

# STM32Cube高效开发教程（高级篇）

## 第4章 进程间通信与消息队列

---

王维波

中国石油大学（华东）控制科学与工程学院

# STM32Cube高效开发教程（高级篇）

作者：王维波，鄢志丹，王钊

人民邮电出版社

2022年2月出版

如果有读者需要本书课件的PPT版本用于备课，可以给作者发邮件免费获取，并可加入专门的教学和技术交流QQ群

邮箱：[wangwb@upc.edu.cn](mailto:wangwb@upc.edu.cn)



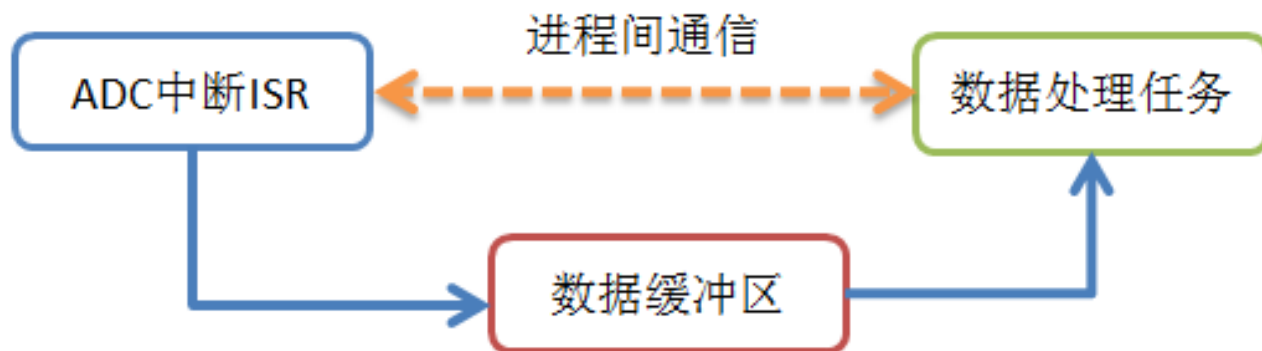
4.1 进程间通信

4.2 队列的特点和基本操作

4.3 队列使用示例

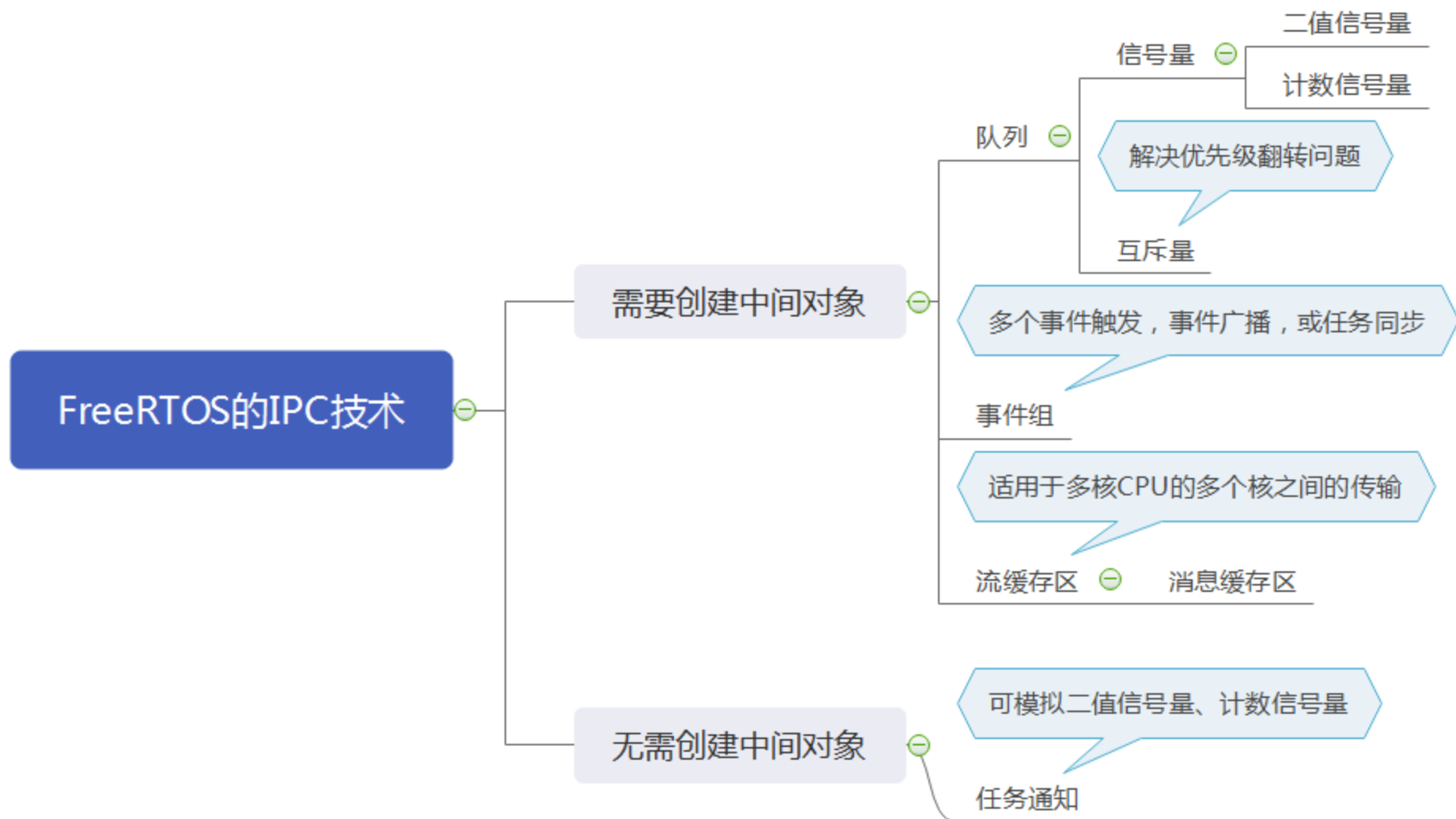
## 任务和ISR统称为进程（process）

任务与任务之间，或任务与ISR之间有时需要进行通讯或同步，这称为进程间通信（IPC，Inter-process communication）



传统编程中会使用标志变量，不断地查询标志变量的状态

FreeRTOS提供了完善的进程间通信技术，包括队列、信号量、互斥量等，与C++的多线程同步类似。



## (1) 队列 (Queue)

队列就是一个缓冲区，用于在进程间传递少量的数据，所以也称消息队列。

## (2) 信号量 ( Semaphore )

分为二值信号量 (Binary Semaphore) 和计数信号量 (Counting Semaphore)。二值信号量使用于进程间同步，计数信号量一般用于共享资源的管理。

## (3) 互斥量 ( Mutex )

分为互斥量 (Mutex) 和递归互斥量 (Recursive Mutex)。  
互斥量具有**优先级继承机制**，可以减轻**优先级翻转问题**

## （4）事件组（Event Group）

事件组适用于多个事件触发一个或多个任务的运行，可以实现事件的广播，还可以实现多个任务的同步运行。

## （5）流缓冲区（Stream Buffer）和消息缓冲区（Message Buffer）

是FreeRTOS V10版本新增的功能，是一种优化的进程间通信机制，专门应用于只有一个写入者（writer）和一个读取者（reader）的场景，还可用于多核CPU的两个内核之间高效传输数据。

## （6）任务通知（Task Notification）

使用任务通知不需要创建任何中间对象，可以直接从任务向任务，或ISR向任务发送通知，传递一个通知值。

任务通知可以模拟二值信号量、计数信号量，或长度为1的消息队列，使用任务通知通常效率更高，消耗内存更少。



## 4.2 队列的特点和基本操作

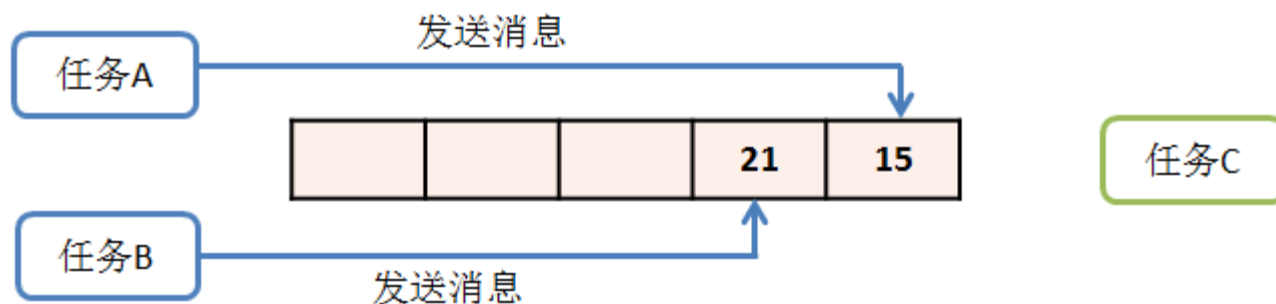
4.2.1 队列的创建和存储

4.2.2 向队列写入数据

4.2.3 从队列读取数据

4.2.4 队列操作相关函数

设置队列的每个单元是uint16\_t类型



- ◆ 队列的创建和存储
- ◆ 向队列写入数据
- ◆ 从队列读取数据

## 4.2.1 队列的创建和存储

队列创建时被分配固定个数的存储单元，每个存储单元存储固定大小的数据，进程间传递的数据就保存在队列的存储单元里。

函数xQueueCreate()以动态分配内存的方式创建队列，队列需要用的存储空间由FreeRTOS从堆空间自动分配。

```
#define xQueueCreate( uxQueueLength, uxItemSize )  
    xQueueGenericCreate( ( uxQueueLength ), ( uxItemSize ),  
        ( queueQUEUE_TYPE_BASE ) )
```

函数xQueueGenericCreate()是创建队列、信号量、互斥量等对象的通用函数。函数原型是：

```
QueueHandle_t xQueueGenericCreate( const UBaseType_t uxQueueLength, const  
    UBaseType_t uxItemSize, const uint8_t ucQueueType )
```

```
QueueHandle_t xQueueGenericCreate( const UBaseType_t uxQueueLength,  
                                   const UBaseType_t uxItemSize, const uint8_t ucQueueType )
```

- **uxQueueLength**表示队列的长度，也就是存储单元的个数
- **uxItemSize**是每个存储单元的字节数
- **ucQueueType**表示创建的对象类型，有以下几种常数取值：

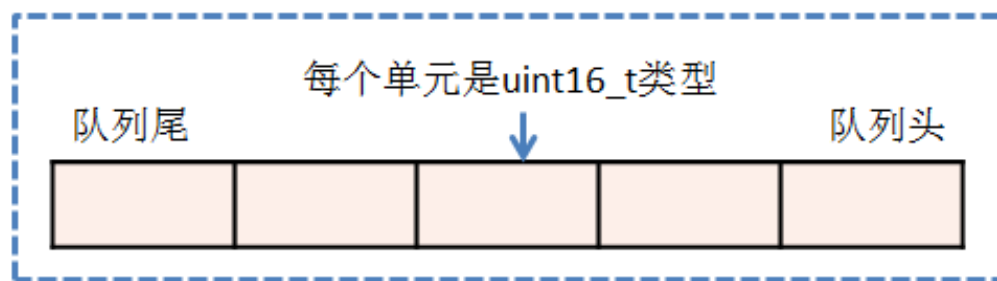
```
#define queueQUEUE_TYPE_BASE      ( ( uint8_t ) 0U )      //队列  
#define queueQUEUE_TYPE_SET      ( ( uint8_t ) 0U )      //队列集合  
#define queueQUEUE_TYPE_MUTEX    ( ( uint8_t ) 1U )      //互斥量  
#define queueQUEUE_TYPE_COUNTING_SEMAPHORE ( ( uint8_t ) 2U ) //计数信号量  
#define queueQUEUE_TYPE_BINARY_SEMAPHORE  ( ( uint8_t ) 3U ) //二值信号量  
#define queueQUEUE_TYPE_RECURSIVE_MUTEX   ( ( uint8_t ) 4U ) //迭代互斥量
```

函数xQueueGenericCreate()的返回值是QueueHandle\_t类型，是所创建队列的句柄

调用函数xQueueCreate()的示例如下：

```
Queue_KeysHandle = xQueueCreate (5, sizeof(uint16_t));
```

这行代码创建了一个具有5个存储单元的队列，每个单元占用sizeof(uint16\_t)个字节，也就是2个字节。



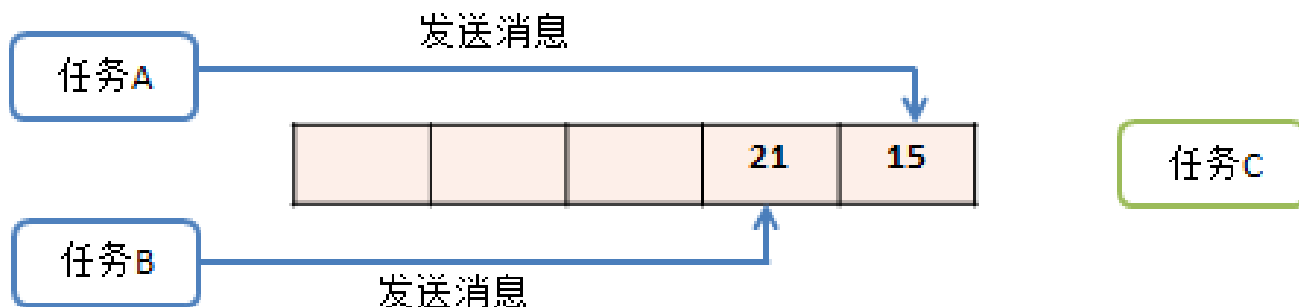
- ◆ 队列的存储单元可以设置任意大小，可以存储任意数据类型
- ◆ 队列存储数据采用数据复制的方式

数据项比较大（比如数组），复制数据会占用较大空间，怎么办？

传递数据的指针，通过指针再去读取原始数据

## 26.2.2 向队列写入数据

一个任务或ISR向队列写入数据称为**发送消息**。队列是一个共享的存储区域，可以被多个进程写入，也可以被多个进程读取。



**xQueueSendToBack():** 向队列后端写入数据（FIFO模式）

**xQueueSendToFront():** 向队列前段写入数据（LIFO模式）

它们都是宏函数，调用了函数xQueueGenericSend()

## 函数xQueueSendToBack()的定义如下----FIFO模式

```
#define xQueueSendToBack( xQueue, pvItemToQueue, xTicksToWait )  
    xQueueGenericSend( ( xQueue ), ( pvItemToQueue ), ( xTicksToWait ),  
        queueSEND_TO_BACK )
```

## 函数xQueueSendToFront()的定义如下----LIFO模式

```
#define xQueueSendToFront( xQueue, pvItemToQueue, xTicksToWait )  
    xQueueGenericSend( ( xQueue ), ( pvItemToQueue ), ( xTicksToWait ),  
        queueSEND_TO_FRONT )
```

这两个函数在队列未满时能正常向队列写入数据，函数返回值为pdTRUE；如果队列已满，这两个函数不能再向队列写入数据，函数返回值为errQUEUE\_FULL

## 函数xQueueGenericSend()的定义如下

```
 BaseType_t xQueueGenericSend( QueueHandle_t xQueue, const void * const  
                               pvItemToQueue, TickType_t xTicksToWait, const BaseType_t xCopyPosition )
```

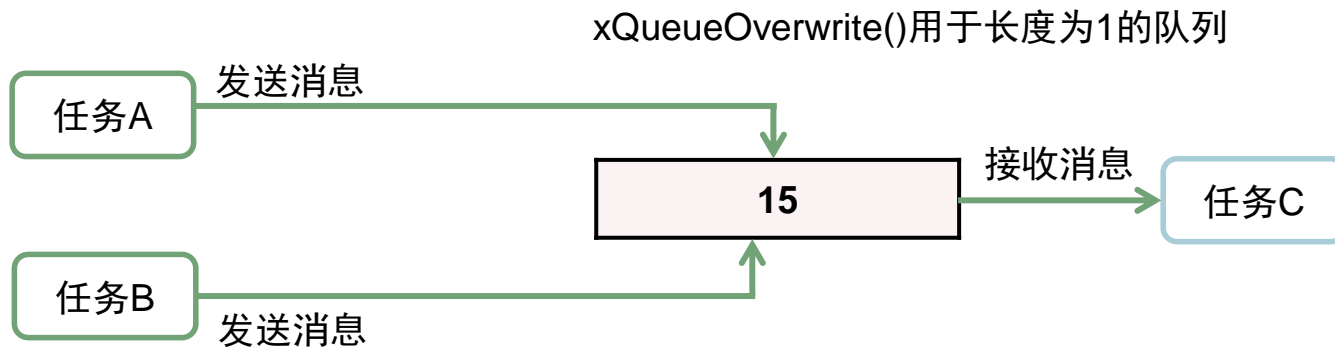
- **xQueue** 是所操作队列的句柄
- **pvItemToQueue** 是需要向队列写入的一个项数据
- **xTicksToWait** 是阻塞方式等待队列出现空闲单元的节拍数，0、**portMAX\_DELAY**、其他数
- **xCopyPosition** 表示写入队列的位置，有3种常数定义

```
#define queueSEND_TO_BACK    ( ( BaseType_t ) 0 ) //写入后端，FIFO方式  
#define queueSEND_TO_FRONT  ( ( BaseType_t ) 1 ) //写入前段，LIFO  
#define queueOVERWRITE       ( ( BaseType_t ) 2 ) //尾端覆盖，在队列满时
```



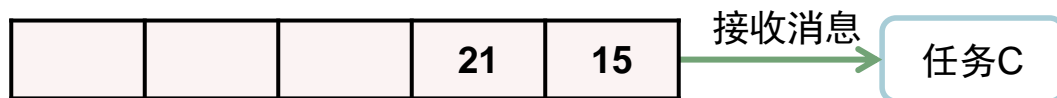
还有一个函数xQueueOverwrite()也可以用于向队列写入数据，但是这个函数只用于队列长度为1的队列，在队列已满时，它会覆盖队列原来的数据。

```
#define xQueueOverwrite( xQueue, pvItemToQueue )  
    xQueueGenericSend( ( xQueue ), ( pvItemToQueue ), 0,  
        queueOVERWRITE )
```



## 26.2.3 从队列读取数据

可以在任务或ISR里读取队列的数据，称为**接收消息**。总是从队列头读取数据，读出后删除这个单元的数据，后面的数据前移



函数xQueueReceive()的函数原型如下：

```
BaseType_t xQueueReceive( QueueHandle_t xQueue, void * const pvBuffer,  
    TickType_t xTicksToWait );
```

- **xQueue** 是所操作的队列句柄；
- **pvBuffer** 是从队列读出数据保存的缓冲区；
- **xTicksToWait** 是阻塞方式等待节拍数

函数的返回值是pdTRUE或pdFALSE

## 26.2.4 队列操作相关函数

### 1. 队列管理

函数名	功能描述
<code>xQueueCreate()</code>	动态分配内存方式创建一个队列
<code>xQueueCreateStatic()</code>	静态分配内存方式创建一个队列
<code>xQueueReset()</code>	将队列复位为空的状态，队列内的所有数据都被丢弃
<code>vQueueDelete()</code>	删除一个队列，也可以用于删除一个信号量

## 2. 获取队列信息

函数名	功能描述
pcQueueGetName()	获取队列的名称，也就是创建队列时设置的队列名称字符串
vQueueSetQueueNumber()	为队列设置一个编号，这个编号由用户设置并使用
uxQueueGetQueueNumber()	获取队列的编号
uxQueueSpacesAvailable()	获取队列剩余空间个数，也就是还可以写入的消息个数
uxQueueMessagesWaiting()	获取队列中等待读取的消息个数
uxQueueMessagesWaitingFromISR()	uxQueueMessagesWaiting()的ISR版本
xQueueIsQueueEmptyFromISR()	查询队列是否为空，返回值为pdTRUE表示队列为空
xQueueIsQueueFullFromISR()	查询队列是否满了，返回值为pdTRUE表示队列满了

### 3. 写入消息

函数名	功能描述
xQueueSend()	写一个消息到队列的后端（FIFO方式），这个函数是早期版本
xQueueSendFromISR()	xQueueSend()的ISR版本
xQueueSendToBack()	与xQueueSend()功能完全相同，建议使用这个函数
xQueueSendToBackFromISR()	xQueueSendToBack()的ISR版本
xQueueSendToFront()	写一个消息到队列的前端（LIFO方式）
xQueueSendToFrontFromISR()	xQueueSendToFront()的ISR版本
xQueueOverwrite()	这个函数只用于长度为1的队列，如果队列已满，会覆盖原来的数据
xQueueOverwriteFromISR()	xQueueOverwrite()的ISR版本

## 4. 读取消息

函数名	功能描述
<code>xQueueReceive()</code>	从队列中读取一个消息，读出后删除队列中的这个消息
<code>xQueueReceiveFromISR()</code>	<code>xQueueReceive()</code> 的ISR版本
<code>xQueuePeek()</code>	从队列中读取一个消息，读出后不删除队列中的这个消息
<code>xQueuePeekFromISR()</code>	<code>xQueuePeek()</code> 的ISR版本

## 4.3 队列使用示例

4.3.1 示例功能和CubeMX项目设置

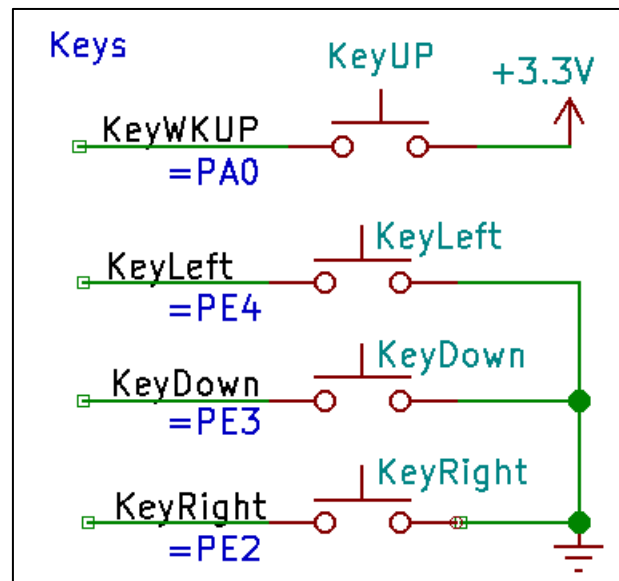
4.3.2 初始化代码分析

4.3.3 实现用户功能

## 4.3.1 示例功能和CubeMX项目设置

### 示例功能

- ◆ 创建一个队列和两个任务
- ◆ 一个任务查询4个按键的状态，某个按键被按下时就向队列中写入代表此按键的值
- ◆ 另外一个任务负责读取队列的数据，根据队列里的按键值在LCD上向上、下、左、右四个方向画线

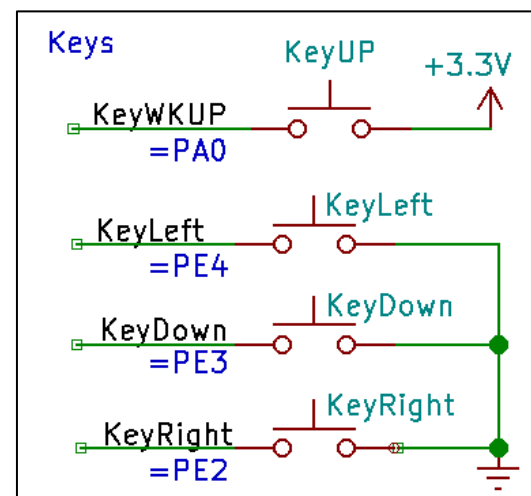


4个按键的电路连接



## (1) 按键输入引脚的设置

Pin Name	User Label	GPIO mode	GPIO Pull-up/Pull-down
PE3	KeyDown	Input mode	Pull-up
PE4	KeyLeft	Input mode	Pull-up
PE2	KeyRight	Input mode	Pull-up
PA0-WKUP	KeyUp	Input mode	Pull-down



## (2) FreeRTOS设置

Tasks

Task Name	Priority	Stack Size...	Entry Function	Allocation	Code ...	Para...	Buffe...	Cont...
Task_Draw	osPriorityBelowNormal	128	AppTask_Draw	Dynamic	Default	NULL	NULL	NULL
Task_ScanKeys	osPriorityNormal	128	AppTask_ScanKeys	Dynamic	Default	NULL	NULL	NULL

AddDelete

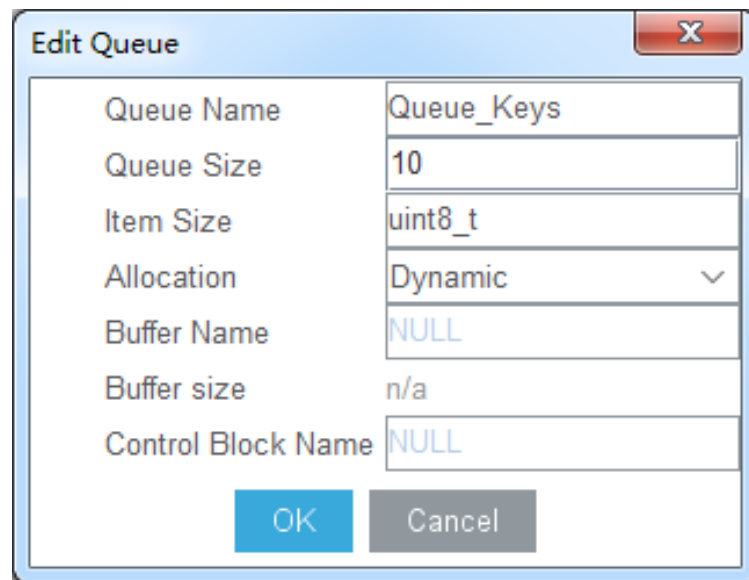
Queues

Queue Name	Queue Size	Item Size	Allocation	Buffer Name	Control Block Name
Queue_Keys	10	uint8_t	Dynamic	NULL	NULL

AddDelete

## 队列的属性

- Queue Name, 队列名称
- Queue Size, 队列大小
- Item Size, 每个项的大小。标准数据类型, 如uint8\_t、uint16\_t等, 或可以直接填写字节数
- Allocation, 内存分配方式
- Buffer Name, 缓冲区名称
- Buffer Size, 缓冲区大小
- Control Block Name, 控制块名称



The screenshot shows a dialog box titled "Edit Queue" with a close button (X) in the top right corner. The dialog contains several input fields and a dropdown menu, each with a label on the left and a value in a text box on the right. The values are: Queue Name: Queue\_Keys; Queue Size: 10; Item Size: uint8\_t; Allocation: Dynamic (with a dropdown arrow); Buffer Name: NULL; Buffer size: n/a; and Control Block Name: NULL. At the bottom right, there are two buttons: "OK" (blue) and "Cancel" (grey).

Property	Value
Queue Name	Queue_Keys
Queue Size	10
Item Size	uint8_t
Allocation	Dynamic
Buffer Name	NULL
Buffer size	n/a
Control Block Name	NULL

## 4.3.2 初始化代码分析

### 1.主程序

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    /* Initialize all configured peripherals */
    MX_GPIO_Init();          //GPIO初始化
    MX_FSMC_Init();

    /* USER CODE BEGIN 2 */
    TFTLCD_Init();           //LCD 初始化
    LCD_ShowString(10, 10, (uint8_t *)"Demo4_1:Using a Queue");
    /* USER CODE END 2 */

    osKernelInitialize(); //内核初始化
    MX_FREERTOS_Init(); //FreeRTOS初始化，创建队列和任务
    osKernelStart(); //启动内核
    while (1)
    {
    }
}
```

## 2. 创建任务和队列

```
/* 文件: freertos.c -----*/
#include "FreeRTOS.h"
#include "task.h"
#include "main.h"
#include "cmsis_os.h"

/* Private variables -----*/
/* 任务Task_Draw相关定义 */
osThreadId_t Task_DrawHandle; //任务Task_Draw的句柄变量
const osThreadAttr_t Task_Draw_attributes = { //任务Task_Draw的属性
    .name = "Task_Draw",
    .priority = (osPriority_t) osPriorityBelowNormal,
    .stack_size = 128 * 4
};

/* 任务 Task_ScanKeys 相关定义 */
osThreadId_t Task_ScanKeysHandle; //任务 Task_ScanKeys的句柄变量
const osThreadAttr_t Task_ScanKeys_attributes = { //任务 Task_ScanKeys的属性
    .name = "Task_ScanKeys",
    .priority = (osPriority_t) osPriorityNormal,
    .stack_size = 128 * 4
};
```

```
/* 队列Queue_Keys相关定义 */
```

```
osMessageQueueId_t Queue_KeysHandle;          //队列Queue_Keys的句柄变量  
const osMessageQueueAttr_t Queue_Keys_attributes = { //队列Queue_Keys的属性  
    .name = "Queue_Keys"  
};
```

```
void MX_FREERTOS_Init(void)
```

```
{  
    /* 创建队列Queue_Keys */  
    Queue_KeysHandle = osMessageQueueNew (10, sizeof(uint8_t),  
    &Queue_Keys_attributes);
```

```
    /* 创建任务 Task_Draw */
```

```
    Task_DrawHandle = osThreadNew(AppTask_Draw, NULL, Task_Draw_attributes);
```

```
    /* 创建任务Task_ScanKeys */
```

```
    Task_ScanKeysHandle = osThreadNew(AppTask_ScanKeys, NULL,  
    &Task_ScanKeys_attributes);
```

```
}
```

使用函数`osMessageQueueNew()`创建队列，这是CMSIS RTOS标准接口函数，它内部会根据队列的属性设置自动调用函数`xQueueCreate()`或`xQueueCreateStatic()`创建队列

结构体`osMessageQueueAttr_t`也是在文件`cmsis_os2.h`中定义的，其定义如下，各成员变量的作用见注释。

```
typedef struct {  
    const char    *name;    //消息队列的字符串名称  
    uint32_t      attr_bits; //属性位  
    void          *cb_mem;  //控制块的存储空间  
    uint32_t      cb_size;  //控制块的存储空间大小，单位：字节  
    void          *mq_mem;  //数据存储空间  
    uint32_t      mq_size;  //数据存储空间大小，单位：字节  
} osMessageQueueAttr_t;
```

### 4.3.3 实现用户功能

示例功能：在任务Task\_ScanKeys中扫码按键，将按键代码发送到消息队列，任务Task\_Draw读取队列中的按键代码后在LCD上画线

在freertos.c中添加定义，编写任务函数代码

```
/* Private variables -----*/  
/* USER CODE BEGIN Variables */  
const uint8_t      KeyCodeLeft      =0x01;    //按键代码KeyLeft  
const uint8_t      KeyCodeRight     =0x02;    //按键代码KeyRight  
const uint8_t      KeyCodeUp        =0x03;    //按键代码KeyUp  
const uint8_t      KeyCodeDown      =0x04;    //按键代码KeyDown  
  
uint16_t  curScreenX=100;                //LCD当前X  
uint16_t  curScreenY=260;                //LCD当前Y  
uint16_t  lastScreenX=100;               // LCD前一步的X  
uint16_t  lastScreenY=260;               // LCD前一步的Y  
/* USER CODE END Variables */
```

两个任务函数的代码较长，看源程序。

## 1. 扫描按键和发送消息

读取到一个按键被按下后调用函数xQueueSendToBack()将按键代码写入队列，如

```
keyState= HAL_GPIO_ReadPin(GPIOE, GPIO_PIN_2); //PE2=KeyRight, 上拉
if (keyState==GPIO_PIN_RESET) //KeyRight 是低输入有效
{
    err= xQueueSendToBack(Queue_KeysHandle, &KeyCodeRight,
                          pdMS_TO_TICKS(100));
    vTaskDelay(delayAfterPress); //延时，去除抖动，同时让任务调度执行
}
```

调用函数vTaskDelay()延时200ms，这是用软件延时的方式消除按键抖动的影响，同时又使任务Task\_ScanKeys进入阻塞状态，让低优先级的任务Task\_Draw可以进入运行状态，及时读取队列里的消息并处理。

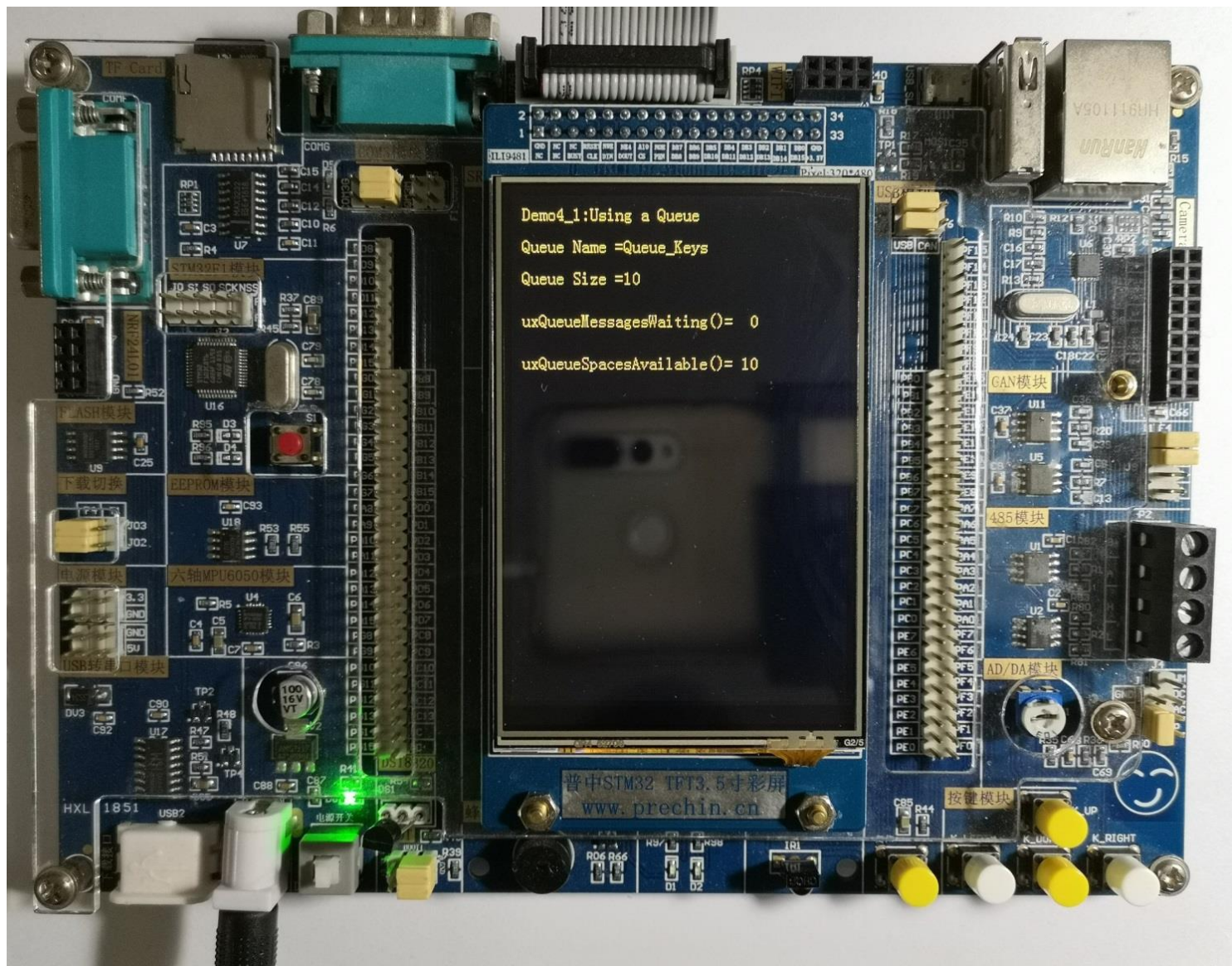


## 2. 读取消息并画线

使用函数xQueueReceive()读取队列中的消息

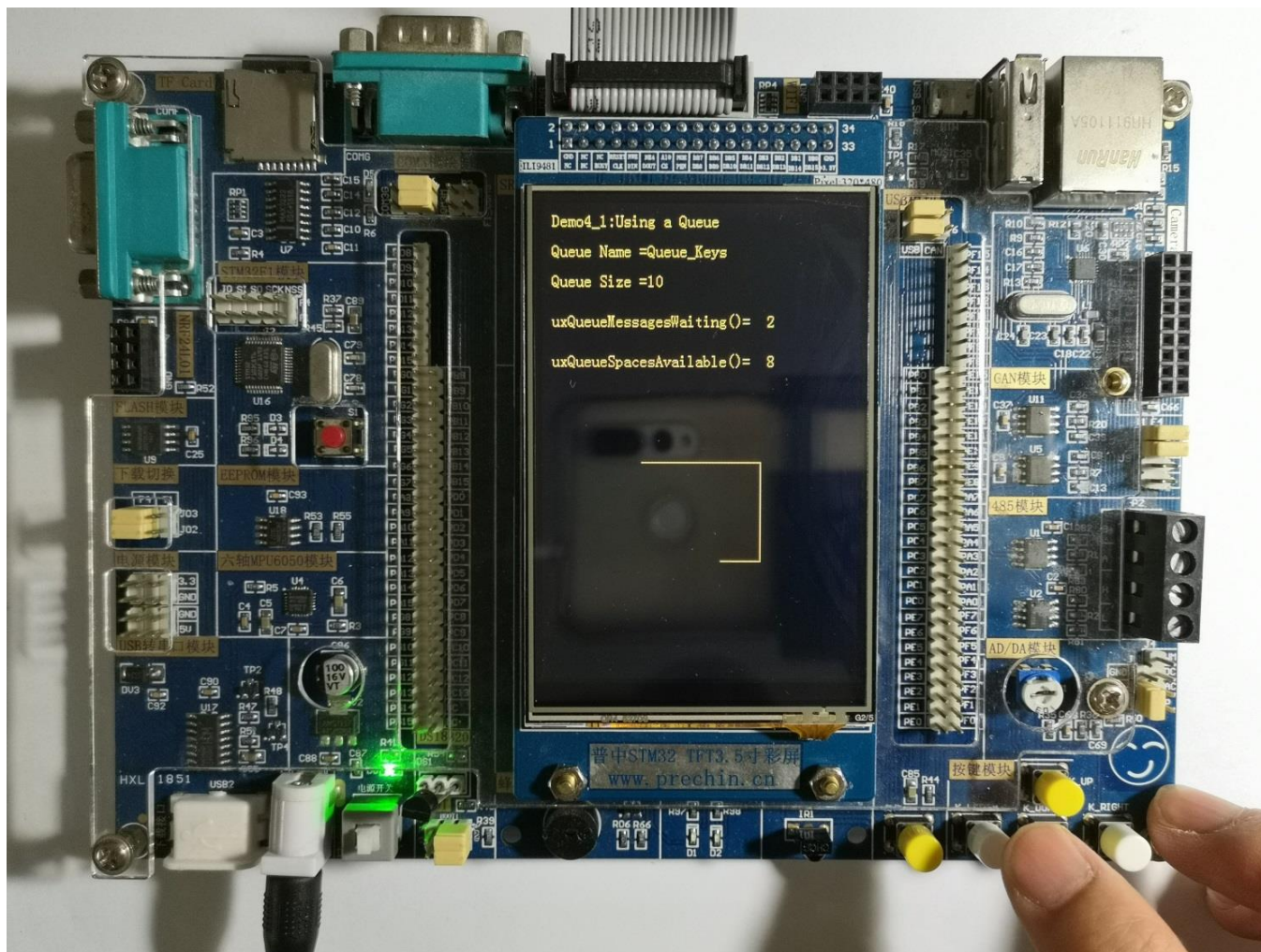
```
BaseType_t result=xQueueReceive(Queue_KeysHandle, &keyCode,  
                                pdMS_TO_TICKS(100));
```

- ◆ 如果队列中没有消息，任务Task\_Draw就会进入阻塞状态等待最多100ms。如果队列中有了消息，就会将读取的消息数据保存到变量keyCode中。
- ◆ 如果函数xQueueReceive()的返回值不是pdTRUE，表示超过了阻塞等待时间仍然没有消息可读。
- ◆ for循环的最后调用函数vTaskDelay()延时400ms，是为了人为的造成比较大的延时。这样，在快速连续按下按键时，会看到LCD上待读取消息条数可以达到2或3。



复位后显示界面，队列有10个存储单元





按上、下、左、右键，LCD上会移动画线，并显示待读取消息条数，和队列剩余存储单元个数。任务Task\_Draw里加了延时，所以按键较快时，待读取消息个数会大于1

# 练习任务

1. 看教材，练习本章的示例。
2. 使用中断方式读取4个按键，实现与本章示例Demo4\_1相同的功能。注意，在ISR函数中只能调用中断级的FreeRTOS API函数。