Advanced Card Systems Ltd.
Card & Reader Technologies

ACR89U Handheld Smart Card Reader

Application Programming Interface

Subject to change without prior notice

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

Table of Contents

Figures

No table of figures entries found.

1.0. Introduction

ACR89 terminal is equipped with 32-bit CPU running the embedded FreeRT Operating System (FreeRTOS) Kernel. FreeRTOS kernel is a scale-able real time kernel designed specifically for small, embedded system. It is open source, portable, free to download and free to deploy software. It can be used in commercial application without any requirement to expose your proprietary source code. It is very portable code structure predominantly written in C language.

This document provides the API (Application Programming Interface) commands to develop standalone application program specifically for ACR89. Application software developers can make use of these APIs to develop their smart-card related application.

1.1. Scope and Limitation

This API document provides a detailed guide on implementing commands for the smart card reader keys and displays, as well as the FreeRTOS feature in ACR89.

1.2. Reference

Refer to this link for the details about the FreeRTOS software environment.

- http://www.freertos.org/

2.0. Compiler Independent Data Types

Data type	Size	Effective range of a number
UINT8	1byte	0 to 255
UINT16	2bytes	0 to 65535
UINT32	4bytes	0 to 4294967295
UINT64	8bytes	0 to 18446744073709551615
INT8	1byte	-128 to 127
INT16	2bytes	-32768 to 32767
INT32	4bytes	-2147483648 to 2147483647
INT64	8bytes	-9223372036854775808 to 9223372036854775807
REAL32	4bytes	1.175e-38 to 3.403e+38 (normalized number)
REAL64	8bytes	2.225e-308 to 1.798e+308 (normalized number)
BOOLEAN	1byte	TRUE/FALSE
UCHAR	1byte	0 to 255
SCHAR	1byte	-128 to 127

Handling of 64-bits integer data type constants requires the suffix LL or ll (INT64 type) or ULL or ull (UINT64 type). If this suffix is not present, a warning is assumed, since the compiler may not be able to recognize long-type constants as such.

Example: INT64 ll_val;

ll_val = 0x1234567812345678;

- Warning: Integer constant is too large for “long” type

Ll_val = 0x1234567812345678LL;

- OK

3.0. Smart Card API Functions

3.1. Firmware Version Records

Kiwi 2012-04-17 v1.2

HW-D2-01-00 FreeRTOS V7.0.1

3.1.1. Hardware Code : HW-AA-BB-CC

Where:

- AA : Large version
  - D1 = ACR89 V1-V3 PCBA
  - D2 = ACR89 V4-V5 PCBA
- BB : Configuration
  - 01 = ACR89U-A1 (Basic)
  - 02 = ACR89U-A2 (Contactless with Felica Support)
  - 03 = ACR89U-B1 (Fingerprint Swipe)
  - 04 = ACR89U-A3 (Contactless)
  - 05 = ACR89U-A4 (Bluetooth)
- CC : Small version

3.1.2. Production Firmware Code: XYYY

- X : Configuration
  - A = ACR89U (Standard)
  - B = ACR89U-CL (Contactless)
  - C = ACR89U-FP (Fingerprint)
- YYY : Released Version Code

3.2. Data Structures

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
```

Used by SCard_Manager_Msg_Receive:

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

Data Member	Value	Description
ucMessage	Bit field 0 to 2	Smart card message type: 0 -- SCARD_MSG_UNKNOWN (undefined message) 1 -- SCARD_MSG_CARDINSERTED (card inserted into slot) 2 -- SCARD_MSG_CARDREMOVED (card removed from slot)
ucData	Bit field 0 or 1	Index of smart card slot where the message comes from: 0 -- 1st slot 1 -- 2nd slot

3.3. Functions

3.3.1. SCard_Manager_Msg_Receive

This function receives smart card message until time out. This function waits until smart card message received within the limit of TimeOut.

```
[SCard_Manager.h]

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

Parameters:

pMsgBuffer

[out] Storage space of smart card message to output.

TimeOut

[in] Wait time of receiving smart card message [0 - *portMAX_DELAY* milliseconds]. Specifying the block time as *portMAX_DELAY* will cause the task to block indefinitely (without a timeout).

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully waiting for smart card message.

3.3.2. SCard_Manager_Select Card

This function selects active working card slot.

```
[SCard_Manager.h]

void SCard_Manager_SelectCard (
        UINT8 ucCard );
```

Parameters:

ucCard

[in] Index of active working card slot [0-4].

3.3.3. SCard_Manager_CardOn

This function powers on active working card slot.

```
[SCard_Manager.h]

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

Parameters:

bAutoVoltage

[in] TRUE -- automatic detect card working voltage, FALSE -- use fixed card working voltage.

ucVoltage

[in] If `bAutoVoltage` is FALSE, `ucVoltage` set a fixed working voltage of card,

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

pucReceiveBuffer

[out] Storage space of ATR data to output.

pusReceiveSize

[out] Size of output ATR data [byte].

Returns:

UINT8

This function returns the state of operation result,

[*SLOT_NO_ERROR* -- successful,

SLOTERROR_BAD_LENGTH -- data length error,

SLOTERROR_BAD_SLOT -- invalid working card slot,

SLOTERROR_ICC_MUTE -- card response time out,

SLOTERROR_XFR_PARITY_ERROR -- data parity error,

SLOTERROR_XFR_OVERRUN -- data transfer overrun,

SLOTERROR_HW_ERROR -- hardware error,

SLOTERROR_BAD_ATR_TS -- TS of ATR is error].

3.3.4. SCard_Manager_CardOff

This function powers off active working card slot.

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

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully powers off active working card slot.

3.3.5. SCard_Manager_SendBlock

This function sends APDU to smart card and receives data from card.

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

Parameters:

pucCmdBlockBuffer

[in] Storage space of input APDU send to card.

pucResBlockBuffer

[out] Storage space of output data from card.

pusBufferSize

[in&out] Storage space of input APDU size and output data size [byte].

Returns:

UINT8

This function returns the state of operation result,

[`SLOT_NO_ERROR` -- successful,

`SLOTERROR_BAD_LENGTH` -- data length error,

`SLOTERROR_ICC_MUTE` -- card response time out,

`SLOTERROR_XFR_PARITY_ERROR` -- data parity error,

`SLOTERROR_HW_ERROR` -- hardware error,

`SLOTERROR_ICC_CLASS_NOT_SUPPORTED` -- functional error,

`SLOTERROR_PROCEDURE_BYTE_CONFLICT` -- procedure byte error].

4.0. Reader API Functions

4.1. Battery API Functions

4.1.1. Data Structures

Here is no special data structure.

4.1.2. Functions

4.1.2.1. Battery_GetMilliVolt

This function gets the battery voltage.

```
[Battery.h]

UINT32 Battery_GetMilliVolt (
        void );
```

Returns:

UINT32

This function returns value of battery voltage [mV].

4.1.2.2. Battery_GetPercent

This function gets the percentage of battery energy volume

```
[Battery.h]

UINT8 Battery_GetPercent (
        void );
```

Returns:

UINT8

This function returns value of the percentage of battery energy volume [0-100].

4.1.2.3. Battery_WaitChargeStateChangeMsg

This function waits for the battery charge status changes. This function waits until charge status changed within the limit of *TimeOut*.

```
[Battery.h]

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

Parameters:

TimeOut

[in] Wait time of charge status changes [0 - *portMAX_DELAY* milliseconds]. Specifying the block time as *portMAX_DELAY* will cause the task to block indefinitely (without a timeout).

pbChargeState

[out] Storage space of charge status to output. If this function returns TRUE, **pbChargeState* outputs new status; if this function returns FALSE, **pbChargeState* outputs current status.

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully waiting for battery charge status change message.

4.2. Buzzer API Functions

4.2.1. Data Structures

4.2.1.1. Buzzer_ScriptDataType

```
[Buzzer_Msg.h]

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

Used by `Buzzer_Msg_SendScript`.

Example:

```
const Buzzer_ScriptDataType Buzzer_SampleScript1 [] =
{
    {1, TRUE},   // buzzer on 100ms
    {2, FALSE},  // buzzer off 200ms
    {3, TRUE},   // buzzer on 300ms
    ...
    {0, FALSE}   // mandatory script end
};
```

Data Member	Value	Description
BuzzerTime	Bit field 0 to 127	The time of a buzzer on/off state: This value represents a multiple of 100 ms.
BuzzerOn	Bit field TRUE or FALSE	The on/off state of buzzer: TRUR – buzzer on FALSE – buzzer off

4.2.2. Functions

4.2.2.1. Buzzer_Msg_SendScript

This function sends buzzer script to buzzer driver.

```
[Buzzer_Msg.h]

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

Parameters:

Script

[in] Storage space of the script data. . If this parameter is a local variable, it should be the data type of “static const” for compiling safety. So use global const variable for this parameter is more preferred.

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully sending buzzer script.

4.2.2.2. Buzzer_Msg_IsPlaying

This function gets the buzzer status that whether is it playing scripts.

```
[Buzzer_Msg.h]

BOOLEAN Buzzer_Msg_IsPlaying (
       void );
```

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state that the buzzer is playing scripts.

4.3. Keypad API Functions

4.3.1. Data Structures

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;
```

Used by `Key_MessageDataType` to transmit the status of key action.

Data Member	Value	Description
Key_release	0	Key released after long press
Key_PressDown	1	Key pressed down after no key pressed
Key_shortPress	2	Key released shorter than long press threshold
Key_longPress	3	Key press down longer than long press threshold every second

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;
```

Used by *Key_MessageDataType* and *Key_Msg_GetKeyPressing* to transmit the name of key.

Data Member	Value	Description
Key_noKeyInput	0	No key
Key_ClearKeyInput	1	Clear key
Key_Num0KeyInput	2	Numeric 0 key
Key_RightKeyInput	4	Direction right key
Key_Num7KeyInput	6	Numeric 7 key
Key_Num8KeyInput	7	Numeric 8 key
Key_Num9KeyInput	8	Numeric 9 key
Key_LeftKeyInput	9	Direction left key
Key_Num4KeyInput	11	Numeric 4 key
Key_Num5KeyInput	12	Numeric 5 key
Key_Num6KeyInput	13	Numeric 6 key
Key_DownKeyInput	14	Direction down key
Key_Num1KeyInput	16	Numeric 1 key
Key_Num2KeyInput	17	Numeric 2 key
Key_Num3KeyInput	18	Numeric 3 key
Key_UpKeyInput	19	Direction up key
Key_F1KeyInput	21	Function F1 key
Key_F2KeyInput	22	Function F2 key
Key_F3KeyInput	23	Function F3 key
Key_F4KeyInput	24	Function F4 key
Key_PowerKeyInput	26	Enter key / Power switch

4.3.1.3. Key_MessageDataType

```
[Key_Msg.h]

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

Used by `Key_Msg_ReceiveKey`.

Data Member	Value	Description
eKeyStatus	KeyStatusEnumType	Status of key action
eInputKey	KeyInputEnumType	The name of key

4.3.2. Functions

4.3.2.1. Key_Port_IsAnyKeyDown

This function gets the status of whether any key is pressed down.

```
[Key_Port.h]

BOOLEAN Key_Port_IsAnyKeyDown (
        void );
```

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state that any key is pressing down.

4.3.2.2. Key_Msg_ReceiveKey

This function waits for the keypad message until it times out. This function waits until the key message is received within the limit of *TimeOut*.

```
[Key_Msg.h]

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

Parameters:

Key_MsgRecBuffer

[out] Storage space of keypad message to output.

TimeOut

[in] Wait time of receiving keypad message.

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully waiting for keypad message.

4.3.2.3. Key_Msg_GetKeyPressing

This function gets the name of the key that is being pressed down.

```
[Key_Msg.h]

KeyInputEnumType Key_Msg_GetKeyPressing (
        void );
```

Returns:

KeyInputEnumType

This function returns the name of the key that is being pressed down.

4.3.2.4. Key_Ctrl_FlushMsgBuffer

This function clears the keypad message buffer. After this function has been called, the function *Key_Msg_ReceiveKey* can only wait for the newly-generated keypad message.

```
[Key_Ctrl.h]

void Key_Ctrl_FlushMsgBuffer (
        void );
```

4.3.2.5. Key_Ctrl_ScanLock

This function disables the keypad generates a new message. By default, the keypad generates new messages.

```
[Key_Ctrl.h]

void Key_Ctrl_ScanLock (
        void );
```

4.3.2.6. Key_Ctrl_ScanUnlock

This function enables the keypad generates a new message. By default, the keypad generates new messages.

```
[Key_Ctrl.h]

void Key_Ctrl_ScanUnlock (
        void );
```

4.3.2.7. Key_Ctrl_SetLongPressThreshold

This function sets the threshold of duration that has been used to distinguish long or short press of a key. By default, the threshold of duration is two seconds.

```
[Key_Ctrl.h]

void Key_Ctrl_SetLongPressThreshold (
        UINT8 Seconds );
```

Parameters:

Seconds

[in] Value of threshold duration [second].

4.3.2.8. Key_Tim_GetKeyDownTime

This function gets the latest duration of key press down. When no key is pressed, the value of the duration is retained.

```
[Key_Tim.h]

UINT16 Key_Tim_GetKeyDownTime (
       void );
```

Returns:

UINT16

This functions returns latest duration of key press down [second].

4.4. EEPROM API Functions

4.4.1. Data Structures

Here is no special data structure.

4.4.2. Functions

4.4.2.1. EEPROM_Write

This function writes data into EEPROM. The memory size of EEPROM is 65536 bytes, so the sum of *usAddress* and *usSize* must be less than or equal to 65536.

```
[EEPROM.h]

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

Parameters:

usAddress

[in] Start address of destination memory of EEPROM to be written [0-65535].

pucData

[in] Storage space of the data to be written into EEPROM.

usSize

[in] Size of input data [byte, 1-65535].

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully writing data into EEPROM.

4.4.2.2. EEPROM_Read

This function reads data from EEPROM. The memory size of EEPROM is 65536 bytes, so the sum of *usAddress* and *usSize* must be less than or equal to 65536.

```
[EEPROM.h]

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

Parameters:

usAddress

[in] Start address of source memory of EEPROM to be read [0-65535].

pucData

[out] Storage space of the data to be read from EEPROM.

usSize

[in] Size of output data [byte, 1-65535].

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully reading data from EEPROM.

4.5. Real-Time Clock API Functions

4.5.1. Data Structures

No special data structure.

4.5.2. Functions

4.5.2.1. EXRTC_Write_Ram

This function writes data to RAM of external RTC. The memory size of external RTC RAM is 238 bytes, so the sum of *usAddress* and *usSize* must be less than or equal to 238.

```
[EXRTC.h]

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

Parameters:

usAddress

[in] Start address of destination memory of external RTC RAM to be written [0-237].

pucData

[in] Storage space of the data to be written into external RTC RAM.

usSize

[in] Size of input data [byte, 1-238].

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully writing data into external RTC RAM.

4.5.2.2. EXRTC_Read_Ram

This function reads data from RAM of external RTC. The memory size of external RTC RAM is 238 bytes, so the sum of *usAddress* and *usSize* must be less than or equal to 238.

```
[EXRTC.h]

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

Parameters:

usAddress

[in] Start address of source memory of external RTC RAM to be read [0-237].

pucData

[out] Storage space of the data to be read from external RTC RAM.

usSize

[in] Size of output data [byte, 1-238].

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully reading data from external RTC RAM.

4.5.2.3. EXRTC_Write_Time

This function sets decimal time value to external RTC.

```
[EXRTC.h]

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

Parameters:

ucHour

[in] Decimal value of hour [0-23].

ucMinute

[in] Decimal value of minute [0-59].

ucSecond

[in] Decimal value of second [0-59].

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully setting time value to external RTC.

4.5.2.4. EXRTC_Write_TimeBCD

This function sets binary-coded decimal time value to external RTC.

```
[EXRTC.h]

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

Parameters:

ucHour

[in] BCD value of hour [0x00-0x23].

ucMinute

[in] BCD value of minute [0x00-0x59].

ucSecond

[in] BCD value of second [0x00-0x59].

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully setting time value to external RTC.

4.5.2.5. EXRTC_Read_Time

This function reads out time value from external RTC, the format of output data is BCD.

```
[EXRTC.h]

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

Parameters:

pucHour

[out] Storage space of output hour value.

pucMinute

[out] Storage space of output minute value.

pucSecond

[out] Storage space of output second value.

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully reading out time value from external RTC.

4.5.2.6. EXRTC_Write_Date

This function sets decimal date value to external RTC.

```
[EXRTC.h]

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

Parameters:

ucYear

[in] Decimal value of year [0-99].

ucMonth

[in] Decimal value of month [1-12].

ucWeekday

[in] Decimal value of weekday [0-6].

ucDay

[in] Decimal value of day [1-31].

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully setting date value to external RTC.

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

4.5.2.7. EXRTC_Write_DateBCD

This function sets binary-coded decimal date value to external RTC.

```
[EXRTC.h]

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

Parameters:

ucYear

[in] BCD value of year [0x00-0x99].

ucMonth

[in] BCD value of month [0x01-0x12].

ucWeekday

[in] BCD value of weekday [0x00-0x06].

ucDay

[in] BCD value of day [0x01-0x31].

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully setting date value to external RTC.

4.5.2.8. EXRTC_Read_Date

This function reads out date value from external RTC, the format of output data is BCD.

```
[EXRTC.h]

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

Parameters:

pucYear

[out] Storage space of output year value.

pucMonth

[out] Storage space of output month value.

pucWeekday

[out] Storage space of output weekday value.

pucDay

[out] Storage space of output day value.

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully reading out date value from external RTC.

4.6. LCD API Functions

4.6.1. Data Structures

No special data structure.

4.6.2. Functions

4.6.2.1. LCD_SetCursor

This function sets the cursor position of LCD.

```
[LCD.h]

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

Parameters:

ucLcdRowPosition

[in] LCD cursor row position [0 - LCD_MAX_ROW].

ucLcdColumnPosition

[in] LCD cursor column position [0 - LCD_MAX_COLUMN].

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully (not out of boundary) setting the cursor position of LCD.

4.6.2.2. LCD_GetCursor

This function gets the current cursor position of LCD.

```
[LCD.h]

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

Parameters:

pucLcdRowPosition

[out] Storage space of LCD row position.

pucLcdColumnPosition

[out] Storage space of LCD column position.

4.6.2.3. LCD_Display_ASCIIChar

This function displays single ASCII character in LCD.

```
[LCD.h]

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

Parameters:

ucLcdCharacterToDisplay

[in] ASCII code to display in LCD [0x20-0x7E].

bSetNextPosCur

[in] TRUE -- set cursor to next position after displayed the code, FALSE -- the cursor remains in the same position.

4.6.2.4. LCD_DisplayASCIIMessage

This function displays a string of characters. The three control characters '\b', '\r' and '\n' can be used in string of characters.

```
[LCD.h]

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

Parameters:

LcdMessageToDisplay

[in] Null terminated string of characters to be displayed.

4.6.2.5. LCD_ClearDisplay

This function clears LCD display by mode of 'whole page', 'whole row' or 'one character'.

```
[LCD.h]

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

Parameters:

index

[in] 0 -- Clear whole page, 1 -- Clear whole rows from current row of cursor, the number of rows to be cleared is *ucNumber*, 2 -- Clear column part of characters(1 character is 6 columns) from current cursor, the number of columns of characters to be cleared is *ucNumber*.

ucNumber

[in] Number of rows or columns to be cleared.

4.6.2.6. LCD_SetContrast

This function sets contrast level of LCD.

```
[LCD.h]

void LCD_SetContrast (
       UINT8 contrast_level );
```

Parameters:

contrast_level

[in] Level of contrast [0-255], the default value is 169.

4.6.2.7. LCD_SetBacklight

This function turns on/off the backlight of LCD.

```
[LCD.h]

void LCD_SetBacklight (
       BOOLEAN bTurnOn );
```

Parameters:

bTurnOn

[in] TRUE -- Turn on backlight, FALSE -- Turn off backlight.

4.6.2.8. LCD_Display_Cursor

This function displays vertical cursor in LCD (8 pixels).

```
[LCD.h]

void LCD_Display_Cursor (
       void );
```

4.6.2.9. LCD_Clear_Cursor

This function clears vertical cursor in LCD.

```
[LCD.h]

void LCD_Clear_Cursor (
       void );
```

4.6.2.10. LCD_Display_Page

This function displays whole image in one screen.

```
[LCD.h]

void LCD_Display_Page (
       UINT8 *pucBitmap );
```

Parameters:

pucBitmap

[in] A string of bitmap raw data to be displayed [resolution: (*LCD_MAX_ROW* x 8) x *LCD_MAX_COLUMN*].

4.6.2.11. LCD_DisplayGraphic

This function displays icon-like bitmap at particular position.

```
[LCD.h]

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

Parameters:

ucLcdRowNumber

[in] Number of row of image occupied.

ucLcdColumnNumber

[in] Number of column of image occupied.

pucBitMap

[in] Array of image data.

4.6.2.12. LCD_DisplayOn

This function turns on/off of LCD display.

```
[LCD.h]

void LCD_DisplayOn (
       BOOLEAN bTurnOn );
```

Parameters:

bTurnOn

[in] TRUE -- Turn on display, FALSE -- Turn off display.

4.6.2.13. LCD_DisplayDecimal

This function displays decimal number in LCD.

```
[LCD.h]

void LCD_DisplayDecimal (
       UINT32 ulDecimal );
```

Parameters:

ulDecimal

[in] Decimal number to be displayed.

4.6.2.14. LCD_DisplayHex

This function displays hex format number in LCD.

```
[LCD.h]

void LCD_DisplayHex (
       UINT8 ucDisplay );
```

Parameters:

ucDisplay

[in] Number to be displayed.

4.6.2.15. LCD_DisplayHexN

This function displays hexadecimal number string in LCD.

```
[LCD.h]

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

Parameters:

pucDisplay

[in] Array of hexadecimal number to be displayed.

Number

[in] Size of hexadecimal number array [bytes].

4.6.2.16. LCD_DisplayFloat

This function displays float format number in LCD.

```
[LCD.h]

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

Parameters:

ulDecimal

[in] Number to be displayed (without radix point).

Exp

[in] Decimal digits after radix point[0-9].

4.6.2.17. LCD_DrawTitleBox

This function displays title box in LCD.

```
[LCD.h]

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

Parameters:

TitleMessage

[in] Null terminated string of characters [only supported `0x20`-`0x7E`] to be displayed.

4.7. Serial Flash API Functions

The difference between serial flash and EEPROM is you must make sure that the memory in serial flash to be written is erased (all of the data in the memory is `0xFF`).

4.7.1. Data Structures

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
```

Used by SerialFlash_Erase_Block to set type of size of block memory to erase.

Data Member	Value	Description
ERASE_4K	SF_ERASE_4K	4K bytes block type
ERASE_64K	SF_ERASE_64K	64K bytes block type

4.7.2. Functions

4.7.2.1. SerialFlash_ReadDataBytes

This function reads data from serial flash.

```
[SFlash.h]

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

Parameters:

ulAddress

[in] Start address of source memory of serial flash to be read [`0x20000-0x7FFFF`].

pucReceiveBufferPtr

[out] Storage space of the data to be read from serial flash.

ulLength

[in] Size of output data [byte].

4.7.2.2. SerialFlash_Erase_Block

This function erases block memory of serial flash.

```
[SFlash.h]

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

Parameters:

BlockType

[in] Type of size of block memory to erase, [*ERASE_4K* -- 4KB block, *ERASE_64K* -- 64KB block].

ulAddress

[in] Start address of destination block memory of serial flash to be erased [>= `0x20000`, must be block size aligned].

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully erasing block memory of serial flash.

4.7.2.3. SerialFlash_WriteDataBytes

This function writes data into serial flash. Before writing data into serial flash, the destination memory should be erased (make sure the data in the memory is all `0xFF`).

```
[SFlash.h]

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

Parameters:

ulAddress

[in] Start address of destination memory of serial flash to be written [`0x20000`-`0x7FFFF`].

pucWriteBufferPtr

[in] Storage space of the data to be written into serial flash.

ulLength

[in] Size of input data [byte].

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully writing data into serial flash.

4.8. RS232 API Functions

4.8.1. Data Structures

4.8.1.1. ParityEnumType

```
[RS232.h]

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

Used by RS232_ParamDataType to transmit parity configuration of RS232.

Data Member	Value	Description
No_Parity	0	RS232 no parity mode
Odd_Parity	1	RS232 odd parity mode
Even_Parity	2	RS232 even parity mode

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;
```

Used by RS232_Config to transmit parameters of RS232 configuration.

Data Member	Value	Description
Baudrate	UINT32	This value is baud rate of RS232 (e.g. 9600 is baud rate of 960 0bps, range from 400 to 115200)
ParityMode	ParityEnumType	Parity mode of RS232 configuration
SevenOrEightDataBit	BOOLEAN	7 or 8 bits data mode: TRUE – 7-bit data mode FALSE – 8-bit data mode
TwoOrOneStopBit	BOOLEAN	2 or 1 stop bit mode: TRUE – 2 stop bit mode FALSE – 1 stop bit mode

4.8.2. Functions

4.8.2.1. RS232_Config

This function sets the parameters of RS232 port. Before using this function to set parameter, the RS232 port should be in the state of closed, if not *RS232_Config* will return false.

```
[RS232.h]

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

Parameters:

Param

[in] Storage space of the parameter data.

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully set parameter of RS232.

4.8.2.2. RS232_OpenPort

This function opens RS232 port. Before using this function, the RS232 port should be in the state of closed, if not RS232_OpenPort will return false.

```
[RS232.h]

BOOLEAN RS232_OpenPort (
    UINT32* pulHandle );
```

Parameters:

pulHandle

[out] Storage space of the handle for control RS232 port.

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully open RS232 port.

4.8.2.3. RS232_ClosePort

This function closes RS232 port.

```
[RS232.h]

BOOLEAN RS232_ClosePort (
    UINT32 ulHandle );
```

Parameters:

ulHandle

[in] The handle value gets from *RS232_OpenPort*. If the value is not from *RS232_OpenPort*, this function will return false.

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully close RS232 port.

4.8.2.4. RS232_ReceivedDataNumber

This function gets the number of bytes received by RS232 port.

```
[RS232.h]

UINT16 RS232_ReceivedDataNumber (
    UINT32 ulHandle );
```

Parameters:

ulHandle

[in] The handle value gets from *RS232_OpenPort*. If the value is not from *RS232_OpenPort*, this function will return 0.

Returns:

UINT16

This function returns number of bytes received by RS232 port.

4.8.2.5. RS232_Receive

This function gets data received by RS232 port. This function waits until data received within the limit of *TimeOut*.

[RS232.h]

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

Parameters:

ulHandle

[in] The handle value gets from *RS232_OpenPort*. If the value is not from *RS232_OpenPort*, this function will return false.

pucRecBuf

[out] Storage space of the data to be gotten from RS232 port.

pusLen

[in&out] Storage space of receive buffer size and output received data size [byte].

TimeOut

[in] Wait time of receiving data from RS232 port.

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully receiving data from RS232 port.

4.8.2.6. RS232_Send

This function sends data to RS232 port.

```
[RS232.h]

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

Parameters:

ulHandle

[in] The handle value gets from *RS232_OpenPort*. If the value is not from *RS232_OpenPort*, this function will return false.

pucSndBuf

[in] Storage space of the data to be sent to RS232 port.

usLen

[in] Size of data to be sent to RS232 port [byte].

Returns:

BOOLEAN

This function returns TRUE/FALSE of the state of successfully sending data to RS232 port.

4.9. Miscellaneous I/O API Functions

4.9.1. Data Structures

1.1.1.1. 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;
```

Used by *IO_ReadDigitalOutput* and *IO_WriteDigitalOutput* to set the address of I/O port to operate.

Data Member	Value	Description
IO_LED0Output	14	Red color of first bi-color LED
IO_LED1Output	15	Green color of first bi-color LED
IO_LED2Output	16	Red color of second bi-color LED
IO_LED3Output	17	Green color of second bi-color LED
IO_LED4Output	18	Red color of third bi-color LED
IO_LED5Output	19	Green color of third bi-color LED
IO_LED6Output	20	Red color of fourth bi-color LED
IO_LED7Output	21	Green color of fourth bi-color LED

4.9.1.1. IO_ActiveInactiveStateType

```
[IO.h]

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

Used by *IO_ReadDigitalOutput* and *IO_WriteDigitalOutput* to transmit I/O port logical state.

Data Member	Value	Description
IO_InactiveState	0	Logical FALSE
IO_ActiveState	1	Logical TRUE
IO_UndefineState	2	Logical invalid

4.9.2. Functions

4.9.2.1. IO_ReadDigitalOutput

This function gets value of digital output port correspond to port name address.

```
[IO.h]

IO_ActiveInactiveStateType IO_ReadDigitalOutput (
       IO_VirtualNameType Address );
```

Parameters:

Address

[in] Port name address of digital output port to be read.

Returns:

IO_ActiveInactiveStateType

This function returns value of digital output port [*IO_InactiveState* -- the port output value is inactive, *IO_ActiveState* – the port output value is active].

4.9.2.2. IO_WriteDigitalOutput

This function sets value of digital output port correspond to port name address.

```
[IO.h]

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

Parameters:

State

[in] Value of digital output port to be set [*IO_InactiveState* -- output value is inactive, *IO_ActiveState* – output value is active].

Address

[in] Port name address of digital output port to be written.

4.9.2.3. IO_WriteLEDOutput

This function sets output value to all LEDs.

```
[IO.h]

void IO_WriteLEDOutput (
        UINT8 ucLED );
```

Parameters:

ucLED

[in] The value of output on/off of all bi-color LEDs

[bit 0~7 correspond to *IO_LED0Output* ~ *IO_LED7Output* of `IO_ActiveInactiveStateType`;

IO_LED0Output is red color of first LED;

IO_LED1Output is green color of first LED;

IO_LED2Output is red color of second LED;

IO_LED3Output is green color of second LED;

IO_LED4Output is red color of third LED;

IO_LED5Output is green color of third LED;

IO_LED6Output is red color of fourth LED;

IO_LED7Output is green color of fourth LED]. The value 1 of a bit outputs active state; the value 0 of a bit outputs inactive state.

4.9.2.4. IO_ReadLEDOutput

This function gets output value to all LEDs.

```
[IO.h]

UINT8 IO_ReadLEDOutput (
       void );
```

Returns:

UINT8

This function returns output value to all bi-color LEDs

[bit 0~7 correspond to *IO_LED0Output* ~ *IO_LED7Output* of *IO_ActiveInactiveStateType*;

IO_LED0Output is red color of first LED; *IO_LED1Output* is green color of first LED; *IO_LED2Output* is red color of second LED;

IO_LED3Output is green color of second LED;

IO_LED4Output is red color of third LED;

IO_LED5Output is green color of third LED;

IO_LED6Output is red color of fourth LED;

IO_LED7Output is green color of fourth LED]. The value 1 of a bit indicates active state; the value 0 of a bit indicates inactive state.

4.9.2.5. IO_SystemShutdown

This function switches the total system off.

```
[IO.h]

void IO_SystemShutdown (
    void );
```

4.9.2.6. IO_SystemSleep

This function puts system to sleep, and pressing any key releases system from sleep.

```
[IO.h]

void IO_SystemSleep (
    void );
```

4.9.2.7. IO_SystemRestart

This function restarts the system.

```
[IO.h]

void IO_SystemRestart (
    void );
```

4.9.2.8. IO_GetSystemVersion

This function returns version information.

```
[IO.h]

Const UCHAR*IO_GetSystemVersion (
    void );
```

Returns:

*const UCHAR**

This function returns pointer of version information which is ASCII string with the end of null character '\0'.

5.0. RF Card API Functions (only for ACR89-CL version)

5.1. Data Structures

No special data structures.

5.2. Functions

For ACR89-CL version, also use *SCard_Manager_SelectCard* to select RF card, index 0 for RF card slot, index 1-5 for smart card slots (while non-ACR89-CL version is 0-4). The APDU data function of RF card slot is compatible with ACR122. *SCard_Manager_CardOn* is also used to get ATR. *SCard_Manager_CardOff* is a dummy function for RF card slot. *SCard_Manager_SendBlock* is also used to send APDU and get response from RF card slot. *SCard_Manager_Msg_Receive* is also used to receive RF card insert/remove message until time out (message RF card slot index is 0, while smart card slot index is 1-2).

5.2.1. RFIF_Sleep

This function sets RF interface power down to reduce power consumption. The default state of RF interface is power on.

```
[RFCard.h]

void RFIF_Sleep (
    void );
```

5.2.2. RFIF_Wakeup

This function sets RF interface power up to run contactless card function. The default state of RF interface is power on.

```
[RFCard.h]

void RFIF_WakeUp (
    void );
```

6.0. FreeRTOS API Functions

FreeRTOS API functions come from http://www.freertos.org/, for more information please visit the website. The porting of FreeRTOS and ISR handling is contained in ACR89U-A1 SDK, so there is no need to use ISR related API functions. **For the duration of system tick, the macro *portTICK_RATE_MS* in portmacro.h provides the value; e.g. if portTICK_RATE_MS equals to 1, that represents the unit of type portTickType is 1 millisecond.** Also in ACR89 SDK, FreeRTOS is configured to use preemption, software timers and not to use co-routines.

6.1. Task Creation

6.1.1. xTaskHandle

This is a data type by which tasks are referenced. For example, a call to *xTaskCreate* returns (via a pointer parameter) an *xTaskHandle* variable that can then be used as a parameter to *vTaskDelete* to delete the task.

```
[task.h]
```

6.1.2. xTaskCreate

This function creates a new task and adds it to the list of tasks that are ready to run.

```
[task.h]

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

Parameters:

pvTaskCode

[in] Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).

pcName

[in] A descriptive name for the task. This is mainly used to facilitate debugging. Maximum length is defined by *configMAX_TASK_NAME_LEN*.

usStackDepth

[in] The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and `usStackDepth` is defined as 100, 200 bytes will be allocated for stack storage. The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type size_t.

pvParameters

[in] Pointer that will be used as the parameter for the task being created.

uxPriority

[in] The priority at which the task should run.

pvCreatedTask

[out] Used to pass back a handle by which the created task can be referenced.

Returns:

portBASE_TYPE

`pdPASS` if the task was successfully created and added to a ready list, otherwise an error code defined in the file *projdefs.h*.

Example usage:

```
// Task to be created.
void vTaskCode( void * pvParameters )
{
    for( ;; )
    {
        // Task code goes here.
    }
}

// Function that creates a task.
void vOtherFunction( void )
{
     static unsigned char ucParameterToPass;
     xTaskHandle xHandle;

  // Create the task, storing the handle.  Note that the passed parameter
     ucParameterToPass
  // must exist for the lifetime of the task, so in this case is declared
     static.  If it was just an
  // automatic stack variable it might no longer exist, or at least have
     been corrupted, by the time
  // the new task attempts to access it.
    xTaskCreate( vTaskCode,
                 "NAME",
                 STACK_SIZE,
                 &ucParameterToPass,
                 tskIDLE_PRIORITY,
                 &xHandle );
```

```
    // Use the handle to delete the task.
    vTaskDelete( xHandle );
}
```

6.1.3. vTaskDelete

This function removes a task from the RTOS real time kernels management. The task being deleted will be removed from all ready, blocked, suspended and event lists.

```
[task.h]

void vTaskDelete( xTaskHandle pxTask );
```

Parameters:

pxTask

[in] The handle of the task to be deleted. Passing NULL will cause the calling task to be deleted.

Example usage:

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

    // Create the task, storing the handle.
    xTaskCreate( vTaskCode,
                 "NAME",
                 STACK_SIZE,
                 NULL,
                 tskIDLE_PRIORITY,
                 &xHandle );

    // Use the handle to delete the task.
    vTaskDelete( xHandle );
}
```

6.2. Task Control

6.2.1. vTaskDelay

This function delays a task for a given number of ticks.

```
[task.h]

void vTaskDelay( portTickType xTicksToDelay );
```

Parameters:

xTicksToDelay

[in] The amount of time, in tick periods, that the calling task should block.

Example usage:

```
  void vTaskFunction( void * pvParameters )
 {
      /* Block for 500ms. */
      const portTickType xDelay = 500 / portTICK_RATE_MS;

     for( ;; )
     {
     /* Simply toggle the LED every 500ms, blocking between each toggle. */
         vToggleLED();
         vTaskDelay( xDelay );
     }
}
```

6.2.2. vTaskDelayUntil

This function delays a task until a specified time. This function can be used by cyclical tasks to ensure a constant execution frequency.

```
[task.h]
void vTaskDelayUntil( portTickType *pxPreviousWakeTime, portTickType
xTimeIncrement );
```

Parameters:

pxPreviousWakeTime

[in] Pointer to a variable that holds the time at which the task was last unblocked. The variable must be initialized with the current time prior to its first use (see the example below). Following this the variable is automatically updated within *vTaskDelayUntil()*.

xTimeIncrement

[in] The cycle time period. The task will be unblocked at time (**pxPreviousWakeTime + xTimeIncrement*). Calling *vTaskDelayUntil* with the same *xTimeIncrement* parameter value will cause the task to execute with a fixed interval period.

Example usage:

```
// Perform an action every 10 ticks.
void vTaskFunction( void * pvParameters )
{
     portTickType xLastWakeTime;
     const portTickType xFrequency = 10;

    // Initialise the xLastWakeTime variable with the current time.
    xLastWakeTime = xTaskGetTickCount();

    for( ;; )
    {
        // Wait for the next cycle.
        vTaskDelayUntil( &xLastWakeTime, xFrequency );

        // Perform action here.
    }
}
```

6.2.3. uxTaskPriorityGet

This function obtains the priority of any task.

```
[task.h]

unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );
```

Parameters:

pxTask

[in] Handle of the task to be queried. Passing a NULL handle results in the priority of the calling task being returned.

Returns:

unsigned portBASE_TYPE

The priority of *pxTask*.

Example usage:

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

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode,
                 "NAME",
                 STACK_SIZE,
                 NULL,
                 tskIDLE_PRIORITY,
                 &xHandle );

    // ...

    // Use the handle to obtain the priority of the created task.
    // It was created with tskIDLE_PRIORITY, but may have changed
    // it itself.
    if( uxTaskPriorityGet( xHandle ) != tskIDLE_PRIORITY )
    {
        // The task has changed its priority.
    }
```

```
    // ...

    // Is our priority higher than the created task?
    if( uxTaskPriorityGet( xHandle ) < uxTaskPriorityGet( NULL ) )
    {
        // Our priority (obtained using NULL handle) is higher.
    }
}
```

6.2.4. vTaskPrioritySet

This function sets the priority of any task. A context switch will occur before the function returns if the priority being set is higher than the currently executing task.

```
[task.h]
```

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

Parameters:

pxTask

[in] Handle to the task for which the priority is being set. Passing a *NULL* handle results in the priority of the calling task being set.

uxNewPriority

[in] The priority to which the task will be set.

Example usage:

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

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode,
                 "NAME",
                 STACK_SIZE,
                 NULL,
                 tskIDLE_PRIORITY,
                 &xHandle );

    // ...

    // Use the handle to raise the priority of the created task.
    vTaskPrioritySet( xHandle, tskIDLE_PRIORITY + 1 );

    // ...

    // Use a NULL handle to raise our priority to the same value.
    vTaskPrioritySet( NULL, tskIDLE_PRIORITY + 1 );
}
```

6.2.5. vTaskSuspend

This function suspends any task. When suspended a task will never get any microcontroller processing time, no matter what its priority. Calls to *vTaskSuspend* are not accumulative - i.e. calling *vTaskSuspend ( )* twice on the same task still only requires one call to *vTaskResume* () to ready the suspended task.

```
[task.h]

void vTaskSuspend( xTaskHandle pxTaskToSuspend );
```

Parameters:

pxTaskToSuspend

[in] Handle to the task being suspended. Passing a NULL handle will cause the calling task to be suspended.

Example usage:

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

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode,
                 "NAME",
                 STACK_SIZE,
                 NULL,
                 tskIDLE_PRIORITY,
                 &xHandle );

    // ...

    // Use the handle to suspend the created task.
    vTaskSuspend( xHandle );

    // ...

    // The created task will not run during this period, unless
    // another task calls vTaskResume( xHandle ).

    //...

    // Suspend ourselves.
```

```
    vTaskSuspend( NULL );

    // We cannot get here unless another task calls vTaskResume
    // with our handle as the parameter.
}
```

6.2.6. vTaskResume

This function resumes a suspended task. A task that has been suspended by one of more calls to `vTaskSuspend ()` will be made available for running again by a single call to `vTaskResume ()`.

```
[task.h]

void vTaskResume( xTaskHandle pxTaskToResume );
```

Parameters:

pxTaskToResume

[in] Handle to the task being readied.

Example usage:

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

    // Create a task, storing the handle.
    xTaskCreate( vTaskCode,
                 "NAME",
                 STACK_SIZE,
                 NULL,
                 tskIDLE_PRIORITY,
                 &xHandle );

    // ...

    // Use the handle to suspend the created task.
    vTaskSuspend( xHandle );

    // ...

    // The created task will not run during this period, unless
    // another task calls vTaskResume( xHandle ).
```

```
    //...

    // Resume the suspended task ourselves.
    vTaskResume( xHandle );

    // The created task will once again get microcontroller processing
    // time in accordance with it priority within the system.
}
```

6.3. Task Utilities

6.3.1. xTaskGetCurrentTaskHandle

This function gets the handle of current task.

```
[task.h]

xTaskHandle xTaskGetCurrentTaskHandle( void );
```

Returns:

xTaskHandle

The handle of the currently running (calling) task.

6.3.2. xTaskGetTickCount

This function gets the value of the count of ticks.

```
[task.h]

volatile portTickType xTaskGetTickCount( void );
```

Returns:

portTickType

The count of ticks since *vTaskStartScheduler* was called.

6.3.3. xTaskGetSchedulerState

This function gets the state of scheduler.

```
[task.h]

portBASE_TYPE xTaskGetSchedulerState( void );
```

Returns:

***portBASE**_TYPE*

One of the following constants (defined within task.h): *taskSCHEDULER_NOT_STARTED*, *taskSCHEDULER_RUNNING*, *taskSCHEDULER_SUSPENDED*.

6.3.4. uxTaskGetNumberOfTasks

This function gets the number of tasks.

```
[task.h]

unsigned portBASE_TYPE uxTaskGetNumberOfTasks( void );
```

Returns:

unsigned portBASE_TYPE

The number of tasks that the real time kernel is currently managing. This includes all ready, blocked and suspended tasks. A task that has been deleted but not freed by the idle task will also be included in the count.

6.4. Kernel Control

6.4.1. taskYIELD

This is a macro for forcing a context switch.

```
[task.h]

taskYIELD();
```

6.4.2. taskENTER_CRITICAL

This is a macro to mark the start of a critical code region. Preemptive context switches cannot occur when in a critical region.

```
[task.h]

taskENTER_CRITICAL();
```

6.4.3. taskEXIT_CRITICAL

This is a macro to mark the end of a critical code region. Preemptive context switches cannot occur when in a critical region.

```
[task.h]

taskEXIT_CRITICAL();
```

6.4.4. vTaskSuspendAll

This function suspends all real time kernel activity while keeping interrupts (including the kernel tick) enabled. After calling *vTaskSuspendAll ()* the calling task will continue to execute without the risk of being swapped out until a call to *xTaskResumeAll ( )* has been made. API functions that have the potential to cause a context switch (for example, *vTaskDelayUntill ( ), xQueueSend ( ),* etc.) must **not** be called while the scheduler is suspended.

```
[task.h]

void vTaskSuspendAll( void );
```

Example usage:

```
void vTask1( void * pvParameters )
{
    for( ;; )
    {
        // Task code goes here.

        // ...

        // At some point the task wants to perform a long operation during
        // which it does not want to get swapped out.  It cannot use
        // taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
        // operation may cause interrupts to be missed - including the
        // ticks.

        // Prevent the real time kernel swapping out the task.
        vTaskSuspendAll ();

        // Perform the operation here.  There is no need to use critical
        // sections as we have all the microcontroller processing time.
        // During this time interrupts will still operate and the kernel
        // tick count will be maintained.

        // ...

        // The operation is complete.  Restart the kernel.
        xTaskResumeAll ();
    }
}
```

6.4.5. xTaskResumeAll

This function resumes real time kernel activity following a call to *vTaskSuspendAll ( )*. After a call to *xTaskSuspendAll ( )* the kernel will take control of which task is executing at any time.

```
[task.h]

portBASE_TYPE xTaskResumeAll( void );
```

Returns:

portBASE_TYPE

If resuming the scheduler caused a context switch then `pdTRUE` is returned, otherwise `pdFALSE` is returned.

Example usage:

```
void vTask1( void * pvParameters )
{
    for( ;; )
    {
        // Task code goes here.

        // ...

        // At some point the task wants to perform a long operation during
        // which it does not want to get swapped out.  It cannot use
        // taskENTER_CRITICAL ()/taskEXIT_CRITICAL () as the length of the
        // operation may cause interrupts to be missed - including the
        // ticks.

        // Prevent the real time kernel swapping out the task.
        xTaskSuspendAll ();

        // Perform the operation here.  There is no need to use critical
        // sections as we have all the microcontroller processing time.
        // During this time interrupts will still operate and the real
        // time kernel tick count will be maintained.

        // ...

        //  The operation is complete.  Restart the kernel.  We want to
        force
        // a context switch - but there is no point if resuming the
```

```
            scheduler
         // caused a context switch already.
         if( !xTaskResumeAll () )
         {
              taskYIELD ();
         }
     }
}
```

6.5. Queue Management

6.5.1. uxQueueMessagesWaiting

This function returns the number of messages stored in a queue.

[queue.h]

```
unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle xQueue );
```

Parameters:

xQueue

[in] A handle to the queue being queried.

Returns:

unsigned portBASE_TYPE

The number of messages available in the queue.

6.5.2. xQueueCreate

This function creates a new queue instance. This allocates the storage required by the new queue and returns a handle for the queue.

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

Parameters:

uxQueueLength

[in] The maximum number of items that the queue can contain.

uxItemSize

[in] The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.

Returns:

xQueueHandle

If the queue is successfully created then a handle to the newly created queue is returned. If the queue cannot be created then 0 is returned.

Example usage:

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

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

   // Create a queue capable of containing 10 unsigned long values.
   xQueue1 = xQueueCreate( 10, sizeof( unsigned portLONG ) );
   if( xQueue1 == 0 )
   {
       // Queue was not created and must not be used.
   }

   // Create a queue capable of containing 10 pointers to AMessage
     structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );
   if( xQueue2 == 0 )
   {
       // Queue was not created and must not be used.
   }

   // ... Rest of task code. }
```

6.5.3. vQueueDelete

This function deletes a queue - freeing all the memory allocated for storing of items placed on the queue.

```
[queue.h]

void vQueueDelete( xQueueHandle xQueue );
```

Parameters:

xQueue

[in] A handle to the queue to be deleted.

6.5.4. xQueueSend

This is a macro that calls *xQueueGenericSend*(). It is equivalent to *xQueueSendToBack*(). Post an item on a queue. The item is queued by copy, not by reference.

```
[queue.h]

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

Parameters:

xQueue

[in] The handle to the queue on which the item is to be posted.

pvItemToQueue

[in] A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from *pvItemToQueue* into the queue storage area.

xTicksToWait

[in] The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if the queue is full and *xTicksToWait* is set to 0. The time is defined in tick periods so the constant *portTICK_RATE_MS* should be used to convert to real time if this is required. Specifying the block time as *portMAX_DELAY* will cause the task to block indefinitely (without a timeout).

Returns:

portBASE_TYPE

pdTRUE if the item was successfully posted, otherwise *errQUEUE_FULL*.

Example usage:

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

unsigned portLONG ulVar = 10UL;

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

   // Create a queue capable of containing 10 unsigned long values.
   xQueue1 = xQueueCreate( 10, sizeof( unsigned portLONG ) );

   // Create a queue capable of containing 10 pointers to AMessage
structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

   // ...

   if( xQueue1 != 0 )
   {
       // Send an unsigned long.  Wait for 10 ticks for space to become
       // available if necessary.
       if( xQueueSend( xQueue1, ( void * ) &ulVar,
           ( portTickType ) 10 ) != pdPASS )
       {
           // Failed to post the message, even after 10 ticks.
       }
   }

   if( xQueue2 != 0 )
   {
       // Send a pointer to a struct AMessage object.  Don't block if the
       // queue is already full.
       pxMessage = & xMessage;
```

```
        xQueueSend( xQueue2, ( void * ) &pxMessage, ( portTickType ) 0 );
    }

    // ... Rest of task code.
}
```

6.5.5. xQueueSendToBack

This is a macro that calls *xQueueGenericSend().* It is equivalent to *xQueueSend()*. Post an item to the back of a queue. The item is queued by copy, not by reference.

```
[queue.h]

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

Parameters:

xQueue

[in] The handle to the queue on which the item is to be posted.

pvItemToQueue

[in] A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from *pvItemToQueue* into the queue storage area.

xTicksToWait

[in] The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0. The time is defined in tick periods so the constant *portTICK_RATE_MS* should be used to convert to real time if this is required. Specifying the block time as *portMAX_DELAY* will cause the task to block indefinitely (without a timeout).

Returns:

portBASE_TYPE

pdTRUE if the item was successfully posted, otherwise *errQUEUE_FULL*.

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

Example usage:

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

unsigned portLONG ulVar = 10UL;

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

   // Create a queue capable of containing 10 unsigned long values.
   xQueue1 = xQueueCreate( 10, sizeof( unsigned portLONG ) );

   // Create a queue capable of containing 10 pointers to AMessage
      structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

   // ...

   if( xQueue1 != 0 )
   {
       // Send an unsigned long.  Wait for 10 ticks for space to become
       // available if necessary.
       if( xQueueSendToBack( xQueue1, ( void * ) &ulVar, ( portTickType )
10 ) != pdPASS )
       {
           // Failed to post the message, even after 10 ticks.
       }
   }
```

```
    if( xQueue2 != 0 )
    {
        // Send a pointer to a struct AMessage object.  Don't block if the
        // queue is already full.
        pxMessage = & xMessage;
        xQueueSendToBack( xQueue2, ( void * ) &pxMessage,
                                             ( portTickType ) 0 );
    }

      // ... Rest of task code.
}
```

6.5.6. xQueueSendToToFront

This is a macro that calls *xQueueGenericSend()*. Post an item to the front of a queue. The item is queued by copy, not by reference.

```
[queue.h]

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

Parameters:

xQueue

[in] The handle to the queue on which the item is to be posted.

pvItemToQueue

[in] A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from *pvItemToQueue* into the queue storage area.

xTicksToWait

[in] The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0. The time is defined in tick periods so the constant *portTICK_RATE_MS* should be used to convert to real time if this is required. Specifying the block time as *portMAX_DELAY* will cause the task to block indefinitely (without a timeout).

Returns:

***portBASE**_TYPE*

pdTRUE if the item was successfully posted, otherwise *errQUEUE_FULL*.

Example usage:

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

unsigned portLONG ulVar = 10UL;

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

   // Create a queue capable of containing 10 unsigned long values.
   xQueue1 = xQueueCreate( 10, sizeof( unsigned portLONG ) );

   // Create a queue capable of containing 10 pointers to AMessage
      structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue2 = xQueueCreate( 10, sizeof( struct AMessage * ) );

   // ...

   if( xQueue1 != 0 )
   {
       // Send an unsigned long.  Wait for 10 ticks for space to become
       // available if necessary.
       if( xQueueSendToFront( xQueue1, ( void * ) &ulVar,
           ( portTickType ) 10 ) != pdPASS )
       {
           // Failed to post the message, even after 10 ticks.
       }
   }
```

```
if( xQueue2 != 0 )
    {
        // Send a pointer to a struct AMessage object.  Don't block if the
        // queue is already full.
        pxMessage = & xMessage;
        xQueueSendToFront( xQueue2, ( void * ) &pxMessage,
                                      ( portTickType ) 0 );
    }

      // ... Rest of task code.
 }
```

6.5.7. xQueueReceive

This is a macro that calls the *xQueueGenericReceive()* function. Receive an item from a queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

```
[queue.h]

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

Parameters:

pxQueue

[in] The handle to the queue from which the item is to be received.

pvBuffer

[out] Pointer to the buffer into which the received item will be copied.

xTicksToWait

[in] The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. Setting *xTicksToWait* to 0 will cause the function to return immediately if the queue is empty. The time is defined in tick periods so the constant *portTICK_RATE_MS* should be used to convert to real time if this is required. Specifying the block time as *portMAX_DELAY* will cause the task to block indefinitely (without a timeout).

Returns:

portBASE_TYPE

pdTRUE if an item was successfully received from the queue, otherwise *pdFALSE*.

Example usage:

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

xQueueHandle xQueue;

// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
   struct AMessage *pxMessage;

   // Create a queue capable of containing 10 pointers to AMessage
      structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
   if( xQueue == 0 )
   {
       // Failed to create the queue.
   }

   // ...

   // Send a pointer to a struct AMessage object.  Don't block if the
   // queue is already full.
   pxMessage = & xMessage;
   xQueueSend( xQueue, ( void * ) &pxMessage, ( portTickType ) 0 );

     // ... Rest of task code.
}

// Task to receive from the queue.
void vADifferentTask( void *pvParameters )
{
   struct AMessage *pxRxedMessage;

   if( xQueue != 0 )
   {
```

```
        // Receive a message on the created queue.  Block for 10 ticks if a
        // message is not immediately available.
        if( xQueueReceive( xQueue, &( pxRxedMessage ),
            ( portTickType ) 10 ) )
        {
            // pcRxedMessage now points to the struct AMessage variable
               posted
            // by vATask.
        }
    }

    // ... Rest of task code.
}
```

6.5.8. xQueuePeek

This is a macro that calls the *xQueueGenericReceive()* function. Receive an item from a queue without removing the item from the queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created. Successfully received items remain on the queue so will be returned again by the next call, or a call to *xQueueReceive()*.

```
[queue.h]

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

Parameters:

xQueue

[in] The handle to the queue from which the item is to be received.

pvBuffer

[out] Pointer to the buffer into which the received item will be copied. This must be at least large enough to hold the size of the queue item defined when the queue was created.

xTicksToWait

[in] The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. The time is defined in tick periods so the constant *portTICK_RATE_MS* should be used to convert to real time if this is required. Specifying the block time as *portMAX_DELAY* will cause the task to block indefinitely (without a timeout).

Returns:

portBASE_TYPE

pdTRUE if an item was successfully received (peeked) from the queue, otherwise *pdFALSE*.

Example usage:

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

xQueueHandle xQueue;

// Task to create a queue and post a value.
void vATask( void *pvParameters )
{
   struct AMessage *pxMessage;

   // Create a queue capable of containing 10 pointers to AMessage
      structures.
   // These should be passed by pointer as they contain a lot of data.
   xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
   if( xQueue == 0 )
   {
       // Failed to create the queue.
   }

   // ...

   // Send a pointer to a struct AMessage object.  Don't block if the
   // queue is already full.
   pxMessage = & xMessage;
   xQueueSend( xQueue, ( void * ) &pxMessage, ( portTickType ) 0 );

   // ... Rest of task code.
}

// Task to peek the data from the queue.
void vADifferentTask( void *pvParameters )
```

```
{
    struct AMessage *pxRxedMessage;

    if( xQueue != 0 )
    {
        // Peek a message on the created queue.  Block for 10 ticks if a
        // message is not immediately available.
        if( xQueuePeek( xQueue, &( pxRxedMessage ), ( portTickType ) 10 ) )
        {
            // pcRxedMessage now points to the struct AMessage variable
               posted
            // by vATask, but the item still remains on the queue.
        }
    }

    // ... Rest of task code.
}
```

6.6. Semaphore / Mutexes

6.6.1. vSemaphoreCreateBinary

This is a macro that creates a **semaphore** by using the existing queue mechanism. The queue length is 1 as this is a binary semaphore. The data size is 0 as we don't want to actually store any data - we just want to know if the queue is empty or full. Binary semaphores and mutexes are very similar but have some subtle differences: ***Mutexes*** include a priority inheritance mechanism; binary semaphores do not. This makes binary semaphores the better choice for implementing synchronization (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion. A binary semaphore need not be given back once obtained, so task synchronization can be implemented by one task/interrupt continuously 'giving' the semaphore while another continuously 'takes' the semaphore. The priority of a task that 'takes' a mutex can potentially be raised if another task of higher priority attempts to obtain the same mutex. The task that owns the mutex 'inherits' the priority of the task attempting to 'take' the same mutex. This means the mutex must always be 'given' back - otherwise the higher priority task will never be able to obtain the mutex, and the lower priority task will never 'disinherit' the priority. An example of a mutex being used to implement mutual exclusion is provided on the *xSemaphoreTake()* documentation page. Both mutex and binary semaphores are assigned to variables of type *xSemaphoreHandle* and can be used in any API function that takes a parameter of this type.

```
[semphr.h]

vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore );
```

Parameters:

xSemaphore

[out] Handle to the created semaphore. Should be of type `xSemaphoreHandle`.

Example usage:

```
xSemaphoreHandle xSemaphore;

void vATask( void * pvParameters )
{
    // Semaphore cannot be used before a call to vSemaphoreCreateBinary ().
    // This is a macro so pass the variable in directly.
    vSemaphoreCreateBinary( xSemaphore );

    if( xSemaphore != NULL )
    {
        // The semaphore was created successfully.
        // The semaphore can now be used.
    }
}
```

6.6.2. xSemaphoreCreateCounting

This is a macro that creates a counting semaphore by using the existing queue mechanism.

Counting semaphores are typically used for two things:

1. **Counting events**.

In this usage scenario an event handler will 'give' a semaphore each time an event occurs (incrementing the semaphore count value), and a handler task will 'take' a semaphore each time it processes an event (decrementing the semaphore count value). The count value is therefore the difference between the number of events that have occurred and the number that have been processed. In this case it is desirable for the initial count value to be zero.

2. **Resource management**.

In this usage scenario the count value indicates the number of resources available. To obtain control of a resource a task must first obtain a semaphore - decrementing the semaphore count value. When the count value reaches zero there are no free resources. When a task finishes with the resource it 'gives' the semaphore back - incrementing the semaphore count value. In this case it is desirable for the initial count value to be equal to the maximum count value, indicating that all resources are free.

```
[semphr.h]

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

Parameters:

uxMaxCount

[in] The maximum count value that can be reached. When the semaphore reaches this value it can no longer be 'given'.

uxInitialCount

[in] The count value assigned to the semaphore when it is created.

Returns:

xSemaphoreHandle

Handle to the created semaphore. NULL if the semaphore could not be created.

Example usage:

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

    //    Semaphore    cannot    be    used    before    a    call    to
xSemaphoreCreateCounting().
    // The max value to which the semaphore can count shall be 10, and the
    // initial value assigned to the count shall be 0.
    xSemaphore = xSemaphoreCreateCounting( 10, 0 );

    if( xSemaphore != NULL )
    {
        // The semaphore was created successfully.
        // The semaphore can now be used.
    }
}
```

6.6.3. xSemaphoreCreateMutex

This is a macro that creates a mutex semaphore by using the existing queue mechanism. Mutexes created using this macro can be accessed using the *xSemaphoreTake()* and *xSemaphoreGive()* macros. The *xSemaphoreTakeRecursive()* and *xSemaphoreGiveRecursive()* macros should not be used.

Mutexes and binary semaphores are very similar but have some subtle differences: Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronization (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion. The priority of a task that 'takes' a mutex can potentially be raised if another task of higher priority attempts to obtain the same mutex. The task that owns the mutex 'inherits' the priority of the task attempting to 'take' the same mutex. This means the mutex must always be 'given' back - otherwise the higher priority task will never be able to obtain the mutex, and the lower priority task will never 'disinherit' the priority. An example of a mutex being used to implement mutual exclusion is provided on the *xSemaphoreTake()* documentation page. A binary semaphore need not be given back once obtained, so task synchronization can be implemented by one task/interrupt continuously 'giving' the semaphore while another continuously 'takes' the semaphore. Both mutex and binary semaphores are assigned to variables of type *xSemaphoreHandle* and can be used in any API function that takes a parameter of this type.

```
[semphr.h]
```

```
xSemaphoreHandle xSemaphoreCreateMutex( void );
```

Returns:

xSemaphoreHandle

Handle to the created semaphore. Should be of type `xSemaphoreHandle`.

Example usage:

```
xSemaphoreHandle xSemaphore;

void vATask( void * pvParameters )
{
   // Mutex semaphores cannot be used before a call to
   // xSemaphoreCreateMutex().  The created mutex is returned.
   xSemaphore = xSemaphoreCreateMutex();

   if( xSemaphore != NULL )
   {
       // The semaphore was created successfully.
       // The semaphore can now be used.
   }
}
```


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

6.6.4. xSemaphoreCreateRecursiveMutex

This is a macro that implements a recursive mutex by using the existing queue mechanism. Mutexes created using this macro can be accessed using the *xSemaphoreTakeRecursive()* and *xSemaphoreGiveRecursive()* macros. The *xSemaphoreTake()* and *xSemaphoreGive()* macros should not be used. A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called *xSemaphoreGiveRecursive()* for each successful 'take' request. For example, if a task successfully 'takes' the same mutex five (5) times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times. This type of semaphore uses a priority inheritance mechanism so a task 'taking' a semaphore MUST ALWAYS 'give' the semaphore back once the semaphore it is no longer required.

```
[semphr.h]
```

```
xSemaphoreHandle xSemaphoreCreateRecursiveMutex( void );
```

Returns:

xSemaphoreHandle

`xSemaphoreHandle` to the created mutex semaphore. Should be of type `xSemaphoreHandle`.

Example usage:

```
xSemaphoreHandle xMutex;

void vATask( void * pvParameters )
{
   // Semaphore cannot be used before a call to xSemaphoreCreateMutex().
   // This is a macro so pass the variable in directly.
   xMutex = xSemaphoreCreateRecursiveMutex();

   if( xMutex != NULL )
   {
       // The mutex type semaphore was created successfully.
       // The mutex can now be used.
   }
}
```

6.6.5. xSemaphoreTake

This is a macro to obtain a semaphore. The semaphore must have previously been created with a call to *vSemaphoreCreateBinary()*, *xSemaphoreCreateMutex()* or *xSemaphoreCreateCounting()*.

```
[semphr.h]

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

Parameters:

xSemaphore

[in] A handle to the semaphore being taken - obtained when the semaphore was created.

xBlockTime

[in] The time in ticks to wait for the semaphore to become available. The macro *portTICK_RATE_MS* can be used to convert this to a real time. A block time of zero can be used to poll the semaphore. Specifying the block time as *portMAX_DELAY* will cause the task to block indefinitely (without a timeout).

Returns:

signed portBASE_TYPE

pdTRUE if the semaphore was obtained. *pdFALSE* if *xBlockTime* expired without the semaphore becoming available.

Example usage:

```
xSemaphoreHandle xSemaphore = NULL;
// A task that creates a semaphore.
void vATask( void * pvParameters )
{
   // Create the semaphore to guard a shared resource.  As we are using
   // the semaphore for mutual exclusion we create a mutex semaphore
   // rather than a binary semaphore.
   xSemaphore = xSemaphoreCreateMutex();
}

// A task that uses the semaphore.
void vAnotherTask( void * pvParameters )
{
   // ... Do other things.

   if( xSemaphore != NULL )
   {
       // See if we can obtain the semaphore.  If the semaphore is not
          available
       // wait 10 ticks to see if it becomes free.
       if( xSemaphoreTake( xSemaphore, ( portTickType ) 10 ) == pdTRUE )
       {
           // We were able to obtain the semaphore and can now access the
           // shared resource.

           // ...

           // We have finished accessing the shared resource.  Release the
           // semaphore.
           xSemaphoreGive( xSemaphore );
       }
       else
       {
           // We could not obtain the semaphore and can therefore not
              access
           // the shared resource safely.
       }
   }
}
```

6.6.6. xSemaphoreTakeRecursive

This is a macro to recursively obtain, or 'take', a mutex type semaphore. The mutex must have previously been created using a call to *xSemaphoreCreateRecursiveMutex()*. This macro must not be used on mutexes created using *xSemaphoreCreateMutex()*. A mutex used recursively can be 'taken' repeatedly by the owner. The mutex does not become available again until the owner has called *xSemaphoreGiveRecursive()* for each successful 'take' request. For example, if a task successfully 'takes' the same mutex five (5) times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

[semphr.h]

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

Parameters:

xMutex

[in] A handle to the mutex being obtained. This is the handle returned by *xSemaphoreCreateRecursiveMutex()*.

xBlockTime

[in] The time in ticks to wait for the semaphore to become available. The macro *portTICK_RATE_MS* can be used to convert this to a real time. A block time of zero can be used to poll the semaphore. If the task already owns the semaphore then *xSemaphoreTakeRecursive()* will return immediately no matter what the value of *xBlockTime*.

Returns:

portBASE_TYPE

pdTRUE if the semaphore was obtained. *pdFALSE* if *xBlockTime* expired without the semaphore becoming available.

Example usage:

```
xSemaphoreHandle xMutex = NULL;
// A task that creates a mutex.
void vATask( void * pvParameters )
{
    // Create the mutex to guard a shared resource.
    xMutex = xSemaphoreCreateRecursiveMutex();
}

// A task that uses the mutex.
void vAnotherTask( void * pvParameters )
{
    // ... Do other things.

    if( xMutex != NULL )
    {
        // See if we can obtain the mutex.  If the mutex is not available
        // wait 10 ticks to see if it becomes free.
        if( xSemaphoreTakeRecursive( xMutex, ( portTickType ) 10 ) ==
pdTRUE )
        {
            // We were able to obtain the mutex and can now access the
            // shared resource.

            // ...
            // For some reason due to the nature of the code further calls
to
            // xSemaphoreTakeRecursive() are made on the same mutex.  In
real
            // code these would not be just sequential calls as this would
make
            // no sense.  Instead the calls are likely to be buried inside
            // a more complex call structure.
            xSemaphoreTakeRecursive( xMutex, ( portTickType ) 10 );
            xSemaphoreTakeRecursive( xMutex, ( portTickType ) 10 );

            // The mutex has now been 'taken' three times, so will not be
            // available to another task until it has also been given back
            // three times.  Again it is unlikely that real code would have
            // these calls sequentially, but instead buried in a more
complex
```

```
            // call structure.  This is just for illustrative purposes.
            xSemaphoreGiveRecursive( xMutex );
            xSemaphoreGiveRecursive( xMutex );
            xSemaphoreGiveRecursive( xMutex );

            // Now the mutex can be taken by other tasks.
        }
        else
        {
            // We could not obtain the mutex and can therefore not access
            // the shared resource safely.
        }
    }
}
```

6.6.7. xSemaphoreGive

This is a macro to release a semaphore. The semaphore must have previously been created with a call to *vSemaphoreCreateBinary()*, *xSemaphoreCreateMutex()* or *xSemaphoreCreateCounting()*, and obtained using *sSemaphoreTake()*. This macro must also not be used on semaphores created using *xSemaphoreCreateRecursiveMutex()*.

```
[semphr.h]

signed portBASE_TYPE xSemaphoreGive( xSemaphoreHandle xSemaphore );
```

Parameters:

xSemaphore

[in] A handle to the semaphore being released. This is the handle returned when the semaphore was created.

Returns:

signed portBASE_TYPE

pdTRUE if the semaphore was released. *pdFALSE* if an error occurred. Semaphores are implemented using queues. An error can occur if there is no space on the queue to post a message - indicating that the semaphore was not first obtained correctly.

Example usage:

```
xSemaphoreHandle xSemaphore = NULL;

void vATask( void * pvParameters )
{
   // Create the semaphore to guard a shared resource.  As we are using
   // the semaphore for mutual exclusion we create a mutex semaphore
   // rather than a binary semaphore.
   xSemaphore = xSemaphoreCreateMutex();

   if( xSemaphore != NULL )
   {
       if( xSemaphoreGive( xSemaphore ) != pdTRUE )
       {
           // We would expect this call to fail because we cannot give
           // a semaphore without first "taking" it!
       }
       // Obtain the semaphore - don't block if the semaphore is not
       // immediately available.
       if( xSemaphoreTake( xSemaphore, ( portTickType ) 0 ) )
```

```
        {
            // We now have the semaphore and can access the shared
resource.
            // ...
            // We have finished accessing the shared resource so can free
               the
            // semaphore.
            if( xSemaphoreGive( xSemaphore ) != pdTRUE )
            {
                // We would not expect this call to fail because we must
                   have
                // obtained the semaphore to get here.
            }
        }
    }
 }
```

6.6.8. xSemaphoreGiveRecursive

This is a macro to recursively release, or 'give', a mutex type semaphore. The mutex must have previously been created using a call to *xSemaphoreCreateRecursiveMutex().* This macro must not be used on mutexes created using *xSemaphoreCreateMutex().* A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has called *xSemaphoreGiveRecursive()* for each successful 'take' request. For example, if a task successfully 'takes' the same mutex five (5) times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

```
[semphr.h]

portBASE_TYPE xSemaphoreGiveRecursive( xSemaphoreHandle xMutex );
```

Parameters:

xMutex

[in] A handle to the mutex being released, or 'given'. This is the handle returned by *xSemaphoreCreateRecursiveMutex()*.

Returns:

portBASE_TYPE

`pdTRUE` if the semaphore was successfully given.

Example usage:

```
xSemaphoreHandle xMutex = NULL;

// A task that creates a mutex.
void vATask( void * pvParameters )
{
   // Create the mutex to guard a shared resource.
   xMutex = xSemaphoreCreateRecursiveMutex();
}

// A task that uses the mutex.
void vAnotherTask( void * pvParameters )
{
   // ... Do other things.

   if( xMutex != NULL )
   {
       // See if we can obtain the mutex.  If the mutex is not available
       // wait 10 ticks to see if it becomes free.
       if( xSemaphoreTakeRecursive( xMutex, ( portTickType ) 10 ) ==
```

```
pdTRUE )
        {
            // We were able to obtain the mutex and can now access the
            // shared resource.

            // ...
            // For some reason due to the nature of the code further calls
               to
            // xSemaphoreTakeRecursive() are made on the same mutex.  In
                real
            // code these would not be just sequential calls as this would
                make
            // no sense.  Instead the calls are likely to be buried inside
            // a more complex call structure.
            xSemaphoreTakeRecursive( xMutex, ( portTickType ) 10 );
            xSemaphoreTakeRecursive( xMutex, ( portTickType ) 10 );

            // The mutex has now been 'taken' three times, so will not be
            // available to another task until it has also been given back
            // three times.  Again it is unlikely that real code would have
            // these calls sequentially, it would be more likely that the
               calls
            // to xSemaphoreGiveRecursive() would be called as a call stack
            // unwound. This is just for demonstrative purposes.
            xSemaphoreGiveRecursive( xMutex );
            xSemaphoreGiveRecursive( xMutex );
            xSemaphoreGiveRecursive( xMutex );

            // Now the mutex can be taken by other tasks.
        }
        else
        {
            // We could not obtain the mutex and can therefore not access
            // the shared resource safely.
        }
    }
}
```

6.7. Software Timers

Timer functionality is provided by a timer service/daemon task. Many of the public FreeRTOS timer API functions send commands to the timer service task though a queue called the timer command queue. The timer command queue is private to the kernel itself and is not directly accessible to application code. The length of the timer command queue is set by the *configTIMER_QUEUE_LENGTH* configuration constant. The timer service/daemon task priority is set by the *configTIMER_TASK_PRIORITY* configuration constant.

6.7.1. xTimerCreate

This function creates a new software timer instance. This allocates the storage required by the new timer, initializes the new timer's internal state, and returns a handle by which the new timer can be referenced. Timers are created in the dormant state. The *xTimerStart()*, *xTimerReset()* and *xTimerChangePeriod()* API functions can all be used to transition a timer into the active state.

```
[timers.h]

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

Parameters:

pcTimerName

[in] A text name that is assigned to the timer. This is done purely to assist debugging. The kernel itself only ever references a timer by its handle, and never by its name.

xTimerPeriod

[in] The timer period. The time is defined in tick periods so the constant *portTICK_RATE_MS* can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then *xTimerPeriod* should be set to 100. Alternatively, if the timer must expire after 500ms, then *xPeriod* can be set to (500/*portTICK_RATE_MS*) provided *configTICK_RATE_HZ* is less than or equal to 1000.

uxAutoReload

[in] If *uxAutoReload* is set to *pdTRUE*, then the timer will expire repeatedly with a frequency set by the *xTimerPeriod* parameter. If *uxAutoReload* is set to `pdFALSE`, then the timer will be a one-shot and enter the dormant state after it expires.

pvTimerID

[in] An identifier that is assigned to the timer being created. Typically this would be used in the timer callback function to identify which timer expired when the same callback function is assigned to more than one timer.

pxCallbackFunction

[in] The function to call when the timer expires. Callback functions must have the prototype defined by *tmrTIMER_CALLBACK*, which is "*void vCallbackFunction(xTimerHandle xTimer);*".

Returns:

xTimerHandle

If the timer is successfully create then a handle to the newly created timer is returned. If the timer cannot be created (because either there is insufficient FreeRTOS heap remaining to allocate the timer structures, or the timer period was set to 0) then 0 is returned.

Example usage:

```
#define NUM_TIMERS 5

 /* An array to hold handles to the created timers. */
 xTimerHandle xTimers[ NUM_TIMERS ];

 /* An array to hold a count of the number of times each timer expires. */
 long lExpireCounters[ NUM_TIMERS ] = { 0 };

 /* Define a callback function that will be used by multiple timer
    instances.
 The callback function does nothing but count the number of times the
 associated timer expires, and stop the timer once the timer has expired
 10 times. */
 void vTimerCallback( xTimerHandle pxTimer )
 {
 long lArrayIndex;
 const long xMaxExpiryCountBeforeStopping = 10;

     /* Optionally do something if the pxTimer parameter is NULL. */
     configASSERT( pxTimer );

     /* Which timer expired? */
     lArrayIndex = ( long ) pvTimerGetTimerID( pxTimer );

     /* Increment the number of times that pxTimer has expired. */
     lExpireCounters[ lArrayIndex ] += 1;

     /* If the timer has expired 10 times then stop it from running. */
     if( lExpireCounters[ lArrayIndex ] == xMaxExpiryCountBeforeStopping )
     {
         /* Do not use a block time if calling a timer API function from a
         timer callback function, as doing so could cause a deadlock! */
         xTimerStop( pxTimer, 0 );
     }
```

```
}

void main( void )
{
long x;

    /* Create then start some timers.  Starting the timers before the
       scheduler
    has been started means the timers will start running immediately that
    the scheduler starts. */
    for( x = 0; x < NUM_TIMERS; x++ )
    {
        xTimers[ x ] = xTimerCreate(
                      "Timer", /* Just a text name, not used by the kernel.*/
                      ( 100 * x ), /* The timer period in ticks.*/
                      pdTRUE, /* The timers will auto-reload themselves when
                                 they expire. */
                      ( void * ) x, /* Assign each timer a unique id equal to
                                       its array index. */
                      vTimerCallback /* Each timer calls the same callback
                                        when it expires. */
                          );

        if( xTimers[ x ] == NULL )
        {
            /* The timer was not created. */
        }
        else
        {
            /* Start the timer.  No block time is specified, and even if
               one was, it would be ignored because the scheduler has not
               yet been started. */
            if( xTimerStart( xTimers[ x ], 0 ) != pdPASS )
            {
                /* The timer could not be set into the Active state. */
            }
        }
    }

    /* ...
    Create tasks here.
    ... */
```

```
    /* Starting the scheduler will start the timers running as they have
       already been set into the active state. */
    xTaskStartScheduler();


    /* Should not reach here. */
    for( ;; );
}
```

6.7.2. xTimerIsTimerActive

This function Queries a timer to see if it is active or dormant.

A timer will be dormant if:

1. It has been created but not started, or
2. It is an expired on-shot timer that has not been restarted.

```
[timers.h]

portBASE_TYPE xTimerIsTimerActive( xTimerHandle xTimer );
```

Parameters:

xTimer

[in] The timer being queried.

Returns:

portBASE_TYPE

`pdFALSE` will be returned if the timer is dormant. A value other than `pdFALSE` will be returned if the timer is active.

Example usage:

```
/* This function assumes xTimer has already been created. */
 void vAFunction( xTimerHandle xTimer )
 {
     if( xTimerIsTimerActive( xTimer ) != pdFALSE )
 /*or more simply and equivalently "if( xTimerIsTimerActive( xTimer ) )" */
     {
         /* xTimer is active, do something. */
     }
     else
     {
         /* xTimer is not active, do something else. */
     }
 }
```

6.7.3. xTimerStart

This function starts a timer that was previously created using the *xTimerCreate()* API function. If the timer had already been started and was already in the active state, then *xTimerStart()* has equivalent functionality to the *xTimerReset()* API function. Starting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the meantime, the callback function associated with the timer will get called 'n 'ticks after *xTimerStart()* was called, where 'n' is the timers defined period.

```
[timers.h]

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

Parameters:

xTimer

[in] The handle of the timer being started/restarted.

xBlockTime

[in] Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the start command to be successfully sent to the timer command queue, should the queue already be full when *xTimerStart()* was called.

Returns:

portBASE_TYPE

pdFAIL will be returned if the start command could not be sent to the timer command queue even after *xBlockTime* ticks had passed. *pdPASS* will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when *xTimerStart()* is actually called.

6.7.4. xTimerStop

This function stops a timer that was previously started using either of the *xTimerStart()*, *xTimerReset()* and *xTimerChangePeriod()* API functions.

```
[timers.h]

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

Parameters:

xTimer

[in] The handle of the timer being stopped.

xBlockTime

[in] Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the stop command to be successfully sent to the timer command queue, should the queue already be full when *xTimerStop()* was called.

Returns:

portBASE_TYPE

pdFAIL will be returned if the stop command could not be sent to the timer command queue even after *xBlockTime* ticks had passed. *pdPASS* will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system.

6.7.5. xTimerChangePeriod

This function changes the period of a timer that was previously created using the *xTimerCreate()* API function. *xTimerChangePeriod()* can be called to change the period of an active or dormant state timer.

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

Parameters:

xTimer

[in] The handle of the timer that is having its period changed.

xNewPeriod

[in] The new period for *xTimer*. Timer periods are specified in tick periods, so the constant *portTICK_RATE_MS* can be used to convert a time that has been specified in milliseconds. For example, if the timer must expire after 100 ticks, then *xNewPeriod* should be set to 100. Alternatively, if the timer must expire after 500ms, then *xNewPeriod* can be set to (500/*portTICK_RATE_MS*) provided *configTICK_RATE_HZ* is less than or equal to 1000.

xBlockTime

[in] Specifies the time, in ticks, that the calling task should be held in the *Blocked* state to wait for the change period command to be successfully sent to the timer command queue, should the queue already be full when *xTimerChangePeriod()* was called.

Returns:

portBASE_TYPE

pdFAIL will be returned if the change period command could not be sent to the timer command queue even after *xBlockTime* ticks had passed. *pdPASS* will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system.

Example usage:

```
/* This function assumes xTimer has already been created.  If the timer
referenced by xTimer is already active when it is called, then the timer
is deleted.  If the timer referenced by xTimer is not active when it is
called, then the period of the timer is set to 500ms and the timer is
started. */
void vAFunction( xTimerHandle xTimer )
{
    if( xTimerIsTimerActive( xTimer ) != pdFALSE )
/* or more simply and equivalently "if( xTimerIsTimerActive( xTimer ) )" */
    {
        /* xTimer is already active - delete it. */
        xTimerDelete( xTimer );
    }
    else
    {
       /* xTimer is not active, change its period to 500ms.  This will also
        cause the timer to start.  Block for a maximum of 100 ticks if the
        change period command cannot immediately be sent to the timer
        command queue. */
        if( xTimerChangePeriod( xTimer, 500 / portTICK_RATE_MS, 100 ) ==
pdPASS )
        {
            /* The command was successfully sent. */
        }
        else
        {
         /* The command could not be sent, even after waiting for 100 ticks
            to pass.  Take appropriate action here. */
        }
    }
}
```

6.7.6. xTimerDelete

This function deletes a timer that was previously created using the *xTimerCreate()* API function.

```
[timers.h]

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

Parameters:

xTimer

[in] The handle of the timer being deleted.

xBlockTime

[in] Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the delete command to be successfully sent to the timer command queue, should the queue already be full when *xTimerDelete()* was called.

Returns:

portBASE_TYPE

pdFAIL will be returned if the delete command could not be sent to the timer command queue even after *xBlockTime* ticks had passed. *pdPASS* will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system.

6.7.7. xTimerReset

This function re-starts a timer that was previously created using the *xTimerCreate()* API function. If the timer had already been started and was already in the active state, then *xTimerReset()* will cause the timer to re-evaluate its expiry time so that it is relative to when *xTimerReset()* was called. If the timer was in the dormant state then *xTimerReset()* has equivalent functionality to the *xTimerStart()* API function. Resetting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the meantime, the callback function associated with the timer will get called 'n' ticks after *xTimerReset()* was called, where 'n' is the timers defined period.

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

Parameters:

xTimer

[in] The handle of the timer being reset/started/restarted.

xBlockTime

[in] Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the reset command to be successfully sent to the timer command queue, should the queue already be full when xTimerReset() was called.

Returns:

portBASE_TYPE

pdFAIL will be returned if the reset command could not be sent to the timer command queue even after *xBlockTime* ticks had passed. *pdPASS* will be returned if the command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service/daemon task relative to other tasks in the system, although the timers expiry time is relative to when *xTimerReset()* is actually called.

Example usage:

```
/* When a key is pressed, an LCD back-light is switched on.  If 5 seconds
pass
without a key being pressed, then the LCD back-light is switched off.  In
this case, the timer is a one-shot timer. */

xTimerHandle xBacklightTimer = NULL;

/* The callback function assigned to the one-shot timer.  In this case the
parameter is not used. */
void vBacklightTimerCallback( xTimerHandle pxTimer )
{
    /* The timer expired, therefore 5 seconds must have passed since a key
    was pressed.  Switch off the LCD back-light. */
    vSetBacklightState( BACKLIGHT_OFF );
}

/* The key press event handler. */
void vKeyPressEventHandler( char cKey )
{
    /* Ensure the LCD back-light is on, then reset the timer that is
    responsible for turning the back-light off after 5 seconds of
    key inactivity.  Wait 10 ticks for the command to be successfully sent
    if it cannot be sent immediately. */
    vSetBacklightState( BACKLIGHT_ON );
    if( xTimerReset( xBacklightTimer, 10 ) != pdPASS )
    {
      /* The reset command was not executed successfully.  Take appropriate
         action here. */
    }

    /* Perform the rest of the key processing here. */
}
```

```
void main( void )
{
long x;

    /* Create then start the one-shot timer that is responsible for turning
    the back-light off if no keys are pressed within a 5 second period. */
    xBacklightTimer = xTimerCreate(
      "BacklightTimer",     /* Just a text name, not used by the kernel. */
      ( 5000 / portTICK_RATE_MS), /* The timer period in ticks. */
      pdFALSE,                    /* The timer is a one-shot timer. */
      0,                        /* The id is not used by the callback so can
                                   take any value. */
      vBacklightTimerCallback  /* The callback function that switches the
                                  LCD back-light off. */
                                 );

    if( xBacklightTimer == NULL )
    {
        /* The timer was not created. */
    }
    else
    {
       /* Start the timer.  No block time is specified, and even if one was
        it would be ignored because the scheduler has not yet been
        started. */
        if( xTimerStart( xBacklightTimer, 0 ) != pdPASS )
        {
            /* The timer could not be set into the Active state. */
        }
    }

    /* ...
     Create tasks here.
    ... */

   /* Starting the scheduler will start the timer running as it has already
    been set into the active state. */
    xTaskStartScheduler();
```

```
    /* Should not reach here. */
    for( ;; );
}
```

6.7.8. pvTimerGetTimerID

This function returns the ID assigned to the timer. IDs are assigned to timers using the *pvTimerID* parameter of the call to *xTimerCreate()* that was used to create the timer. If the same callback function is assigned to multiple timers then the timer ID can be used within the callback function to identify which timer actually expired.

```
[timers.h]

void *pvTimerGetTimerID( xTimerHandle xTimer );
```

Parameters:

xTimer

[in] The timer being queried.

Returns:

void *

The ID assigned to the timer being queried.