

时钟树

如何配置时钟树？

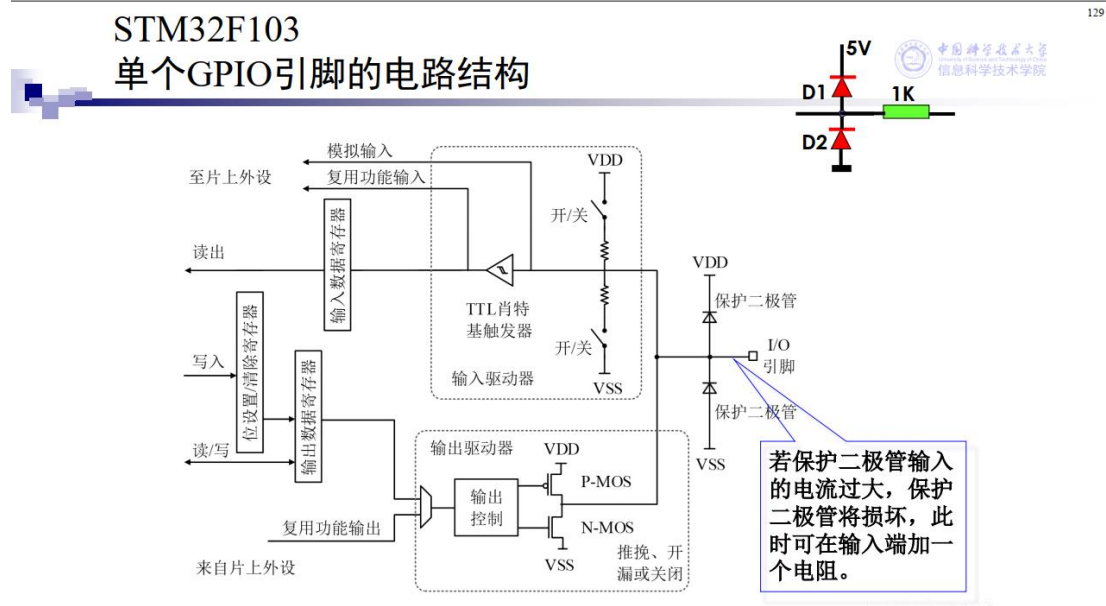
使用STM32F10x标准外设库函数

```
// STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\STM32F10x_StdPeriph_Driver\inc\stm32f10x_rcc.h
typedef struct {
    uint32_t SYSCLK_Frequency; //SYSCLK的时钟频率(Hz)
    uint32_t HCLK_Frequency; //HCLK的时钟频率(Hz)
    uint32_t PCLK1_Frequency; //PCLK1的时钟频率(Hz)
    uint32_t PCLK2_Frequency; //PCLK2的时钟频率(Hz)
    uint32_t ADCCLK_Frequency; //ADCCLK的时钟频率(Hz)
} RCC_ClocksTypeDef;

void RCC_HSEConfig(uint32_t RCC_HSE); //配置外部高速时钟
void RCC_PLLConfig(uint32_t RCC_PLLSource, uint32_t RCC_PLLMul); //配置锁相环
void RCC_PLLCmd(FunctionalState NewState); //使能或禁止锁相环
void RCC_SYSCLKConfig(uint32_t RCC_SYSCLKSource); //配置SYSCLK的源
void RCC_HCLKConfig(uint32_t RCC_SYSCLK); //配置HCLK相对SYSCLK的分频比
void RCC_PCLK1Config(uint32_t RCC_HCLK); //配置PCLK1相对HCLK的分频比
void RCC_PCLK2Config(uint32_t RCC_HCLK); //配置PCLK2相对HCLK的分频比
void RCC_AHBPeriphClockCmd(uint32_t RCC_AHBPeriph, FunctionalState NewState); //使能或禁止AHB时钟
void RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph, FunctionalState NewState); //使能或禁止APB2时钟
void RCC_APB1PeriphClockCmd(uint32_t RCC_APB1Periph, FunctionalState NewState); //使能或禁止APB1时钟
uint8_t RCC_GetSYSCLKSource(void); //返回用作系统时钟的时钟源
void RCC_GetClocksFreq(RCC_ClocksTypeDef* RCC_Clocks); //返回不同总线时钟的频率
```

标准外设库函数

8.6 ARM 中的 GPIO



GPIO 结构图

一、基本配置过程(以 STM32F407 为例)

对一个 GPIO 端口进行初始化编程需要经过三步：

1、使能该 GPIO 端口所在时钟，调用函数 `void RCC_AHB1PeriphClockCmd(uint32_t RCC_AHB1Periph, FunctionalState NewState)`，进行时钟使能，参数 `RCC_AHB1Periph` 可以为代表挂接在 AHB1 上的外设变量值，如 `RCC_AHB1Periph_GPIOA`(端口 GPIOA)，参数 `NewState` 为状态参数，如 `ENABLE` 例如 GPIOA 进行时钟使能，可以使用函数

`RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE)`

这里实验中 GPIO 端口都挂接在了 AHB1 总线上，所以使用含 AHB1 的时钟使能函数

2、对需要使用的 GPIO 端口进行初始化配置

3、调用

`void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)`

函数进行初始化，`GPIOx=GPIOA` 等，`GPIO_InitStruct` 为 `GPIO_InitTypeDef` 结构体

GPIO 端口初始化

在实验对 GPIO 端口初始化需要对 GPIO_InitTypeDef 中五个参数进行赋值，完成初始化，GPIO_InitTypeDef 如下

typedef struct

```
{  
uint32_t GPIO_Pin;  
GPIO_Mode_TypeDef GPIO_Mode;  
GPIO_Speed_TypeDef GPIO_Speed;  
GPIO_OType_TypeDef GPIO_OType;  
GPIO_PuPd_TypeDef GPIO_PuPd;  
}GPIO_InitTypeDef;
```

GPIO_Pin 代表该 GPIO 引脚线，GPIO_Mode 为 GPIO 模式，GPIO_Speed 为输入输出的频率，GPIO_OType 只有当 GPIO 对应引脚向外输出时才进行设置，GPIO_PuPd

为 GPIO 是上拉还是下拉，对应高电平有效还是低电平有效

GPIO_Mode 有如下四种模式：

- 1、GPIO_Mode_IN ： 复位状态的输入
- 2、GPIO_Mode_OUT:通用输出模式，
- 3、GPIO_Mode_AF:是复用功能模式，
- 4、GPIO_Mode_AN:是模拟输入模式

GPIO_Speed 有如下四种模式

```
typedef enum
{
    GPIO_Low_Speed      = 0x00, /*!< Low speed      */
    GPIO_Medium_Speed   = 0x01, /*!< Medium speed */
    GPIO_Fast_Speed      = 0x02, /*!< Fast speed    */
    GPIO_High_Speed      = 0x03  /*!< High speed    */
}GPIOSpeed_TypeDef;

/* Add legacy definition */

#define  GPIO_Speed_2MHz    GPIO_Low_Speed
#define  GPIO_Speed_25MHz   GPIO_Medium_Speed
#define  GPIO_Speed_50MHz   GPIO_Fast_Speed
#define  GPIO_Speed_100MHz  GPIO_High_Speed
```

GPIO_Speed 模式

GPIO_Otype 有以下两种模式

- 1、GPIO_OType_PP 推挽输出，既可以向外输出高电平，又可以输出低电平
- 2、GPIO_OType_OD 开漏输出，表示对外仅输出低电平

GPIO_PuPd 有三种模式

- 1、GPIO_PuPd_NOPULL ： 不使用上下拉，
- 2、 GPIO_PuPd_UP :上拉，
- 3、 GPIO_PuPd_DOWN :下拉

如在 LED 实验中对 GPIOG11(对应 LED0)进行推挽输出设置，过程如下

`RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOF, ENABLE);`//时钟使能

GPIO_InitTypeDef GPIO_InitStructure; //定义一个 GPIO_InitTypeDef 结构体，用于初始化

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;

GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;

GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;


GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;

GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;

GPIO_Init(GPIOG, &GPIO_InitStructure);

二、GPIO 端口的复用功能

135

 中国科学技术大学
信息科学技术学院

引脚复用、复用的重映射

- ❑ **引脚复用**通常有以下几种情况：
 - I/O引脚除通用功能外，还可设置为一些片上外设的复用功能；
 - 一个I/O引脚除可作为某个默认外设的复用引脚外，还可作为其他（多个）不同外设的复用引脚；
 - 一个片上外设，除了默认的复用引脚，还可有多个备用的复用引脚。
- ❑ **I/O引脚的复用功能重映射**
 - 可把某些外设的复用功能从默认引脚转移到备用引脚上。
- ❑ **引脚复用的优点**
 - 节约芯片引脚，可分时复用外设，虚拟增加端口数量，优化引脚配置和布线设计PCB，同时减少信号交叉干扰

清华大学

引脚复用

对 GPIO 引脚复用进行配置时，共需要两步

1、需要将该GPIO端口模式GPIO_Mode置为复用模式， GPIO_Mode_AF

2、使用

void GPIO_PinAFConfig(GPIO_TypeDef GPIOx, uint16_t GPIO_PinSource, uint8_t GPIO_AF)

，函数进行端口复用，GPIO_TypeDef GPIOx与uint16_t GPIO_PinSource组成需要

使用复用功能的GPIO端口，如GPIOA1， uint8_t GPIO_AF为使用该复用 GPIO 端

口的外设器件，如在实验四中，将PA9作为USART1的一个输出端口，使用复用映射为

GPIO_PinAFConfig(GPIOA,GPIO_PinSource10,GPIO_AF_USART1)

8.7 定时器

定时器基本电路模型

定时器的基本电路模型

□ 定时器的基本电路模型(左图)

□ STM32中的基本定时器模型(右图)

- 存放计数值，Counter register (TIMx_CNT)，16bits
- 存放预分频比，Prescaler register (TIMx_PSC)，16bits
- 存放预置数，Auto-Reload register (TIMx_ARR)，16bits

定时器有关的配置

时钟源路径如下：

- ①APB1预分频
- ②TIMxCLK
- ③CK_INT
- ④预分频(PSC)
- ⑤CK_CNT

使用定时器前应配置：

- 各级预分频器
- 预置数(TIMx_ARR)
- 是否使能计数？
- UEV是否产生中断？
-

计数模式：递增/递减/先递增再递减

一、基本定时功能实现

<p>基本定时器中，16 位计数器只能工作在向上计数模式，自动重装载寄存器 ARR 保存定时器溢出值。</p> <p>工作时，计数器从 0 开始，在定时器时钟 CK_CNT 触发下不断累加计数。当计数器 CNT 计数值等于 ARR 中保存的预设值时，产生溢出事件，可触发中断或 DMA 请求（已使能时）。然后，CNT 计数值被清零，重新开始向上计数。</p> <p>基本定时器延时时间，可由以下公式计算：延时时间=(ARR+1)*(PSC+1)/TIMxCLK</p>
基本定时器

其编程代码如下

```
void TIM3_Int_Init(u16 arr,u16 psc)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3,ENABLE);

    TIM_TimeBaseInitStructure.TIM_Period = arr;
    TIM_TimeBaseInitStructure.TIM_Prescaler=psc;
    TIM_TimeBaseInitStructure.TIM_CounterMode=TIM_CounterMode_Up;
    TIM_TimeBaseInitStructure.TIM_ClockDivision=TIM_CKD_DIV1;

    TIM_TimeBaseInit(TIM3,&TIM_TimeBaseInitStructure);
    TIM_ITConfig(TIM3,TIM_IT_Update,ENABLE);
    TIM_Cmd(TIM3,ENABLE);
}
```

二、输入捕获

1、捕获的定义及普通输入捕获

输入捕捉 (Input capture)

- 什么是输入捕捉？
 - 某个引脚上有事件发生时，记录某个定时器的数值
- 可以被捕捉的事件如
 - 在IO引脚输入电平的下降沿/上升沿捕捉定时器值
 - ◆ 引脚TIMx_CH1 ~ TIMx_CH4
- 为什么要使用输入捕捉？
 - 利用输入捕捉功能，可以对一个事件从发生到结束的时间进行**精确记录**。如**精确测量一个脉冲的宽度**。
 - 假如用2次中断的间隔来记录一个脉冲的宽度，由于中断响应需要时间，进入中断的现场保护需要时间，从而无法精确测量

图10-1-1 输入捕捉原理图

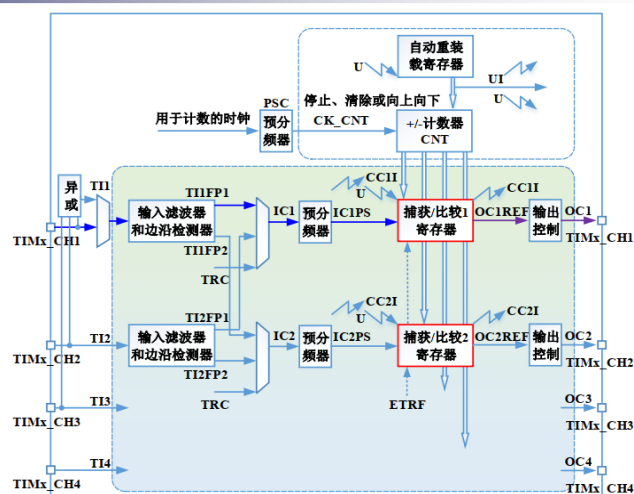
输入捕捉和输出比较电路支持

捕获/比较寄存器

CCR(Capture/Compare Register)

- 4个捕获通道IC1~IC4
- TIMx_CH1引脚信号变化时
- 捕获寄存器CCR1~CCR4
- 发生捕获时CNT计数值锁存到CCR1~CCR4

- 4个输出比较通道OC1~OC4
- CNT计数值与CCR1~CCR4匹配时，产生OC1REF，OC1REF经输出控制输出信号OC1



2、普通比较输出

输出比较 (Output compare)

- 什么是输出比较？
 - 将所选定时器计数值与寄存器预设值作比较，在比较匹配事件发生时能输出一个波形
- 可以被匹配的事件
 - 单个寄存器与单个定时器的计数值
 - 多个寄存器与多个定时器的计数值
- 可被输出的信号
 - 高电平或低电平、单个脉冲、一连串脉冲
- 为什么要使用输出比较？
 - 在精确的时间点输出一些简单的控制信号。如输出一个精确宽度的脉冲。
 - 如果用指令控制GPIO端口输出，需要计算输出高电平和低电平的持续时间，得到需要的循环次数来控制代码执行时间。难以做到精准。

图 10-10 输出比较输入/输出

输入捕获和输出比较 电路支持

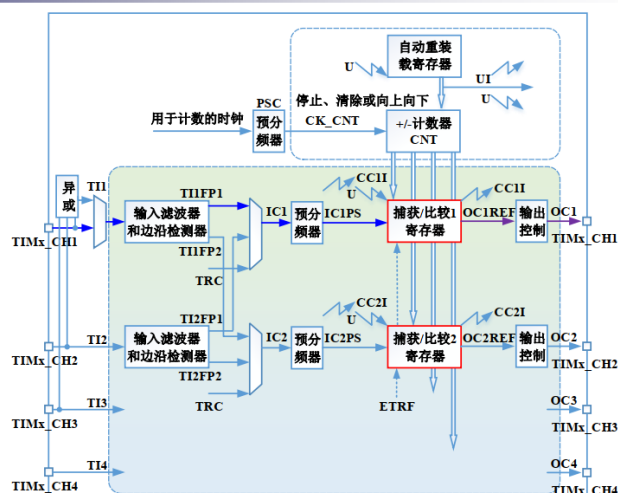
捕获/比较寄存器

CCR(Capture/Compare Register)

- 4个捕获通道IC1~IC4
- TIMx_CH1引脚信号变化时
- 捕获寄存器CCR1~CCR4
- 发生捕获时CNT计数值锁存到CCR1~CCR4

- 4个输出比较通道OC1~OC4

- CNT计数值与CCR1~CCR4匹配时，产生OC1REF，OC1REF经输出控制输出信号OC1



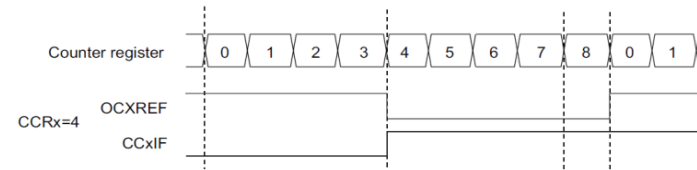
3、PWM 输入捕获

Figure 132. Example of one-pulse mode

PWM示例：输出连续波形

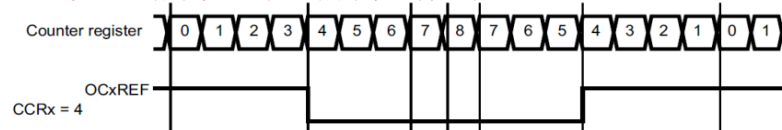
Edge-aligned PWM waveforms (ARR=8)

CNT递增计数，自然溢出后依旧递增计数...



Center-aligned PWM waveforms (ARR=8)

CNT先是递增计数，然后递减计数，再递增...



PWM 编程实现

```
void TIM2_PWM_Init(u32 arr, u32 psc)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    TIM_OCInitTypeDef TIM_OCInitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);

    GPIO_PinAFConfig(GPIOB, GPIO_PinSource10, GPIO_AF_TIM2);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;

    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    TIM_TimeBaseStructure.TIM_Prescaler = psc;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseStructure.TIM_Period = arr;
    TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
```

```

TIM_TimeBaseInit(TIM2,&TIM_TimeBaseStructure);
TIM_OCInitStructure.TIM_OCMode=TIM_OCMode_PWM1;
TIM_OCInitStructure.TIM_OutputState=TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High ;
TIM_OC3Init(TIM2, &TIM_OCInitStructure);
TIM_Cmd(TIM2, ENABLE);
}

```

8.8 中断控制器

1、NVIC 基本概念

1. 嵌套向量中断控制器 NVIC

NVIC 集成于内核中，控制并统一管理整个芯片（向量）中断，其核心功能为中断优先级分组及配置、读及清除中断请求标志、使能/禁止中断等。

STM32F10x 的中断分类如表 8.44 所示，其 NVIC 具有以下主要特性：

- ☐ 支持 84 个中断，包括 16 个内核中断和 68 个可屏蔽非内核中断；
- ☐ 使用 4 位优先级设置，具有 16 级可编程中断优先级；
- ☐ 中断响应/返回时处理器状态的自动保存/恢复均无须额外指令；
- ☐ 支持嵌套和向量中断；
- ☐ 支持中断尾链技术。

表 8.44 STM32F10x 的中断分类

STM32F10x 中断，共 84 个

133

中国科学技术大学信息科学技术学院【内部资料，禁止传播】

内核 中断 16 个	可屏蔽中断 68 个，其中可供用户编程使用 60 个					
	内部中断 41 个			外部中断 19 个		
	定时器中断	片上外设中断	外部特定中断 3 个			EXTI 线 0~15
	基本/通用/ 高级定时器	USART/SPI/ CAN/DMA...	EXTI 线 16	EXTI 线 17	EXTI 线 18	同一时刻， 线 n 只能对应 PA~Gn 中的一个
			PVD 输出	RTC 闹钟	USB 唤醒	
19 个外部中断，可独立设置上升沿/下降沿/双沿触发						

2、中断优先级

2. 中断优先级

中断优先级决定一个中断是否能被屏蔽及在未屏蔽情况下何时可以响应。

1) 优先级分组

中断优先级分为抢占优先级和子优先级。

- 抢占优先级，又称主/组/占先优先级，标识一个中断抢占式优先响应能力高低，决定是否会有中断嵌套发生。如，一个具有高抢占先优先级的中断会打断当前正在执行的中断服务程序（转而执行较高优先级中断所对应 ISR）。
- 子优先级，又称从优先级，仅在抢占优先级相同时才有影响，标识一个中断非抢占式优先响应能力高低。在抢占优先级相同时：如果有中断正被处理，高子优先级中断须等待正被响应的低子优先级中断处理结束后才能得到响应；如果没有中断正被处理，高子优先级中断将优先被响应。

2) 优先级实现

每个中断源有 4 位优先级，具有 16 级可编程中断优先级。可根据实际应用需求编程设定 4 位优先级中抢占优先级的位数。

3) 中断响应顺序

遵循以下原则：

- 先比较抢占优先级，抢占优先级高的中断优先响应；
- 当抢占优先级相同时，比较子优先级，子优先级高的中断优先响应；
- 抢占优先级和子优先级都相同时，比较中断向量表中位置，位置低的中断优先响应。

3、中断向量表

3. 中断向量表

中断向量表用于统一存放各中断对应的 ISR 入口地址，一般默认位于存储器（Flash 等）的开头（0 地址处）。在运行过程中无须修改向量表。标准的异常和中断向量表文件可参阅 `startup_stm32f10x_hd.s`，其标明了中断处理函数名称，用户不能随意定义。中断通道（即类型）NVIC_IRQChannel 在 `stm32f10x.h` 文件中进行了宏定义。68 个可屏蔽中断优先级均可设置（除个别被固定外）。

中断向量表中，优先级 7~66（中断号 0~59）代表 60 个中断，号越小，优先级越高。当某中断被触发，将通过其在表中对应地址处所存放的一条跳转指令，跳转至相应 ISR 处执行。

4、中断服务函数

4. 中断服务函数 ISR

ISR 结构上与函数相似，但 ISR 一般无参数及返回值，只有中断发生时才会被自动隐式调用执行。每个中断都有自己的 ISR，用于中断发生后执行真正意义上的处理操作。

134

中国科学技术大学信息科学技术学院【内部资料，禁止传播】

1) 特点

所有 ISR 在启动代码文件中都有预先定义，常以 XXXX_IRQHandler 命名（XXXX 为中断对应外设名，如 TIM1、USART1 等），应用时，可根据需求添加或修改（在 stm32f10x_it.c 中）。在链接生成可执行程序阶段，系统将用户（自定义的）同名 ISR 替代启动代码中的默认 ISR（以实现用户自定义的功能），因此用户不能随意更改或重定义 ISR 名称。

ISR 具有以下特点：

- 预置弱定义 WEAK 属性。除复位程序外，其他所有 ISR 均在启动代码中预设弱定义属性，使之能被其他文件中同名 ISR 替代；
- 全 C 语言实现。无须在 ISR 首尾加上汇编语言保护和恢复现场（寄存器）。中断处理过程中，保护和恢复现场工作由硬件自动完成。

2) ISR 实例

例如，要更新定时器 2 的 ISR，可直接在 stm32f10x_it.c 中新增或修改定时器 2 的 ISR：

```
void TIM2_IRQHandler(void)
{
    ...    //user code
}
```

更新时，须确保 stm32f10x_it.c 中定义的 ISR 和启动代码 startup_stm32f10x_ys.s 中相应 ISR 同名（如，对应 TIM2 的 ISR 同为 TIM2_IRQHandler），否则链接生成可执行文件时无法使用自定义 ISR 替换系统默认程序。

5、设置过程

5. 中断设置过程

设置一个中断的过程（也可称为建立一个中断），如图 8.55 所示，大体分为 5 步，简述如下。

启动代码 startup_stm32f10x_ys.s	①建立中断向量表 ②分配栈空间并初始化
用户应用程序 main.c	③设置中断优先级 ④使能中断
中断服务程序 stm32f10x_it.c	⑤编写对应中断服务程序

图 8.55 STM32F103 中断的建立过程

1) 建立中断向量表

可根据应用需求选择在 Flash 或 RAM 中建立中断向量表，二者区别如下。此步骤须在应用程序执行前完成（通常在启动过程中完成）。

- ☐ 在 Flash 中建立中断向量表：无须重新定位中断向量表；应用程序运行过程中，每个中断对应固定的 ISR 不能更改；这也是默认设置；
- ☐ 在 RAM 中建立中断向量表：需重新定位中断向量表；应用程序运行过程中，可根据需要动态地改变 ISR。

2) 分配栈空间并初始化

执行 ISR 时，进入 Handler 模式，会使用主堆栈的栈顶指针 MSP。类似地，此步骤也须在应用程序执行前完成（通常也是在启动过程中完成）。

- ☐ 分配栈空间：栈空间的分配通常位于启动代码的起始位置；为保证中断响应和返回时有足够空间保护和恢复现场(xPSR、PC、LR、R12、R3~R0 共 8 个寄存器)，应在 RAM 中为栈分配足够大空间，避免中断发生（尤其是嵌套中断）时主堆栈溢出。

3) 设置中断优先级

中断优先级的设置通过配置 NVIC 实现，可依次分两步完成：

①设置中断优先级分组的位数。确定在表示中断优先级的 4 位中抢占优先级和子优先级各自所占位数。根据中断总数及是否存在中断嵌套，可有 5 种方式：

- ☐ NVIC_PriorityGroup_0：抢占优先级 0 位，子优先级 4 位，此时不会发生中断嵌套；
- ☐ NVIC_PriorityGroup_1：抢占优先级 1 位，子优先级 3 位；即抢占优先级在 0~1（1 位范围 0~1）中取值，子优先级在 0~7（3 位范围 000~111）中取值；以下类推；
- ☐ NVIC_PriorityGroup_2：抢占优先级 2 位，子优先级 2 位；
- ☐ NVIC_PriorityGroup_3：抢占优先级 3 位，子优先级 1 位；
- ☐ NVIC_PriorityGroup_4：抢占优先级 4 位，子优先级 0 位。

②设置中断的抢占优先级和子优先级。根据①中所给出优先级分组情况，分别设置抢占优先级和子优先级，即在各自取值范围中设置优先级的确定值。

4) 使能中断

设置完中断优先级后，通过禁止中断总屏蔽位和分屏蔽位，可以使能对应中断。

5) 编写对应 ISR 代码

ISR 具体内容由用户使用 C 语言编写，实现中断具体处理。通常在最后，退出 ISR 前清除对应中断标志位，表示该中断已处理完毕，否则，该中断请求始终存在，该 ISR 将被反复执行。

6、库函数编程

基本流程

(1)、使能 IO 口时钟，初始化 IO 口为输入

首先，我们要使用 IO 口作为中断输入，所以我们要使能相应的 IO 口时钟，以

及初始化相应的 IO 口为输入模式，具体的使用方法与第二章 GPIO 实验是一致的，这里就不做过多讲解

(2)、开启 SYSCFG 时钟，设置 IO 口与中断线的映射关系。

配置 GPIO 与中断线的映射关系，那么我们首先需要打开 SYSCFG 时钟。实验中使用的函数是：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);//使能 SYSCFG 时钟
```

在库函数中，配置 GPIO 与中断线的映射关系的函数 SYSCFG_EXTILineConfig() 来实现的：

```
void SYSCFG_EXTILineConfig(uint8_t EXTI_PortSourceGPIOx, uint8_t EXTI_PinSourcex);
```

该函数将 GPIO 端口与中断线映射起来，使用示例是：

```
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource0);
```

GPIOA0与EXTI0中断线进行映射连接，

(3)、初始化线上中断，设置触发条件等。

中断线上中断的初始化是通过函数 EXTI_Init()实现的。EXTI_Init()函数的定义是：

```
void EXTI_Init(EXTI_InitTypeDef* EXTI_InitStructure);
```

下面我们用一个使用范例来说明这个函数的使用：

```
EXTI_InitTypeDef EXTI_InitStructure;
EXTI_InitStructure.EXTI_Line=EXTI_Line0;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt; //中断事件
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling; //下降沿触发
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure); //初始化外设 EXTI 寄存器
```

上面的例子设置中断线 0 上的中断为下降沿触发。STM32 的外设的初始化都是通过结构体来设置初始值的，这里就不再讲解结构体初始化的过程了，结构体 EXTI_InitTypeDef 的成员变量：

```
typedef struct
{
    uint32_t EXTI_Line;
    EXTIMode_TypeDef EXTI_Mode;
    EXTITrigger_TypeDef EXTI_Trigger;
    FunctionalState EXTI_LineCmd;
}EXTI_InitTypeDef;
```

第一个参数是中断线的标号，对于我们的外部中断，取值范围为 EXTI_Line0

~EXTI_Line15。也就是说，这个函数配置的是某个中断线上的中断参数。第二个参数是中断模式，可选值为中断 EXTI_Mode_Interrupt 和事件 EXTI_Mode_Event。第三个参数是触发方式，可以是下降沿触发 EXTI_Trigger_Falling，上升沿触发 EXTI_Trigger_Rising，或者任意电平（上升沿和下降沿）触发 EXTI_Trigger_Rising_Falling，最后一个参数就是使能中断线了。

(4) 配置中断分组 (NVIC)，并使能中断。

设置好中断线和 GPIO 映射关系，然后又设置好了中断的触发模式等初始化参数。既然是外部中断，涉及到中断还要设置 NVIC 中断优先级，设置中断线 0 的中断优先级。

```
NVIC_InitTypeDef NVIC_InitStructure;
NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;           //使能按键外部
中断通道
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x02; //抢占优先级 2,
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x02;      //响应优先级 2
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;           //使能外部中断通道
NVIC_Init(&NVIC_InitStructure);                           //中断
优先级分组初始化
```

(5)、编写中断服务函数

配置完中断优先级之后，需要编写中断服务函数。中断服务函数的名称在系统启动文件中事先定义，STM32F4 的 IO 口外部中断函数有 7 个，分别为：

```
EXPORT  EXTI0_IRQHandler
EXPORT  EXTI1_IRQHandler
EXPORT  EXTI2_IRQHandler
EXPORT  EXTI3_IRQHandler
EXPORT  EXTI4_IRQHandler
EXPORT  EXTI9_5_IRQHandler
EXPORT  EXTI15_10_IRQHandler
```

这些函数在系统启动文件中定义，名称不能有任何变化，我们使用时直接使用即可。

中断线 0-4 每个中断线对应一个中断函数，中断线 5-9 共用中断函数 EXTI9_5_IRQ

Handler，中断线 10-15 共用中断函数 EXTI15_10_IRQHandler。在编写中断服务函数的时候会经常使用到两个函数，第一个函数是判断某个中断线上的中断是否发生（标志位是否置位）：ITStatus EXTI_GetITStatus(uint32_t EXTI_Line); 这个函数一般使用在中断服务函数的开头判断中断是否发生。另一个函数是清除某个中断线上的中断标志位：

```
void EXTI_ClearITPendingBit(uint32_t EXTI_Line);
```

这个函数一般应用在中断服务函数结束之前，清除中断标志位。

常用的中断服务函数格式为：

```
void EXTI0_IRQHandler(void)
{
    if (EXTI_GetITStatus(EXTI_Line0)!=RESET)//判断线上的中断是否发生
    {
        delay_ms(10);//延时调用，需包含相关头文件
        ..... //中断操作，实现 LED 控制，需添加 LED 相关头文件
        EXTI_ClearITPendingBit(EXTI_Line0); //清除 LINE 上的中断标志位
    }
}
```

注意：该函数添加在 USER 目录下的文件 stm32f4xx_it.c 中。

具体代码

```

void EXTIX_Init(void)
{
    NVIC_InitTypeDef  NVIC_InitStructure;
    EXTI_InitTypeDef  EXTI_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);
    SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOE, EXTI_PinSource0);

    EXTI_InitStructure.EXTI_Line = EXTI_Line0;
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising; // 上升沿触发
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
    EXTI_Init(&EXTI_InitStructure);

    NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x00;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x02;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
}

```

```

int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
    delay_init(168);
    LED_Init();
    KEY_Init();
    EXTIX_Init();
    while(1)
    {

    }
}

```

8.9 USART

串口基本介绍

4.1 STM32 的串口

USART 即通用同步异步收发器，它能够灵活地与外部设备进行全双工数据传输，满足外部设备对工业标准 NRZ 异步串行数据格式的要求。UART 即通用异步收发器，它是在 USART 基础上裁剪掉了同步通信功能，同步和异步主要看其是否需要同步时钟。

STM32F407 有 2 个 UART（通用异步收发器），4 个 USART（通用异步/同步收发器）。它们都具有串口通信功能，USART 支持同步单向通信和半双工单线通信；还支持 LIN（域互连网络）、智能卡协议，以及调制解调器操作（CTS/RTS）。同时还支持多处理器通信和 DMA 功能，使用 DMA 可实现高速数据通信。

基本流程

(1)、串口时钟和 GPIO 时钟使能

串口 1 是挂载在 APB2 下面的外设，所以使能函数为：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE); //使能 USART 时钟
```

GPIO 时钟使能

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE); //使能 GPIOA 端口
```

(2)、设置引脚复用器映射

```
GPIO_PinAFConfig(GPIOA, GPIO_PinSource9, GPIO_AF_USART1); //PA9 复用为 USART1 发送端
```

```
GPIO_PinAFConfig(GPIOA, GPIO_PinSource10, GPIO_AF_USART1); // PA10 复用为 USART1 接收端
```

(3)、GPIO 端口模式设置： PA9 和 PA10 要设置为复用功能

```

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 ;//GPIOA9
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;//复用功能
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //速度 50MHz
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; //推挽复用输出
GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA9

GPIO_InitStructure.GPIO_Pin =  GPIO_Pin_10;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化 PA10

```

(4)、串口参数初始化： 设置波特率， 字长， 奇偶校验等参数

串口初始化是调用函数 USART_Init 来实现的，具体设置方法如下：

```

USART_InitStructure.USART_BaudRate = bound;
USART_InitStructure.USART_WordLength = USART_WordLength_8b;//字长为 8 位数据
USART_InitStructure.USART_StopBits = USART_StopBits_1;//一个停止位
USART_InitStructure.USART_Parity = USART_Parity_No;//无奇偶校验位
USART_InitStructure.USART_HardwareFlowControl =
USART_HardwareFlowControl_None;

```

```

USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;//收发模式
USART_Init(USART1, &USART_InitStructure); //初始化串口

```

(5)、使能串口

```

USART_Cmd(USART1, ENABLE); //使能串口 1

```

(6)、串口数据发送与接收

```
void USART_SendData(USART_TypeDef * USARTx, uint16_t Data);
```

该函数向串口发送数据寄存器 USART_DR 写入一个数据。

```
uint16_t USART_ReceiveData(USART_TypeDef * USARTx);
```

该函数从串口接受数据寄存器 USART_DR 读取接收到的数据。

(7)、串口状态

串口的状态可以通过状态寄存器 USART_SR 读取。主要是 RXNE 和 TXE。RXNE（读数据寄存器非空），当该位被置 1 的时候，提示已经有数据被接收到了，并且可以读出。通过读取 USART_DR 可以读取数据，通过读 USART_DR 可以将该位清零，也可以向该位写 0，直接清除。

TXE（发送寄存器空），当 TDR 寄存器的内容已传输到移位寄存器时，该位由硬件置 1。如果 USART_CR1 寄存器中 TXEIE 位 = 1，则会生成中断。通过对 USART_DR 寄存器执行写入操作将该位清零。0：数据未传输到移位寄存器，1：数据传输到移位寄存器。

(8)、开启中断并且初始化 NVIC，使能相应中断

要开启串口中断必须要配置 NVIC 中断优先级分组，通过调用函数 NVIC_Init 来设置。

```
NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;  
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=1;//抢占优先级 1  
NVIC_InitStructure.NVIC_IRQChannelSubPriority =1;      //响应优先级 1  
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;        //IRQ 通道使能  
NVIC_Init(&NVIC_InitStructure);    //根据指定的参数初始化 NVIC 寄存器、
```

同时还需要使能相应中断：

```
void USART_ITConfig(USART_TypeDef* USARTx, uint16_t USART_IT,  
FunctionalState NewState)
```

这个函数的第二个入口参数是标示使能串口的类型，也就是使能哪种中断，因为串口的中断类型有很多种。比如在接收到数据的时候（RXNE 读数据寄存器非空），要产生中断，开启中断的方法是：

```
USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);//开启中断，接收到数据中  
断
```

发送数据结束时（TC，发送完成）要产生中断，方法是：

```
USART_ITConfig(USART1, USART_IT_TC, ENABLE);
```

因为我们实验开启了串口中断，所以我们在系统初始化的时候需要先设置系统的中断优先级分组，代码如下：

```
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //设置系统中断优先级分组 2  
设置分组为 2，也就是 2 位抢占优先级，2 位响应优先级。
```

(9)、获取相应中断状态

使能了某个中断的时候，当该中断发生了，就会设置状态寄存器中的某个标志位。经常在中断处理函数中，要判断该中断是哪种中断，使用的函数是：

```
ITStatus USART_GetITStatus(USART_TypeDef* USARTx, uint16_t USART_IT)
```

使能了串口接收完成中断，那么当中断发生了，可以在中断处理函数中调用这个函数来判断到底是否是串口完成中断，方法是：

```
USART_GetITStatus(USART1, USART_IT_RXNE) != RESET
```

(10) 中断服务函数

串口 1 中断服务函数为：

void USART1_IRQHandler(void) ;

当发生中断的时候，程序就会执行中断服务函数。

具体代码

```
void usart1_init(u32 bound)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    USART_InitTypeDef USART_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA,ENABLE);

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1,ENABLE);

    GPIO_PinAFConfig(GPIOA,GPIO_PinSource9,GPIO_AF_USART1);
    GPIO_PinAFConfig(GPIOA,GPIO_PinSource10,GPIO_AF_USART1);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;GPIO_Init(GPIOA,&GPIO_InitStructure);
    USART_InitStructure.USART_BaudRate = bound;
```

```

USART_InitStructure.USART_WordLength =USART_WordLength_8b;
USART_InitStructure.USART_StopBits = USART_StopBits_1;
USART_InitStructure.USART_Parity = USART_Parity_No;
USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
USART_Init(USART1, &USART_InitStructure);
USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);
NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=1;
NVIC_InitStructure.NVIC_IRQChannelSubPriority =1;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
USART_Cmd(USART1, ENABLE);
USART_TX_EN=1;
USART_RX_STA=0;
}

```

```

void USART1_IRQHandler(void)
{
    u8 Res;
    if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
    {
        Res =USART_ReceiveData(USART1);
        USART_RX_BUF[USART_RX_STA]=Res;
        USART_RX_STA++;
        USART_RX_STA=USART_RX_STA%200;

        USART_ClearITPendingBit(USART1, USART_IT_RXNE);
    }
}

```