# 目录

1. [实验 1《GPIO 端口实验》](#_bookmark0) 3

[1.1 实验目的](#_bookmark1) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

[1.2 实验内容](#_bookmark2) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

[1.3 实验方法](#_bookmark3) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

[1.3.1 实验原理](#_bookmark4) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

[1.3.2 实验方案及调试过程](#_bookmark5) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

* 1. [实验步骤](#_bookmark6) 10
     1. [准备实验环境](#_bookmark7) 10
     2. [串口接收设置](#_bookmark8) 10
     3. [打开实验例程](#_bookmark9) 10
     4. [观察实验结果](#_bookmark10) 11
  2. [实验结果](#_bookmark11) 11
  3. [实验结论](#_bookmark12) 12
  4. [实验小结](#_bookmark13) 12

1. [实验 2《中断实验》](#_bookmark14) 13
   1. [实验目的](#_bookmark15) 13
   2. [实验内容](#_bookmark16) 13
   3. [实验方法](#_bookmark17) 13
      1. [实验原理](#_bookmark18) 13
      2. [实验方案及调试过程](#_bookmark19) 19
   4. [实验步骤](#_bookmark20) 23
      1. [准备实验环境](#_bookmark21) 23
      2. [串口接收设置](#_bookmark22) 23
      3. [打开实验例程](#_bookmark23) 23
      4. [观察实验结果](#_bookmark24) 24
   5. [实验结果](#_bookmark25) 24
   6. [实验结论](#_bookmark26) 25
   7. [实验小结](#_bookmark27) 25
2. [实验 3《Uart+ 定时器实验》](#_bookmark28) 26
   1. [实验目的](#_bookmark29) 26
   2. [实验内容](#_bookmark30) 26
   3. [实验方法](#_bookmark31) 27
      1. [实验原理](#_bookmark32) 27
      2. [实验方案及调试过程](#_bookmark33) 31
   4. [实验步骤](#_bookmark34) 32
      1. [准备实验环境](#_bookmark35) 32
      2. [串口接收设置](#_bookmark36) 32
      3. [打开实验例程](#_bookmark37) 32
      4. [观察实验结果](#_bookmark38) 32
   5. [实验结果](#_bookmark39) 33
   6. [实验结论](#_bookmark40) 34
   7. [实验小结](#_bookmark41) 34
3. [实验 4《µC/OS-III 实验——信号量》](#_bookmark42) 35
   1. [实验目的](#_bookmark43) 35
   2. [实验内容](#_bookmark44) 35
   3. [实验方法](#_bookmark45) 35
      1. [实验原理](#_bookmark46) 35
      2. [实验方案及调试过程](#_bookmark47) 36
   4. [实验步骤](#_bookmark48) 39
      1. [准备实验环境](#_bookmark49) 39
      2. [串口接收设置](#_bookmark50) 39
      3. [打开实验例程](#_bookmark51) 39
      4. [观察实验结果](#_bookmark52) 40
   5. [实验结果](#_bookmark53) 40
   6. [实验结论](#_bookmark54) 41
   7. [实验小结](#_bookmark55) 41

[5 附录](#_bookmark56) 42

* 1. [实验 1 源代码](#_bookmark57) 42
  2. [实验 2 源代码](#_bookmark58) 44
  3. [实验 3 源代码](#_bookmark59) 46
  4. [实验 4 源代码](#_bookmark60) 49

1. 实验 **1**《**GPIO** 端口实验》

实验学时：2 每组人数：3

实验类别：2 （1：基础性 2：综合性 3：设计性 4：研究性） 实验要求：1 （1：必修 2：选修 3：其它）

实验类别：3 （1：基础 2：专业基础 3：专业 4：其它）

## 实验目的

* + 1. 熟悉并掌握 Keil MDK 开发环境的使用以及在线调试方法；
    2. 掌握 STM32F746NG 芯片 GPIO 端口寄存器的配置；
    3. 通过实验掌握 Cortex-M7 控制 GPIO 端口的方法，实现对 LED 的控制。

## 实验内容

编写程序，对指定 GPIO 端口进行初始化并完成配置过程，实现对 LED 的控制。学习使用超级终端，对其进行配置并完成串口调试。实验中观察 GPIO 端口输出数据寄存器

（GPIOx\_ODR）的值对 LED 灯的明灭的影响，学习 GPIO 端口的输入输出方式、输出类型和输出速度的设置方法。

## 实验方法

* + 1. 实验原理

STM32F746NG 芯片共有 168 个 GPIO（通用 I/O）端口（GPIOA GPIOK），每个 GPIO

端口均可以通过软件配置为输出、输入或复用功能。

* + - 1. GPIO 主要特性
         * 输出状态：推挽或开漏 + 上拉/下拉；
         * 从输出数据寄存器（GPIOx\_ODR）或外设（复用功能输出）输出数据；
         * 可为每个 I/O 选择不同的速度；
         * 输入状态：浮空、上拉/下拉、模拟；
         * 将数据输入到输入数据寄存器（GPIOx\_ODR）或外设（复用功能输入）；
         * 置位和复位寄存器（GPIOx\_BSRR），对 GPIOx\_ODR 具有按位写权限；
         * 锁定机制（GPIOx\_LCKR），可冻结 I/O 端口配置；
         * 模拟功能；
         * 复用功能选择寄存器；
         * 快速翻转，每次翻转最快只需要两个时钟周期；
         * 引脚复用非常灵活，允许将 I/O 引脚用作 GPIO 或多种外设功能中的一种。
      2. GPIO 功能描述

每个 GPIO 端口的各个端口位均可以自由编程，通过对相关寄存器的修改可以配置为多种模式：

* + - * + 浮空输入
        + 上拉输入
        + 下拉输入
        + 模拟输入/输出
        + 具有上拉或下拉功能的开漏输出
        + 具有上拉或下拉功能的推挽输出
        + 具有上拉或下拉功能的复用功能推挽
        + 具有上拉或下拉功能的复用功能开漏

对 I/O 端口进行编程作为输入时，输出缓冲器被禁止，施密特触发器输入被打开，根据 GPIOx\_PUPDR 寄存器中的值决定是否打开上拉和下拉电阻，输入数据寄存器每隔 1 个 AHB 时钟周期对 I/O 引脚上的数据进行一次采样，对输入数据寄存器的读访问可获取 I/O 状态。

对 I/O 端口进行编程作为输出时，输出缓冲器被打开，施密特触发器输入被打开，根据 GPIOx\_PUPDR 寄存器中的值决定是否打开上拉和下拉电阻，输入数据寄存器每隔 1 个 AHB 时钟周期对 I/O 引脚上的数据进行一次采样，对输入数据寄存器的读访问可获取 I/O 状态，对输出数据寄存器的读访问可获取最后的写入值。

对 I/O 端口进行编程作为复用功能时，可将输出缓冲器配置为开漏或推挽模式，输出缓冲器由来自外设的信号驱动，施密特触发器输入被打开，根据 GPIOx\_PUPDR 寄存器中的值决定是否打开上拉和下拉电阻，输入数据寄存器每隔 1 个 AHB 时钟周期对 I/O 引脚上的数据进行一次采样，对输入数据寄存器的读访问可获取 I/O 状态。

对 I/O 端口进行编程作为模拟配置时，输出缓冲器被禁止，施密特触发器输入停用， I/O 引脚的每个模拟输入的功耗变为零，施密特触发器的输出被强制处理为恒定值（0），弱上拉和下拉电阻被硬件关闭，对输入数据寄存器的读访问值为“0”。

* + - 1. GPIO 寄存器

每个通用 I/O 端口包括 4 个 32 位配置寄存器（GPIOx\_MODER、GPIOx\_OTYPER、GPIOx\_OSPEEDR 和 GPIOx\_PUPDR）、2 个 32 位数据寄存器（GPIOx\_IDR 和GPIOx\_ODR）和 1 个 32 位置位/复位寄存器 (GPIOx\_BSRR)。此外，所有 GPIO 都包括 1 个 32 位锁定寄存器 (GPIOx\_LCKR) 和 2 个 32 位复用功能选择寄存器

（GPIOx\_AFRH 和 GPIOx\_AFRL）。I/O 端口寄存器必须按 32 位字、半字或字节进行访问。

* + - 1. GPIO 寄存器边界地址

STM43F746NG 的 GPIO 寄存器边界地址如表所示

|  |  |
| --- | --- |
| 外设 | 寄存器边界地址 |
| GPIOA | 0x4002 0000–0x4002 03FF |
| GPIOB | 0x4002 0400–0x4002 07FF |
| GPIOC | 0x4002 0800–0x4002 0BFF |
| GPIOD | 0x4002 0C00–0x4002 0FFF |
| GPIOE | 0x4002 1000–0x4002 13FF |
| GPIOF | 0x4002 1400–0x4002 17FF |
| GPIOG | 0x4002 1800–0x4002 1BFF |
| GPIOH | 0x4002 1C00–0x4002 1FFF |
| GPIOI | 0x4002 2000–0x4002 23FF |

* + - 1. 实验电路

实验电路如图所示

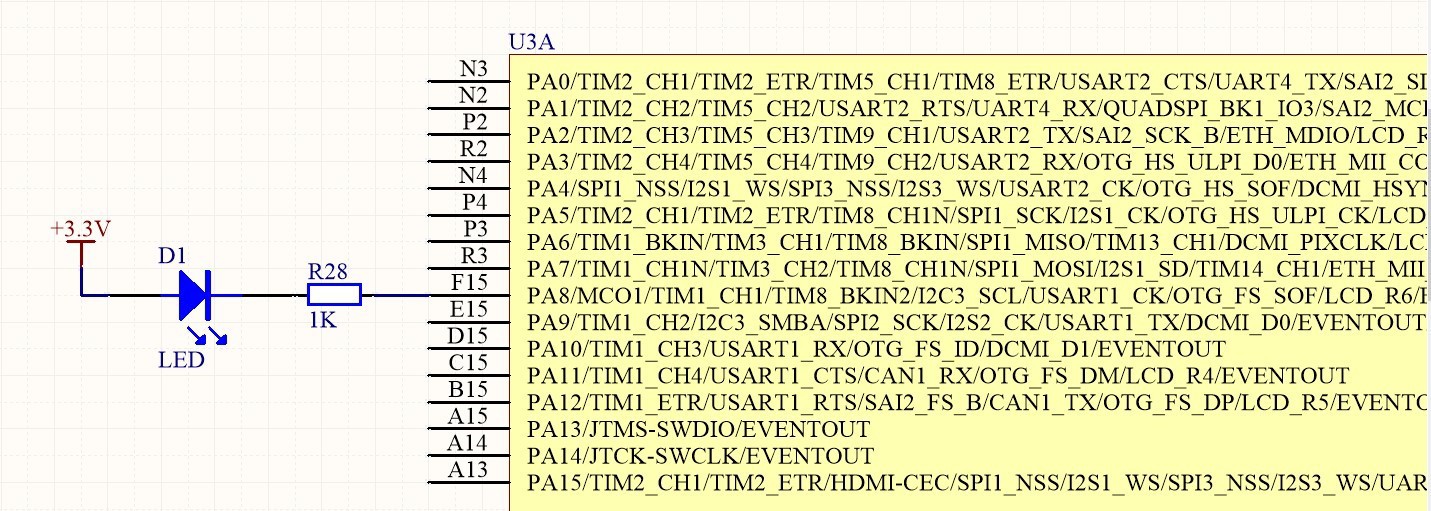


图 1: 实验电路

如图中所示，U3A 单元为 STM32F746 芯片，发光二极管 D1 一端与 VCC 相连，另一端经过 1K 电阻与 Cortex-M7 的 PA8（GPIOA8）相连，将 PA8 配置为输出 I/O 口，

即可通过控制其高低电平状态进而控制发光二极管 D1 的的亮与灭。当 PA8 输出低电平时，电路导通，发光二极管 D1 点亮；反之，发光二极管 D1 熄灭。实验例程中，每次对 PA8 的输出状态取反后调用延时函数，使发光二极管保持亮或灭，循环往复即可实现发光二极管的闪烁。通过修改延时长度，可以改变发光二极管的闪烁频率。

* + 1. 实验方案及调试过程

基础实验：首先复现例程代码，得到指导书期望现象并记录实验例程如下：

/∗∗

主程序： 系统上电初始化后对GPIO中的PA8进行配置， 将其配置为输出端口并控制LED 闪烁。

∗∗/

#i n c l u d e ” main . h”

#i n c l u d e ” system\_init . h”

/∗ Private v a r i a b l e s ∗/ uint16 \_t delay = 10 0 ;

void System\_Init ( void ) ; void GPIO\_Config ( void ) ;

/∗ main ∗/

i n t main ( void )

{

System\_Init ( ) ; GPIO\_Config ( ) ;

p r i n t f ( ”\n\ rExample f i n i s h e d \n\ r ” ) ;

/∗ Toggle IOs in an i n f i n i t e loop ∗/ while ( 1 )

{

HAL\_GPIO\_TogglePin(GPIOA, GPIO\_PIN\_8) ;

/∗ I n s e r t delay ∗/ HAL\_Delay( delay ) ;

}

}

1

3

5

7

9

11

13

15

17

19

21

23

25

27

29

调试过程如下：

首先定义了私有变量 delay，用于控制延时时间：uint16\_t delay = 100;

然后调用 System\_Init() 函数进行系统初始化，调用 GPIO\_Config() 函数对 GPIO 端口进行初始化。

void System\_Init ( void )

{

/∗ Enable the CPU Cache ∗/ CPU\_CACHE\_Enable( ) ;

/∗ STM32F7xx HAL l i b r a r y i n i t i a l i z a t i o n ∗/ HAL\_Init ( ) ;

/∗ Configure the system c l o c k to 216 MHz ∗/ SystemClock\_Config ( ) ;

/∗ Configure UART ∗/ UART\_Config ( ) ;

p r i n t f ( ”\n\ rSystem i n i t i a l i z e s u c c e s s \n\ r ” ) ;

}

1

3

5

7

9

11

13

15

GPIO\_Config() 函数的定义如下：

2

void GPIO\_Config ( void )

{

/∗ Enable each GPIO Clock ( to be able to program the c o n f i g u r a t i o n r e g i s t e r s )

∗/

HAL\_RCC\_GPIOA\_CLK\_ENABLE( ) ;

/∗ Configure IOs in output push−p u l l mode to d r i v e e x te r n a l LED ∗/ GPIO\_InitStruct . Mode = GPIO\_MODE\_OUTPUT\_PP;

GPIO\_InitStruct . Pull = GPIO\_PULLUP;

GPIO\_InitStruct . Speed = GPIO\_SPEED\_HIGH;

// 推挽输出

// 上拉

// 高速

GPIO\_InitStruct . Pin = GPIO\_PIN\_8 ;

HAL\_GPIO\_Init(GPIOA, &GPIO\_InitStruct ) ;

// 引脚为PA8

}

4

6

8

10

12

然后进入死循环状态，通过调用 HAL\_GPIO\_TogglePin() 函数对 GPIOA 的 8 号引脚进行翻转，即实现 LED 灯的闪烁效果。具体的实现如下：

2

void HAL\_GPIO\_TogglePin( GPIO\_TypeDef∗ GPIOx, uint16 \_t GPIO\_Pin)

{

4

/∗ Check the parameters ∗/

assert\_param ( IS\_GPIO\_PIN( GPIO\_Pin) ) ;

GPIOx−>ODR ^= GPIO\_Pin ;

}

6

利用异或操作实现对 GPIO 引脚状态的翻转。

每次翻转后调用 HAL\_Delay() 函数进行延时，延时时间为 delay，以控制 LED 灯的闪烁频率。具体实现如下：

weak void HAL\_Delay( IO uint32 \_t Delay )

{

uint32 \_t t i c k s t a r t = 0 ;

t i c k s t a r t = HAL\_GetTick ( ) ;

while ( ( HAL\_GetTick ( ) − t i c k s t a r t ) < Delay )

{

}

}

1

3

5

7

HAL\_Delay() 函数通过调用 HAL\_GetTick() 函数获取当前的系统滴答计数值，然后通过循环判断当前的系统滴答计数值与 tickstart 的差值是否小于 Delay，若小于则继续循环，否则退出循环，从而实现延时效果。

进阶实验：请自行搜索摩尔斯密码表，通过控制 D1 的亮灭间隔，实现自己姓氏拼音的电码实现。

摩尔斯电码（Morse code）也被称作摩斯密码，是一种时通时断的信号代码，通过不同的排列顺序来表达不同的英文字母、数字和标点符号。这里我们通过控制 D1 的亮灭间隔， 实现自己姓氏拼音的电码。

具体的算法思路如下：

* + - 1. 姓氏的拼音用英文打出，姓氏拼音有多个英文字母组成，英文字母之间用空格隔开，每 一个姓之间用逗号隔开；
      2. 定义短信号和长信号的时间分别为 200ms 和 600ms，空格时长 1000ms，逗号时长2000ms，正常信号之间的间隔为 200ms；

第一步中，我们小组有三位成员，分别姓：

* WU=”.– ..-”
* GUO=”–. ..- —”
* ZOU=”–.. — ..-”

因此我们定义字符数组：

char lname [ ] = ”.−− .. −, −−. .. − −−−,−−.. −−− .. − ” ;

然后定义短信号和长信号的时间分别为 200ms 和 600ms，空格时长 1000ms，逗号时长2000ms，正常信号之间的间隔为 200ms。

|  |  |  |
| --- | --- | --- |
| 1 | uint16 \_t uint16 \_t | delay = 10 0 ;  short\_delay =200; |
| 3 | uint16 \_t | long\_delay =600; |
|  | uint16 \_t | blank\_delay =1000; |
| 5 | uint16 \_t | gap\_delay =2000; |

根据电码序列的不同字母 (’.’/’-’/’ ’/’,’) 分别使用短延迟、长延迟、空格延迟、逗号延迟的时长进行延迟，从而控制灯亮灭的间隔。

在死循环中，对整个字符数组进行遍历，根据字符的不同，进行不同的延迟操作，从而 实现对 LED 灯的控制。

while ( 1 ) {

// turn on the LED l i g h t HAL\_GPIO\_TogglePin(GPIOA, GPIO\_PIN\_8) ;

i f ( lname [ i%31 ] == ’ . ’) { HAL\_Delay( short\_delay ) ;

} e l s e i f ( lname [ i%31]==’ −’){ HAL\_Delay( long\_delay ) ;

}

// turan o f f the LED l i g h t HAL\_GPIO\_TogglePin(GPIOA, GPIO\_PIN\_8) ;

// c o n t r o l delay o f o f f s t a t e i f ( lname [ ( i +1)%31]==’ ’ ) {

HAL\_Delay( blank\_delay ) ; i +=2;

}

e l s e i f ( lname [ ( i +1)%31 ] == ’ , ’) { HAL\_Delay( gap\_delay ) ;

i +=2;

} e l s e {

HAL\_Delay( delay ) ;

1

3

5

7

9

11

13

15

17

19

21

i ++;

}

}

23

## 实验步骤

* + 1. 准备实验环境

使用 ULINK2 USB-JTAG 仿真器连接 ARM Cortex-M7 实验板与 PC，实验板一侧接P1 接口。使用串口线，连接实验板上的串口 J3 和 PC 机的串口。接口 P1 和 J3 的位置如图所示。

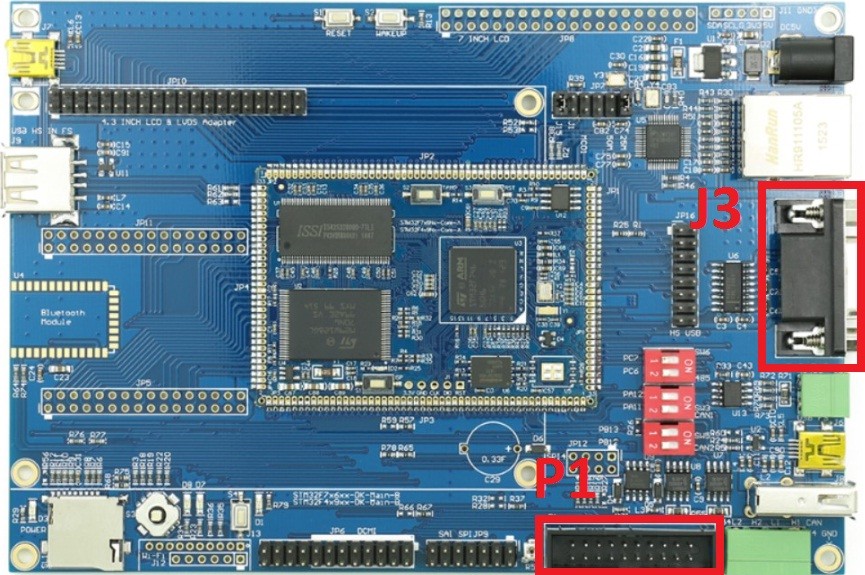


图 2: 接口 P1 和 J3 在实验板中的位置图

* + 1. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200 、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

* + 1. 打开实验例程

拷贝实验平台附带程序“02\_GPIO”，使用 *µ*Vision IDE for ARM 通过 ULINK2 USB- JTAG 仿真器连接实验板，打开工程文件，编译链接工程，点击 MDK 的 Project 菜单，选

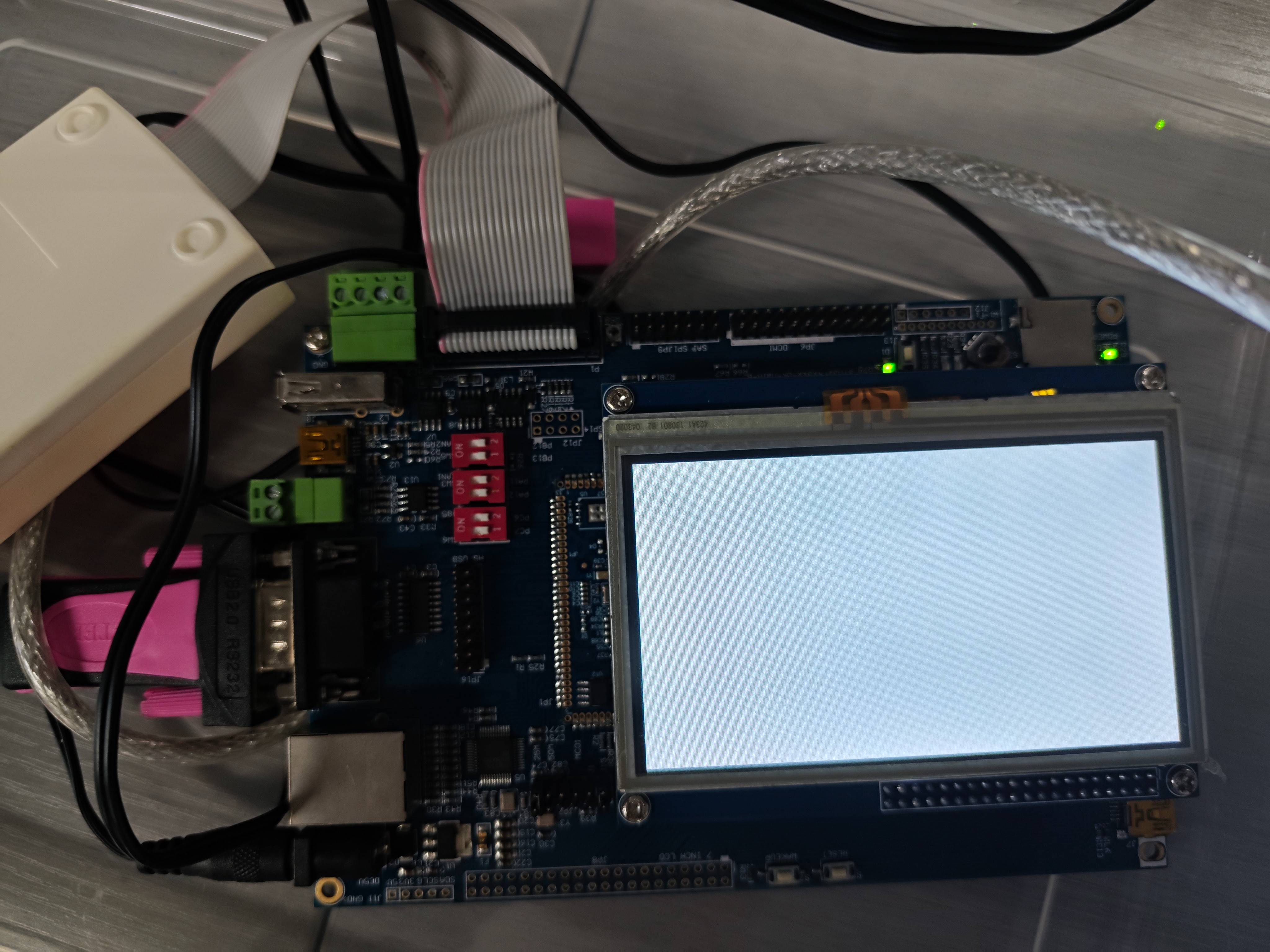
择 Rebuild all target files 进行编译，编译成功后，点击 Debug 菜单，选择 Start/Stop Debug Session 项或点击工具栏中的图标，下载工程生成的.axf 文件到目标板的 RAM 中调试运行。

* + 1. 观察实验结果

结合实验内容和相关资料，使用一些调试命令，观察程序运行。注意观察发光二极管的 亮灭情况，观察到的现象与前面实验内容中的相符，则说明实验程序通过配置 GPIO 引脚的输出状态实现了对发光二极管的控制。修改代码，实现自己姓氏的摩尔斯电码。

## 实验结果

开发板中的灯泡呈现不断闪烁的状态，如图所示。同时，实现自己姓氏拼音的电码操作 将改变灯泡的亮灭间隔，状态表示同理。



(a) 状态一：灯泡亮 (b) 状态二：灯泡灭

图 3: 实验结果



## 实验结论

本实验中，我们通过 HAL\_GPIO\_TogglePin() 可以切换 LED 灯状态，利用异或（XOR） 操作取反 GPIOx 对应位的值，达到反转指定引脚状态的效果；同时，我们可以通过控制HAL\_GPIO\_TogglePin() 的间隔来控制亮灯和灭灯的时间，对不同的字符设置不同的间隔时间来实现自己姓氏拼音的电码。实验源代码见附录。

## 实验小结

通过本实验，我更加熟练地掌握了 vscode+Keil MDK 开发环境的使用以及在线调试方法，对 STM32F746NG 芯片 GPIO 端口寄存器的配置有了更加直观的认识，并对 C 语言嵌入式编程有了初步的认识，收获很大。

# 实验 **2**《中断实验》

实验学时：2 每组人数：3

实验类别：2 （1：基础性 2：综合性 3：设计性 4：研究性） 实验要求：1 （1：必修 2：选修 3：其它）

实验类别：3 （1：基础 2：专业基础 3：专业 4：其它）

## 实验目的

* 掌握外部中断的处理流程；
* 掌握 Cortex-M7 处理器的中断方式和中断处理过程；
* 通过实验学习 Cortex-M7 处理器的中断响应流程；
* 通过实验掌握 Cortex-M7 处理器中断处理的软件编程方法；
* 通过实验掌握 Cortex-M7 处理器中断响应过程中相关寄存器的使用方法。

## 实验内容

编写程序，对指定 GPIO 端口进行初始化，完成外部中断相关寄存器的配置，使用 ARM Cortex-M7 实验平台的按键 S3 产生外部中断，在中断响应过程中对 LED 进行控制，并采用不同的中断设置方法实现多种中断触发方式。实验过程中观察上升沿触发选择寄存器

（EXTI\_RTSR）和下降沿触发选择寄存器（EXTI\_FTSR）的值对中断触发条件的影响，学 习 Cortex-M7 外部中断线的设置方法和初始化，以及外部中断的触发方式和响应过程。

## 实验方法

* + 1. 实验原理
       1. STM32F746NG 的外部中断和事件控制器（EXTI）

STM32F746NG 具有多达 24 个用于产生中断/事件请求的边沿检测器（输入线）。每根输入线都可以单独进行配置，以选择类型（中断或事件）和响应的触发事件（上升沿 触发、下降沿触发或边沿触发），每根输入线还可以单独屏蔽。挂起寄存器用于保持中断请求。EXTI 控制器的主要特性如下：

* + - * + 每个中断/事件线上都具有独立的触发和屏蔽；
        + 每个中断线具有专用的状态位；
        + 支持多大 24 个软件事件/中断请求；
        + 检测脉冲宽度低于 APB2 时钟宽度的外部信号。

要产生中断，必须先配置好并使能中断线。根据需要的边沿检测设置 2 个触发寄存器， 同时在中断屏蔽寄存器的相应位写“1”使能中断请求。当外部中断线上出现选定信号沿时，便会产生中断请求，对应的挂起位也会置 1。在挂起寄存器的对应位写“1”，将清除该中断请求。要产生事件，必须先配置好并使能事件线。根据需要的边沿检测设置 2 个触发寄存器，同时在事件屏蔽寄存器的相应位写“1”使能事件请求。当事件线上出现选定信号沿时，便会产生事件脉冲，对应的挂起位会置 1。通过在软件中对中 断/事件寄存器写“1”，也可以产生中断/事件请求。要将一根输入线配置为中断源，需执行以下步骤：

1. 配置相应的屏蔽位（EXTI\_IMR）；
2. 配置中断线的触发选择位（EXTI\_RTSR 和 EXTI\_FTSR）；
3. 配置对应到外部中断控制器（EXTI）的 NVIC 中断通道的使能和屏蔽位，使得

24 个中断线中的请求可以被正确的响应。要将一根输入线配置为事件源，需执行以下步骤：

1. 配置相应的屏蔽位（EXTI\_EMR）；
2. 配置事件线的触发选择位（EXTI\_RTSR 和 EXTI\_FTSR）；
   * + 1. 外部中断/事件线映射及控制器框图

如图所示，多达 168 个 GPIO 通过图中方式连接到 16 个外部中断/事件线。另外 8 根

EXTI 线连接方式如下：

* + - * + EXTI16 连接到 PVD 输出；
        + EXTI17 连接到 RTC 闹钟事件；
        + EXTI18 连接到 USB OTG FS 唤醒事件；
        + EXTI19 连接到以太网唤醒事件；
        + EXTI20 连接到 USB OTG HS 唤醒事件
        + EXTI21 连接到 RTC 侵入和时间戳事件
        + EXTI22 连接到 RTC 唤醒事件；
        + EXTI23 连接到 LPTIM1 异步事件。

EXTI 控制器框图如图所示



图 4: 外部中断/事件 GPIO 映射



图 5: EXTI 控制器框图

* + - 1. EXTI 寄存器

1. 中断屏蔽寄存器（EXTI\_IMR） 偏移地址：0x00

复位值：0x0000 0000

中断屏蔽寄存器如图所示



图 6: 中断屏蔽寄存器

位 31:24 保留，必须保持复位值。MRx：x 线上的中断屏蔽0：屏蔽来自 x 线的中断请求

1：开放来自 x 线的中断请求

1. 事件屏蔽寄存器（EXTI\_EMR） 偏移地址：0x04

复位值：0x0000 0000

事件屏蔽寄存器如图所示。

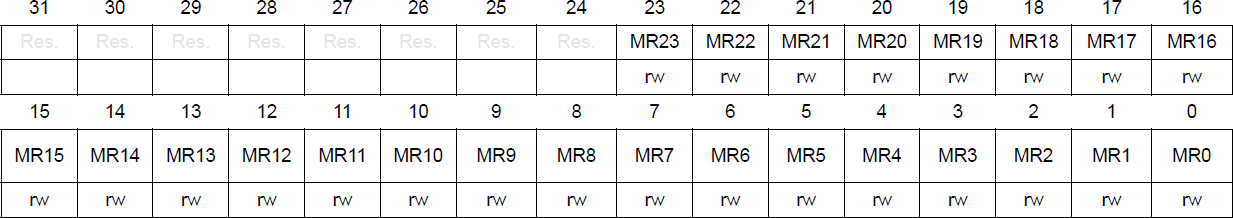


图 7: 事件屏蔽寄存器

位 31:24 保留，必须保持复位值。MRx：x 线上的事件屏蔽0：屏蔽来自 x 线的事件请求

1：开放来自 x 线的事件请求

1. 上升沿触发选择寄存器（EXTI\_RTSR） 偏移地址：0x08

复位值：0x0000 0000

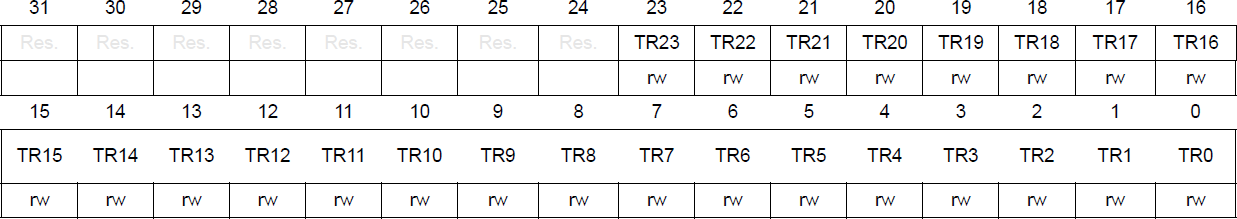


图 8: 上升沿触发选择寄存器

上升沿触发选择寄存器如图所示。位 31:24 保留，必须保持复位值。TRx：x 线的上升沿触发事件配置位

0：禁止输入线上升沿触发（事件和中断）

1：开放输入线上升沿触发（事件和中断）

注：外部唤醒线配置为边沿触发时，在这些线上不能出现毛刺信号。

如果在向 EXTI\_RTSR 寄存器写入值的同时外部中断线上产生上升沿，挂起位将被置位。

在同一中断线上，可以同时设置上升沿和下降沿触发，即任一边沿都可触发中断。

1. 下降沿触发选择寄存器（EXTI\_FTSR） 偏移地址：0x0C

复位值：0x0000 0000

下降沿触发选择寄存器如图所示。

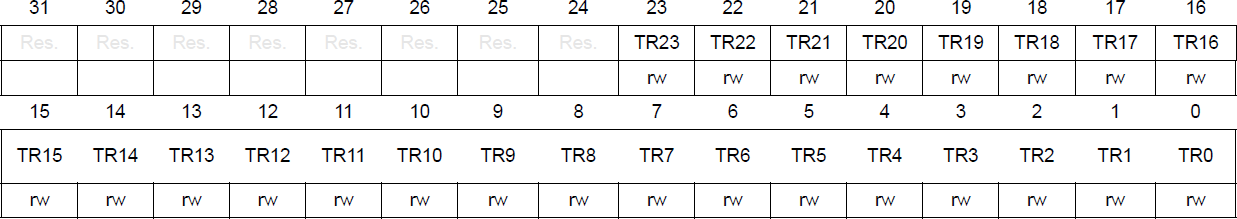


图 9: 下降沿触发选择寄存器

位 31:24 保留，必须保持复位值。TRx：x 线的下降沿触发事件配置位0：禁止输入线下降沿触发（事件和中断）

1：开放输入线下降沿触发（事件和中断）

注：外部唤醒线配置为边沿触发时，在这些线上不能出现毛刺信号。

如果在向 EXTI\_FTSR 寄存器写入值的同时外部中断线上产生下降沿，挂起位将被置位。

在同一中断线上，可以同时设置上升沿和下降沿触发，即任一边沿都可触发中断。

1. 软件中断事件寄存器（EXTI\_SWIER） 偏移地址：0x10

复位值：0x0000 0000

软件中断事件寄存器如图所示。

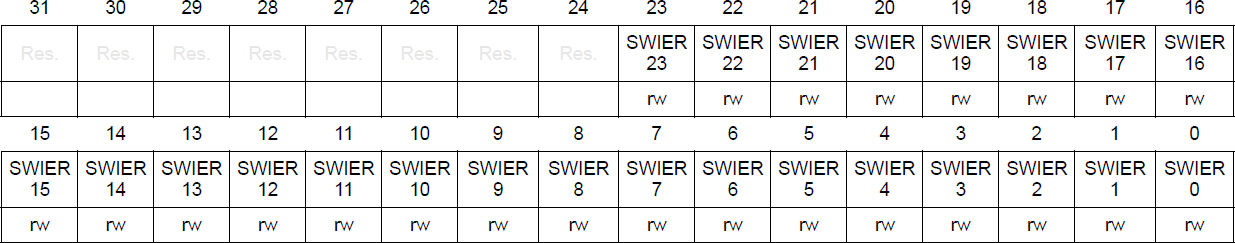


图 10: 软件中断事件寄存器

位 31:24 保留，必须保持复位值。

SWIERx：x 线的软件中断

当 SWIERx 位设置为“0”时，将“1”写入该位会将 EXTI\_PR 寄存器中相应挂起位置 1。如果在 EXTI\_IMR 寄存器中允许在 x 线上产生该中断，则产生中断请求。通过清除 EXTI\_PR 的对应位（写入“1”），可以清除该位为“0”。

1. 挂起寄存器（EXTI\_PR） 偏移地址：0x14

复位值：未定义

挂起寄存器如图所示。

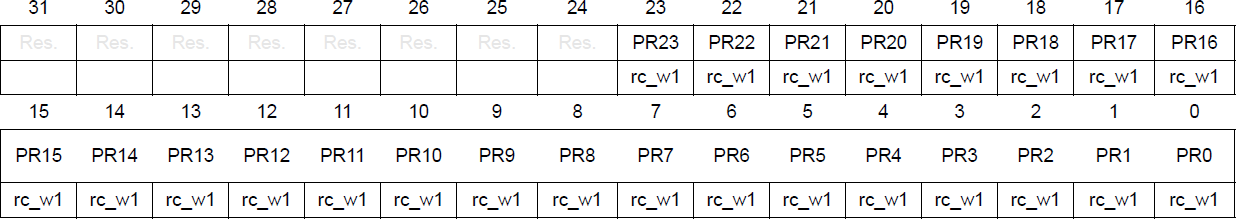


图 11: 挂起寄存器

位 31:24 保留，必须保持复位值。PRx：x 线的挂起0：未发生触发请求

1：发生了选择的触发请求

注：当在外部中断线上发生了选择的边沿事件，该位被置 1，将此位编程为“1” 可清除此位。

* + - 1. EXTI 寄存器边界地址

EXTI 寄存器边界地为 0x4001 3C00–0x4001 3FFF。

* + - 1. 实验电路如图所示

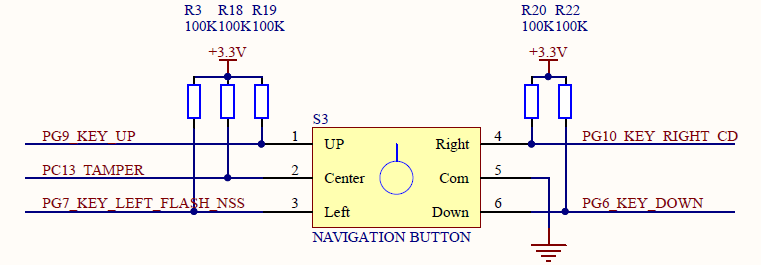


图 12: 实验电路

如图中所示，STM32F746 芯片的 PC13 外接上拉电路，串联开关 S3 的 Center（对应五向导航键 S3 的确定功能）后接地。由于 I/O 口外接上拉电路，所以在对 I/O 口进行初始化时可设置为浮空输入。开关 S3 断开时，PC13 输入高电平；反之，PC13 输入低电平。所以，当按下开关 S3 时，PC13 输入由高变低，产生一个下降沿；当释放开关 S3 时，PC13 输入由低变高，产生一个上升沿。根据外部中断触发条件设置，当满足所需的边沿条件时，触发中断，MCU 响应中断点亮/熄灭发光二极管 D1。

* + 1. 实验方案及调试过程

基础实验：本实验的实验例程（STM32F746\_Experiment\_v1.1/03\_EXTI） 实验例程如下：

/∗∗

主程序： 系统上电初始化后对LED1进行初始化， 配置PC13 作为外部中断源并开启中断， 产生中断后点亮/ 熄灭LED。

∗∗/

#i n c l u d e ” main . h”

#i n c l u d e ” system\_init . h”

void System\_Init ( void ) ;

void EXTI15\_10\_IRQHandler\_Config ( void ) ;

/∗ main ∗/

i n t main ( void )

{

System\_Init ( ) ;

1

3

5

7

9

11

13

15

/∗ I n i t i a l i z e LED1 mounted on board ∗/ BSP\_LED\_Init(LED1) ;

/∗ Configure EXTI15\_10 ( connected to PC. 1 3 pin ) in i n t e r r u p t mode ∗/ EXTI15\_10\_IRQHandler\_Config ( ) ;

while ( 1 )

{

}

}

/∗ EXTI l i n e d e te c t i o n c a l l b a c k s , GPIO\_Pin : S p e c i f i e s the pins connected EXTI l i n e ∗/

void HAL\_GPIO\_EXTI\_Callback( uint16 \_t GPIO\_Pin) // 此处定义了 HAL\_GPIO\_EXTI\_Callback函数， 原使用 weak 定义的函数被忽略

{

i f ( GPIO\_Pin == GPIO\_PIN\_13)

{

/∗ Toggle LED1 ∗/ BSP\_LED\_Toggle(LED1) ;

p r i n t f ( ”\n\rLED1 switched \n\ r ” ) ;

}

}

17

19

21

23

25

27

29

31

33

35

调试过程分析如下：

首先调用 System\_Init() 函数进行系统初始化，内部函数内容与实验一相同，不再赘述。然后调用 BSP\_LED\_Init() 函数对 LED1 进行初始化，函数初始化板载 LED1，使能

LED1 使其工作调用

2

void BSP\_LED\_Init( Led\_TypeDef Led )

{

GPIO\_InitTypeDef g p i o \_i n i t\_s tr u c tu r e ; GPIO\_TypeDef∗ gpio\_led ;

// Actualy only one LED switch ( Led )

{

case LED1 :

/∗ Enable the GPIO\_LED c l o c k ∗/ LED1\_GPIO\_CLK\_ENABLE( ) ;

gpio\_led = LED1\_GPIO\_PORT;

4

6

8

10

12

break ;

case LED2 :

/∗ Enable the GPIO\_LED c l o c k ∗/ LED2\_GPIO\_CLK\_ENABLE( ) ;

gpio\_led = LED2\_GPIO\_PORT; break ;

case LED3 :

/∗ Enable the GPIO\_LED c l o c k ∗/ LED3\_GPIO\_CLK\_ENABLE( ) ;

gpio\_led = LED3\_GPIO\_PORT; break ;

case LED4 :

/∗ Enable the GPIO\_LED c l o c k ∗/ LED4\_GPIO\_CLK\_ENABLE( ) ;

gpio\_led = LED4\_GPIO\_PORT; break ;

d e f a u l t : break ;

}

/∗ Configure the GPIO\_LED pin ∗/

g p i o \_i n i t\_s tr u c tu r e . Pin = GPIO\_PIN[ Led ] ;

g p i o \_i n i t\_s tr u c tu r e . Mode = GPIO\_MODE\_OUTPUT\_PP; g p i o \_i n i t\_s tr u c tu r e . Pull = GPIO\_PULLUP;

g p i o \_i n i t\_s tr u c tu r e . Speed = GPIO\_SPEED\_HIGH; HAL\_GPIO\_Init( gpio\_led , &g p i o \_i n i t\_s tr u c tu r e ) ;

/∗ By d e f a u l t , turn o f f LED ∗/

HAL\_GPIO\_WritePin( gpio\_led , GPIO\_PIN[ Led ] , GPIO\_PIN\_SET) ;

}

14

16

18

20

22

24

26

28

30

32

34

36

38

40

接着通过 EXTI15\_10\_IRQHandler\_Config() 函数配置 PC13 作为外部中断源并开启中断。

void EXTI15\_10\_IRQHandler\_Config ( void )

{

GPIO\_InitTypeDef GPIO\_InitStructure ;

/∗ Enable GPIOC c l o c k ∗/

HAL\_RCC\_GPIOC\_CLK\_ENABLE( ) ;

/∗ Configure PC. 1 3 pin as input f l o a t i n g ∗/

GPIO\_InitStructure . Mode = GPIO\_MODE\_IT\_FALLING;

触发

// 中断触发方式： 下降沿

1

3

5

7

9

11

GPIO\_InitStructure . Pull = GPIO\_NOPULL; GPIO\_InitStructure . Pin = GPIO\_PIN\_13 ;

HAL\_GPIO\_Init(GPIOC, &GPIO\_InitStructure ) ;

// GPIO内部无上拉或下拉

// 初始化PC13

/∗ Enable and s e t EXTI l i n e s 15 to 10 In te r r u p t to the l o we s t p r i o r i t y ∗/ HAL\_NVIC\_SetPriority ( EXTI15\_10\_IRQn , 2 , 0 ) ; // 设置中断优先级

HAL\_NVIC\_EnableIRQ( EXTI15\_10\_IRQn) ; // 开中断

}

13

15

17

然后进入死循环，定义了下面的函数 HAL\_GPIO\_EXTI\_Callback()，当发生中断时，自动调用该函数。该函数能够翻转 LED1 灯状态，从而完成改变 LED 灯亮灭状态的功能

void HAL\_GPIO\_EXTI\_Callback( uint16 \_t GPIO\_Pin)

// 此处定义了

HAL\_GPIO\_EXTI\_Callback函数， 原使用 weak 定义的函数被忽略

{

i f ( GPIO\_Pin == GPIO\_PIN\_13)

{

/∗ Toggle LED1 ∗/ BSP\_LED\_Toggle(LED1) ;

p r i n t f ( ”\n\rLED1 switched \n\ r ” ) ;

}

}

1

3

5

7

9

进阶实验：按下按键触发中断 LED 灯高频闪烁，提起按键触发中断 LED 灯熄灭具体的算法思路如下：

在 config.c 文件中配置中断触发方式为上升沿和下降沿（GPIO\_MODE\_IT\_RISING\_ FALLING），并设置一个全局变量 down 用来记录按键状态。HAL\_GPIO\_ReadPin () 可以读取按键状态，按下按键时为低电压，则为 0，抬起按键时为高电压，即为 1，实现的函数如下：

2

void HAL\_GPIO\_EXTI\_Callback( uint16 \_t GPIO\_Pin) { i f ( GPIO\_Pin==GPIO\_PIN\_13) {

down=HAL\_GPIO\_ReadPin(GPIOC, GPIO\_Pin) ;

}

}

4

由此我们可以判断按键是否按键，并通过读取全局变量 down 值控制 LED 灯状态，实现按下按键时 LED 灯闪烁，松开按键时 LED 灯关闭。

1

i n t main ( void )

{

System\_Init ( ) ;

/∗ I n i t i a l i z e LED1 mounted on board ∗/ BSP\_LED\_Init(LED1) ;

/∗ Configure EXTI15\_10 ( connected to PC. 1 3 pin ) in i n t e r r u p t mode ∗/ EXTI15\_10\_IRQHandler\_Config ( ) ;

while ( 1 )

{

i f ( down==0){ BSP\_LED\_Toggle(LED1) ; HAL\_Delay( short\_delay ) ; BSP\_LED\_Toggle(LED1) ; HAL\_Delay( short\_delay ) ;

}

}

}

3

5

7

9

11

13

15

17

19

## 实验步骤

* + 1. 准备实验环境

使用 ULINK2 USB-JTAG 仿真器连接 ARM Cortex-M7 实验板与 PC，实验板一侧接右下方的 P1 接口。使用串口线，连接实验板右侧的串口 J3 和 PC 机的串口。

* + 1. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200 、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

* + 1. 打开实验例程

拷贝实验平台附带程序“03\_EXTI”，使用 *µ*Vision IDE for ARM 通过 ULINK2 USB- JTAG 仿真器连接实验板，打开工程文件，编译链接工程，根据本实验指导书中 2.3.2 小节中“编译配置”部分对工程进行配置（工程默认已经配置正确），点击 MDK 的 Project 菜单，选择 Rebuild all target files 进行编译，编译成功后，点击 Debug 菜单，选择 Start/Stop

Debug Session 项或点击工具栏中的图标，下载工程生成的.axf 文件到目标板的 RAM 中调试运行。

* + 1. 观察实验结果

结合实验内容和相关资料，使用一些调试命令，观察程序运行。注意观察按键 S3 按下和释放时发光二极管 D1 的亮灭情况，观察到的现象与前面实验内容中的相符，则说明实验程序通过将 GPIO 配置为 EXTI 的中断源，通过按键开关触发外部中断，MCU 响应中断并点亮/熄灭发光二极管。修改部分代码，实现按下高频闪烁，松开熄灭的功能。

## 实验结果

按下按键 LED 灯高频闪烁



## 实验结论

本实验中，我们首先观察到程序通过将 GPIO 配置为 EXTI 的中断源，通过按键开关触发外部中断，MCU 响应中断并点亮/熄灭发光二极管。我们配置中断触发方式为上升沿和下降沿，并通过 HAL\_GPIO\_ReadPin() 读取按键状态，并根据按键状态控制 LED 灯状态， 从而完成了实验要求的功能。实验源代码见附录。

## 实验小结

通过本实验，我更加熟练地掌握了 Cortex-M7 处理器的中断方式和中断处理过程，并熟悉了 Cortex-M7 处理器中断处理的软件编程方法和其中断响应过程中相关寄存器的使用方法，同时对嵌入式系统中电路与程序的关系有了进一步的认识，收获很大。

1. 实验 **3**《**Uart+** 定时器实验》

实验学时：2 每组人数：3

实验类别：2 （1：基础性 2：综合性 3：设计性 4：研究性） 实验要求：1 （1：必修 2：选修 3：其它）

实验类别：3 （1：基础 2：专业基础 3：专业 4：其它）

## 实验目的

* + 1. 掌握 UART 串口的工作原理；
    2. 掌握 Cortex-M7 的 UART 串口配置方法；
    3. 通过实验掌握 printf() 函数重定向的方法；
    4. 通过实验掌握 Cortex-M7 串口通信及调试方法；
    5. 掌握 Cortex-M7 定时器的工作原理；
    6. 掌握 Cortex-M7 定时器的配置及初始化方法；
    7. 通过实验掌握定时器中断的设置和使用方法；
    8. 通过实验掌握定时器中断的响应流程。

## 实验内容

* + 1. 引导输入功能：编译运行，超级终端可以通过 Uart 接收到 ARM 板子回传的信息，要求用户输入一串数字（浮点数或整数）
    2. 实时回显功能：ARM 接收到用户的每个字符，需要将本字符在回传到超级终端，使得用户可以看到自己输入的字符，按下回车键，ARM 程序开始处理用户输入字符串，如果不是数字字符（小数或整数）则回显提示错误；如果是数字字符开始执行后面功能；
    3. 数字识别功能：将用户输入的字符串，识别成具体的数字，这个数字表示 LED 灯闪烁的时间间隔。
    4. 灯光闪烁频率（单位 Hz）控制：以上面收到的数据去设定定时器，控制 LED 灯的闪烁间隔。示例：用户输入 1，1Hz 表示那么 LED 灯应该 1S 完成一次亮灭循环，也就是亮 0.5s 灭 0.5s，如此循环往复

## 实验方法

* + 1. 实验原理
       1. UART 通信协议

UART（Universal Asynchronous Receiver and Transmitter）通用异步收发器是异步串行通信接口的总称，支持 RS-232、RS-422、RS-485 等接口标准规范和总线标准规范。

UART 作为异步串口通信协议的一种，工作原理是将传输数据的每个字符一位接一位的传输其中每一位 (Bit) 的意义如下：

* + - * + 起始位：先发出一个逻辑“0”的信号，表示传输字符的开始。
        + 数据位：紧接着起始位之后。数据位的个数可以是 4、5、6、7、8 等，构成一个字符。从最低位开始传送，靠时钟定位
        + 奇偶校验位：数据位加上这一位后，使得“1”的位数应为偶数 (偶校验) 或奇数

(奇校验)，以此来校验数据传送的正确性。

* + - * + 停止位：它是一个字符数据的结束标志。可以是 1 位、1.5 位、2 位的高电平。由于数据是在传输线上定时的，并且每一个设备有其自己的时钟，很可能在通信中 两台设备间出现了小小的不同步。因此停止位不仅仅是表示传输的结束，并且提 供计算机校正时钟同步的机会。适用于停止位的位数越多，不同时钟同步的容忍 程度越大，但是数据传输率同时也越慢。
        + 空闲位：处于逻辑“1”状态，表示当前线路上没有数据传送

UART 协议传输时序如图所示

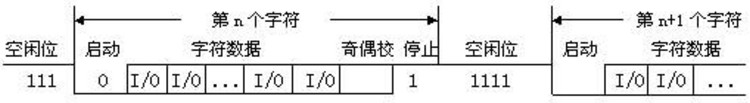


图 13: UART 协议传输时序

发送数据过程：空闲状态，线路处于高电位；当收到发送数据指令后，拉低线路一个 数据位的时间 T，接着数据按低位到高位依次发送，数据发送完毕后，接着发送奇偶校验位和停止位（停止位为高电位），一帧数据发送结束。

发送数据过程：空闲状态，线路处于高电位；当收到发送数据指令后，拉低线路一个 数据位的时间 T，接着数据按低位到高位依次发送，数据发送完毕后，接着发送奇偶校验位和停止位（停止位为高电位），一帧数据发送结束。

波特率是衡量数据传输速率的指标，表示每秒传送数据的字符数，单位为 Baud。UART 的接收和发送是按照相同的波特率进行收发的。波特率发生器产生的时钟频率不是波 特率时钟频率，而是波特率时钟频率的 16 倍，目的是为在接收时进行精确地采样，以提取出异步的串行数据。根据给定的晶振时钟和要求的波特率，可以算出波特率分频 计数值。

* + - 1. printf() 函数重定向

标准库函数的默认输出设备是显示器，因此必须对重新定义 printf() 函数中与串口输出相关的函数，才能通过调用 printf() 函数向串口发送数据。

使用以下代码可以完成 printf() 函数的重定义：

2

#i f d e f GNUC

/∗ With GCC/RAISONANCE, small p r i n t f ( option LD Linker −>L i b r a r i e s −>Sm all p r i n t f

s e t to ’ Yes ’ ) c a l l s \_\_io\_putchar ( ) ∗/

#d e f i n e PUTCHAR\_PROTOTYPE i n t \_\_io\_putchar ( i n t ch ) #e l s e

#d e f i n e PUTCHAR\_PROTOTYPE i n t fp utc ( i n t ch , FILE ∗ f ) #e n d i f /∗ GNUC ∗/

/∗ Retargets the C l i b r a r y p r i n t f f u n c t i o n to the UART ∗/ PUTCHAR\_PROTOTYPE

{

/∗ Place your implementation o f fp utc here ∗/

/∗ e . g . w r i te a c h a r a c te r to the COM1 and Loop u n t i l the end o f tr a n s m i s s i o n ∗/

HAL\_UART\_Transmit(&Uart\_Handle , ( uint8 \_t ∗)&ch , 1 , 0xFFFF) ;

retur n ch ;

}

4

6

8

10

12

14

16

由于 GNU 编译器中 printf() 调用的是 putchar() 函数执行底层输出任务，所以前半段代码使用宏定义可以兼容不同编译器。

后半段即重定义过程，使用 HAL\_UART\_Transmit() 函数完成 fputc() 或 putchar() 函数的底层输出任务。HAL\_UART\_Transmit() 函数的具体定义可以在实验例程“03\_U ART”中查看

* + - 1. 定时器

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 定时器类型 | 定时器 | 位数 | 计数模式 | 分频系数 |
| 高级控制定时器 | TIM1,TIM8 | 16 | 递增，递减，增/减 | 1~65536 之间 |
| 通用定时器 | TIM2,TIM5 | 32 | 递增，递减，增/减 | 1~65536 之间 |
| TIM3,TIM4 | 16 | 递增，递减，增/减 | 1~65536 之间 |
| TIM9 | 16 | 递增 | 1~65536 之间 |
| TIM11,TIM10 | 16 | 递增 | 1~65536 之间 |
| TIM12 | 16 | 递增 | 1~65536 之间 |
| TIM13,TIM14 | 16 | 递增 | 1~65536 之间 |
| 基本定时器 | TIM6,TIM7 | 16 | 递增 | 1~65536 之间 |

Cortex-M7 具有 2 个高级控制定时器、10 个通用定时器、2 个基本定时器和 2 个看门狗定时器。特性对比如表所示

下面以实验例程中所用的通用定时器为例，介绍 Cortex-M7 中定时器的主要功能和用法

* + - 1. 通用定时器

通用定时器包含一个 16 位或 32 位自动重载计数器，该计数器由可编程预分频器驱动。它们可用于多种用途，包括测量输入信号的脉冲宽度（输入捕获）或生成输出波形（输出比较和 PWM）。

使用定时器预分频器和 RCC 时钟控制器预分频器，可将脉冲宽度和波形周期从几微秒调制到几毫秒

这些定时器彼此完全独立，不共享任何资源。通用 TIMx 定时器具有以下特性：

* + - * + 16 位（TIM3 和 TIM4）或 32 位（TIM2 和 TIM5）递增、递减和递增/递减自动重载计数器。
        + 16 位可编程预分频器，用于对计数器时钟频率进行分频（可在运行时修改），分频系数介于 1 到 65535 之间。
        + 多达 4 个独立通道，可用于

输入捕获

输出比较

PWM 生成（边沿和中心对齐模式）

单脉冲模式输出

* + - * + 使用外部信号控制定时器且可实现多个定时器互连的同步电路。
        + 发生如下事件时生成中断/DMA 请求：

更新：计数器上溢/下溢、计数器初始化（通过软件或内部/外部触发）

触发事件（计数器启动、停止、初始化或通过内部/外部触发计数）

输入捕获

输出比较

* + - * + 支持定位用增量（正交）编码器和霍尔传感器电路
        + 触发输入作为外部时钟或者逐周期电流管理

本实验例程中使用定时器的时基单元功能，即定时功能

可编程定时器的主要模块由一个 16 位/32 位计数器及其相关的自动重装寄存器组成。计数器可递增计数、递减计数或同时递增和递减计数。计数器的时钟可通过预分频器进行分频。计数器、自动重载寄存器和预分频器寄存器可通过软件进行读写。即使在计数器运行时也可执行读写操作。

时基单元包括：

* + - * + 计数器寄存器 (TIMx\_CNT)
        + 预分频器寄存器 (TIMx\_PSC)
        + 自动重载寄存器 (TIMx\_ARR)

本实验中采用递增计数模式，计数器从 0 计数到自动重载值（TIMx\_ARR 寄存器的内容），然后重新从 0 开始计数并生成计数器上溢事件，即定时器中断。

若采用递减计数模式，计数器从自动重载值（TIMx\_ARR 寄存器的内容）开始递减计数到 0，然后重新从自动重载值开始计数并生成计数器下溢事件。

计数器由预分频器输出 CK\_CNT 提供时钟，仅当 TIMx\_CR1 寄存器中的计数器启动位 (CEN) 置 1 时，才会启动计数器

预分频器说明：

预分频器可对计数器时钟频率进行分频，分频系数介于 1 和 65536 之间。该预分频器基于 16 位/32 位寄存器（TIMx\_PSC 寄存器）所控制的 16 位计数器。由于该控制寄存器具有缓冲功能，因此预分频器可实现实时更改。而新的预分频比将在下一更新事 件发生时被采用。定时器频率计算方法及定时器中断原理定时器 3 频率计算方法如下：

TIMER3 时钟频率（TIM3CLK）为 APB1 总线时钟频率（PCLK1）的 2 倍，APB1 总线时钟频率为系统时钟的 1/4，

即

*TIM* 3*CLK* = 2 *∗ PCLK*1

*PCLK*1 = *SystemCoreClock*/4

所以 *TIM* 3*CLK* = *SystemCoreClock*/2

本实验例程中需要将 TIMER3 时钟频率配置为 10KHz，由于采用递增计数模式，预分频计算方式如下：

*Prescaler* = (*TIM* 3*CLK*/*TIM* 3*counterclock*) *−* 1 *Prescaler* = ((*SystemCoreClock*/2)/(10*KHz*) *−* 1

所以在对定时器 3 初始化时，需要按照以上方法计算预分频数值

* + 1. 实验方案及调试过程

本实验要求通过超级终端输入闪烁频率（单位 Hz），LED 灯按设定频率闪烁，可连续输入数字改变频率。

具体的算法思路如下：

从 UART 串口中接收数据，调用 printf() 函数将接收到的数据回传到超级终端，将接收到的字符判断是否合规，不合规的字符通过 printf() 函数回传错误信息，实现“引导输入”、“实时回显”功能，合规的字符转换为数字，并存储到栈中

2

HAL\_UART\_Receive(&UartHandle , ( uint8 \_t ∗)&uRxBuffer , 1 ,TIMEOUT) ; p r i n t f ( ”%c ” , uRxBuffer ) ;

i f ( u RxBuffer != ’ \ r ’&&uRxBuffer != ’ \n ’ ) {

i f ( ( uRxBuffer>=’ 0 ’&&uRxBuffer<=’ 9 ’ ) | | u RxBuffer==’ . ’ ) { s tack [ s t o r e ]=u RxBuffer ;

s t o r e ++;

} e l s e {

p r i n t f ( ”\n\ r INPUT ERROR! \n\ r ” ) ; break ;

}

}

4

6

8

10

接收到的合法小数字符数组通过 sscanf 函数解析成浮点数，根据公式计算值计算 ARR 值

HAL\_UART\_Receive(&UartHandle , ( uint8 \_t ∗)&uRxBuffer , 1 ,TIMEOUT) ; p r i n t f ( ”%c ” , uRxBuffer ) ;

i f ( u RxBuffer != ’ \ r ’&&uRxBuffer != ’ \n ’ ) {

i f ( ( uRxBuffer>=’ 0 ’&&uRxBuffer<=’ 9 ’ ) | | u RxBuffer==’ . ’ ) { s tack [ s t o r e ]=u RxBuffer ;

1

3

5

7

s t o r e ++;

} e l s e {

p r i n t f ( ”\n\ r INPUT ERROR! \n\ r ” ) ; break ;

}

}

9

11

使用定时器 Tim3 配置中断，控制灯光的闪烁频率，完成程序功能。

Timer\_Config ( ARRValue) ;

HAL\_TIM\_Base\_Start\_IT(&TimHandle ) ;

1

3

## 实验步骤

* + 1. 准备实验环境

使用 ULINK2 USB-JTAG 仿真器连接 ARM Cortex-M7 实验板与 PC，实验板一侧接右下方的 P1 接口。使用串口线，连接实验板右侧的串口 J3 和 PC 机的串口。

* + 1. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200 、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

* + 1. 打开实验例程

拷贝实验平台附带程序“05\_TIMER”，使用 *µ*Vision IDE for ARM 通过 ULINK2 USB- JTAG 仿真器连接实验板，打开工程文件，编译链接工程，对工程进行配置，点击 MDK 的Project 菜单，选择 Rebuild all target files 进行编译，编译成功后，点击 Debug 菜单，选择Start/Stop Debug Session 项或点击工具栏中的图标，下载工程生成的.axf 文件到目标板的RAM 中调试运行。

* + 1. 观察实验结果

观察 PC 中超级终端显示信息，根据提示使用键盘输入数据，MCU 接收到数据后对TIMER3 的 TIM3\_ARR 寄存器进行相应配置并完成定时器中断的初始化。初始化结束后开启 TIMER3 开始计时，产生定时器中断后对 LED 执行跳变操作，即观察到 LED 以高或低频率闪烁。

## 实验结果

超级终端显示如下，成功回显了输入的字符

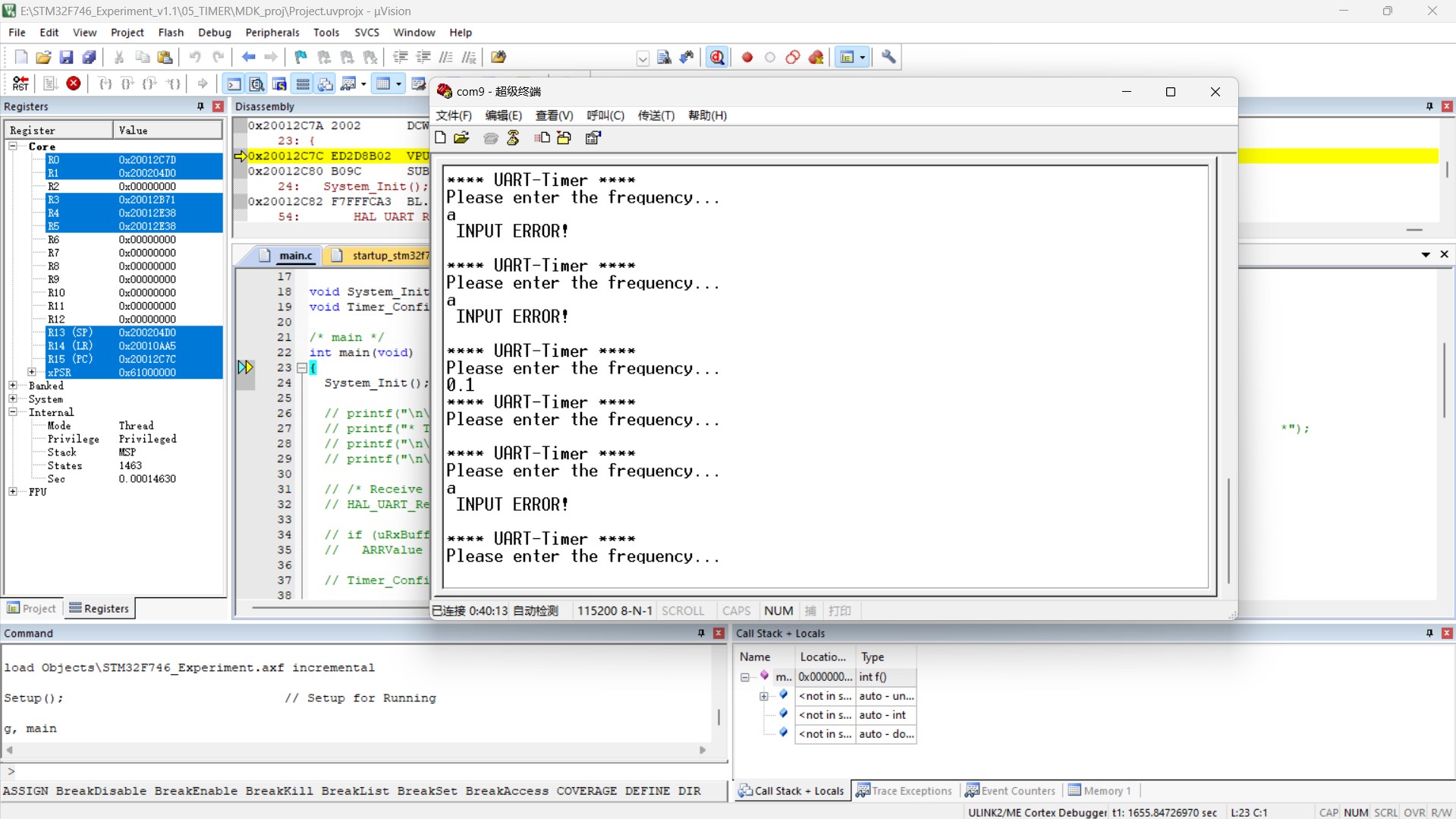


图 14: 超级终端显示以下是输入 1HZ 的结果，LED 灯每秒闪烁一次



## 实验结论

本实验中，我们在 while 循环中通过 Uart 通信协议逐字符接收消息并回显在超级终端上，并使用 sscanf() 函数将数字字符转化为 double 型变量使得程序可以接收小数；通过设置定时器让 LED 灯持续闪烁，当下一次输入新值后会重新设置 LED 灯闪烁频率，达到程序目的。实验源代码见附录。

## 实验小结

通过本实验，我掌握了 Cortex-M7 的 UART 串口配置方法，串口通信及调试方法，并学习了定时器中断的设置和使用方法，熟悉了定时器中断的响应流程，同时进一步熟悉了 C 语言嵌入式编程，感受到嵌入式系统应用的魅力所在，收获很大。

1. 实验 **4**《*µ***C/OS-III** 实验——信号量》

实验学时：2 每组人数：3

实验类别：2 （1：基础性 2：综合性 3：设计性 4：研究性） 实验要求：1 （1：必修 2：选修 3：其它）

实验类别：3 （1：基础 2：专业基础 3：专业 4：其它）

## 实验目的

* + 1. 掌握 *µ*C/OS-III 操作系统中信号量的创建方法；
    2. 通过实验掌握 *µ*C/OS-III 系统中使用信号量进行任务间通信的方法。

## 实验内容

用 *µ*C/OS-III 系统的信号量机制实现按下按键 LED 灯闪烁，松开按键停止闪烁。在实验过程中学习 *µ*C/OS-III 操作系统中信号量的创建和使用方法，在实验中注意观察 LED 的闪烁以及串口数据的变化情况，掌握使用信号量进行任务间通信的方法。

## 实验方法

* + 1. 实验原理
       1. 信号量

信号量是在多任务环境下使用的一种设施，是可以用来保证两个或多个关键代码段不被并发调用。在进入一个关键代码段之前，线程必须获取一个信号量；一旦该关键代码段完成了，那么该线程必须释放信号量。其它想进入该关键代码段的线程必须等待直到第一个线程释放信号量。为了完成这个过程，需要创建一个信号量 Semaphore，然后将获取 Semaphore 以及释放 Semaphore 的代码分别放置在每个关键代码段的首末端。

抽象的来讲，信号量的特性如下：信号量是一个非负整数，所有使用它的线程/进程都会将该整数减一，当该整数值为零时，所有试图使用它的线程都将处于等待状态。在 信号量上我们定义两种操作：Wait（等待）和 Release（释放）。当一个线程调用 Wait 操作时，它要么得到资源然后将信号量减一，要么一直等下去（指放入阻塞队列），直到信号量大于等于一时。Release（释放）实际上是在信号量上执行加操作，该操作之 所以叫做“释放”是因为释放了由信号量守护的资源。

在 *µ*C/OS-III 中，使用 OSSemCreate 函数可以创建一个信号量，其原型如下：

1

void OSSemCreate (OS\_SEM ∗p\_sem , CPU\_CHAR ∗p\_name ,

OS\_SEM\_CTR cnt , OS\_ERR ∗p\_err )

3

使用 OSSemPend 函数可以获取一个信号量，若获取到信号量，则任务继续执行，否则该挂起该任务，直到该任务获取到信号量或超时。OSSemPend 函数的原型如下：

OS\_SEM\_CTR OSSemPend (OS\_SEM ∗p\_sem ,

1

|  |  |  |
| --- | --- | --- |
|  | OS\_TICK | timeout , |
| 3 | OS\_OPT | opt , |
|  | CPU\_TS | ∗p\_ts , |
| 5 | OS\_ERR | ∗p\_err ) |

使用 OSSemPost 函数可以释放一个信号量，其原型如下：

OS\_SEM\_CTR OSSemPost (OS\_SEM ∗p\_sem , OS\_OPT opt ,

OS\_ERR ∗p\_err )

1

3

* + - 1. 实验例程说明
         1. 系统初始化（HAL 库、内存管理模块和数学算法模块的初始化）；
         2. 创建启动任务 AppTaskStart；
         3. 在启动任务中对 BSP 和 CPU 进行初始化；
         4. 在启动任务中创建通信过程中使用的信号量等对象；
         5. 在启动任务中创建两个应用任务 Task\_A 和 Task\_B，启动任务进入死循环；
         6. Task\_A 检测 S3 是否被按下并释放，当 S3 释放时向 Task\_B 发送信号量；
         7. Task\_B 等待 Task\_A 发送来的信号量，当获取到信号量时点亮/熄灭 LED1 并向串口发送数据。
    1. 实验方案及调试过程

本实验要求通过 *µ*C/OS-III 操作系统中的信号量机制完成按下按键 LED 灯高频闪烁， 提起按键 LED 灯熄灭的功能。

具体的算法思路如下：

首先创建等待键盘按键检测的任务和信号量机制相关的操作系统对象，定义任务名为

Task A 和 Task B

s t a t i c void AppTaskCreate ( void )

2 {

OS\_ERR os\_err ;

4

/∗ −−−−−−−−−− 创建应用任务 −−−−−−−−− ∗/

6 OSTaskCreate(&Task\_ATCB,

” Kernel Objects Task A” ,

8 Task\_A,

0 ,

10 APP\_CFG\_TASK\_OBJ\_PRIO, &Task\_AStk [ 0 ] ,

12 Task\_AStk [APP\_CFG\_TASK\_OBJ\_STK\_SIZE / 10u ] , APP\_CFG\_TASK\_OBJ\_STK\_SIZE,

14 0u ,

0u ,

16 0 ,

(OS\_OPT\_TASK\_STK\_CHK | OS\_OPT\_TASK\_STK\_CLR) ,

18 &os\_err ) ;

20 OSTaskCreate(&Task\_BTCB,

” Kernel Objects Task B” ,

22 Task\_B ,

0 ,

24 APP\_CFG\_TASK\_OBJ\_PRIO, &Task\_BStk [ 0 ] ,

26 Task\_BStk [APP\_CFG\_TASK\_OBJ\_STK\_SIZE / 10u ] , APP\_CFG\_TASK\_OBJ\_STK\_SIZE,

28 0u ,

0u ,

30 0 ,

(OS\_OPT\_TASK\_STK\_CHK | OS\_OPT\_TASK\_STK\_CLR) ,

32 &os\_err ) ;

}

34

s t a t i c void AppObjCreate ( void )

36 {

OS\_ERR os\_err ;

38

#i f (OS\_CFG\_SEM\_EN > 0u )

40

OSSemCreate(&wait\_key\_sem ,

”Key Detection ” , 0u ,

&os\_err ) ;

OSSemCreate(&wait\_key\_sem2 , ”Key Detection ” , 0 u,& os\_err ) ; #e n d i f

}

42

44

46

当检测到按键被按下时，任务 Task\_A 会发送信号量 wait\_key\_sem：

2

void Task\_A( void ∗p\_arg )

{

OS\_ERR e r r ;

unsigned char key\_press = 0 ;

while (DEF\_TRUE)

{

// OSTimeDly ( 30 , OS\_OPT\_TIME\_DLY, &e r r ) ;

// i f ( BSP\_PB\_GetState(BUTTON\_TAMPER) == RESET)

// key\_press = 1 ;

// i f ( BSP\_PB\_GetState(BUTTON\_TAMPER) == SET)

// {

// i f ( key\_press == 1 )

// {

// key\_press = 0 ;

// OSSemPost(&wait\_key\_sem , OS\_OPT\_POST\_1, &e r r ) ;

// }

// }

OSTimeDly ( 60 , OS\_OPT\_TIME\_DLY, &e r r ) ;

i f ( BSP\_PB\_GetState(BUTTON\_TAMPER) == RESET) {

OSSemPost(&wait\_key\_sem , OS\_OPT\_POST\_1, &e r r ) ;

}

}

}

4

6

8

10

12

14

16

18

20

22

24

当任务 Task\_B 接收到信号量信息后，会通过 BSP\_LED\_Toggle(LED1) 函数控制 LED 灯的高速闪烁；且发送信号量任务的系统时钟节拍与接收信号量任务的系统时钟节拍保持一 致，从而使得当按键抬起时 LED 灯不再闪烁。具体代码如下：

2

void Task\_B( void ∗p\_arg )

{

4

OS\_ERR e r r ; CPU\_TS t s ;

while (DEF\_TRUE)

{

OSTimeDly ( 30 , OS\_OPT\_TIME\_DLY, &e r r ) ; OSSemPend(&wait\_key\_sem ,

0 , OS\_OPT\_PEND\_BLOCKING,

&ts , &e r r ) ;

BSP\_LED\_Toggle(LED1) ; // Toggle a f t e r key r e l e a s e OSTimeDly ( 30 , OS\_OPT\_TIME\_DLY, &e r r ) ; BSP\_LED\_Toggle(LED1) ;

APP\_TRACE\_DBG( ( ” Get a Semaphore from Task A\n\ r ” ) ) ;

}

}

6

8

10

12

14

16

18

## 实验步骤

* + 1. 准备实验环境

使用 ULINK2 USB-JTAG 仿真器连接 ARM Cortex-M7 实验板与 PC，实验板一侧接右下方的 P1 接口。使用串口线，连接实验板右侧的串口 J3 和 PC 机的串口。

* + 1. 串口接收设置

在 PC 机上运行 windows 自带的超级终端串口通信程序（波特率 115200 、1 位停止位、无校验位、无硬件流控制）；或者使用其它串口通信程序。

* + 1. 打开实验例程

拷贝实验平台附带程序“14\_uCOS-III\_Semaphore”，使用 *µ*Vision IDE for ARM 通过 ULINK2 USB-JTAG 仿真器连接实验板，打开工程文件，编译链接工程，根据本实验指导书中 2.3.2 小节中“编译配置”部分对工程进行配置（工程默认已经配置正确），点击 MDK 的Project 菜单，选择 Rebuild all target files 进行编译，编译成功后，点击 Debug 菜单，选择Start/Stop Debug Session 项或点击工具栏中的图标，下载工程生成的.axf 文件到目标板的RAM 中调试运行。

* + 1. 观察实验结果

我们观察到按下按键LED 灯高频闪烁，提起按键 LED 灯熄灭，说明我们通过 *µ*C/OS-III

操作系统中的信号量机制完成了实验需要的功能

## 实验结果

按下按键 LED 灯高频闪烁，提起按键 LED 灯熄灭



## 实验结论

本实验中，首先创建等待键盘按键检测的应用进程，通过 *µ*C/OS-III 操作系统中的信号量机制，实现当接收到按键按下信号后通过 BSP\_LED\_Toggle(LED1) 函数控制 LED 灯的高速闪烁，并保持发送信号量任务的系统时钟节拍与接收信号量任务的系统时钟节拍一致， 从而当提起按键后 LED 灯停止闪烁。实验源代码见附录。

## 实验小结

通过本实验，我掌握了 *µ*C/OS-III 系统中信号量的创建方法和使用信号量进行任务间通信的方法，对 C 语言嵌入式编程有了更多的认识，直观地了解到信号量机制在嵌入式系统中的广泛用途与应用，收获很大。

# 附录

* 1. 实验 **1** 源代码

1 /∗∗

主程序： 系统上电初始化后对GPIO中的PA8进行配置， 将其配置为输出端口并控制LED 闪烁。

3 ∗∗/

#i n c l u d e ” main . h”

5 #i n c l u d e ” system\_init . h”

7 /∗ Private v a r i a b l e s ∗/ uint16 \_t delay = 10 0 ;

9

uint16 \_t short\_delay =200;

11 uint16 \_t long\_delay =600;

uint16 \_t blank\_delay =1000;

13 uint16 \_t gap\_delay =2000;

15 void System\_Init ( void ) ; void GPIO\_Config ( void ) ;

17

/∗ main ∗/

19 i n t main ( void )

{

21 System\_Init ( ) ;

23 GPIO\_Config ( ) ;

25 p r i n t f ( ”\n\ rExample f i n i s h e d \n\ r ” ) ;

27 // wjy−WU=”.−− .. − ”

// gx−GUO=”−−. .. − −−−”

29 // zzy−ZOU=” −−.. −−− .. − ”

char lname [ ] = ”.−− .. −, −−. .. − −−−,−−.. −−− .. − ” ;

31 i n t i =0;

33 /∗ Toggle IOs in an i n f i n i t e loop ∗/

// while ( 1 )

35 // {

// HAL\_GPIO\_TogglePin(GPIOA, GPIO\_PIN\_8) ;

37 // /∗ I n s e r t delay ∗/

// HAL\_Delay( delay ) ;

39

// }

// 打印姓氏

// 方法：

// 形势的拼音用英文字母打出， 每个字有多个英文字母组成， 英文字母之间用空格隔开， 每一个姓之间用逗号隔开

// 设置长信号为600ms ， 短信号为200ms ， 空格为1000ms ， 姓之间间隔为2000ms ， 信号间隔100ms

while ( 1 ) {

// turn on the LED l i g h t HAL\_GPIO\_TogglePin(GPIOA, GPIO\_PIN\_8) ;

i f ( lname [ i%31 ] == ’ . ’) { HAL\_Delay( short\_delay ) ;

} e l s e i f ( lname [ i%31]==’ −’){ HAL\_Delay( long\_delay ) ;

}

// turan o f f the LED l i g h t HAL\_GPIO\_TogglePin(GPIOA, GPIO\_PIN\_8) ;

// c o n t r o l delay o f o f f s t a t e i f ( lname [ ( i +1)%31]==’ ’ ) {

HAL\_Delay( blank\_delay ) ; i +=2;

}

e l s e i f ( lname [ ( i +1)%31 ] == ’ , ’) { HAL\_Delay( gap\_delay ) ;

i +=2;

} e l s e {

HAL\_Delay( delay ) ; i ++;

}

}

}

41

43

45

47

49

51

53

55

57

59

61

63

65

67

* 1. 实验 **2** 源代码

/∗∗

2 主程序： 系统上电初始化后对LED1进行初始化， 配置PC13 作为外部中断源并开启中断， 产生中断后点亮/ 熄灭LED。

∗∗/

4 #i n c l u d e ” main . h”

#i n c l u d e ” system\_init . h”

6

void System\_Init ( void ) ;

8 void EXTI15\_10\_IRQHandler\_Config ( void ) ;

10 uint16 \_t short\_delay =100; uint16 \_t down=1;

12

/∗ main ∗/

14 i n t main ( void )

{

16 System\_Init ( ) ;

18 /∗ I n i t i a l i z e LED1 mounted on board ∗/ BSP\_LED\_Init(LED1) ;

20

/∗ Configure EXTI15\_10 ( connected to PC. 1 3 pin ) in i n t e r r u p t mode ∗/

22 EXTI15\_10\_IRQHandler\_Config ( ) ;

24 while ( 1 )

{

26 i f ( down==0){ BSP\_LED\_Toggle(LED1) ;

28 HAL\_Delay( short\_delay ) ; BSP\_LED\_Toggle(LED1) ;

30 HAL\_Delay( short\_delay ) ;

}

32 }

}

34

/∗ EXTI l i n e d e te c t i o n c a l l b a c k s , GPIO\_Pin : S p e c i f i e s the pins connected EXTI l i n e ∗/

36 // void HAL\_GPIO\_EXTI\_Callback( uint16 \_t GPIO\_Pin) // 此处定义了

HAL\_GPIO\_EXTI\_Callback函数， 原使用 weak 定义的函数被忽略

// {

38 // i f ( GPIO\_Pin == GPIO\_PIN\_13)

40

// {

// /∗ Toggle LED1 ∗/

// BSP\_LED\_Toggle(LED1) ;

//

//

// }

p r i n t f ( ”\n\rLED1 switched \n\ r ” ) ;

}

void HAL\_GPIO\_EXTI\_Callback( uint16 \_t GPIO\_Pin) { i f ( GPIO\_Pin==GPIO\_PIN\_13) {

down=HAL\_GPIO\_ReadPin(GPIOC, GPIO\_Pin) ;

}

}

42

44

46

48

50

* 1. 实验 **3** 源代码

|  |  |  |
| --- | --- | --- |
| 1  3  5  7  9  11 | /∗∗  主程序： 系统初始化后使用串口通信对Timer3 的自动重载寄存器（ARR） 初值进行选择， 通过配置预分频，  将Timer3 的时钟频率设置为10KHz， 在配置好定时器后开启定时器中断。 Timer3 的计数器寄存器从0 开始以10KHz的频率递增，  当其值大于ARR寄 存器中的数值时， 产生上溢， 即定时器中断。  注意： 此例程中， 每产生一次定时器中断，LED发生一次跳变， 即模式1 中LED闪烁频率为0 . 5 Hz， 模式2 中LED闪烁频率为5Hz  ∗∗/  #i n c l u d e ” main . h”  #i n c l u d e ” system\_init . h”  #d e f i n e TIMEOUT 10000 // 10 seconds  /∗ Pr e s c a l e r d e c l a r a t i o n ∗/ | |
| 13  15 | uint8 \_t uRxBuffer = ’ 1 ’ ; // 默认选择模式” 1 . 10000”  uint16 \_t ARRValue = 10000 − 1 ;// 默认ARR寄 存器值为 10000 − 1 extern UART\_HandleTypeDef UartHandle ; |  |
|  | extern TIM\_HandleTypeDef TimHandle ; |  |
| 17 |  |  |
| 19 | void System\_Init ( void ) ;  void Timer\_Config ( uint16 \_t ARRValue) ; |  |
| 21  23 | /∗ main ∗/  i n t main ( void )  { |  |
|  | System\_Init ( ) ; |  |
| 25 |  |  |
| 27 | // p r i n t f ( ”\n\ r ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗\ n\ r ” ) ;  // p r i n t f ( ” ∗ Timer counter f requency i s 10KHz ∗\ n\ r ∗ Upcounting mode |  |
| 29 | ∗\ n\ r ∗ I n i t i a l value i s 0 ∗” ) ;  // p r i n t f ( ”\n\ r ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗\ n\ r ” ) ;  // p r i n t f ( ”\n\ r S e l e c t the value o f ARR r e g i s t e r : \ n\ r 1 . 10000 ( d e f a u l t ) | 2 . |
|  | 1000\ n\ r ” ) ; |  |
| 31 | // /∗ Receive data from UART ∗/  // HAL\_UART\_Receive(&UartHandle , ( uint8 \_t ∗)&uRxBuffer , 1 , TIMEOUT) ; |  |
| 33 |  |  |
| 35 | // i f ( u RxBuffer == ’ 2 ’ )  // ARRValue = 1000 − 1 ; |  |

37 // Timer\_Config ( ARRValue) ; // 使用Tim3 在这里面配置中断

|  |  |  |
| --- | --- | --- |
| 39 | //  // | /∗ Sta r t the TIM Base g e n e r a t i o n in i n t e r r u p t mode ∗/  HAL\_TIM\_Base\_Start\_IT(&TimHandle ) ; |
| 41 |  |  |
|  | // | p r i n t f ( ”\n\ rExample f i n i s h e d \n\ r ” ) ; |
| 43 |  |  |
| 45 | while ( 1 )  { | |
| 47 | uint8 \_t s tack [ 1 0 0 ] ;  i n t s t o r e =0; | |

49 double num=0;

p r i n t f ( ”\n\ r ∗∗∗∗ UART−Timer ∗∗∗∗\ n\ r Pl e a s e e n te r the f requency . . . \ n\ r ” ) ;

51

// r e c e i v e data from uart

53 while ( 1 ) {

HAL\_UART\_Receive(&UartHandle , ( uint8 \_t ∗)&uRxBuffer , 1 ,TIMEOUT) ;

55 p r i n t f ( ”%c ” , uRxBuffer ) ;

i f ( u RxBuffer != ’ \ r ’&&uRxBuffer != ’ \n ’ ) {

57 i f ( ( uRxBuffer>=’ 0 ’&&uRxBuffer<=’ 9 ’ ) | | u RxBuffer==’ . ’ ) { s tack [ s t o r e ]=u RxBuffer ;

59 s t o r e ++;

} e l s e {

61 p r i n t f ( ”\n\ r INPUT ERROR! \n\ r ” ) ; break ;

63 }

}

65

67

69

71 }

i f ( u RxBuffer==’ \ r ’ | | uRxBuffer==’ \n ’ | | s to r e >10){ s s c a n f ( ( const char ∗) stack , ”% l f ”,&num) ; ARRValue=5000/num−1;

break ;

}

73 Timer\_Config ( ARRValue) ;

75

77 }

HAL\_TIM\_Base\_Start\_IT(&TimHandle ) ;

}

79 void HAL\_TIM\_PeriodElapsedCallback ( TIM\_HandleTypeDef ∗htim )

81

{

BSP\_LED\_Toggle(LED1) ;

}

* 1. 实验 **4** 源代码

1 /∗∗

主程序： 上电后进行系统硬件初始化以及uC/OS−I I I 初始化， 完成初始化工作后创建启动

3 任务： AppTaskStart ， 在该任务中完成板级支持包BSP和CPU模块的初始化， 并创建

两个应用任务Task\_A和Task\_B， 其中Task\_A检测按键S3 是否被按下并释放， 当S3 释放时

5 产生信号量， Task\_B等待Task\_A发送来的信号量， 当获取到信号量时点亮/ 熄灭LED1并向串口发送数据。

7 ∗∗/

#i n c l u d e <stdarg . h>

9 #i n c l u d e <s t d i o . h> #i n c l u d e <math . h>

11 #i n c l u d e <stm32f7xx\_hal . h> #i n c l u d e ” stm32756g\_eval . h”

13

#i n c l u d e <cpu . h>

15 #i n c l u d e <lib\_math . h> #i n c l u d e <lib\_mem . h>

17 #i n c l u d e <os . h>

#i n c l u d e <os\_app\_hooks . h>

19

#i n c l u d e <app\_cfg . h>

21 #i n c l u d e <bsp . h>

23 #i f (APP\_CFG\_SERIAL\_EN == DEF\_ENABLED) #i n c l u d e <app\_serial . h>

25 #e n d i f

/∗

27 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

29 ∗

∗ LOCAL DEFINES

31 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

33 ∗

∗/

35

#d e f i n e APP\_TASK\_EQ\_0\_ITERATION\_NBR 16u

37 #d e f i n e APP\_TASK\_EQ\_1\_ITERATION\_NBR 18u

39

/∗

41 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

43 ∗

∗ LOCAL GLOBAL VARIABLES

45 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

47 ∗

∗/

49 /∗ −−−−−−−−−−−−−−− APPLICATION GLOBALS −−−−−−−−−−−−−−−− ∗/

s t a t i c OS\_TCB AppTaskStartTCB ;

51 s t a t i c CPU\_STK AppTaskStartStk [APP\_CFG\_TASK\_START\_STK\_SIZE] ;

|  |  |  |  |
| --- | --- | --- | --- |
| 53 | /∗ −−−−−−−−−−−−−−− SEMAPHORE TASK TEST −−−−−−−−−−−−−−− ∗/ | | |
| 55 | s t a t i c s t a t i c | OS\_TCB CPU\_STK | Task\_ATCB;  Task\_AStk [APP\_CFG\_TASK\_OBJ\_STK\_SIZE] ; |
| 57 | s t a t i c s t a t i c | OS\_TCB CPU\_STK | Task\_BTCB;  Task\_BStk [APP\_CFG\_TASK\_OBJ\_STK\_SIZE] ; |
| 59 |  |  |  |

#i f (OS\_CFG\_SEM\_EN > 0u )

61 s t a t i c OS\_SEM wait\_key\_sem ; s t a t i c OS\_SEM wait\_key\_sem2 ;

63 #e n d i f

65 /∗

∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

67 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

∗

69 ∗ FUNCTION PROTOTYPES

∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

71 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

∗

73 ∗/

75 s t a t i c void AppTaskStart ( void ∗p\_arg ) ; s t a t i c void AppTaskCreate ( void ) ;

77 s t a t i c void AppObjCreate ( void ) ;

79 s t a t i c void Task\_A( void ∗p\_arg ) ; s t a t i c void Task\_B( void ∗p\_arg ) ;

81

/∗

83 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

85 ∗

∗ main ( )

87 ∗

∗ De s c r i p t i o n : This i s the standard entry point f o r C code . I t i s assumed that your code w i l l c a l l

89 ∗ main ( ) once you have performed a l l n e c e s s a r y i n i t i a l i z a t i o n .

∗

91 ∗ Arguments : none

∗

93 ∗ Returns : none

∗

95 ∗ Notes : 1 ) STM32F7xx HAL l i b r a r y i n i t i a l i z a t i o n :

∗ a ) Configures the Flash p r e f e tc h , i n t r u c t i o n and data caches .

97 ∗ b ) Configures the Sy s t i c k to g e n e r ate an i n t e r r u p t . However , the f u n c t i o n ,

∗ HAL\_InitTick ( ) , that i n i t i a l i z e s the Sy s t i c k has been o ve r w r i tte n s i n c e Micrium ’ s

99 ∗ RTOS has i t s own Sy s t i c k i n i t i a l i z a t i o n and i t i s recommended to i n i t i a l i z e the

∗ Sy s t i c k a f t e r m u l t i ta s k i n g has s ta r te d .

101 ∗

∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

103 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

∗

105 ∗/

107 i n t main ( void )

{

109 OS\_ERR e r r ;

#i f (CPU\_CFG\_NAME\_EN == DEF\_ENABLED)

111 CPU\_ERR cpu\_err ; #e n d i f

113

HAL\_Init ( ) ; /∗ 初始化HAL库

∗/

115

Mem\_Init ( ) ; /∗ 初始化内存管

理模块 ∗/

117 Math\_Init ( ) ; /∗ 初始化数学算

法模块（ 生成随机数种子） ∗/

119 #i f (CPU\_CFG\_NAME\_EN == DEF\_ENABLED)

CPU\_NameSet( (CPU\_CHAR ∗) ”STM32F746xx” ,

121 (CPU\_ERR ∗)&cpu\_err ) ; #e n d i f

123

BSP\_IntDisAll ( ) ; /∗ 关闭中断

∗/

125

OSInit(& e r r ) ; /∗ 初始化uC/OS−

I I I ∗/

127 App\_OS\_SetAllHooks ( ) ;

129 OSTaskCreate(&AppTaskStartTCB , /∗ 创建启动进程

∗/

”App Task Sta r t ” ,

131 AppTaskStart ,

0u ,

133 APP\_CFG\_TASK\_START\_PRIO, &AppTaskStartStk [ 0 u ] ,

135 AppTaskStartStk [APP\_CFG\_TASK\_START\_STK\_SIZE / 10u ] , APP\_CFG\_TASK\_START\_STK\_SIZE,

137 0u ,

0u ,

139 0u ,

(OS\_OPT\_TASK\_STK\_CHK | OS\_OPT\_TASK\_STK\_CLR) ,

141 &e r r ) ;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 143 | OSStart(& e r r ) ;  I I I | ∗/ | | | /∗ 开启uC/OS− |
| 145  147 | while (DEF\_ON)  Get Here .  ;  } | { | ∗/ | /∗ Should Never | |
|  | } | | | |  |
| 149 |  | | | |  |
| 151  153 | /∗  ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗  ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗ | | | |  |
| 155 | ∗  ∗ STARTUP TASK | | | |  |
| 157 | ∗  ∗ De s c r i p t i o n : This i s an example o f a s ta r tu p task . | | | | As mentioned in the book |

’ s text , you MUST

∗ i n i t i a l i z e the t i c k e r only once m u l t i ta s k i n g has s ta r te d .

159 ∗

∗ Arguments : p\_arg i s the argument passed to ’ AppTaskStart ( ) ’ by ’ OSTaskCreate ( ) ’ .

161 ∗

∗ Returns : none

163 ∗

∗ Notes : 1 ) The f i r s t l i n e o f code i s used to prevent a compiler warning because ’ p\_arg ’ i s not

165 ∗ used . The compiler should not g e n e r ate any code f o r t h i s statement .

∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

167 ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗

∗

169 ∗/

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 171 s t a  { | t i c void AppTaskStart | ( void | ∗p\_arg ) |  | |
| 173  175 | OS\_ERR e r r ; CPU\_INT32U r 0 ;  CPU\_INT32U r 1 ; |  |  |
| 177 | CPU\_INT32U r 2 ;  CPU\_INT32U r 3 ; |  |  |
| 179 | CPU\_INT32U r 4 ;  CPU\_INT32U r 5 ; |  |  |
| 181 | CPU\_INT32U r 6 ;  CPU\_INT32U r 7 ; |  |  |
| 183 | CPU\_INT32U r 8 ;  CPU\_INT32U r 9 ; |  |  |
| 185 | CPU\_INT32U r 10 ;  CPU\_INT32U r 11 ; |  |  |
|  | CPU\_INT32U r 12 ; |  |  |
| 187 |  |  |  |
| 189 | ( void ) p\_arg ; |  |  |
| 191  193 | r 0 = 0u ;  l o c a l v a r i a b l e s . r 1 = 1u ;  r 2 = 2u ; |  |  | ∗/ | /∗ I n i t i a l i z e |
| 195 | r 3 = 3u ;  r 4 = 4u ; |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 197 | r 5  r 6 | =  = | 5u ;  6u ; | | | |
| 199 | r 7  r 8 | =  = | 7u ;  8u ; | | | |
| 201 | r 9  r 10 | =  = | 9u ;  10u ; | | | |
| 203 | r 11  r 12 | =  = | 11u ;  12u ; | | | |
| 205 | BSP\_Init ( ) ;持包CPU\_Init ( ) ;  块 | | | ∗/  ∗/ | /∗  /∗ | 初始化板级支初始化CPU模 |

207

#i f OS\_CFG\_STAT\_TASK\_EN > 0u

209 OSStatTaskCPUUsageInit(& e r r ) ; /∗ Compute CPU c a p a c i ty with no task running ∗/

#e n d i f

211

#i f d e f CPU\_CFG\_INT\_DIS\_MEAS\_EN

213 CPU\_IntDisMeasMaxCurReset ( ) ; #e n d i f

215

UART\_Config ( ) ;

217

/∗ I n i t i a l i z e LCD Communication f o r Ap p l i c ati o n . . . ∗/

219 BSP\_LCD\_Config ( ) ;

APP\_TRACE\_DBG( ( ” Creating Ap p l i c ati o n k e r n e l o b j e c t s \ r \n” ) ) ;

221 AppObjCreate ( ) ; /∗ 创建任务通信

过程中使用的信号量等对象 ∗/

223 APP\_TRACE\_DBG( ( ” Creating Ap p l i c ati o n Tasks \ r \n” ) ) ;

AppTaskCreate ( ) ; /∗ 创建应用任务

∗/

225

BSP\_LED\_Off(LED1) ;

227

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | while (DEF\_TRUE) {  无限循环 | ∗/ | /∗ | 启动任务进入 |
| 229 | OSTimeDlyHMSM( 0 u , 0u , 0u , 100 u , |  |  |  |
|  | OS\_OPT\_TIME\_HMSM\_STRICT, | | | |
| 231 | &e r r ) ; | | | |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 233 | i f | ( ( r 0 | != | 0u ) | | | |  | /∗ | Check | task |
|  |  | conte | xt . |  |  | ∗/ |  |  |  |
|  |  | ( r 1 | != | 1u ) | | | |  |  |  |  |
| 235 |  | ( r 2 | != | 2u ) | | | |  |  |  |  |
|  |  | ( r 3 | != | 3u ) | | | |  |  |  |  |
| 237 |  | ( r 4 | != | 4u ) | | | |  |  |  |  |
|  |  | ( r 5 | != | 5u ) | | | |  |  |  |  |
| 239 |  | ( r 6 | != | 6u ) | | | |  |  |  |  |
|  |  | ( r 7 | != | 7u ) | | | |  |  |  |  |
| 241 |  | ( r 8 | != | 8u ) | | | |  |  |  |  |
|  |  | ( r 9 | != | 9u ) | | | |  |  |  |  |
| 243 |  | ( r 10 | != | 10u ) | | | |  |  |  |  |
|  |  | ( r 11 | != | 11u ) | | | |  |  |  |  |
| 245 |  | ( r 12 | != | 12u ) ) | { |  |  |  |  |

APP\_TRACE\_INFO( ( ” Context Error \ r \n” ) ) ;

247 }

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 249 | } | | | |
| 251 | /∗ | | | |
|  | ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗ | | | |
| 253 | ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗ | | | |
|  | ∗ | | | |
| 255 | ∗ |  |  | AppTaskCreate ( ) |
|  | ∗ |  |  |  |
| 257 | ∗ | De s c r i p t i o n : | Create Ap p l i c ati o n Tasks . |  |
|  | ∗ |  |  |  |
| 259 | ∗ | Argument ( s ) : | none |  |
|  | ∗ |  |  |  |
| 261 | ∗ | Return ( s ) : | none |  |
|  | ∗ |  |  |  |
| 263 | ∗ | Ca l l e r ( s ) : | AppTaskStart ( ) |  |
|  | ∗ |  |  |  |
| 265 | ∗ | Note ( s ) : | 该任务创建应用任务 |  |
|  | ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗ | | | |
| 267 | ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗ | | | |
|  | ∗ | | | |
| 269 | ∗/ | | | |

271 s t a t i c void AppTaskCreate ( void )

{

273 OS\_ERR os\_err ;

275 /∗ −−−−−−−−−− 创建应用任务 −−−−−−−−− ∗/ OSTaskCreate(&Task\_ATCB,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 277  279 | ” Kernel Objects Task A” , Task\_A,  0 , | | | |
| 281 | APP\_CFG\_TASK\_OBJ\_PRIO,  &Task\_AStk [ 0 ] , | | | |
| 283 | Task\_AStk [APP\_CFG\_TASK\_OBJ\_STK\_SIZE / 10u ] ,  APP\_CFG\_TASK\_OBJ\_STK\_SIZE, | | | |
| 285 | 0u ,  0u , | | | |
| 287 | 0 ,  (OS\_OPT\_TASK\_STK\_CHK | OS\_OPT\_TASK\_STK\_CLR) , | | | |
|  | &os\_err ) ; | | | |
| 289 |  | | | |
| 291 | OSTaskCreate(&Task\_BTCB,  ” Kernel Objects Task B” , | | | |
| 293 | Task\_B ,  0 , | | | |
| 295 | APP\_CFG\_TASK\_OBJ\_PRIO,  &Task\_BStk [ 0 ] , | | | |
| 297 | Task\_BStk [APP\_CFG\_TASK\_OBJ\_STK\_SIZE / 10u ] ,  APP\_CFG\_TASK\_OBJ\_STK\_SIZE, | | | |
| 299 | 0u ,  0u , | | | |
| 301 | 0 ,  (OS\_OPT\_TASK\_STK\_CHK | OS\_OPT\_TASK\_STK\_CLR) , | | | |
| 303 | &os\_err ) ;  } | | | |
| 305  307 | /∗  ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗  ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗ | | | |
|  | ∗ | | | |
| 309  311 | ∗  ∗  ∗ | De s c r i p t i o n | : | AppObjCreate ( )  Create Ap p l i c ati o n Kernel Objects . |
| 313 | ∗  ∗ | Argument ( s ) | : | none |
| 315 | ∗  ∗ | Return ( s ) | : | none |
| 317 | ∗  ∗ | Ca l l e r ( s ) | : | AppTaskStart ( ) |

|  |  |
| --- | --- |
|  | ∗ |
| 319 | ∗ Note ( s ) : 该任务创建任务通信过程中使用的信号量。 |
|  | ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗ |
| 321 | ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗ |
|  | ∗ |
| 323 | ∗/ |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 325 | s t a t i c void AppObjCreate ( void )  { | | | |
|  | OS\_ERR os\_err ; | | | |
| 327 |  | | | |
| 329 | #i f (OS\_CFG\_SEM\_EN > 0u )  OSSemCreate(&wait\_key\_sem , | | | |
| 331 | ”Key Detection ” ,  0u , | | | |
| 333 | &os\_err ) ;  OSSemCreate(&wait\_key\_sem2 , ”Key Detection ” , 0 u,& os\_err ) ; | | | |
| 335 | #e n d i f  } | | | |
| 337  339 | /∗  ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗  ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗ | | | |
|  | ∗ | | | |
| 341  343 | ∗  ∗  ∗ | De s c r i p t i o n | : | Task\_A( )  Test uC/OS−I I I o b j e c t s . |
| 345 | ∗  ∗ | Argument ( s ) | : | p\_arg i s the argument passed to ’Task\_A ’ by ’ OSTaskCreate ( ) ’ . |
| 347 | ∗  ∗ | Return ( s ) | : | none |
| 349 | ∗  ∗ | Ca l l e r ( s ) | : | This i s a task |
| 351 | ∗  ∗ | Note ( s ) | : | 该任务检测S3 是否被按下并释放， 当S3 释放时向Task\_B发送信号量。 |
| 353 | ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗  ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗ | | | |
| 355 | ∗  ∗/ | | | |
| 357 | void Task\_A( void ∗p\_arg )  { | | | |
| 359 | OS\_ERR e r r ;  unsigned char key\_press = 0 ; | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 361  363 | whil  {  // | | e (DEF\_TRUE)  OSTimeDly ( 30 , OS\_OPT\_TIME\_DLY, &e r r ) ; | | |
|  | // | | i f ( BSP\_PB\_GetState(BUTTON\_TAMPER) == RESET) | | |
| 365 | // | | key\_press = 1 ; | | |
|  | // | | i f ( BSP\_PB\_GetState(BUTTON\_TAMPER) == SET) | | |
| 367 | // | | { | | |
|  | // | | i f ( key\_press == 1 ) | | |
| 369 | // | | { | | |
|  | // | | key\_press = 0 ; | | |
| 371 | // | | OSSemPost(&wait\_key\_sem , OS\_OPT\_POST\_1, &e r r ) ; | | |
|  | // | | } | | |
| 373 | // | | } | | |
|  |  | | OSTimeDly ( 60 , OS\_OPT\_TIME\_DLY, &e r r ) ; | | |
| 375 |  | | i f ( BSP\_PB\_GetState(BUTTON\_TAMPER) == RESET) { | | |
|  |  | | OSSemPost(&wait\_key\_sem , OS\_OPT\_POST\_1, &e r r ) ; | | |
| 377 |  | | } | | |
|  | } | |  | | |
| 379 | } | | | | |
|  | /∗ | | | | |
| 381 | ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗ | | | | |
|  | ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗ | | | | |
| 383 | ∗ | | | | |
| ∗ | | | Task\_B( ) | | |
|  | | |  | | |
| 385 | ∗ |  | |  |  |
|  | ∗ | De s c r i p t i o n : | | Test uC/OS−I I I o b j e c t s . |  |
| 387 | ∗ |  | |  |  |
|  | ∗ | Argument ( s ) : | | p\_arg i s the argument passed to ’ Task\_B ’ by | ’ OSTaskCreate ( ) ’ . |
| 389 | ∗ |  | |  |  |
|  | ∗ | Return ( s ) : | | none |  |
| 391 | ∗ |  | |  |  |
|  | ∗ | Ca l l e r ( s ) : | | This i s a task |  |
| 393 | ∗ |  | |  |  |
|  | ∗ | Note ( s ) : | | 该任务等待Task\_A发送来的信号量， 当获取到信号 | 量时点亮/ 熄灭LED1并 |
| 395 | 向串口发送数据。  ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗ | | | | |
|  | ∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗ | | | | |
| 397 | ∗ | | | | |
|  | ∗/ | | | | |
| 399 | void Task\_B( void ∗p\_arg ) | | | | |
|  | { | | | | |
| 401 | OS\_ERR e r r ; | | | | |
|  | CPU\_TS t s ; | | | | |

|  |  |
| --- | --- |
| 403 | while (DEF\_TRUE) |
| 405 | { |
|  | OSTimeDly ( 30 , OS\_OPT\_TIME\_DLY, &e r r ) ; |
| 407 | OSSemPend(&wait\_key\_sem , |
|  | 0 , |
| 409 | OS\_OPT\_PEND\_BLOCKING, |
|  | &ts , |
| 411 | &e r r ) ; |
|  | BSP\_LED\_Toggle(LED1) ; // Toggle a f t e r key r e l e a s e |
| 413 | OSTimeDly ( 30 , OS\_OPT\_TIME\_DLY, &e r r ) ; |
|  | BSP\_LED\_Toggle(LED1) ; |
| 415 | APP\_TRACE\_DBG( ( ” Get a Semaphore from Task A\n\ r ” ) ) ; |
|  | } |
| 417 | } |