蓝桥杯嵌入式备赛-指导手册

常州信息职业技术学院 应电191 翟宇涵

STM32 代码规范

考虑到程序模块化，代码可读性，需要制定一套自己的代码规范。下面给出一份参考：

代码放置位置

用户头文件的引用

```
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */
```

用户自定义数据类型

```
/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */
```

用户宏函数定义

```
/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */
```

用户常量定义

```
/* Private variables ---------------------------------------------------------*/

/* USER CODE BEGIN PV */

/* USER CODE END PV */
```

用户变量定义

```
/* Private variables ---------------------------------------------------------*/

/* USER CODE BEGIN PV */

/* USER CODE END PV */
```

用户函数声明

```
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */
```

功能实现位置

```
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */
```

各种初始化位置

```
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  /* USER CODE BEGIN 2 */

  /* USER CODE END 2 */
```

后台程序

```
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */


    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
```

前台程序&回调函数

可以尝试把各类模块的回调函数放在USER CODE 4之间，这样可以使得函数划分更清晰。

```
/* USER CODE BEGIN 4 */

/* USER CODE END 4 */
```

命名规范

全局变量

对于声明在程序开头，即PV之间的全局变量，采用大写。例如，设置`GPIO->ODR`的`Led_Val`，或者是储存显示界面，或者设置选项序列`View_Model`，`Setting_Index`。

局部变量

如果是用在函数内部的局部变量，用小写。例如放在定时器中断回调函数中计数用的静态变量，用来统计计时的，如`static uint8_t led_count`，或者`static uint8_t key_count`。

函数命名

对于函数命令，应尽量体现其功能意义。如打开LED，即`TurnOn_LED()`，采用动词加名词的命名方式。

涉及到一些模块功能实现时，可以采用模块名加Proc的命名方式。例如LCD模块的显示功能，就是`LCD_Proc()`。或者是KEY模块的处理功能，`KEY_Proc()`。

变量放置规则

标志位应当处于开头。即开头应当是

```
1   _Bool Adc_Flag = 1;//ADC的标志位
2   _Bool Key_Flag = 0;//KEY的标志位
3   _Bool Led_Flag = 0;//LED的标志位
```

之类。这样的好处是通过阅读标志位就可以知道本程序大致使用了哪些模块。

接着是全局会用的到的变量，例如`Led_Val`。

由于涉及到界面切换，所以设置界面序列的`uint8_t View_Model`应该先写出来。根据根据服务不同界面作为标准，将同一界面会用到的变量放置在一块。当然也可以使使用结构体把同一界面用到的变量统一存放。

注释规范

多写注释，便于后期阅读。尤其是注释要简单易懂。例如，变量标志位的注释可以这么写，`//LED的标志位`。要清楚的变量该变量的归属，是用来干什么的。当设计到一个模块初始化的时候，可以用注释来作为间隔，例如，`//初始化LCD //初始化I2C`。一个功能的开头也可以用注释来，例如`//按键1 界面切换`。

编写要点

善用define

为了考虑代码阅读的方便，应当define一些变量。例如在界面切换时，通常使用`View_Model = View_Model % 3 + 1;`在后续的处理函数中，不管是用if语句还是case语句判断条件，采用define后的名称，例如`SETTINGVIEW`，就代表了是设置界面，方便阅读。

有条理的初始化

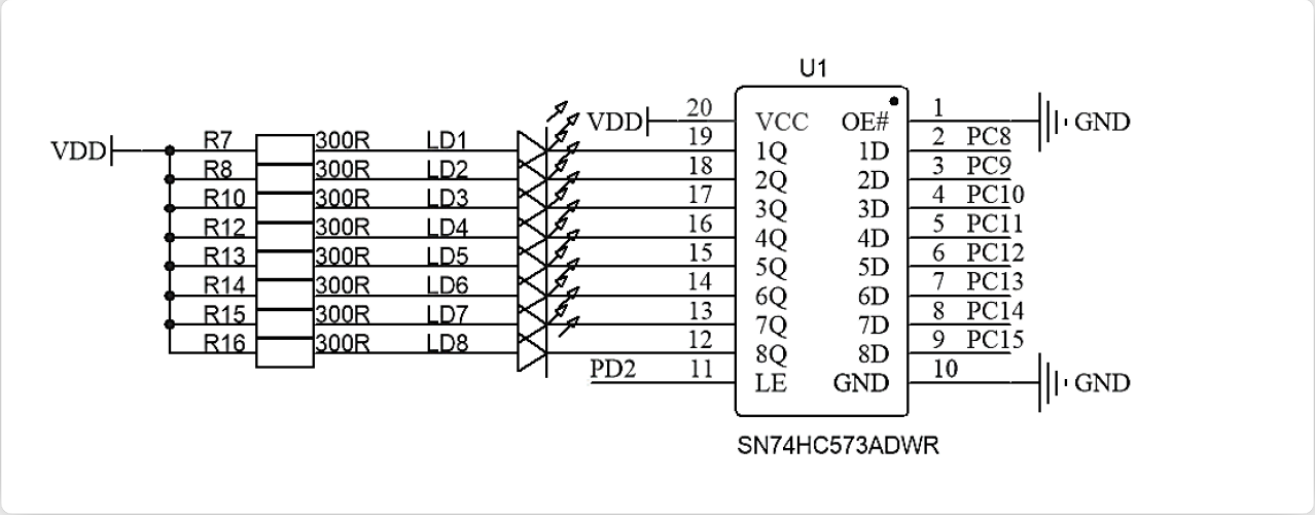
要在CubeMX工具初始化各外设好后进行自己的模块初始化。建议在USER CODE BEGIN 2之间进行初始化，根据功能和应用进行有条理的初始化，将会使程序条理更加清晰，减少失误。本指导手册建议初始化LCD，如果有数据读取要求，可以先初始化I2C，读取EEPROM数据，便于将数据显示在LCD屏幕上，更符合工作流程。但是如果有用到定时器，串口之类的中断功能，一定要记得在初始化LCD等模块前先开启中断，避免因为忘记开启中断导致后期功能无法实现，会浪费时间用于检查。

下面给出一段示例：

```
HAL_TIM_Base_Start_IT(&htim4);
HAL_UART_Receive_IT(&huart1, (uint8_t *)rx_buffer, 1);
//初始化I2C读取数据
I2CInit();
Th1 = x24c02_read(0x01);
HAL_Delay(5);
Th2 = x24c02_read(0x02);
HAL_Delay(5);
Th3 = x24c02_read(0x03);
HAL_Delay(5);
//初始化LCD
LCD_Init();
LCD_Clear(White);
LCD_SetBackColor(White);
LCD_SetTextColor(Black);
//关闭LED
TurnOff_LEDS();
```

考点 LED

LED与LCD会发生冲突，在初始化LCD之后需要关闭LED。单独编写操作LED的函数。



首先需要配置PD2口，这里可以锁住LED。将PD2配置为GPIO_Output，并且另外命名为LD_CLK，其余参数默认即可。

接着需要编写一些基础功能函数。例如关闭全部LED，打开LED，关闭LED等。

第一步：定义Led_Val

主要是把Led_Val的值赋给GPIOC口的ODR寄存器，来控制LED。

```
uint16_t Led_Val;
```

第二步：关闭全部LED

```
void TurnOff_LEDS()
{
  Led_Val = 0xFF00;
  HAL_GPIO_WritePin(LD_CLK_GPIO_Port, LD_CLK_Pin, GPIO_PIN_SET);
  GPIOC->ODR = (uint32_t)Led_Val;
  HAL_GPIO_WritePin(LD_CLK_GPIO_Port, LD_CLK_Pin, GPIO_PIN_RESET);
}
```

第三步：打开LED

```
void TurnOn_LED(uint8_t _led)
{
  Led_Val &= ~(0x01<<(7+_led));
  HAL_GPIO_WritePin(LD_CLK_GPIO_Port, LD_CLK_Pin, GPIO_PIN_SET);
  GPIOC->ODR = (uint32_t)Led_Val;
  HAL_GPIO_WritePin(LD_CLK_GPIO_Port, LD_CLK_Pin, GPIO_PIN_RESET);
}
```

第四步：关闭LED

```
void TurnOff_LED(uint8_t _led)
{
  Led_Val |= (0x01<<(7+_led));
  HAL_GPIO_WritePin(LD_CLK_GPIO_Port, LD_CLK_Pin, GPIO_PIN_SET);
  GPIOC->ODR = (uint32_t)Led_Val;
  HAL_GPIO_WritePin(LD_CLK_GPIO_Port, LD_CLK_Pin, GPIO_PIN_RESET);
}
```

第五步：控制LED

```
void Control_LED(uint8_t _led, uint8_t status)
{
  HAL_GPIO_WritePin(LD_CLK_GPIO_Port, LD_CLK_Pin, GPIO_PIN_SET);
  if(status != 0)
  {
    GPIOC->BRR = _led << 8;
  }else{
    GPIOC->BSRR = _led << 8;
  }
  HAL_GPIO_WritePin(LD_CLK_GPIO_Port, LD_CLK_Pin, GPIO_PIN_RESET);
}
```

第六步：翻转LED

```
void Toogle_LED(uint8_t _led)
{
  HAL_GPIO_WritePin(LD_CLK_GPIO_Port, LD_CLK_Pin, GPIO_PIN_SET);
  HAL_GPIO_TogglePin(GPIOC, _led << 8);
  HAL_GPIO_WritePin(LD_CLK_GPIO_Port, LD_CLK_Pin, GPIO_PIN_RESET);
}
```

真题演练

第7届省赛的LED考点

> 以0.2秒的间隔闪烁5次

为了实现该功能，定义一个Led_200ms_Flag的标志位，通过定时器定时置1。到达该时间后，进行闪烁，并定义一个led_count变量，用来统计这是第几次的闪烁。一亮一灭才叫一次闪烁。

```
if(Led3_Flag)
{
    if(Led3_200ms_Flag)
    {
      Led3_200ms_Flag = 0;
      led3_count++;
      if(led3_count <= 11)
      {
        LD3_Blink();
      }else
      {
        ledodd3 = 0;    //最后一次要清0，因为再次开始时状态重置
        led3_count = 0;
        Led3_Flag = 0;
      }
    }
}
```

考点 LCD

初始化

初始化LCD首先需要将lcd.c文件添加进工程目录里，其次将lcd.h头文件引入。

在/* USER CODE BEGIN 2 * /和 /* USER CODE BEGIN 2 * /之间初始化LCD。

```
//初始化LCD
LCD_Init();
LCD_Clear(White);
LCD_SetBackColor(White);
LCD_SetTextColor(Black);
//关闭LED
TurnOff_LEDS();
```

可以在初始化LCD之后关闭LED。

常用函数

`void LCD_DisplayChar(u8 Line, u16 Column, u8 Ascii);`
`void LCD_DisplayStringLine(u8 Line, u8 *ptr);`

上面是用来显示字符，下面是用来显示字符串的。不管是显示字符还是字符串，都需要指定显示的行数。从Line0到Line9都是可用的。显示字符时还需要指定列。LCD屏幕的宽度是范围是0~319，一个字符的占到了16，因此将一个字符a显示在第一行第一列需要这么写：

`void LCD_DisplayChar(Line0, 319 - 16, 'a');`

LCD_DisplayChar接收的是ascii码，如果想显示的是数字，可用在数字后面加上48，也可以加上'0'，将它转成字符。

格式化输出

通常我们会将数据通过sprintf()函数格式化以后，进行输出。

```
uint8_t buf[30];
sprintf((char *)buf, "%02d : %02d : %02d", hour, min, sec);
```

常用的格式字符：

- 输出字符：%c
- 输出字符串：%s
- 输出整数：%d
- 输出浮点数：%f

输出进制数

- 带先导八进制：%#o
- 不带先导八进制：%o
- 带先导十六进制：%#x
- 不带先导十六进制：%x

特定格式

- 自动添0：%02d
- 指定小数位: %.2f
- 左对齐：%-d
- 百分号：%%

注意点

通常，使用按键进行界面的切换，在进行界面切换时，如果不清除原屏的话，有可能会导致上一界面的内容还留在屏幕上。因此在界面切换的时候，先使用LCD_Clear(White)函数进行清屏。

此外，可以关注一下时刻刷新LCD屏是否会造成系统的阻塞，比如利用定时器进行10ms的按键扫描，有可能会因为LCD屏幕刷新的频繁导致30ms才能进行一次按键扫描，此时可以适当减缓LCD屏幕刷新速度。

真题演练

第11届LCD的考点

在进行时间设置的切换时，高亮来显示设置的是时还是分还是秒。

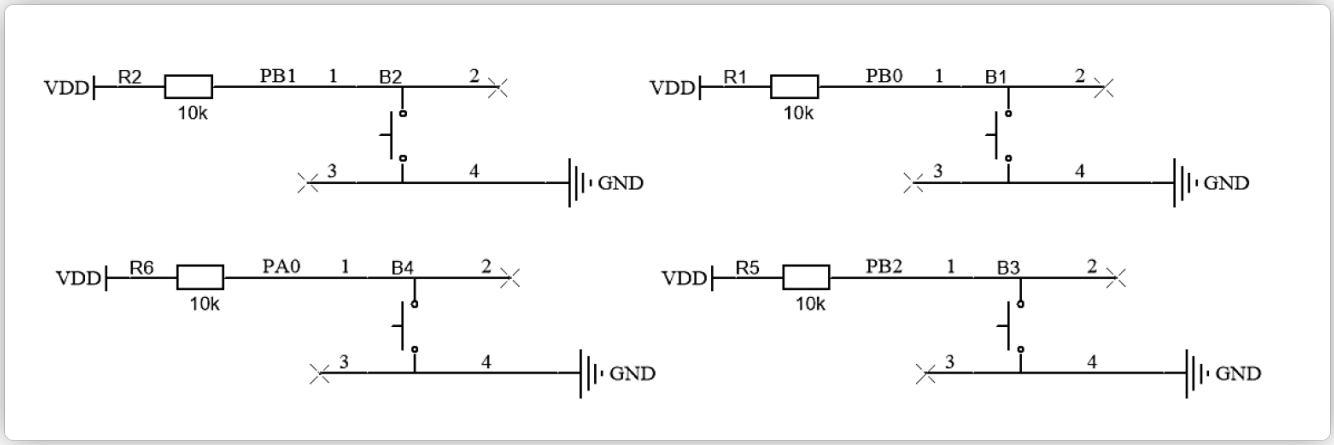
通常我们会定义一个SettingModel来选择设置哪一个变量。

```
switch(SettingModel)
{
  case HOUR_SELECT:
    LCD_SetTextColor(Red);
    LCD_DisplayChar(Line3, 319-16* 8, hour/10+48);
    LCD_DisplayChar(Line3, 319-16* 9, hour%10+48);
    LCD_SetTextColor(Black);
    LCD_DisplayChar(Line3, 319-16* 11, min/10+48);
    LCD_DisplayChar(Line3, 319-16* 12, min%10+48);
    LCD_DisplayChar(Line3, 319-16* 14, sec/10+48);
    LCD_DisplayChar(Line3, 319-16* 15, sec%10+48);
    //...
}
```

也可以使用if语言。

```
if(SettingModel == HOUR_SELECT)
  LCD_SetTextColor(Red);
  LCD_DisplayChar(Line3, 319-16* 8, hour/10+48);
  LCD_DisplayChar(Line3, 319-16* 9, hour%10+48);
  LCD_SetTextColor(Black);
else
  LCD_DisplayChar(Line3, 319-16* 8, hour/10+48);
  LCD_DisplayChar(Line3, 319-16* 9, hour%10+48);
```

考点 KEY



在配置时，将PB0，PB1，PB2，PA0设为GPIO_Input，其余参数默认即可。

使用#define简化程序

在/* USER CODE BEGIN PM * /和/* USER CODE END PM * /之间编写

```
#define KB1 HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin)
#define KB2 HAL_GPIO_ReadPin(B2_GPIO_Port, B2_Pin)
#define KB3 HAL_GPIO_ReadPin(B3_GPIO_Port, B3_Pin)
#define KB4 HAL_GPIO_ReadPin(B4_GPIO_Port, B4_Pin)
```

使用定时器进行10ms的定时检测

与LED相同，定义一个专门的count与flag，来进行按键的检测。定义一个key_count，当计数到10ms的时候，将Key_Flag置1，此时在主函数的while循环中检测到该标志位，调用Key_Proc()函数，运行按键方面的功能。

Key_Proc()函数

适合数字加减的按键业务函数

```
void Key_Proc()
{
  static uint16_t b1_sum = 0;
  if(KB1 == 0)
  {
    b1_sum++;
    if(b1_sum == 1)
```

```
      {
        //短按
      }
      if(b1_sum == 100)
      {
        b1_sum = 90;
        //长按
      }
    }
    if(KB1 == 1)
    {
      b1_sum = 0;//松手
    }
  }
```

当按键按下时，检测到低电平，所以为0时按键按下，开始计数，松开时则计数清0。当计数超过指定时间时，将计数值减小到一个靠近的值，以此来实现长按效果。

适合长按切换界面的函数

```
void Key_Proc()
{
  static uint16_t b1_sum = 0;
  if(KB1 == 0)
  {
    b1_sum++;
    if(b1_sum == 100)
    {
      b1_sum = 0;
            //长按
    }
  }
  if(KB1 == 1)
  {
    if(b1_sum >= 1 && b1_sum < 100)
    {
      //短按
    }
    b1_sum = 0;//松手
  }
}
```

将短按的判断写在松手检测里，是为了避免计数到1的时候，就触发了按键短按的功能。达到长按要求时就清0，避免了二次触发。

双击检测

实现双击的思路就是，完成一次短按之后，定义一个变量，将变量自增。利用定时器，在固定时间内对该变量清0。如果可以在主循环中检测到该变量为2，那么就可以判断完成了一次双击操作。

```
void Key_Proc()
{
  static uint16_t b1_sum = 0;
  if(KB1 == 0)
  {
    b1_sum++;
    if(b1_sum == 1)
    {
      KB1_DCLK++;
    }
  }
  if(KB1 == 1)
  {
    b1_sum = 0;
  }
}
```

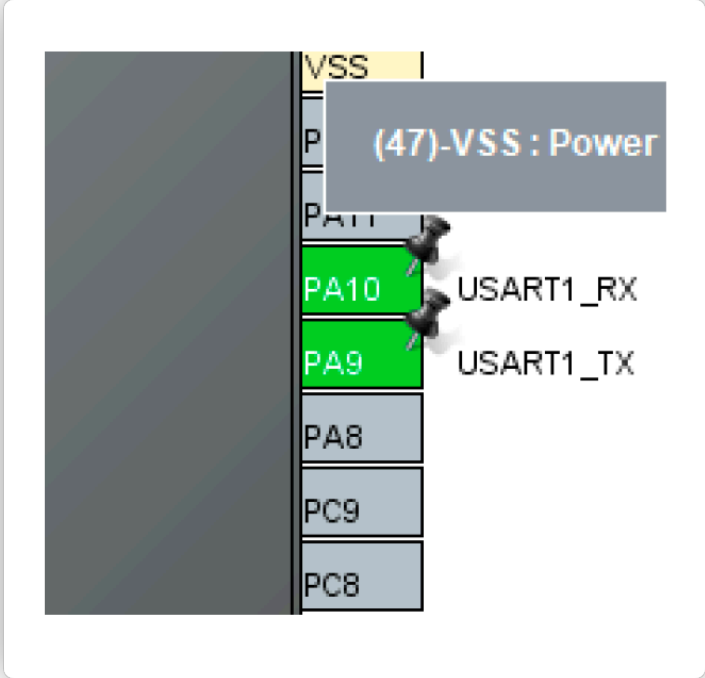
一个完整的Key_Proc()

```
void Key_Proc()
{
  static uint16_t b1_sum = 0;
  if(KB1 == 0)
  {
    b1_sum++;
    if(b1_sum == 1)
    {
      KB1_DCLK++;
    }
  }
  if(KB1 == 1)
  {
    if(b1_sum >= 1 && b1_sum < 10)
    {
```

```
16          //完成了一次短按
17          KB1_ShortClick_Sum++;
18          b1_sum = 0;
19        }
20        if(b1_sum >= 10)
21        {
22          b1_sum = 0;
23          //完成了一次长按
24          KB1_LongClick_Sum++;
25        }
26      }
27
28  }
```

考点 USART

参考原理图，PA9为发端，PA10为收端，所以要在CubeMX配置的时候，选择好PA9，PA10为串口的收发口。



通过重定向fputc(int ch, FILE *f)来实现printf()发送数据

```c
int fputc(int ch, FILE *f)
{
  HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, HAL_MAX_DELAY);
  return ch;
}
```

接收数据

使用中断的方式进行数据接收。

首先要定义如下变量

```
/* USER CODE BEGIN PV */
uint8_t rx_buff[1];//串口接收的缓存
uint8_t rx_data[100];//串口接收的数据
_Bool rx_Flag;//串口的标志位
/* USER CODE END PV */
```

接着在中断接收的回调函数中写到

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
  static uint8_t rx_count;
  if(huart->Instance == USART1)
  {
    //接收单字
    if(rx_buff[0] == 'S' || rx_buff[0] == 'C')
    {
      rx_count = 0;
      memset(rx_data, 0, sizeof(rx_data));
      printf("Print S");
    }
    rx_data[rx_count] = rx_buff[0];

    //接收带帧尾
    if((rx_data[rx_count - 1] == 0x0D) && (rx_data[rx_count] == 0x0A))
    {                           //0x0D即\r 0x0A即\n
      rx_count = 0;
      rx_Flag = 1;
    }else{
      rx_count++;
    }
    HAL_UART_Receive_IT(&huart1, rx_buff, 1);
  }
}
```

需要注意的是，在CubeMX自动生成的初始化函数之后，要开启中断接收。

```
/* USER CODE BEGIN 2 */
  HAL_UART_Receive_IT(&huart1, rx_buff, 1);//启动中断接收方式
/* USER CODE END 2 */
```

闲时处理

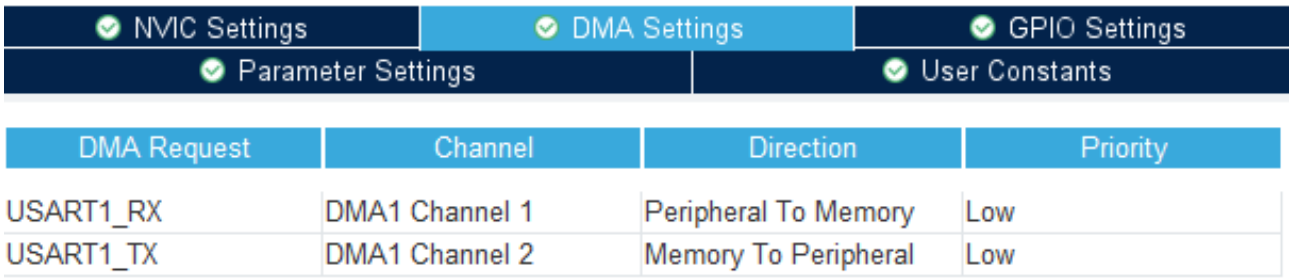
当接收到错误的数据时，可以通过错误标志位清除，也可以使用闲时处理进行清除。即定义一个标志位，使用定时器，定时清空接收的数据。

```
memset(rx_data, 0, sizeof(rx_data));
```

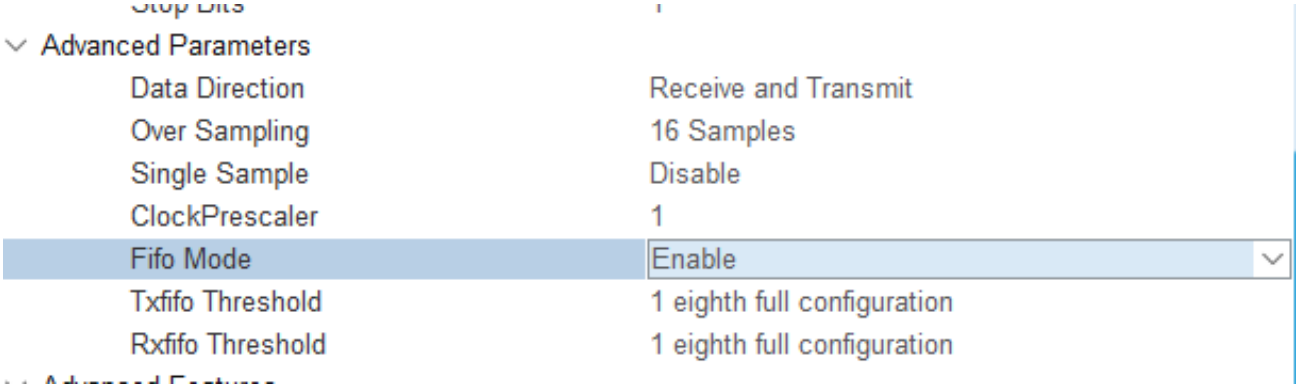
使用memset()需要引入string.h头文件

DMA&空闲中断

开启串口之后，还要配置DMA



可以启用FIFO



需要定义缓冲区的长度，缓冲区，完成接收的标志位，

```
#define BUFFER_SIZE 100
extern uint8_t rx_buffer[];
extern uint8_t rx_Flag;
extern uint16_t rx_len;
```

在串口1中断函数中要这么调整

```
void USART1_IRQHandler(void)
{
  /* USER CODE BEGIN USART1_IRQn 0 */
  uint32_t temp;
  /* USER CODE END USART1_IRQn 0 */
  HAL_UART_IRQHandler(&huart1);
  /* USER CODE BEGIN USART1_IRQn 1 */
  if(__HAL_UART_GET_FLAG(&huart1, UART_FLAG_IDLE) != RESET)
  {
    __HAL_UART_CLEAR_IDLEFLAG(&huart1);
    temp = huart1.Instance->ISR;
    temp = huart1.Instance->RDR;
    HAL_UART_AbortReceive(&huart1);
    temp = hdma_usart1_rx.Instance->CNDTR;
    rx_len = BUFFER_SIZE-temp;
    rx_Flag = 1;
  }
  /* USER CODE END USART1_IRQn 1 */
}
```

接着就是要开启串口的空闲中断以及DMA接收

```
__HAL_UART_ENABLE_IT(&huart1, UART_IT_IDLE);/*启动串口DMA接收*/
HAL_UART_Receive_DMA(&huart1,rx_buffer,BUFFER_SIZE);
```

```
if(rx_Flag == 1)
{
  HAL_UART_Transmit_DMA(&huart1,rx_buffer,rx_len);
  rx_len=0;
  rx_Flag=0;
  HAL_UART_Receive_DMA(&huart1,rx_buffer,BUFFER_SIZE);
}
```

使用库函数处理接收数据

熟练使用sscanf()和strcmp()来处理接收数据。

尤其是通过串口进行一些设置指令匹配时，使用strcmp()来代替==。

```
if(strcmp((char *)rx_data, (char*)rx_order) == 0)
{
    //do something...
}
```

真题演练

第五届UART考点

当系统处于“串口设定”方式时，可通过串口设定 2 个通道的倍频数，格式如下：

`SET:x:yy` 其中：SET：设定标志；x：通道号，1 或 2；yy：倍频数，2 到 10。

根据接收数据的特点可以定义一个专门用来接收数据的结构体

```
struct typdef{
  char c1,c2, c3, c4;
  int x;
  char c5;
  int y;
}rx_data;
```

在检测到接收完成的标志位后可以这么处理。

```
if(rx_flag == 1)
{
  sscanf((char*)rx_buffer,
          "%c %c %c %c %u %c %u",
          &rx_data.c1, &rx_data.c2, &rx_data.c3, &rx_data.c4, &rx_data.x,
&rx_data.c5,&rx_data.y);
}
```

通过这样就可以把数据拆分开来。

实际在C语言的库函数中，可以有匹配方式。但在测试中，发现无法成功，据推测是堆栈空间不足。但是可以匹配":"。

```
sscanf("SET:1:2", "%[^:]", buffer);
```

考点 ADC

ADC校准

在开启ADC之前，要进行校准。

```
HAL_ADCEx_Calibration_Start(&hadc1, ADC_SINGLE_ENDED);                    //校准ADC
```

DMA方式传输

```
HAL_ADC_Start_DMA(&hadc2, (uint32_t *)&ADC_buf,10);                  //启动ADC转换DMA方式
```

普通均值法滤波

```
void getADC()
{
  uint8_t i;
  Adc_Val = 0;
  for(i = 0; i < 10; i++)
  {
    Adc_Val += ADC_buf[i];
  }
  V_Val = (Adc_Val / 10) / 4096 * 3.3f;
}
```

ADC完成中断回调函数

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
  if(hadc == &hadc2)
  {
      Adc_Con_Flag = 1;
  }
}
```

ADC中位值滤波

```
float Adc_Proc(uint32_t _buf[])
{
  uint8_t i, j;
  float Adc_Val = 0;
  float temp;
  for(i = 0; i < 10 - 1; i++)
  {
    for(j = 0; j < 10 - 1 - i; j++)
    {
      if (_buf[j] < _buf[j+1])
      {
          temp = _buf[j];
          _buf[j] = _buf[j+1];
          _buf[j+1] = temp;
      }
    }
  }

  for(i = 1; i < 10-1; i++)
      Adc_Val += _buf[i];
  return (Adc_Val / 8) / 4096 * 3.3f;
}
```

考点DAC

```c
void Dac1_Set_Vol(float vol)
{
  uint16_t temp;
  temp = (4096*vol/3.3f);
  HAL_DAC_SetValue(&hdac1, DAC_CHANNEL_1,DAC_ALIGN_12B_R,temp);
}
```

考点 TIM

基本的定时功能

通常要定时进行按键扫描，或者ADC采集工作。通常会定义一个标志位，当检测到标志位置1的时候，就进行操作。这个标志位置1的工作放在定时器的中断回调函数里。

如果设置工作频率在80MHz，那么可以在CubeMX中这么设置，来进行1ms的定时。

Counter Settings	
Prescaler (PSC - 16 bits value)	80-1
Counter Mode	Up
Dithering	Disable
Counter Period (AutoReload Register - 1...	1000-1
Internal Clock Division (CKD)	No Division
auto-reload preload	Disable

最不要忘记的就是开启定时器中断。

```c
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  HAL_TIM_Base_Start_IT(&htim4);
  static uint8_t key_count;
  if(htim->Instance == TIM4)
  {
    if(++key_count == 10)
    {
      key_count = 0;
      Key_Flag = 1;
    }
  }
}
```

PWM输出

$$\text{频率} = \text{定时器时钟} / (\text{预分频} + 1) / (\text{计数} + 1) Hz$$

$$\text{占空比} = \text{对比值} / \text{计数值} \times 100\%$$

单个定时器不同通道输出不同占空比的PWM波

```
__HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_1, p_value);
```

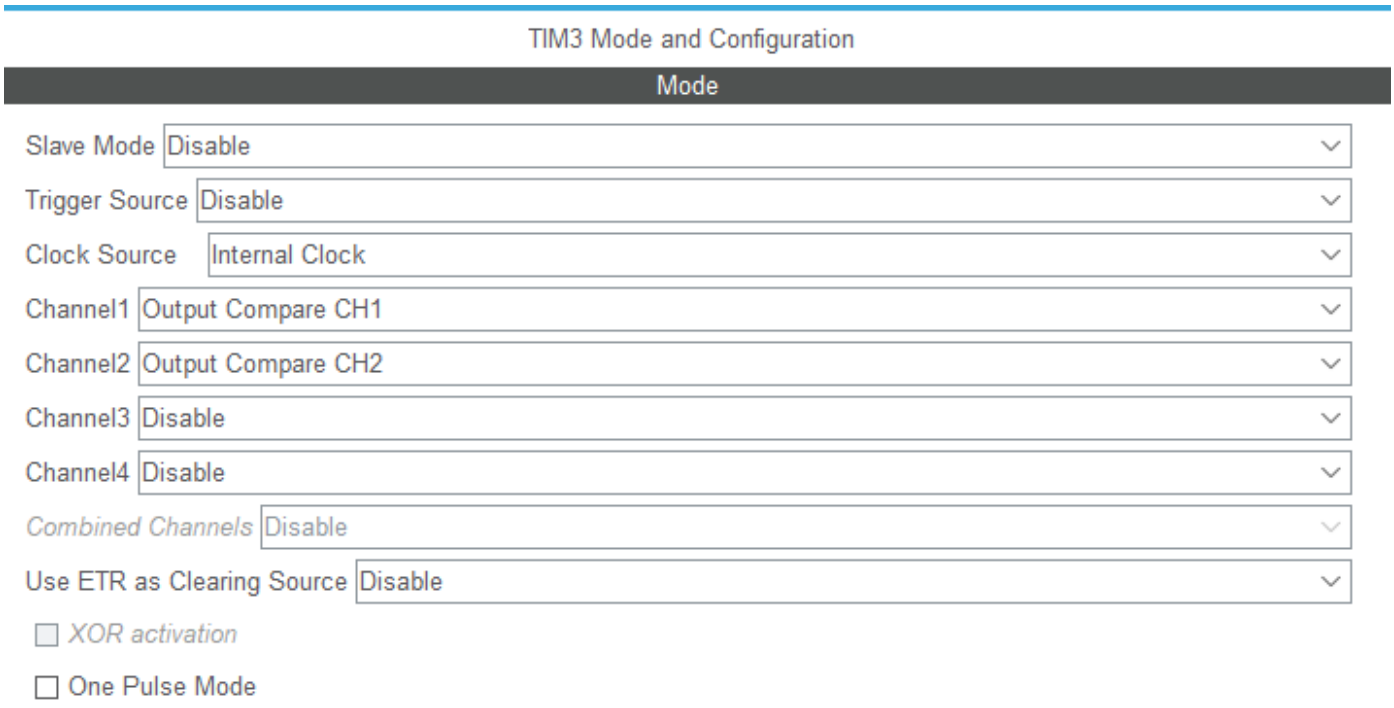
单个定时器单个通道输出不同频率不同占空比的PWM波

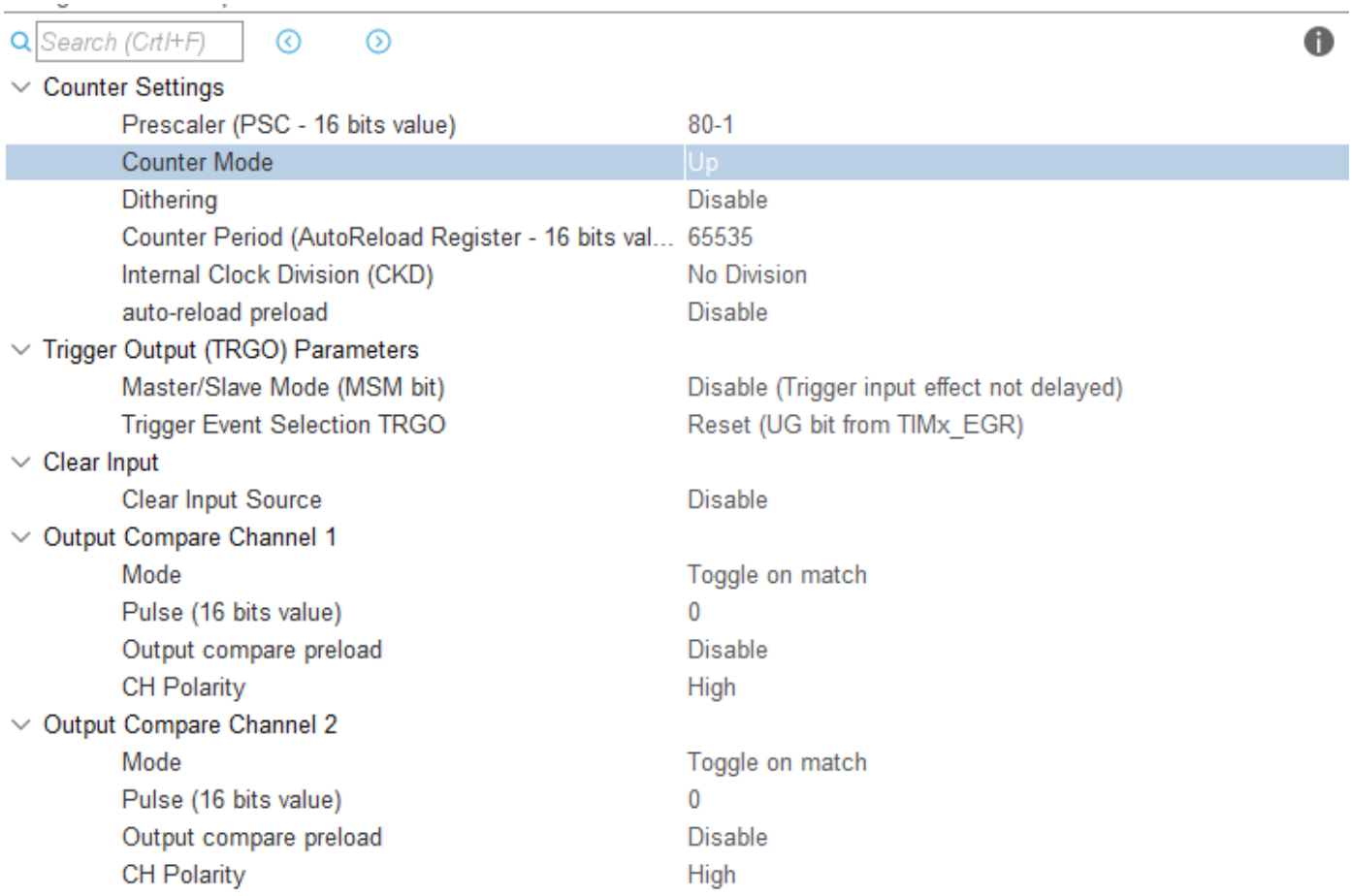
```
__HAL_TIM_SET_COUNTER(&htim2,0);
__HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_2, 5000);
__HAL_TIM_SET_AUTORELOAD(&htim2, 10000);//输出100Hz，50%占空比的PWM波
```

单个定时器多个通道输出不同频率不同占空比的PWM波

如果要单个定时器输出不同频率不同占空比的方波需要使用输出比较模式，输出比较模式是在时基单元上添加对应的通道输出功能，以此达到规律的电平输出功能。在F103RBT6的时候，定时器数量过少，而使用G431RBT6可以考虑使用不同的定时器进行输出。

赛题经常是使用PA6和PA7进行输出。可以使用定时器3的通道2和通道3，在CubeMX中配置如下





对应的需要定义每个通道的PSC和对应Pulse的值

```
uint32_t tim3_ch1_ccr = 100;
uint32_t tim3_ch1_pulse = 20;
uint32_t tim3_ch2_ccr = 200;
uint32_t tim3_ch2_pulse = 100;
```

那么根据频率和占空比的计算方法， 既然可以得出在定时器3通道1输出了10kHz，占空比为百分之20的PWM波。

在运行过程中可以对这些值进行修改。之后就开启输出比较的中断。

```
HAL_TIM_OC_Start_IT(&htim3, TIM_CHANNEL_1);
HAL_TIM_OC_Start_IT(&htim3, TIM_CHANNEL_2);
```

可以使用以下两个中断回调函数

```
HAL_TIM_OC_DelayElapsedCallback(htim);
HAL_TIM_PWM_PulseFinishedCallback(htim);
```

```
void HAL_TIM_PWM_PulseFinishedCallback(TIM_HandleTypeDef *htim)
{
  static uint8_t cc1_flag = 0;
  static uint8_t cc2_flag = 0;
  uint32_t capture = 0;
  if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1)
  {
    capture = HAL_TIM_ReadCapturedValue(&htim3, TIM_CHANNEL_1);
    if(cc1_flag)
    {
      __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_1, capture + tim3_ch1_ccr - tim3_ch1_pulse);
    }else{
      __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_1, capture + tim3_ch1_pulse);
    }
    cc1_flag = !cc1_flag;
  }
  if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_2)
  {
    capture = HAL_TIM_ReadCapturedValue(&htim3, TIM_CHANNEL_2);
    if(cc2_flag)
    {
      __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_2, capture + tim3_ch2_ccr - tim3_ch2_pulse);
    }else{
      __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_2, capture + tim3_ch2_pulse);
    }
    cc2_flag = !cc2_flag;
  }


  HAL_TIM_OC_Start_IT(&htim3, TIM_CHANNEL_1);
  HAL_TIM_OC_Start_IT(&htim3, TIM_CHANNEL_2);

}
```

值得讨论的是，当使用PA1和PA2进行输出，即32位定时器2的时候出现了问题。当Debug的时候发现CCR寄存器的值根本没有发生变化。猜测是ARR寄存器已经是0XFFFFFFFF。可以在初始化定时器时改变设置`htim2.Init.Period = 65535;`

输入捕获

测量频率

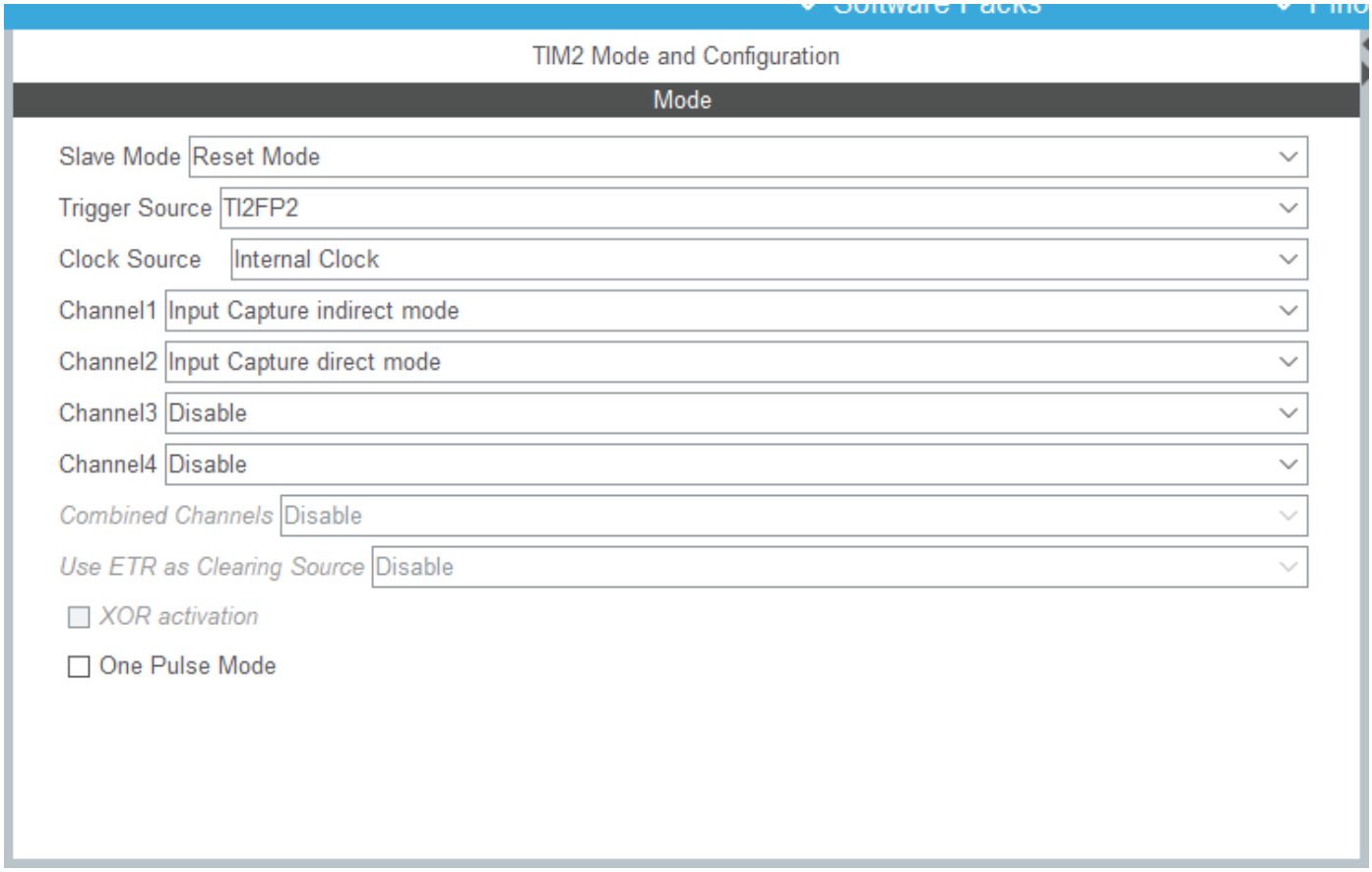
```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
  static uint8_t CaptureIndex = 0;
    if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1)
    {
      if(CaptureIndex == 0)
      {
        IC2_Value1 = HAL_TIM_ReadCapturedValue(&htim2, TIM_CHANNEL_1);
        CaptureIndex = 1;
      }
      else if(CaptureIndex == 1)
      {
        IC2_Value2 = HAL_TIM_ReadCapturedValue(&htim2, TIM_CHANNEL_1);
        if (IC2_Value2 > IC2_Value1)
        {
            uwDiffCapture = (IC2_Value2 - IC2_Value1);
        }
        else if (IC2_Value2 < IC2_Value1)
        {
             uwDiffCapture = ((0xffffffff - IC2_Value1) + IC2_Value2) + 1;
        }
        uwFrequency = HAL_RCC_GetPCLK2Freq() / uwDiffCapture;
        CaptureIndex = 0;
      }
    }
  HAL_TIM_IC_Start_IT(&htim2, TIM_CHANNEL_1);
}
```

测量占空比

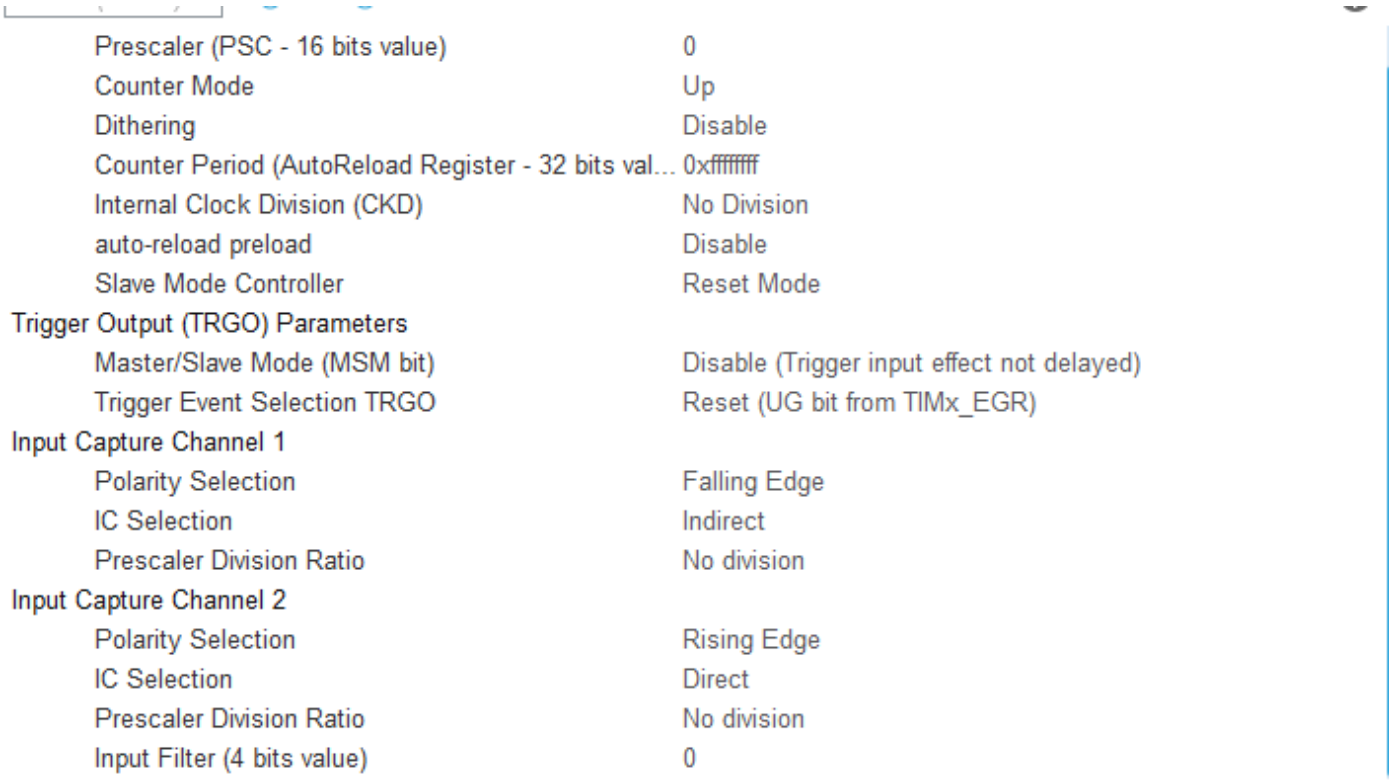
方法一：双通道捕获

使用定时器的两个通道进行捕获，第二个通道要设置成从机复位模式。

在此使用PA1进行输入捕获，PA6输出PWM波。PA1对应定时器2，通道2，在CubeMX中设置如下



通道2进行上升沿捕获，通道1进行下降沿捕获



```
__IO uint32_t uwIC2Value = 0; //捕获到的值
__IO uint32_t uwFrequency = 0;//频率
__IO float    uwDutyCycle = 0;//占空比
```

记得开启两个通道的输入捕获中断

```
HAL_TIM_PWM_Start(&htim3, TIM_CHANNEL_1);
HAL_TIM_IC_Start_IT(&htim2, TIM_CHANNEL_1);
HAL_TIM_IC_Start_IT(&htim2, TIM_CHANNEL_2);
```

在输入捕获的中断回调函数里进行处理

```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
  if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_2)
  {
    uwIC2Value = HAL_TIM_ReadCapturedValue(&htim2, TIM_CHANNEL_2);//首先捕获通道2
    if (uwIC2Value != 0)
    {
      uwDutyCycle = (float)((HAL_TIM_ReadCapturedValue(&htim2, TIM_CHANNEL_1)) * 100) / uwIC2Value;//根据第二次捕获到的值计算周期
      uwFrequency = ( HAL_RCC_GetSysClockFreq()  ) / uwIC2Value;
      //将处理好的值显示出来
      sprintf((char *)buf, "  Fre: %d       ", uwFrequency);
      LCD_DisplayStringLine(Line1, buf);
      sprintf((char *)buf, "  Duty: %.0f      ", uwDutyCycle);
      LCD_DisplayStringLine(Line2, buf);
    }
    else
    {
      uwDutyCycle = 0;
      uwFrequency = 0;
    }
  }
}
```

建议使用浮点数进行占空比和频率的运算，这样可以减小误差。

方法二：单通道捕获

使用方法一可以实现高精度的测量，但是如果使用一个通道进行捕获的话，可以通改变上下降沿分别捕获。之后通过计算得出频率与占空比。依旧使用PA1与PA6，关闭定时器2的通道1。

定义两个变量存储上下沿的捕获值，在定义一个变量表示捕获过程。

```
uint8_t CaptureNumber = 0;
uint32_t IC2ReadValue1 = 0;
uint32_t IC2ReadValue2 = 0;
uint32_t TIM2_Freq = 0;
uint32_t TIM2_Duty = 0;
```

开启定时器3的PWM输出和定时器2的输入捕获后，在中断回调函数中这么写

```
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
  if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_2)
  {
  if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_2)
    {
    if(CaptureNumber == 0)
    {
      IC2ReadValue1 = 0;
      IC2ReadValue2 = 0;
      __HAL_TIM_SET_COUNTER(&htim2, 0);
      __HAL_TIM_SET_CAPTUREPOLARITY(&htim2, TIM_CHANNEL_2,
TIM_INPUTCHANNELPOLARITY_FALLING);
      CaptureNumber = 1;
    }else if(CaptureNumber == 1){
      IC2ReadValue1 = HAL_TIM_ReadCapturedValue(&htim2, TIM_CHANNEL_2);
      __HAL_TIM_SET_CAPTUREPOLARITY(&htim2, TIM_CHANNEL_2, TIM_INPUTCHANNELPOLARITY_RISING);
      CaptureNumber = 2;
    }else if(CaptureNumber == 2){
      IC2ReadValue2 = HAL_TIM_ReadCapturedValue(&htim2, TIM_CHANNEL_2);
      CaptureNumber = 3;
    }
  }
}
```

当在主循环中检测到了捕获进程进行要到第三步的时候，计算捕获到的值。

```c
if(capture_index == 3)
{
    //将处理好的值显示出来
  if(CaptureNumber == 3)
  {
    TIM2_Freq = (uint32_t)(HAL_RCC_GetSysClockFreq() / 80 / (IC2ReadValue2 + 1));
    TIM2_Duty = (uint32_t)((IC2ReadValue1+1) * 100.0 / (IC2ReadValue2+1));
    sprintf((char *)buf, "  Fre: %d      ", TIM2_Freq);
    LCD_DisplayStringLine(Line1, buf);
    sprintf((char *)buf, "  Duty: %d      ", TIM2_Duty);
    LCD_DisplayStringLine(Line2, buf);
    CaptureNumber = 0;
  }
}
```

考点EEPROM

使用前准备

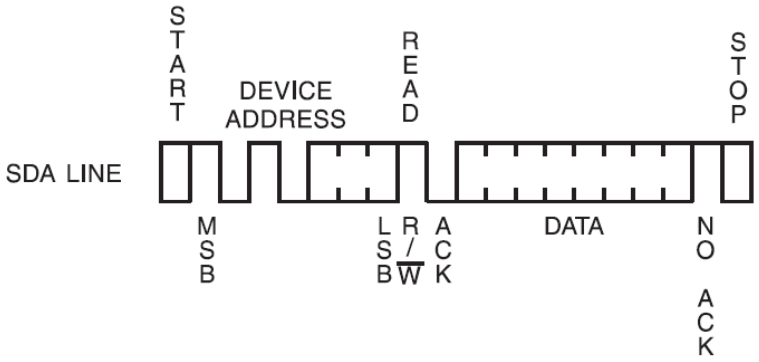
注意修改i2c.c的源代码

```
unsigned char I2CWaitAck(void)
{
    unsigned short cErrTime = 5;
    SDA_Input_Mode();
    delay1(DELAY_TIME);
    SCL_Output(1);
    delay1(DELAY_TIME);
    while(SDA_Input())
    {
        cErrTime--;
        delay1(DELAY_TIME);
        if (0 == cErrTime)
        {
            SDA_Output_Mode();
            I2CStop();
            return ERROR;
        }
    }
    SCL_Output(0);
    delay1(DELAY_TIME);
    SDA_Output_Mode();
    return SUCCESS;
}
```

由于SCL为高电平，SDA下拉是停止信号，这里的SDA状态不确定，所以SDA_Output_Mode()最好放到后面。

基本的读写操作

Figure 10. Current Address Read



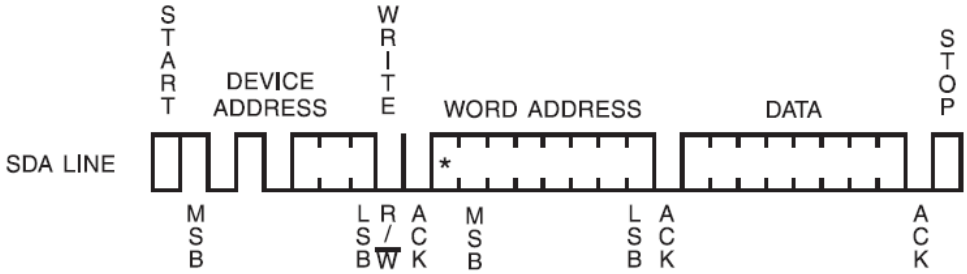

```
uint8_t ee_read(uint8_t address)
{
  uint8_t val;
  I2CStart();
  I2CSendByte(0xa0);
  I2CWaitAck();


  I2CSendByte(address);
  I2CWaitAck();

  I2CStart();
  I2CSendByte(0xa1);
  I2CWaitAck();
  val = I2CReceiveByte();
  I2CWaitAck();
  I2CStop();
  return val;

}
```

Figure 8. Byte Write



```c
void ee_write(uint8_t address, uint8_t val)
{
  I2CStart();
  I2CSendByte(0xa0);
  I2CWaitAck();

  I2CSendByte(address);
  I2CWaitAck();

  I2CSendByte(val);
  I2CWaitAck();
  I2CStop();
}
```

使用共用体进行数据读写

为了读写float这种类型的数据，可以使用共用体。

```c
union ee_float
{
  float value;
  uint8_t data[4];
}float_write, float_read;
```

确定data的空间时，可以使用sizeof()计算该类型的空间大小。

```c
for(int i = 0; i < sizeof(float); i++)
{
  ee_write(0x10 + i, float_write.data[i]);
  HAL_Delay(5);
}
```

```c
for(int i=0; i < sizeof(float); i++)
{
  float_read.data[i] = ee_read(0x10+i);
}
```

想要拿出这些数据，只需要读取共用体的value。

```
sprintf((char *)buf, "%f", float_read.value);
LCD_DisplayStringLine(Line2, buf);
```

考点 RTC

基本功能

使用RTC的时候记得开启LSE

使用RTC需要以下变量

```
RTC_TimeTypeDef sTime;
RTC_DateTypeDef sDate;
RTC_AlarmTypeDef sAlarm;
```

获取时间

```
HAL_RTC_GetTime(&hrtc, &sTime, RTC_FORMAT_BIN);
HAL_RTC_GetDate(&hrtc, &sDate, RTC_FORMAT_BIN);
```

一定要注意，**要先GetTime()再GetData()**

显示时间

```
  hour = sTime.Hours;
  min = sTime.Minutes;
  sec = sTime.Seconds;
  sprintf(lcd_buf, "   RTC: %02d:%02d:%02d   ", hour, min, sec);
  LCD_DisplayStringLine(Line3, (uint8_t *)lcd_buf);
```

暂停或运行RTC

想要停止或者开启RTC直接使能或关闭RTC时钟

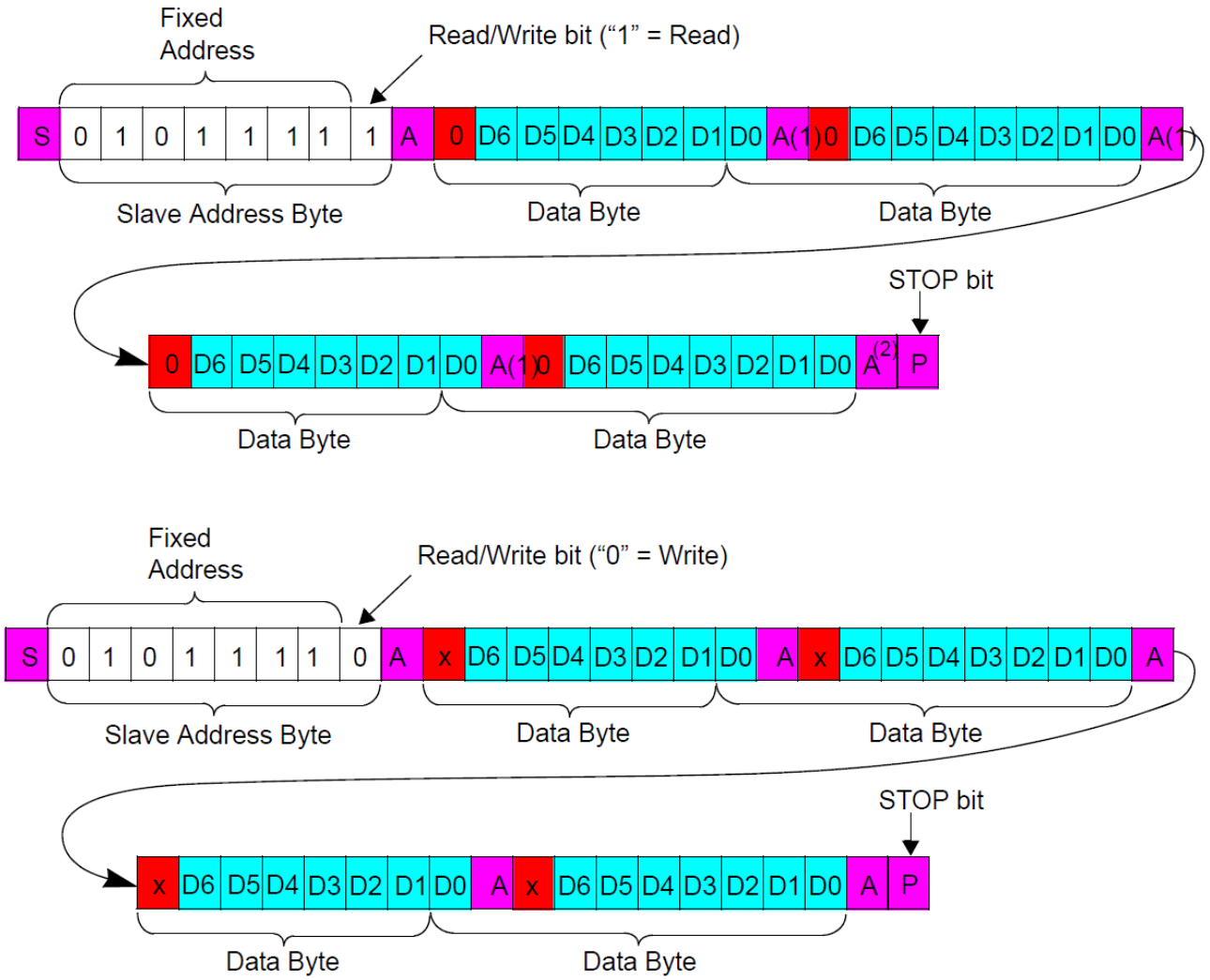
```
__HAL_RCC_RTC_ENABLE();
__HAL_RCC_RTC_DISABLE();
```

闹钟功能

```
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
  static uint16_t i = 5;
  sAlarm.AlarmTime.Hours = 0;
  sAlarm.AlarmTime.Minutes = 0;
  sAlarm.AlarmTime.Seconds = i;
  i += 5;
  sAlarm.Alarm = RTC_ALARM_A;
  // 再次启动闹钟中断事件
  if(HAL_RTC_SetAlarm_IT(hrtc, &sAlarm, RTC_FORMAT_BIN) == HAL_OK)
  {
    printf("%d % d", sAlarm.AlarmTime.Hours, sAlarm.AlarmTime.Minutes);
  }
}
```

考点MCP4017

MCP4017是G4才出现的一个模块，距离本手册该版本发布为止，还未有真题出现。下面给出基本操作。



从MCP4017的手册中可以看到写操作命令是0x5E，读操作是0x5F。

读写操作

```
void write_resistor(uint8_t value)
{
  I2CStart();
  I2CSendByte(0x5E);
  I2CWaitAck();

  I2CSendByte(value);
  I2CWaitAck();
  I2CStop();

```

```
}

uint8_t read_resistor(void)
{
  uint8_t value;
  I2CStart();
  I2CSendByte(0x5F);
  I2CWaitAck();

  value = I2CReceiveByte();
  I2CSendNotAck();
  I2CStop();

  return value;

}
```

MCP4017/18/19-104E	Min.	80000	629.921
	Typical	100000	787.402
	Max.	120000	944.882

当使用语句`write_resistor(20);`代表设置阻值为

$$20 * 0.787402k\Omega$$

写在最后

今年是第一年参加蓝桥杯，也是蓝桥杯嵌入式改革的第一年。使用了新型号的板子和HAL库。

我是来自常州信息职业技术学院应用电子技术专业的学生，接到比赛通知的时候还没有学习过STM32，在寒假期间购买了蓝桥杯的板子并开始学习STM32。HAL库相关的资料不如标准库丰富，当时看的还是野火的视频教程，而且更新的不多，可以说学起来磕磕碰碰走了不少弯路。

在准备蓝桥杯嵌入式的这些日子里，于蓝桥杯嵌入式的群里认识了不少网络好友，这些好友大多逻辑能力不错，但是不太熟悉STM32的外设，以至于准备的时候要花很长的时间用在学习外设上。我写下这份手册，希望以后这样的同学，可以借助该手册，快速上手。这样就可以把时间花在做真题上，思考和优化自己的程序上。

有的年份重外设轻逻辑，有的年份重逻辑轻外设，今年第一次参加就碰上了一个注重逻辑思考的赛题，充分暴露了自身的短板。希望后来人注意加强逻辑训练。

我将该手册分享出来，一个是为了和全国的同学交流，希望蓝桥杯嵌入式这个赛项更加有活力

。另一个是给后来人一个参考。希望我们常信应电专业可以有人在以后的赛项中拿到省一，参加国赛，弥补我的遗憾。