始的 50%。

⑤ 响应时间：白炽灯的响应时间为毫秒级，LED 灯的响应时间为纳秒级。

⑥ 对环境污染：无有害金属汞。

⑦ 颜色：改变电流可以变色，发光二极管方便地通过化学修饰方法，调整材料的能带结构和带隙，实现红黄绿兰橙多色发光。如小电流时为红色的 LED，随着电流的增加，可以依次变为橙色、黄色，最后为绿色。

⑧ 价格：LED 的价格比较昂贵，较之于白炽灯，几只 LED 的价格就可以与一只白炽灯的价格相当，而通常每组信号灯需由上 300～500 只二极管构成。

- 单色光 LED 的种类及其发展史

最早应用于半导体 P-N 结发光原理制成的 LED 光源问世于 20 世纪 60 年代初。当时所用的材料是 GaAsP，发红光（$\lambda_p = 650nm$），在驱动电流为 20mA 时，光通量只有千分之几个流明，相应的发光效率约 0.1 流明/瓦。

20 世纪 70 年代中期，引入元素 In 和 N，使 LED 产生绿光（$\lambda_p = 555nm$）、黄光（$\lambda_p = 590nm$）和橙光（$\lambda_p = 610nm$），光效也提高到 1 流明/瓦。

到了 20 世纪 80 年代初，出现了 GaAlAs 的 LED 光源，使得红色 LED 的光效达到 10 流明/瓦。

20 世纪 90 年代初，发红光、黄光的 GaAlInP 和发绿、蓝光的 GaInN 两种新材料的开发成功，使 LED 的光效得到大幅度的提高。在 2000 年，前者做成的 LED 在红、橙区（$\lambda_p = 615nm$）的光效达到 100 流明/瓦，而后者制成的 LED 在绿色区域（$\lambda_p = 530nm$）的光效可以达到 50 流明/瓦。

- 单色光 LED 的应用

最初 LED 用作仪器仪表的指示光源，后来各种光色的 LED 在交通信号灯和大面积显示屏中得到了广泛应用，产生了很好的经济效益和社会效益。以 12 英寸的红色交通信号灯为例，本来是采用长寿命、低光效的 140 瓦白炽灯作为光源，它产生 2 000 流明的白光。经红色滤光片后，光损失 90%，只剩下 200 流明的红光。而在新设计的灯中，Lumileds 公司采用了 18 个红色 LED 光源，包括电路损失在内，共耗电 14W，即可产生同样的光效。

汽车信号灯也是 LED 光源应用的重要领域。1987 年，我国开始在汽车上安装高位刹车灯，LED 响应速度快（纳秒级），可以及早让后车司机知道行驶状况，减少汽车追尾事故的发生。

另外，LED 灯在室外红、绿、蓝全彩显示屏，匙扣式微型电筒等领域都得到了应用。

- LED 驱动

由于单只 LED 管的工作电压低（1.5V～2V），个别需达到 4V，同时工作电流仅为 1mA～5mA，因此可以用 CPU 的通用输入/输出管脚（GPIO）就可直接控制。

图 5.2 所示为 STM32V100 评估板中由 CPU 直接驱动控制 LED 的原理图。以 LD1 为例，它的阳极通过串联电阻接在 CPU 的 PC6 管脚上，阴极接地。假设 LD1 的正向导通压降为 1.5V，当 PC6 输出高电平时，LD1 将会有电流通过，并导致 LD1 发光。反之，当 PC6 输出低电平是，LD1 几乎没有电流通过，因此将不会发光。

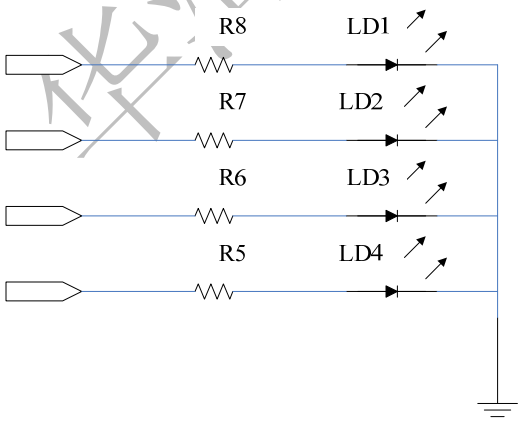

图 5.2　LED 硬件连接图

5.1.3　I/O 控制实例操作步骤

（1）准备实验环境：连接好主机——μlinker——目标板。
（2）启动 Keil μVersion 3，打开所需工程 GPIO.Uv2。
（3）选择 RAM 连接文件 gpio.sct。
（4）选择 RAM 调试文件 RAM.ini。
（5）使用“Debug——Debug session”加载 image 文件。
（6）选择“run”命令运行程序，观察 LED 灯闪烁现象。

（7）在主函数处设置断点，单步运行，注意 LED 灯相关寄存器的变化。
（8）如果有必要，可修改源代码，重新编译，然后使用 reload 命令重新调试。
（9）退出系统。

5.1.4 I/O 控制实例参考程序及说明

（1）GPIO 位写函数 GPIO_WriteBit()。
该函数设置/清除所选数据端口值。输入参数分别为 GPIOx、GPIO_Pin 和 BitVal，其中，GPIOx 指定所选 GPIO 组；GPIO_Pin 指定将要被修改的端口号；BitVal 为枚举值，当 BitVal 等于 Bit_RESET 时，表示要清除该位，等于 Bit_SET 时，表示要对该位置 1。
该函数主要实现程序代码如下：

```
void GPIO_WriteBit(GPIO_TypeDef* GPIOx, u16 GPIO_Pin, BitAction BitVal)
{
  /* 检测输入参数是否正确 */
  assert(IS_GET_GPIO_PIN(GPIO_Pin));
  assert(IS_GPIO_BIT_ACTION(BitVal));

  if (BitVal != Bit_RESET)
  {
    GPIOx->BSRR = GPIO_Pin;
  }
  else
  {
    GPIOx->BRR = GPIO_Pin;
  }
}
```

（2）GPIO 位设置函数 void GPIO_SetBits(GPIO_TypeDef* GPIOx, u16 GPIO_Pin)。
该函数对 GPIO 的某一位置位。输入参数分别为 GPIOx 和 GPIO_Pin，其中，GPIOx 指定所选 GPIO 组，GPIO_Pin 指定将要被修改的端口号。
该函数主要实现程序代码如下：

```
void GPIO_SetBits(GPIO_TypeDef* GPIOx, u16 GPIO_Pin)
{
  /* 检测输入参数是否正确 */
  assert(IS_GPIO_PIN(GPIO_Pin));
  GPIOx->BSRR = GPIO_Pin;
}
```

（3）实例主程序 main。
实例主程序以 main 为入口，该函数主体为 while(1)循环内的函数体，在循环内，程序通过不断设置 LED 灯相关 GPIO 端口，使 LED 灯实现不断的亮/灭循环。
main 函数代码如下：

```
int main(void)
{
#ifdef DEBUG
  debug();
#endif

  /* 配置系统时钟 */
  RCC_Configuration();
```

```
    /* NVIC 设置 */
    NVIC_Configuration();

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7 | GPIO_Pin_8 | GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    GPIO_SetBits(GPIOC, GPIO_Pin_6);

    /* 将 PB.09 配置为输入 */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    /* 检测 PB.09 输入是否为低(即按键被按下) */
    if (GPIO_ReadInputDataBit(GPIOB, GPIO_Pin_9) == 0x00)
    { /* 按键被按下 */

      /* 使串口 Stag 调试串口 SWJ-DP 无效 */
      GPIO_PinRemapConfig(GPIO_Remap_SWJ_Disable, ENABLE);

      /* 配置 PA.13 (JTMS/SWDAT)、PA.14 (JTCK/SWCLK)和 PA.15 (JTDI)为输入
         */
      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
      GPIO_Init(GPIOA, &GPIO_InitStructure);

      /* 配置 PB.03 (JTDO) 和 PB.04 (JTRST) 为输出 */
      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3 | GPIO_Pin_4;
      GPIO_Init(GPIOB, &GPIO_InitStructure);

      while (1)
      {
        /* JTMS/SWDAT pin 取反 */
        GPIO_WriteBit(GPIOA,  GPIO_Pin_13,  (BitAction)(1  -  GPIO_ReadOutputDataBit  (GPIOA,
GPIO_Pin_13)));
        /* 延时 */
        Delay(0x5FFFF);

        /* JTCK/SWCLK pin 取反 */
        GPIO_WriteBit(GPIOA,  GPIO_Pin_14,  (BitAction)(1  -  GPIO_ReadOutputDataBit  (GPIOA,
GPIO_Pin_14)));
        /* 延时 */
        Delay(0x5FFFF);

        /* JTDI pin 取反 */
        GPIO_WriteBit(GPIOA,  GPIO_Pin_15,  (BitAction)(1  -  GPIO_ReadOutputDataBit  (GPIOA,
GPIO_Pin_15)));
        /* 延时 */
        Delay(0x5FFFF);
```

```
        /* JTDO pin 取反 */
        GPIO_WriteBit(GPIOB,  GPIO_Pin_3,  (BitAction)(1  -  GPIO_ReadOutputDataBit  (GPIOB,
GPIO_Pin_3)));
        /* 延时 */
        Delay(0x5FFFF);

        /* JTRST pin 取反 */
        GPIO_WriteBit(GPIOB,  GPIO_Pin_4,  (BitAction)(1  -  GPIO_ReadOutputDataBit  (GPIOB,
GPIO_Pin_4)));
        /* 延时 */
        Delay(0x5FFFF);
      }
    }
    else
    {
      while (1)
      {
       GPIO_SetBits(GPIOC, GPIO_Pin_8);
      }
    }
  }
```

5.2 中断控制模块 C 编程与实例

5.2.1 实例内容与目标

该实例重点演示如何使用 TM32F103 处理器的 NVIC 控制处理器中断。
实例中配置了 3 个 TIM（TIM2、TIM3 和 TIM4），定时产生定时器更新事件，即定时器中断。其中，所配置的 3 个定时器均连接在 CPU 的 IRQ 中断上，并通过 NVIC 优先级配置寄存器分别配置了 3 个定时器中断的优先级：TIM2 优先级为 0，TIM 3 优先级为 1，TIM4 优先级为 2。同时，为了更清楚地说明中断产生，TIM2 每隔 1s 切换到 PC06 的 LED 灯上，TIM3 每隔 2s 切换到 PC07 的 LED 灯上，TIM4 每隔 3s 切换到 PC08 的 LED 灯上，这样，当实例正常执行时，通过 LED 的灯亮/灭可以判断是否有中断发生。
通过该实例，重点掌握以下内容：

- 熟悉 STM32F103 处理器的 NVIC 中断编程方法；
- 掌握中断程序的编写方法。

5.2.2 中断模块 NVIC 的操作原理

STM32F103 处理器中使用 NVIC（Nested Vectored Interrupt Controller，嵌套向量中断控件器）实现处理器的中断控制。NVIC 和 CPU 实现“紧耦合”，减少了中断响应时间。关于 NVIC 的更过内容，请参见本书第 2 章。
表 5.7 所示为 STM32F103 处理器的中断向量表和中断优先级。

表 5.7　　STM32F103 处理器的中断向量表和中断优先级定义

位置	优先级	优先级类型	中断名称	中断描述	向量地址
	-	-	-	保留	0x0000_0000

	−3	固定	Reset	系统重启	0x0000_0004
	−2	固定	NMI	非掩码中断。RCC 时钟安全系统（CSS）连接到此 NMI 向量上	0x0000_0008
	−1	固定	硬件错误	-	0x0000_000C
	0	可设	内存管理	内存管理错误	0x0000_0010
	1	可设	总线错误	预取错误，内存访问错误	0x0000_0014
	2	可设	使用错误	未定义指令或不合法状态	0x0000_0018
	-	-	-	保留	0x0000_001C-0x0000_002B
	3	可设	SVCall	系统 SWI 调用	0x0000_002C
	4	可设	调试监控	调试状态监控	0x0000_0030
	-	-	-	保留	0x0000_0034
	5	可设	PendSV	请求系统服务中断（可挂起）	0x0000_0038

续表

位置	优先级	优先级类型	中断名称	中断描述	向量地址
	6	可设	SysTick	系统时钟	0x0000_003C
0	7	可设	WWDG	窗口看门狗中断	0x0000_0040
1	8	可设	PVD	通过 EXTI 检测的 PVD 中断	0x0000_0044
2	9	可设	TAMPER	Tamper 中断	0x0000_0048
3	10	可设	RTC	RTC 全局中断	0x0000_004C
4	11	可设	Flash	Flash 全局中断	0x0000_0050
5	12	可设	RCC	RCC 全局中断	0x0000_0054
6	13	可设	EXTI0	EXTI 检测的线号 0 中断	0x0000_0058
7	14	可设	EXTI1	EXTI 检测的线号 1 中断	0x0000_005C
8	15	可设	EXTI2	EXTI 检测的线号 2 中断	0x0000_0060
9	16	可设	EXTI3	EXTI 检测的线号 3 中断	0x0000_0064
10	17	可设	EXTI4	EXTI 检测的线号 4 中断	0x0000_0068
11	18	可设	DMAChannel1	DMA 通道 1 全局中断	0x0000_006C
12	19	可设	DMAChannel2	DMA 通道 2 全局中断	0x0000_0070
13	20	可设	DMAChannel3	DMA 通道 3 全局中断	0x0000_0074
14	21	可设	DMAChannel4	DMA 通道 4 全局中断	0x0000_0078
15	22	可设	DMAChannel5	DMA 通道 5 全局中断	0x0000_007C
16	23	可设	DMAChannel6	DMA 通道 6 全局中断	0x0000_0080
17	24	可设	DMAChannel7	DMA 通道 7 全局中断	0x0000_0084
18	25	可设	ADC	ADC 全局中断	0x0000_0088
19	26	可设	USB_HP_CAN_TX	USB 高优先级中断或 CAN TX 发送中断	0x0000_008C
20	27	可设	USB_LP_CAN_RX0	USB 低优先级中断或 CAN RX0 接受中断	0x0000_0090
21	28	可设	CAN_RX1	CAN RX1 中断	0x0000_0094

22	29	可设	CAN_SCE	CAN SCE 中断	0x0000_0098
23	30	可设	EXTI9_5	EXTI 线[9：5]中断	0x0000_009C
24	31	可设	TIM1_BRK	TIM1 Break 中断	0x0000_00A0

续表

位置	优先级	优先级类型	中断名称	中断描述	向量地址
25	32	可设	TIM1_UP	TIM1 更新中断	0x0000_00A4
26	33	可设	TIM1_TRG_COM	TIM1 Trigger 和通信中断	0x0000_00A8
27	34	可设	TIM1_CC	TIM1 捕捉比较中断	0x0000_00AC
28	35	可设	TIM2	TIM2 全局中断	0x0000_00B0
29	36	可设	TIM3	TIM3 全局中断	0x0000_00B4
30	37	可设	TIM4	TIM4 全局中断	0x0000_00B8
31	38	可设	I2C1_EV	I^2C1 事件中断	0x0000_00BC
32	39	可设	I2C1_ER	I^2C1 错误中断	0x0000_00C0
33	40	可设	I2C2_EV	I^2C2 事件中断	0x0000_00C4
34	41	可设	I2C2_EV	I^2C2 错误中断	0x0000_00C8
35	42	可设	SPI1	SPI1 全局中断	0x0000_00CC
36	43	可设	SPI2	SPI2 可设中断	0x0000_00D0
37	44	可设	USART1	USART1 全局中断	0x0000_00D4
38	45	可设	USART2	USART2 全局中断	0x0000_00D8
39	46	可设	USART3	USART3 全局中断	0x0000_00DC
40	47	可设	EXTI15_10	EXTI 线[15：10]中断	0x0000_00E0
41	48	可设	RTCAlarm	通过 EXT1 的 RTC 闹钟中断	0x0000_00E4
42	49	可设	USBWakeUp	USB 唤醒中断或通过 EXTI 的可挂起唤醒中断	0x0000_00E8

使用 ARM 体系结构的芯片，中断编程的一般思路如下。

- 对 CPU 进行初始化，保证 CPU 能正常工作，注意：CPU 初始化时，一般都要屏蔽中断。
- 设置 IRQ 堆栈，如果使用了 FIQ 中断，那么还应设置 FIQ 堆栈。
- 设置 I/O，因为在实验中用到了外部中断，而外部中断和 I/O 共用。
- 设置中断模式，如在本次实验中，都设置为 IRQ 中断。
- 使能中断。
- 发生中断后，首先压栈保存现场，然后清中断，再关闭中断，执行中断服务程序，最后打开中断，出栈以恢复现场。

5.2.3 中断控制实例操作步骤

（1）准备实验环境，连接好主机-μlinker-目标板。
（2）启动 Keil μVersion 3，打开所需工程 NVIC.Uv2。
（3）选择 RAM 连接文件 nvic.sct。
（4）选择 RAM 调试文件 RAM.ini。
（5）使用“Debug-Debug session”加载 image 文件。
（6）选择“run”命令全速运行程序，观察 LED 灯闪烁现象。
（7）在主函数处设置断点，单步运行，注意 NCIC 相关寄存器的变化。
（8）如果有必要，可修改源代码，重新编译，然后使用 reload 命令重新调试。

（9）退出系统。

5.2.4 中断控制实例参考程序及说明

（1）优先级初始化结构 NVIC_InitTypeDef。

为了编程方便，程序根据 CPU 寄存器定义了 NVIC 初始化结构体 NVIC_InitTypeDef，其定义如下：

```
typedef struct
{
  u8 NVIC_IRQChannel;
  u8 NVIC_IRQChannelPreemptionPriority;
  u8 NVIC_IRQChannelSubPriority;
  FunctionalState NVIC_IRQChannelCmd;
} NVIC_InitTypeDef;
```

程序中对 NVIC 的初始化必须首先根据程序需要填充该结构体。

（2）NVIC 地址配置函数 NVIC_SetVectorTable()。

该函数设置 NVIC 寄存器在内存中的地址和偏移量。该函数源码如下：

```
void NVIC_SetVectorTable(u32 NVIC_VectTab, u32 Offset)
{
  /* 检测输入参数 */
  assert(IS_NVIC_VECTTAB(NVIC_VectTab));
  assert(IS_NVIC_OFFSET(Offset));

  SCB->ExceptionTableOffset = NVIC_VectTab | (Offset & (u32)0x1FFFFF80);
}
```

其中，输入变量 NVIC_VectTab 指定 NVIC 存在于内存空间的 Flash 还是 RAM 中，可选值为 NVIC_VectTab_RAM 和 NVIC_VectTab_FLASH；输入变量 Offset 指定在所选域的偏移量。

（3）NVIC 初始化函数 NVIC_Init()。

该函数通过初始化结构体 NVIC_InitTypeDef 对 NVIC 实行初始化操作，函数体为：

```
void NVIC_Init(NVIC_InitTypeDef* NVIC_InitStruct)
{
  u32 tmppriority = 0x00, tmpreg = 0x00, tmpmask = 0x00;
  u32 tmppre = 0, tmpsub = 0x0F;

  /* 检测输入参数 */
  assert(IS_FUNCTIONAL_STATE(NVIC_InitStruct->NVIC_IRQChannelCmd));
  assert(IS_NVIC_IRQ_CHANNEL(NVIC_InitStruct->NVIC_IRQChannel));
  assert(IS_NVIC_PREEMPTION_PRIORITY(NVIC_InitStruct->NVIC_IRQChannelPreemption Priority));
  assert(IS_NVIC_SUB_PRIORITY(NVIC_InitStruct->NVIC_IRQChannelSubPriority));

  if (NVIC_InitStruct->NVIC_IRQChannelCmd != DISABLE)
  {
    /* 比较排起的 IRQ 的优先级 ----------------------------------*/
    tmppriority = (0x700 - (SCB->AIRC & (u32)0x700))>> 0x08;
    tmppre = (0x4 - tmppriority);
    tmpsub = tmpsub >> tmppriority;

    tmppriority = (u32)NVIC_InitStruct->NVIC_IRQChannelPreemptionPriority << tmppre;
    tmppriority |=  NVIC_InitStruct->NVIC_IRQChannelSubPriority & tmpsub;

    tmppriority = tmppriority << 0x04;
```

```
    tmppriority = ((u32)tmppriority) << ((NVIC_InitStruct->NVIC_IRQChannel & (u8)0x03) * 0x08);

    tmpreg = NVIC->Priority[(NVIC_InitStruct->NVIC_IRQChannel >> 0x02)];
    tmpmask = (u32)0xFF << ((NVIC_InitStruct->NVIC_IRQChannel & (u8)0x03) * 0x08);
    tmpreg &= ~tmpmask;
    tmppriority &= tmpmask;
    tmpreg |= tmppriority;

    NVIC->Priority[(NVIC_InitStruct->NVIC_IRQChannel >> 0x02)] = tmpreg;

    /* 使能选择的 IRQ 通路 ----------------------------------------*/
    NVIC->Enable[(NVIC_InitStruct->NVIC_IRQChannel >> 0x05)] =
      (u32)0x01 << (NVIC_InitStruct->NVIC_IRQChannel & (u8)0x1F);
  }
  else
  {
    /* Disable the Selected IRQ Channels ----------------------------------*/
    NVIC->Disable[(NVIC_InitStruct->NVIC_IRQChannel >> 0x05)] =
      (u32)0x01 << (NVIC_InitStruct->NVIC_IRQChannel & (u8)0x1F);
  }
}
```

（4）实例主程序 main。

该函数首先设置了 NVIC 在内存中的地址，然后初始化 NVIC 并开始相应定时器中断。

```
int main(void)
{
#ifdef DEBUG
  debug();
#endif

  /* 配置系统时钟 */
  RCC_Configuration();

  /* 配置 GPIO */
  GPIO_Configuration();

  /* 配置 TIM TIM */
  TIM_Configuration();

#ifdef  VECT_TAB_RAM
  /* 将向量表基地址设置在 0x20000000 */
  NVIC_SetVectorTable(NVIC_VectTab_RAM, 0x0);
#else  /* VECT_TAB_FLASH  */
  /* 将向量表基地址设置在 0x08000000 */
  NVIC_SetVectorTable(NVIC_VectTab_FLASH, 0x0);
#endif

  /* 设置抢占优先级 */
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);

  /* 使能 TIM2 中断 */
  NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQChannel;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
```

```
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

    /* 使能 TIM3 中断 */
    NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQChannel;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
    NVIC_Init(&NVIC_InitStructure);

    /* 使能 TIM4 中断 */
    NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQChannel;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2;
    NVIC_Init(&NVIC_InitStructure);

    while (1)
    {
    }
  }
```

篇幅所限，关于定时器的相关设置，请参见 STM32F103 处理器手册。

5.3 A/D 转换和 DMA 编程实例

5.3.1 实例内容与目标

该实例是 A/D 转换控制和 DMA 操作的综合实例。实例中连续使用 ADC 控制器和 DMA 传输通道，将通过数/模转换的数据转入数据缓存区。程序使用 ADC 的 14 号通道，连续读取外部模拟信号并将其转化为数据信号。每次数/模转换结束后，激活 DMA 中断传输数据，将 ADC1 DR 寄存器的值传入自定义变量 ADC_ConvertedValue，转换过程中将 ADC1 的时钟频率设为 14MHz。

通过该实例，重点掌握以下内容：

- 了解 A/D 控制原理与 DMA 传输数据的基本方法；
- 掌握 A/D 控制器与 DMA 控制单元驱动的编写方法；
- 进一步了解 TM32F103 处理器中断处理方法。

5.3.2 A/D 转换控制器与 DMA 控制器操作原理

1．A/D 转换控制器操作原理

TM32F103 处理器共有 18 个 A/D 转换通道，其中 16 个用于外部数据转换，2 个用于内部数/模转换，转换精度为 12bit。每一个转换通道均可配置为独立转换、连续转换、扫描方式或不连续工作方式。转换结果保存于“左对齐”或“右对齐”的 16bit 寄存器中。

A/D 控制器的引脚定义如表 5.8 所示。

表 5.8　　A/D 控制器引脚定义

引 脚 名 称	信 号 类 型	描　　述
V_{REF+}	模拟信号输入参考（正）	$V_{SSA} \leqslant V_{REF+} \leqslant V_{DDA}$
V_{DDA}	输入	$2.4V \leqslant V_{DDA} \leqslant V_{DD}$ (3.6V)
V_{REF}	输入，模拟信号参考（负）	$V_{REF-} = V_{SSA}$

V_{SSA}	输入，模拟输入地	等于 V_{SS}
ADC_IN[15:0]	模拟输入信号	16 个模拟信号输入通道
EXTSEL[2:0]	输入，数据信号	–
JEXTSEL[2:0]	输入，数据信号	–

（1）A/D 转换控制器主要寄存器描述。

- ADC 状态寄存器（ADC_SR）

地址偏移量：0x00h；初始值：0x0。

该寄存器各位说明如表 5.9 所示。

表 5.9　　ADC_SR 寄存器描述

寄 存 器 位	位　描　述
Bit[31：5]	保留位
Bit[4]	STRT：A/D 通道开始标志位，当转换开始时该位由硬件自动设置，但必须由软件清除。 0：没有通道正在进行 A/D 转换。 1：A/D 转换已经开始
Bit[3]	JSTRT：注入通路 A/D 转换标志位，当转换开始时该位由硬件自动设置，但必须由软件清除。 0：A/D 转换没有结束。 1：A/D 转换已经开始
Bit[2]	JEOC：注入通道结束转换，注入转换结束时，该位由硬件自动置位。 0：转换没有完成。 1：转换结束
Bit[1]	EOC：组转换结束标志位。组转换结束时，该位由硬件自动置位，通过读 ADC_DR 清除该位。 0：转换没有完成。 1：转换结束
Bit[0]	AWD：逻辑看门狗标志位。当转换电压超出 ADC_LRT 和 ADC_HTR 标示的范围，该位置位。 0：没有逻辑看门狗事件发生。 1：逻辑看门狗事件发生（电压超出范围）

- ADC 取样时间寄存器（ADC_SMPR1）

地址偏移量：0x0C；初始值：0x0。

寄存器位描述如表 5.10 所示。

表 5.10　　DC 取样时间寄存器描述

寄 存 器 位	位　描　述
Bit[31：27]	保留位
Bit[26：0]	SMPx[2：0] 通道 x 取样时间选择 000：1.5 周期。 001：7.5 周期。 010：13.5 周期。 011：28.5 周期。 100：41.5 周期。 101：55.5 周期。 110：71.5 周期。 111：239.5 周期

更多关于 ADC 寄存器的内容请参见 STM32F10x 用户手册。

（2）ADC 开/关控制。

通过设置 ADC_CR1 寄存器的 ADON 位给 ADC 上电。初始化过程为：第一次对 ADON 设置时，系统将 ADC 从睡眠模式唤醒；间隔 ADC “Power-up” 时间（tSTAB）后，软件再次对 ADON 置位，数据转换开始。

设置 ADON 位为 ADC“power down”模式，ADC 停止数据转换。

（3）ADC 时钟。

ADC 时钟由系统时钟控制器提供，与 PCLK2（APB2 时钟）同步。系统时钟控制器为 ADC 时钟提供可编程的精确分频，更详细内容请参考 STM32F10x 用户手册 CLK 章节。

（4）通道选择。

STM32F103 片上共 16 个混合通道，分 2 组：通用通道和注入通道。通用通道包括 16 路转换，转换序列可通过 ADC_SQRx 寄存器设置；注入通道包括 4 路转换，转换序列可通过 ADC_JSQR 寄存器设置。

2．DMA 控制器操作原理

DMA（Direct Memory Access，直接存储器存取）传输方式无需 CPU 直接控制传输，也没有中断处理方式那样保留现场和恢复现场的过程，通过硬件为 RAM 与 I/O 设备开辟一条直接传送数据的通路，使 CPU 的效率大为提高。在实现 DMA 传输时，是由 DMA 控制器直接掌管总线，因此，存在着一个总线控制权转移问题。即 DMA 传输前，CPU 要把总线控制权交给 DMA 控制器，而在结束 DMA 传输后，DMA 控制器应立即把总线控制权再交回给 CPU。

一个完整的 DMA 传输过程必须经过下面的 4 个步骤。

（1）DMA 请求。CPU 对 DMA 控制器初始化，并向 I/O 接口发出操作命令，I/O 接口提出 DMA 请求。

（2）DMA 响应。DMA 控制器对 DMA 请求判别优选级及屏蔽，向总线裁决逻辑提出总线请求。当 CPU 执行完当前总线周期即可释放总线控制权。此时，总线裁决逻辑输出总线应答，表示 DMA 已经响应，通过 DMA 控制器通知 I/O 接口开始 DMA 传输。

（3）DMA 传输。DMA 控制器获得总线控制权后，CPU 即刻挂起或只执行内部操作，由 DMA 控制器输出读写命令，直接控制 RAM 与 I/O 接口进行 DMA 传输。

（4）DMA 结束。当完成规定的成批数据传送后，DMA 控制器即释放总线控制权，并向 I/O 接口发出结束信号。当 I/O 接口收到结束信号后，一方面停止 I/O 设备的工作，另一方面向 CPU 提出中断请求，使 CPU 从不介入的状态解脱，并执行一段检查本次 DMA 传输操作正确性的代码。最后，带着本次操作结果及状态继续执行原来的程序。

STM32F103 DMA 控制器共 7 个通信通道，每个通道指定访问 1 个或多个外设。同时，DMA 控制器拥有一个硬件仲裁逻辑，管理 7 个通道间的优先级，该优先级可通过软件设置。图 5.3 所示为 STM32F103 的 DMA 控制器结构图。图中清晰显示了 DMA 控制器与 CPU、外设和内存间的关系。

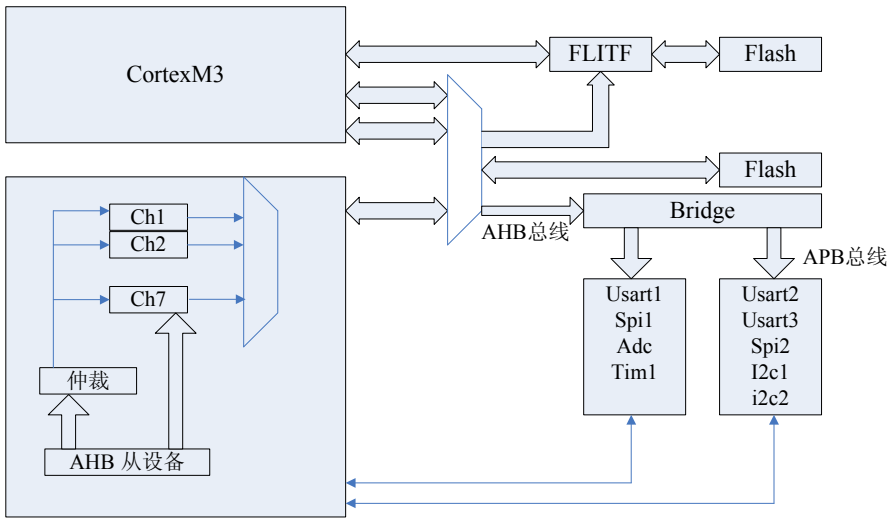

图 5.3　DMA 控制器结构图

（1）DMA 通道配置。

STM32F103 的 7 个 DMA 通道相互独立，每一个通道可以单独配置。每次最多可传输 65 535 字节，传输数据大小可通过 DMA_CCRx 寄存器的 PSIZE 和 MSIZE 位进行配置。具体配置步骤如下，其中 x 代表通道编号。

① 将外设地址存入 DMA_CPARx 寄存器，DMA 开始工作后将把该地址的数据和内存数据进行交换。

② 将要交换数据的内存地址存入 DMA_CMARx 寄存器，外设事件发生后，该地址数据将通过 DMA 和外设数据进行交互。

③ 将要传输的数据量字节数存入 DMA_CNDTRx 寄存器。每次传输结束后，该值将减去实际数据传输量。

④ 通过寄存器 DMA_CCRx 的 PL[1：0]位配置通道优先级。

⑤ 通过 DMA_CCRx 寄存器配置传输方向、循环模式、外设/内存地址增长方向、外设/内存数据传输大小和中断模式。

⑥ 设置 DMA_CCRx 寄存器 ENABLE 位激活 DMA，数据传输开始。

DMA 数据传输过程中，一旦发生错误，传输将停止，同时硬件将相应通道的 DMA_CCRx 寄存器 EN 位清零。传输结束后，DMA_IFR 寄存器的传输错误中断标志位 TEIE 被置位，并向 CPU 产生传输错误中断。表 5.11 所示为各 DMA 通道和外设的对应关系。

表 5.11　　DMA 通道和外设对应关系

外设	通道 1	通道 2	通道 3	通道 4	通道 5	通道 6	通道 7
ADC	ADC1						
SPI		SPI1_RX	SPI1_TX	SPI2_RX	SPI2_TX		
USART		USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I^2C				I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1		TIM1_CH1	TIM1_CH2	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
TIM2	TIM2_CH3	TIM2_UP			TIM2_CH1		TIM2_CH2 TIM2_CH4
TIM3		TIM3_CH3	TIM3_CH4 TIM3_UP			TIM3_CH1 TIM3_TRIG	
TIM4	TIM4_CH1			TIM4_CH2	TIM4_CH3		TIM4_UP

（2）DMA 主要寄存器描述。

- DMA 中断状态寄存器 DMA_ISR

地址偏移量 0x0；初始值：0x0。

寄存器位描述如表 5.12 所示。

表 5.12　　DMA 中断状态寄存器 DMA_ISR 描述

寄 存 器 位	位　描　述
Bit[31：28]	保留位
Bits27，23，19，15，11，7，3	TEIFx：通道 x 传输错误（TE，Transfer Error）标志位（x = 1, …, 7） 0：通道 x 没有传输错误。 1：通道 x 存在传输错误
Bits26，22，18，14，10，6，2	HTIFx：通道 x 半传输（HT，half transfer）标志位（x = 1, …, 7） 0：通道 x 没有半传输 HT。 1：通道 x 存在半传输 HT
Bits25，21，17，13，9，5，1	TCIFx：通道 x 传输完成（TC，Transfer Complete）标志位（x = 1, …, 7） 0：通道 x 没有传输完成标志。

	1：通道 x 存在传输完成标志
Bits24，20，16，12，8，4，0	GIFx：通道 x 全局中断标志位（x = 1, …, 7） 0：通道 x 没有 TE、HT、TC 传输标志。 1：通道 x 上至少有一个 TE、HT、TC 传输标志

- DMA 中断清除寄存器 DMA_IFCR

地址偏移量 0x4；初始值：0x0。

寄存器位描述如表 5.13 所示。

表 5.13　　DMA 中断清除寄存器 DMA_IFCR 描述

寄 存 器 位	位　描　述
Bit[31：28]	保留位
Bits27，23，19，15，11，7，3	CTEIFx：通道 x 传输错误（TE，Transfer Error）清除标志位（x = 11, …, 7） 0：无影响。 1：清除通道 x 存在传输错误
Bits26，22，18，14，10，6，2	CHTIFx：通道 x 半传输（HT，half transfer）清除标志位（x = 1, …, 7） 0：无影响。 1：清除通道 x 存在半传输 HT
Bits25，21，17，13，9，5，1	CTCIFx：通道 x 传输完成（TC，Transfer Complete）清除标志位（x = 1, …, 7） 0：无影响。 1：清除通道 x 存在传输完成标志
Bits24，20，16，12，8，4，0	CGIFx：通道 x 全局中断清除标志位（x = 1, …, 7） 0：无影响。 1：清除通道 x 上 TE，HT，TC 传输标志

- DMA 通道配置寄存器 DMA_CCRx（x = 1, …, 7）

地址偏移量 0x8 + 20d × x（通道编号）；初始值：0x0。

寄存器位描述如表 5.14 所示。

表 5.14　　DMA 通道配置寄存器 DMA_CCRx 描述

寄 存 器 位	位　描　述
Bit[31：15]	保留位
Bit 14	MEM2MEE：内存到内存数据交换模式设置位 0：内存到内存数据交换模式禁止。 1：使能内存到内存数据交换模式
Bits[13：12]	PL[1：0] 通道优先级设置位 00：低。 01：中。 10：高。 11：最高
Bit[11：10]	MSIZE[1：0]传输数据大小设置位 00：8 比特。 01：16 比特。 10：32 比特。 11：保留
Bit[9：8]	PSIZE[1：0] 外设数据大小设置位 00：8 比特。 01：16 比特。 10：32 比特。 11：保留
Bit [7]	MINC：内存地址增加模式设置位

	0：内存地址增加模式禁止。 1：内存地址增加模式使能
Bit [6]	PINC：外设地址增加模式设置位 0：外设地址增加模式禁止。 1：外设地址增加模式使能
Bit[5]	CIRC：循环模式设置位 0：循环模式禁止。 1：循环模式使能
Bit[4]	DIR：数据传输方向设置位 0：从外设读。 1：从内存读

续表

寄 存 器 位	位 描 述
Bit[3]	TEIE：传输错误中断使能设置位 0：传输错误中断禁止。 1：传输错误中断使能
Bit[2]	HTIE：半字节传输中断使能 0：半字节传输中断禁止。 1：半字节传输中的使能
Bit[1]	TCIE：传输完成中断使能 0：传输中断禁止。 1：传输中断使能
Bit[0]	EN：通道使能（Channel enable） 0：通道禁止。 1：通道使能

更多关于 DMA 寄存器的内容请参见 STM32F10x 用户手册。

5.3.3 A/D 转换和 DMA 编程实例操作步骤

（1）准备实验环境，连接好主机-μlinker-目标板。
（2）启动 Keil μVersion 3，打开所需工程 ADC.Uv2。
（3）选择 RAM 连接文件 adc.sct。
（4）选择 RAM 调试文件 RAM.ini。
（5）使用“Debug——Debug session”加载 image 文件。
（6）选择“run”命令运行程序。
（7）调节 STM32F10x-EVAL 评估板点位计 RV1，使模拟信号输入发生变化。
（8）使用 break 命令，使程序停止运行，观察 ADC1 DR 寄存器。
（9）使用 watch 窗口观察 ADC_ConvertedValue 变量值的变化。
（10）退出系统。

5.3.4 A/D 转换和 DMA 编程实例参考程序及说明

（1）ADC 初始化函数 ADC_Init()。
该函数根据输入参数初始化相应 ADC 控制器状态。参数 ADCx 指定使用哪个 ADC 控制器进行数据转换，数据结构 ADC_InitStruct 保持所有对 ADC 寄存器设置的初值。函数代码如下：

```
void ADC_Init(ADC_TypeDef* ADCx, ADC_InitTypeDef* ADC_InitStruct)
{
  u32 tmpreg1 = 0;
  u8 tmpreg2 = 0;

  /* 检测输入参数 */
  assert(IS_ADC_MODE(ADC_InitStruct->ADC_Mode));
  assert(IS_FUNCTIONAL_STATE(ADC_InitStruct->ADC_ScanConvMode));
  assert(IS_FUNCTIONAL_STATE(ADC_InitStruct->ADC_ContinuousConvMode));
  assert(IS_ADC_EXT_TRIG(ADC_InitStruct->ADC_ExternalTrigConv));
  assert(IS_ADC_DATA_ALIGN(ADC_InitStruct->ADC_DataAlign));
  assert(IS_ADC_REGULAR_LENGTH(ADC_InitStruct->ADC_NbrOfChannel));

  /*---------------------------- 配置 ADCx CR1 -----------------*/
  /* Get the ADCx CR1 value */
  tmpreg1 = ADCx->CR1;
  /* Clear DUALMODE and SCAN bits */
  tmpreg1 &= CR1_CLEAR_Mask;
  /* Configure ADCx: Dual mode and scan conversion mode */
  /* Set DUALMODE bits according to ADC_Mode value */
  /* Set SCAN bit according to ADC_ScanConvMode value */
  tmpreg1 |= (u32)(ADC_InitStruct->ADC_Mode | ((u32)ADC_InitStruct->ADC_Scan ConvMode << 8));
  /* Write to ADCx CR1 */
  ADCx->CR1 = tmpreg1;

  /*---------------------------- 配置 ADCx CR2 -----------------*/
  /* Get the ADCx CR2 value */
  tmpreg1 = ADCx->CR2;
  /* Clear CONT, ALIGN and EXTTRIG bits */
  tmpreg1 &= CR2_CLEAR_Mask;
  /* Configure ADCx: external trigger event and continuous conversion mode */
  /* Set ALIGN bit according to ADC_DataAlign value */
  /* Set EXTTRIG bits according to ADC_ExternalTrigConv value */
  /* Set CONT bit according to ADC_ContinuousConvMode value */
  tmpreg1 |= (u32)(ADC_InitStruct->ADC_DataAlign | ADC_InitStruct->ADC_External TrigConv |
            ((u32)ADC_InitStruct->ADC_ContinuousConvMode << 1));
  /* Write to ADCx CR2 */
  ADCx->CR2 = tmpreg1;

  /*---------------------------- 配置 ADCx SQR1 -----------------*/
  /* Get the ADCx SQR1 value */
  tmpreg1 = ADCx->SQR1;
  /* Clear L bits */
  tmpreg1 &= SQR1_CLEAR_Mask;
  /* Configure ADCx: regular channel sequence length */
  /* Set L bits according to ADC_NbrOfChannel value */
  tmpreg2 |= (ADC_InitStruct->ADC_NbrOfChannel - 1);
  tmpreg1 |= ((u32)tmpreg2 << 20);
  /* Write to ADCx SQR1 */
  ADCx->SQR1 = tmpreg1;
}
```

（2）ADC 中断设置函数 ADC_ITConfig()。

该函数使能/禁止指定的 ADC 中断，输入参数 ADCx 可选值为 1 或 2，指定开启哪个 ADC 通道；输入参数 ADC_IT 指定要使能/禁止的中断源，可取值为 ADC_IT_EOC、ADC_IT_AWD 和 ADC_IT_JEOC，分别表示转换结束中断、模拟看门狗中断和输入转换结束中断。其函数源码为：

```
void ADC_ITConfig(ADC_TypeDef* ADCx, u16 ADC_IT, FunctionalState NewState)
{
  u8 itmask = 0;

  /*检测输入参数*/
  assert(IS_FUNCTIONAL_STATE(NewState));
  assert(IS_ADC_IT(ADC_IT));

  /* 取得 ADC IT 索引 */
  itmask = (u8)ADC_IT;

  if (NewState != DISABLE)
  {
    /*使能 ADC 中断 */
    ADCx->CR1 |= itmask;
  }
  else
  {
    /* 禁止选择的 ADC 中断 */
    ADCx->CR1 &= (~(u32)itmask);
  }
}
```

（3）A/D 转换 DMA 传输使能函数 DC_DMACmd()。

该函数使能/禁止指定 ADC 通道的 DMA 传输。该函数直接对 ADC 控制寄存器进行操作，函数原型如下：

```
void ADC_DMACmd(ADC_TypeDef* ADCx, FunctionalState NewState)
{
  /*检测输入参数*/
  assert(IS_FUNCTIONAL_STATE(NewState));

  if (NewState != DISABLE)
  {
    /* 使能所选 ADC 的 DMA 请求 */
    ADCx->CR2 |= CR2_DMA_Set;
  }
  else
  {
    /* 禁止所选 ADC 的 DMA 请求*/
    ADCx->CR2 &= CR2_DMA_Reset;
  }
}
```

（4）DMA 初始化函数 DMA_Init()。

该函数根据指定的参数初始化 DMA 通道，输入参数 DMA_Channelx 指定将要初始化的 DMA 通道编号，DMA_InitStruct 为 DMA 初始化结构。其函数原型如下：

```
void DMA_Init(DMA_Channel_TypeDef* DMA_Channelx, DMA_InitTypeDef* DMA_InitStruct)
{
  u32 tmpreg = 0;

  /*检测输入参数*/
```

```
  assert(IS_DMA_DIR(DMA_InitStruct->DMA_DIR));
  assert(IS_DMA_BUFFER_SIZE(DMA_InitStruct->DMA_BufferSize));
  assert(IS_DMA_PERIPHERAL_INC_STATE(DMA_InitStruct->DMA_PeripheralInc));
  assert(IS_DMA_MEMORY_INC_STATE(DMA_InitStruct->DMA_MemoryInc));
  assert(IS_DMA_PERIPHERAL_DATA_SIZE(DMA_InitStruct->DMA_PeripheralDataSize));
  assert(IS_DMA_MEMORY_DATA_SIZE(DMA_InitStruct->DMA_MemoryDataSize));
  assert(IS_DMA_MODE(DMA_InitStruct->DMA_Mode));
  assert(IS_DMA_PRIORITY(DMA_InitStruct->DMA_Priority));
  assert(IS_DMA_M2M_STATE(DMA_InitStruct->DMA_M2M));

/*-------------------------- 配置 DMA 通路 CCR 寄存器-----------------*/
  /* Get the DMA_Channelx CCR value */
  tmpreg = DMA_Channelx->CCR;
  /* Clear MEM2MEM, PL, MSIZE, PSIZE, MINC, PINC, CIRCULAR and DIR bits */
  tmpreg &= CCR_CLEAR_Mask;
  /* Configure DMA Channelx: data transfer, data size, priority level and mode */
  /* Set DIR bit according to DMA_DIR value */
  /* Set CIRCULAR bit according to DMA_Mode value */
  /* Set PINC bit according to DMA_PeripheralInc value */
  /* Set MINC bit according to DMA_MemoryInc value */
  /* Set PSIZE bits according to DMA_PeripheralDataSize value */
  /* Set MSIZE bits according to DMA_MemoryDataSize value */
  /* Set PL bits according to DMA_Priority value */
  /* Set the MEM2MEM bit according to DMA_M2M value */
  tmpreg |= DMA_InitStruct->DMA_DIR | DMA_InitStruct->DMA_Mode |
            DMA_InitStruct->DMA_PeripheralInc | DMA_InitStruct->DMA_MemoryInc |
            DMA_InitStruct->DMA_PeripheralDataSize | DMA_InitStruct->DMA_Memory DataSize |
            DMA_InitStruct->DMA_Priority | DMA_InitStruct->DMA_M2M;
  /* 写入 DMA CCR 寄存器 */
  DMA_Channelx->CCR = tmpreg;

/*-------------------------- 配置 DMA 通路 CNBTR 寄存器 ---------------*/
  /* Write to DMA Channelx CNBTR */
  DMA_Channelx->CNDTR = DMA_InitStruct->DMA_BufferSize;

/*-------------------------- 配置 DMA Channelx CPAR 寄存器 ----------------*/
  /* Write to DMA Channelx CPAR */
  DMA_Channelx->CPAR = DMA_InitStruct->DMA_PeripheralBaseAddr;

/*-------------------------- 配置 DMA Channelx CMAR 寄存器 ----------------*/
  /* Write to DMA Channelx CMAR */
  DMA_Channelx->CMAR = DMA_InitStruct->DMA_MemoryBaseAddr;
}
```

（5）DMA 传输开始/结束函数。

该函数开始/结束指定 DMA 通道数据传输，该函数直接对 DMA 寄存器进行操作，其函数原型如下：

```
void DMA_Cmd(DMA_Channel_TypeDef* DMA_Channelx, FunctionalState NewState)
{
  /*检测输入参数*/
  assert(IS_FUNCTIONAL_STATE(NewState));

  if (NewState != DISABLE)
```

```
    {
      /* 使能所选 DMA 通路 */
      DMA_Channelx->CCR |= CCR_ENABLE_Set;
    }
    else
    {
      /* 禁止所选 DMA 通路 */
      DMA_Channelx->CCR &= CCR_ENABLE_Reset;
    }
  }
```

（6）实例主函数 main()。

程序使用 ADC 的 14 号通道，连续读取外部模拟信号并将其转化为数据信号。每次数/模转换结束后，激活 DMA 中断传输数据，将 ADC1 DR 寄存器的值传入自定义变量 ADC_ConvertedValue，转换过程中将 ADC1 的时钟频率设为 14MHz。主函数源程序如下：

```
int main(void)
{
#ifdef DEBUG
  debug();
#endif

  /* 配置系统时钟----------------------------------------------*/
  RCC_Configuration();

  /* 配置 NVIC -----------------------------------------------------*/
  NVIC_Configuration();

  /* 配置 GPIO -----------------------------------------------------*/
  GPIO_Configuration();

   /* 配置 USART1 */
  USART_Configuration1();

  printf("\r\n usart1 print AD_value ------------------------- \r\n");

  /* 配置 DMA channel1 ---------------------------------------------*/
  DMA_DeInit(DMA_Channel1);
  DMA_InitStructure.DMA_PeripheralBaseAddr = ADC1_DR_Address;
  DMA_InitStructure.DMA_MemoryBaseAddr = (u32)&ADC_ConvertedValue;
  DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralSRC;
  DMA_InitStructure.DMA_BufferSize = 1;
  DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
  DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Disable;
  DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
  DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
  DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
  DMA_InitStructure.DMA_Priority = DMA_Priority_High;
  DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;
  DMA_Init(DMA_Channel1, &DMA_InitStructure);

  /*使能 DMA channel1 */
  DMA_Cmd(DMA_Channel1, ENABLE);
```

```
    /* ADC1 configuration ------------------------------------------------------*/
    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
    ADC_InitStructure.ADC_ScanConvMode = ENABLE;
    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
    ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
    ADC_InitStructure.ADC_NbrOfChannel = 1;
    ADC_Init(ADC1, &ADC_InitStructure);

    /* 配置 ADC1 channel14 */
    ADC_RegularChannelConfig(ADC1, ADC_Channel_14, 1, ADC_SampleTime_55Cycles5);

    /* 使能 ADC1 DMA */
    ADC_DMACmd(ADC1, ENABLE);

    /* 使能 ADC1 */
    ADC_Cmd(ADC1, ENABLE);

    /* 使能 ADC1 reset 寄存器 */
    ADC_ResetCalibration(ADC1);
    /* Check the end of ADC1 reset calibration register */
    while(ADC_GetResetCalibrationStatus(ADC1));

    /*启动 ADC1 */
    ADC_StartCalibration(ADC1);
    /* 检测 ADC1 calibration 是否结束*/
    while(ADC_GetCalibrationStatus(ADC1));

    /* 启动 ADC1 反转 */
    ADC_SoftwareStartConvCmd(ADC1, ENABLE);

   while(1)
    {
    AD_value=ADC_GetConversionValue(ADC1);
     if (ticks++ >= 9 999) {              /* Set Clock1s to 1 every 1 second   */
      ticks   = 0;
      Clock1s = 1;
    }

     /* 每 1s 将 AD 采样值从串口输出           */
      if (Clock1s) {
        Clock1s = 0;
        printf("\033[42;31m AD value = 0x%04X \033[0m\r\n", AD_value);
      }

    }
  }
```

5.4 实时钟 RTC 编程实例

5.4.1 实例内容与目标

实例演示了如何使用 STM32F103 的实时钟模块。实例通过对预分频寄存器和外设中断寄存器设置正确初始值，使系统每一秒产生一次实时钟中断，从而产生实时效果。另外，STM32F103 的实时钟模块位于系统的 BKP（Back-up）域，可以由电池供电，达到掉电不丢失效果，从而实现时钟功能。

通过该实例重点掌握以下内容：

- 了解实时时钟的硬件控制原理及设计方法；
- 掌握 STM32F103 处理器的 RTC 模块程序设计方法。

5.4.2 STM32F103 实时钟操作原理

实时时钟（RTC）器件是一种能提供日历/时钟、数据存储功能的专用集成电路，常用作各种计算机系统的时钟信号源和参数设置存储电路。RTC 具有计时准确、耗电低和体积小等特点，特别适用于在各种嵌入式系统中记录事件发生的时间和相关信息，尤其是在通信工程、电力自动化、工业控制等自动化程度较高领域的无人值守环境。随着集成电路技术的不断发展，RTC 器件的新品也不断推出。这些新品不但具有准确的 RTC，还有大容量的存储器、温度传感器和 A/D 数据采集通道等，已经成为集成 RTC、数据采集和存储于一体的综合功能器件，特别适用于以微控制器为核心的嵌入式系统。

STM32F103 处理器的 RTC 模块包含 4 个主要寄存器，分别为 RTC 控制高寄存器 RTC_PRL、RTC 控制低寄存器 RTC_ALR、分频加载寄存器 RTC_CNT 和分频除数寄存器 RTC_DIV。这些寄存器的值不会随系统的重启而重置，只有系统备份域重启时，这些寄存器的内容才会清零。

只有系统进入配置模式时，才可以修改 RTC_PRL、RTC_ALR、RTC_CNT 寄存器的值，且必须遵循以下标准配置过程：

- “Poll RTOFF”，直到 RTOFF 的值为“1”；
- 设置 CNF 位，使系统进入配置模式；
- 写一个或多个 RTC 寄存器；
- 清除 CNF 位，使系统退出配置模式；
- “Poll RTOFF”直到 RTOFF 的值为“1”；
- 操作结束。

整个过程至少要 3 个 RTCCLK 周期才能完成。

表 5.15 所示为 RTC 高控制寄存器的位定义。

表 5.15　　RTC 高控制寄存器位定义

寄 存 器 位	位　描　述
Bit[15：3]	保留位
Bit[2]	OWIE：OverfloW 中断使能位 0：OverfloW 中断掩码。 1：OverfloW 中断使能
Bit[1]	ALRIE：报警中断使能位 0：报警中断掩码。 1：报警中断使能
Bit[0]	SECIE：“秒”中断使能位 0：“秒”中断掩码。 1：“秒”中断使能

表 5.16 所示为 RTC 低控制寄存器的位定义。

表 5.16　　RTC 低控制寄存器位定义

寄 存 器 位	位　描　述
Bit[15：6]	保留位

Bit[5]	RTOFF：RTC 操作结束标志位 0：RTC 寄存器最后一次写操作未完成。 1：RTC 寄存器最后一次写操作已完成
Bit[4]	CNF：寄存器配置标志位 0：系统退出配置模式。 1：系统进入配置模式
Bit[3]	RSF：登记同步信号标志位 0：未登记同步。 1：登记同步
Bit[2]	OWF：OverfloW 标志位 0：未检测到 OverfloW。 1：OverfloW 已发生
Bit[1]	ALRF：警报标志位 0：未方式警报。 1：警报发生
Bit[0]	SECF：“秒”标志位 0：秒标志条件未满足。 1：秒标志条件已满足

更多关于 RTC 寄存器的内容请参见 STM32F10x 用户手册。

5.4.3 实时钟 RTC 编程实例操作步骤

（1）准备实验环境，连接好主机-μlinker-目标板。
（2）启动 Keil μVersion 3，打开所需工程 RTC.Uv2。
（3）选择 RAM 连接文件 rtc.sct。
（4）选择 RAM 调试文件 RAM.ini。
（5）使用“Debug-Debug session”加载 image 文件。
（6）选择“run”命令运行程序。
（7）程序检测 Register1 的值，若该值未设，串口终端输出提示信息，并接收用户输入的时间信息。
（8）从串口终端中输出已保存的系统时间信息。
（9）按压评估板上的 Reset 开关，触发外部 Reset 信号，观察系统，保存的时间信息未丢失。
（10）将评估板的 Vbat 引脚与电池相连，按下“Power On”按键，观察系统，系统时间信息未丢失。
在整个实验过程中，与 PC.06 引脚相连的 LED 灯每隔 1s 闪烁一次。

5.4.4 实时钟 RTC 编程实例参考程序及说明

（1）实时钟中断配置函数 RTC_ITConfig()。
该函数使能/禁止 RTC 中断。输入参数 RTC_IT 指定要使能/禁止的 RTC 中断，该参数值为 RTC_IT_OW、RTC_IT_ALR 和 RTC_IT_SEC，分别代表 Overflow 中断、警报中断和秒中断。该函数原型如下：

```
void RTC_ITConfig(u16 RTC_IT, FunctionalState NewState)
{
  /* 检测输入参数 */
  assert(IS_RTC_IT(RTC_IT));
  assert(IS_FUNCTIONAL_STATE(NewState));
```

```
  if (NewState != DISABLE)
  {
    RTC->CRH |= RTC_IT;
  }
  else
  {
    RTC->CRH &= (u16)~RTC_IT;
  }
}
```

（2）设置计数值函数 RTC_SetCounter()。

该函数根据 RTC 时钟源频率设置实时时钟控制器的计数值，函数原型如下：

```
void RTC_SetCounter(u32 CounterValue)
{
  RTC_EnterConfigMode();

  /* 设置 RTC COUNTER MSB 值 */
  RTC->CNTH = (CounterValue & RTC_MSB_Mask) >> 16;
  /* 设置 RTC COUNTER LSB 值 */
  RTC->CNTL = (CounterValue & RTC_LSB_Mask);

  RTC_ExitConfigMode();
}
```

（3）设置预分频函数 RTC_SetPrescaler()。

该函数直接操作 RTC 寄存器设置 RTC 分频值，函数原型如下：

```
void RTC_SetPrescaler(u32 PrescalerValue)
{
  /* 检测输入参数 */
  assert(IS_RTC_PRESCALER(PrescalerValue));

  RTC_EnterConfigMode();

  /* 设置 RTC PRESCALER MSB 值 */
  RTC->PRLH = (PrescalerValue & PRLH_MSB_Mask) >> 0x10;
  /* 设置 RTC PRESCALER LSB 值 */
  RTC->PRLL = (PrescalerValue & RTC_LSB_Mask);

  RTC_ExitConfigMode();
}
```

（4）实时时钟主函数 main()。

主函数 main()负责初始化系统，使用串口终端与用户交互，提示用户输入系统时钟值，并将输入值保存，其原型如下：

```
int main(void)
{
#ifdef DEBUG
  debug();
#endif

  /* 配置系统时钟 */
  RCC_Configuration();
```

```
/* 配置 NVIC */
NVIC_Configuration();

/* 配置 GPIO */
GPIO_Configuration();

/* 配置 USART1 */
USART_Configuration();

if(BKP_ReadBackupRegister(BKP_DR1) != 0xA5A5)
{
  /* Backup data 寄存器的值还不正确或还未编程(程序第 1 次执行时，该条件满足
     ) */

  printf("\r\n\n RTC not yet configured....");

  /* 配置 RTC */
  RTC_Configuration();

   printf("\r\n RTC configured....");

/* 用户通过超级终端输入 Time 值 */
  Time_Adjust();

  BKP_WriteBackupRegister(BKP_DR1, 0xA5A5);
}
else
{
  /* 检测 Power On Reset 标志是否置位 */
  if(RCC_GetFlagStatus(RCC_FLAG_PORRST) != RESET)
  {
    printf("\r\n\n Power On Reset occurred....");
  }
  /*检测 Pin Reset 标志是否置位 */
  else if(RCC_GetFlagStatus(RCC_FLAG_PINRST) != RESET)
  {
    printf("\r\n\n External Reset occurred....");
  }

  printf("\r\n No need to configure RTC....");
  /* 等待 RTC 寄存器同步 */
  RTC_WaitForSynchro();

  /*使能 RTC Second */
  RTC_ITConfig(RTC_IT_SEC, ENABLE);
  /* 等待 RTC 写操作完成 */
  RTC_WaitForLastTask();
}

/* 清除 Reset 标志 */
RCC_ClearFlag();
```

```
    /* 无限循环显示时间 */
    Time_Show();
  }
```

5.5 串行外设接口 SPI 编程实例

5.5.1 实例内容与目标

该实例演示如何编写 SPI 接口的 Flash 读写程序。
通过该实例重点掌握以下内容：

- 了解 SPI 接口控制原理；
- 掌握 SPI 接口的基本使用方法；
- 熟悉如何使用 SPI 接口读写 Flash 内容。

5.5.2 SPI 接口操作原理

SPI（Serial Peripheral Interface，串行外设接口）总线系统是一种同步串行外设接口，它可以使 MCU 与各种外围设备以串行方式进行通信以交换信息。外围设置 FLASH、RAM、网络控制器、LCD 显示驱动器、A/D 转换器和 MCU 等。SPI 总线系统可直接与各个厂家生产的多种标准外围器件直接接口，该接口一般使用 4 条线：串行时钟线（SCK）、主机输入/从机输出数据线 MISO、主机输出/从机输入数据线 MOSI 和低电平有效的从机选择线 SS（有的 SPI 接口芯片带有中断信号线 INT 或 INT，有的 SPI 接口芯片没有主机输出/从机输入数据线 MOSI）。
SPI 接口是 Motorola 首先在其 MC68HCXX 系列处理器上定义的。该接口在 CPU 和外围低速器件之间进行同步串行数据传输，在主器件的移位脉冲下，数据按位传输，高位在前，低位在后，为全双工通信，数据传输速度总体来说比 I^2C 总线要快，速度可达到几 Mbit/s。
SPI 接口是以主从方式工作的，这种模式通常有一个主器件和一个或多个从器件，其接口包括以下 4 种信号。

- MOSI（Master In/Slave Out data）：主器件数据输出，从器件数据输入。
- MISO（Master Out/Slave In data）：主器件数据输入，从器件数据输出。
- SCLK（Serial Clock）：时钟信号，由主器件产生。
- NSS（Slave Select）：从器件使能信号，由主器件控制。

在点对点的通信中，SPI 接口不需要进行寻址操作，且为全双工通信，简单高效。在多个从器件的系统中，每个从器件需要独立的使能信号，硬件上比 I^2C 系统要稍微复杂一些。
SPI 接口在内部硬件实际上是两个简单的移位寄存器，传输的数据为 8 位，在主器件产生的从器件使能信号和移位脉冲下，按位传输，高位在前，低位在后。
（1）SPI 从模式。
当设备 SPI 处于从模式下，串口时钟通过 SCK 引脚接收主设备时钟。该值存在于 SPI_CR1 寄存器的 BR[2：0]位，该值并不影响传输速率。
从模式下，传输过程如下：

- 设置 DFF 域定义数据帧格式为 8bit 或 16bit；
- 设置 CPOL 和 CPHA 数据位定义传输和串口时钟之间的关系。传输过程中，主设备和从设备 CPOL 和 CPHA 值必须相同；
- 设置数据帧传输方式，高位在前（MSB-first）或低位在前（LSB-first）。
- 设置 SPI_CR1 寄存器的 SSM 位，同时将 SSI 位清零；

- 设置 SPI_CR1 寄存器的 SPE 位，同时将 MSTR 位清零。

（2）SPI 主模式。

当设备 SPI 处于主模式下，串口时钟由 SCK 引脚提供。

主模式下，传输过程如下：

- 设置 SPI_CR1 寄存器的 BR[2：0]位，定义传输波特率；
- 设置 CPOL 和 CPHA 位定义数据传输和串口时钟直接的关系；
- 设置 DFF 域定义数据帧格式为 8bit 或 16bit；
- 设置 SPI_CR1 寄存器的 LSBFIRST 位定义数据帧传输方式：高位在前（MSB-first）或低位在前（LSB-first）；
- 设置 MSTR 和 SPE 位。

（3）SPI 中断。

表 5.17 所示为 STM32F103 处理器支持的 SPI 中断类型和相应的中断标志。

表 5.17　　STM32F103 处理器 SPI 中断

中 断 类 型	事 件 标 志	控　制　位
传输 buffer 空标志	TXE	TXEIE
接收 buffer 非空标志	RXNE	RXNEIE
主模式错误事件	MODF	ERRIE
Overrun 错误	OVR	
CRC 校验错误标志	CRCERR	

（4）SPI 寄存器描述。

- SPI 控制寄存器 1 SPI_CR1

地址偏移量：0x0；初始值：0x0。

表 5.18 所示为控制寄存器 SPI_CR1 的位定义。

表 5.18　　控制寄存器 SPI_CR1 位定义

寄 存 器　位	位　描　述
Bit[15]	BIDMODE：双向（Bidirectional）数据模式使能设置位 0：2 线非双向模式选择。 1：1 线双向模式选项
Bit[14]	BIDIOE：双向模式输出使能设置位 0：输出禁止（仅接收模式）。 1：输出使能（仅传输模式）
Bit[13]	CRCEN：硬件 CRC 校验使能设置位 0：禁止硬件 CRC。 1：使能硬件 CRC
Bit[12]	CRCNEXT：传输下一个 CRC 设置位 0：使用 Tx buffer 传输下一个数据。 1：使用 Tx CRC 寄存器传输下一个数据
Bit[11]	DFF：数据帧格式设置位 0：数据以 8-bit 为最小传输单元。 1：数据以 16-bit 为最小传输单元
Bit[10]	RXONLY：接收模式设置位 0：全双工方式进行接收。 1：禁止输出（只接收数据）
Bit[9]	SSM：“Software slave”管理设置位 0：“Software slave”管理禁止。 1：“Software slave”管理使能

Bit[8]	SSI：“Internal slave”选择设置位 该位只在 SSM 标志设置时才起作用
Bit[7]	LSBFIRST：帧格式设置位 0：高有效位先传输。 1：低有效位先传输
Bit[6]	SPE：SPI 使能位 0：外设禁止。 1：外设使能

续表

寄 存 器 位	位 描 述
Bit[5]	BR[2：0] 波特率设置位 000：fCPU/2 001：fCPU/4 010：fCPU/8 011：fCPU/16 100：fCPU/32 101：fCPU/64 110：fCPU/128 111：fCPU/256
Bit[2]	MSTR：主模式选择设置位 0：从模式配置。 1：主模式配置
Bit[1]	CPOL：时钟设置位 0：空闲时 SCK 为 0。 1：空闲时 SCK 为 1
Bit[0]	CPHA：时钟相位选择 0：第一个时钟传输。 1：第二个时钟传输

- SPI 控制寄存器 2 SPI_CR2

地址偏移量：0x4；初始值：0x0。

表 5.19 所示为控制寄存器 SPI_CR2 的位定义。

表 5.19　　控制寄存器 SPI_CR2 位定义

寄 存 器 位	位 描 述
Bit[15：8]	保留位
Bit[7]	TXEIE：Tx buffer 空中断使能 0：TXE 中断掩码。 1：TXE 中断使能
Bit[6]	RXNEIE：RX buffer 非空中断（RX buffer not empty enable）使能设置位 0：RXNE 中断掩码。 1：RXNE 中断使能
Bit[5]	ERRIE：错误中断（Error interrupt）使能 0：错误中断掩码。 1：错误中断使能

续表

寄 存 器 位	位 描 述
Bits[4：3]	保留位

Bit[2]	SSOE：SS 输出使能 0：SS 输出禁止。 1：SS 输出使能
Bit[1]	TXDMAEN：Tx Buffer DMA 使能设置位 0：Tx buffer DMA 禁止。 1：Tx buffer DMA 使能
Bit[0]	TXDMAEN：Rx Buffer DMA 使能设置位 0：Rx buffer DMA 禁止。 1：Rx buffer DMA 使能

- SPI 状态寄存器 SPI_SR

地址偏移量：0x8；初始值：0x10。

表 5.20 所示为状态寄存器 SPI_SR 的位定义。

表 5.20　　态寄存器 SPI_SR 位定义

寄 存 器 位	位　描　述
Bit[15：8]	保留位
Bit[7]	BSY：SPI 忙标志位 0：SPI 非忙状态。 1：SPI 忙状态
Bit[6]	OVR：overrun 标志位 0：无 overrun 发生。 1：overrun 发生
Bit[5]	MODF：模式错误标志位 0：无模式错误发生。 1：模式错误发生
Bit[4]	CRCERR：CRC 错误标志位 0：CRC 接收值与 SPI_RXCRCR 匹配。 1：CRC 接收值与 SPI_RXCRCR 不匹配
Bit[3：2]	保留位
Bit[1]	TXE：传输 buffer 空标志位 0：Tx buffer 不空。 1：Tx buffer 空
Bit[0]	RXNE：接收 buffer 非空标志位 0：Rx buffer 空。 1：Rx buffer 不空

5.5.3　SPI 接口实例操作步骤

该实例首先读取 SPI Flash 的 ID 号和程序预存 ID 比较，如果二者相同则设置 PC.06 引脚（点亮开发板 06 号 LED 灯），否则设置 PC.07 引脚（点亮开发板 07 号 LED 灯）；接下来，程序将预存在变量 Tx_buffer 的数据通过 SPI 接口写入 Flash，再读出与原数据比较，将比较结果存入变量 TransferStatus1。

然后测试执行擦除操作，将所有擦除标志位与“0xFF”比较，并将比较结果存入变量 TransferStatus2。

在两次测试过程中，SPI1 接口始终配置为主模式下的 8-bit 数据格式，系统时钟频率为 72MHz，传输波特率为 18Mbit/s。

实验步骤如下所述。

（1）准备实验环境，连接好主机-μlinker-目标板。

（2）启动 Keil μVersion 3，打开所需工程 SPI.Uv2。

（3）选择 RAM 连接文件 spi.sct。

（4）选择 RAM 调试文件 RAM.ini。

（5）使用“Debug——Debug session”加载 image 文件。
（6）选择“run”命令运行程序。
（7）从 Watch 窗口观测变量 TransferStatus1 值的变化。
（8）观察开发板 LED 灯的变化，判断程序中数据的比较结果。
（9）结束调试。

5.5.4 SPI 接口实例参考程序及说明

（1）SPI 初始化函数 SPI_Init()。
该函数根据已定义好的 SPI 初始化结构初始化 SPI 接口，函数原型如下：

```
void SPI_Init(SPI_TypeDef* SPIx, SPI_InitTypeDef* SPI_InitStruct)
{
  u16 tmpreg = 0;

  /* 检测输入参数 */
  assert(IS_SPI_DIRECTION_MODE(SPI_InitStruct->SPI_Direction));
  assert(IS_SPI_MODE(SPI_InitStruct->SPI_Mode));
  assert(IS_SPI_DATASIZE(SPI_InitStruct->SPI_DataSize));
  assert(IS_SPI_CPOL(SPI_InitStruct->SPI_CPOL));
  assert(IS_SPI_CPHA(SPI_InitStruct->SPI_CPHA));
  assert(IS_SPI_NSS(SPI_InitStruct->SPI_NSS));
  assert(IS_SPI_BAUDRATE_PRESCALER(SPI_InitStruct->SPI_BaudRatePrescaler));
  assert(IS_SPI_FIRST_BIT(SPI_InitStruct->SPI_FirstBit));
  assert(IS_SPI_CRC_POLYNOMIAL(SPI_InitStruct->SPI_CRCPolynomial));

/*---------------------------配置 SPIx CR1 ------------------------*/
  /* 读取 SPIx CR1 值 */
  tmpreg = SPIx->CR1;
  /* 清除 BIDIMode, BIDIOE, RxONLY, SSM, SSI, LSBFirst, BR, MSTR, CPOL 和 CPHA 位 */
  tmpreg &= CR1_CLEAR_Mask;
  /* 配置 SPIx: 方向, NSS 管理, 第一个传输位, 波特兰分频值
     主/从模式, CPOL 和 CPHA */
  /* 根据 SPI 设置的方向值设置 BIDImode, BIDIOE 和 RxONLY 位 */
  /* 根据 SPI_Mode 和 SPI_NSS 值，设置 SSM, SSI 和 MSTR 位 */
  /* 根据 FirstBit 位值，设置 LSBFirst 位 */
  /* 根据 SPI_BaudRate 分频值设置 BR 位 */
  /* 根据 SPI_CPOL 值设置 CPOL 位 */
  /* 根据 SPI_CPHA 值设置 CPHA 位 */
  tmpreg |= (u16)((u32)SPI_InitStruct->SPI_Direction | SPI_InitStruct->SPI_Mode |
                  SPI_InitStruct->SPI_DataSize | SPI_InitStruct->SPI_CPOL |
                  SPI_InitStruct->SPI_CPHA | SPI_InitStruct->SPI_NSS |
                  SPI_InitStruct->SPI_BaudRatePrescaler | SPI_InitStruct-> SPI_FirstBit);
  /* 将设置好的值写入 SPIx CR1 寄存器 */
  SPIx->CR1 = tmpreg;

/*---------------------------配置 SPIx CRCPOLY --------------------*/
  /* Write to SPIx CRCPOLY */
  SPIx->CRCPR = SPI_InitStruct->SPI_CRCPolynomial;
}
```

（2）使能/禁止指定的 SPI 外设 SPI_Cmd()。

该函数使能/禁止 PI_TypeDef 结构指定的 SPI 端口，函数原型如下：

```
void SPI_Cmd(SPI_TypeDef* SPIx, FunctionalState NewState)
{
  /* 检测输入参数 */
  assert(IS_FUNCTIONAL_STATE(NewState));

  if (NewState != DISABLE)
  {
    /* 使能所选 SPI 外设 */
    SPIx->CR1 |= CR1_SPE_Set;
  }
  else
  {
    /* 禁止所选 SPI 外设*/
    SPIx->CR1 &= CR1_SPE_Reset;
  }
}
```

（3）SPI 中断使能/禁止函数 SPI_ITConfig()。

SPI_ITConfig()使能禁止指定的 SPI 端口中断，函数输入参数 SPIx 指定要配置的 SPI 端口，取值为 1 或 2。参数 SPI_IT 指定要操作的中断的源，取值为 SPI_IT_TXE、SPI_IT_RXNE 和 SPI_IT_ERR，分别表示 Tx buffer 空中断、Rx buffer 非空中断和错误中断（Error interrupt）。函数原型如下：

```
void SPI_ITConfig(SPI_TypeDef* SPIx, u8 SPI_IT, FunctionalState NewState)
{
  u16 itpos = 0, itmask = 0 ;

  /* 检测输入 */
  assert(IS_FUNCTIONAL_STATE(NewState));
  assert(IS_SPI_CONFIG_IT(SPI_IT));

  /* 得到 SPI IT 索引*/
  itpos = SPI_IT >> 4;
  /* 设置 IT 掩码*/
  itmask = (u16)((u16)1 << itpos);

  if (NewState != DISABLE)
  {
    /* 使能所选 SPI 中断 */
    SPIx->CR2 |= itmask;
  }
  else
  {
    /*禁止使能 SPI 中断 */
    SPIx->CR2 &= (u16)~itmask;
  }
}
```

（4）SPI DMA 设置函数 SPI_DMACmd()。

SPI 接口可以使用 DMA 进行数据传输，其配置函数如下：

```
void SPI_DMACmd(SPI_TypeDef* SPIx, u16 SPI_DMAReq, FunctionalState NewState)
{
  /* 检测输入参数 */
```

```
  assert(IS_FUNCTIONAL_STATE(NewState));
  assert(IS_SPI_DMA_REQ(SPI_DMAReq));

  if (NewState != DISABLE)
  {
    /* 使能所选 SPI DMA 请求 */
    SPIx->CR2 |= SPI_DMAReq;
  }
  else
  {
    /* 禁止所选 SPI DMA 请求*/
    SPIx->CR2 &= (u16)~SPI_DMAReq;
  }
}
```

（5）实例主函数 main()。

实例主函数通过 SPI 接口对 Flash 进行擦除、读和写操作，并在每次操作结束后进行验证，将验证结果存入预先定义的状态变量中。实例首先读取 SPI Flash 的 ID 号和程序预存 ID 进行比较，接下来，程序将预存在变量 Tx_buffer 的数据通过 SPI 接口写入 Flash，再读出与原数据进行比较，将比较结果存入变量 TransferStatus1。而后，程序执行擦除操作，将所有擦除标志位与“0xFF”比较，并将比较结果存入变量 TransferStatus2。函数原型如下：

```
int main(void)
{
#ifdef DEBUG
  debug();
#endif

  /* 配置系统时钟 ---------------------------------------------*/
  RCC_Configuration();

  /* 配置 NVIC -----------------------------------------------------*/
  NVIC_Configuration();

  /* 配置 GPIO -----------------------------------------------------*/
  GPIO_Configuration();

  /* 使能 SPI FLASH 驱动 -----------------------------------------*/
  SPI_FLASH_Init();

  /* 取得 SPI Flash ID */
  FLASH_ID = SPI_FLASH_ReadID();
  /* 检测 SPI Flash ID */
  if(FLASH_ID == M25P64_FLASH_ID)
  {
    /* 如果 ID 正确，点亮与 PC6 相连的 led 灯 */
    GPIO_WriteBit(GPIOC, GPIO_Pin_6, Bit_SET);
  }
  else
  {
    /*不正确，点亮与 PC7 相连的 led 灯 */
    GPIO_WriteBit(GPIOC, GPIO_Pin_7, Bit_SET);
  }
```

```
  /* 读操作之后执行一次写操作 -----*/
  /* 擦除 要写入的 SPI FLASH Sector  */
  SPI_FLASH_SectorErase(FLASH_SectorToErase);

  /* 将 Tx_Buffer 数据写入 SPI FLASH memory */
  SPI_FLASH_BufferWrite(Tx_Buffer, FLASH_WriteAddress, BufferSize);

  /* 将数据从 SPI FLASH memory 读出*/
  SPI_FLASH_BufferRead(Rx_Buffer, FLASH_ReadAddress, BufferSize);

  /* 检测写入数据是否正确*/
  TransferStatus1 = Buffercmp(Tx_Buffer, Rx_Buffer, BufferSize);
  /* 如果传输的数据和接收的数据相同，则 TransferStatus1 = PASSED */
  /* 如果传输的数据和接收的数据不同，则 TransferStatus1 = FAILED */

  /* 读操作之后执行一次擦除操作 ----*/
  /*擦除 要写入的 SPI FLASH Sector */
  SPI_FLASH_SectorErase(FLASH_SectorToErase);

  /*将数据从 SPI FLASH memory 读出*/
  SPI_FLASH_BufferRead(Rx_Buffer, FLASH_ReadAddress, BufferSize);

  /* 检测擦除的数据是否正确 */

  for(i=0; i<BufferSize; i++)
  {
    if(Rx_Buffer[i] != 0xFF)
    {
      TransferStatus2 = FAILED;
    }
  }
  /* 如果指定 Sector 被正确擦除，则 TransferStatus2 = PASSED */
  /* 如果指定 Sector 被未被正确擦除，则 TransferStatus2 = FAILED */

  while(1)
  {
  }
}
```

5.6 CAN 总线编程实例

5.6.1 实例内容与目标

该实例将 bxCAN 接口设置成 loopback 模式进行数据传输：通过 100kbit/s 的轮询速率对标准数据帧进行接收和传输；同时，以 500kbit/s 的速率使用中断模式对扩展数据帧进行接收和传输。

通过该实例，重点掌握以下内容：

- 了解 CAN 接口控制原理；

- 掌握 CAN 接口的基本使用方法；
- 熟悉如何使用 CAN 接口传输数据。

5.6.2 CAN 总线操作原理

CAN（Controller Area Network，控制器局部网）是 BOSCH 公司为现代汽车应用领先推出的一种多主机局部网，由于其高性能、高可靠性、实时性等优点现已广泛应用于工业自动化、多种控制设备、交通工具、医疗仪器以及建筑、环境控制等众多部门。控制器局部网将在我国迅速普及推广。CAN 总线通信接口中集成了 CAN 协议的物理层和数据链路层功能，可完成对通信数据的成帧处理，包括位填充、数据块编码、循环冗余检验、优先级判别等多项工作。

CAN 总线采用了多主竞争式总线结构，具有多主站运行和分散仲裁的串行总线以及广播通信的特点。CAN 总线上任意节点可在任意时刻主动地向网络上其他节点发送信息而不分主次，因此可在各节点之间实现自由通信。CAN 总线协议已被国际标准化组织认证，技术比较成熟，控制的芯片已经商品化，性价比高，特别适用于分布式测控系统之间的数据通信。CAN 总线插卡可以任意插在 PC AT XT 兼容机上，方便地构成分布式监控系统。

CAN 协议的一个最大特点是废除了传统的站地址编码，而代之以对通信数据块进行编码。采用这种方法的优点可使网络内的节点个数在理论上不受限制，数据块的标识码可由 11 位或 29 位二进制数组成，因此可以定义 211 或 229 个不同的数据块，这种按数据块编码的方式，还可使不同的节点同时接收到相同的数据，这一点在分布式控制系统中非常有用。数据段长度最多为 8 字节，可满足通常工业领域中控制命令、工作状态及测试数据的一般要求。同时，8 字节不会占用总线时间过长，从而保证了通信的实时性。CAN 协议采用 CRC 检验并可提供相应的错误处理功能，保证了数据通信的可靠性。CAN 卓越的特性、极高的可靠性和独特的设计，特别适合工业过程监控设备的互连，因此，越来越受到工业界的重视，并已公认为是最有前途的现场总线之一。

STM32F103 处理器的 CAN 为 bxCAN，即基本扩展 CAN（Basic Extended CAN）。它支持 2.0A 和 2.0B 版本的 CAN 协议。为了增加 CAN 的安全性，STM32F103 的 CAN 控制器在硬件层提供了支持 CAN 时间激活通信选项（CAN Time Triggered Communication option）。

CAN 总线拓扑结构如图 5.4 所示。

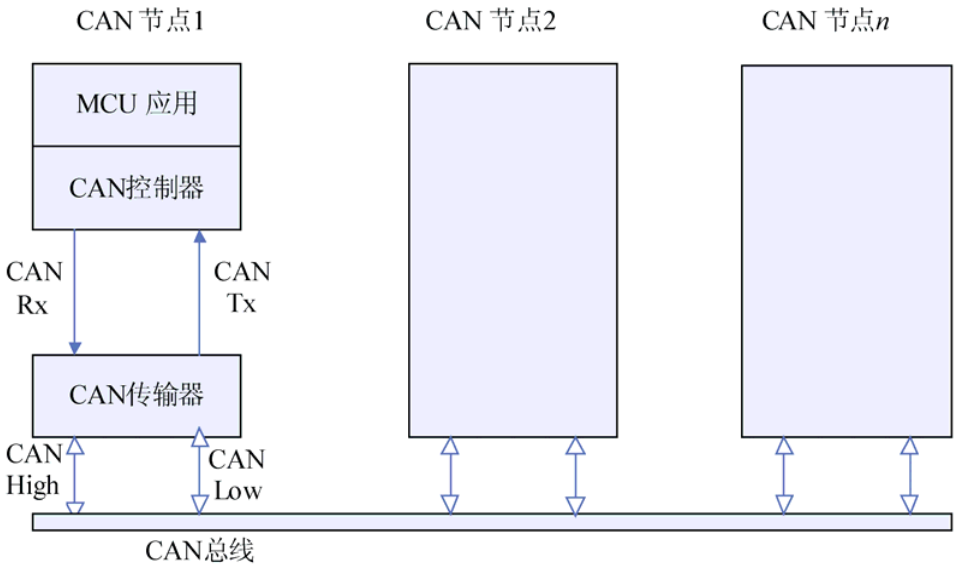

图 5.4 CAN 总线拓扑结构

CAN 总线测控系统的通信软件分为 3 部分：CAN 初始化、数据发送和数据接收。

（1）CAN 总线数据传输过程。

使用 CAN 总线进行数据传输的标准操作过程如下：

- 指定空的传输邮箱（transmit mailbox），并为该邮箱设置唯一标识；
- 设置要传输的数据长度 DLC（Data Length Code）和要传输的数据；
- 设置 CAN_TIxR 寄存器的 TXRQ 传输标志位，邮箱进入挂起（pending）状态，等待被系统分配最高优先级；
- 该邮箱得到最高优先级后，数据传输开始；
- 数据传输结束，邮箱重新进入空闲（empty）状态。

整个传输过程中，邮箱的状态转化如图 5.5 所示。

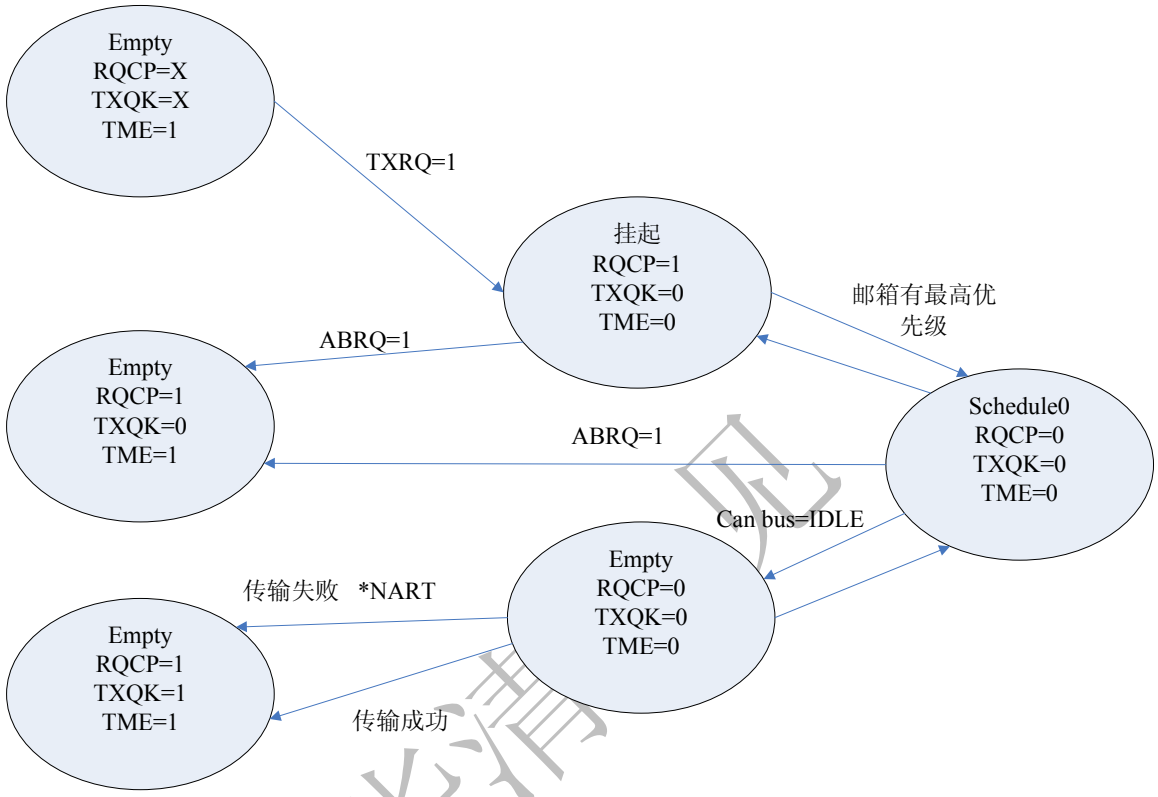

图 5.5 传输过程邮箱状态转换

（2）CAN 总线数据接收过程。

CAN 总线接收数据时，将 3 个邮箱组织成一个 FIFO buffer，为了提高系统性能，STM32F103 处理器的 CAN 控制器完全使用硬件控制接收 FIFO buffer。应用程序通过 FIFO 输出邮箱（FIFO Output Mailbox）访问接收到的数据。

数据接收的标准过程如下：

- 邮箱由空状态通过 FIFO 接收消息进入到 Pending_1 状态；
- 硬件发出接收到消息信号，将 CRFR 寄存器的 FMP[1:0]位设为 0x01；
- 软件从该邮箱读走消息，并将 CRFR 寄存器的 RFOM 位置 1；
- 邮箱重新标识为 empty 状态。
- 如果第 2 个消息到达前，FIFO 中的第 1 个邮箱非 empty 状态，则 FIFO 中第 2 个邮箱接收消息，并将状态置为 Pending_2（FMP[1：0] = 0x10）；
- 以此类推，第 3 个消息到达时，邮箱将被置为 Pending_3（FMP[1：0] = 0x11）；
- 当第 3 个消息到达后，仍有消息到达，系统发出 overrun 中断信号。

整个数据接收过程，如图 5.6 所示。

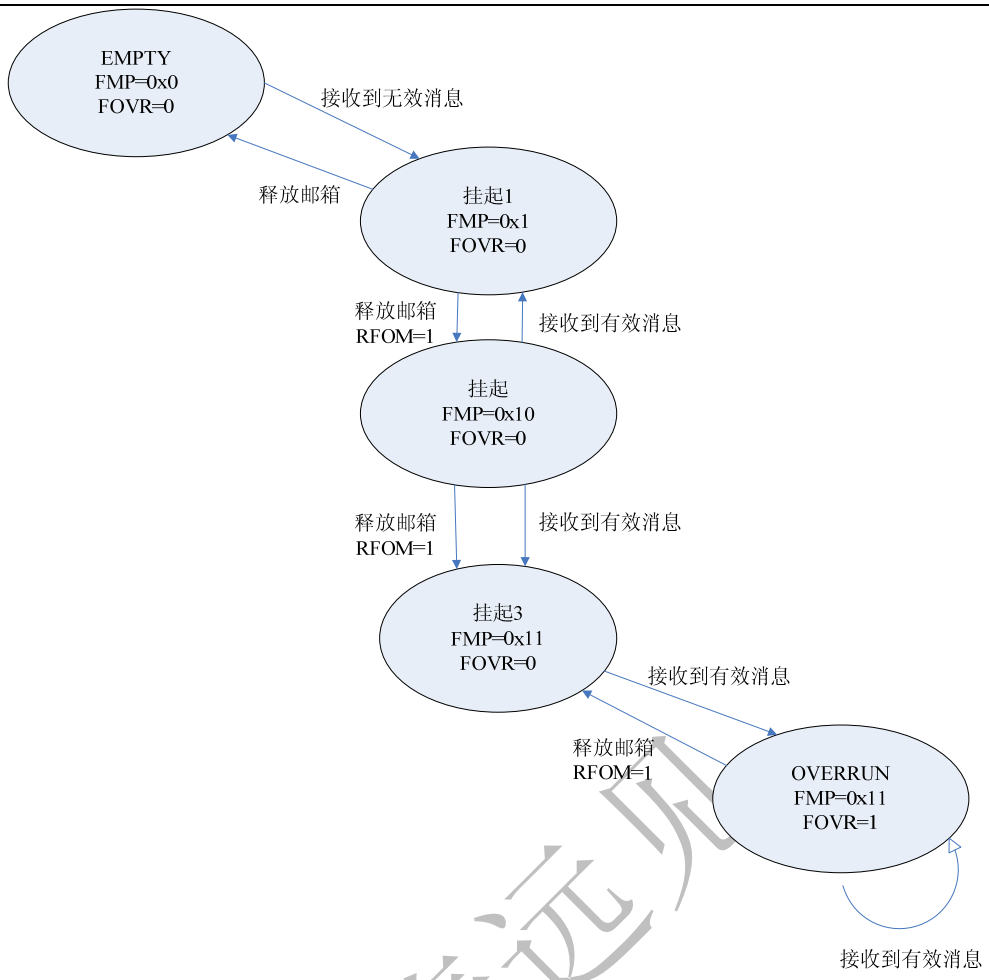

图 5.6 FIFO 接收消息过程

（3）CAN 总线相关寄存器描述。

- CAN 总线状态控制寄存器 CAN_MCR

地址偏移量：0x0；初始值：0x00010002。

表 5.21 所示为总线状态控制寄存器 CAN_MCR 的位定义。

表 5.21　　总线状态控制寄存器 CAN_MCR 位定义

寄 存 器 位	位　描　述
Bit[31：16]	保留位
Bit 16	DBF（Debug Freeze）：调试冻结设置位 0：调试状态 CAN 总线可以工作。 1：调试状态禁止 CAN 总线工作
Bit 15	RESET：bxCAN 总线软件重启 0：正常操作。 1：系统重启后强制 CAN 软件重启（强制设置 bxCAN- SLEEP 模式）
Bit[14：8]	保留位
Bit[7]	TTCM：时间激发通信模式 0：时间激发通信模式禁止。 1：时间激发通信模式使能
Bit[6]	ABOM（Automatic Bus-off Management）：自动 Bus-off 管理 0：软件设置 Bus-off 状态。 1：硬件自动设置 Bus-off 状态

Bit[5]	AWUM：自动唤醒设置位 0：通过软件清除 CAN_MCR 寄存器的 SLEEP 位使 CAN 总线进入 SLEEP 模式。 1：硬件总动设置 CAN 总线进入 SLEEP 模式
Bit[4]	NART：自动传输设置位 0：CAN 总线根据 CAN 协议标准自动判断数据是否需要重发，如果上一次数据发送状态不成功，总线自动重复该数据。 1：数据只发生一次，无论数据发送成功与否
Bit[3]	BFLM：接收 FIFO 锁定模式 0：系统发出 overrun 信号，FIFO 不会自动锁定。 1：系统发出 overrun 信号，FIFO 自动锁定
Bit[2]	TXFP：传输 FIFO 优先级 0：根据信息标识判断发送优先级。 1：根据信息到达的先后设定优先级
Bit[1]	SLEEP：SLEEP 模式请求设置位。软件设置该位，使 CAN 进入 SLEEP 模式，软件清除该位，CAN 退出 SLEEP 模式
Bit[0]	INRQ：初始化请求设置位。软件清除该位，使 CAN 进入正常工作模式

- CAN 总线主模式状态寄存器 CAN_MSR

地址偏移量：0x4；初始值：0x00000C02。

表 5.22 所示为 CAN 总线主模式状态寄存器 CAN_MSR 的位定义。

表 5.22　　CAN 总线主模式状态寄存器 CAN_MSR 位定义

寄 存 器 位	位 描 述
Bit[31：12]	保留位
Bit[11]	RX：CAN Rx 信号标识位。该位存储 CAN_RX 引脚状态
Bit[10]	SAMP：最后取样点标识位。该位标识 RX 最后取样值
Bit[9]	RXM：接收模式。设置 CAN 总线进入接收模式
Bit[8]	TXM：传输模式。设置 CAN 总线进入传输模式
Bit[7：5]	保留位
Bit[4]	SLAKI：SLEEP 模式确认中断。当 SLKIE = 1 时，该位由硬件设置，使 bxCAN 进入 SLEEP 模式
Bit[3]	WKUI：唤醒中断。当接收到 SOF 信号后，该位由硬件置位
Bit[2]	ERRI：错误中断。当 CAN_ESR 寄存器的一位由错误检测置位时，该位由硬件自动置 1
Bit[1]	SLAK：SLEEP 确认标识位
Bit[0]	初始化确认标识。当 CAN 进入初始化状态后，该位由硬件置位

- CAN 总线中断使能寄存器 CAN_IER

地址偏移量：0x14；初始值：0x0。

表 5.23 所示为 CAN 总线中断使能寄存器 CAN_IER 的位定义。

表 5.23　　CAN 总线中断使能寄存器 CAN_IER 位定义

寄 存 器 位	位 描 述
Bit[31：18]	保留位
Bit[17]	SLKIE：SLEEP 中断使能设置位 0：无中断产生。 1：有中断到来

Bit[16]	WKUIE：Wake-up 中断使能 0：当 WKUI 设置时，无中断产生。 1：当 WKUI 设置时，有中断到来
Bit[15]	ERRIE：错误中断使能 0：当错误条件在 CAN_ESR 寄存器中挂起时，无中断产生。 1：当错误条件在 CAN_ESR 寄存器中挂起时，有中断到来

续表

寄 存 器 位	位 描 述
Bit[14：12]	保留位
Bit[11]	LECIE：最后错误代码中断使能 0：当错误检测设置 LEC[2：0]时，ERRI 位不被设置。 1：当错误检测设置 LEC[2：0]时，ERRI 位将被设置
Bit[10]	BOFIE：Bus-off 中断使能 0：BOFF 设置时，ERRI 不被设置。 1：BOFF 设置时，ERRI 将被设置
Bit[9]	EPVIE：Error Passive 中断使能 0：当 EPVF 设置时，ERRI 不被设置。 1：当 EPVF 设置时，ERRI 将被设置
Bit[8]	EWGIE：错误警告中断使能 0：当 EWGF 设置时，ERRI 不被设置。 1：当 EWGF 设置时，ERRI 将被设置
Bit[7]	保留
Bit[6]	POVIE1：FIFO overrun 中断使能 0：当 FOVR 设置时，无中断。 1：当 FOVR 设置时，中断产生
Bit[5]	FFIE1：FIFO 满（FIFO full）中断使能 0：当 FULL 设置时，无中断产生。 1：当 FULL 设置时，中断产生
Bit[4]	FMPIE1：FIFO 消息挂起中断使能 0：当 FMP[1：0]位为 0 时，无中断产生。 1：当 FMP[1：0]位为 0 时，产生中断
Bit[3]	FOVIE0：FIFO overrun 中断使能 0：FOVR 设置时，无中断产生。 1：FOVR 设置时，产生中断
Bit[2]	FFIE0：FIFO 满中断使能 0：当 FULL 置位时，无中断产生。 1：当 FULL 置位时，产生中断
Bit[1]	FOVIE1：FIFO overrun 中断使能 0：FOVR 设置时，无中断产生。 1：FOVR 设置时，产生中断
Bit[0]	TMEIE：传输邮箱空中断使能 0：当 RQCPx 置位时，产生中断。 1：当 RQCPx 置位时，不产生中断

- CAN 总线错误状态寄存器 CAN_ESR

地址偏移量：0x18；初始值：0x0。

表 5.24 所示为 CAN 总线错误状态寄存器 CAN_ESR 的位定义。

表 5.24　　CAN 总线错误状态寄存器 CAN_ESR 位定义

寄 存 器 位	位 描 述

Bit[31：24]	REC[7：0] 接收错误计数。CAN 协议中错误限制机制的实现部分。当错误发生时，该寄存器根据 CAN 协议的定义对错误数加 1 或 8
Bit[23：16]	TEC[7：0] 记录错误传输中的最后一个字节。CAN 协议中错误限制机制的实现部分
Bit[15：7]	保留位
Bit[6：4]	LEC[2：0] 最后一次错误代码 000：无错误。 001：stuff 错误。 010：Form 错误。 011：Acknowledgment 错误。 100：Bit recessive 错误。 101：Bit dominant 错误。 110：CRC 错误。 111：软件设置
Bit[3]	保留
Bit[2]	BOFF：Bus-off 标志。当 CAN 总线进入 bus-off 状态后，该位由硬件自动设置
Bit[1]	EPVF：Error Passive 标志。当接收错误数或发生错误数大于 127 时，该位由硬件自动置位
Bit[0]	EWGF：错误警告标志位。当接收错误数或发生错误数大于等于 96 时，该位由硬件自动置位

更多关于 CAN 总线寄存器的内容请参见 STM32F10x 用户手册。

5.6.3 CAN 总线编程实例操作步骤

该实例使 bxCAN 工作在 loopback 模式下，使用轮询和中断两种方式进行数据传输。
实验步骤如下。
（1）准备实验环境，连接好主机-μlinker-目标板。
（2）启动 Keil μVersion 3，打开所需工程 CAN.Uv2。
（3）选择 RAM 连接文件 nvic.sct。
（4）选择 RAM 调试文件 RAM.ini。
（5）使用“Debug-Debug session”加载 image 文件。
（6）将断点设在 CAN_Polling()函数入口处，跟踪使用 CAN 轮询方式进行数据传输的过程。
（7）将断点设在 CAN_Interrupt()函数入口处，跟踪使用 CAN 中断方式进行数据传输的过程。
（8）单步跟踪 GPIO_SetBits()函数，观察如何设置 GPIO，转换 CAN 传输方式。
（9）结束调试。

5.6.4 CAN 总线实例参考程序及说明

（1）CAN 总线初始化函数 CAN_Init()。
根据 CAN 初始化结构 CAN_InitStruct 初始化 CAN 总线，其函数原型如下：

```
u8 CAN_Init(CAN_InitTypeDef* CAN_InitStruct)
{
  u8 InitStatus = 0;

  /* 检测输入参数 */
  assert(IS_FUNCTIONAL_STATE(CAN_InitStruct->CAN_TTCM));
  assert(IS_FUNCTIONAL_STATE(CAN_InitStruct->CAN_ABOM));
  assert(IS_FUNCTIONAL_STATE(CAN_InitStruct->CAN_AWUM));
```

```
  assert(IS_FUNCTIONAL_STATE(CAN_InitStruct->CAN_NART));
  assert(IS_FUNCTIONAL_STATE(CAN_InitStruct->CAN_RFLM));
  assert(IS_FUNCTIONAL_STATE(CAN_InitStruct->CAN_TXFP));
  assert(IS_CAN_MODE(CAN_InitStruct->CAN_Mode));
  assert(IS_CAN_SJW(CAN_InitStruct->CAN_SJW));
  assert(IS_CAN_BS1(CAN_InitStruct->CAN_BS1));
  assert(IS_CAN_BS2(CAN_InitStruct->CAN_BS2));
  assert(IS_CAN_PRESCALER(CAN_InitStruct->CAN_Prescaler));

  /* 初始化 CAN 请求 */
  CAN->MCR = CAN_MCR_INRQ;

  /* 检测是否有确认信号 */
  if ((CAN->MSR & CAN_MSR_INAK) == 0)
  {
    InitStatus = CANINITFAILED;
  }
  else
  {
    /* 设置连接模式 */
    if (CAN_InitStruct->CAN_TTCM == ENABLE)
    {
      CAN->MCR |= CAN_MCR_TTCM;
    }
    else
    {
      CAN->MCR &= ~CAN_MCR_TTCM;
    }

    /* 设置自动 bus-off 管理 */
    if (CAN_InitStruct->CAN_ABOM == ENABLE)
    {
      CAN->MCR |= CAN_MCR_ABOM;
    }
    else
    {
      CAN->MCR &= ~CAN_MCR_ABOM;
    }

    /* 设置自动 wake-up 模式 */
    if (CAN_InitStruct->CAN_AWUM == ENABLE)
    {
      CAN->MCR |= CAN_MCR_AWUM;
    }
    else
    {
      CAN->MCR &= ~CAN_MCR_AWUM;
    }

    /* 设置自动重传模式 */
    if (CAN_InitStruct->CAN_NART == ENABLE)
    {
```

```
        CAN->MCR |= CAN_MCR_NART;
      }
      else
      {
        CAN->MCR &= ~CAN_MCR_NART;
      }

      /* 设置接受 FIFO 锁定模式 */
      if (CAN_InitStruct->CAN_RFLM == ENABLE)
      {
        CAN->MCR |= CAN_MCR_RFLM;
      }
      else
      {
        CAN->MCR &= ~CAN_MCR_RFLM;
      }
      /* 设置传输 FIFO 优先级 */
      if (CAN_InitStruct->CAN_TXFP == ENABLE)
      {
        CAN->MCR |= CAN_MCR_TXFP;
      }
      else
      {
        CAN->MCR &= ~CAN_MCR_TXFP;
      }

      /* 设置 bit timing 寄存器*/
      CAN->BTR = (u32)((u32)CAN_InitStruct->CAN_Mode << 30) | ((u32)CAN_InitStruct-> CAN_SJW << 24)
|
                ((u32)CAN_InitStruct->CAN_BS1 << 16) | ((u32)CAN_InitStruct-> CAN_BS2 << 20)
|
                ((u32)CAN_InitStruct->CAN_Prescaler - 1);
      InitStatus = CANINITOK;

      /* 请求退出初始化状态 */
      CAN->MCR &= ~CAN_MCR_INRQ;

      /* 检测确认信号*/
      if ((CAN->MSR & CAN_MSR_INAK) != CAN_MSR_INAK)
      {
        InitStatus = CANINITFAILED;
      }
    }
    /* 返回初始化状态 */
    return InitStatus;
  }
```

（2）CAN 总线中断设置函数 CAN_ITConfig()。

该函数根据输入参数 CAN_IT，使能/禁止指定中断。可设置的中断源类型有 CAN_IT_TME、CAN_IT_FMP0、CAN_ITFF0、CAN_IT_POV0、CAN_IT_FMP1、CAN_IT_FF1、CAN_IT_FOV1、CAN_IT_EWG、CAN_IT_EPV、CAN_IT_LEC、CAN_IT_ERR、CAN_IT_WKU 和 CAN_IT_SLK。各中断的触发条件请参见 CAN 总线寄存器一节。函数体直接使用指针变量对 CAN 中断寄存器操作，函数原型如下：

```
void CAN_ITConfig(u32 CAN_IT, FunctionalState NewState)
{
  /* 检测输入参数 */
  assert(IS_CAN_ITConfig(CAN_IT));
  assert(IS_FUNCTIONAL_STATE(NewState));

  if (NewState != DISABLE)
  {
    /* 使能所选 CAN 中断 */
    CAN->IER |= CAN_IT;
  }
  else
  {
    /* 禁止所选 CAN 中断*/
    CAN->IER &= ~CAN_IT;
  }
}
```

（3）数据传输函数 CAN_Transmit()。

该函数对要传输的消息进行初始化，结构体 TxMessage 指定了 CAN 的传输信息，包括 CAN ID、CAN DLC 和 CAN 数据。函数原型如下：

```
u8 CAN_Transmit(CanTxMsg* TxMessage)
{
  u8 TransmitMailbox = 0;

  /* 检测输入参数*/
  assert(IS_CAN_STDID(TxMessage->StdId));
  assert(IS_CAN_EXTID(TxMessage->StdId));
  assert(IS_CAN_IDTYPE(TxMessage->IDE));
  assert(IS_CAN_RTR(TxMessage->RTR));
  assert(IS_CAN_DLC(TxMessage->DLC));

  /* 选择一个空的 mailbox */
  if ((CAN->TSR&CAN_TSR_TME0) == CAN_TSR_TME0)
  {
    TransmitMailbox = 0;
  }
  else if ((CAN->TSR&CAN_TSR_TME1) == CAN_TSR_TME1)
  {
    TransmitMailbox = 1;
  }
  else if ((CAN->TSR&CAN_TSR_TME2) == CAN_TSR_TME2)
  {
    TransmitMailbox = 2;
  }
  else
  {
    TransmitMailbox = CAN_NO_MB;
  }

  if (TransmitMailbox != CAN_NO_MB)
  {
    /* 启动 Id */
```

```
    TxMessage->StdId &= (u32)0x000007FF;
    TxMessage->StdId = TxMessage->StdId << 21;
    TxMessage->ExtId &= (u32)0x0003FFFF;
    TxMessage->ExtId <<= 3;

    CAN->sTxMailBox[TransmitMailbox].TIR &= CAN_TMIDxR_TXRQ;
    CAN->sTxMailBox[TransmitMailbox].TIR |= (TxMessage->StdId | TxMessage->ExtId |
                                             TxMessage->IDE | TxMessage->RTR);

    /* 启动 DLC */
   // TxMessage->DLC &= (u32)0x0000000F;
    TxMessage->DLC &= (u8)0x0000000F;
    CAN->sTxMailBox[TransmitMailbox].TDTR &= (u32)0xFFFFFFF0;
    CAN->sTxMailBox[TransmitMailbox].TDTR |= TxMessage->DLC;

    /* 启动数据域 */
    CAN->sTxMailBox[TransmitMailbox].TDLR = (((u32)TxMessage->Data[3] << 24) |
                                             ((u32)TxMessage->Data[2] << 16) |
                                             ((u32)TxMessage->Data[1] << 8) |
                                             ((u32)TxMessage->Data[0]));
    CAN->sTxMailBox[TransmitMailbox].TDHR = (((u32)TxMessage->Data[7] << 24) |
                                             ((u32)TxMessage->Data[6] << 16) |
                                             ((u32)TxMessage->Data[5] << 8) |
                                             ((u32)TxMessage->Data[4]));

  /* CAN->sTxMailBox[TransmitMailbox].TDLR = (((u32)TxMessage->Data[3] << 24) |
                                             ((u32)TxMessage->Data[2] << 16) |
                                             ((u32)TxMessage->Data[1] << 8) |
                                             ((u32)TxMessage->Data[0]));
    CAN->sTxMailBox[TransmitMailbox].TDHR = (((u32)TxMessage->Data[7] << 24) |
                                             ((u32)TxMessage->Data[6] << 16) |
                                             ((u32)TxMessage->Data[5] << 8) |
                                             ((u32)TxMessage->Data[4]));  */


    /* 请求传输 */
  CAN->sTxMailBox[TransmitMailbox].TIR |= CAN_TMIDxR_TXRQ;
  }

  return TransmitMailbox;
}
```

（4）程序主函数 main()。

主函数体分别使用轮询和中断两种方式进行数据传输。CAN_Polling()是 CAN 轮询数据传输函数，CAN_Interrupt()将 CAN 设置成中断方式进行数据传输。

```
int main(void)
{

#ifdef DEBUG
  debug();
#endif

  /* 设置系统时钟 */
  RCC_Configuration();
```

```
    /* 配置 NVIC */
    NVIC_Configuration();

    /* 配置 GPIO */
    GPIO_Configuration();

    /* 以 100Kbit/s 速率传输并且在 polling in loopback mode -1 模式下接收*/

    TestRx = CAN_Polling();

    if (TestRx == FAILED)
    {
      /* 点亮与 PC.08 pin 相连的 LED */
      GPIO_SetBits(GPIOC, GPIO_Pin_8);
    }
    else
    {
      /* 点亮与 PC.06 pin 相连的 LED */
      GPIO_SetBits(GPIOC, GPIO_Pin_6);
    }

    /* 以 500Kbit/s 速率传输并在 interrupt in loopback mode 模式下接收 */
    TestRx = CAN_Interrupt();

    if (TestRx == FAILED)
    {
      /* 点亮与 PC.09 pin 相连的 LED */
      GPIO_SetBits(GPIOC, GPIO_Pin_9);
    }
    else
    {
      /* 点亮与 PC.07 pin 相连的 LED*/
      GPIO_SetBits(GPIOC, GPIO_Pin_7);
    }

    while(1)
    {
    }
  }
```

5.7 窗口看门狗 WWDG 实例

5.7.1 实例内容与目标

该实例显示了系统在 8MHz 时钟频率下，如何对看门狗进行设置，并通过软件模拟中断，演示了在系统发生故障时，看门狗如何重启系统的过程。

通过该实例重点掌握以下内容：

- 了解 WWDG 接口控制原理；
- 掌握 WWDG 接口的基本使用方法；
- 熟悉如何使用 WWDG 监控系统。

5.7.2 WWDG 操作原理

看门狗，又叫 watchdog timer，是一个定时器电路，一般有一个输入，叫喂狗（kicking the dog or service the dog），一个输出到 MCU 的 RST 端。MCU 正常工作的时候，每隔一段时间输出一个信号到喂狗端，给 WDT 清零，如果超过规定的时间不喂狗（一般在程序跑飞时），WDT 定时超过预先设定值，就回给出一个复位信号到 MCU，使 MCU 重新开始工作。看门狗的作用就是防止程序发生死循环，或者说程序跑飞。
STM32F103 的片上看门狗组成如图 5.7 所示。

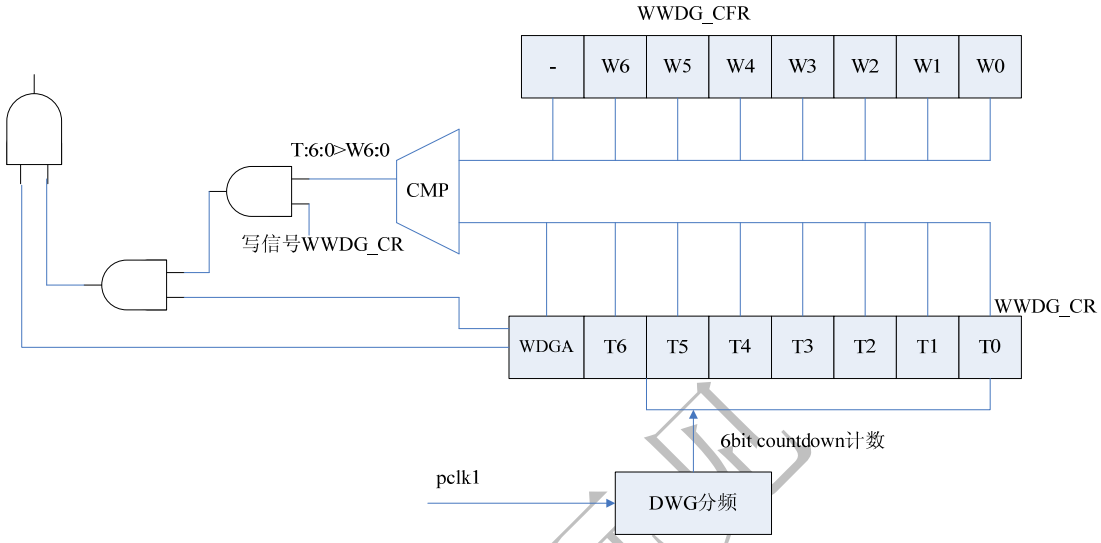

图 5.7 看门狗组成结构图

从图 5.7 可以看出，应用程序必须每隔一个固定时间“写”WWDG_CR 寄存器一次，即“喂狗”，否则系统将被重启。

（1）看门狗使能。

系统重启后，看门狗默认关闭。设置 WWDG_CR 寄存器的 WDGA 位使其进入工作状态。

注 当看门狗进入工作状态后，将不能被禁止。如果要禁止看门狗工作，只能重启系统。

（2）降值计数器（Downcounter）控制。

Downcounter 计数器的值时钟根据系统时钟不断递减，即使在看门狗模块被禁止的状态下，其值仍然在变。Bit[5：0]包含的值代表看门狗产生一次重启时间的时间延时。延时的计算方法如下：

$$T_{WWDG} = T_{PCLK1} \times 4\,096 \times 2^{WDGTB} \times (Bit[5：0] + 1)(ms)$$

其中，T_{WWDG} 为看门狗延时时间，T_{PCLK1} 为 APB1 时钟。

表 5.25 所示为系统在 36MHz（PCLK1）时钟下，看门狗的延时范围。

表 5.25 36MHz 时钟下看门狗延时范围

WDGTB	最少延时值	最大延时值
0	113μs	7.28ms
1	227μs	14.56ms
2	455μs	29.12ms
3	910μs	58.25ms

（3）看门狗寄存器。

- 看门狗控制寄存器 WWDG_CR

地址偏移量：0x0；初始值：0x7F。

表 5.26 所示为看门狗控制寄存器 WWDG_CR 的位定义。

表 5.26 看门狗控制寄存器 WWDG_CR 位定义

寄 存 器 位	位 描 述
Bit[31：8]	保留
Bit[6：0]	看门狗计数值

- 看门狗配置寄存器 WWDG_CFR

地址偏移量：0x04；初始值：0x7F。

表 5.27 所示为看门狗配置寄存器 WWDG_CFR 的位定义。

表 5.27 看门狗配置寄存器 WWDG_CFR 位定义

寄 存 器 位	位 描 述
Bit[31：10]	保留
Bit[9]	EWI：早唤醒（Early Wakeup）中断设置位。该位设置，当看门狗计数值达到 0x40 时，产生 reset 中断
Bit[8：7]	WDGTB[1：0]：时间基准设置位 00：基准时间为 PCLK1/4096/1。 01：基准时间为 PCLK1/4096/2。 10：基准时间为 PCLK1/4096/4。 11：基准时间为 PCLK1/4096/8
Bit[6：0]	WDGTB[6：0]，该值为看门狗计数值的比较值

更多关于看门狗寄存器的内容请参见 STM32F10x 用户手册。

5.7.3 看门狗实例操作步骤

（1）准备实验环境，连接好主机-μlinker-目标板。
（2）启动 Keil μVersion 3，打开所需工程 WWDG.Uv2。
（3）选择 RAM 连接文件 wwdg.sct。
（4）选择 RAM 调试文件 RAM.ini。
（5）使用“Debug-Debug session”加载 image 文件。
（6）将断点设在看门狗中断服务程序入口处，跟踪喂狗程序的基本步骤。
（7）修改程序，在看门狗中断服务程序中添加计数变量，程序执行一段时候后，从 watch 窗口观察该变量的值。
（8）按下评估板的 push 按键，跟踪程序，掌握看门狗重启系统过程。
（9）结束调试。

5.7.4 看门狗实例参考程序及说明

（1）看门狗使能函数 WWDG_Enable()。
该函数加载计数值，启动看门狗。函数原型如下：

```
void WWDG_Enable(u8 Counter)
{
  /* Check the parameters */
  assert(IS_WWDG_COUNTER(Counter));

  WWDG->CR = CR_WDGA_Set | Counter;
}
```

（2）看门狗设置预分频函数 WWDG_SetPrescaler()。

该函数设置看门狗的计数时钟预分频值，该分频值和看门狗配置寄存器 WWDG_CFR 的 Bit[8：7]相对应，取值可以为 PCLK1/4096/1、PCLK1/4096/2、PCLK1/4096/3 和 PCLK1/4096/4 中的一个。

```
void WWDG_SetPrescaler(u32 WWDG_Prescaler)
{
  u32 tmpreg = 0;

  /* Check the parameters */
  assert(IS_WWDG_PRESCALER(WWDG_Prescaler));

  /* Clear WDGTB[8:7] bits */
  tmpreg = WWDG->CFR & CFR_WDGTB_Mask;

  /* Set WDGTB[8:7] bits according to WWDG_Prescaler value */
  tmpreg |= WWDG_Prescaler;

  /* Store the new value */
  WWDG->CFR = tmpreg;
}
```

（3）设置看门狗窗口寄存器值 WWDG_SetWindowValue()。

该函数设置看门狗窗口寄存器的值，该值必须小于 0x80。

```
void WWDG_SetWindowValue(u8 WindowValue)
{
  u32 tmpreg = 0;

  /* Check the parameters */
  assert(IS_WWDG_WINDOW_VALUE(WindowValue));

  /* Clear W[6:0] bits */
  tmpreg = WWDG->CFR & CFR_W_Mask;

  /* Set W[6:0] bits according to WindowValue value */
  tmpreg |= WindowValue & BIT_Mask;

  /* Store the new value */
  WWDG->CFR = tmpreg;
}
```

（4）程序主函数 main()。

程序主函数将看门狗超时设为 262ms，刷新窗口寄存器值设为 0x41。当看门狗计数到 0x40 时，系统产生 EWI 中断。在 WWDG 中断服务程序中，刷新计数值。

主函数使用 EXT1 模拟系统故障，在 NVIC 中，将 EXT1 的优先级设为 0，WWDG 中断优先级设为 1（EXT1 的优先级高于 WWDG 优先级）。在 STM32F10x-EVAL 评估板上按下 push-button 按键，产生 EXT1 中断，程序在 EXT1 的中断服务过程中，无法刷新看门狗窗口寄存器（喂狗），从而导致系统重启。函数源程序如下：

```
int main(void)
{
#ifdef DEBUG
  debug();
#endif

  /* 配置系统时钟 -----------------------------------------------*/
  RCC_Configuration();
```

```
  /* 配置 GPIO ---------------------------------------------------*/
  GPIO_Configuration();

  /* 检测系统是否从 WWDG reset 状态返回-------------------------*/
  if(RCC_GetFlagStatus(RCC_FLAG_WWDGRST) != RESET)
  { /* WWDGRST flag set */
    /* 点亮与 PC.06 相连的 LED*/
    GPIO_WriteBit(GPIOC, GPIO_Pin_6, Bit_SET);

    /* 清除 reset 标志 */
    RCC_ClearFlag();
  }
  else
  { /* WWDGRST 标识未置*/
    /* 点亮与 PC.06 相连的 LED*/
    GPIO_WriteBit(GPIOC, GPIO_Pin_6, Bit_RESET);
  }

  /* 配置 EXTI Line9 产生下降沿中断 -----------*/
  EXTI_Configuration();

/* 配置 NVIC ---------------------------------------------------*/
  NVIC_Configuration();

/* 配置 WWDG ---------------------------------------------------*/
  /* 使能 WWDG 时钟 */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG, ENABLE);

  /* WWDG clock counter = (PCLK1/4096)/8 = 244 Hz (~4 ms)  */
  WWDG_SetPrescaler(WWDG_Prescaler_8);

  /* 设置 Window 值为 0x41 */
  WWDG_SetWindowValue(0x41);

  /* 使能 WWDG 并设置 counter 值为 0x7F, WWDG timeout = ~4 ms * 64 = 262 ms */
  WWDG_Enable(0x7F);

  /* 清除 EWI 标志 */
  WWDG_ClearFlag();

  /* 使能 EW 中断 */
  WWDG_EnableIT();

  while (1)
  {
  }
}
```

5.8 UART 编程实例

5.8.1 实例内容与目标

本例演示使用 stm32f103 UART 进行数据传输。
通过该实例，重点掌握以下内容：

- 了解 UART 接口控制原理；
- 熟悉如何使用 UART 接口传输数据；
- 了解串行数据通信的数据格式和编程方法。

5.8.2 UART 编程原理

（1）UART 通信基础知识。
DTE（Data Terminal Equipment），简称数据终端设备。DCE（Data Communications Equipment），简称数据通信设备。
事实上，RS-232C 标准的正规名称是“数据终端设备和数据通信设备之间串行二进制数据交换的接口”。通常，将通信线路终端一侧的计算机或终端称为 DTE，而把连结通信线路一侧的调制解调称为 DCE。RS-232C 标准中所提到的“发送”和“接收”，都是站在 DTE 立场上，而不是站在 DCE 的立场来定义的。由于在计算机系统中往往是 CPU 和 I/O 设备之间传送信息，两者都是 DTE，因此双方都能发送接收。
所谓“串行通信”是指 DTE 和 DCE 之间使用一根数据信号线（另外需要地线，可能还需要控制线），数据在一根数据信号线上一位一位地进行传输，每一位数据都占据一个固定的时间长度，如图 5.8 所示。这种通信方式使用的数据线少，在远距离通信中可以节约通信成本，当然，其传输速度比并行传输慢。

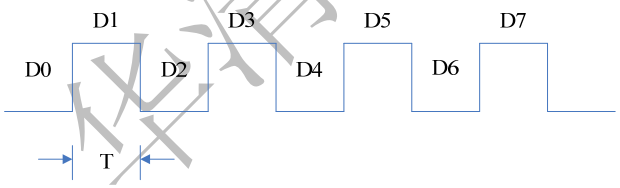

图 5.8　串口通信位传输示意图

串行通信有两种基本的类型：异步串行通信和同步串行通信。两者之间最大的差别是前者是以一个字符为单位，后者以一字符序列为单位。
本节将以异步串行通信为例，讲解串行通信的编程方法。

- 异步串行传输格式

串行通信的格式如图 5.9 所示，包括起始位、数据位和停止位。一般的，数据位和停止位是可以编程设置的。数据位有 5、6、7、8 位可选择，停止位有 1、2 位可选择。

图 5.9　异步串行通信数据传输格式

图 5.9 表明，一个完整的异步通信传输必须经历的步骤为：无传输、起始传输、数据传输、奇偶传输和停止传输。

- 串行通信数据传输过程

由于 CPU 与接口之间按并行方式传输，接口与外设之间按串行方式传输，因此，在串行接口中，必须要有“接收移位寄存器”（串→并）和“发送移位寄存器”（并→串）。典型的串行接口的结构如图 5.10 所示。在数据输入过程中，数据一位一位地从外设进入接口的“接收移位寄存器”，当“接收移位寄存器”中已接收完一个字符的各位后，数据就从“接收移位寄存器”进入“数据输入寄存器”。CPU 从“数据输入寄存器”中读取接收到的字符（并行读取，即 D7～D0 同时被读至累加器中）。“接收移位寄存器”的移位速度由“接收时钟”确定。

在数据输出过程中，CPU 把要输出的字符（并行地）送入“数据输出寄存器”，“数据输出寄存器”的内容传输到“发送移位寄存器”，然后由“发送移位寄存器”移位，把数据一位一位地送到外设。“发送移位寄存器”的移位速度由“发送时钟”确定。

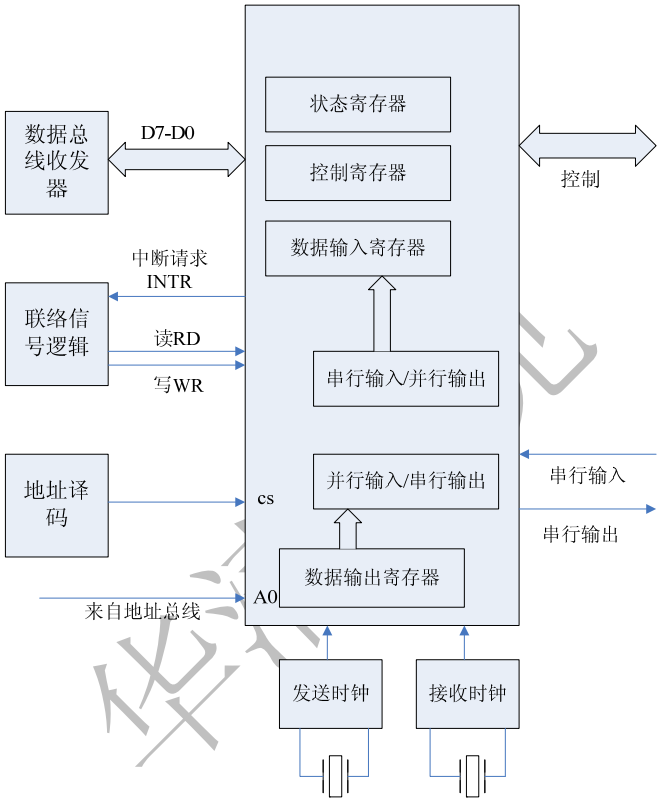

图 5.10　串口结构示意图

接口中的“控制寄存器”用来容纳 CPU 送给此接口的各种控制信息，这些控制信息决定接口工作方式。“状态寄存器”的各位称为“状态位”，每一个状态位都可以用来指示数据传输过程中的状态或某种错误。例如，用状态寄存器的 D5 位为“1”表示“数据输出寄存器”空，用 D0 位表示“数据输入寄存器”满，用 D2 位表示“奇偶检验错”等。能够完成上述“串/并”转换功能的电路，通常称为“通用异步收发器”（UART，Universal Asynchronous Receiver and Transmitter）。

（2）串口通信基本接线方法。

目前较为常用的串口有 9 针串口（DB9）和 25 针串口（DB25），通信距离较近时（＜12m），可以用电缆线直接连接标准 RS-232 端口，若距离较远，需附加调制解调器（MODEM）。最为简单且常用的是三线制接法，即地、接收数据和发送数据三脚相连。

表 5.28 所示为 DB9 和 DB25 常用信号引脚说明。

表 5.28　　DB9 和 DB25 常用信号引脚说明

9 针串口（DB9）			25 针串口（DB25）		
针号	功能说明	缩写	针号	功能说明	缩写

1	数据载波检测	DCD	8	数据载波检测	DCD
2	接收数据	RXD	3	接收数据	RXD
3	发生数据	TXD	2	发生数据	TXD
4	数据中断准备	DTR	20	数据中断准备	DTR
5	信号地	GND	7	信号地	GND
6	数据设备准备好	DSR	6	数据准备好	DSR
7	请求发送	RTS	4	请求发送	RTS
8	清除发生	CTS	5	清除发生	CTS
9	振铃指示	DELL	22	振铃指示	DELL

三线制串口数据只有接收数据引脚和发生引脚即能实现数据传输，接线方法为：同一个串口的接收引脚和发生引脚直接用线相连。表 5.29 所示为不同针脚数串口的连接方法。

表 5.29 不同针脚数串口的连接方法

9 针-9 针		25 针-25 针		9 针-25 针	
2	3	3	2	2	2
3	2	2	3	3	3
5	5	7	7	5	7

在串口调试过程中，需注意以下几点。

- 不同编制机制不能混接，如 RS-232C 不能直接与 RS-422 接口相连，必须通过转换器才能连接。
- 线路焊接要牢固，不然程序没问题，却因为接线问题误事。
- 串口调试时，准备一个好用的调试工具，如串口调试助手、串口精灵等，有事半功倍的效果。
- 通信双方的数据格式要一致。
- 建议不要带点插拔串口，插拔时至少有一端是断线的，否则串口易损坏。

（3）RS-232 串行接口标准。

RS-232 是串行数据接口标准，最初都是由电子工业协会（EIA）制订并发布的，RS-232 在 1962 年发布，命名为 EIA-232-E，作为工业标准，以保证不同厂家产品之间的兼容。

RS-232 标准只对接口的电气特性做出规定，而不涉及接插件、电缆或协议，在此基础上用户可以建立自己的高层通信协议。因此在视频界的应用，许多厂家都建立了一套高层通信协议，或公开或厂家独家使用。目前 RS-232 是 PC 机与通信工业中应用最广泛的一种串行接口。RS-232 被定义为一种在低速率串行通信中增加通信距离的单端标准。RS-232 采取不平衡传输方式，即所谓单端通信。所以 RS-232 适合本地设备之间的通信。其有关电气参数参见表 5.30 所示。

表 5.30 电气参数

规 定 标 识	RS232
工作方式	单端
节点数	1 收，1 发
最大传输电缆长度	50 英尺
最大传输速率	20kbit/s
最大驱动输出电压	+/−25V
驱动器输出信号电平（负载最小值）	+/−5V～ + /−15V
驱动器输出信号电平（空载最大值）	+/−25V
驱动器负载阻抗（Ω）	3K～7K

摆率（最大值）	30V/μs
接收器输入电压范围	+/−15V
接收器输入门限	+/−3V
接收器输入电阻（Ω）	3K～7K

在 TxD 和 RxD 上：逻辑 1(MARK) = −3V～−15V，逻辑 0(SPACE) = +3～+15V；在 RTS、CTS、DSR、DTR 和 DCD 等控制线上：信号有效（接通，ON 状态，正电压）= +3V～ + 15V，信号无效（断开，OFF 状态，负电压）= −3V～−15V。

以上规定说明了 RS-323C 标准对逻辑电平的定义。对于数据（信息码），逻辑“1”（传号）的电平低于−3V，逻辑“0”（空号）的电平高于 + 3V；对于控制信号，接通状态（ON）即信号有效的电平高于 + 3V，断开状态（OFF）即信号无效的电平低于−3V，也就是当传输电平的绝对值大于 3V 时，电路可以有效地检查出来，介于−3～ + 3V 的电压无意义，低于−15V 或高于 + 15V 的电压也认为无意义，因此，实际工作时，应保证电平在±（3～15）V。

（4）串行接口电路设计。

几乎所有的微控制器、PC 都提供串行接口。串行接口是最常用的 I/O 接口方式。

要完成最基本的串行通信功能，实际上只需要 RXD、TXD、GND 即可。但如前所述，RS-232C 标准所定义的高、低电平信号，与一般的微控制器系统的 LVTTL 电路所定义的高、低电平信号完全不同，如 S3C2410 系统的标准逻辑“1”对应 2V～3.3V 电平，标准逻辑“0”对应 0V～0.4V 电平。显然，与前面所述的 RS-232C 标准所述的电平信号完全不同。两者之间要进行通信，必须经过信号电平的转换，目前常使用的电平转换芯片有 MAX232，MAX3221～MAX3243。

STM32F10x-EVAL 评估板 UART 连接电路如图 5.11 所示。

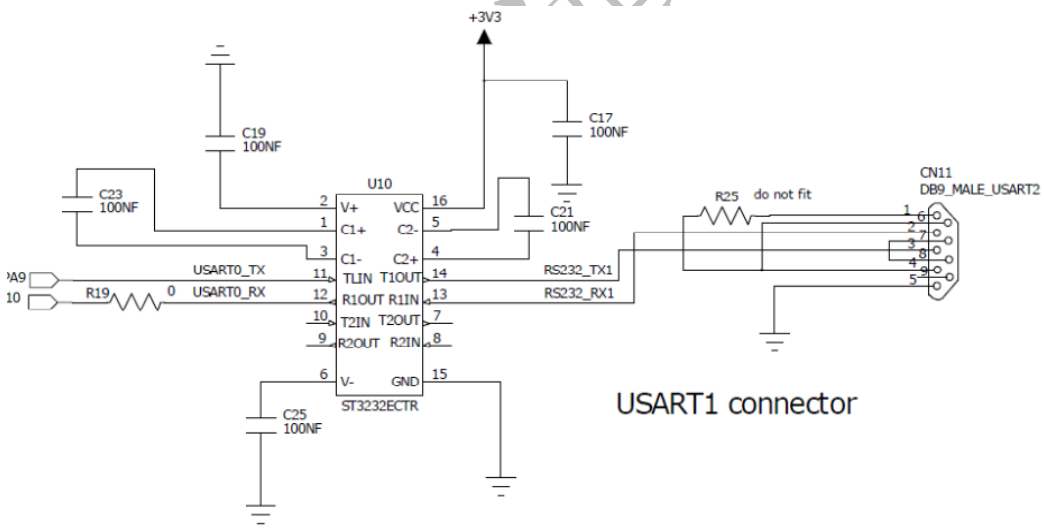

图 5.11　STM32F10x-EVAL 评估板 UART 连接电路

（5）STM32F103 串行通信工作原理。

① 串口传输过程

- 将 USART_CR1 寄存器的 UE 位置 1，使能串口。
- 设置 USART_CR1 寄存器的 M 位定义传输数据长度。
- 设置 USART_CR2 寄存器设置数据传输停止位。
- 如果使用 DMA 方式进行数据传输，设置 USART_CR3 寄存器的 DMAT 位使能串口 DMA 传输，同时设置 DMA 寄存器开始串口 DMA 传输。
- 设置 USART_CR1 寄存器的 TE 位，发生一个空闲帧，作为数据传输的开始。
- 设置 USART_BRR 寄存器设置串口传输波特率。
- 将要传输的数据写入 USART_DR 寄存器。

- 如果传输未完成，重复上述过程。

② 串口接收过程

- 将 USART_CR1 寄存器的 UE 位置“1”，使能串口。
- 设置 USART_CR1 寄存器的 M 位定义传输数据长度。
- 设置 USART_CR2 寄存器设置数据传输停止位。
- 如果使用 DMA 方式进行数据传输，设置 USART_CR3 寄存器的 DMAT 位使能串口 DMA 传输，同时设置 DMA 寄存器开始串口 DMA 传输。
- 设置 USART_BRR 寄存器设置串口传输波特率。
- 将 USART_CR1 寄存器的 RE 位置“1”，串口接收器开始搜索传输数据起始位。

当数据接收器搜索到有数据输入时，RXNE 位被硬件置“1”，同时产生串口接收中断。

> **注** 当系统发现有接收中断时，进入串口中断接收服务程序。在中断服务程序中，必须由软件清除 RXNE 位，否则，串口无法接收下一个数据。

（6）串口相关寄存器描述。

- 串口状态寄存器 USART_SR

地址偏移量：0x0；初始值：0xC0。

表 5.31 所示为串口状态寄存器 USART_SR 的位定义。

表 5.31　　串口状态寄存器 USART_SR 位定义

寄 存 器 位	位　描　述
Bit[31：10]	保留
Bit[9]	CTS：CTS 标志位 0：nCTS 无变化。 1：nCTS 发生变化
Bit[8]	LBD：LIN Break 检测标志位 0：未检测到 LIN Break 状态。 1：检测到 LIN Break 状态
Bit[7]	TXE：传输数据寄存器空标志位 0：数据传输未使用移位寄存器。 1：数据传输使用移位寄存器
Bit[6]	TC：传输空标志位 0：传输寄存器不空（当前数据未传输完）。 1：传输寄存器空
Bit[5]	RXEN：接收数据寄存器不空标志位 0：未接收到数据。 1：接收到数据，等待软件读
Bit[4]	IDLE：IDLE 状态监测 0：无 IDLE 状态。 1：IDLE 被检测
Bit[3]	ORE：overrun 错误 0：无 overrun 错误。 1：overrun 错误被检测

续表

寄 存 器 位	位　描　述
Bit[2]	NE：噪声错误标志位 0：未检测到噪声。 1：检测到数据噪声
Bit[1]	FE：数据帧错误标志位

	0：无数据帧错误。 1：串口检测到数据帧错误
Bit[0]	PE：奇偶校验错误标志位 0：无奇偶校验错。 1：检测到奇偶校验错

- 串口控制寄存器 USART_CR1

地址偏移量：0xC；初始值：0x00。

表 5.32 所示为串口控制寄存器 USART_CR1 的位定义。

表 5.32　　串口控制寄存器 USART_CR1 位定义

寄 存 器 位	位 描 述
Bit[31：14]	保留
Bit[13]	UE：串口使能设置位 0：串口分频/输出禁止。 1：串口使能
Bit[12]	M：字长设置位 0：1 位开始位，8 个数据位，*n* 个停止位。 1：1 位开始位，9 个数据位，*n* 个停止位
Bit[11]	WAKE：唤醒方法设置位 0：IDLE 线唤醒。 1：地址标识唤醒
Bit[10]	PCE：奇偶校验使能位 0：禁止奇偶校验。 1：使能奇偶校验
Bit[9]	PS：奇偶选择 0：奇校验。 1：偶校验
Bit[8]	PEIE：PE 中断使能位 0：中断禁止。 1：当 PE = 1 时，产生串口中断

续表

寄 存 器 位	位 描 述
Bit[7]	TXEIE：TXE 中断使能位 0：中断禁止。 1：当 TXE = 1 时，产生串口中断
Bit[6]	TCIE：传输完成中断使能位 0：中断禁止。 1：当 TC = 1 时，产生串口中断
Bit[5]	RXNEIE：RXNE 中断使能设置位 0：中断禁止。 1：当 ORE = 1 或 RXNE = 1 时，产生中断
Bit[4]	IDLEIE：IDLE 中断使能位 0：中断禁止。 1：当 IDLE = 1 时，产生中断
Bit[3]	TE：传输使能设置位 0：禁止传输。 1：传输使能
Bit[2]	RE：接收使能

	0：禁止串口接收。 1：使能串口接收并开始搜索是否有数据到达
Bit[1]	RWU：接收唤醒标识位 0：在 active 模式下接收。 1：在 mute 模式下接收
Bit[0]	SBK：发送终止 0：无终止符发生。 1：终止符被发生

- 串口控制寄存器 USART_CR2

地址偏移量：0x10；初始值：0x00。

表 5.33 所示为串口控制寄存器 USART_CR2 的位定义。

表 5.33　　串口控制寄存器 USART_CR2 位定义

寄 存 器 位	位　描　述
Bit[31：15]	保留
Bit[14]	LINEN：LIN 模式使能 0：LIN 模式禁止。 1：LIN 模式使能

续表

寄 存 器 位	位　描　述
Bit[13：12]	STOP：stop 位设置 00：1 位停止位。 01：0.5 位停止位。 10：2 位停止位。 11：1.5 位停止位
Bit[11]	CLKEN：时钟使能 0：SCLK 禁止。 1：SCLK 使能
Bit[10]	CPOL：时钟极性设置位 0：时钟稳定输出低。 1：时钟稳定输出高
Bit[9]	CPHA：时钟相位设置 0：数据从时钟第一个周期开始传输。 1：数据从时钟第二个周期开始传输
Bit[8]	LBCL：最后一位时钟脉冲设置位 0：时钟脉冲最后一位不从 SCLK 输出。 1：时钟脉冲从 SCLK 输出
Bit[7]	保留
Bit[6]	LBDIE：LIN 中断检测中断使能设置位 0：中断禁止。 1：当 USART_SR 寄存器的 LBD = 1 时，产生中断
Bit[5]	LBDL：LIN 中断检查长度设置位 0：10 位中断检测。 1：11 位中断检测

Bit[4]	保留
Bit[3：0]	ADD[3：0]：USART 节点地址设置位 该位存储 USART 节点地址

- 串口控制寄存器 USART_CR3

地址偏移量：0x14；初始值：0x00。

表 5.34 所示为串口控制寄存器 USART_CR3 的位定义。

表 5.34　　串口控制寄存器 USART_CR3 位定义

寄 存 器 位	位　描　述
Bit[31：11]	保留
Bit[10]	CTSIE：CTS 中断使能设置位 0：中断禁止。 1：当 USART 寄存器的 CTS = 1 时，产生中断
Bit[9]	CTSE：CTS 使能设置位 0：CTS 硬件流控禁止。 1：CTS 硬件流控使能
Bit[8]	RTSE：RTS 使能设置位 0：RTS 硬件流控禁止。 1：RTS 硬件流控使能
Bit[7]	DMAT：DMA 传输使能 1：DMA 模式使能。 0：DMA 模式禁止
Bit[6]	DMAR：DMA 接收使能设置位 1：DMA 接收使能。 0：DMA 接收禁止
Bit[5]	SCEN：智能卡模式使能设置位 0：智能卡模式禁止。 1：智能卡模式使能
Bit[4]	NACK：只能卡 NACK 使能设置位 0：当 Parity error 禁止时，使能 NACK 传输。 1：当 Parity error 使能时，使能 NACK 传输
Bit[3]	HDSEL：Half-Duplex 选择设置位 0：Half-Duplex 模式禁止。 1：Half-Duplex 模式使能
Bit[2]	IRLP：红外低电压设置位 0：红外正常模式。 1：红外低电压模式
Bit[1]	IRLP：红外设置位 0：禁止红外模式。 1：使能红外模式
Bit[0]	EIE：错误中断使能设置位 0：中断禁止。 1：当 USART_CR3 寄存器的 DMAR = 1 时，产生中断

5.8.3　串口编程实例操作步骤

（1）准备实验环境，连接好主机-μlinker-目标板。

（2）启动 Keil μVersion 3，打开所需工程 WWDG.Uv2。
（3）选择 RAM 连接文件 wwdg.sct。
（4）选择 RAM 调试文件 RAM.ini。
（5）使用“Debug-Debug session”加载 image 文件。
（6）打开 PC 机的超级终端或其他串口调试工具，将参数设为：波特率 115 200、8 位数据位、1 位停止位、无奇偶校验。
（7）从 PC 机串口输入数据，目标板将接收到的数据从串口输出。
（8）设置断点，停止程序运行，从 Watch 串口观察数组 RxBuffer 的值与接收的数据是否一致。
（9）结束调试。

5.8.4 串口编程实例参考程序及说明

（1）串口初始化函数 USART_Init()。
该函数根据输入的串口参数结构 USART_InitStruct 初始化指定串口，其中输入参数 USARTx 的取值为 1～3。程序原型如下：

```
    void USART_Init(USART_TypeDef* USARTx, USART_InitTypeDef* USART_InitStruct)
    {
      u32 tmpreg = 0x00, apbclock = 0x00;
      u32 integerdivider = 0x00;
      u32 fractionaldivider = 0x00;
      RCC_ClocksTypeDef RCC_ClocksStatus;

      /* 检测输入参数 */
      assert(IS_USART_BAUDRATE(USART_InitStruct->USART_BaudRate));
      assert(IS_USART_WORD_LENGTH(USART_InitStruct->USART_WordLength));
      assert(IS_USART_STOPBITS(USART_InitStruct->USART_StopBits));
      assert(IS_USART_PARITY(USART_InitStruct->USART_Parity));
assert(IS_USART_HARDWARE_FLOW_CONTROL(USART_InitStruct->USART_HardwareFlow Control));
      assert(IS_USART_MODE(USART_InitStruct->USART_Mode));
      assert(IS_USART_CLOCK(USART_InitStruct->USART_Clock));
      assert(IS_USART_CPOL(USART_InitStruct->USART_CPOL));
      assert(IS_USART_CPHA(USART_InitStruct->USART_CPHA));
      assert(IS_USART_LASTBIT(USART_InitStruct->USART_LastBit));

    /*---------------------------配置 USART CR2 ----------------------*/
      tmpreg = USARTx->CR2;
      /* Clear STOP[13:12], CLKEN, CPOL, CPHA and LBCL bits */
      tmpreg &= CR2_CLEAR_Mask;

      /* 配置 USART 停止位, Clock, CPOL, CPHA 和 LastBit ------------*/
      /* 根据 USART_Mode 值设置 STOP[13:12] 位 */
      /* 根据 USART_CPOL 值设置 CPOL 位*/
      /* 根据 USART_CPHA 值设置 CPHA 位 */
      /* 根据 USART_LastBit 值设置 Set LBCL 位*/
      tmpreg |= (u32)USART_InitStruct->USART_StopBits | USART_InitStruct->USART_ Clock |
                USART_InitStruct->USART_CPOL | USART_InitStruct->USART_CPHA |
                USART_InitStruct->USART_LastBit;

      /* 将配置值写入 USART CR2 寄存器 */
```

```
  USARTx->CR2 = (u16)tmpreg;

/*---------------------------- 配置USART CR1 ----------------------*/
  tmpreg = 0x00;
  tmpreg = USARTx->CR1;
  /* Clear M, PCE, PS, TE and RE bits */
  tmpreg &= CR1_CLEAR_Mask;

  /* 配置 USART 字长, 奇偶校验和模式---------------------- */
  /*根据USART_WordLength设置 M 位 */
  /* 根据USART_Parity 值设置 PCE 和 PS位 */
  /* 根据USART_Mode 值设置 TE 和 RE 位 */
  tmpreg |= (u32)USART_InitStruct->USART_WordLength | USART_InitStruct->USART_ Parity |
            USART_InitStruct->USART_Mode;

  /* 将设置值写入 USART CR1 寄存器*/
  USARTx->CR1 = (u16)tmpreg;

/*----------------------------配置 USART CR3 ----------------------*/
  tmpreg = 0x00;
  tmpreg = USARTx->CR3;
  /* 清除 CTSE 和 RTSE 位 */
  tmpreg &= CR3_CLEAR_Mask;

  /* 配置 USART HFC ------------------------------------------------*/
  /* 根据USART_HardwareFlowControl值设置 CTSE 和 RTSE 位 */
  tmpreg |= USART_InitStruct->USART_HardwareFlowControl;

  /* 将设置值写入USART CR3 寄存器*/
  USARTx->CR3 = (u16)tmpreg;

/*---------------------------- 配置USART BRR ----------------------*/
  tmpreg = 0x00;

  /* 配置 USART Baud Rate -----------------------------------------*/
  RCC_GetClocksFreq(&RCC_ClocksStatus);
  if ((*(u32*)&USARTx) == USART1_BASE)
  {
    apbclock = RCC_ClocksStatus.PCLK2_Frequency;
  }
  else
  {
    apbclock = RCC_ClocksStatus.PCLK1_Frequency;
  }

  /* 确定整数部分*/
  integerdivider = ((0x19 * apbclock) / (0x04 * (USART_InitStruct->USART_ BaudRate)));
  tmpreg = (integerdivider / 0x64) << 0x04;

  /* 确定小数部分 */
  fractionaldivider = integerdivider - (0x64 * (tmpreg >> 0x04));
```

```
    tmpreg |= ((((fractionaldivider * 0x10) + 0x32) / 0x64)) & ((u8)0x0F);

    /* 写入 USART BRR */
    USARTx->BRR = (u16)tmpreg;
  }
```

（2）中断配置函数 USART_ITConfig()。

该函数使能你/禁止指定中断，可配置的中断类型为 USART_IT_PE、USART_IT_TXE、USART_IT_TC、USART_IT_RXNE、USART_IT_IDLE、USART_IT_LBD、USART_IT_CTS 和 USART_IT_ERR，对应的中断类型分别为奇偶校验错（Parity Error）、传输数据寄存器空（Transmit Data Register Empty）、传输完成中断（Transmisson Complete）、接收数据准备好（Received Data Ready To be Read）、检测到 Edle（Idle Line Detected）、打断标志（Break Flag）、CTS 标志（CTS Flag）和传输噪声标志中断（Noise Flag）。程序源码如下：

```
  void USART_ITConfig(USART_TypeDef* USARTx, u16 USART_IT, FunctionalState NewState)
  {
    u32 usartreg = 0x00, itpos = 0x00, itmask = 0x00;
    u32 address = 0x00;

    /* 检测输入参数 */
    assert(IS_USART_CONFIG_IT(USART_IT));
    assert(IS_FUNCTIONAL_STATE(NewState));

    /* 取得 USART 寄存器索引 */
    usartreg = (((u8)USART_IT) >> 0x05);

    /* 得到中断位置 */
    itpos = USART_IT & USART_IT_Mask;

    itmask = (((u32)0x01) << itpos);
    address = *(u32*)&(USARTx);

    if (usartreg == 0x01) /* The IT  is in CR1 register */
    {
      address += 0x0C;
    }
    else if (usartreg == 0x02) /* The IT  is in CR2 register */
    {
      address += 0x10;
    }
    else /* The IT  is in CR3 register */
    {
      address += 0x14;
    }
    if (NewState != DISABLE)
    {
      *(u32*)address  |= itmask;
    }
    else
    {
      *(u32*)address &= ~itmask;
    }
  }
```

（3）串口发送数据函数 USART_SendData()。

串口发送数据的方式是当串口准备好发送数据后，直接向发送数据寄存器写数据。程序直接操作串口发送数据寄存器，源码如下：

```
void USART_SendData(USART_TypeDef* USARTx, u16 Data)
{
  /*检测输入参数*/
  assert(IS_USART_DATA(Data));

  /* 传输数据 */
  USARTx->DR = (Data & (u16)0x01FF);
}
```

（4）串口接收数据函数 USART_ReceiveData()。

串口接收数据的方式是当检测到有数据到达后，直接读数据接收寄存器，程序源码如下：

```
u16 USART_ReceiveData(USART_TypeDef* USARTx)
{
  /* 接收数据 */
  return (u16)(USARTx->DR & (u16)0x01FF);
}
```

（5）实例源码主程序 main()。

程序主函数首先从配置好的串口输出数据，然后轮询串口，将接收到的数据直接从串口输出，程序源码如下：

```
int main(void)
{  u16 i=0;


#ifdef DEBUG
  debug();
#endif

  /* 配置系统时钟 */
  RCC_Configuration();

  /* 配置 NVIC */
  NVIC_Configuration();

  /* 配置 GPIO */
  GPIO_Configuration();

  /* 配置 USART1 */
  USART_Configuration();


  printf("\r\n welcome to Embest  \r\n");
  printf("\r\n Please input number from keyboard \r\n");
while(1 )
      {
        if(USART_GetFlagStatus(USART1,USART_IT_RXNE)==SET)
        {
             i = USART_ReceiveData(USART1);
             printf("  %c",i&0xFF);     /* print the input char */
```

```
            }
        }

    }
```

联系方式

集团官网：www.hqyj.com 嵌入式学院：www.embedu.org 移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn 物联网学院：www.topsight.cn 研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218