Advanced Card Systems Ltd.
Card & Reader Technologies

ACR89U-A2
手持式
智能卡读写器

应用程序编程接口 V1.00

info@acs.com.hk
www.acs.com.hk

目录

图目录

表目录

1.0. 简介

ACR89U-A2 终端配备 32 位 CPU，运行嵌入式 FreeRTOS 内核。FreeRTOS 内核是专门为小型嵌入式系统设计的可伸缩型实时内核， 具有开源、便携、免费下载、免费部署软件等特点。商业用途中，FreeRTOS 不会泄露私人源代码。它的代码主要采用 C 语言编写，可移植性非常强。

本文档介绍了 ACR89U 专有的用于开发脱机应用程序的 API 命令。应用软件开发人员可以用这些 API 开发智能卡应用。

1.1. 使用范围和限制

本文档详细介绍了智能卡读写器按键和显示屏的相关命令，以及 ACR89 的 FreeRTOS 特性。

1.2. 参考

关于 FreeRTOS 软件环境的详细信息，请访问：

http://www.freertos.org/

2.0. 通用数据类型

数据类型	大小	数字有效范围
UINT8	1 字节	0 至 255
UINT16	2 字节	0 至 65535
UINT32	4 字节	0 至 4294967295
UINT64	8 字节	0 至 18446744073709551615
INT8	1 字节	-128 至 127
INT16	2 字节	-32768 至 32767
INT32	4 字节	-2147483648 至 2147483647
INT64	8 字节	-9223372036854775808 至 9223372036854775807
REAL32	4 字节	1.175e-38 至 3.403e+38（标准数字）
REAL64	8 字节	2.225e-308 至 1.798e+308（标准数字）
BOOLEAN	1 字节	TRUE/FALSE
UCHAR	1 字节	0 至 255
SCHAR	1 字节	-128 至 127

图1：通用数据类型

处理 64 位整数数据类型需要后缀 LL 或 ll（INT64 类型）或 ULL 或 ull（UINT64 类型）。如果缺少该后缀，编译程序不能识别长型常量，会报错。

例：INT64 ll_val;

ll_val = 1234567812345678h;

- 警告：整数常量值超出其类型

Ll_val = 1234567812345678LLh;

- OK

3.0. 智能卡的API函数

3.1. 固件版本记录

- Kiwi 2012-07-13 v1.3
- HW-D2-02-00 FreeRTOS V7.0.1

3.1.1. 硬件代码：HW-AA-BB-CC

其中：

- AA：大版本
  - D1 = ACR89 V1-V3 PCBA
  - D2 = ACR89 V4-V5 PCBA
- BB：配置
  - 01 = ACR89U-A1（基本）
  - 02 = ACR89U-A2（非接触式，支持 FeliCa）
  - 04 = ACR89U-A3（非接触式）
- CC：小版本

3.1.2. 产品固件代码：XYYY

- X：配置
- A = ACR89U（标准）
- B = ACR89U-CL（非接触式）
- YYY：发布版本代码

3.2. 数据结构

3.2.1. SCARD_MSG_TYPE

```
[SCard_Msg.h]
typedef struct
{
      UINT8   ucMessage:5;
      UINT8   ucData:3;
} SCARD_MSG_TYPE;

      #define SCARD_MSG_UNKNOWN       0x00
      #define SCARD_MSG_CARDINSERTED  0x01
      #define SCARD_MSG_CARDREMOVED   0x02
```

用于 Scard_Manager_Msg_Receive。

数据成员	值	说明
ucMessage	位域 0 至 2	智能卡消息类型： 0 - SCARD_MSG_UNKNOWN （未定义消息） 1 - SCARD_MSG_CARDINSERTED （卡片已插入） 2 - SCARD_MSG_CARDINSERTED （卡片已移除）
ucData	位域 0 或 1	消息来源的智能卡插槽的索引 0 - 第一个插槽 1 - 第二个插槽

表1 ：SCARD _MSG_TYPE Data 数据结构

3.3. 函数

3.3.1. SCard_Manager_Msg_Receive

此函数用于接收智能卡消息，如果接收失败则反复尝试，直到超时。

```
[SCard_Manager.h]

BOOLEAN SCard_Manager_Msg_Receive (
SCARD_MSG_TYPE *pMsgBuffer,
portTickType TimeOut );
```

参数：

pMsgBuffer [out] 待输出的智能卡消息。

TimeOut [in] 接收智能卡消息的等待时长 [0 - portMAX_DELAY 毫秒]。阻塞时间指定为 portMAX_DELAY 会导致任务无限期阻塞（不会超时）。

返回值：

BOOLEAN

此函数返回 TRUE/FALSE，分别表示接收智能卡消息成功/失败。

3.3.2. SCard_Manager_Select Card

此函数用于选取主卡槽。

```
[SCard_Manager.h]

void SCard_Manager_SelectCard (
      UINT8 ucCard );
```

参数：

ucCard [in] 主卡槽的索引 [0-4]。

3.3.3. SCard_Manager_CardOn

此函数用于给主卡槽上电。

```
[SCard_Manager.h]

UINT8 SCard_Manager_CardOn (
      BOOLEAN bAutoVoltage,
      UINT8   ucVoltage,
      UCHAR   *pucReceiveBuffer,
      UINT16  *pusReceiveSize );
```

参数：

bAutoVoltage [in] TRUE -- 自动检测卡片的工作电压

FALSE -- 采用固定的卡片工作电压

ucVoltage [in] 如果 bAutoVoltage 的值是 FALSE，则用 ucVoltage 设置卡片的固定工作电压。

[CVCC_1_8_VOLT -- 1.8 V，CVCC_3_VOLT – 3 V，CVCC_5_VOLT – 5 V]

pucReceiveBuffer [out] 待输出的 ATR 数据。

pusReceiveSize [out] 输出的 ATR 数据大小[字节]。

返回值：

UINT8 **此函数返回操作结果，**

[SLOT_NO_ERROR -- 成功，

SLOTERROR_BAD_LENGTH -- 数据长度错误，

SLOTERROR_BAD_SLOT -- 无效卡槽，

SLOTERROR_ICC_MUTE -- 卡片反应超时，

SLOTERROR_XFR_PARITY_ERROR -- 数据奇偶校验错误，

SLOTERROR_XFR_OVERRUN -- 数据传输超时，

SLOTERROR_HW_ERROR -- 硬件错误，

SLOTERROR_BAD_ATR_TS -- ATR 的 TS 错误]。

3.3.4. SCard_Manager_CardOff

此函数用于给主卡槽下电。

```
[SCard_Manager.h]
BOOLEAN SCard_Manager_CardOff (
   void );
```

返回值：

BOOLEAN **此函数返回 TRUE/FALSE，分别表示给主卡槽下电操作成功/失败。**

3.3.5. SCard_Manager_SendBlock

此函数用于发送 APDU 命令给智能卡并接收智能卡数据。

```
[SCard_Manager.h]
UINT8 SCard_Manager_SendBlock (
      UCHAR  *pucCmdBlockBuffer,
      UCHAR  *pucResBlockBuffer,
      UINT16 *pusBufferSize );
```

参数：

pucCmdBlockBuffer [in] 输入的 APDU，此 APDU 被发送给卡片。

pucResBlockBuffer [out] 卡片输出数据。

pusBufferSize [in&out] 输入的 APDU 大小和输出的数据大小［字节］

返回值：

UINT8 此函数返回操作结果，

[SLOT_NO_ERROR -- 成功，

SLOTERROR_BAD_LENGTH -- 数据长度错误，

SLOTERROR_ICC_MUTE -- 卡片反应超时，

SLOTERROR_XFR_PARITY_ERROR -- 数据奇偶校验错误，

SLOTERROR_HW_ERROR -- 硬件错误，

SLOTERROR_ICC_CLASS_NOT_SUPPORTED -- 功能错误，

SLOTERROR_PROCEDURE_BYTE_CONFLICT -- 过程字节错误]。

4.0. 读写器的API函数

4.1. 电池的API函数

4.1.1. 数据结构

4.1.2. 函数

4.1.2.1. Battery_GetMilliVolt

此函数用于获取电池电压。

```
[Battery.h]

UINT32 Battery_GetMilliVolt (
      void );
```

返回值：

UINT32 此函数用于返回电池电压值 [mV]。

4.1.2.2. Battery_GetPercent

此函数用于获取电池电量百分比。

```
[Battery.h]

UINT8 Battery_GetPercent (
void );
```

返回值：

UINT8 此函数用于返回电池电量百分比的值[0-100]。

4.1.2.3. Battery_WaitChargeStateChangeMsg

此函数用于 TimeOut 时限内确认电池充电状态变化。

```
[Battery.h]

BOOLEAN Battery_WaitChargeStateChangeMsg (
         portTickType TimeOut,
         BOOLEAN *pbChargeState );
```

参数：

TimeOut [in] 充电状态变化的等待时长[0 - portMAX_DELAY 毫秒]。阻塞时间指定为 portMAX_DELAY 会导致任务无限期阻塞（不会超时）。

pucReceiveBuffer [out] 待输出的充电状态。如果此函数返回 TRUE，则*pbChargeState 输出新状态; 如果此函数返回 FALSE，则*pbChargeState 输出当前状态。

返回值：

BOOLEAN **此函数返回 TRUE/FALSE，分别表示电池充电状态有/无变化。**

4.2. 蜂鸣器的API函数

4.2.1. 数据结构

4.2.1.1. Buzzer_ScriptDataType

```
[Buzzer_Msg.h]

struct Buzzer_Script
{
      UINT8 BuzzerTime:7;
      UINT8 BuzzerOn:1;
};
typedef struct Buzzer_Script Buzzer_ScriptDataType;
```

用于 Buzzer_Msg_SendScript。

例：

```
const Buzzer_ScriptDataType Buzzer_SampleScript1 [] =
{
      {1, TRUE},  // 打开蜂鸣器 100ms
      {2, FALSE}, // 关闭蜂鸣器 200ms
      {3, TRUE},  // 打开蜂鸣器 300ms
      ...
      {0, FALSE}  // 强制结束脚本
};
```

数据成员	值	说明
BuzzerTime	位域 0 - 127	蜂鸣器开/关状态的时长： 该值是 100 ms 的倍数。
BuzzerOn	位域 TRUE 或 FALSE	蜂鸣器开/关状态： TRUE – 蜂鸣器开 FALSE – 蜂鸣器关

表2：Buzzer_ ScriptDataType 数据结构

4.2.2. 函数

4.2.2.1. Buzzer_Msg_SendScript

此函数用于发送蜂鸣器脚本给蜂鸣器驱动。

```
[Buzzer_Msg.h]

BOOLEAN Buzzer_Msg_SendScript (
        Buzzer_ScriptDataType const* const Script );
```

参数：

Script [in] 脚本数据。如果此参数是本地变量，它的数据类型应为“静态常量”，以保证编译安全性。本参数推荐使用变量 global const。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示蜂鸣器脚本发送成功/失败。

4.2.2.2. Buzzer_Msg_IsPlaying

此函数用于确认蜂鸣器是否正在运行脚本。

```
[Buzzer_Msg.h]

BOOLEAN Buzzer_Msg_IsPlaying (
        void );
```

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示蜂鸣器是/否正在运行脚本。

4.3. 键盘的API函数

4.3.1. 数据结构

4.3.1.1. KeyStatusEnumType

[Key_Port.h]

```
enum KeyStatus
{
	Key_release = 0,
	Key_PressDown = 1,
	Key_shortPress = 2,
	Key_longPress = 3
};
typedef enum KeyStatus KeyStatusEnumType;
```

被 Key_MessageDataType 用于传输按键动作状态。

数据成员	值	说明
Key_release	0	长按后释放的按键
Key_PressDown	1	最先按下的键
Key_shortPress	2	短按（短于长按时间门槛）后释放的按键
Key_longPress	3	长按（长于长按时间门槛）的按键

表3： KeyStatusEnumType 数据结构

4.3.1.2. KeyInputEnumType

[Key_Port.h]

```
enum KeyInput
{
	Key_noKeyInput = 0,
	Key_ClearKeyInput = 1,
	Key_Num0KeyInput = 2,
	Key_RightKeyInput = 4,
	Key_Num7KeyInput = 6,
	Key_Num8KeyInput = 7,
	Key_Num9KeyInput = 8,
	Key_LeftKeyInput = 9,
	Key_Num4KeyInput = 11,
	Key_Num5KeyInput = 12,
	Key_Num6KeyInput = 13,
	Key_DownKeyInput = 14,
	Key_Num1KeyInput = 16,
	Key_Num2KeyInput = 17,
	Key_Num3KeyInput = 18,
	Key_UpKeyInput = 19,
	Key_F1KeyInput = 21,
	Key_F2KeyInput = 22,
	Key_F3KeyInput = 23,
```

```
        Key_F4KeyInput = 24,
        Key_PowerKeyInput = 26
};
typedef enum KeyInput KeyInputEnumType;
```

被 Key_MessageDataType 和 Key_Msg_GetKeyPressing 用于传输按键名。

数据成员	值	说明
Key_noKeyInput	0	无按键
Key_ClearKeyInput	1	清除按键
Key_Num0KeyInput	2	数字键 0
Key_RightKeyInput	4	右方向键
Key_Num7KeyInput	6	数字键 7
Key_Num8KeyInput	7	数字键 8
Key_Num9KeyInput	8	数字键 9
Key_LeftKeyInput	9	左方向键
Key_Num4KeyInput	11	数字键 4
Key_Num5KeyInput	12	数字键 5
Key_Num6KeyInput	13	数字键 6
Key_DownKeyInput	14	下方向键
Key_Num1KeyInput	16	数字键 1
Key_Num2KeyInput	17	数字键 2
Key_Num3KeyInput	18	数字键 3
Key_UpKeyInput	19	上方向键
Key_F1KeyInput	21	F1 功能键
Key_F2KeyInput	22	F2 功能键
Key_F3KeyInput	23	F3 功能键
Key_F4KeyInput	24	F4 功能键
Key_PowerKeyInput	26	回车键/电源开关

表4 ：KeyInputEnumType 数据结构

4.3.1.3. Key_MessageDataType

```
[Key_Msg.h]

struct Key_Message
{
      KeyStatusEnumType   eKeyStatus;
      KeyInputEnumType    eInputKey;
};
typedef struct Key_Message Key_MessageDataType;
```

用于 Key_Msg_ReceiveKey。

数据成员	值	说明
eKeyStatus	KeyStatusEnumType	按键动作的状态
eInputKey	KeyInputEnumType	按键名

表5：Key_MessageDataType 数据结构

4.3.2. 函数

4.3.2.1. Key_Port_IsAnyKeyDown

此函数用于确认是否有按键按下。

```
[Key_Port.h]

BOOLEAN Key_Port_IsAnyKeyDown (
         void );
```

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示是/否有按键按下。

4.3.2.2. Key_Msg_ReceiveKey

此函数用于 TimeOut 时限内接收键盘消息，直到接收成功。

```
[Key_Msg.h]

BOOLEAN Key_Msg_ReceiveKey (
         Key_MessageDataType* Key_MsgRecBuffer,
         portTickType TimeOut );
```

参数：

Key_MsgRecBuffer [out] 待输出的键盘消息。

TimeOut [in] 接收键盘信息的等待时间。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示接收键盘消息成功/失败。

4.3.2.3. Key_Msg_GetKeyPressing

此函数用于获取当前被按下的按键名。

[Key_Msg.h]

```
KeyInputEnumType Key_Msg_GetKeyPressing (
        void );
```

返回值：

KeyInputEnumType 此函数返回当前被按下的按键名。

4.3.2.4. Key_Ctrl_FlushMsgBuffer

此函数用于清除键盘消息缓存。调用此函数后，函数 Key_Msg_ReceiveKey 只能收取新生成的键盘消息。

[Key_Ctrl.h]

```
void Key_Ctrl_FlushMsgBuffer (
        void );
```

4.3.2.5. Key_Ctrl_ScanLock

此函数用于关闭键盘生成新消息的功能。默认情况下，键盘生成新消息的功能是开启的。

[Key_Ctrl.h]

```
void Key_Ctrl_ScanLock (
        void );
```

4.3.2.6. Key_Ctrl_ScanUnlock

此函数用于开启键盘生成新消息的功能。默认情况下，键盘生成新消息的功能是开启的。

[Key_Ctrl.h]

```
void Key_Ctrl_ScanUnlock (
        void );
```

4.3.2.7. Key_Ctrl_SetLongPressThreshold

此函数用于设置区分长按/短按的阀值时长。默认情况下，阀值时长为 2 秒。

[Key_Ctrl.h]

```
void Key_Ctrl_SetLongPressThreshold (
        UINT8 Seconds );
```

参数：

Seconds [in] 阀值时长 [秒]。

4.3.2.8. Key_Tim_GetKeyDownTime

此函数用于获取当前按键操作的时长。如果没有按键操作，保留时长值。

```
[Key_Tim.h]

UINT16 Key_Tim_GetKeyDownTime (
        void );
```

返回值：

UINT16 返回按键操作的当前时长。[秒]

4.4. EEPROM的API函数

4.4.1. 数据结构

4.4.2. 函数

4.4.2.1. EEPROM_Write

此函数用于写数据到 EEPROM。EEPROM 的存储大小为 65536 字节，因而 usAddress 和 usSize 的和必然小于等于 65536。

[EEPROM.h]

```
BOOLEAN EEPROM_Write (
        UINT16 usAddress,
        const UINT8 *pucData,
        UINT16 usSize );
```

参数：

usAddress [in] 待写入 EEPROM 的目的存储的起始地址[0-65535]。

pucData [in] 待写入 EEPROM 的数据。

usSize [in] 输入数据的大小[单位：字节，范围：1-65535]。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示数据写入 EEPROM 成功/失败。

4.4.2.2. EEPROM_Read

此函数用于从 EEPROM 读取数据。EEPROM 的存储大小为 65536 字节，因而 usAddress 和 usSize 的和必然小于等于 65536。

[EEPROM.h]

```
BOOLEAN EEPROM_Read (
        UINT16 usAddress,
        UINT8 *pucData,
        UINT16 usSize );
```

参数：

usAddress [in] EEPROM 待读取的源存储的起始地址[0-65535]。

pucData [out] EEPROM 待读取的数据。

usSize [in] 输出数据的大小[单位：字节，范围：1-65535]。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示读取 EEPROM 数据成功/失败。

4.5. 实时时钟的API函数

4.5.1. 数据结构

4.5.2. 函数

4.5.2.1. EXRTC_Write_Ram

此函数用于向外部 RTC RAM 写入数据。外部 RTC RAM 存储大小为 238 字节，因而 usAddress 和 usSize 的和必须小于等于 238。

[EXRTC.h]

```
BOOLEAN EXRTC_Write_Ram (
        UINT16 usAddress,
        const UINT8 *pucData,
        UINT16 usSize );
```

参数：

usAddress [in] 待写入外部 RTC RAM 的目的存储的起始地址[0-237]。

pucData [in] 待写入外部 RTC RAM 的数据。

usSize [in] 输入数据的大小[单位：字节，范围：1-238]。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示向外部 RTC RAM 写入数据成功/失败。

4.5.2.2. EXRTC_Read_Ram

此函数用于从外部 RTC RAM 读取数据。外部 RTC RAM 存储大小为 238 字节，因而 usAddress 和 usSize 的和必须小于等于 238。

[EXRTC.h]

```
BOOLEAN EXRTC_Read_Ram (
        UINT16 usAddress,
        UINT8 *pucData,
        UINT16 usSize );
```

参数：

usAddress [in] 外部 RTC RAM 被读取数据的源存储的起始地址[0-237]。

pucData [out] 外部 RTC RAM 被读取数据的。

usSize [in] 输出数据的大小[单位：字节，范围：1-238]。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示从外部 RTC RAM 读取数据成功/失败。

4.5.2.3. EXRTC_Write_Time

此函数用于将十进制时间值设置成外部 RTC 时间。

```
[EXRTC.h]

BOOLEAN EXRTC_Write_Ram (
          UINT8 ucHour,
          UINT8 ucMinute,
          UINT8 ucSecond );
```

参数：

ucHour [in] 小时的十进制值 [0-23]。

ucMinute [in] 分钟的十进制值 [0-59]。

ucSecond [in] 秒的十进制值 [0-59]。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示将十进制时间值设置成外部 RTC 时间成功/失败。

4.5.2.4. EXRTC_Write_TimeBCD

此函数用于将二进制编码的十进制时间设置成外部 RTC 时间。

```
[EXRTC.h]

BOOLEAN EXRTC_Write_Ram (
          UINT8 ucHour,
          UINT8 ucMinute,
          UINT8 ucSecond );
```

参数：

ucHour [in] 小时的 BCD 值 [00h-23h]。

ucMinute [in] 分钟的 BCD 值 [00h-59h]。

ucSecond [in] 秒钟的 BCD 值 [00h-59h]。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示将时间值设置成外部 RTC 时间成功/失败。

4.5.2.5. EXRTC_Read_Time

此函数用于从外部 RTC 读取时间值，输出数据采用 BCD 格式。

```
[EXRTC.h]

BOOLEAN EXRTC_Read_Time (
        UINT8 *pucHour,
        UINT8 *pucMinute,
        UINT8 *pucSecond );
```

参数：

pucDay [out] 输出小时值。

pucDay [out] 输出分钟值。

pucDay [out] 输出秒钟值。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示从外部 RTC 读取时间值成功/失败。

4.5.2.6. EXRTC_Write_Date

此函数用于将十进制日期值设置成外部 RTC 日期。

```
[EXRTC.h]

BOOLEAN EXRTC_Write_Date (
        UINT8 ucYear,
        UINT8 ucMonth,
        UINT8 ucWeekday,
        UINT8 ucDay );
```

参数：

ucYear [in] 年份的十进制值 [0-99]。

ucMonth [in] 月的十进制值 [1-12]。

ucWeekday [in] 工作日的十进制值 [0-6]。

ucDay [in] 天的十进制值 [1-31]。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示将日期值设置成外部 RTC 日期成功/失败。

4.5.2.7. EXRTC_Write_DateBCD

此函数用于将二进制编码的十进制日期值设置成外部 RTC 日期。

[EXRTC.h]

```
BOOLEAN EXRTC_Write_DateBCD (
        UINT8 ucYear,
        UINT8 ucMonth,
        UINT8 ucWeekday,
        UINT8 ucDay );
```

参数：

ucYear [in] 年份的 BCD 值 [00h-99h]。

ucMonth [in] 月份的 BCD 值 [01h-12h]。

ucWeekday [in] 工作日的 BCD 值 [00h-06h]。

ucDay [in] 天的 BCD 值 [01h-31h]。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示将日期值设置成外部 RTC 日期成功/失败。

4.5.2.8. EXRTC_Read_Date

此函数用于从外部 RTC 读取日期值。输出数据采用 BCD 格式。

[EXRTC.h]

```
BOOLEAN EXRTC_Read_Date (
        UINT8 *pucYear,
        UINT8 *pucMonth,
        UINT8 *pucWeekday,
        UINT8 *pucDay );
```

参数：

pucYear [out] 输出年份值。

pucMonth [out] 输出月份值。

pucWeekday [out] 输出工作日值。

pucDay [out] 输出天值。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示从外部 RTC 读取日期值成功/失败。

4.6. LCD的API函数

4.6.1. 数据结构

4.6.2. 函数

4.6.2.1. LCD_SetCursor

此函数用于设置 LCD 光标位置。

```
[LCD.h]

BOOLEAN LCD_SetCursor (
         UINT8 ucLcdRowPosition,
         UINT8 ucLcdColumnPosition );
```

参数：

ucLcdRowPosition [in] LCD 光标的行位置 [0 - LCD_MAX_ROW]。

ucLcdColumnPosition [in] LCD 光标的列位置 [0 - LCD_MAX_COLUMN]。

返回值：

此函数返回 TRUE/FALSE，分别表示 LCD 光标位置设置成功（不超出范围）/失败。

4.6.2.2. LCD_GetCursor

此函数用于获取 LCD 当前的光标位置。

```
[LCD.h]

void LCD_GetCursor (
      UINT8 *pucLcdRowPosition,
      UINT8 *pucLcdColumnPosition );
```

参数：

pucLcdRowPosition [out] LCD 行位置。

pucLcdRowPosition [out] LCD 列位置。

4.6.2.3. LCD_Display_ASCIIChar

此函数用于在 LCD 上显示单个 ASCII 字符。

[LCD.h]

```
void LCD_Display_ASCIIChar (
        UINT8 ucLcdCharacterToDisplay,
        BOOLEAN bSetNextPosCur );
```

参数：

ucLcdCharacterToDisplay [in] 将在 LCD 上显示的 ASCII 码 [20h-7Eh]。

bSetNextPosCur [in] TRUE -- 光标由当前位置后移到下一个位置。

FALSE -- 光标仍处于当前位置。

4.6.2.4. LCD_DisplayASCIIMessage

此函数用于显示字符串。字符串可包含三个控制字符'\b'，'\r' 和'\n'。

[LCD.h]

```
void LCD_DisplayASCIIMessage (
        const UINT8 *LcdMessageToDisplay );
```

参数：

LcdMessageToDisplay [in] 待显示的以空字符结尾的字符串。

4.6.2.5. LCD_ClearDisplay

此函数用于按页，行，或字符清除 LCD 显示的内容。

[LCD.h]

```
void LCD_ClearDisplay (
        UINT8 index,
        UINT8 ucNumber );
```

参数：

Index [in] 0 -- 清除整页。

1—光标当前位置起，清除整行字符。ucNumber 表明将要清除行的行数。

2—光标当前位置起，清除列里的字符（一个字符是 6 列）。ucNumber 表明将要清除的字符列数。

ucNumber [in] 待清除的行数或列数。

4.6.2.6. LCD_SetContrast

此函数用于设置 LCD 对比度级别。

[LCD.h]

```
void LCD_SetContrast (
        UINT8 contrast_level );
```

参数：

contrast_level [in] 对比度级别 [0-255]。默认值为 169。

4.6.2.7. LCD_SetBacklight

此函数用于开/关 LCD 背光。

[LCD.h]

```
void LCD_SetBacklight (
        BOOLEAN bTurnOn );
```

参数：

bTurnOn [in] TRUE -- 背光开 FALSE -- 背光关

4.6.2.8. LCD_Display_Cursor

此函数用于显示 LCD 纵向光标（8 像素）。

[LCD.h]

```
void LCD_Display_Cursor (
        void );
```

4.6.2.9. LCD_Clear_Cursor

此函数用于清除 LCD 纵向光标。

[LCD.h]

```
void LCD_Clear_Cursor (
        void );
```

4.6.2.10. LCD_Display_Page

此函数用于全屏显示图片。

[LCD.h]

```
void LCD_Display_Page (
        UINT8 *pucBitmap );
```

参数：

pucBitmap [in] 待显示的位图原始数据串[分辨率：（LCD_MAX_ROW x 8） x LCD_MAX_COLUMN]。

4.6.2.11. LCD_DisplayGraphic

此函数用于在指定位置显示类似于图标的位图图像。

[LCD.h]

```
void LCD_DisplayGraphic (
        UINT8 ucLcdRowNumber,
        UINT8 ucLcdColumnNumber,
        const UINT8 *pucBitMap );
```

参数：

ucLcdRowNumber [in] 图像占用的行数

ucLcdColumnNumber [in] 图像占用的列数

pucBitMap [in] 图像数据数组

4.6.2.12. LCD_DisplayOn

此函数用于开/关 LCD 显示屏。

[LCD.h]

```
void LCD_DisplayOn (
        BOOLEAN bTurnOn );
```

参数：

bTurnOn [in] TRUE -- 显示屏开 FALSE -- 显示屏关

4.6.2.13. LCD_DisplayDecimal

此函数用于在 LCD 上显示十进制数。

[LCD.h]

```
void LCD_DisplayDecimal (
UINT32 ulDecimal );
```

参数：

ulDecimal [in] 待显示的十进制数。

4.6.2.14. LCD_DisplayHex

此函数用于在 LCD 上显示 16 进制数。

[LCD.h]

```
void LCD_DisplayHex (
      UINT8 ucDisplay );
```

参数：

ucDisplay [in] 待显示的数字。

4.6.2.15. LCD_DisplayHexN

此函数用于在 LCD 上显示 16 进制数字串。

[LCD.h]

```
void LCD_DisplayHexN (
        const UINT8 *pucDisplay,
        UINT8 Number );
```

参数：

pucDisplay [in] 待显示的 16 进制数字型数组。

Number [in] 16 进制数字型数组的大小 [字节]。

4.6.2.16. LCD_DisplayFloat

此函数用于在 LCD 上显示浮点数。

[LCD.h]

```
void LCD_DisplayFloat (
        UINT32 ulDecimal,
        UINT8 Exp );
```

参数：

ulDecimal [in] 待显示的数（不带小数点）。

Exp [in] 小数点后面的十进制数字[0-9]。

4.6.2.17. LCD_DrawTitleBox

此函数用于在 LCD 上显示标题框。

[LCD.h]

```
void LCD_DrawTitleBox (
        const UINT8 *TitleMessage );
```

参数：

TitleMessage [in] 待显示的以空字符结尾的字符串（仅支持 20-7E）。

4.7. 串行闪存的API函数

串行闪存的存储必须先擦除后才能写数据。务必保证存储的所有数据为 FFh。

4.7.1. 数据结构

4.7.1.1. SFlash_EraseBlockType

```
[SFlash.h]

enum SFlash_EraseBlock
{
        ERASE_4K    = SF_ERASE_4K,
        ERASE_64K   = SF_ERASE_64K
};
typedef enum SFlash_EraseBlock SFlash_EraseBlockType;

        #define SF_ERASE_4K                     0x20
        #define SF_ERASE_64K                    0xD8
```

被 SerialFlash_Erase_Block 用于设置待擦除的块存储的大小类型。

数据成员	值	说明
ERASE_4K	SF_ERASE_4K	4K 字节的块类型
ERASE_64K	SF_ERASE_64K	64K 字节的块类型

表6 ：SFlash_EraseBlockType 数据结构

4.7.2. 函数

4.7.2.1. SerialFlash_ReadDataBytes

此函数用于从串行闪存读取数据。

```
[SFlash.h]

void SerialFlash_ReadDataBytes (
        UINT32 ulAddress,
        UINT8* pucReceiveBufferPtr,
        UINT32 ulLength );
```

参数：

usAddress [in] 串行闪存待读取的源存储的起始地址[20000h-7FFFFh]。

pucReceiveBufferPtr [out] 串行闪存待读取的数据。

ulLength [in] 输出数据的大小[单位：字节]。

4.7.2.2. SerialFlash_Erase_Block

此函数用于擦除串行闪存的块存储。

```
[SFlash.h]

BOOLEAN SerialFlash_Erase_Block (
        SFlash_EraseBlockType BlockType,
        UINT32 ulAddress );
```

参数：

BlockType [in] 待擦除的块存储的大小类型，
[ERASE_4K -- 4KB 块， ERASE_64K -- 64KB 块]。

ulAddress [in] 串行闪存待擦除的目的存储的起始地址[>= 20000h，必须按块大小对齐]。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示擦除串行闪存的块存储成功/失败。

4.7.2.3. SerialFlash_WriteDataBytes

此函数用于向串行闪存写数据。向串行闪存写入数据前，应当先擦除目的存储（保证所有存储数据为 FFh）。

```
[SFlash.h]

BOOLEAN SerialFlash_WriteDataBytes (
        UINT32 ulAddress,
        const UINT8* pucWriteBufferPtr,
        UINT32 ulLength );
```

参数：

usAddress [in] 串行闪存待写入数据的目的存储的起始地址[20000h-7FFFFh]。

pucWriteBufferPtr [in] 待写入串行闪存的数据。

ulLength [in] 输入数据的大小[单位：字节]。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示数据写入串行闪存成功/失败。

4.8. RS232 的API函数

4.8.1. 数据结构

4.8.1.1. ParityEnumType

[RS232.h]

```
enum Parity
{
    No_Parity   = 0,
    Odd_Parity  = 1,
    Even_Parity = 2
};
typedef enum Parity ParityEnumType;
```

被 RS232_ParamDataType 用于传输 RS232 的奇偶校验设置。

数据成员	值	说明
No_Parity	0	RS232 无奇偶校验模式
Odd_Parity	1	RS232 奇校验模式
Even_Parity	2	RS232 偶校验模式

表7：ParityEnumType 数据结构

4.8.1.2. RS232_ParamDataType

[RS232.h]

```
struct RS232_Parameter
{
    UINT32         Baudrate;
    ParityEnumType  ParityMode;
    BOOLEAN      SevenOrEightDataBit;
    BOOLEAN      TwoOrOneStopBit;
};
typedef struct RS232_Parameter RS232_ParamDataType;
```

被 RS232_Config 用于传输 RS232 的配置参数。

数据成员	值	说明
Baudrate	UINT32	该值表示 RS232 的波特率。（例：值 9600 表示 RS232 的波特率为 960 0bps，范围为 400 到 115200）
ParityMode	ParityEnumType	RS232 配置的奇偶校验模式

数据成员	值	说明
SevenOrEightDataBit	BOOLEAN	7 或 8 位数据模式： TRUE – 7 位数据模式 FALSE – 8 位数据模式
TwoOrOneStopBit	BOOLEAN	2 或 1 停止位模式： TRUE – 2 停止位模式 FALSE – 1 停止位模式

表8：RS232_ParamDataType 数据结构

4.8.2. 函数

4.8.2.1. RS232_Config

此函数用于设置 RS232 端口的参数。使用此函数前，RS232 端口应处于关闭状态；否则，RS232_Config 将返回 false。

[RS232.h]

```
BOOLEAN RS232_Config (
    const RS232_ParamDataType* Param );
```

参数：

Param [in] 参数数据。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示 RS232 参数设置成功/失败。

4.8.2.2. RS232_OpenPort

此函数用于打开 RS232 端口。使用此函数前，RS232 端口应处于关闭状态；否则，RS232_OpenPort 将返回 false。

[RS232.h]

```
BOOLEAN RS232_OpenPort (
    UINT32* pulHandle );
```

参数：

pulHandle [out] 控制 RS232 端口的句柄。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示打开 RS232 端口成功/失败。

4.8.2.3. RS232_ClosePort

此函数用于关闭 RS232 端口。

```
[RS232.h]

BOOLEAN RS232_ClosePort (
    UINT32 ulHandle );
```

参数：

ulHandle [in] 从 RS232_OpenPort 得到的句柄值。如果该值不是从 RS232_OpenPort 得到的，函数将返回 false。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示关闭 RS232 端口成功/失败。

4.8.2.4. RS232_ReceivedDataNumber

此函数用于获取 RS232 端口接收的字节数。

```
[RS232.h]

UINT16 RS232_ReceivedDataNumber (
    UINT32 ulHandle );
```

参数：

ulHandle [in] 从 RS232_OpenPort 得到的句柄值。如果该值不是从 RS232_OpenPort 得到的，函数将返回 0。

返回值：

UINT16 此函数将返回 RS232 端口接收的字节数。

4.8.2.5. RS232_Receive

此函数用于获取 RS232 端口接收的数据。此函数 TimeOut 时限内等待接收数据，直到接收成功。

```
[RS232.h]

BOOLEAN RS232_Receive (
    UINT32 ulHandle,
    UINT8* pucRecBuf,
    UINT16* pusLen,
    portTickType TimeOut );
```

参数：

ulHandle [in] 从 RS232_OpenPort 得到的句柄值。如果该值不是从 RS232_OpenPort 得到的，函数将返回 false。

pucRecBuf [out] 待从 RS232 端口获取的数据。

pusLen [in&out] 接收缓存大小和输出数据大小[字节]

TimeOut [in] 从 RS232 端口接收数据的等待时间

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示从 RS232 端口接收数据成功/失败。

4.8.2.6. RS232_Send

此函数用于向 RS232 端口发送数据。

```
[RS232.h]

BOOLEAN RS232_Send (
    UINT32 ulHandle,
    const UINT8* pucSndBuf,
    UINT16 usLen );
```

参数：

ulHandle [in] 从 RS232_OpenPort 得到的句柄值。如果该值不是从 RS232_OpenPort 得到的，函数将返回 false。

pucSndBuf [in] 待发送到 RS232 端口的数据。

usLen [in] 待发送到 RS232 端口的数据大小。

返回值：

BOOLEAN 此函数返回 TRUE/FALSE，分别表示向 RS232 端口发送数据成功/失败。

4.9. 混合I/O的API函数

4.9.1. 数据结构

4.9.1.1. IO_VirtualNameType

[IO.h]

```
enum IO_VirtualName
{
        IO_LED0Output                   = 14,
        IO_LED1Output                   = 15,
        IO_LED2Output                   = 16,
        IO_LED3Output                   = 17,
        IO_LED4Output                   = 18,
        IO_LED5Output                   = 19,
        IO_LED6Output                   = 20,
        IO_LED7Output                   = 21
};
typedef enum IO_VirtualName IO_VirtualNameType;
```

被 IO_ReadDigitalOutput 和 IO_WriteDigitalOutput 用于设置待操作的 I/O 端口的地址。

数据成员	值	说明
IO_LED0Output	14	第一个双色 LED 的红色
IO_LED1Output	15	第一个双色 LED 的绿色
IO_LED2Output	16	第二个双色 LED 的红色
IO_LED3Output	17	第二个双色 LED 的绿色
IO_LED4Output	18	第三个双色 LED 的红色
IO_LED5Output	19	第三个双色 LED 的绿色
IO_LED6Output	20	第四个双色 LED 的红色
IO_LED7Output	21	第四个双色 LED 的绿色

表9：IO_VirtualNameType 数据结构

4.9.1.2. IO_ActiveInactiveStateType

[IO.h]

```
enum IO_ActiveInactiveState
{
        IO_InactiveState = 0,
        IO_ActiveState = 1,
        IO_UndefineState = 2
};
typedef enum IO_ActiveInactiveState IO_ActiveInactiveStateType;
```

被 IO_ReadDigitalOutput 和 IO_WriteDigitalOutput 用于传输 I/O 端口的逻辑状态。

数据成员	值	说明
IO_InactiveState	0	逻辑错误
IO_ActiveState	1	逻辑正确
IO_UndefineState	2	逻辑无效

表10 ：IO_ActiveInactiveStateType 数据结构

4.9.2. 函数

4.9.2.1. IO_ReadDigitalOutput

此函数用于获取端口名地址对应的数字输出端口值。

```
[IO.h]

IO_ActiveInactiveStateType IO_ReadDigitalOutput (
        IO_VirtualNameType Address );
```

参数：

Address [in]待读取的数字输出端口的端口名地址。

返回值：

IO_ActiveInactiveStateType 此函数用于返回数字输出端口的值 [IO_InactiveState —端口未激活；IO_ActiveState —端口已激活]。

4.9.2.2. IO_WriteDigitalOutput

此函数用于设置端口名地址对应的数字输出端口值。

```
[IO.h]

void IO_WriteDigitalOutput (
        IO_ActiveInactiveStateType State,
        IO_VirtualNameType Address );
```

参数：

State [in] 待设置的数字输出端口值 [IO_InactiveState —端口未激活；IO_ActiveState —端口已激活]。

Address [in]待写入的数字输出端口的端口名地址。

4.9.2.3. IO_WriteLEDOutput

此函数用于给所有 LED 设置输出值。

[IO.h]

```
void IO_WriteLEDOutput (
        UINT8 ucLED );
```

参数：

ucLED [in]所有双色 LED 灯开/关的输出值。

[位 0~7 对应 IO_ActiveInactiveStateType 的 IO_LED0Output ~ IO_LED7Output；

IO_LED0Output：第一个 LED 的红色；

IO_LED1Output： 第一个 LED 的绿色；

IO_LED2Output：第二个 LED 的红色；

IO_LED3Output：第二个 LED 的绿色；

IO_LED4Output：第三个 LED 的红色；

IO_LED5Output：第三个 LED 的绿色；

IO_LED6Output：第四个 LED 的红色；

IO_LED7Output：第四个 LED 的绿色；值为 1 的位输出已激活状态；值为 0 的位输出未激活状态。

4.9.2.4. IO_ReadLEDOutput

此函数用于获取给所有 LED 设置的输出值。

[IO.h]

```
UINT8 IO_ReadLEDOutput (
        void );
```

返回值：

UINT8 此函数用于返回给所有双色 LED 设置的输出值。

[位 0~7 对应 IO_ActiveInactiveStateType 的 IO_LED0Output ~ IO_LED7Output；

IO_LED0Output：第一个 LED 的红色；
IO_LED1Output：第一个 LED 的绿色；
IO_LED2Output：第二个 LED 的红色；

IO_LED3Output：第二个 LED 的绿色；

IO_LED4Output：第三个 LED 的红色；

IO_LED5Output：第三个 LED 的绿色；

IO_LED6Output：第四个 LED 的红色；

IO_LED7Output：第四个 LED 的绿色；值为 1 的位表示已激活状态；值为 0 的位表示未激活状态。

4.9.2.5. IO_SystemShutdown

此函数用于关闭整个系统。

[IO.h]

```
void IO_SystemShutdown (
    void );
```

4.9.2.6. IO_SystemSleep

此函数使系统进入睡眠状态。可以按任意键恢复系统。

[IO.h]

```
void IO_SystemSleep (
    void );
```

4.9.2.7. IO_SystemRestart

此函数用于重启系统。

[IO.h]

```
void IO_SystemRestart (
    void );
```

4.9.2.8. IO_GetSystemVersion

此函数用于返回版本信息。

[IO.h]

```
Const UCHAR*IO_GetSystemVersion (
    void );
```

返回值：

const UCHAR* 此函数用于返回版本信息的指针，版本信息是以空字符'\0'结尾的 ASCII 串。

5.0. RF卡的API函数（仅限ACR89-CL版本）

5.1. 数据结构

5.2. 函数

对于 ACR89-CL 版本，用 SCard_Manager_SelectCard 为 RF 卡插槽选择 RF 卡，索引 0，为智能卡插槽选择索引 1-5（其他版本，选择索引 0-4）。RF 卡插槽的 APDU 数据功能与 ACR122 兼容。
SCard_Manager_CardOn 也可获取 ATR。SCard_Manager_CardOff 是 RF 卡插槽的虚拟功能。
SCard_Manager_SendBlock 也用于发送 APDU 并从 RF 卡插槽获取反应。
SCard_Manager_Msg_Receive 也用于接收 RF 卡插入/移出消息，直到超时。（信息 RF 卡插槽的索引是 0，而智能卡插槽的索引是 1-2）

5.2.1. RFIF_Sleep

此函数用于设置 RF 接口下电，以减少耗电。RF 接口默认上电。

```
[RFCard.h]

void RFIF_Sleep (
    void );
```

5.2.2. RFIF_Wakeup

此函数用于设置 RF 接口上电，以实现非接触卡功能。RF 接口默认上电。

```
[RFCard.h]

void RFIF_WakeUp (
    void );
```

6.0. FreeRTOS的API函数

ACR89U-A1 SDK 包括 FreeRTOS 移植和 ISR 处理。因此，不再需要使用 ISR 相关的 API 函数。系统心跳期间，portmacro.h 中的宏 **portTICK_RATE_MS** 提供值；

例，如果 **portTICK_RATE_MS** 等于 1，则表示 type portTickType 的单位是 1 毫秒。对于 ACR89U-A2 SDK，FreeRTOS 配有优先权和软件计时器，但不使用协程。

更多关于 FreeRTOS 的细节，请访问：

http://www.freertos.org.

6.1. 任务创建

6.1.1. xTaskHandle

关联任务时使用的数据类型。例：

调用 xTaskCreate 后，将通过指针参数返回变量 xTaskHandle，该变量可用作 vTaskDelete 的参数以删除任务。

```
[task.h]
```

6.1.2. xTaskCreate

此函数用于创建并添加新任务到就绪任务列表。

```
[task.h]

portBASE_TYPE xTaskCreate(
                            pdTASK_CODE pvTaskCode,
                            const portCHAR * const pcName,
                            unsigned portSHORT usStackDepth,
                            void *pvParameters,
                            unsigned portBASE_TYPE uxPriority,
                            xTaskHandle *pvCreatedTask
                          );
```

参数：

pvTaskCode	[in] 指向任务输入功能的指针。任务执行不能回退（例，死循环）。
pcName	[in] 描述性的任务名。仅仅是方便调试所用。最大长度由 configMAX_TASK_NAME_LEN 定义。
usStackDepth	[in] 任务堆栈的深度，表示堆栈内可容纳的变量数，而不是字节数。例：如果栈宽 16 位，usStackDepth 为 100，则会为堆栈存储分配 200 字节。栈深和栈宽的乘积不得超过 size_t 类型变量所含的最大值。
pvParameters	[in] 指针，将用作正在创建的任务的参数。
uxPriority	[in] 任务优先级。
pvCreatedTask	[out] 关联创建的任务时，用此参数回传句柄。

返回值：

portBASE_TYPE	如果任务创建成功并被添加到就绪任务列表，则 pdPASS；否则，projdefs.h 将定义一个错误码。

例：

```
// 待创建的任务。
void vTaskCode( void * pvParameters )
{
    for( ;; )
    {
        //任务码。
    }
}

// 任务创建函数。
void vOtherFunction( void )
{
  static unsigned char ucParameterToPass;
  xTaskHandle xHandle;

  // 创建任务，存储句柄。注意传递的参数  ucParameterToPass
  // 在任务的生命周期内必须存在，以声明为静态。如果它只是一个
  // 自动堆栈变量，它可能不存在，
  // 或者，当新任务尝试访问它时，它已经损坏。
    xTaskCreate( vTaskCode,
                 "NAME",
                 STACK_SIZE,
                 &ucParameterToPass,
                 tskIDLE_PRIORITY,
                 &xHandle );

    // 用句柄删除任务。
    vTaskDelete( xHandle );
}
```

6.1.3. vTaskDelete

此函数用于删除 RTOS 实时内核管理的一个任务。该任务将从所有事件列表（包括所有就绪，阻塞，挂起的事件列表）上移除。

```
[task.h]

void vTaskDelete( xTaskHandle pxTask );
```

参数：

pxTask [in] 待删除的任务的句柄。传递 NULL 句柄导致调用者被删除。

例：

```
void vOtherFunction( void )
{
  xTaskHandle xHandle;

    // 创建任务，存储句柄。
    xTaskCreate( vTaskCode,
                 "NAME",
```

```
                STACK_SIZE,
                NULL,
                tskIDLE_PRIORITY,
                &xHandle );

    // 用句柄删除任务。
    vTaskDelete( xHandle );
}
```

6.2. 任务控制

6.2.1. vTaskDelay

此函数用于将任务延长指定的心跳计数。

```
[task.h]

void vTaskDelay( portTickType xTicksToDelay );
```

参数：

xTicksToDelay [in] 以心跳周期为单位的时间量，表示调用者任务被阻塞的时长

例：

```
 void vTaskFunction( void * pvParameters )
 {
   /* 阻塞 500ms。*/
   const portTickType xDelay = 500 / portTICK_RATE_MS;

     for( ;; )
     {
     /* 每隔 500ms 切换一次 LED，切换之间进行阻塞。*/
         vToggleLED();
         vTaskDelay( xDelay );
     }
}
```

6.2.2. vTaskDelayUntil

此函数用于延迟任务直到一个指定的时间结束。此函数用于以固定的频率执行周期任务。

```
[task.h]

void vTaskDelayUntil( portTickType *pxPreviousWakeTime, portTickType
xTimeIncrement );
```

参数：

pxPreviousWakeTime [in] 指向变量的指针，该变量表明任务最后开启的时间。初次使用，必须用当前时间初始化。（请参见下面的例子）然后，这个变量在vTaskDelayUntil()内会自动更新。

xTimeIncrement [in] 循环时间周期。任务将在一定时间解除阻塞（*pxPreviousWakeTime + xTimeIncrement）。用相同的 xTimeIncrement 值调用 v TaskDelayUntil 将使任务按固定周期执行。

例：

```
// 每 10 次心跳执行一个动作。
void vTaskFunction( void * pvParameters )
{
  portTickType xLastWakeTime;
  const portTickType xFrequency = 10;

    // 用当前时间初始化变量 xLastWakeTime。
    xLastWakeTime = xTaskGetTickCount();

    for( ;; )
    {
        // 等待下一周期。
        vTaskDelayUntil( &xLastWakeTime, xFrequency );

        // 执行动作。
    }
}
```

6.2.3. uxTaskPriorityGet

此函数用于获取任务优先级。

```
[task.h]

unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );
```

参数：

pxTask [in] 待查询任务的句柄。传递一个 NULL 句柄将返回调用者的优先级。

返回值：

unsigned portBASE_TYPE pxTask 的优先级。

例：

```
void Get_ATR( void )
{
 xTaskHandle xHandle;

    // 创建任务，存储句柄。
    xTaskCreate( vTaskCode,
                 "NAME",
                 STACK_SIZE,
```

```
                NULL,
                tskIDLE_PRIORITY,
                &xHandle );

    // ...

    // 用句柄获取所创建任务的优先级。
    // 创建时使用的是 tskIDLE_PRIORITY，而后
    // 任务可能更改自己的优先级。
    if( uxTaskPriorityGet( xHandle ) != tskIDLE_PRIORITY )
    {
        // 任务优先级已经更改。
    }

    // ...

    // 我们的优先级是否高于所创建任务呢？
    if( uxTaskPriorityGet( xHandle ) < uxTaskPriorityGet( NULL ) )
    {
        // 我们的优先级(用 NULL 句柄获取)高于所创建任务。
    }
}
```

6.2.4. vTaskPrioritySet

此函数用于设置任务优先级。如果设置的优先级高于当前执行任务，函数返回优先级前将进行现场切换。

```
[task.h]

void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE
uxNewPriority );
```

参数：

pxTask	[in] 当前设置优先级的任务的句柄。传递一个 NULL 句柄将返回所设置的调用者的优先级。
uxNewPriority	[in] 将为任务设置的优先级。

例：

```
void Get_ATR( void )
{
  xTaskHandle xHandle;

    // 创建任务，存储句柄。
    xTaskCreate( vTaskCode,
                "NAME",
                STACK_SIZE,
                NULL,
                tskIDLE_PRIORITY,
                &xHandle );
```

```
        // ...

        // 用句柄提高所创建任务的优先级。
        vTaskPrioritySet( xHandle, tskIDLE_PRIORITY + 1 );

        // ...

        // 用 NULL 句柄将我们的优先级提高到同一值。
        vTaskPrioritySet( NULL, tskIDLE_PRIORITY + 1 );
    }
```

6.2.5. vTaskSuspend

此函数用于挂起任务。挂起后，不论优先级如何，任务将没有任何微控制器处理时间。多次调用 vTaskSuspend 时，效果并不累加。

例：

对同一任务两次调用 vTaskSuspend ()后，要恢复该任务时，只需调用 vTaskResume ()一次。

```
[task.h]

void vTaskSuspend( xTaskHandle pxTaskToSuspend );
```

参数：

pxTaskToSuspend [in] 所挂起任务的句柄。传递 NULL 句柄将挂起调用者任务。

例：

```
void Get_ATR( void )
{
  xTaskHandle xHandle;

    // 创建任务，存储句柄。
    xTaskCreate( vTaskCode,
                 "NAME",
                 STACK_SIZE,
                 NULL,
                 tskIDLE_PRIORITY,
                 &xHandle );

    // ...

    // 用句柄挂起创建的任务。
    vTaskSuspend( xHandle );

    // ...

    // 在这期间，不执行创建的任务，除非
    // 另一个任务调用 vTaskResume( xHandle )。

    //...

    // 我们自己先挂起。
```

```
        vTaskSuspend( NULL );

        // 不会到这一步，除非另一个任务用我们的句柄作参数
        // 调用了 vTaskResume。
    }
```

6.2.6. vTaskResume

此函数用于恢复挂起的任务。如需恢复多次调用 vTaskSuspend ()后挂起的任务，只需要调用 vTaskResume ()一次。

```
[task.h]

void vTaskResume( xTaskHandle pxTaskToResume );
```

参数：

pxTaskToResume [in] 就绪任务的句柄。

例：

```
void Get_ATR( void )
{
  xTaskHandle xHandle;

    // 创建任务，存储句柄。
    xTaskCreate( vTaskCode,
                 "NAME",
                 STACK_SIZE,
                 NULL,
                 tskIDLE_PRIORITY,
                 &xHandle );

    // ...

    // 用句柄挂起创建的任务。
    vTaskSuspend( xHandle );

    // ...

    // 在这期间，不执行创建的任务，除非
    // 另一个任务调用 vTaskResume( xHandle )。

    //...

    // 恢复挂起的任务。
    vTaskResume( xHandle );

    // 创建的任务将再次获得
    // 与其在系统内的优先级一致的微控制器处理时间。
}
```

6.3. 任务函数

6.3.1. xTaskGetCurrentTaskHandle

此函数用于获取当前任务的句柄。

```
[task.h]

xTaskHandle xTaskGetCurrentTaskHandle( void );
```

返回值：

xTaskHandle 当前执行的任务（调用者）的句柄。

6.3.2. xTaskGetTickCount

此函数用于获取心跳计数。

```
[task.h]

volatile portTickType xTaskGetTickCount( void );
```

返回值：

portTickType 心跳计数，从 vTaskStartScheduler 被调用开始算起。

6.3.3. xTaskGetSchedulerState

此函数用于获取调度器状态。

```
[task.h]

portBASE_TYPE xTaskGetSchedulerState( void );
```

返回值：

portBASE_TYPE 下列常量中的一个 （由 task.h 定义)：taskSCHEDULER_NOT_STARTED、taskSCHEDULER_RUNNING、taskSCHEDULER_SUSPENDED

6.3.4. uxTaskGetNumberOfTasks

此函数用于获取任务数。

```
[task.h]

unsigned portBASE_TYPE uxTaskGetNumberOfTasks( void );
```

返回值：

unsigned portBASE_TYPE 目前由实时内核管理的任务数。这个任务数包括就绪，阻塞和挂起的任务。也包括已经被删除但还没有被空闲任务释放的任务。

6.4. 内核控制

6.4.1. taskYIELD

此宏用于强制性现场切换。

[task.h]

```
taskYIELD();
```

6.4.2. taskENTER_CRITICAL

此宏用于标示代码临界段的开端。不能在代码临界段抢先进行现场切换。

[task.h]

```
taskENTER_CRITICAL();
```

6.4.3. taskEXIT_CRITICAL

此宏用于标示代码临界段的结尾。不能在代码临界段抢先进行现场切换。

[task.h]

```
taskEXIT_CRITICAL();
```

6.4.4. vTaskSuspendAll

此函数用于保持中断（包括内核心跳中断）使能时挂起所有实时内核的活动。调用 vTaskSuspendAll ()后，调用该函数的任务继续执行，并不会被置换出运行态，直到 xTaskResumeAll ()被调用。调度器挂起期间，禁止调用可能引起现场切换的 API 函数，例如 vTaskDelayUntill ()、xQueueSend ()等。

[task.h]

```
void vTaskSuspendAll( void );
```

例：

```
void vTask1( void * pvParameters )
{
    for( ;; )
    {
        //任务码。

        // ...

        // 某些情况下，任务需要执行时间较长的操作。
        // 在此期间，任务不宜置换出运行态。不能使用
        // taskENTER_CRITICAL、()/taskEXIT_CRITICAL ()，因为
        // 时间较长的操作可能会错过中断，包括
        // 心跳。

        // 防止实时内核置换任务。
        vTaskSuspendAll ();
```

```
            // 此处执行操作。无需使用代码临界段，
            // 因为我们有全部的微控制器处理时间。
            // 在此期间，中断设置仍有效，并且
            // 内核心跳计数保持不变。

            // ...

            操作完成。重启内核。
            xTaskResumeAll ();
        }
    }
```

6.4.5. xTaskResumeAll

此函数用于 vTaskSuspendAll ()被调用后恢复实时内核功能。vTaskSuspendAll ()被调用后，内核将控制所有被执行的任务。

```
[task.h]

portBASE_TYPE xTaskResumeAll( void );
```

返回值：

portBASE_TYPE 如果恢复调度器引起现场切换，返回 pdTRUE；否则，返回 pdFALSE。

例：

```
void vTask1( void * pvParameters )
{
    for( ;; )
    {
        //任务码。

        // ...

        // 某些情况下，任务需要执行时间较长的操作。
        // 在此期间，任务不宜置换出运行态。不能使用
        // taskENTER_CRITICAL、()/taskEXIT_CRITICAL ()，因为
        // 时间较长的操作可能会错过中断，包括
        // 心跳。

        // 防止实时内核置换任务。
        xTaskSuspendAll ();

        // 此处执行操作。无需使用代码临界段，
        // 因为我们有全部的微控制器处理时间。
        // 在此期间，中断设置仍有效，并且
        // 实时内核心跳计数保持不变。

        // ...
```

```
            操作完成。重启内核。需要强制进行
            // 现场切换 - 但是，如果已经引发现场切换，
            // 再恢复调度器就毫无意义了。
            if( !xTaskResumeAll () )
            {
                 taskYIELD ();
            }
        }
    }
```

6.5. 队列管理

6.5.1. uxQueueMessagesWaiting

此函数用于返回队列存储的消息数。

```
[queue.h]

unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle xQueue );
```

参数：

xQueue [in] 被查询队列的句柄。

返回值：

unsigned portBASE_TYPE 队列的消息数。

6.5.2. xQueueCreate

此函数用于创建队列实例。为新队列分配所需存储，并返回队列句柄。

```
[queue.h]
xQueueHandle xQueueCreate(
                              unsigned portBASE_TYPE uxQueueLength,
                              unsigned portBASE_TYPE uxItemSize
                          );
```

参数：

uxQueueLength [in] 队列能容纳的最大消息数量。

uxItemSize [in] 队列内每个消息所需的字节数。消息通过复制而不是关联排队，因此，将复制所需的字节数给每个消息。队列中每个消息必须分配同样大小的字节数。

返回值：

xQueueHandle 如果队列创建成功，则返回一个句柄给新建队列。如果不能创建队列，则返回 0。

例：

```
struct AMessage
{
   portCHAR ucMessageID;
   portCHAR ucData[ 20 ];
};

void vATask( void *pvParameters )
{
  xQueueHandle xQueue1, xQueue2;

   // 创建一个队列，能容纳 10 个类型为 unsigned long 的值。
   xQueue1 = xQueueCreate( 10, sizeof( unsigned portLONG ) );
   if( xQueue1 == 0 )
```

```
{
    // 队列创建失败，不能使用。
}

// 创建一个队列，能容纳 10 个指向 AMessage 结构的指针。
// 含有大量数据，需要用指针传递。
xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );
if( xQueue2 == 0 )
{
    // 不能使用未创建的队列。。
}

// ...剩余任务码。}
```

6.5.3. vQueueDelete

此函数用于删除消息队列—释放分配给队列储存消息用的存储。

```
[queue.h]

void vQueueDelete( xQueueHandle xQueue );
```

参数：

xQueue [in] 待删除队列的句柄。

6.5.4. xQueueSend

此宏可调用 xQueueGenericSend()传送一个消息到队列，等效于 xQueueSendToBack()。消息是靠拷贝而非关联列入队列。

```
[queue.h]

portBASE_TYPE xQueueSend(
                            xQueueHandle xQueue,
                            const void * pvItemToQueue,
                            portTickType xTicksToWait
                        );
```

参数：

xQueue [in] 消息所要进入的队列的句柄。

pvItemToQueue [in] 指针，指向待放入队列的消息。创建队列时会定义队列能容纳的消息大小，然后从 pvItemToQueue 拷贝相应字节到消息队列的存储区。

xTicksToWait [in] 如果消息队列已满，需要阻塞任务以等待队列释放可用空间的最长时间。如果消息队列已满，且 xTicksToWait 设置为 0，调用将立即返回。这个时间以心跳周期为单位；如有需要，可用常量 portTICK_RATE_MS 转换真实时间。阻塞时间指定为 portMAX_DELAY 会导致任务无限期阻塞（不会超时）。

返回值：

portBASE_TYPE 如果消息成功进入队列，则返回 pdTRUE；否则，返回 errQUEUE_FULL。

例：

```
struct AMessage
{
   portCHAR ucMessageID;
   portCHAR ucData[ 20 ];
} xMessage;

unsigned portLONG ulVar = 10UL;

void vATask( void *pvParameters )
{
xQueueHandle xQueue1, xQueue2;
struct AMessage *pxMessage;

   // 创建一个队列，能容纳 10 个类型为 unsigned long 的值。
   xQueue1 = xQueueCreate( 10, sizeof( unsigned portLONG ) );

   // 创建一个队列，能容纳 10 个指向 AMessage 结构的指针。
   // 含有大量数据，需要用指针传递。
   xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

   // ...

   if( xQueue1 != 0 )
   {
       // 发送一个 unsigned long 值。如果没有可用空间，
       // 等待 10 次心跳，直到有可用空间。
       if( xQueueSend( xQueue1, ( void * ) &ulVar,
     ( portTickType ) 10 ) != pdPASS )
       {
           // 等待 10 次心跳后，仍不能将消息放入队列。
       }
   }

   if( xQueue2 != 0 )
   {
       // 发送一个指向 AMessage 对象结构的指针。如果队列已满，
       // 不要阻塞任务。
       pxMessage = & xMessage;
       xQueueSend( xQueue2, ( void * ) &pxMessage, ( portTickType ) 0 );
   }

  // ...剩余任务码。
}
```

6.5.5. xQueueSendToBack

此宏可调用 xQueueGenericSend()把消息送入一个队列的后面，与 xQueueSend()等效。消息是靠拷贝而非关联列入队列。

```
[queue.h]

portBASE_TYPE xQueueSendToBack(
                                    xQueueHandle xQueue,
                                    const void * pvItemToQueue,
                                    portTickType xTicksToWait
                                );
```

参数：

xQueue	[in] 消息所要进入的队列的句柄。
pvItemToQueue	[in] 指针，指向待放入队列的消息。创建队列时会定义队列能容纳的消息大小，然后从 pvItemToQueue 拷贝相应字节到消息队列的存储区。
xTicksToWait	[in] 如果消息队列已满，需要阻塞任务以等待队列释放可用空间的最长时间。如果参数值为 0，则调用将立即返回。这个时间以心跳周期为单位；如有需要，可用常量 portTICK_RATE_MS 转换真实时间。阻塞时间指定为 portMAX_DELAY 会导致任务无限期阻塞（不会超时）。

返回值：

portBASE_TYPE 如果消息成功进入队列，则返回 pdTRUE；否则，返回 errQUEUE_FULL。

例：

```
struct AMessage
{
   portCHAR ucMessageID;
   portCHAR ucData[ 20 ];
} xMessage;

unsigned portLONG ulVar = 10UL;

void vATask( void *pvParameters )
{
xQueueHandle xQueue1, xQueue2;
struct AMessage *pxMessage;

   // 创建一个队列，能容纳 10 个类型为 unsigned long 的值。
   xQueue1 = xQueueCreate( 10, sizeof( unsigned portLONG ) );

   // 创建一个队列，能容纳 10 个指向 AMessage 结构的指针。
   // 含有大量数据，需要用指针传递。
   xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

   // ...

   if( xQueue1 != 0 )
   {
       // 发送一个 unsigned long 值。如果没有可用空间，
```

```
            // 等待 10 次心跳，直到有可用空间。
            if( xQueueSendToBack( xQueue1, ( void * ) &ulVar, ( portTickType
) 10 ) != pdPASS )
            {
                // 等待 10 次心跳后，仍不能将消息放入队列。
            }
        }
        if( xQueue2 != 0 )
        {
            // 发送一个指向 AMessage 对象结构的指针。如果队列已满，
            // 不要阻塞任务。
            pxMessage = & xMessage;
            xQueueSendToBack( xQueue2, ( void * ) &pxMessage,
                    ( portTickType ) 0 );
        }

      // ...剩余任务码。
    }
```

6.5.6. xQueueSendToToFront

此宏可调用 xQueueGenericSend() 将消息送到队列的前面。消息是靠拷贝而非关联列入队列。

```
[queue.h]

portBASE_TYPE xQueueSendToToFront(
                                    xQueueHandle xQueue,
                                    const void * pvItemToQueue,
                                    portTickType xTicksToWait
                                );
```

参数：

xQueue	[in] 消息所要进入的队列的句柄。
pvItemToQueue	[in] 指针，指向待放入队列的消息。创建队列时会定义队列能容纳的消息大小，然后从 pvItemToQueue 拷贝相应字节到消息队列的存储区。
xTicksToWait	[in] 如果消息队列已满，需要阻塞任务以等待队列释放可用空间的最长时间。如果参数值为 0，则调用将立即返回。这个时间以心跳周期为单位；如有需要，可用常量 portTICK_RATE_MS 转换真实时间。阻塞时间指定为 portMAX_DELAY 会导致任务无限期阻塞（不会超时）。

返回值：

portBASE_TYPE	如果消息成功进入队列，则返回 pdTRUE；否则，返回 errQUEUE_FULL。

例：

```
struct AMessage
{
   portCHAR ucMessageID;
   portCHAR ucData[ 20 ];
} xMessage;

unsigned portLONG ulVar = 10UL;

void vATask( void *pvParameters )
{
   xQueueHandle xQueue1, xQueue2;
   struct AMessage *pxMessage;

   // 创建一个队列，能容纳 10 个类型为 unsigned long 的值。
   xQueue1 = xQueueCreate( 10, sizeof( unsigned portLONG ) );

   // 创建一个队列，能容纳 10 个指向 AMessage 结构的指针。
   // 含有大量数据，需要用指针传递。
   xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

   // ...

   if( xQueue1 != 0 )
   {
       // 发送一个 unsigned long 值。如果没有可用空间，
       // 等待 10 次心跳，直到有可用空间。
       if( xQueueSend( xQueue1, ( void * ) &ulVar,
     ( portTickType ) 10 ) != pdPASS )
       {
           // 等待 10 次心跳后，仍不能将消息放入队列。
       }
   }


if( xQueue2 != 0 )
    {
        // 发送一个指向 AMessage 对象结构的指针。如果队列已满，
        // 不要阻塞任务。
        pxMessage = & xMessage;
        xQueueSendToFront( xQueue2, ( void * ) &pxMessage,
               ( portTickType ) 0 );
    }

  // ...剩余任务码。

}
```

6.5.7. xQueueReceive

此宏可调用 xQueueGenericReceive()函数，从队列中接收消息。消息是通过拷贝方式接收的，必须提供足够大的缓存。拷贝到缓存的字节数量是在队列创建时定义的。

[queue.h]

```
portBASE_TYPE xQueueReceive(
                                xQueueHandle xQueue,
                                void *pvBuffer,
                                portTickType xTicksToWait
                            );
```

参数：

pxQueue	[in] 提供消息以供接收的队列的句柄。
pvBuffer	[out] 指针，指向接收数据要拷贝到的缓存。
xTicksToWait	[in] 如果调用时消息队列为空，需要阻塞任务以等待队列有消息可供接收的最长时间。如果队列为空并且 xTicksToWait 设置为 0，函数将立即返回。这个时间以心跳周期为单位；如有需要，可用常量 portTICK_RATE_MS 转换真实时间。阻塞时间指定为 portMAX_DELAY 会导致任务无限期阻塞（不会超时）。

返回值：

portBASE_TYPE	成功接收消息，返回 pdTRUE；否则，返回 pdFALSE。

例：

```
struct AMessage
{
   portCHAR ucMessageID;
   portCHAR ucData[ 20 ];
} xMessage;

xQueueHandle xQueue;

// 用于创建队列和传递值的任务。
void vATask( void *pvParameters )
{
   struct AMessage *pxMessage;

   // 创建一个队列，能容纳 10 个指向 AMessage 结构的指针。
   // 含有大量数据，需要用指针传递。
   xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
   if( xQueue == 0 )
   {
       // 队列创建失败。
   }

   // ...

   // 发送一个指向 AMessage 对象结构的指针。如果队列已满，
   // 不要阻塞任务。
```

```
    pxMessage = & xMessage;
    xQueueSend( xQueue, ( void * ) &pxMessage, ( portTickType ) 0 );

  // ...剩余任务码。
}

// 接收队列消息的任务。
void vADifferentTask( void *pvParameters )
{
    struct AMessage *pxRxedMessage;

    if( xQueue != 0 )
    {
        // 用创建的任务接收消息。如果目前没有消息可接收，
        // 阻塞 10 次心跳的时间。
        if( xQueueReceive( xQueue, &( pxRxedMessage ),
      ( portTickType ) 10 ) )
        {
            // pcRxedMessage 现在指向 vATask 传送的
            AMessage 变量结构。
        }
    }

  // ...剩余任务码。
}
```

6.5.8. xQueuePeek

此宏可调用 xQueueGenericReceive()函数，接收队列消息，但不会将消息从队列中移除。消息是通过拷贝方式接收的，必须提供足够大的缓存。拷贝到缓存的字节数量是在队列创建时定义的。成功接收的消息仍存在队列中，因此下次返回可以继续调用此函数或 xQueueReceive()获取消息。

```
[queue.h]

portBASE_TYPE xQueuePeek(
                            xQueueHandle xQueue,
                            void *pvBuffer,
                            portTickType xTicksToWait
                        );
```

参数：

xQueue	[in] 提供消息以供接收的队列的句柄。
pvBuffer	[out] 指针，指向接收数据要拷贝到的缓存。缓存必须大于等于队列创建时定义的容量。
xTicksToWait	[in] 如果调用时消息队列为空，需要阻塞任务以等待队列有消息可供接收的最长时间。这个时间以心跳周期为单位；如有需要，可用常量 portTICK_RATE_MS 转换真实时间。阻塞时间指定为 portMAX_DELAY 会导致任务无限期阻塞（不会超时）。

返回值：

portBASE_TYPE	成功接收（偷窥）消息，返回 pdTRUE；否则，返回 pdFALSE。

例：

```
struct AMessage
{
   portCHAR ucMessageID;
   portCHAR ucData[ 20 ];
} xMessage;

xQueueHandle xQueue;

// 用于创建队列和传递值的任务。
void vATask( void *pvParameters )
{
   struct AMessage *pxMessage;

   // 创建一个队列，能容纳 10 个指向 AMessage 结构的指针。
   // 含有大量数据，需要用指针传递。
   xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
   if( xQueue == 0 )
   {
       // 队列创建失败。
   }

   // ...

   // 发送一个指向 AMessage 对象结构的指针。如果队列已满，
   // 不要阻塞任务。
   pxMessage = & xMessage;
   xQueueSend( xQueue, ( void * ) &pxMessage, ( portTickType ) 0 );

   // ...剩余任务码。
}

// 从队列偷窥消息的任务。
void vADifferentTask( void *pvParameters )
{
   struct AMessage *pxRxedMessage;

   if( xQueue != 0 )
   {
       // 在所创建的任务中偷窥消息。如果目前没有消息可接收，
       // 阻塞 10 次心跳的时间。
       if( xQueuePeek( xQueue, &( pxRxedMessage ), ( portTickType ) 10
) )
       {
           // pcRxedMessage 现在指向由 vATask 传送的 AMessage 变量结构
           // 但是消息仍在队列中。
       }
   }
   // ...剩余任务码。
}
```

6.6. 信号量/互斥信号量

6.6.1. vSemaphoreCreateBinary

此宏以现成的消息队列的机制创建一个二进制信号量。此消息队列长度为 1，因为这是一个二进制信号量。无需存储任何数据，只需知道队列为空或为满，因此每个消息的数据大小为 0 字节。二进制信号量和互斥量非常相似但还是有些区别。互斥量包括一个优先级继承机制；二进制信号量则不包括，因此二进制信号量更适合实现任务与任务之间、任务与中断之间的同步，而互斥量则适合实现简单的互斥。获得二进制信号量后，不需要发回。所以，为了实现任务同步，可以让一个任务/中断连续发送信号量，由另一个任务/中断不停接收信号量。如果一个优先级更高的任务（A）试图获取互斥量失败时，成功取走互斥量的另一个任务（B）的优先级有可能被抬升。任务 B 继承了任务 A 的优先级。这意味着互斥量必须被发回，否则高优先级的任务将永远得不到互斥量，且低优先级的任务将永远享有优先继承权。xSemaphoreTake() 文档页提供了一个用互斥量实现互斥的实例。互斥量和二进制信号量均标记为 xSemaphoreHandle 类型的变量，且均可用在任何含有此类参数的 API 函数中。

```
[semphr.h]

vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore );
```

参数：

xSemaphore [out] 待创建的信号量的句柄。必须是 xSemaphoreHandle 类型。

例：

```
xSemaphoreHandle xSemaphore;

void vATask( void * pvParameters )
{
   // 调用 vSemaphoreCreateBinary ()前不得使用信号量。
   // 此宏间接传递变量。
   vSemaphoreCreateBinary( xSemaphore );

   if( xSemaphore != NULL )
   {
       // 信号量创建成功。
       // 信号量目前可用。
   }
}
```

6.6.2. xSemaphoreCreateCounting

此宏以现成的消息队列的机制创建一个计数器信号量。

计数器信号量通常有两种方法。

1. 事件计数

   假定每一次事件发生时，事件处理器将产生一个信号量（增加信号量计数）；事件处理器每处理一个事件，处理器任务将取走一个信号量（减少信号计数）。因此计数值表示的是已发生事件数量和已处理事件数量之间的差异。这种情况下，建议将信号量初始值设置为 0。

2. 资源管理

假定计数值代表可用资源的数量。为了获取一个资源的控制权，任务必须先获取一个信号量—减去信号量的计数器值。当值达到 0 时，表示目前没有可用资源。当一个任务结束时，应返还信号量—增加信号量计数值。这种情况下，建议将信号量初始值设置为最大计数值，表示所有资源都处于空闲状态。

[semphr.h]

```
xSemaphoreHandle xSemaphoreCreateCounting
(
  unsigned portBASE_TYPE uxMaxCount,
  unsigned portBASE_TYPE uxInitialCount
);
```

参数：

uxMaxCount [in] 能达到的最大计数值。当信号量达到此值时，任务将不能再发计数器信号量。

uxInitialCount [in] 创建信号量时分配给计数器信号量的初始值。

返回值：

xSemaphoreHandle 待创建的信号量的句柄。如果信号量无法创建则返回 NULL。

例：

```
void vATask( void * pvParameters )
{
   xSemaphoreHandle xSemaphore;

   // 调用 xSemaphoreCreateCounting()前不得使用信号量。
   // 信号量最大计数值为 10，
   // 并且分配的初始值应为 0。
   xSemaphore = xSemaphoreCreateCounting( 10, 0 );

   if( xSemaphore != NULL )
   {
       // 信号量创建成功。
       // 信号量目前可用。
   }
}
```

6.6.3. xSemaphoreCreateMutex

此宏以现成的消息队列的机制创建一个互斥信号量。访问此宏创建的互斥信号量，须使用 SemaphoreTake() 和 xSemaphoreGive()，不应使用 xSemaphoreTakeRecursive()和 xSemaphoreGiveRecursive()。

互斥量和二进制信号量非常相似但还是有些区别。互斥量包括一个优先级继承机制，二进制信号量则不包括，因此二进制信号量更适合实现任务与任务之间、任务与中断之间的同步，互斥量则适合实现简单的互斥。如果一个优先级更高的任务（A）试图获取互斥量失败时，成功取走互斥量的另一个任务（B）的优先级有可能被抬升。任务 B 继承了任务 A 的优先级。这意味着互斥量必须被发回，否则高优先级的任务将永远得不到互斥量，且低优先级的任务将永远享有优先继承权。xSemaphoreTake() 文档页提供了一个用互斥量实现互斥的实例。获得二进制信号量后，不需要发回。所以，为了实现任务同步，可以让一个任务/中断连续发送信号量，由另一个任务/中断不停接收信号量。互斥量和二进制信号量均标记为 xSemaphoreHandle 类型的变量，且均可用在任何含有此类参数的 API 函数中。

```
[semphr.h]

xSemaphoreHandle xSemaphoreCreateMutex( void );
```

返回值：

xSemaphoreHandle 待创建的信号量的句柄。必须是 xSemaphoreHandle 类型。

例：

```
xSemaphoreHandle xSemaphore;

void vATask( void * pvParameters )
{
   // 调用 xSemaphoreCreateMutex()前
   // 不得使用互斥量。返回创建的互斥量。
   xSemaphore = xSemaphoreCreateMutex();

   if( xSemaphore != NULL )
   {
       // 信号量创建成功。
       // 信号量目前可用。
   }
}
```

6.6.4. xSemaphoreCreateRecursiveMutex

此宏以现成的消息队列的机制创建一个递归互斥量。用此宏创建的互斥信号量可以用宏 xSemaphoreTakeRecursive()和 xSemaphoreGiveRecursive()进行访问。不应使用宏 xSemaphoreTake()和 xSemaphoreGive()。一个递归互斥量可以被拥有者重复获取。直到拥有者为每次请求成功地调用 xSemaphoreGiveRecursive() 后，互斥量才再次可用。例如，一个任务成功获取 5 次互斥量，则只有拥有者 5 次返还同一个信号量后，互斥量才再次可用。此类信号量使用优先级继承机制，所以任务获取信号量后，一旦不再需要该信号量，任务总是并必须返还信号量。

```
[semphr.h]

xSemaphoreHandle xSemaphoreCreateRecursiveMutex( void );
```

返回值：

xSemaphoreHandle 创建的互斥信号量的 xSemaphoreHandle。必须是 xSemaphoreHandle 类型。

例：

```
xSemaphoreHandle xMutex;

void vATask( void * pvParameters )
{
   // 调用 xSemaphoreCreateMutex()前不得使用信号量。
   // 此宏间接传递变量。
   xMutex = xSemaphoreCreateRecursiveMutex();

   if( xMutex != NULL )
   {
       // 互斥类信号量创建成功。
       // 互斥量目前可用。
   }
}
```

6.6.5. xSemaphoreTake

此宏用于获取信号量。这个信号量必须事先调用 vSemaphoreCreateBinary()、xSemaphoreCreateMutex() 或 xSemaphoreCreateCounting()创建完成。

```
[semphr.h]

signed portBASE_TYPE xSemaphoreTake
(
 xSemaphoreHandle xSemaphore,
 portTickType xBlockTime
);
```

参数：

xSemaphore [in] 创建信号量时获取的信号量的句柄。

xBlockTime [in] 以心跳为单位，表示等待信号量可获取所需的时间。可用 portTICK_RATE_MS 转换为真实时间。**xBlockTime** 设置为 0 时，可以查询信号量。阻塞时间指定为 portMAX_DELAY 会导致任务无限期阻塞（不会超时）。

返回值：

signed portBASE_TYPE 信号量获取成功，则返回 pdTRUE。如果 xBlockTime 过期时仍无信号量可获取，则返回 pdFALSE。

例：

```
xSemaphoreHandle xSemaphore = NULL;
// 创建信号量的任务。
void vATask( void * pvParameters )
{
   // 创建信号量以保护共享资源。用
   // 信号量实现互斥时，需要创建互斥信号量，
   // 而不是二进制信号量。
   xSemaphore = xSemaphoreCreateMutex();
}

// 使用信号量的任务。
void vAnotherTask( void * pvParameters )
{
   // ...做其他事情。

   if( xSemaphore != NULL )
   {
       // 确认能否获取信号量。如果信号量不可获取，
       // 则等待 10 次心跳，再查看是否可获取。
       if( xSemaphoreTake( xSemaphore, ( portTickType ) 10 ) == pdTRUE
)
       {
           // 可获取信号量并访问
           // 共享资源。

           // ...

           // 已经完成对共享资源的访问。释放
           // 信号量。
           xSemaphoreGive( xSemaphore );
       }
       其他
       {
           // 不能获取信号量，所以不能对
           // 共享资源进行安全访问。
       }
   }
}
```

6.6.6. xSemaphoreTakeRecursive

此宏用于获取 xSemaphoreCreateRecursiveMutex()创建的递归互斥量，不能获取 xSemaphoreCreateMutex()创建的互斥量。为每一个成功的获取请求调用 xSemaphoreGiveRecursive() 后，拥有者可重复获取递归互斥量。例如，一个任务成功获取 5 次互斥量，则只有拥有者 5 次返还同一个信号量后，互斥量才再次可用。

```
[semphr.h]

portBASE_TYPE xSemaphoreTakeRecursive(
                                      xSemaphoreHandle xMutex,
                                      portTickType xBlockTime
                                     );
```

参数：

xMutex [in] 待获取队列的句柄。该句柄是由 xSemaphoreCreateRecursiveMutex()返回的。

xBlockTime [in] 以心跳为单位，表示等待信号量可获取所需的时间。可用 portTICK_RATE_MS 转换为真实时间。xBlockTime 设置为 0 时，可以查询信号量。如果任务已经拥有了这个信号量，则 xSemaphoreTakeRecursive()会立即返回，无论 xBlockTime 的值是多少。

返回值：

portBASE_TYPE 信号量获取成功，则返回 pdTRUE。如果 xBlockTime 过期时仍无信号量可获取，则返回 pdFALSE。

例：

```
xSemaphoreHandle xMutex = NULL;
// 创建互斥量的任务。
void vATask( void * pvParameters )
{
   // 创建互斥量以保护共享资源。
   xMutex = xSemaphoreCreateRecursiveMutex();
}

// 使用互斥量的任务。
void vAnotherTask( void * pvParameters )
{
   // ...做其他事情。

   if( xMutex != NULL )
   {
       // 确认能否获取互斥量。如果不可获取，
       // 则等待 10 次心跳，再查看是否可获取。
       if( xSemaphoreTakeRecursive( xMutex, ( portTickType ) 10 ) ==
pdTRUE )
       {
           // 可获取互斥量并访问
           // 共享资源。

           // ...
```

```
// 由于代码性质的原因，
// 须由同一个互斥量访问 xSemaphoreTakeRecursive()。对于真正的代码，
// 这些将不是简单的有序调用，
            // 因为这样毫无意义。相反，这些调用很可能嵌在
            // 一个更复杂的调用结构里。
            xSemaphoreTakeRecursive( xMutex, ( portTickType ) 10 );
            xSemaphoreTakeRecursive( xMutex, ( portTickType ) 10 );

            // 这个互斥量已经被获取 3 次，因此
            //被返还三次后，才可再次获取。
            //同样，在真正的代码里，这些将不是简单的有序调用，
// 而是嵌在一个更复杂的
            // 调用结构里。此处描述仅为辅助理解。
            xSemaphoreGiveRecursive( xMutex );
            xSemaphoreGiveRecursive( xMutex );
            xSemaphoreGiveRecursive( xMutex );

            // 现在其他任务可以获取该互斥量。
        }
        其他
        {
            // 不能获取互斥量，因此不能对
            // 共享资源进行安全访问。
        }
    }
}
```

6.6.7. xSemaphoreGive

此宏用于释放信号量。这个信号量必须事先调用 vSemaphoreCreateBinary()、xSemaphoreCreateMutex() 或 xSemaphoreCreateCounting()创建完成，并且可用 sSemaphoreTake()获取。此宏不能操作 xSemaphoreCreateRecursiveMutex()创建的信号量。

```
[semphr.h]

signed portBASE_TYPE xSemaphoreGive( xSemaphoreHandle xSemaphore );
```

参数：

xSemaphore	[in] 被释放队列的句柄。这是创建信号量时返还的句柄。

返回值：

signed portBASE_TYPE	信号量释放成功，则返回 pdTRUE。如果发生错误，则返回 pdFALSE。须用消息队列实现信号量。如果队列空间不足，则报错，表示信号量获取方式错误。

例：

```
xSemaphoreHandle xSemaphore = NULL;

void vATask( void * pvParameters )
{
   // 创建信号量以保护共享资源。用
   // 信号量实现互斥时，需要创建互斥信号量，
   // 而不是二进制信号量。
   xSemaphore = xSemaphoreCreateMutex();

   if( xSemaphore != NULL )
   {
       if( xSemaphoreGive( xSemaphore ) != pdTRUE )
       {
           // 没有获取信号量，也就不用返还。
           // 因此，此次调用将失败。
       }
       // 获取信号量—如果信号量目前不可获取，
       // 先别阻塞。
       if( xSemaphoreTake( xSemaphore, ( portTickType ) 0 ) )
       {
           // 已经获取信号量，可以访问共享资源。
           // ...
           // 完成访问后，须释放
           // 信号量。
           if( xSemaphoreGive( xSemaphore ) != pdTRUE )
           {
               // 已获取信号量，
               // 调用可成功。
           }
       }
   }
}
```

6.6.8. xSemaphoreGiveRecursive

此宏用于释放递归互斥量。该互斥量必须事先调用 xSemaphoreCreateRecursiveMutex()创建完成。此宏不能操作 xSemaphoreCreateMutex()创建的互斥量。一个递归互斥量可以被拥有者重复获取。直到拥有者为每次请求成功地调用 xSemaphoreGiveRecursive() 后，互斥量才再次可用。例如，一个任务成功获取 5 次互斥量，则只有拥有者 5 次返还同一个信号量后，互斥量才再次可用。

```
[semphr.h]

portBASE_TYPE xSemaphoreGiveRecursive( xSemaphoreHandle xMutex );
```

参数：

xMutex [in] 被释放互斥量的句柄。该句柄是由 xSemaphoreCreateRecursiveMutex()返回的。

返回值：

portBASE_TYPE 信号量释放成功，则返回 pdTRUE。

例：

```
    xSemaphoreHandle xMutex = NULL;

    // 创建互斥量的任务。
    void vATask( void * pvParameters )
    {
       // 创建互斥量以保护共享资源。
       xMutex = xSemaphoreCreateRecursiveMutex();
    }

    // 使用互斥量的任务。
    void vAnotherTask( void * pvParameters )
    {
       // ...做其他事情。

       if( xMutex != NULL )
       {
           // 确认能否获取互斥量。如果不可获取，
           // 则等待 10 次心跳，再查看是否可获取。
           if( xSemaphoreTakeRecursive( xMutex, ( portTickType ) 10 ) ==
pdTRUE )
           {
               // 可获取互斥量并访问
               // 共享资源。

               // ...
               // 由于代码性质的原因，
               // 须由同一个互斥量访问 xSemaphoreTakeRecursive()。对于真正的代码，
               // 这些将不是简单的有序调用，
               // 因为这样毫无意义。相反，这些调用很可能嵌在
               // 一个更复杂的调用结构里。
               xSemaphoreTakeRecursive( xMutex, ( portTickType ) 10 );
               xSemaphoreTakeRecursive( xMutex, ( portTickType ) 10 );

               // 这个互斥量已经被获取 3 次，因此
               //返还 3 次后，才再次可获取。
               //同样，在真正的代码里，这些将不是简单的有序调用，
               // 相反，
               //调用 xSemaphoreGiveRecursive() 同样调用栈
               // 进行松散调用。此处描述仅为辅助理解。
               xSemaphoreGiveRecursive( xMutex );
               xSemaphoreGiveRecursive( xMutex );
               xSemaphoreGiveRecursive( xMutex );
```

```
            // 现在其他任务可以获取该互斥量。
        }
        其他
        {
            // 不能获取互斥量，因此不能对
            // 共享资源进行安全访问。
        }
    }
}
```

6.7. 软件计时器

计时器功能是由计时器服务/守护进程任务提供的。很多公用的 FreeRTOS 计时器的 API 函数通过定时器命令消息队列向定时器服务任务发送命令。应用代码不能访问定时器命令消息队列，它专属于内核。定时器命令消息队列的长度是用配置常量 configTIMER_QUEUE_LENGTH 设置的。定时器的任务优先级是用配置常量 configTIMER_TASK_PRIORITY 设置的。

6.7.1. xTimerCreate

此函数用于创建软件定时器实例。为新定时器分配所需存储，初始化新定时器的内部状态，并返回可以关联新定时器的句柄。创建的定时器处于休眠状态。可以用 API 函数 xTimerStart()，xTimerReset() 和 xTimerChangePeriod() 激活定时器。

```
[timers.h]

xTimerHandle xTimerCreate( const signed char *pcTimerName,
                           portTickType xTimerPeriod,
                           unsigned portBASE_TYPE uxAutoReload,
                           void * pvTimerID,
                           tmrTIMER_CALLBACK pxCallbackFunction );
```

参数：

pcTimerName	[in] 分配给定时器的文本名。文本名只是方便调试所用。内核关联定时器的时候只用句柄，从不用文本名。
xTimerPeriod	[in] 定时器周期。这个时间以心跳周期为单位；如有需要，可用常量 portTICK_RATE_MS 转换以毫秒为单位的时间。例： 如果定时器在 100 次心跳后必须终止，xTimerPeriod 应设为 100。另一种情况是，configTICK_RATE_HZ 小于等于 1000，而定时器在 500ms 后必须过期，则 xTimerPeriod 应设为（500/portTICK_RATE_MS）。
uxAutoReload	[in] 如果 uxAutoReload 设为 pdTRUE， 定时器将按照参数 xTimerPeriod 设置的频率重复终止。如果 uxAutoReload 设为 pdFALSE，则为单稳定时器，终止一次后就进入休眠状态。
pvTimerID	[in] 分配给正在创建的定时器的标识符。典型地，在同一个回调函数分配给多个定时器的情况下，此标识符可以在定时器回调函数中标识终止的定时器。
pxCallbackFunction	[in] 定时器终止时要调用的函数。回调函数必须拥有 tmrTIMER_CALLBACK 定义的原型，即“void vCallbackFunction(xTimerHandle xTimer);”。

返回值：

xTimerHandle	如果定时器创建成功，则返回一个句柄给新建定时器。如果定时器创建失败，则返回 0。失败原因可能是剩余的 FreeRTOS 堆不足以分配定时器结构，或者定时器周期设置为 0。

例：

```
#define NUM_TIMERS 5

/* 包含新建定时器的句柄的数组。*/
xTimerHandle xTimers[ NUM_TIMERS ];

/* 包含每个定时器终止次数的数组。*/
long lExpireCounters[ NUM_TIMERS ] = { 0 };

/* 定义将被多个定时器实例使用的回调函数。
回调函数用于计算相关定时器的终止次数，
一旦定时器终止满 10 次，
回调函数将终止定时器。*/
void vTimerCallback( xTimerHandle pxTimer )
{
long lArrayIndex;
const long xMaxExpiryCountBeforeStopping = 10;

    /* 如果参数 pxTimer 的值是 NULL，也可以有选择的执行一些操作。*/
    configASSERT( pxTimer );

    /* 哪个定时器终止?*/
    lArrayIndex = ( long ) pvTimerGetTimerID( pxTimer );

    /* 增加 pxTimer 的终止次数。*/
    lExpireCounters[ lArrayIndex ] += 1;

    /* 满 10 次，则使定时器停止运行。*/
    if( lExpireCounters[ lArrayIndex ] == xMaxExpiryCountBeforeStopping
)
    {
        /* 从定时器回调函数调用定时器 API 函数时，
        不要使用阻塞时间，因为这样可能造成死锁。*/
        xTimerStop( pxTimer, 0 );
    }
}

void main( void )
{
long x;

    /* 创建并开启一些定时器。开启调度器前开启定时器，意味着
    调度器一旦开启，
    定时器就开始运行。*/
    for( x = 0; x < NUM_TIMERS; x++ )
```

```
    {
        xTimers[ x ] = xTimerCreate(
                    "Timer", /* 仅是一个文本名，内核不会使用。*/
                    ( 100 * x ), /* 以心跳为单位的定时器周期。*/
                    pdTRUE, /* 定时器终止后将自动重新载入。*/
                    ( void * ) x, /* 为每个定时器分配一个唯一的 ID，等于它的数
组索引。*/
                    vTimerCallback /* 每个定时器终止时都会调用同一个回调函数。
*/
                                );

        if( xTimers[ x ] == NULL )
        {
            /* 没有创建该定时器。*/
        }
        其他
        {
            /* 启动定时器。不管有没有设置，阻塞时间都会被忽略，因为调度器还没有开
启。*/
            if( xTimerStart( xTimers[ x ], 0 ) != pdPASS )
            {
                /* 定时器不能被设为激活状态。*/
            }
        }
    }

    /* ...
    在此处创建任务。
    ...*/

    /* 调度器开启后，处于激活状态的定时器就开始运行。*/
    xTaskStartScheduler();

    /* 不应到这一步。*/
    for( ;; );
}
```

6.7.2. xTimerIsTimerActive

此函数用于确认定时器处于激活或休眠状态。

以下情况下，定时器处于休眠状态：

1. 定时器已创建但还没有开启，或者
2. 定时器终止一次后再也没有重启。

```
[timers.h]

portBASE_TYPE xTimerIsTimerActive( xTimerHandle xTimer );
```

参数：

xTimer [in] 正在查询的定时器。

返回值：

portBASE_TYPE 如果定时器休眠，将返回 pdFALSE。如果定时器已激活，将返回 pdFALSE 之外的值。

例：

```
/* 假设已经创建 xTimer。*/
 void vAFunction( xTimerHandle xTimer )
 {
     if( xTimerIsTimerActive( xTimer ) != pdFALSE )
 /* 或采用更简单的等效语句 "if( xTimerIsTimerActive( xTimer ) )" */
     {
         /* xTimer 已激活，执行操作。*/
     }
     其他
     {
         /* xTimer 未激活，执行其他操作。*/
     }
 }
```

6.7.3. xTimerStart

此函数用于开启先前用 xTimerCreate() API 函数创建的定时器。如果定时器已开启并且已激活，则 xTimerStart()的功能与 xTimerReset() API 函数一样。开启定时器前，确保定时器处于激活状态。同时，如果没有停止，删除，重置时器，则在 xTimerStart()被调用 后'n'次心跳后，定时器相关的回调函数将被调用。'n'是定时器定义的周期。

```
[timers.h]

portBASE_TYPE xTimerStart( xTimerHandle xTimer, portTickType xBlockTime
);
```

参数：

xTimer [in] 正开启/重启的定时器的句柄。

xBlockTime [in] 时长，以心跳为单位，表示调用 xTimerStart() 时消息队列已满的情况下，任务应被阻塞的时间，以等待开启命令成功发送到定时器命令消息队列。

返回值：

portBASE_TYPE 如果 xBlockTime 超时后，开启命令仍没有发送到定时器命令消息队列，则返回 pdFAIL。如果命令成功发送到定时器命令消息队列，则返回 pdPASS。尽管定时器终止时间与成功调用 xTimerStart()的时间相关联，但实际的命令处理时间是由定时器服务/守护进程任务相对系统内部其他任务的优先级而定的。

6.7.4. xTimerStop

此函数用于终止先前用 xTimerStart()、xTimerReset()或 xTimerChangePeriod()API 函数开启的定时器。

```
[timers.h]

portBASE_TYPE xTimerStop( xTimerHandle xTimer, portTickType xBlockTime
);
```

参数：

xTimer [in] 被终止的定时器的句柄。

xBlockTime [in] 时长，以心跳为单位，表示调用 xTimerStop() 时消息队列已满的情况下，任务应被阻塞的时间，以等待终止命令成功发送到定时器命令消息队列。

返回值：

portBASE_TYPE 如果 xBlockTime 超时后，终止命令仍没有发送到定时器命令消息队列，则返回 pdFAIL。如果命令成功发送到定时器命令消息队列，则返回 pdPASS。实际的命令处理时间是由定时器服务/守护进程任务相对系统内部其他任务的优先级而定的。

6.7.5. xTimerChangePeriod

此函数用于改变 xTimerCreate() API 函数创建的定时器的周期。可以用 xTimerChangePeriod() 改变激活或休眠的定时器的周期。

```
[timers.h]
portBASE_TYPE xTimerChangePeriod( xTimerHandle xTimer,
                                  portTickType xNewPeriod,
                                  portTickType xBlockTime );
```

参数：

xTimer [in] 被改变周期的定时器的句柄。

xNewPeriod [in] xTimer 的新周期。这个时间以心跳周期为单位；如有需要，可用常量 portTICK_RATE_MS 转换以毫秒为单位的时间。例：
如果定时器在 100 次心跳后必须终止，xNewPeriod 应设为 100。另一种情况是，configTICK_RATE_HZ 小于等于 1000，而定时器在 500ms 后必须过期，则 xNewPeriod 应设为（500/portTICK_RATE_MS）。

xBlockTime [in] 时长，以心跳为单位，表示调用 xTimerChangePeriod() 时消息队列已满的情况下，调用者任务应被阻塞的时间，以等待周期更改命令成功发送到定时器命令消息队列。

返回值：

portBASE_TYPE 如果 xBlockTime 超时后，周期更改命令仍没有发送到定时器命令消息队列，则返回 pdFAIL。如果命令成功发送到定时器命令消息队列，则返回 pdPASS。实际的命令处理时间是由定时器服务/守护进程任务相对系统内部其他任务的优先级而定的。

例：

```
/* 假设已经创建 xTimer。如果调用
xTimer 关联的定时器时，该定时器已激活，
定时器将被删除。如果调用 xTimer 关联的定时器时，该定时器未激活，
则将设置定时器的周期为 500ms，并开启
定时器。*/
void vAFunction( xTimerHandle xTimer )
{
    if( xTimerIsTimerActive( xTimer ) != pdFALSE )
/* 或采用更简单的等效语句 "if( xTimerIsTimerActive( xTimer ) )" */
    {
        /* xTimer 已激活 - 删除该定时器。*/
        xTimerDelete( xTimer );
    }
    其他
    {
       /* xTimer 未激活，则将其周期改为 500ms。同时，
        定时器也将开启。如果周期更改命令
        没有即时发送到定时器命令消息队列，
        则最多等待 100 次心跳的时间。*/
        if( xTimerChangePeriod( xTimer, 500 / portTICK_RATE_MS, 100 ) ==
pdPASS )
        {
            /* 命令发送成功。*/
        }
        其他
        {
         /* 等待 100 次心跳的时间后，
            命令仍未发送成功。执行适当操作。*/
        }
    }
}
```

6.7.6. xTimerDelete

此函数用于删除先前用 xTimerCreate() API 函数创建的定时器。

```
[timers.h]

portBASE_TYPE xTimerDelete( xTimerHandle xTimer, portTickType xBlockTime
);
```

参数：

xTimer [in] 待删除的定时器的句柄。

xBlockTime [in] 时长，以心跳为单位，表示调用 xTimerDelete() 时消息队列已满的情况下，调用者任务应被阻塞的时间，以等待删除命令成功发送到定时器命令消息队列。

返回值：

portBASE_TYPE 如果 xBlockTime 超时后，删除命令仍没有发送到定时器命令消息队列，则返回 pdFAIL。如果命令成功发送到定时器命令消息队列，则返回 pdPASS。实际的命令处理时间是由定时器服务/守护进程任务相对系统内部其他任务的优先级而定的。

6.7.7. xTimerReset

此函数用于重置先前用 xTimerCreate() API 函数创建的定时器。如果定时器已开启并且已激活，那么 xTimerReset()会重新评估定时器的终止时间，即定时器的终止时间将与 xTimerReset()被调用的时间相关联。如果定时器休眠，则 xTimerReset()具有与 xTimerStart() API 函数一样的功能。重置定时器，确保定时器处于激活状态。如果没有停止，删除，重置定时器，则在 xTimerReset()被调用 后'n'次心跳后，定时器相关的回调函数将被调用。'n'是定时器定义的周期。

```
[timers.h]
portBASE_TYPE xTimerReset( xTimerHandle xTimer, portTickType xBlockTime
);
```

参数：

xTimer [in] 被重置/开启/重启的定时器的句柄。

xBlockTime [in] 时长，以心跳为单位，表示调用 xTimerReset()时消息队列已满的情况下，调用者任务应被阻塞的时间，以等待重置命令成功发送到定时器命令消息队列。

返回值：

portBASE_TYPE 如果 xBlockTime 超时后，重置命令仍没有发送到定时器命令消息队列，则返回 pdFAIL。如果命令成功发送到定时器命令消息队列，则返回 pdPASS。尽管定时器终止时间与成功调用 xTimerReset()的时间相关联，但实际的命令处理时间是由定时器服务/守护进程任务相对系统内部其他任务的优先级而定的。

例：

```
/* 按任意键，LCD 背光将打开。5 秒内无按键操作，
LCD 背光将关闭。这种情况下，
用的是单稳定时器。*/

xTimerHandle xBacklightTimer = NULL;

/* 分配给单稳定时器的回调功能。这种情况下，
不会用到该参数。*/
void vBacklightTimerCallback( xTimerHandle pxTimer )
{
    /* 定时器超时，因此距上次按键操作一定超过了
    5 秒。关闭 LCD 背光。*/
    vSetBacklightState( BACKLIGHT_OFF );
}

/* 按键事件的句柄。*/
void vKeyPressEventHandler( char cKey )
{
    /* 确保 LCD 背光亮，
    然后重置 5 秒钟无按键操作的情况下
    负责关闭背光的定时器。如果命令发送不成功，
```

```
        则等待 10 次心跳的时间。*/
        vSetBacklightState( BACKLIGHT_ON );
        if( xTimerReset( xBacklightTimer, 10 ) != pdPASS )
        {
          /* 重置命令执行失败。执行
             适当操作。*/
        }

        /* 进行其他的按键处理。*/
}

void main( void )
{
long x;

        /* 创建并开启 5 秒钟无按键操作的情况下
        负责关闭背光的定时器。*/
        xBacklightTimer = xTimerCreate(
          "BacklightTimer",     /* 仅是一个文本名，内核不会使用。*/
          ( 5000 / portTICK_RATE_MS), /* 以心跳为单位的定时器周期。*/
          pdFALSE,                    /* 这是一个单稳定时器。*/
          0,                    /* 此 id 未使用在任何回调中，因此可以采用任意值。*/
          vBacklightTimerCallback /* 负责关闭 LCD 背光的回调函数。*/
                                 );

        if( xBacklightTimer == NULL )
        {
            /* 没有创建该定时器。*/
        }
        其他
        {
           /* 启动定时器。无论有无指定，
            阻塞时间都会被忽略，因为调度器还未
            开启。*/
            if( xTimerStart( xBacklightTimer, 0 ) != pdPASS )
            {
                /* 定时器不能被设为激活状态。*/
            }
        }

        /* ...
         在此处创建任务。
        ...*/

       /* 开启调度器将使定时器开始运行，
        因为调度器已激活。*/
        xTaskStartScheduler();

        /* 不应到这一步。*/
        for( ;; );
}
```

6.7.8. pvTimerGetTimerID

此函数用于返还分配给定时器的 ID。调用 xTimerCreate()创建定时器时，其参数 pvTimerID 用于给定时器分配 ID。如果分配同一个回调函数给多个定时器时，可以用定时器 ID 在回调函数里标识实际上终止的定时器。

```
[timers.h]

void *pvTimerGetTimerID( xTimerHandle xTimer );
```

参数：

xTimer [in] 正被查询的定时器。

返回值：

void * 分配给被查询的定时器的 ID。