M25-QuecOpen

User Guide

GSM/GPRS Module Series

Rev. M25-QuecOpen_User_Guide_V1.0

Date: 2019-11-14

Status: Released

Our aim is to provide customers with timely and comprehensive service. For any assistance, please contact our company headquarters:

Quectel Wireless Solutions Co., Ltd.
Building 5, Shanghai Business Park Phase III (Area B), No.1016 Tianlin Road, Minhang District, Shanghai, China 200233
Tel: +86 21 5108 6236
Email: info@quectel.com

Or our local office. For more information, please visit:
http://www.quectel.com/support/sales.htm

For technical support, or to report documentation errors, please visit:
http://www.quectel.com/support/technical.htm
Or email to: support@quectel.com

GENERAL NOTES

QUECTEL OFFERS THE INFORMATION AS A SERVICE TO ITS CUSTOMERS. THE INFORMATION PROVIDED IS BASED UPON CUSTOMERS' REQUIREMENTS. QUECTEL MAKES EVERY EFFORT TO ENSURE THE QUALITY OF THE INFORMATION IT MAKES AVAILABLE. QUECTEL DOES NOT MAKE ANY WARRANTY AS TO THE INFORMATION CONTAINED HEREIN, AND DOES NOT ACCEPT ANY LIABILITY FOR ANY INJURY, LOSS OR DAMAGE OF ANY KIND INCURRED BY USE OF OR RELIANCE UPON THE INFORMATION. ALL INFORMATION SUPPLIED HEREIN IS SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.

COPYRIGHT

About the Document

History

Revision	Date	Author	Description
1.0	2019-11-14	Allan LIANG/ Edwin WEN	Initial

Contents

Table Index

Figure Index

1 Introduction

QuecOpen® is an embedded development solution for M2M applications where Quectel modules can be designed as the main processor. It has been designed to facilitate the design and accelerate the application development. QuecOpen® makes it possible to create innovative applications and embed them directly into Quectel modules to run without an external MCU. It has been widely used in M2M field, such as tracker & tracing, automotive, energy, wearable devices, etc.

This document mainly introduces how to use QuecOpen® solution on Quectel GSM/GPRS M25 module.

2 QuecOpen® Platform

2.1. System Architecture

The following figure shows the fundamental principle of QuecOpen® software architecture.

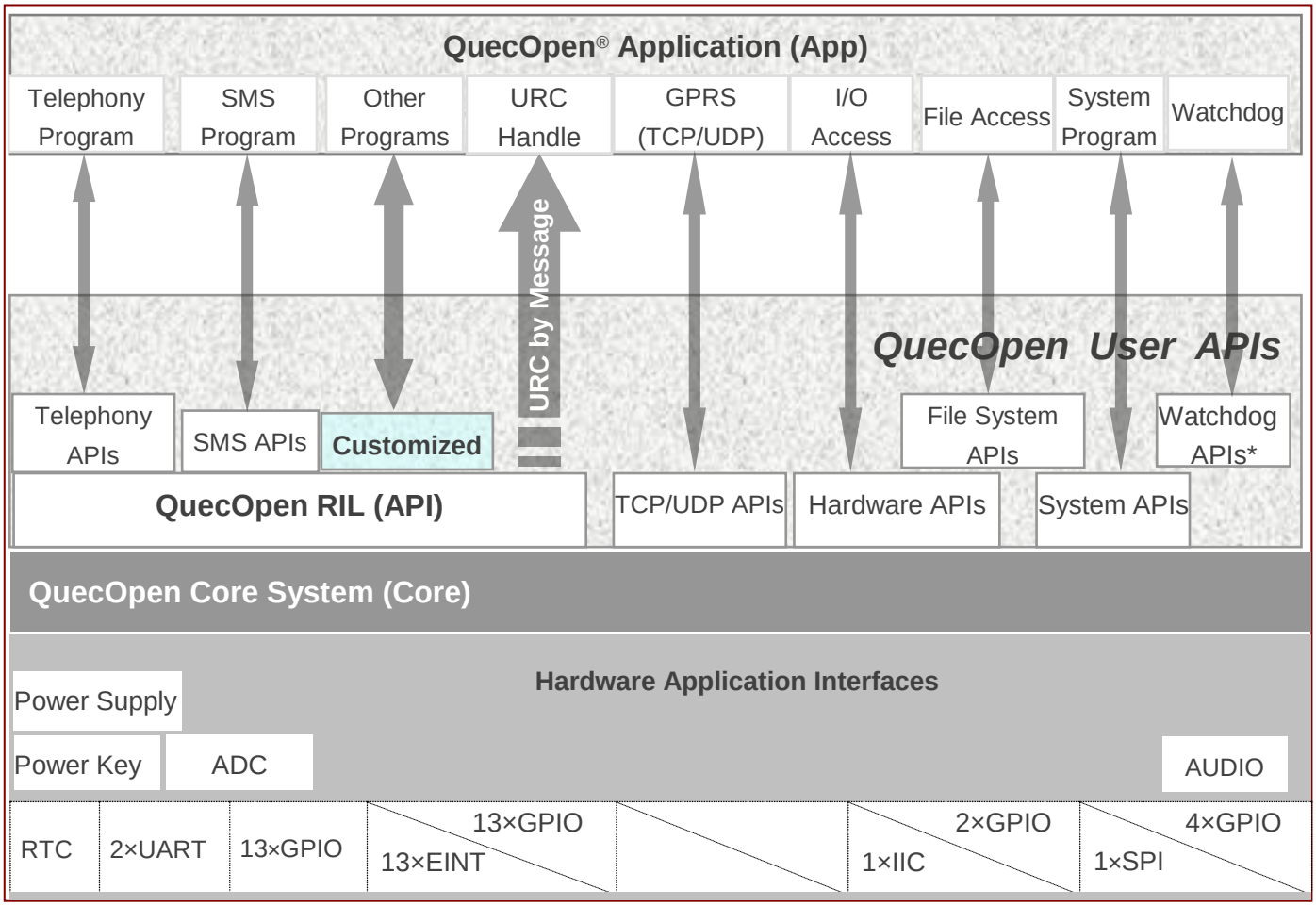


Figure 1: Fundamental Principle of QuecOpen® Software Architecture

EINT, IIC and SPI are multiplexing interfaces with GPIOs.

QuecOpen core system is a combination of hardware and software of GSM/GPRS module. It has built-in 32-bit XCPU RISC core, and has been built over SX RTOS with the characteristics of micro-kernel, real-time, multi-tasking, etc.

QuecOpen user APIs are designed for accessing to hardware resources, radio communications resources, user file system, or external devices. All APIs are introduced in ***Chapter 5***.

QuecOpen RIL is an open source layer, which enables developers to simply call APIs to send AT commands and get the response when API returns. Additionally, new APIs can be easily added to implement AT commands. For more details, please refer to *Quectel_QuecOpen_RIL_Application_Note*.

In QuecOpen RIL, all URC messages of the module have already been reinterpreted and the result is informed to App by system message. App will receive the message MSG_ID_URC_INDICATION when a URC arrives.

NOTE

"*" means under development.

2.2. Open Resources

2.2.1. Processor

32-bit XCPU RISC 208MHz.

2.2.2. Memory Schemes

User App Code Space: 320KB space available for image bin.
RAM Space: 100KB static memory and 500KB dynamic memory.
User File System Space: 400KB available (DFOTA included).

2.3. Interfaces

2.3.1. Serial Interfaces

M25-QuecOpen provides 2 configurable UART ports: main UART (UART1) and auxiliary UART (UART2). UART1 supports hardware flow control. Please refer to ***Chapter 5.7.1*** for software API functions.

The Debug port can only be used to debug or download firmware, and the debug function can only use the "CoolWatcher" tool to capture the log.

2.3.2. GPIOs

There are 13 I/O pins that can be configured into general purpose I/Os. All pins can be accessed by corresponding API functions. Please refer to ***Chapter 5.7.2*** for details.

2.3.3. EINT

QuecOpen supports external interrupt inputs. All general purpose I/Os can be configured into external interrupt inputs. Please do not use the EINT for highly frequent interrupt detections so as to avoid instability of the module. The EINT pins can be accessed by APIs. Please refer to ***Chapter 5.7.3*** for details.

2.3.4. ADC

There is an analog input pin that can be configured into ADC. The sampling period and count can be configured by APIs. Please refer to ***Chapter 5.7.4***.

Please refer to *Quectel_M25-QuecOpen_Hardware_Design* for the characteristics of ADC interface.

2.3.5. IIC

M25-QuecOpen module provides a hardware IIC interface, and supports simulated IIC interface. Please refer to ***Chapter 5.7.5*** for programming API functions.

2.3.6. SPI

M25-QuecOpen module provides a simulated SPI interface. Please refer to ***Chapter 5.7.6*** for programming API functions.

2.4. Development Environment

2.4.1. SDK

QuecOpen SDK provides the resources as follows for developers:

- Compiling environment.
- Development guide and other related documents.
- A set of header files that defines all API functions and type declaration.
- Source code for examples.
- Open source code for RIL.
- Download tool for application image bin file.
- Package tool for upgrade via DFOTA.

Please obtain the latest SDK package from Quectel Technical Supports support@quectel.com.

2.4.2. Editor

Text editors are available for editing codes, such as Source Insight, Visual Studio and even Notepad.

The Source Insight tool is recommended to be used to edit and manage codes. It is an advanced code editor and browser with built-in analysis for C/C++ program, and provides syntax highlighting, code navigation and customizable keyboard shortcuts.

2.4.3. Compiler and Compiling

2.4.3.1. Complier

QuecOpen uses GCC as the compiler, and the compiler edition is "CSDTK4".

2.4.3.2. Compiling

In QuecOpen, compiling commands are executed in command lines. The clean and compiling commands are respectively defined as follows.

```
make clean
make new
```

2.4.3.3. Compiling Output

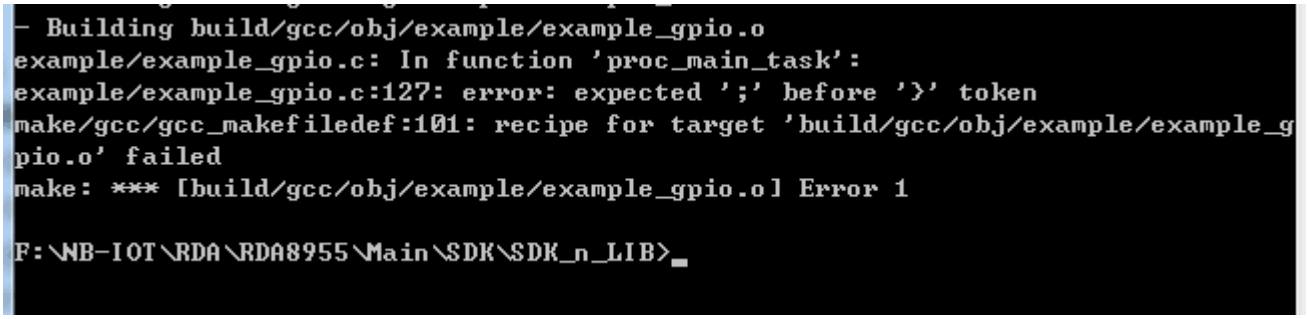
In a command line, some compiler processing information will be outputted during compiling. All warnings and errors are recorded in *\SDK\build\gcc\build.log*.

Therefore, if there exists any compilation error during compiling, please check *build.log* for the error line number and the error hints.

For example, in line 126 in *example_gpio.c*, the semicolon is missed intentionally.

```
00122:      // Register & open UART port
00123:      ret = Ql_UART_Register(UART_PORT1, CallBack_UART_Hdlr, NULL);
00124:      if (ret < QL_RET_OK)
00125:      {
00126:          Ql_Debug_Trace("Fail to register serial port[%d], ret=%d\r\n", UART_PORT1, ret)
00127:      }
```

When compiling this example, a compilation error will be shown in *build.log* as follows:



If there is no compilation error during compiling, the prompt for successful compiling will be given as below.

2.4.4. Download

QFlash tool is typically used to download the application bin. Please refer to *Quectel_QFlash_User_Guide* for more details about the tool and its usage.

2.4.5. How to Program

By default, the source code files store in *SDK\custom* directory.

2.4.5.1. Program Composition

The composition of QuecOpen program is described as follows.

Table 1: QuecOpen Program Composition

Item	Description
.h, *.def* files	Declarations for variables, functions and macros.
.c files	Source code implementations.
makefile	Define the destination object files and directories to be compiled.

2.4.5.2. Program Framework

The following codes are the least codes that comprise an QuecOpen embedded application.

```
void proc_main_task(s32 taskId)
{
    s32 ret;
    ST_MSG msg;
    //Register and open UART port
    ret = Ql_UART_Register(m_myUartPort, CallBack_UART_Hdlr, NULL);
    if (ret < QL_RET_OK)
    {
        Ql_Debug_Trace("Fail to register serial port[%d], ret=%d\r\n", m_myUartPort, ret);
    }
    ret = Ql_UART_Open(m_myUartPort, 115200, FC_NONE);
    if (ret < QL_RET_OK)
    {
        Ql_Debug_Trace("Fail to open serial port[%d], ret=%d\r\n", m_myUartPort, ret);
    }
        APP_DEBUG("QuecOpen: Customer Application\r\n");
        //Start message loop of this task
        while(TRUE)
        {
            Ql_OS_GetMessage(&msg);
            switch(msg.message)
            {
            case MSG_ID_RIL_READY:
                APP_DEBUG("<-- RIL is ready -->\r\n");
                Ql_RIL_Initialize();
                break;
            case MSG_ID_URC_INDICATION:
                //APP_DEBUG("<-- Received URC: type: %d, -->\r\n", msg.param1);
                switch (msg.param1)
                {
                case URC_SYS_INIT_STATE_IND:
                    APP_DEBUG("<-- Sys Init Status %d -->\r\n", msg.param2);
                    break;
                case URC_SIM_CARD_STATE_IND:
                    APP_DEBUG("<-- SIM Card Status:%d -->\r\n", msg.param2);
                    break;
                case URC_GSM_NW_STATE_IND:
                    APP_DEBUG("<-- GSM Network Status:%d -->\r\n", msg.param2);
                    break;
                case URC_GPRS_NW_STATE_IND:
```

```
                        APP_DEBUG("<-- GPRS Network Status:%d -->\r\n", msg.param2);
                        break;
                    case URC_CFUN_STATE_IND:
                        APP_DEBUG("<-- CFUN Status:%d -->\r\n", msg.param2);
                        break;
                    case URC_COMING_CALL_IND:
                        {
                            ST_ComingCall* pComingCall = (ST_ComingCall*)msg.param2;
                            APP_DEBUG("<-- Coming call, number:%s, type:%d -->\r\n",
pComingCall->phoneNumber, pComingCall->type);
                            break;
                        }
                    case URC_CALL_STATE_IND:
                        APP_DEBUG("<-- Call state:%d\r\n", msg.param2);
                        break;
                    case URC_NEW_SMS_IND:
                        APP_DEBUG("<-- New SMS Arrives: index=%d\r\n", msg.param2);
                        break;
                    case URC_MODULE_VOLTAGE_IND:
                        APP_DEBUG("<-- VBatt Voltage Ind: type=%d\r\n", msg.param2);
                        break;
                    default:
                        APP_DEBUG("<-- Other URC: type=%d\r\n", msg.param1);
                        break;
                    }
                    break;
                default:
                    break;
                }
            }
        }
```

The *proc_main_task* function is the entrance of embedded applications, just like the *main()* in C application.

Ql_OS_GetMessage is an important system function which enables the embedded application to receive messages from message queue of the task.

MSG_ID_RIL_READY is a system message sent by the RIL module to the main task.

MSG_ID_URC_INDICATION is a system message that indicates a new URC is coming.

2.4.5.3. Makefile

In QuecOpen, the program can be automatically compiled by the compiler according to the definitions in *makefile* from the path of *\SDK\make\gcc\gcc_makefile*. The profile of *makefile* has been pre-designed and is ready for use. However, it is necessary to change some settings before compiling program according to native conditions, such as the compiler environment path.

\SDK\make\gcc\gcc_makefile needs to be maintained. This *makefile* mainly includes:

- Environment path definition of compiler
- Preprocessor definitions
- Definitions for the paths that include files
- Source code directories and files to be compiled
- Library files to link

2.4.5.4. How to Add a .c File

Suppose that the new file is in *custom* directory, and the newly added *.c* files will be compiled automatically.

2.4.5.5. How to Add a Directory

If it is necessary to add a new directory in *custom*, please follow the steps below.

1. Add the name of the new directory in variable "SRC_DIRS" in \SDK\make\gcc\gcc_makefile, and define the source code files to be compiled.

```
#-------------------------------------------------------------------------------
# Configure source code dirctories
#-------------------------------------------------------------------------------
SRC_DIRS=example    \
         custom     \
         custom/config     \
         custom/fota/src   \
         custom/dfota/src   \
         ril/src    \
```

2. Define the source code files to be compiled in the new directory.

```
#-------------------------------------------------------------------------------
# Configure source code files to compile in the source code directories
#-------------------------------------------------------------------------------
SRC_SYS=$(wildcard custom/config/*.c)
SRC_SYS_RIL=$(wildcard ril/src/*.c)
SRC_EXAMPLE=$(wildcard example/*.c)
SRC_CUS=$(wildcard custom/*.c)
SRC_FOTA=$(wildcard custom/fota/src/*.c)
SRC_DFOTA=$(wildcard custom/dfota/src/*.c)

OBJS=\
    $(patsubst %.c, $(OBJ_DIR)/%.o, $(SRC_SYS))        \
    $(patsubst %.c, $(OBJ_DIR)/%.o, $(SRC_SYS_RIL))    \
    $(patsubst %.c, $(OBJ_DIR)/%.o, $(SRC_EXAMPLE))    \
    $(patsubst %.c, $(OBJ_DIR)/%.o, $(SRC_CUS))        \
    $(patsubst %.c, $(OBJ_DIR)/%.o, $(SRC_FOTA))      \
    $(patsubst %.c, $(OBJ_DIR)/%.o, $(SRC_DFOTA))      \
```

3 Base Data Types

3.1. Required Header File

In QuecOpen, the base data types are defined in *ql_type.h* header file.

3.2. Base Data Type

Table 2: Base Data Type

Type	Description
bool	Boolean variable (should be TRUE or FALSE). This variable is declared as follows: typedef unsigned char bool;
s8	8-bit signed integer. This variable is declared as follows: typedef signed char s8;
u8	8-bit unsigned integer. This variable is declared as follows: typedef unsigned char u8;
s16	16-bit signed integer. This variable is declared as follows: typedef signed short s16;
u16	16-bit unsigned integer. This variable is declared as follows: typedef unsigned short u16;
s32	32-bit signed integer. This variable is declared as follows: typedef int s32;
u32	32-bit unsigned integer. This variable is declared as follows: typedef unsigned int u32;
u64	64-bit unsigned integer. This variable is declared as follows: typedef unsigned long lone u64;

s64	64-bit signed integer. This variable is declared as follows: typedef long s64;
ticks	32-bit unsigned integer. This variable is declared as follows: typedef unsigned int ticks;

4 System Configuration

In *\SDK\custom\config* directory, applications can be reconfigured according to specific requirements for tasks addition, task stack size configuration and GPIO initialization status. All configuration files are named with a prefix "custom_".

Table 3: List of System Configuration Files

Configuration File	Description
custom_feature_def.h	Enable QuecOpen features including RIL and DFOTA. Generally, there is no need to change this file.
custom_task_cfg.h	Multitask configuration.
custom_sys_cfg.c	Other system configurations, including configuration files for power key and the specified GPIO pin for external watchdog.

4.1. Configuration of Tasks

QuecOpen supports multitask processing. It is recommended to simply follow suit to add a record in *custom_task_cfg.h* file to define a new task. QuecOpen supports one main task, and maximum ten subtasks.

If there are file operations in a task, the stack size must be 5KB at least.

Please avoid calling functions *Ql_Sleep()*, *Ql_OS_TakeSemaphore()* and *Ql_OS_TakeMutex()* as these functions will block the task and thus will make the task unable to fetch message from the message queue. If the message queue is filled up, the system may reboot unexpectedly.

4.2. Configuration of Customization Items

All customization items are configured in TLV (Type-Length-Value) in *custom_sys_cfg.c*. Please change the corresponding value to change App's features.

```
const ST_SystemConfig SystemCfg[] = {
    {SYS_CONFIG_APP_ENABLE_ID,          SYS_CONFIG_APPENABLE_DATA_SIZE,
    (void*)&appEnableCfg},
    {SYS_CONFIG_PWRKEY_DATA_ID,         SYS_CONFIG_PWRKEY_DATA_SIZE,
    (void*)&pwrkeyCfg    },
    {SYS_CONFIG_WATCHDOG_DATA_ID,   SYS_CONFIG_WATCHDOG_DATA_SIZE,
    (void*)&wtdCfg         },
    {SYS_CONFIG_DEBUG_MODE_ID,        SYS_CONFIG_DEBUGMODE_DATA_SIZE,
    (void*)&debugPortCfg},
    {SYS_CONFIG_END, 0,
NULL                                                                }
};
```

Table 4: Customization Items

Item	Type (T)	Length (L)	Default Value	Possible Value	Description
App Enabling	SYS_CONFIG_APP_ENABLE_ID	4	APP_ENABLE	APP_ENABLE APP_DISABLE	App enabling configuration
PWRKEY Pin Config	SYS_CONFIG_PWRKEY_DATA_ID	2	TRUE TRUE	TRUE/FALSE	Power on/off working mode.
GPIO for WTD Config	SYS_CONFIG_WATCHDOG_DATA_ID	8	PINNAME_PCM_OUT PINNAME_END	One value of *Enum_PinName*	GPIO for feeding watchdog. Please refer to ***Chapter 4.2.1***

4.2.1. GPIO for External Watchdog*

When an external watchdog is adopted to monitor the App, the module has to feed the watchdog in the whole period of the module's power-on including the processes of startup, App activation and upgrade.

Table 5: Participants for Feeding External Watchdog

Period	Feeding Host
Booting	Core system
App Running	App
Upgrading App by DFOTA	Core system

Therefore, it is suggested to only specify which GPIO is designed to feed the external watchdog.

```
static const ST_ExtWatchdogCfg wtdCfg = {
    PINNAME_PCM_OUT,    //Specify a pin or another GPIO to connect to the external watchdog.
    PINNAME_END         //Specify another pin for watchdog if needed.
};
```

NOTE

"*" means under development.

5 API Functions

5.1. System APIs

The header file *ql_system.h* declares system related API functions. These functions are essential to all customer applications. Please make sure the header file is included when using these functions.

QuecOpen provides interfaces that support multitasking, message, mutex, semaphore and event mechanism functions. These interfaces are used for multitask programming. The example *example_multitask.c* in QuecOpen SDK shows the proper usages of these API functions.

5.1.1. Usage

This chapter introduces critical operations and the API functions in system-level programming.

5.1.1.1. Receive Message

Please call *Ql_OS_GetMessage* to retrieve a message from the current task's message queue. The message can be a system message or a customized message.

5.1.1.2. Send Message

Please call *Ql_OS_SendMessage* to send messages to other tasks. To send a message, a message ID has to be defined. In QuecOpen, user message ID must be greater than 0x1000.

Step 1: Define a message ID.

```
#define     MSG_ID_USER_START     0x1000
#define     MSG_ID_MESSAGE1       (MSG_ID_USER_START + 1)
```

Step 2: Send the message.

```
Ql_OS_SendMessage(ql_subtask1, MSG_ID_MESSAGE1, 0, 0);
```

5.1.1.3. Mutex

A mutex object is a synchronization object whose state is set to signaled when it is not owned by any task while non-signaled when it is owned. A task can only own one mutex object at a time. For example, to prevent two tasks from being written to a shared memory at the same time, each task waits for ownership of a mutex object before executing the code for accessing the memory. After writing to the shared memory, the task releases the mutex object.

Step 1: Create a mutex. Please call *Ql_OS_CreateMutex* to create a mutex.
Step 2: Get a specified mutex. If mutex mechanism is to be used for programming, please call *Ql_OS_TakeMutex* to get a specified mutex ID.
Step 3: Release the specified mutex. Please call *Ql_OS_GiveMutex* to release the specified mutex.

5.1.1.4. Semaphore

A semaphore object is a synchronization object that maintains a count between zero and a specified maximum value. The count is decremented each time a task finishes waiting for the semaphore object and is incremented each time a task releases the semaphore. When the count reaches zero, no more tasks can successfully wait for the semaphore object state to be signaled. The state of a semaphore is set as signaled when its count is greater than zero and non-signaled when its count is zero.

Step 1: Create a semaphore. Please call *Ql_OS_CreateSemaphore* to create a semaphore.
Step 2: Get a specified semaphore. If semaphore mechanism is to be used for programming, please call *Ql_OS_TakeSemaphore* to get a specified semaphore ID.
Step 3: Release the specified semaphore. Please call *Ql_OS_GiveSemaphore* to release the specified semaphore.

5.1.1.5. Event

An event object is a synchronization object, which is useful in sending a signal to a thread indicating that a particular event has occurred. A task uses *Ql_OS_CreateEvent* function to create an event object, whose state can be explicitly set as signaled by the *Ql_OS_SetEvent* function.

5.1.1.6. Backup Critical Data*

QuecOpen has been designed with 13 blocks of system storage space to backup critical user data. Among the storage blocks, each block from block 1~8 can store 50 bytes of data, each block from block 9~12 can store 100 bytes of data, and block 13 can store 500 bytes of data.

Ql_SecureData_Store can be called to backup data, and *Ql_Userdata_Read()* can be called to read the backup data from backup space.

NOTE

"*" means under development.

5.1.2. API Functions

5.1.2.1. Ql_Reset

This function resets the system.

- **Prototype**

```
void Ql_Reset(u8 resetType)
```

- **Parameters**

resetType:
[In] Reset type. It must be 0.

- **Return Value**

None.

5.1.2.2. Ql_Sleep

This function suspends the execution of the current task until timeout interval elapses. The sleep time should not exceed 500ms, since if the task is suspended too long, it may receive too many messages to be dealt with.

- **Prototype**

```
void Ql_Sleep(u32 msec)
```

- **Parameters**

msec:
[In] The time interval for the execution to be suspended. Unit: ms.

- **Return Value**

None.

5.1.2.3. Ql_GetUID

This function gets the module's UID. UID is a 20-byte serial number identification. The probability that different modules have the same UID is 1ppm (1/10000000).

- **Prototype**

```
s32 Ql_GetUID(u8* ptrUID, u32 len)
```

- **Parameters**

ptrUID:
[In] Pointer to the buffer that is used to store the UID. The buffer length needs to be at least 20 bytes.

len:
[In] The *ptrUID* buffer length. The value must be less than or equal to the size of the buffer that *ptrUID* points to.

- **Return Value**

If the *ptrUID* is null, this function will return *QL_RET_ERR_INVALID_PARAMETER*. If this function reads the UID successfully, the length of UID will be returned.

5.1.2.4. Ql_GetCoreVer

This function gets the version ID of the core. The core version ID is a string with no more than 35 characters and ends with '\0'.

- **Prototype**

```
s32 Ql_GetCoreVer(u8* ptrVer, u32 len)
```

- **Parameters**

ptrVer:
[Out] Pointer to the buffer that is used to store the version ID of the core. The buffer length needs to be at least 35 bytes.

len:
[In] The *ptrVer* buffer length. The value must be less than or equal to the size of the buffer that *ptrVer* points to.

- **Return Value**

The return value is the length of version ID of the core if this function succeeds. Otherwise, the return value will be an error code. To get extended error information, please refer to the error codes in header file *ql_error.h*.

5.1.2.5. Ql_GetSDKVer

This function gets the version ID of SDK. The SDK version ID is a string with no more than 20 characters and ends with '\0'.

- **Prototype**

```
s32 Ql_GetSDKVer(u8* ptrVer, u32 len)
```

- **Parameters**

ptrVer:
[In] Pointer to the buffer that is used to store the version ID of SDK. The buffer length needs to be at least 20 bytes.

len:
[In] The *ptrVer* length. The value must be less than or equal to the size of the buffer that *ptrVer* points to.

- **Return Value**

The return value is the length of version ID if this function succeeds. Otherwise, the return value will be an error code. To get extended error information, please refer to the error codes in header file *ql_error.h*.

5.1.2.6. Ql_GetMsSincePwrOn

This function returns the number of milliseconds since the device has been booted.

- **Prototype**

```
u64 Ql_GetMsSincePwrOn (void)
```

- **Parameters**

Void.

- **Return Value**

Number of milliseconds.

5.1.2.7. Ql_OS_GetMessage

This function gets a message from the message queue of the current task. When there is no message in the queue, the task will be in waiting state.

- **Prototype**

```
s32 Ql_OS_GetMessage(ST_MSG* msg)
```

```
typedef struct {
    u32   message;
    u32   param1;
    u32   param2;
    u32   srcTaskId;
} ST_MSG;
```

- **Parameters**

msg:
[In] Pointer to the *ST_MSG* struct.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.

5.1.2.8. Ql_OS_SendMessage

This function sends messages between tasks. The destination task receives messages with *Ql_OS_GetMessage*.

- **Prototype**

```
s32 Ql_OS_SendMessage (s32 destTaskId, u32 msgId, u32 param1, u32 param2)
```

- **Parameters**

destTaskId:
[In] The maximum value is 10. The destination task is main task if the value is 0. The destination task is subtask if the value is between 1 and 10.

msgId:
[In] User message ID, which must be bigger than 0xFF.

param1:
[In] User data.

param2:
[In] User data.

- **Return Value**

OS_SUCCESS: indicates the function is executed successfully.
OS_INVALID_ID: indicates the message ID is invalid.
OS_Q_FULL: indicates the message queue is full.
Other values: indicates it fails to send the message.

5.1.2.9. Ql_OS_CreateMutex

This function creates a mutex. A handle of created mutex will be returned if creation succeeds. 0 indicates failure. If the same mutex has already been created, this function may also return a valid handle, but the *Ql_GetLastError* function returns *ERROR_ALREADY_EXISTS*.

- **Prototype**

```
u32 Ql_OS_CreateMutex(char *mutexName)
```

- **Parameters**

mutexName:
[In] Name of the mutex to be created.

- **Return Value**

A handle of the created mutex. 0 indicates failure.

5.1.2.10. Ql_OS_TakeMutex

This function obtains an instance of a specified mutex. If the mutex ID is invalid, the system may crash.

- **Prototype**

```
void Ql_OS_TakeMutex(u32 mutexId)
```

- **Parameters**

mutexId:
[In] Destination mutex to be taken.

- **Return Value**

None.

5.1.2.11. Ql_OS_GiveMutex

This function releases an instance of a specified mutex.

- **Prototype**

```
void Ql_OS_GiveMutex(u32 mutexId)
```

- **Parameters**

mutexId:
[In] Destination mutex to be given.

- **Return Value**

None.

5.1.2.12. Ql_OS_CreateSemaphore

This function creates a counting semaphore. A handle of created semaphore will be returned, if creation succeeds. 0 indicates failure. If the same semaphore has already been created, this function may also return a valid handle, but the *Ql_GetLastError* function returns *ERROR_ALREADY_EXISTS*.

- **Prototype**

```
u32 Ql_OS_CreateSemaphore(char *semName, u32 maxCount)
```

- **Parameters**

semName:
[In] Name of the semaphore to be created.

maxCount:
[In] The maximum count of the semaphore.

- **Return Value**

A handle of the created semaphore. 0 indicates failure.

5.1.2.13. Ql_OS_TakeSemaphore

This function obtains an instance of a specified semaphore. If the mutex ID is invalid, the system may be crushed.

- **Prototype**

```
u32 Ql_OS_TakeSemaphore(u32 semId, bool wait)
```

- **Parameters**

semId:
[In] The destination semaphore to be taken.

wait:
[In] The waiting style determining if a task waits infinitely (TRUE) or returns immediately (FALSE).

- **Return Value**

OS_SUCCESS: indicates the function is executed successfully.
OS_SEM_NOT_AVAILABLE: indicates the semaphore is unavailable immediately.

5.1.2.14. Ql_OS_CreateEvent

This function waits until a specified type of event is in the signaled state. Different types of events can be specified for different purposes. The event flags are defined in *Enum_EventFlag*.

- **Prototype**

```
u32 Ql_OS_CreateEvent(char* evtName)
```

- **Parameters**

evtName:
[In] Name of the event to be created.

- **Return Value**

An event ID that identifies this event is unique.

5.1.2.15. Ql_OS_WaitEvent

This function waits until a specified type of event is in signaled state. Different types of events can be specified for different purposes. The event flags are defined in *Enum_EventFlag*.

- **Prototype**

```
s32 Ql_OS_WaitEvent(u32 evtId, u32 evtFlag)
```

- **Parameters**

evtId:
Event ID that is returned by calling *Ql_OS_CreateEvent()*.

evtFlag:
Event flag type. See *Enum_EventFlag*.

- **Return Value**

0 indicates the function is executed successfully and any other value indicates it fails to execute the function.

5.1.2.16. Ql_OS_SetEvent

This function sets a specified event flag. Any task waiting on the event, whose event flag request is satisfied, is resumed.

- **Prototype**

```
s32 Ql_OS_SetEvent(u32 evtId, u32 evtFlag)
```

- **Parameters**

evtId:
Event ID that is returned by calling *Ql_OS_CreateEvent()*.

evtFlag:
Event flag type. See *Enum_EventFlag*.

- **Return Value**

0 indicates the function is executed successfully and any other value indicates it fails to execute the function.

5.1.2.17. Ql_OS_GiveSemaphore

This function releases an instance of a specified semaphore.

```
void Ql_OS_GiveSemaphore(u32 semId)
```

- **Parameters**

semId:
[In] The destination semaphore to be given.

- **Return Value**

None.

5.1.2.18. Ql_OS_GetCurrenTaskLeftStackSize

This function gets the number of bytes left in the current task stack.

- **Prototype**

```
u32 Ql_OS_GetCurrenTaskLeftStackSize(void)
```

- **Parameters**

Void.

- **Return Value**

The return value is the number of bytes if this function succeeds. Otherwise an error code is returned.

5.1.3. Possible Error Codes

The frequent error codes, which could be returned by APIs in multitask programming, are enumerated in the *Enum_OS_ErrCode*.

```
/*************************************************************
* Error Code Definition
 *************************************************************/
typedef enum {
    OS_SUCCESS,
    OS_ERROR,
    OS_Q_FULL,
    OS_Q_EMPTY,
    OS_SEM_NOT_AVAILABLE,
    OS_WOULD_BLOCK,
    OS_MESSAGE_TOO_BIG,
    OS_INVALID_ID,
    OS_NOT_INITIALIZED,
    OS_INVALID_LENGHT,
    OS_NULL_ADDRESS,
    OS_NOT_RECEIVE,
    OS_NOT_SEND,
    OS_MEMORY_NOT_VALID,
    OS_NOT_PRESENT,
    OS_MEMORY_NOT_RELEASE
```

```
} Enum_OS_ErrCode;
```

5.1.4. Examples

1. Mutex Example:

```
static int    s_iMutexId = 0;
static int    s_iSemMutex = 0;
//Create a mutex first.
s_iMutexId = Ql_OS_CreateMutex("MyMutex");

void MutextTest(int iTaskId)    //Two tasks run this function at the same time.
{

        //Get the mutex.
        Ql_OS_TakeMutex(s_iMutexId);

        //Another caller prints this sentence 3 seconds later.
        Ql_Sleep(3000);

      //Release the mutex 3 seconds later.
      Ql_OS_GiveMutex(s_iMutexId);
}
```

2. Semaphore Example:

```
static int s_iSemaphoreId = 0;   //Define a semaphore ID.
static int s_iTestSemNum =4;    //Set the maximum semaphore number as 4.

//Create a semaphore first.
s_iSemaphoreId = Ql_OS_CreateSemaphore("MySemaphore", s_iTestSemNum);
void SemaphoreTest(int iTaskId)
{
      int iRet = -1;

      //Get the mutex.
      iRet = Ql_OS_TakeSemaphore(s_iSemaphoreId, TRUE);//TRUE or FLASE indicates the task should
                                                        wait infinitely or return immediately.
      Ql_OS_TakeMutex(s_iSemMutex);
      s_iTestSemNum--;            //One semaphore is being used.
      Ql_OS_GiveMutex(s_iSemMutex);

      Ql_Sleep(3000);
```

```
    //Release the semaphore 3 seconds later.
    Ql_OS_GiveSemaphore(s_iSemaphoreId);
    s_iTestSemNum++;          //One semaphore is released.
    Ql_Debug_Trace("\r\n<--=========Task[%d]: s_iTestSemNum=%d-->", iTaskId, s_iTestSemNum);
}
```

5.2. Time APIs

QuecOpen provides time-related APIs including setting or getting local time, convertion between seconds and calendar time.

5.2.1. Usage

Calendar time is measured from a standard point in time to the current time elapsed in seconds, and generally 00:00:00 on January 1st, 1970 is set as the standard point in time.

5.2.2. API Functions

The time structure is defined as follows:

```
typedef struct {
    s32 year;           //Range: 2000~2127
    s32 month;
    s32 day;
    s32 hour;           //In 24-hour time system
    s32 minute;
    s32 second;
    s32 timezone;       //Range: -12~12
}ST_Time;
```

The field "timezone" defines the time zone. A negative number indicates the Western time zone, and a positive number indicates the Eastern time zone. For instance, the time zone of Beijing is East Area 8, then timezone=8; the time zone of Washington is West Zone 5, the timezone=-5.

5.2.2.1. Ql_SetLocalTime

This function sets the current local date and time.

- **Prototype**

```
s32 Ql_SetLocalTime(ST_Time *datetime)
```

- **Parameters**

datetime:
[In] Pointer to the "ST_Time" structure.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_PARAM: indicates parameter error.

5.2.2.2. Ql_GetLocalTime

This function gets the current local date and time.

- **Prototype**

```
ST_Time * Ql_GetLocalTime(ST_Time * dateTime)
```

- **Parameters**

dateTime:
[Out] Pointer to the "ST_Time" Structure.

- **Return Value**

If the function is executed successfully, the current local date and time are returned. NULL indicates failure.

5.2.2.3. Ql_Mktime

This function gets the total seconds elapsed since 00:00:00 on January 1st, 1970.

- **Prototype**

```
u64 Ql_Mktime(ST_Time *dateTime)
```

- **Parameters**

dateTime:
[In] Pointer to the "ST_Time" Structure.

- **Return Value**

Return the total seconds.

5.2.2.4. Ql_MKTime2CalendarTime

This function converts the seconds elapsed since 00:00:00 on January 1st, 1970 to the local date and time.

- **Prototype**

```
ST_Time *Ql_MKTime2CalendarTime(u64 seconds, ST_Time *pOutDateTime)
```

- **Parameters**

seconds:
[In] The seconds elapsed since 00:00:00 on January 1st, 1970.

pOutDateTime:
[Out] Pointer to the "ST_Time" Structure.

- **Return Value**

If the function is executed successfully, the current local date and time are returned. NULL indicates failure.

5.2.3. Example

The following codes show how to use the time-related APIs.

```
s32 ret;
u64 sec;
ST_Time datetime, *tm;
datetime.year=2013;
datetime.month=6;
datetime.day=12;
datetime.hour=18;
datetime.minute=12;
datetime.second=13;
datetime.timezone=-8;

//Set local time.
ret=Ql_SetLocalTime(&datetime);
Ql_Debug_Trace("\r\n<--Ql_SetLocalTime,ret=%d -->\r\n",ret);
Ql_Sleep(5000);

//Get local time.
tm=Ql_GetLocalTime(&datetime);
Ql_Debug_Trace("<--%d/%d/%d %d:%d:%d %d -->\r\n",tm->year, tm->month, tm->day, tm->hour, tm
```

```
->minute, tm->second, tm->timezone);

//Get total seconds elapsed since 00:00:00 on January 1st, 1970.
sec=Ql_Mktime(tm);
Ql_Debug_Trace("\r\n<--Ql_Mktime,sec=%lld -->\r\n",sec);

//Convert the seconds elapsed since 00:00:00 on January 1st, 1970 to local date and time.
tm=Ql_MKTime2CalendarTime(sec, & datetime);
Ql_Debug_Trace("<--%d/%d/%d %d:%d:%d %d -->\r\n",tm->year, tm->month, tm->day, tm->hour, tm
->minute, tm->second, tm->timezone);
```

5.3. Timer APIs

QuecOpen provides two kinds of timers. One is "Common Timer", and the other is "Fast Timer". QuecOpen system allows maximum 10 Common Timers running at the same time in a task. The system provides only one Fast Timer for App. The accuracy of the Fast Timer is relatively higher than a common timer.

5.3.1. Usage

Ql_Timer_Register() can be used to create a common timer and register an interrupt handler. And a timer ID, which is an unsigned integer, must be specified. *Ql_Timer_Start()* can start the created timer while *Ql_Timer_Stop()* can stop the running timer.

Ql_Timer_RegisterFast() can be called to create the Fast Timer and register an interrupt handler. *Ql_Timer_Start()* can start the created timer while *Ql_Timer_Stop()* can stop the running timer. The minimum interval for Fast Timer should be an integral multiple of 10ms.

5.3.2. API Functions

5.3.2.1. Ql_Timer_Register

This function registers a Common Timer. Each task supports 10 Common Timers running at the same time. Only the task which registers the timer can start and stop the timer.

- **Prototype**

```
s32 Ql_Timer_Register(u32 timerId, Callback_Timer_OnTimer callback_onTimer, void* param)
typedef void(*Callback_Timer_OnTimer)(u32 timerId, void* param)
```

- **Parameters**

timerId:
[In] Timer ID. It must be ensured that the ID is the only one under QuecOpen task. It should also differ from the ID registered by *Ql_Timer_RegisterFast*.

callback_onTimer:
[Out] Give notifications when the timer arrives.

param:
[In] One customized parameter that can be passed into the callback functions.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_ERROR: indicates the register is failed.
QL_RET_ERR_PARAM: indicates parameter error.
QL_RET_ERR_INVALID_PARAM: indicates the timer is invalid.
QL_RET_ERR_FULL: indicates all timers are used up.

5.3.2.2. Ql_Timer_RegisterFast

This function registers a Fast Timer. It only supports one Fast Timer for App. Please do not add any task schedule in the interrupt handler of the Fast Timer.

- **Prototype**

```
s32 Ql_Timer_RegisterFast(u32 timerId, Callback_Timer_OnTimer callback_onTimer, void* param)
typedef void(*Callback_Timer_OnTimer)(u32 timerId, void* param)
```

- **Parameters**

timerId:
[In] Timer ID. It should not be the same as the one that is registered by *Ql_Timer_Register*.

callback_onTimer:
[Out] Give notifications when the timer arrives.

param:
[In] One customized parameter that can be passed into the callback functions.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_ERROR: indicates the register is failed.
QL_RET_ERR_PARAM: indicates parameter error.

QL_RET_ERR_INVALID_PARAM: indicates the timer is invalid.
QL_RET_ERR_FULL: indicates all timers are used up.

5.3.2.3. Ql_Timer_Start

This function starts up a specified timer. The task in which a specified timer is started or stopped should be the one that registers the timer.

- **Prototype**

```
s32 Ql_Timer_Start(u32 timerId, u32 interval, bool autoRepeat)
```

- **Parameters**

timerId:
[In] Timer ID, which must be registered.

interval:
[In] Set the timer interval. Unit: ms. If a Common Timer is started, the interval must be greater than or equal to 1ms. If a Fast Timer is started, the interval must be an integer multiple of 10ms.

autoRepeat:
[In] TRUE or FALSE. FALSE indicates the timer is executed once; TRUE indicates the timer is executed repeatedly.

- **Return Value**

QL_RET_OK: indicates the timer is successfully started.
QL_RET_ERR_ERROR: indicates it fails to start the timer.
QL_RET_ERR_NOT_INIT: indicates it fails to start the timer, possibly because the timer is not registered.
QL_RET_ERR_PARAM: indicates parameter error.

5.3.2.4. Ql_Timer_Stop

This function stops a specified timer. The task in which a specified timer is started or stopped should be the one that registers the timer.

- **Prototype**

```
s32 Ql_Timer_Stop(u32 timerId)
```

- **Parameters**

timerId:
[In] Timer ID. The timer has been started by calling *Ql_Timer_Start* previously.

- **Return Value**

QL_RET_OK: indicates the timer is successfully stopped.
QL_RET_ERR_ERROR: indicates it fails to stop the timer.
QL_RET_ERR_PARAM: indicates parameter error.

5.3.3. Example

The following codes show how to register and start a Common Timer.

```
s32 ret;
u32 timerId=999;              //Timer ID is 999
u32 interval=2 * 1000;        //2 seconds
bool autoRepeat=TRUE;
u32 param=555;

//Callback function.
void Callback_Timer(u32 timerId, void* param)
{
    ret=Ql_Timer_Stop(timerId);
    Ql_Debug_Trace("\r\n<--Stop: timerId=%d,ret = %d -->\r\n", timerId ,ret);
}

//Register the timer.
ret=Ql_Timer_Register(timerId, Callback_Timer, &param);
Ql_Debug_Trace("\r\n<--Register: timerId=%d, param=%d,ret=%d -->\r\n", timerId ,param,ret);

//Start the timer.
ret=Ql_Timer_Start(timerId, interval, autoRepeat);
Ql_Debug_Trace("\r\n<--Start: timerId=%d,repeat=%d,ret=%d -->\r\n", timerId , autoRepeat,ret);
```

5.4. Power Management APIs

Power management contains the power-related operations, such as power-down, power key control and low power consumption mode enablement/disablement.

5.4.1. Usage

5.4.1.1. Power on/off

Ql_PowerDown function can be called to power off the module if PWRKEY is not short-circuited to ground. And this action will reset the module if PWRKEY has been short-circuited to ground.

5.4.1.2. Sleep Mode

Ql_SleepEnable function is used to enable the sleep mode of module. The module enters sleep mode when it is idle.

The incoming call, SMS or TCP data can wake up the module from sleep mode. And the module will automatically enter sleep mode when it is idle again.

If the timeout of timer or an interrupt event wakes up the module, the module will not enter sleep mode automatically and *Ql_SleepEnable* has to be called to make it enter sleep mode.

Ql_SleepDisable can disable the sleep mode when module is woken up.

5.4.2. API Functions

5.4.2.1. Ql_PowerDown

This function powers off the module. When calling this API to power down the module, the module will complete the network anti-registration first. So it takes more time to power off the module.

- **Prototype**

```
void Ql_PowerDown(u8 pwrDwnType)
```

- **Parameters**

pwrDwnType:
[In] Power-off type of this function. 1 indicates normal power-off.

- **Return Value**

None.

5.4.2.2. Ql_SleepEnable

This function enables the sleep mode of module. The module will enter sleep mode when it is under idle state.

- **Prototype**

```
s32 Ql_ SleepEnable(void)
```

- **Parameters**

Void.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
Ql_RET_NOT_SUPPORT: indicates the function is not supported by the current SDK version.

5.4.2.3. Ql_SleepDisable

This function disables the sleep mode of the module.

- **Prototype**

```
s32 Ql_SleepDisable(void)
```

- **Parameters**

Void.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
Ql_RET_NOT_SUPPORT: indicates the function is not supported by the current SDK version.

5.4.3. Example

The following sample codes show how to enter and exit from sleep mode in the interrupt handler.

```
void Eint_CallBack _Hdlr (Enum_PinName eintPinName, Enum_PinLevel pinLevel, void* customParam)
{
    If (0==pinLevel)
    {
        SYS_DEBUG( DBG_Buffer,"DTR set to low=%d    wake !!\r\n", level);
        Ql_SleepDisable(); //Enter sleep mode.
```

```
    }else{
            SYS_DEBUG( DBG_Buffer,"DTR set to high=%d    Sleep \r\n", level);
          Ql_SleepEnable(); //Exit from sleep mode.
    }
}
```

5.5. Memory APIs

QuecOpen operating system supports dynamic memory management. *Ql_MEM_Alloc* and *QL_MEM_Free* functions are used to allocate and release the dynamic memory, respectively.

The dynamic memory is system heap space. And the maximum available system heap of an application is 500KB.

Ql_MEM_Alloc and *QL_MEM_Free* must be present in pairs. Otherwise, memory leakage occurs.

5.5.1. Usage

Step 1: Call *Ql_MEM_Alloc()* to apply for a block of memory with a specified size. The memory allocated by *Ql_MEM_Alloc()* is from system heap.

Step 2: If the memory block is not needed anymore, please call *Ql_MEM_Free()* to free the memory block that is previously allocated by calling *Ql_MEM_Alloc()*.

5.5.2. API Functions

5.5.2.1. Ql_MEM_Alloc

This function allocates memory with a specified size in memory heap.

- **Prototype**

```
void   *Ql_MEM_Alloc (u32 size)
```

- **Parameters**

size:
[In] Number of memory bytes to be allocated.

- **Return Value**

A pointer of void type to the address of allocated memory. NULL will be returned if the allocation fails.

5.5.2.2. Ql_MEM_Free

This function frees the memory that is allocated by *Ql_MEM_Alloc*.

- **Prototype**

```
void Ql_MEM_Free (void *ptr)
```

- **Parameters**

ptr:
[In] Previously allocated memory block to be freed.

- **Return Value**

None.

5.5.3. Example

The following codes show how to allocate and free a specified size memory.

```
char *pch=NULL;

//Allocate the memory.
pch=(char*)Ql_MEM_Alloc(1024);
if (pch !=NULL)
{
    Ql_Debug_Trace("Successfully apply for memory, pch=0x%x\r\n", pch);
}else{
    Ql_Debug_Trace("Fail to apply for memory, size=%d\r\n", 1024);
}
//Free the memory.
Ql_MEM_Free(pch);
pch=NULL;
```

5.6. File System APIs

QuecOpen supports user file system, and provides a set of complete API functions to create, access and delete files and directories. This chapter describes these API usages and detailed functions.

5.6.1. Usage

The type of storage is divided into two kinds. One is the UFS in the flash, and the other is RAM file system. The RAM file does not support directory structure. Please select the storage location according to specific needs. A relative path must be used if a file or directory is to be created/opened. For example, the file can be set as *filename.ext* when a file is created in the root of the UFS.

- *Ql_FS_GetTotalSpace* function is used to get the total space on the flash or SD card.
- *Ql_FS_GetFreeSpace* function is used to get the free space on the flash or SD card.
- *Ql_FS_GetSize* function is used to get the size of a specified file in bytes.
- *Ql_FS_Open* function is used to create or open a file. The opening and access modes of the file must be defined. To know the usage of this function, please refer to ***Chapter 5.6.2.1***.
- *Ql_FS_Read* and *Ql_FS_Write* functions are used to read and write a file. Please ensure that the file has been opened before reading/writing.
- *Ql_FS_Seek* and *Ql_FS_GetFilePosition* functions are used to set and get the position of the file pointer. Please ensure that the file has been opened before the operations.
- *Ql_FS_Truncate* function is used to truncate a specified file to zero length.
- *Ql_FS_Delete* and *Ql_FS_Check* functions are used to delete and check a file.
- *Ql_FS_CreateDir*, *Ql_FS_DeleteDir* and *Ql_FS_CheckDir* functions are used to create, delete and check a specified directory.

NOTES

1. The RAM file does not support directory structure.
2. This stack size of the task, in which file operations will be executed, cannot be less than 5KB.

5.6.2. API Functions

5.6.2.1. Ql_FS_Open

This function opens or creates a file with a specified name.

- **Prototype**

```
s32 Ql_FS_Open(char* lpFileName, u32 flag)
```

- **Parameters**

lpFileName:
[In] The file name. The name should be within 252 characters. A relative path, such as *filename.ext* or *dirname\filename.ext* must be used.

flag:
[In] A u32 data type that defines opening and access modes of the file. The possible values are shown as follows:
QL_FS_READ_WRITE: indicates the file can be read and written.
QL_FS_READ_ONLY: indicates the file can be read only.
QL_FS_CREATE: indicates to open the file if it exists and to create the file if it does not exist.
QL_FS_CREATE_ALWAYS: indicates to create a new file. If the file already exists, the function overwrites the file and clears the existing attributes.

- **Return Value**

The return value specifies a file handle if this function succeeds. Otherwise an error code is returned.

QL_RET_ERR_FILE_NO_CARD: indicates there is no SD card.
QL_RET_ERR_PARAM: indicates parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: indicates the file name is too long.
QL_RET_ERR_FILEOPENFAILED: indicates it fails to open the file.
QL_RET_ERR_FS_FATAL_ERR1: indicates occurrence of some fatal error.

5.6.2.2. Ql_FS_OpenRAMFile

This function opens or creates a file with a specified name in the RAM. A prefix "RAM:" needs to be added in front of the file name, and at most 15 files can be created.

- **Prototype**

```
s32 Ql_FS_OpenRAMFile(char *lpFileName, u32 flag, u32 ramFileSize)
```

- **Parameters**

lpFileName:
[In] The file name. The name should be within 252 characters. A relative path such as *RAM: filename.ext* must be used.

flag:
[In] A u32 data type that defines the file's opening and access modes. The possible values are shown as follows:
QL_FS_READ_WRITE: indicates the file can be read and written.
QL_FS_READ_ONLY: indicates the file can be read only.
QL_FS_CREATE: indicates to open the file if it exists and to create the file if it does not exist.
QL_FS_CREATE_ALWAYS: indicates to create a new file. If the file already exists, the function overwrites the file and clears the existing attributes.

ramFileSize:
[In] The size of the specified file to be created.

- **Return Value**

The return value specifies a file handle if this function succeeds. Otherwise an error code is returned.

QL_RET_ERR_PARAM: indicates parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: indicates the file name is too long.
QL_RET_ERR_FILEOPENFAILED: indicates it fails to open the file.

5.6.2.3. Ql_FS_Read

This function reads the data in a specified file from the position indicated by the file pointer. After the reading operation is completed, the file pointer is adjusted by the number of bytes actually read.

- **Prototype**

```
s32 Ql_FS_Read(s32 fileHandle, u8 *readBuffer, u32 numberOfBytesToRead, u32
*numberOfBytesRead)
```

- **Parameters**

fileHandle:
[In] The file handle to be read. A return value of *Ql_FS_Open* function.

readBuffer:
[Out] Pointer to the buffer that is used to receive the data read from the file.

numberOfBytesToRead:
[In] Number of bytes to be read.

numberOfBytesRead:
[Out] Number of bytes that have been read. Set this value to zero before starting reading or checking errors.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_FILEREADFAILED: indicates it fails to read the file.

5.6.2.4. Ql_FS_Write

This function writes data from a buffer to a specified file, and returns the actual number of written bytes.

- **Prototype**

```
s32 Ql_FS_Write(s32 fileHandle, u8 *writeBuffer, u32 numberOfBytesToWrite, u32
*numberOfBytesWritten)
```

- **Parameters**

fileHandle:
[In] The file handle to be written, which is a return value of *Ql_FS_Open* function.

writeBuffer:
[In] Pointer to the buffer that is used to contain the data to be written to the file.

numberOfBytesToWrite:
[In] Number of bytes to be written to the file.

numberOfBytesWritten:
[Out] Pointer to the number of bytes already written by calling the function.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_FILEDISKFULL: indicates the file disk is full.
QL_RET_ERR_FILEWRITEFAILED: indicates it fails to write file.

5.6.2.5. Ql_FS_Seek

This function repositions the pointer in the previously opened file.

- **Prototype**

```
s32 Ql_FS_Seek(s32 fileHandle, s32 offset, u32 whence)
```

- **Parameters**

fileHandle:
[In] The file handle, which is the return value of *Ql_FS_Open* function.

offset:
[In] Number of bytes to be moved in the file pointer.

whence:
[In] Pointer movement mode, which must be one of the following values.

```
typedef enum
{
    QL_FS_FILE_BEGIN,
    QL_FS_FILE_CURRENT,
    QL_FS_FILE_END
} Enum_FsSeekPos;
```

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_FILESEEKFAILED: indicates it fails to seek the file.

5.6.2.6. Ql_FS_GetFilePosition

This function gets the current value of the file pointer.

- **Prototype**

```
s32 Ql_FS_GetFilePosition(s32 fileHandle)
```

- **Parameters**

fileHandle:
[In] The file handle, which is the return value of *Ql_FS_Open* function.

- **Return Value**

The return value is the current offset from the beginning of the file if this function succeeds. Otherwise, the return value is an error code.

QL_RET_ERR_FILEFAILED: indicates it fails to operate the file.

5.6.2.7. Ql_FS_Truncate

This function truncates the specified file to zero length.

- **Prototype**

```
s32 Ql_FS_Truncate(s32 fileHandle)
```

- **Parameters**

fileHandle:
[In] The file handle, which is the return value of *Ql_FS_Open* function.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_FILEFAILED: indicates it fails to operate the file.

5.6.2.8. Ql_FS_Flush

This function forces the data remaining in the file buffer to be written to the file.

- **Prototype**

```
void Ql_FS_Flush(s32 fileHandle)
```

- **Parameters**

fileHandle:
[In] The file handle, which is the return value of *Ql_FS_Open* function.

- **Return Value**

None.

5.6.2.9. Ql_FS_Close

This function closes the file associated with the file handle and makes the file unavailable for reading or writing.

- **Prototype**

```
void Ql_FS_Close(s32 fileHandle)
```

- **Parameters**

fileHandle:
[In] The file handle, which is the return value of *Ql_FS_Open* function.

- **Return Value**

None.

5.6.2.10. Ql_FS_GetSize

This function retrieves the size of a specified file in bytes.

- **Prototype**

```
s32 Ql_FS_Delete(char *lpFileName)
```

- **Parameters**

lpFileName:
[In] The file name. The name should be within 252 characters. A relative path, such as *filename.ext* or *dirname\filename.ext*, must be used.

- **Return Value**

The return value is the bytes of the file if this function succeeds. Otherwise, the return value is an error code.

QL_RET_ERR_PARAM: indicates parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: indicates the file name is too long.
QL_RET_ERR_FILEFAILED: indicates it fails to operate the file.
QL_RET_ERR_FILE_NO_CARD: indicates there is no SD card.

5.6.2.11. Ql_FS_Delete

This function deletes an existing file.

- **Prototype**

```
s32 Ql_FS_Delete(char *lpFileName)
```

- **Parameters**

lpFileName:
[In] The file name. The name should be within 252 characters. A relative path, such as *filename.ext* or *dirname\filename.ext*, must be used.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_PARAM: indicates parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: indicates the file name is too long.
QL_RET_ERR_FILEFAILED: indicates it fails to operate the file.
QL_RET_ERR_FILENOTFOUND: indicates the file is not found.

5.6.2.12. Ql_FS_Check

This function checks whether the file exists or not.

- **Prototype**

```
s32 Ql_FS_Check(char *lpFileName)
```

- **Parameters**

lpFileName:
[In] The file name. The name should be within 252 characters. A relative path, such as *filename.ext* or *dirname\filename.ext*, must be used.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_PARAM: indicates parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: indicates the file name is too long.
QL_RET_ERR_FILEFAILED: indicates it fails to operate the file.
QL_RET_ERR_FILENOTFOUND: indicates the file is not found.

5.6.2.13. Ql_FS_Rename

This function renames an existing file.

- **Prototype**

```
s32 Ql_FS_Rename(char *lpFileName, char *newLpFileName)
```

- **Parameters**

lpFileName:
[In] The current name of the file. The name should be within 252 characters. A relative path, such as *filename.ext* or *dirname\filename.ext*, must be used.

newLpFileName:
[In] The new name of the file. The new name is different from the existing names and should be within 252 characters. A relative path, such as *filename.ext* or *dirname\filename.ext*, must be used.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_PARAM: indicates parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: indicates the file name is too long.

QL_RET_ERR_FILEFAILED: indicates it fails to operate the file.
QL_RET_ERR_FILE_NO_CARD: indicates there is no SD card.

5.6.2.14. Ql_FS_CreateDir

This function creates a directory.

- **Prototype**

```
s32 Ql_FS_CreateDir(char *lpDirName)
```

- **Parameters**

lpDirName:
[In] The name of the directory. The name should be within 252 characters. A relative path, such as *dirname1* or *dirname1\dirname2,* must be used.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_PARAM: indicates parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: indicates the file name is too long.
QL_RET_ERR_FILEFAILED: indicates it fails to operate the file.
QL_RET_ERR_FILE_NO_CARD: indicates there is no SD card.

5.6.2.15. Ql_FS_DeleteDir

This function deletes an existing directory.

- **Prototype**

```
s32 Ql_FS_DeleteDir(char *lpDirName)
```

- **Parameters**

lpDirName:
[In] The name of the directory. The name should be within 252 characters. A relative path, such as *dirname1* or *dirname1\dirname2*, must be used.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_PARAM: indicates parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: indicates the file name is too long.
QL_RET_ERR_FILEFAILED: indicates it fails to operate the file.

QL_RET_ERR_FILE_NO_CARD: indicates there is no SD card.

5.6.2.16. Ql_FS_CheckDir

This function checks whether the directory exists or not.

- **Prototype**

```
s32 Ql_FS_CheckDir(char *lpDirName)
```

- **Parameters**

lpDirName:
[In] The name of the directory. The name should be within 252 characters. A relative path, such as *dirname1* or *dirname1\dirname2*, must be used.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_PARAM: indicates parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: indicates the file name is too long.
QL_RET_ERR_FILEFAILED: indicates it fails to operate the file.
QL_RET_ERR_FILENOTFOUND: indicates the file is not found.
QL_RET_ERR_FILE_NO_CARD: indicates there is no SD card.

5.6.2.17. Ql_FS_ GetFreeSpace

This function obtains the free space on the flash or SD card.

- **Prototype**

```
s64 Ql_FS_GetFreeSpace (u32 storage)
```

- **Parameters**

storage:
[In] The type of storage, which should be one value of *Enum_FSStorage*.

```
typedef enum
{
    Ql_FS_UFS = 1,
    Ql_FS_SD = 2,
    Ql_FS_RAM = 3,
}Enum_FSStorage;
```

- **Return Value**

The return value is the total number of bytes of the free space in the specified storage if this function succeeds. Otherwise, the return value is an error code.

Ql_RET_ERR_UNKNOWN: indicates unknown error.
QL_RET_ERR_FS_FATAL_ERR1: indicates occurrence of some fatal errors.

5.6.2.18. Ql_FS_GetTotalSpace

This function obtains the total space on the flash or SD card.

- **Prototype**

```
s64 Ql_FS_GetTotalSpace(u32 storage)
```

- **Parameters**

storage:
[In] The type of storage, which should be one value of *Enum_FSStorage*.

```
typedef enum
{
    Ql_FS_UFS = 1,
    Ql_FS_SD = 2,
    Ql_FS_RAM = 3,
}Enum_FSStorage;
```

- **Return Value**

The return value is the total number of bytes in the specified storage if this function succeeds. Otherwise, the return value is an error code.

Ql_RET_ERR_UNKNOWN: indicates unknown error.
QL_RET_ERR_FS_FATAL_ERR1: indicates occurrence of some fatal errors.

5.6.2.19. Ql_FS_Format

This function format the UFS.

- **Prototype**

```
s32 Ql_FS_Format(u8 storage)
```

- **Parameters**

storage:
[In] The type of storage, which should be one value of *Enum_FSStorage*.

- **Return Value**

- *QL_RET_OK*: indicates this function is executed successfully.
- *QL_RET_ERR_PARAM*: indicates parameter error.
- *QL_RET_ERR_FILENAMETOOLENGTH*: indicates the file name is too long.
- *QL_RET_ERR_FILENOTFOUND*: indicates the file is not found.
- *QL_RET_ERR_PATHNOTFOUND*: indicates the path is not found.
- *QL_RET_ERR_GET_MEM*: indicates failed to get memory.
- *QL_RET_ERR_GENERAL_FAILURE*: indicates general failure.

5.6.2.20. Ql_FS_Safe_Write

This function can create a new file or writer write a file securely. If there is no file, it will create a file and also a backup automatically. It is more secure to create files than using Ql_FS_Open and write files than using Ql_FS_Write, but it will frequently erase the file system and increase the load.

- **Prototype**

```
s32 Ql_FS_Safe_Write(u8 *filename, u8 *writeBuffer, u32 numberOfBytesToWrite, u32 *numberOfBytesWritten)
```

- **Parameters**

filename:
[In] The name of the file to be created or written. The name is limited to 252 characters. An absolute path such as *ufs/filename.text* in UFS space must be used.

writeBuffer:
[In] Point to the buffer containing the data to be written to the file.

numberOfBytesToWrite:
[In] Number of bytes to be write to the file.

numberOfBytesWritten:
[Out] The number of bytes has been written.

- **Return Value**

- *QL_RET_OK*: indicates this function is executed successfully.
- *QL_RET_ERR_PARAM*: indicates parameter error.
- *QL_RET_ERR_FILEWRITEFAILED*: failed to write the file.

5.6.2.21. Ql_FS_Safe_Read

This function reads the file created by Ql_FS_Safe_Write.

- **Prototype**

```
s32 Ql_FS_Safe_Read(u8 *filename, u8 *readBuffer, u32 numberOfBytesToRead, u32 *numberOfBytesRead)
```

- **Parameters**

filename:
[In] The name of the file to be read. The name is limited to 252 characters. An absolute path such as *ufs/filename.text* in UFS space must be used.

readBuffer:
[In] Point to the buffer that receives the data read from the file.

numberOfBytesToRead:
[In] Number of bytes to be read from the file.

numberOfBytesRead:
[Out] The number of bytes has been read.

- **Return Value**

- *QL_RET_OK*: indicates this function is executed successfully.
- *QL_RET_ERR_PARAM*: read file failed.
- *QL_RET_ERR_FILEWRITEFAILED, QL_RET_ERR_FILEREADFAILED*: failed to read the file or the file does not exist.

5.6.3. Example

The following codes show how to use the file system.

```
#define MEMORY_TYPE     1
#define FILE_NAME       "test.txt"
#define NEW_FILE_NAME   "file.txt"
#define DIR_NAME        "DIR\\"
#define LPPATH          "\\*"
#define LPPATH2         "\\DIR\\*"
#define XDELETE_PATH    "\\"
#define WRITE_DATA      "1234567890"
#define OFFSET          0
```

```
void API_TEST_File(void)
{
    s32 ret;
    s64 size;
    s32 filehandle, findfile;
    u32 writeedlen, readedlen ;
    u8 strBuf[100];
    s32 position;
    s32 filesize;
    bool isdir;

    //Get the amount of free space on flash or SD card.
    size=Ql_FS_GetFreeSpace(MEMORY_TYPE);
    Ql_Debug_Trace("Ql_FS_GetFreeSpace()=%lld,type =%d\r\n",size,MEMORY_TYPE);

    //Get the amount of total space on flash or SD card.
    size=Ql_FS_GetTotalSpace(MEMORY_TYPE);
    Ql_Debug_Trace("Ql_FS_GetTotalSpace()=%lld,type =%d\r\n",size,MEMORY_TYPE);

    //Format the UFS.
    ret=Ql_FS_Format(MEMORY_TYPE);
    Ql_Debug_Trace("Ql_FS_Format()=%d    type =%d\r\n",ret,MEMORY_TYPE);

    //Create a file test.txt.
    ret=Ql_FS_Open(FILE_NAME, QL_FS_READ_WRITE|QL_FS_CREATE);
    if(ret >= QL_RET_OK)
    {
        filehandle = ret;
    }
    Ql_Debug_Trace("Ql_FS_OpenCreate(%s,%08x)=%d\r\n",FILE_NAME,
    QL_FS_READ_WRITE|QL_FS_CREATE, ret);

    //Write "1234567890" to file.
    ret=Ql_FS_Write(filehandle, WRITE_DATA, Ql_strlen(WRITE_DATA), &writeedlen);
    Ql_Debug_Trace("Ql_FS_Write()=%d: writeedlen=%d\r\n",ret, writeedlen);

    //Write data remaining in the file buffer to the file.
    Ql_FS_Flush(filehandle);

    //Move the file pointer to the starting position.
    ret=Ql_FS_Seek(filehandle, OFFSET , QL_FS_FILE_BEGIN);
    Ql_Debug_Trace("Ql_FS_Seek()=%d: offset=%d\r\n",ret, OFFSET);

    //Read data from file.
```

```
Ql_memset(strBuf,0,100);
ret = Ql_FS_Read(filehandle, strBuf, 100, &readedlen);
Ql_Debug_Trace("Ql_FS_Read()=%d: readedlen=%d, strBuf=%s\r\n",ret, readedlen, strBuf);

//Move the file pointer to the starting position.
ret=Ql_FS_Seek(filehandle, OFFSET , QL_FS_FILE_BEGIN);
Ql_Debug_Trace("Ql_FS_Seek()=%d: offset=%d\r\n",ret, OFFSET);

//Truncate the file to zero length.
ret=Ql_FS_Truncate(filehandle);
Ql_Debug_Trace("Ql_FS_Truncate()=%d\r\n",ret);

//Read data from file.
Ql_memset(strBuf,0,100);
ret=Ql_FS_Read(filehandle, strBuf, 100, &readedlen);
Ql_Debug_Trace("Ql_FS_Read()=%d: readedlen=%d, strBuf=%s\r\n",ret, readedlen, strBuf);

//Get the position of the file pointer.
Position=Ql_FS_GetFilePosition(filehandle);
Ql_Debug_Trace("Ql_FS_GetFilePosition(): Position=%d\r\n",Position);

//Close the file.
Ql_FS_Close(filehandle);
filehandle=-1;
Ql_Debug_Trace("Ql_FS_Close()\r\n");

//Get the size of the file.
filesize=Ql_FS_GetSize(FILE_NAME);
Ql_Debug_Trace((char*)("Ql_FS_GetSize(%s), filesize=%d\r\n"), FILE_NAME, filesize);

//Check whether the file exists or not.
ret=Ql_FS_Check(FILE_NAME);
Ql_Debug_Trace("Ql_FS_Check(%s)=%d\r\n", FILE_NAME, ret);

//Rename the file name from "test.txt" to "file.txt".
ret=Ql_FS_Rename(FILE_NAME, NEW_FILE_NAME);
Ql_Debug_Trace("Ql_FS_Rename(\"%s\",\"%s\")=%d\r\n", FILE_NAME, NEW_FILE_NAME, ret);

//Delete the file file.txt.
ret=Ql_FS_Delete(NEW_FILE_NAME);
Ql_Debug_Trace("Ql_FS_Delete(%s)=%d\r\n", NEW_FILE_NAME, ret);

//Create a file test.txt.
ret=Ql_FS_Open(FILE_NAME, QL_FS_READ_WRITE|QL_FS_CREATE);
```

```
    if(ret >=QL_RET_OK)
    {
      filehandle=ret;
    }
    Ql_Debug_Trace("Ql_FS_Open Create (%s,%08x)=%d\r\n", FILE_NAME,
    QL_FS_READ_WRITE|QL_FS_CREATE, ret);

    //Write "1234567890" to file.
    ret=Ql_FS_Write(filehandle, WRITE_DATA, Ql_strlen(WRITE_DATA), &writeedlen);
    Ql_Debug_Trace("Ql_FS_Write()=%d: writeedlen=%d\r\n",ret, writeedlen);

    //Close the file.
    Ql_FS_Close(filehandle);
    filehandle=-1;
    Ql_Debug_Trace("Ql_FS_Close()\r\n");

    //Create a directory.
    ret=Ql_FS_CreateDir(DIR_NAME);
    Ql_Debug_Trace("Ql_FS_CreateDir(%s)=%d\r\n", DIR_NAME, ret);

    //Check whether the directory exists or not.
    ret=Ql_FS_CheckDir(DIR_NAME);
    Ql_Debug_Trace("Ql_FS_CheckDir(%s)=%d\r\n", DIR_NAME, ret);

    //Delete the directory.
    ret=Ql_FS_DeleteDir(DIR_NAME);
    Ql_Debug_Trace("Ql_FS_DeleteDir(%s)=%d\r\n", DIR_NAME, ret);

    //Create a directory.
    ret=Ql_FS_CreateDir(DIR_NAME);
    Ql_Debug_Trace("Ql_FS_CreateDir(%s)=%d\r\n", DIR_NAME, ret);

}
```

5.7. Hardware Interface APIs

5.7.1. UART

5.7.1.1. UART Overview

In QuecOpen, UART ports include physical and virtual UART ports. The physical UART ports can be connected to external devices, and the virtual UART ports are used for communication between an application and the bottom operating system.

One of the physical UART ports (UART1) has hardware handshaking function, and others are three-wire interfaces.

QuecOpen provides two virtual UART ports that are used for communication between the App and the core. Virtual ports are designed according to the features of physical port. They have their RI and DCD information. The level of DCD can be used to indicate the virtual port is in data mode or AT command mode.

The working chart of UARTs is shown below:

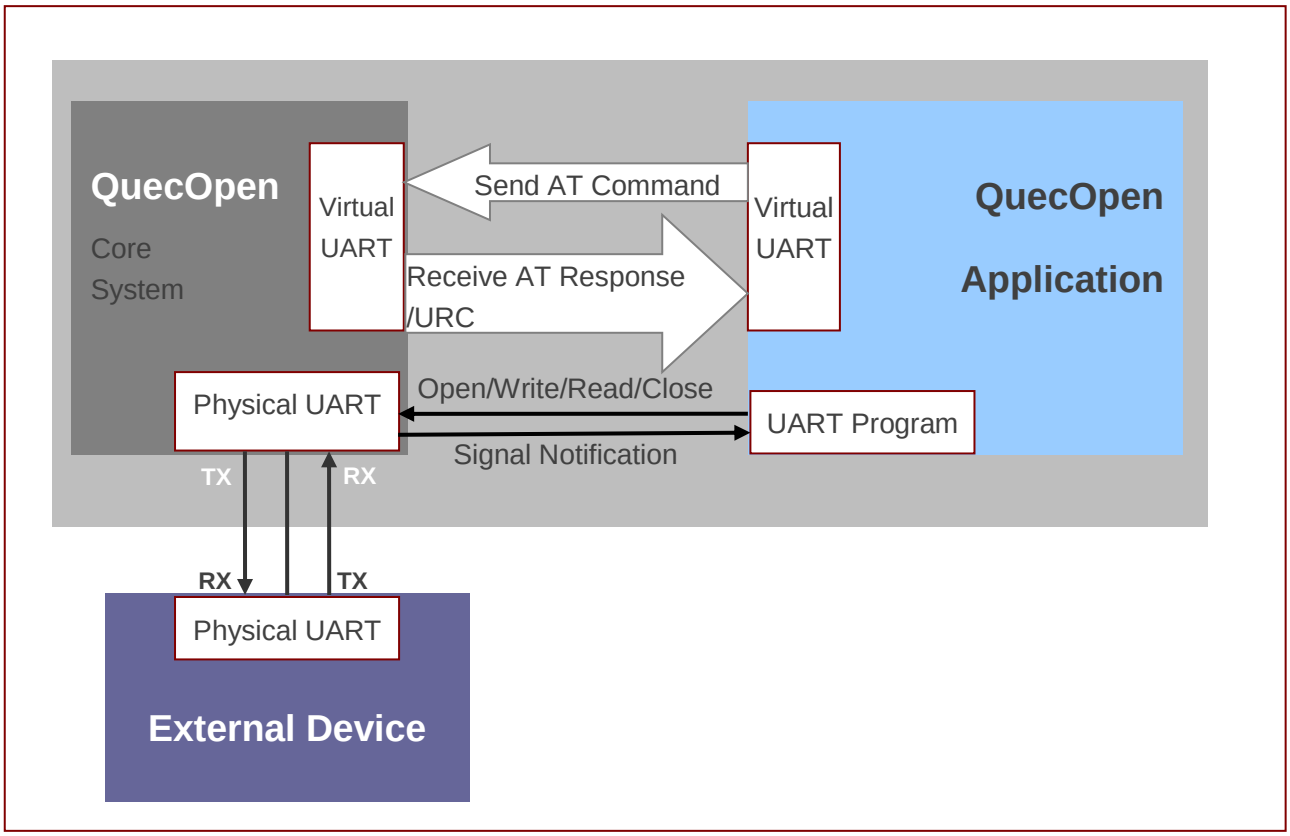


Figure 2: The Working Chart of UARTs

5.7.1.2. UART Usage

The following steps can be applied as for physical/virtual UART initialization and usage.

Step 1: Call *QI_UART_Register* to register callback function of the UART.
Step 2: Call *QI_UART_Open* to open a specified UART port.
Step 3: Call *QI_UART_Write* to write data to a specified UART port. When the number of bytes actually sent is less than that to be sent, the application should stop sending data and will receive an event EVENT_UART_READY_TO_WRITE later in callback function. After receiving this event, the application can continue to send data, and the previously unsent data should be resent.

Step 4: Deal with the UART's notification in the callback function. If the notification type is EVENT_UART_READY_TO_READ, please read out all data in the UART RX buffer. Otherwise, there will not be such notification to be reported to application when new data comes to UART RX buffer later.

5.7.1.3. API Functions

5.7.1.3.1. Ql_UART_Register

This function registers the callback function for a specified UART port. UART callback function is used to receive the UART notification from core system.

- **Prototype**

```
s32 Ql_UART_Register(Enum_SerialPort port, CallBack_UART_Notify callback_uart,void *
customizePara)
typedef void (*CallBack_UART_Notify)( Enum_SerialPort port, Enum_UARTEventType event, bool
pinLevel,void *customizePara)
```

- **Parameters**

port:
[In] Port name.

callback_uart:
[In] Pointer of the UART callback function.

event:
[Out] Indication of the event type of UART callback. One value of *Enum_UARTEventType*.

pinLevel:
[Out] If the event type is EVENT_UART_RI_IND, EVENT_UART_DCD_IND or EVENT_UART_DTR_IND, the pinLevel indicates the related pin's current level. Otherwise this parameter is meaningless and can be ignored.
customizePara:
[In] Customized parameter. If not used, just set it to NULL.

- **Return Value**

The return value is *QL_RET_OK* if this function succeeds. Otherwise, the return value is an error code. To get extended error information, please refer to the error codes in header file *ql_error.h*.

5.7.1.3.2. Ql_UART_Open

This function opens a specified UART port with the specified flow control mode. The task that calls this function will own the specified UART port.

- **Prototype**

```
s32 Ql_UART_Open(Enum_SerialPort port,u32 baudrate, Enum_FlowCtrl flowCtrl)
```

- **Parameters**

port:
[In] Port name.

baudrate:
[In] The baud rates of the UART to be opened.
The physical UART supports baud rates as follows: 300bps, 600bps, 1200bps, 2400bps, 4800bps, 9600bps, 14400bps, 19200bps, 28800bps, 38400bps, 57600bps and 115200bps. This parameter does not take effect for VIRTUAL_PORT1 and VIRTUAL_PORT2, so just set it to 0.

flowCtrl:
[In] See *Enum_flowCtrl* below for the physical UART ports. Only UART1 supports hardware flow control.

```
typedef enum {
    FC_NONE=1,      //Flow control closed
    FC_HW,          //Hardware flow control
    FC_SW           //Software flow control
} Enum_FlowCtrl;
```

- **Return Value**

The return value is *QL_RET_OK* if this function succeeds. Otherwise, the return value is an error code. To get extended error information, please refer to the ERROR CODES in header file *ql_error.h*.

5.7.1.3.3. Ql_UART_OpenEx

This function opens a specified UART port with the specified DCB parameters. The task that calls this function will own the specified UART port.

- **Prototype**

```
s32 Ql_UART_OpenEx(Enum_SerialPort port, ST_UARTDCB *dcb)
```

- **Parameters**

port:
[In] Port name.

dcb:
[In] Pointer to the UART DCB settings, including baud rates, data bits, stop bits, parity, and flow control. Only physical serial port1 (UART1) supports hardware flow control. This parameter does not take effect for VIRTUAL_PORT1 and VIRTUAL_PORT2, so just set it to NULL.

- **Return Value**

The return value is *QL_RET_OK* if this function succeeds. Otherwise, the return value is an error *code*. To get extended error information, please refer to the error codes in header file *ql_error.h*.

5.7.1.3.4. Ql_UART_Write

This function is used to send data to a specified UART port. When the number of bytes actually sent is less than that to be sent, the application should stop sending data, and then it (in callback function) will receive an event EVENT_UART_READY_TO_WRITE later. After receiving this event, application can continue to send data, and the previously unsent data should be resent.

- **Prototype**

```
s32 Ql_UART_Write(Enum_SerialPort port, u8* data, u32 writeLen)
```

- **Parameters**

port:
[In] Port name.

data:
[In] Pointer to data to write.

writeLen:
[In] The length of the data to write. For VIRTUAL_UART1 and VIRTUAL_UART2, the maximum length that can be written at one time is 1024 bytes which cannot be modified programmatically in application.

- **Return Value**

Actual number of bytes written. If this function fails to write data, a negative number will be returned. To get extended error information, please refer to the error codes in header file *ql_error.h*.

5.7.1.3.5. Ql_UART_Read

This function reads data from a specified UART port. When the UART callback is invoked, and the notification is EVENT_UART_READY_TO_READ, please read out all data in the UART RX buffer by calling this function in loop; otherwise, there will not be such notification to be reported to the application when new data comes to UART RX buffer later.

- **Prototype**

```
s32 Ql_UART_Read(Enum_SerialPort port, u8* data, u32 readLen)
```

- **Parameters**

port:
[In] Port name.

data:
[In] Pointer to the buffer for the read data.

readLen:
[In] The length of the data to be read. The maximum data length of the receive buffer for physical UART buffer is 3584 bytes, and 1024 bytes for virtual UART. The buffer size cannot be modified programmatically in applications.

- **Return Value**

Actual number of read bytes. If *readLen* equals to the actual read length, please continue reading the UART until the actual read length is less than the *readLen*. To get extended error information, please refer to the error codes in header file *ql_error.h*.

5.7.1.3.6. Ql_UART_SetDCBConfig

This function sets the parameters of a specified UART port and works only for physical UART ports.

- **Prototype**

```
s32 Ql_UART_SetDCBConfig(Enum_SerialPort port, ST_UARTDCB *dcb)
```

The enumerations for DCB are defined as follows.

```
typedef enum {
    DB_5BIT = 5,
    DB_6BIT,
    DB_7BIT,
```

```
    DB_8BIT
} Enum_DataBits;

typedef enum {
    SB_ONE=1,
    SB_TWO,
    SB_ONE_DOT_FIVE
} Enum_StopBits;

typedef enum {
    PB_NONE=0,
    PB_ODD,
    PB_EVEN,
    PB_SPACE,
    PB_MARK
} Enum_ParityBits;

typedef enum {
    FC_NONE=1,      //Flow control closed
    FC_HW,          //Hardware flow control
    FC_SW           //Software flow control
} Enum_FlowCtrl;

typedef struct {
    u32                 baudrate;
    Enum_DataBits       dataBits;
    Enum_StopBits       stopBits;
    Enum_ParityBits     parity;
    Enum_FlowCtrl       flowCtrl;
}ST_UARTDCB;
```

- **Parameter**

port:
[In] Port name.

dcb:
[In] Pointer to the UART DCB Structure, which includes baud rates, data bits, stop bits and parity.

- **Return Value**

The return value is *QL_RET_OK* if this function succeeds. Otherwise, the return value is an error code. To get extended error information, please refer to the error codes in header file *ql_error.h*.

5.7.1.3.7. Ql_UART_GetDCBConfig

This function gets the configuration parameters of a specified UART port and works only for physical UART ports.

- **Prototype**

```
s32 Ql_UART_GetDCBConfig(Enum_SerialPort port, ST_UARTDCB *dcb)
```

- **Parameters**

port:
[In] Port name.

dcb:
[In] Current DCB configuration parameters of the specified UART port including baud rates, data bits, stop bits and parity.

- **Return Value**

The return value is *QL_RET_OK* if this function succeeds. Otherwise, the return value is an error code. To get extended error information, please refer to the error codes in header file *ql_error.h*.

5.7.1.3.8. Ql_UART_ClrRxBuffer

This function clears the receive buffer of a specified UART port.

- **Prototype**

```
void Ql_UART_ClrRxBuffer(Enum_SerialPort port)
```

- **Parameters**

port:
[In] Port name.

- **Return Value**

None.

5.7.1.3.9. Ql_UART_ClrTxBuffer

This function clears the send buffer of a specified UART port.

- **Prototype**

```
void Ql_UART_ClrTxBuffer(Enum_SerialPort port)
```

- **Parameters**

port:
[In] Port name.

- **Return Value**

None.

5.7.1.3.10.Ql_UART_GetPinStatus

This function gets the status indication pins (including RI, DCD and DTR) of the virtual UART port and does not work for the physical UART ports.

- **Prototype**

```
s32 Ql_UART_GetPinStatus(Enum_SerialPort port, Enum_UARTPinType pin)
```

The enumerations for UART pin types are defined as follows.

```
typedef    enum {
UART_PIN_RI=0,            //RI read operator is only valid on the virtual UART.
                          //RI set operator is invalid both on virtual and physical UARTs.
UART_PIN_DCD,             //DCD read operator is only valid on the virtual UART.
                          //DCD set operatir is invalid both on virtual and physical UARTs.
} Enum_UARTPinType;
```

- **Parameters**

port:
[In] Virtual UART port name.

pin:
[In] Pin name. One value of *Enum_UARTPinType*.

- **Return Value**

If the return value ≥ 0, it indicates the function is executed successfully, and a special pin level value is returned: 0 indicates low level, and 1 indicates high level. If the return value < 0, it indicates it fails to execute the function.

5.7.1.3.11.Ql_UART_SetPinStatus

This function sets the pin level status of the virtual UART port. It does not work for the physical UART ports.

- **Prototype**

```
s32 Ql_UART_SetPinStatus(Enum_SerialPort port, Enum_UARTPinType pin, bool pinLevel)
```

- **Parameters**

port:
[In] Virtual UART port name.

pin:
[In] Pin name. One value of *Enum_UARTPinType*.

pinLevel:
[In] The pin level to be set. 0 indicates low level and 1 indicates high level.

- **Return Value**

The return value is *QL_RET_OK* if this function succeeds. Otherwise, the return value is an error code. To get extended error information, please refer to the error codes in header file *ql_error.h*.

5.7.1.3.12.Ql_UART_SendEscap

This function notifies the virtual serial port to exit from data mode and return back to command mode. This function works only for virtual ports.

- **Prototype**

```
s32 Ql_UART_SendEscap (Enum_SerialPort port)
```

- **Parameters**

port:
[In] Port name.

- **Return Value**

The return value is *QL_RET_OK* if this function succeeds. Otherwise, the return value is an error code. To get extended error information, please refer to the error codes in header file *ql_error.h*.

5.7.1.3.13.Ql_UART_Close

This function closes the specified UART port.

- **Prototype**

```
void Ql_UART_Close(Enum_SerialPort port)
```

- **Parameters**

port:
[In] Port name.

- **Return Value**

None.

5.7.1.4. Example

This chapter gives the example of how to use the UART port APIs.

```
//Write the callback function for dealing with the UART notifications.
static void CallBack_UART_Hdlr(Enum_SerialPort port, Enum_UARTEventType msg, bool level, void*
customizedPara)                 //Callback function.
{
 switch(msg)
 case EVENT_UART_READ_TO_READ：
       //Read data from the UART port.
       Ql_UART_Read (port,buffer,rlen);
       break;
 case EVENT_UART_READ_TO_WRITE:
       //Resume the operation of writing data to UART.
       QL_UART_Write(port,buffer,wlen);
       break;
 case EVENT_UART_RI_CHANGE:
       break;
 case EVENT_UART_DCD_CHANGE
       break;
 case EVENT_UART_DTR_CHANGE:
       break;
 case EVENT_UART_FE_IND:
       break;
 default:
       break;
}
```

```
//Register the callback function.
Ql_UART_Register(UART_PORT1, CallBack_UART_Hdlr,NULL);
//Open the specified UART port
Ql_UART_Open(UART_PORT1);
//Write data to UART port
QL_UART_Write(UART_PORT1,buffer,len);
```

5.7.2. GPIO

5.7.2.1. GPIO Overview

There are 13 I/O pins that can be designed as general purpose I/Os. All these pins can be accessed by corresponding API functions.

5.7.2.2. GPIO List

Table 6: Multiplexing Pins

Pin name	Pin No.	Mode 1	Mode 2	Mode 3	Mode 4
PINNAME_NETLIGHT	16	NETLIGHT	GPIO	EINT	
PINNAME_DTR	19	DTR	GPIO	EINT	
PINNAME_RI	20	RI	GPIO	I2C2_SCL	EINT
PINNAME_DCD	21	DCD	GPIO	I2C2_SDA	EINT
PINNAME_CTS	22	CTS	GPIO	EINT	
PINNAME_RTS	23	RTS	GPIO	EINT	
PINNAME_RFTXMON	25	RFTXMON	GPIO	EINT	
PINNAME_RXD_AUX	28	RXD_AUX	GPIO	EINT	
PINNAME_TXD_AUX	29	TXD_AUX	GPIO	EINT	
PINNAME_PCM_CLK	30	PCM_CLK	GPIO	EINT	
PINNAME_PCM_SYNC	31	PCM_SYNC	GPIO	EINT	
PINNAME_PCM_IN	32	PCM_IN	GPIO	EINT	
PINNAME_PCM_OUT	33	PCM_OUT	GPIO	EINT	

NOTES

1. "MODE1" defines the original status of pin in standard module.
2. "EINT" means external interrupt input.

5.7.2.3. GPIO Initial Configuration

In QuecOpen, GPIO-related APIs can be called to initialize GPIOs after App starts.

5.7.2.4. GPIO Usage

The following steps show how to use the multifunctional GPIOs:

Step 1: GPIO initialization. Call *Ql_GPIO_Init* function, and set a specified pin as the GPIO function and initialize configurations such as direction, level and pull selection.

Step 2: GPIO control. When the pin is initialized as a GPIO, GPIO-related APIs can be called to change the GPIO level.

Step 3: Release the pin. If this pin is intended to be used for other purposes (such as EINT), please call *Ql_GPIO_Uninit* to release the pin first. This step is optional.

5.7.2.5. API Functions

5.7.2.5.1. Ql_GPIO_Init

This function enables the GPIO function of a specified pin and initializes configurations such as direction, level and pull selection.

- **Prototype**

```
s32 Ql_GPIO_Init(Enum_PinName pinName, Enum_PinDirection dir, Enum_PinLevel level, Enum_PinPullSel pullSel)
```

- **Parameters**

pinName:
[In] Pin name. One value of *Enum_PinName*.

dir:
[In] The initial direction of GPIO. One value of *Enum_PinDirection*.

level:
[In] The initial level of GPIO. One value of *Enum_PinLevel*.

pullSel:
[In] Pull selection. One value of *Enum_PinPullSel*.

- **Return Value**

QL_RET_OK indicates this function is executed successfully. Negative integer indicates this function fails.

5.7.2.5.2. Ql_GPIO_GetLevel

This function gets the level of a specified GPIO.

- **Prototype**

```
s32 Ql_GPIO_GetLevel(Enum_PinName pinName)
```

- **Parameters**

pinName:
[In] Pin name. One value of *Enum_PinName*.

- **Return Value**

Return the level of the specified GPIO. 1 indicates high level, and 0 indicates low level.

5.7.2.5.3. Ql_GPIO_SetLevel

This function sets the level of a specified GPIO.

- **Prototype**

```
s32 Ql_GPIO_SetLevel(Enum_PinName pinName, Enum_PinLevel level)
```

- **Parameters**

pinName:
[In] Pin name. One value of *Enum_PinName*.

level:
[In] The initial level of GPIO. One value of *Enum_PinLevel*.

- **Return Value**

QL_RET_OK indicates this function is executed successfully. Negative integer indicates this function fails.

5.7.2.5.4. Ql_GPIO_GetDirection

This function gets the direction of a specified GPIO.

- **Prototype**

```
s32 Ql_GPIO_GetDirection(Enum_PinName pinName)
```

- **Parameters**

pinName:
[In] Pin name. One value of *Enum_PinName*.

- **Return Value**

Return the direction of the specified GPIO. 1 indicates output and 0 indicates input.

5.7.2.5.5. Ql_GPIO_SetDirection

This function sets the direction of a specified GPIO.

- **Prototype**

```
s32 Ql_GPIO_SetDirection(Enum_PinName pinName, Enum_PinDirection dir)
```

- **Parameters**

pinName:
[In] Pin name. One value of *Enum_PinName*.

dir:
[In] The initial direction of GPIO. One value of *Enum_PinDirection*.

- **Return Value**

QL_RET_OK indicates this function is executed successfully. Negative integer indicates this function fails.

5.7.2.5.6. Ql_GPIO_Uninit

This function releases a specified GPIO that has been initialized by calling *Ql_GPIO_Init* previously. After releasing, the GPIO can be used for other purposes.

- **Prototype**

```
s32 Ql_GPIO_Uninit(Enum_PinName pinName)
```

- **Parameters**

pinName:
[In] Pin name. One value of *Enum_PinName*.

- **Return Value**

QL_RET_OK indicates this function is executed successfully. Negative integer indicates this function fails.

5.7.2.5.7. Ql_GPIO_SetPullSelection*

This function sets the pull selection of a specified GPIO.

- **Prototype**

```
s32 Ql_GPIO_SetPullSelection(Enum_PinName pinName,Enum_PinPullSel pullSel)
```

- **Parameters**

pinName:
[in] Pin name. One value of *Enum_PinName*.

pullSel:
[in] Pull selection. One value of *Enum_PinPullSel*.

- **Return Value**

QL_RET_OK indicates this function is executed successfully. Negative integer indicates this function fails.

5.7.2.6. Example

This chapter gives the example of how to use the GPIO APIs.

```
void API_TEST_gpio(void)
{
    s32 ret;
    Ql_Debug_Trace("\r\n<**********    GPIO API Test    **********>\r\n");

    ret=Ql_GPIO_Init(PINNAME_NETLIGHT, PINDIRECTION_OUT, PINLEVEL_HIGH,
```

```
PINPULLSEL_PULLUP);
    Ql_Debug_Trace("\r\n<--pin(%d) Ql_GPIO_Init ret=%d-->\r\n",PINNAME_NETLIGHT,ret);

    ret=Ql_GPIO_SetLevel(PINNAME_NETLIGHT,PINLEVEL_HIGH);
    Ql_Debug_Trace("\r\n<--pin(%d) Ql_GPIO_SetLevel =%d ret=%d-->\r\n",
                                PINNAME_NETLIGHT,PINLEVEL_HIGH,ret);

    ret=Ql_GPIO_SetDirection(PINNAME_NETLIGHT,PINDIRECTION_IN);
    Ql_Debug_Trace("\r\n<--pin(%d) Ql_GPIO_SetDirection =%d ret=%d-->\r\n",
                               PINNAME_NETLIGHT,PINDIRECTION_IN,ret);

    ret=Ql_GPIO_GetLevel(PINNAME_NETLIGHT);
   Ql_Debug_Trace("\r\n<--pin(%d) Ql_GPIO_GetLevel =%d ret=%d-->\r\n",
                              PINNAME_NETLIGHT,ret,ret);

    ret=Ql_GPIO_GetDirection(PINNAME_NETLIGHT);
    Ql_Debug_Trace("\r\n<--pin(%d) Ql_GPIO_GetDirection =%d ret=%d-->\r\n",
                                PINNAME_NETLIGHT,ret,ret);

    ret=Ql_GPIO_SetPullSelection(PINNAME_NETLIGHT,PINPULLSEL_PULLDOWN);
    Ql_Debug_Trace("\r\n<--pin(%d) Ql_GPIO_SetPullSelection =%d ret=%d-->\r\n",
                               PINNAME_NETLIGHT,PINPULLSEL_PULLDOWN,ret);

    ret=Ql_GPIO_Uninit(PINNAME_NETLIGHT);
    Ql_Debug_Trace("\r\n<--pin(%d) Ql_GPIO_Uninit ret=%d-->\r\n",PINNAME_NETLIGHT,ret);
}
```

5.7.3. EINT

5.7.3.1. EINT Overview

QuecOpen module has 13 external interrupt pins, and please refer to ***Chapter 5.7.2.2*** for details. The interrupt trigger mode only supports edge-triggered mode. External interrupt enjoys higher priority, so frequent interrupt is not allowed. It is strongly recommended that the interrupt frequency should be no more than 2, and too frequent interrupt will prevent other tasks from being scheduled, which probably leads to unwanted exception.

NOTE

The interrupt response time is 100ms by default, and can be re-programmed to a greater value in QuecOpen. However, it is strongly recommended that the interrupt frequency should not be more than 3Hz so as to ensure stable working of the module.

5.7.3.2. EINT Usage

The following steps show how to use the external interrupt function:

Step 1: Register an external interrupt function. Please choose one external interrupt pin and use *Ql_EINT_Register* (or *Ql_EINT_RegisterFast*) to register an interrupt handler function.
Step 2: Initialize the interrupt configurations. Call *Ql_EINT_Init* function to configure the software debounce time and set the edge-triggered interrupt mode.
Step 3: Interrupt handle. The interrupt callback function will be called if the level has changed. It can also be processed in the handler.
Step 4: Mask the interrupt. When external interrupt is not needed, please call *Ql_EINT_Mask* function to disable it. When it is needed afterwards, please call *Ql_EINT_Unmask* function to enable it again.
Step 5: Release the specified EINT pin. Call *Ql_EINT_Uninit* function to release the specified EINT pin, and the pin can be used for other purposes after it is released. This step is optional.

5.7.3.3. API Functions

5.7.3.3.1. Ql_EINT_Register

This function registers an EINT I/O, and specifies the interrupt handler.

- **Prototype**

```
s32 Ql_EINT_Register(Enum_PinName eintPinName, Callback_EINT_Handle callback_eint,void* customParam)
typedef void (*Callback_EINT_Handle)( Enum_PinName eintPinName, Enum_PinLevel pinLevel, void* customParam)
```

- **Parameters**

eintPinName:
[In] EINT pin name. One value of *Enum_PinName* that has the interrupt function.

callback_eint:
[In] The interrupt handler.

pinLevel:
[In] The EINT pin level value. One value of *Enum_PinLevel*.

customParam:
[In] Customized parameter. If not used, just set it to NULL.

- **Return Value**

QL_RET_OK indicates this function is executed successfully. Negative integer indicates this function fails.

5.7.3.3.2. Ql_EINT_RegisterFast

This function registers an EINT I/O, and specifies the interrupt handler. The EINT that is registered by calling this function is a top half interrupt. The response to interrupt request is timelier. Please do not add any task schedule in the interrupt handler which cannot consume much CPU time. Otherwise it may lead to system exception or resetting.

- **Prototype**

```
s32 Ql_EINT_RegisterFast(Enum_PinName eintPinName, Callback_EINT_Handle callback_eint, void* customParam)
```

- **Parameters**

eintPinName:
[In] EINT pin name. One value of *Enum_PinName* that has the interrupt function.

callback_eint:
[In] The interrupt handler.

customParam:
[In] Customized parameter. If not used, just set it to NULL.

- **Return Value**

QL_RET_OK indicates this function is executed successfully. Negative integer indicates this function fails.

5.7.3.3.3. Ql_EINT_Init

This function initializes an external interrupt function.

- **Prototype**

```
s32 Ql_EINT_Init(Enum_PinName eintPinName,Enum_EintType eintType, bool automask)
```

- **Parameters**

eintPinName:
[In] EINT pin name. One value of *Enum_PinName* that has the interrupt function.

eintType:
[In] Interrupt type. Only edge-triggered interrupts are supported.
EINT_EDGE_FALLING_AND_RISING: Both the rising and falling edge triggers.
EINT_EDGE_RISING: Rising edge trigger.
EINT_EDGE_FALLING: Falling edge trigger.

autoMask:
[In] Whether automatically mask the external interrupt after the interrupt happens. 0 indicates no, and 1 indicates yes.

- **Return Value**

QL_RET_OK indicates this function is executed successfully. Negative integer indicates this function fails.

5.7.3.3.4. Ql_EINT_Uninit

This function releases a specified EINT pin.

- **Prototype**

```
s32 Ql_EINT_Uninit(Enum_PinName eintPinName)
```

- **Parameters**

eintPinName:
[In] EINT pin name.

- **Return Value**

QL_RET_OK indicates this function is executed successfully. Negative integer indicates this function fails.

5.7.3.3.5. Ql_EINT_GetLevel

This function gets the level of a specified EINT pin.

- **Prototype**

```
s32 Ql_EINT_GetLevel(Enum_PinName eintPinName)
```

- **Parameters**

eintPinName:
[In] EINT pin name.

- **Return Value**

1 indicates high level, and 0 indicates low level.

5.7.3.3.6. Ql_EINT_Mask

This function masks a specified EINT pin.

- **Prototype**

```
void Ql_EINT_Mask(Enum_PinName eintPinName)
```

- **Parameters**

eintPinName:
[In] EINT pin name.

- **Return Value**

None.

5.7.3.3.7. Ql_EINT_Unmask

This function unmasks a specified EINT pin.

- **Prototype**

```
void Ql_EINT_Unmask(Enum_PinName eintPinName)
```

- **Parameters**

eintPinName:
[In] EINT pin name.

- **Return Value**

None.

5.7.3.4. Example

The following sample codes show how to use the EINT function.

```
static void callback_eint_handle(Enum_PinName eintPinName, Enum_PinLevel pinLevel, void*
customParam)
{
```

```
    s32 ret;
    //Mask the specified EINT pin.
    Ql_EINT_Mask(eintPinName);

    APP_DEBUG("<--Eint callback: pin(%d), levle(%d)-->\r\n",eintPinName,pinLevel);
    ret = Ql_EINT_GetLevel(eintPinName);
    APP_DEBUG("<--Get Level, pin(%d), levle(%d)-->\r\n",eintPinName,ret);

    //Unmask the specified EINT pin.
    Ql_EINT_Unmask(eintPinName);
}


void API_TEST_eint(void)
{
    s32 ret;

     //Register PINNAME_DCD pin for a top half external interrupt pin.
    ret=Ql_EINT_RegisterFast(PINNAME_DCD,eint_callback_handle,(void *)&EintcustomParam);

    //Initialize some parameters and the auto mask is set to FALSE.
    ret=Ql_EINT_Init(PINNAME_DCD, EINT_EDGE_RISING, 0,5,0);
    Ql_Debug_Trace("\r\n<--pin(%d) Ql_EINT_Init ret=%d-->\r\n", PINNAME_DCD,ret);

    //Register PINNAME_DTR pin for an external interrupt pin.
    ret=Ql_EINT_Register(PINNAME_DTR,eint_callback_handle, (void *)&fastEintcustomParam);

    //Initialize some parameters and the auto mask is set to TRUE.
    ret=Ql_EINT_Init( PINNAME_DTR, EINT_EDGE_RISING, 0, 5,1);
}
```

5.7.4. ADC

5.7.4.1. ADC Overview

QuecOpen module provides an analog input pin that can be used to detect the external voltage. Please refer to *Quectel_M25-QuecOpen_Hardware_Design* for the pin definition and ADC hardware characteristics. The voltage range that can be detected is 0mV~1800mV.

5.7.4.2. ADC Usage

The following steps describe the usage of ADC function:

Step 1: Open ADC channel. Call *Ql_ADC_Open* to open an ADC channel.
Step 2: Read ADC value. *Ql_ADC_Read* can be used to read ADC value from the current pin.
Step 3: Close ADC channel. *Ql_ADC_Close* can be used to close the ADC channel.

5.7.4.3. API Functions

5.7.4.3.1. Ql_ADC_Open

This function is used to open an ADC channel.

- **Prototype**

```
s32 Ql_ADC_Open(Enum_ADCPin adcPin,QL_ADC_Period adcPeriod)
```

- **Parameters**

adcPin:
[In] ADC pin name. One value of *Enum_ADCPin*.

adcPeriod:
[In] ADC period. One value of *QL_ADC_Period*.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_PARAM: the input pin is invalid.
QL_RET_ERR_ERROR: indicates it fails to open the ADC channel.

5.7.4.3.2. Ql_ADC_Read

This function is used to read the ADC value from the current pin.

- **Prototype**

```
s32 Ql_ADC_Read(Enum_ADCPin adcPin,u16 *adcValue)
```

- **Parameters**

adcPin:
[In] ADC pin name. One value of *Enum_ADCPin*.

adcValue:
[out] ADC value. The voltage range that can be detected is 0mV~1800mV.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_PARAM: indicates the input pin is invalid.
QL_RET_ERR_NOT_INIT: indicates ADC value cannot be read, possibly because the ADC channel is not opened.

5.7.4.3.3. Ql_ADC_Close

This function is used to close an ADC channel.

- **Prototype**

```
s32 Ql_ADC_Close(Enum_ADCPin adcPin)
```

- **Parameters**

adcPin:
[In] ADC pin name. One value of *Enum_ADCPin*.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_PARAM: indicates the input pin is invalid.
QL_RET_ERR_ERROR: indicates it fails to close the ADC channel.

5.7.4.4. Example

The following example demonstrates how to use ADC related APIs.

```
Enum_PinName adcPin = PIN_ADC0;
void ADC_Timer_handler(u32 timerId, void* param)
{
        u16 adcvalue = 0;

    *((s32*)param) +=1;
    if(ADC_timer == timerId)
    {
        //Stack timer repeats
        if(*((s32*)param) >= ADC_COUNT)
        {
            Ql_Timer_Stop(ADC_timer);
            APP_DEBUG("<-- ADC closed(%d) -->\r\n",Ql_ADC_Close(adcPin));
        }
        else
```

```
        {
            Ql_ADC_Read(adcPin,&adcvalue);
            APP_DEBUG("<-- read voltage(mV)=%d -->\r\n",adcvalue);
        }
    }
}
void API_TEST_adc(void)
{
    s32 ret;

    //Open the ADC channel.
    ret = Ql_ADC_Open(adcPin,ADC_PERIOD_1MS);
    if(ret < 0)
    {
        APP_DEBUG("\r\n<--failed!!adc open failed-->\r\n",ret);
    }
    APP_DEBUG("\r\n<--adc open successful-->\r\n");

    //Register a timer.
    ret = Ql_Timer_Register(ADC_timer, ADC_Timer_handler, &m_param);
    if(ret <0)
    {
        APP_DEBUG("\r\n<--failed!!, Ql_Timer_Register: timer(%d) fail ,ret = %d
-->\r\n",ADC_timer,ret);
    }
    //Start a timer, repeat=true.
    ret = Ql_Timer_Start(ADC_timer,ADC_time_Interval,TRUE);
    if(ret < 0)
    {
        APP_DEBUG("\r\n<--failed!! stack timer Ql_Timer_Start ret=%d-->\r\n",ret);
    }
}
```

5.7.5. IIC

5.7.5.1. IIC Overview

The module provides a hardware IIC interface. The IIC interface can be simulated by GPIO pins, which can be any two GPIOs in the GPIO list in ***Chapter 5.7.2.2.*** Therefore, one or more IIC interfaces are possible.

5.7.5.2. IIC Usage

The following steps tell how to work with IIC function:

Step 1: Initialize IIC interface. Call *Ql_IIC_Init* function to initialize an IIC channel, including the specified GPIO pins for IIC and an IIC channel number.

Step 2: Configure IIC interface. Call *Ql_IIC_Config* to configure parameters that the slave device needs. Please refer to the API description for extended information.

Step 3: Read data from slave. Call *Ql_IIC_Read* function to read data from the specified slave. The following figure shows the data exchange direction.

Step 4: Write data to slave. Call *Ql_IIC_Write* function to write data to the specified slave. The following figure shows the data exchange direction.

Step 5: Write the data to the register (or the specified address) of the slave. Call *Ql_IIC_Write* function to write the data to a register of the slave. The following figure shows the data exchange direction.

Step 6: Read the data from the register (or the specified address) of the slave. Call *Ql_IIC_Write_Read* function to read the data from a register of the slave. The following figure shows the data exchange direction.

Step 7: Release the IIC channel. Call *Ql_IIC_Uninit* function to release the specified IIC channel.

5.7.5.3. API Functions

5.7.5.3.1. Ql_IIC_Init

This function initializes the configurations for an IIC channel, including the specified pins for IIC, IIC type, and IIC channel number.

- **Prototype**

```
s32 Ql_IIC_Init(u32 chnnlNo,PinName pinSCL,PinName pinSDA, u32 IICtype)
```

- **Parameters**

chnnlNo:
[In] IIC channel number. The range is 0~254.

pinSCL:
[In] IIC SCL pin.

pinSDA:
[In] IIC SDA pin.

IICtype:
[In] IIC type. FALSE indicates simulated IIC, and TRUE indicates hardware IIC.

- **Return Value**

QL_RET_OK indicates this function is executed successfully. Negative integer indicates this function fails.

5.7.5.3.2. Ql_IIC_Config

This function configures the IIC interface for one slave.

- **Prototype**

```
s32 Ql_IIC_Config(u32 chnnlNo, bool isHost, u8 slaveAddr, u32 speed)
```

- **Parameters**

chnnlNo:
[In] IIC channel number. It is specified by *Ql_IIC_Init* function.

isHost:
[In] Whether to use host mode or not. It must be TRUE and only supports host mode.

slaveAddr:
[In] Slave address.

speed:
[In] IIC communication speed. The parameter is just for IIC controller, and can be ignored if simulated IIC is used.

- **Return Value**

QL_RET_OK indicates this function is executed successfully. Negative integer indicates this function fails.

5.7.5.3.3. Ql_IIC_Write

This function writes data to a specified slave through IIC interface.

- **Prototype**

```
s32 Ql_IIC_Write(u32 chnnlNo,u8 slaveAddr,u8 *pData,u32 len)
```

- **Parameters**

chnnlNo:
[In] IIC channel number. It is specified by *Ql_IIC_Init* function.

slaveAddr:
[In] Slave address.

pData:
[In] Setting value to be written to the slave.

len:
[In] Number of bytes to be written. If *IICtype*=1, then 1<*len*<8 because Quectel IIC controller supports 8 bytes at most for transmission at a time.

- **Return Value**

If no error occurs, the length of the written data will be returned. Negative integer indicates this function fails.

5.7.5.3.4. Ql_IIC_Read

This function reads data from a specified slave through IIC interface.

- **Prototype**

```
s32 Ql_IIC_Read(u32 chnnlNo,u8 slaveAddr,u8 *pBuffer,u32 len)
```

- **Parameters**

chnnlNo:
[In] IIC channel number. It is specified by *Ql_IIC_Init* function.

slaveAddr:
[In] Slave address.

pBuffer:
[Out] The buffer that stores the data read from a specific slave.

len:
[Out] Number of bytes to be read. If *IICtype*=1, then 1<*len*<8 because Quectel IIC controller supports 8 bytes at most for transmission at a time.

- **Return Value**

If no error occurs, the length of the read data will be returned. Negative integer indicates this function fails.

5.7.5.3.5. Ql_IIC_WriteRead

This function reads data from a specified register (or address) of a specified slave.

- **Prototype**

```
s32 Ql_IIC_Write_Read(u32 chnnlNo,u8 slaveAddr,u8 * pData,u32 wrtLen,u8 * pBuffer,u32 rdLen)
```

- **Parameters**

chnnlNo:
[In] IIC channel number. It is specified by *Ql_IIC_Init* function.

slaveAddr:
[In] Slave address.

pData:
[In] Setting values of the specified register of the slave.

wrtLen:
[In] Number of bytes to be written. If *IICtype*=1, then 1<*wrtLen*<8.

pBuffer:
[Out] The buffer that stores the data read from a specific slave

rdLen:
[Out] Number of bytes to be read. If *IICtype*=1, then 1<*wrtLen*<8.

- **Return Value**

If no error occurs, the length of the read data will be returned. Negative integer indicates this function fails.

5.7.5.3.6. Ql_IIC_Uninit

This function releases specified IIC pins.

- **Prototype**

```
s32 Ql_IIC_Uninit(u32 chnnlNo)
```

- **Parameters**

chnnlNo:
[In] IIC channel number. It is specified by *Ql_IIC_Init* function.

- **Return Value**

QL_RET_OK indicates this function is executed successfully. Negative integer indicates this function fails.

5.7.5.4. Example

The following example code demonstrates the use of IIC interface.

```
void API_TEST_iic(void)
{
    s32 ret;
    u8 write_buffer[4]={0x10,0x02,0x50,0x0a};
    u8 read_buffer[6]={0x14,0x22,0x33,0x44,0x55,0x66};
    u8 registerAdrr[2]={0x01,0x45};
    Ql_Debug_Trace("\r\n<**********    IIC API Test    **********>\r\n");

    //Simulate IIC test.
    ret=Ql_IIC_Init(0, PINNAME_RI, PINNAME_DCD,0);

    //Simulated IIC interface. The IIC speed can be ignored.
    ret=Ql_IIC_Config(0, TRUE,0x07, 0);

    ret=Ql_IIC_Write(0, 0x07, write_buffer, sizeof(write_buffer));
    ret=Ql_IIC_Read(0, 0x07, read_buffer, sizeof(read_buffer));
    ret=Ql_IIC_Write_Read(0, 0x07, registerAdrr, sizeof(registerAdrr),read_buffer, sizeof(read_buffer));

    //IIC controller test
    ret=Ql_IIC_Init(1, PINNAME_CTS ,PINNAME_RTS,1);

    //IIC controller speed setting is necessary.
    ret=Ql_IIC_Config(1, TRUE, 0x07, 300);
```

```
    ret=Ql_IIC_Write(1, 0x07, write_buffer, sizeof(write_buffer));
    ret=Ql_IIC_Read(1, 0x07, read_buffer, sizeof(read_buffer));
    ret=Ql_IIC_Write_Read(1, 0x07, registerAdrr, sizeof(registerAdrr),read_buffer, sizeof(read_buffer));

    ret=Ql_IIC_Uninit(1);
}
```

5.7.6. SPI

5.7.6.1. SPI Overview

The module provides a simulate SPI interface. The interface can be simulated by GPIO pins, which can be any GPIO in the GPIO list in ***Chapter 5.7.2.2***.

5.7.6.2. SPI Usage

The following steps tell how to use the SPI function:

Step 1: Initialize SPI Interface. Call *Ql_SPI_Init* function to initialize the configurations for a SPI channel, including the specified pins for SPI, SPI type, and SPI channel number.
Step 2: Configure parameters. Call *Ql_SPI_Config* function to configure parameters for the SPI interface, including the clock polarity and clock phase.
Step 3: Write data. Call *Ql_SPI_Write* function to write bytes to the specified slave bus.
Step 4: Write and read. Call *Ql_SPI_WriteRead* function to write and read data at the same time.
Step 5: Release SPI interface. Call *Ql_SPI_Uniti* function to release SPI pins. This step is optional.

5.7.6.3. API Functions

5.7.6.3.1. Ql_SPI_Init

This function initializes the configurations for a SPI channel, including the SPI channel number and the specified GPIO pins for SPI.

- **Prototype**

```
s32 Ql_SPI_Init(u32 chnnlNo,PinName pinClk,PinName pinMiso,PinName pinMosi,bool spiType)
```

- **Parameters**

chnnlNo:
[In] SPI channel number. The range is 0~254.

pinClk:
[In] SPI CLK pin.

pinMiso:
[In] SPI MISO pin.

pinMosi:
[In] SPI MOSI pin.
spiType:
[In] SPI type. It must be 0, which indicates simulated SPI.

- **Return Value**

QL_RET_OK indicates this function is executed successfully. Negative integer indicates this function fails.

5.7.6.3.2. Ql_SPI_Config

This function configures the SPI interface.

- **Prototype**

```
s32 Ql_SPI_Config (u32 chnnlNo, bool isHost, bool cpol, bool cpha, u32 clkSpeed)
```

- **Parameters**

chnnlNo:
[In] SPI channel number. It is specified by *Ql_SPI_Init* function.

isHost:
[In] Whether to use host mode or not. It must be TRUE and only supports host mode.

cpol:
[In] Clock polarity. Please refer to the SPI standard protocol for more information.

cpha:
[In] Clock phase. Please refer to the SPI standard protocol for more information.

clkSpeed:
[In] SPI speed. It is only used for hardware SPI. The range of SPI speed is 1~28 (unit: MHz).

- **Return Value**

If no error occurs, the length of the write data will be returned. Negative integer indicates this function fails

5.7.6.3.3. Ql_SPI_Write

This function writes data to the specified slave through SPI interface.

- **Prototype**

```
s32 Ql_SPI_Write(u32 chnnlNo,u8 * pData,u32 len)
```

- **Parameters**

chnnlNo:
[In] SPI channel number. It is specified by *Ql_SPI_Init* function.

pData:
[In] Setting value to be written to the slave.

len:
[In] Number of bytes to be written.

- **Return Value**

If no error occurs, the length of the write data will be returned. Negative integer indicates this function fails.

5.7.6.3.4. Ql_SPI_Read

This function reads data from a specified slave through SPI interface.

- **Prototype**

```
s32 Ql_SPI_Read(u32 chnnlNo,u8 *pBuffer,u32 rdLen)
```

- **Parameters**

chnnlNo:
[In] SPI channel number. It is specified by *Ql_SPI_Init* function.

pBuffer:
[Out] The buffer that stores the data read from a specific slave.

rdLen:
[Out] Number of bytes to be read.

- **Return Value**

If no error occurs, the length of the read data will be returned. Negative integer indicates this function fails.

5.7.6.3.5. Ql_SPI_WriteRead

This function is used for SPI half-duplex communication.

- **Prototype**

```
s32 Ql_SPI_WriteRead(u32 chnnlNo,u8 *pData,u32 wrtLen,u8 * pBuffer,u32 rdLen)
```

- **Parameters**

chnnlNo:
[In] SPI channel number. It is specified by *Ql_SPI_Init* function.

pData:
[In] Setting value to be written to the slave.

wrtLen:
[In] Number of bytes to be written.

pBuffer:
[Out] The buffer that stores the data read from a specific slave.

rdLen:
[Out] Number of bytes to be read.

- **Return Value**

If no error occurs, the length of the read data will be returned. Negative integer indicates this function fails.

5.7.6.3.6. Ql_SPI_WriteRead_Ex

This function is used for SPI full-duplex communication.

- **Prototype**

```
s32 Ql_SPI_WriteRead_Ex(u32 chnnlNo,u8 *pData,u32 wrtLen,u8 * pBuffer,u32 rdLen)
```

- **Parameters**

chnnlNo:
[In] SPI channel number. It is specified by *Ql_SPI_Init* function.

pData:
[In] Setting value to be written to the slave.

wrtLen:
[In] Number of bytes to be written.

pBuffer:
[Out] The buffer that stores the data read from a specific slave.

rdLen:
[Out] Number of bytes to be read.

- **Return Value**

If no error occurs, the length of the read data will be returned. Negative integer indicates this function fails.

5.7.6.3.7. Ql_SPI_Uninit

This function releases the SPI pins.

- **Prototype**

```
s32 Ql_SPI_Uninit(u32 chnnlNo)
```

- **Parameters**

chnnlNo:
[In] SPI channel number. It is specified by *Ql_SPI_Init* function.

- **Return Value**

QL_RET_OK indicates this function is executed successfully. Negative integer indicates this function fails.

5.7.6.4. Example

The following example shows the use of the SPI interface.

```
void API_TEST_spi(void)
{
    s32 ret;
    u32 rdLen=0;
    u32 wdLen=0;
    u8   spi_write_buffer[]={0x01,0x02,0x03,0x0a,0x11,0xaa};
    u8   spi_read_buffer[100];
```

```
    APP_DEBUG("\r\n<***********    TEST API Test    ***********>\r\n");

    ret=Ql_SPI_Init(1,PINNAME_PCM_IN,PINNAME_PCM_SYNC,PINNAME_PCM_OUT,PINNAME_PCM_CLK,0);
    APP_DEBUG ("\r\n<--SPI channel 1 Ql_SPI_Init ret=%d-->\r\n",ret);

    ret=Ql_SPI_Config(1,1,0,0,10);    //isHost=1, cpol=0, cpha=0
    APP_DEBUG ("<--Ql_SPI_Config(), SPI channel 1, ret=%d-->",ret);

    wdLen=Ql_SPI_Write(1,spi_write_buffer,6);
    APP_DEBUG ("\r\n<--SPI channel 1 Ql_SPI_Write data len =%d-->\r\n",wdLen);

    rdLen=Ql_SPI_Read(1,spi_read_buffer,6);
    APP_DEBUG ("\r\n<--SPI channel 1 Ql_SPI_Read data len =%d-->\r\n",rdLen);

    rdLen=Ql_SPI_WriteRead(1,spi_write_buffer,6,spi_read_buffer,3);
    APP_DEBUG ("\r\n<--SPI channel 1 Ql_SPI_WriteRead Read data len =%d-->\r\n",rdLen);

    ret=Ql_SPI_Uninit(1);
    APP_DEBUG ("\r\n<--SPI channel 1 Ql_SPI_Uninit ret =%d-->\r\n",ret);
}
```

5.8. GPRS APIs

5.8.1. Overview

The API functions in this chapter are declared in header file *ql_gprs.h*.

The module supports defining and activating of 2 PDP contexts at the same time. Each PDP context supports at most 6 client socket connections and 5 server socket connections.

The examples in the *example_tcpclient.c* and *example_tcpserver.c* of QuecOpen SDK show the proper usages of these methods.

5.8.2. Usage

The following steps tell how to work with GPRS PDP context:

Step 1: Register PDP callback. Call *Ql_GPRS_Register* function to register the GPRS's callback function.

Step 2: Set PDP context. Call *Ql_GPRS_Config* function to configure the GPRS PDP context, including APN name, user name and password.

Step 3: Activate PDP. Call *Ql_GPRS_Activate* function to activate the GPRS PDP context. The result for activating GPRS will usually be informed in *Callback_GPRS_Actived*. See also the description for *Ql_GPRS_Activate* below.
Calling of *Ql_GPRS_AcitvateEx* may activate the GPRS and get the result when this API function returns. The callback function *Callback_GPRS_Actived* will not be invoked. It means this API function will be executed in blocking mode. See also the description for *Ql_GPRS_ActivateEx* below.
The maximum possible time for activating GPRS is 180s.

Step 4: Get local IP. Call *Ql_GPRS_GetLocalIPAddress* function to get the local IP address.

Step 5: Get host IP by domain name if needed. Call *Ql_GPRS_GetDNSAddress* function to retrieve the host IP address by the domain name address if a domain name address for server is used.

Step 6: Deactivate PDP context. Call *Ql_GPRS_Deactivate* function to close the GPRS PDP context. The result for deactivating GPRS is usually informed in *Callback_GPRS_Deactived*. The callback function *Callback_GPRS_Deactived* will be invoked when GPRS drops down. See also the description for *Ql_GPRS_Deactivate* below.
Calling of *Ql_GPRS_DeacitvateEx* may deactivate the GPRS and get the result when this API function returns. The callback function *Callback_GPRS_Deactived* will not be invoked. It means this API function will be executed in blocking mode. See also the description for *Ql_GPRS_DeactivateEx* below.
The maximum possible time for deactivating GPRS is 90s.

5.8.3. API Functions

5.8.3.1. Ql_GPRS_Register

This function registers the GPRS related callback functions. And these callback functions will be invoked only in the registered task.

- **Prototype**

```
s32 Ql_GPRS_Register(u8 contextId,ST_PDPContxt_Callback* callback_func,void* customParam)
```

```
typedef struct {
    void (*Callback_GPRS_Actived)(u8 contexId, s32 errCode, void* customParam);
    void (*CallBack_GPRS_Deactived)(u8 contextId, s32 errCode, void* customParam );
} ST_PDPContxt_Callback;
```

- **Parameters**

contextId:
[In] PDP context ID, which can be 0 or 1.

callback_func:
[In] Callback function, which is called by QuecOpen to inform embedded applications whether this function succeeds or not. It should be implemented by embedded applications.

customParam:
[In] One customized parameter that can be passed into callback functions.

- **Return Value**

The return value is 0 if this function succeeds. Otherwise, a value of *Enum_SocError* is returned.

5.8.3.2. Callback_GPRS_Actived

When the return value of *Ql_GPRS_Activate* is *SOC_WOULDBLOCK*, this callback function will be invoked later.

- **Prototype**

```
void (*Callback_GPRS_Actived)(u8 contexId, s32 errCode, void* customParam)
```

- **Parameters**

contextId:
[Out] PDP context ID that is specified when calling *Ql_GPRS_Activate*. It can be 0 or 1.

errCode:
[Out] The result code of activating GPRS. 0 indicates successful GPRS activation.

customParam:
[Out] One customized parameter that can be passed into *Ql_GPRS_Register*. It may be NULL.

- **Return Value**

None.

5.8.3.3. CallBack_GPRS_Deactived

When the return value of *Ql_GPRS_Deactivate* is *SOC_WOULDBLOCK*, this callback function will be invoked by core system later.

- **Prototype**

```
void (*CallBack_GPRS_Deactived)(u8 contextId, s32 errCode, void* customParam )
```

- **Parameters**

contextId:
[Out] PDP context ID that is specified when calling *Ql_GPRS_Activate*. It may be 0 or 1.

errCode:
[Out] The result code of deactivating GPRS. 0 indicates successful GPRS deactivation.

customParam:
[Out] One customized parameter that can be passed into *Ql_GPRS_Register*. It may be NULL.

- **Return Value**

None.

5.8.3.4. Ql_GPRS_Config

This function configures GPRS parameters including APN name, user name, password and authentication type for the specified PDP context.

- **Prototype**

```
s32 Ql_GPRS_Config(u8 contextId, ST_GprsConfig* cfg)
```

```
typedef struct {
    u8 apnName[MAX_GPRS_APN_LEN];
    u8 apnUserId[MAX_GPRS_USER_NAME_LEN];
    u8 apnPasswd[MAX_GPRS_PASSWORD_LEN];
    u8 authtype; //PAP or CHAP
    void* Reserved1;    //QoS
    void* Reserved2;
    } ST_GprsConfig;
```

- **Parameters**

apnName:
[In] APN name. Null-terminated characters.

apnUserId:
[In] APN user ID. Null-terminated characters.

apnPasswd:
[In] APN password. Null-terminated characters.

Authtype:
[In] Authentication method. 1 indicates PAP authentication, and 2 indicates CHAP authentication.

- **Return Value**

The possible return values are as follows:
SOC_SUCCESS: indicates this function is executed successfully.
SOC_INVAL: indicates invalid argument.
SOC_ALREADY: indicates this function is running.

5.8.3.5. Ql_GPRS_Activate

This function activates GPRS PDP context. On the basis of network status, the PDP context activation will take some time, and the longest activation time is 150s. When the PDP activation succeeds or fails, *Callback_GPRS_Actived* callback function will be called and the activation result is given.

- **Prototype**

```
s32 Ql_GPRS_Activate(u8 contextId)
```

- **Parameters**

contextId:
[In] PDP context ID, which can be 0 or 1.

- **Return Value**

The possible return values are as follows:

GPRS_PDP_SUCCESS: indicates GPRS is activated successfully.
GPRS_PDP_WOULDBLOCK: indicates the application should wait till the callback function is called. The application gets the information of success or failure in callback function. The maximum possible time for activating GPRS is 180s.
GPRS_PDP_INVAL: indicates invalid argument.
GPRS_PDP_ALREADY: indicates the activating operation is in process.
GPRS_PDP_BEARER_FAIL: indicates the bearer is broken.

- **Example**

The following codes show the process of activating GPRS.

```
{
    s32 ret;
    ret=Ql_GPRS_Activate(0);
    if (GPRS_PDP_SUCCESS==ret)
    {
```

```
        //GPRS is activated successfully.
    }
    else if (GPRS_PDP_WOULDBLOCK==ret)
    {
        //GPRS is being activated, and module needs to wait for the reault of calling
          Callback_GPRS_Actived.
    }
    else if (GPRS_PDP_ALREADY==ret)
    {
        //GPRS has been activated.
    }else{
        //Failed to activate GPRS, and the error code is in "ret".
        //Developers may retry to activate GPRS, and reset the module after 3 successive failures.
    }
}
```

5.8.3.6. Ql_GPRS_ActivateEx

This function activates a specified GPRS PDP context. The maximum possible time for activating GPRS is 180s.

This function supports two modes:

Blocking Mode: When *isBlocking* is set to TRUE, this function works under blocking mode. The result will be returned only after the operation is done.

Non-blocking Mode: When *isBlocking* is set to FALSE, this function works under non-blocking mode. The result will be returned even if the operation is not done, and the result will be reported in callback.

If working under non-blocking mode, this function is as same as *Ql_GPRS_Activate()*.

- **Prototype**

```
s32 Ql_GPRS_ActivateEx(u8 contxtId, bool isBlocking)
```

- **Parameters**

contextId:
[In] PDP context ID, which can be 0 or 1.

isBlocking:
[In] The mode that the function works in. TRUE means blocking mode, and FALSE means non-blocking mode.

- **Return Value**

The possible return values are as follows:

GPRS_PDP_SUCCESS: indicates GPRS is activated successfully.
GPRS_PDP_INVAL: indicates invalid argument.
GPRS_PDP_ALREADY: indicates the activating operation is in process.
GPRS_PDP_BEARER_FAIL: indicates the bearer is broken.

- **Example**

The following codes show the process of activating GPRS.

```
{
    s32 ret;
    ret=Ql_GPRS_Activate(0, TRUE);
    if (GPRS_PDP_SUCCESS==ret)
    {
        //GPRS is activated successfully.
    }
    else if (GPRS_PDP_ALREADY==ret)
    {
        //GPRS has been activated.
    }else{
        //Fail to activate GPRS, and the error code is in "ret".
        //Retry to activate GPRS, and reset the module after 3 successive failures.
    }
}
```

5.8.3.7. Ql_GPRS_Deactivate

This function deactivates a specified PDP context. On the basis of the network status, PDP deactivation will take some time and the maximum time is 90s. When the PDP deactivation succeeds or fails, *CallBack_GPRS_Deactived* callback function will be called and the deactivation result will be given.

- **Prototype**

```
s32 Ql_GPRS_Deactivate(u8 contextId)
```

- **Parameters**

contextId:
[In] PDP context ID that is specified when calling *Ql_GPRS_Activate*.

- **Return Value**

The return value is 0 if this function succeeds. Otherwise, a value of *ql_soc_error_enum* is returned. Please refer to the possible error codes in ***Chapter 5.9.4***.

- **Example**

The following codes show the process of deactivating GPRS.

```
{
    s32 ret;
    ret=Ql_GPRS_Deactivate(0);
    if (GPRS_PDP_SUCCESS==ret)
    {
        //GPRS is deactivated successfully.
    }
    else if (GPRS_PDP_WOULDBLOCK==ret)
    {
        //GPRS is being deactivated, and module needs to wait for the reault of calling
          Callback_GPRS_Deactived.
    }else{
        //Fail to deactivate GPRS, and the error code is in "ret".
    }
}
```

5.8.3.8. Ql_GPRS_DeactivateEx

This function deactivates a specified PDP context. The maximum possible time for deactivating GPRS is 90s.

This function supports two modes:

Non-blocking Mode: When *isBlocking* is set to FALSE, this function works under non-blocking mode. The result will be returned even if the operation is not done, and the result will be reported in callback.

Blocking Mode: When *isBlocking* is set to TRUE, this function works under blocking mode. The result will be returned only after the operation is done.

If working under non-blocking mode, this function is as same as *Ql_GPRS_Deactivate()*.

- **Prototype**

```
s32 Ql_GPRS_DeactivateEx(u8 contextId, bool isBlocking)
```

- **Parameters**

contextId:
[In] PDP context ID that is specified when calling *Ql_GPRS_Activate*.

isBlocking:
[In] The mode that the function works in. TRUE indicates blocking mode, and FALSE indicates non-blocking mode.

- **Return Value**

The possible return values are as follows:

GPRS_PDP_SUCCESS: indicates GPRS is deactivated successfully.
GPRS_PDP_INVAL: indicates invalid argument.
GPRS_PDP_ALREADY: indicates the deactivating operation is in process.
GPRS_PDP_BEARER_FAIL: indicates the bearer is broken.

- **Example**

The following codes show the process of deactivating GPRS.

```
{
    s32 ret;
    ret=Ql_GPRS_Deactivate(0, TRUE);
    if (GPRS_PDP_SUCCESS==ret)
    {
        //GPRS is deactivated successfully.
    }else{
        //Fail to deactivate GPRS, and the error code is in "ret".
    }
}
```

5.8.3.9. Ql_GPRS_GetLocalIPAddress

This function retrieves the local IP of a specified PDP context.

- **Prototype**

```
s32 Ql_GPRS_GetLocalIPAddress(u8 contxtId, u32* ipAddr)
```

- **Parameters**

contextId:
[In] PDP context ID that is specified when calling *Ql_GPRS_Activate*.

ipAddr:
[Out] Pointer to the buffer that is used to store the local IPv4 address.

- **Return Value**

If no error occurs, this return value will be *SOC_SUCCESS (0)*. Otherwise, a value of *Enum_SocError* is returned.

5.8.3.10. Ql_GPRS_GetDNSAddress

This function retrieves the DNS server's IP addresses including the first DNS address and the second DNS address.

- **Prototype**

```
s32 Ql_GPRS_GetDNSAddress(u8 contextId, u32* firstAddr, u32* secondAddr)
```

- **Parameters**

contextId:
[In] PDP context ID that is specified when calling *Ql_GPRS_Activate*.

firstAddr:
[Out] Pointer to the buffer that is used to store the primary DNS server's IP address.

secondAddr:
[Out] Pointer to the buffer that is used to store the secondary DNS server's IP address.

- **Return Value**

If no error occurs, this return value will be *SOC_SUCCESS (0)*. Otherwise, a value of *Enum_SocError* is returned.

5.8.3.11. Ql_GPRS_SetDNS Address

This function sets the DNS server's IP address.

- **Prototype**

```
s32 Ql_GPRS_SetDNSAddress(u8 contextId, u32 firstAddr, u32 secondAddr)
```

- **Parameters**

contextId:
[In] PDP context ID that is specified when calling *Ql_GPRS_Activate*.

firstAddr:
[In] A u32 integer that stores the IPv4 address.
secondAddr:
[In] A u32 integer that stores the IPv4 address.

- **Return Value**

If no error occurs, this return value will be *SOC_SUCCESS (0)*. Otherwise, a value of *Enum_SocError* is returned.

5.9. Socket APIs

5.9.1. Overview

Socket program implements the TCP and UDP protocols. In QuecOpen, API functions are used to program TCP/UDP instead of using AT commands. Each PDP context supports at most 6 client socket connections and 5 server socket connections.

The API functions in this chapter are declared in *ql_socket.h*.

5.9.2. Usage

5.9.2.1. TCP Client Socket Usage

The following steps tell how to work with TCP client socket:

Step 1: Register socket-related callback functions. Call *Ql_SOC_Register* function to register the socket-related callback functions.
Step 2: Create a socket. Call Ql_SOC_Create function to create a socket. The "contextId" argument should be the same as the one that Ql_GPRS_Register uses, and the "socketType" should be set as "SOCK_TCP".
Step 3: Connect to socket. Call *Ql_SOC_Connect* to request a socket connection. *Callback_Socket_Connect* function will be invoked no matter the connection is successful or not.
Step 4: Send data to socket. Call *Ql_SOC_Send* function to send data to socket. After the data is sent out, *Ql_SOC_GetAckNumber* function can be called to check whether the data is received by the server. If *Ql_SOC_Send* returns *SOC_WOULDBLOCK*, the application must wait for *Callback_Socket_Write* function to send data again.
Step 5: Receive data from socket. When there is data coming from the socket, *callback_socket_read* function will be invoked to inform App. After receiving the notification, App may call *Ql_SocketRecv* to receive the data. App must read out all the data. Otherwise, the callback function will not be invoked when new data comes.
Step 6: Close the socket. App can call *Ql_SOC_Close* function to close the socket. When App receives the notification that the server side has closed the socket, it has to call *Ql_SOC_Close* to close

the socket from the client side.

5.9.2.2. TCP Server Socket Usage

The following steps tell how to work with the TCP Server:

Step 1: Register the socket-related callback functions. Call *Ql_SOC_Register* function to register the socket-related callback functions.
Step 2: Create a socket. Call *Ql_SOC_Create* function to create a socket.
Step 3: Bind. Call *Ql_SOC_Bind* function to associate a local address with a socket.
Step 4: Listen. Call *Ql_SOC_Listen* function to start to listen to the connection request from listening port.
Step 5: Accept connection request. When a connection request comes, *Callback_Socket_Accept* will be invoked to inform App. App can call *Ql_SOC_Accept* function to accept the connection request.
Step 6: Send data to socket. Call *Ql_SOC_Send* function to send data to socket. After the data is sent out, please call *Ql_SOC_GetAckNumber* function to check whether the data is received by the client. When this function returns *SOC_WOULDBLOCK*, the application has to wait till *Callback_Socket_Write* is invoked, and then application can continue to send data.
Step 7: Receive data from socket. When data comes from the socket, the *callback_socket_read* will be invoked to inform App, and App can call *Ql_SocketRecv* to receive the data. App must read out all the data. Otherwise, the callback function will not be invoked when new data comes.
Step 8: Close socket. App can call *Ql_SOC_Close* function to close the socket. When App receives the notification the client side has closed the socket, it has to call *Ql_SOC_Close* to close the socket from the server side.

5.9.2.3. UDP Server Socket Usage

The following steps tell how to work with UDP Server:

Step 1: Register the socket-related callback functions. Call *Ql_SOC_Register* function to register the socket-related callback functions.
Step 2: Create a socket. Call *Ql_SOC_Create* function to create a socket. The "contextId" argument should be the same as the one that *Ql_GPRS_Register* uses, and the "socketType" should be set as "SOCK_UDP".
Step 3: Bind. Call *Ql_SOC_Bind* function to associate a local address with a socket.
Step 4: Send data to socket. Call *Ql_SOC_SendTo* function to send data. When this function returns *SOC_WOULDBLOCK*, the application has to wait till *Callback_Socket_Write* is invoked, and then App can continue to send data.
Step 5: Receive data from socket. When data comes from the socket, the *Callback_Socket_Read* function will be invoked to inform application which will then call *Ql_SocketRecvFrom* to receive the data. App must read out all the data. Otherwise, the callback function will not be invoked when new data comes.
Step 6: Close socket. App can call *Ql_SOC_Close* function to close the socket.

5.9.3. API Functions

5.9.3.1. Ql_SOC_Register

This function registers callback functions for a specified socket.

- **Prototype**

```
s32 Ql_SOC_Register(ST_SOC_Callback cb, void* customParam)
```

```
typedef struct {
    void (*callback_socket_connect)(s32 socketId, s32 errCode, void* customParam );
    void (*callback_socket_close)(s32 socketId, s32 errCode, void* customParam );
    void (*callback_socket_accept)(s32 listenSocketId, s32 errCode, void* customParam );
    void (*callback_socket_read)(s32 socketId, s32 errCode, void* customParam );
    void (*callback_socket_write)(s32 socketId, s32 errCode, void* customParam );
}ST_SOC_Callback;
```

- **Parameters**

cb:
[In] Pointer of the socket related callback function.

customParam:
[In] Customized parameter. If not used, just set it to NULL.

5.9.3.2. Callback_Socket_Connect

This callback function is invoked by *Ql_SocketConnect* when the return value of *Ql_SocketConnect* is *SOC_WOULDBLOCK*.

- **Prototype**

```
typedef void(*callback_socket_connect)(s32 socketId, s32 errCode, void* customParam)
```

- **Parameters**

socketId:
[Out] Socket ID that is returned when calling *Ql_SOC_Create*.

errCode:
[Out] Error code.

customParam:
[Out] Customized parameter. If not used, just set it to NULL.

5.9.3.3. Callback_Socket_Close

This callback function will be invoked when the socket connection is closed by the remote side. This function is valid for TCP socket only. If the socket connection is closed by the module, this function will not be invoked.

- **Prototype**

```
typedef void(*callback_socket_close)(s32 socketId, s32 errCode, void* customParam)
```

- **Parameters**

socketId:
[Out] Socket ID that is returned when calling *Ql_SOC_Create*.

errCode:
[Out] Error code.

customParam:
[Out] Customized parameter. If not used, just set it to NULL.

5.9.3.4. Callback_Socket_Accept

This function accepts a connection on a socket when the module is a server. It is valid when the module is used as TCP server only.

- **Prototype**

```
typedef void(*callback_socket_accept)(s32 listenSocketId, s32 errCode, void* customParam)
```

- **Parameters**

listenSocketId:
[Out] Socket ID that is returned when calling *Ql_SOC_Create*.

errCode:
[Out] Error code.

customParam:
[Out] Customized parameter. If not used, just set it to NULL.

- **Return Value**

None.

5.9.3.5. Callback_Socket_Read

This function will be invoked when receiving data from the socket. Please read the data via *Ql_SOC_Recv* (for TCP) or *Ql_SOC_RecvFrom* (for UDP) APIs.

- **Prototype**

```
typedef void(*callback_socket_read)(s32 socketId, s32 errCode, void* customParam)
```

- **Parameters**

socketId:
[Out] Socket ID that is returned when calling *Ql_SOC_Create*.

errCode:
[Out] Error code.

customParam:
[Out] Customized parameter. If not used, just set it to NULL.

- **Return Value**

None.

5.9.3.6. Callback_Socket_Write

When the return value of *Ql_SOC_Send* is *SOC_WOULDBLOCK*, this callback function will be invoked to enable applications to continue to send TCP data.

- **Prototype**

```
typedef void(*callback_socket_write)(s32 socketId, s32 errCode, void* customParam )
```

- **Parameters**

socketId:
[Out] Socket ID that is returned when calling *Ql_SOC_Create*.

errCode:
[Out] Error code.

customParam:
[Out] Customized parameter. If not used, just set it to NULL.

- **Return Value**

None.

5.9.3.7. Ql_SOC_Create

This function creates a socket with a specified socket ID on a specified PDP context.

- **Prototype**

```
s32 Ql_SOC_Create(u8 contextId, u8 socketType)
```

- **Parameters**

contextId:
[In] PDP context ID that is specified when calling *Ql_GPRS_Activate*. It can be 0 or 1.

socketType:
[In] Socket type. One value of *Enum_SocketType*. Socket types are enumerated in the *Enum_SocketType* as follows.

```
typedef enum{
    SOCK_TCP = 0,       //Stream socket, TCP.
    SOCK_UDP,           //Datagram socket, UDP.
} Enum_SocketType;
```

- **Return Value**

The return value is the socket ID. Otherwise, a value of *Enum_SocError* is returned. The possible return values are as follows:

SOC_INVAL: indicates invalid argument.
SOC_BEARER_FAIL: indicates the bearer is broken.
SOC_LIMIT_RESOURCE: indicates the maximum socket number exceeds.

5.9.3.8. Ql_SOC_Close

This function closes a socket.

- **Prototype**

```
s32 Ql_SOC_Close(s32 socketId)
```

- **Parameters**

socketId:
[In] Socket ID that is returned when calling *Ql_SOC_Create*.

- **Return Value**

This return value will be *SOC_SUCCESS (0)* if this function succeeds. Otherwise, a value of *Enum_SocError* is returned.

5.9.3.9. Ql_SOC_Connect

This function establishes a socket connection to the host. The host is specified by an IP address and a port number. This function is used for the TCP client only. The connecting process will take some time, and the maximum time is 75s, which depends on the network quality. When the TCP socket connection succeeds, the *Callback_Socket_Connect* callback function will be invoked.

- **Prototype**

```
s32 Ql_SOC_Connect(s32 socketId, u32 remoteIP, u16 remotePort)
```

- **Parameters**

socketId:
[In] Socket ID that is returned when calling *Ql_SOC_Create*.

remoteIP:
[In] Peer IPv4 address.

remotePort:
[In] Peer IPv4 port.

- **Return Value**

This return value will be *SOC_SUCCESS (0)* if this function succeeds. Otherwise, a value of *Enum_SocError* is returned. The possible return values are as follows:

SOC_SUCCESS: indicates this function is executed successfully.
SOC_WOULDBLOCK: indicates the application should wait till the *Callback_Socket_Connect* function is called. The application can get the information of success or failure in the callback function.
SOC_INVALID_SOCKET: indicates invalid socket.

5.9.3.10. Ql_SOC_ConnectEx

This function establishes a socket connection to the host. The host is specified by an IP address and a port number. This function is used for the TCP client only. The connecting processing will take some time, and the maximum time is 75s, which depends on the network quality. After the TCP socket connection succeeds or fails, this function returns, and the *Callback_Socket_Connect* callback function will not be invoked.

This function supports two modes:

Blocking Mode: When *isBlocking* is set to TRUE, this function works in blocking mode. The result will be returned only after the operation is done.

Non-blocking Mode: When *isBlocking* is set to FALSE, this function works under non-blocking mode. The result will be returned even if the operation is not done, and the result will be reported in callback.

If working under non-blocking mode, this function is same as *Ql_SOC_Connect()* functionally.

- **Prototype**

```
s32 Ql_SOC_ConnectEx(s32 socketId, u32 remoteIP, u16 remotePort, bool isBlocking)
```

- **Parameters**

socketId:
[In] Socket ID that is returned when calling *Ql_SOC_Create*.

remoteIP:
[In] Peer IPv4 address.

remotePort:
[In] Peer IPv4 port.

isBlocking:
[In] The mode that the function works in. TRUE indicates blocking mode, and FALSE indicates non-blocking mode.

- **Return Value**

This return value will be *SOC_SUCCESS (0)* if this function succeeds. Otherwise, a value of *Enum_SocError* is returned. The possible return values are as follows:

SOC_SUCCESS: indicates this function is executed successfully.
SOC_INVALID_SOCKET: indicates invalid socket.

Other values: indicates error codes. See *Enum_SocError* in ***Chapter 5.9.4***.

5.9.3.11. Ql_SOC_Send

This function sends data to a host which has already connected previously. It is used for TCP socket only. If *Ql_SOC_Send* function is called to send too much data to the socket buffer, this function will return *SOC_WOULDBLOCK*. Then please stop sending data. When the socket buffer has enough space, the *Callback_Socket_Write* function will be called, and the data can be send continuously. This function just sends data to the network, while it is unknown that whether the data is received by the server. So *Ql_SOC_GetAckNumber* function maybe needs to be called to check whether the data has been received by the server.

- **Prototype**

```
s32 Ql_SOC_Send(s32 socketId, u8* pData, s32 dataLen)
```

- **Parameters**

socketId:
[In] Socket ID that is returned when calling *Ql_SOC_Create*.

pData:
[In] Pointer to the data to be sent.

dataLen:
[In] Number of bytes to be sent.

- **Return Value**

If no error occurs, *Ql_SOC_Send* returns the total number of bytes sent, which can be less than the number requested to be sent by the *dataLen* parameter. Otherwise, a value of *Enum_SocError* is returned.

NOTES

1. The application should call *Ql_SOC_Send* cyclically to send data till all the data in *pData* are sent out. If the number of bytes actually sent is less than the number requested to be sent in the *dataLen* parameter, the application should keep sending out the left data.
2. If the *Ql_SocketSend* returns a negative number instead of *SOC_WOULDBLOCK*, it indicates some error happens to the socket. Therefore, the application has to close the socket by calling *Ql_SocketClose* and reestablish a connection to the socket. If the return value is *SOC_WOULDBLOCK*, the embedded application should stop sending data, and wait for the *Ql_Callback_Socket_Write()* to be invoked to continue to send data.

5.9.3.12. Ql_SOC_Recv

This function receives the TCP socket data from a connected or bound socket. When the TCP data comes from the network, the *Callback_Socket_Read* function will be called. Please use *Ql_SOC_Recv* to read the data cyclically until it returns *SOC_WOULDBLOCK* in the callback function. The *Callback_Socket_Read* function will be called if the new data is from the network again.

- **Prototype**

```
s32 Ql_SOC_Recv(s32 socketId, u8* pData, s32 dataLen)
```

- **Parameters**

socketId:
[In] Socket ID that is returned when calling *Ql_SOC_Create*.

pData:
[Out] Pointer to the buffer that is used to store the received data.

dataLen:
[Out] Length of *pData* (unit: byte).

- **Return Value**

If no error occurs, *Ql_SOC_Recv* returns the total number of bytes received. Otherwise, a value of *Enum_SocError* is returned.

NOTES

1. The application should call *Ql_SOC_Recv* cyclically in *Callback_Socket_Read* function to receive data and do data processing work till the *SOC_WOULDBLOCK* is returned.
2. If this function returns 0, it indicates the server has closed the socket. Therefore, the application has to close the socket by calling *Ql_SOC_Close* and reestablish a connection to the socket.
3. If the *Ql_SOC_Recv* returns a negative number instead of *SOC_WOULDBLOCK*, it indicates some errors happens to the socket. Therefore, the application has to close the socket by calling *Ql_SOC_Close* and reestablish a connection to the socket.

5.9.3.13. Ql_SOC_GetAckNumber

This function gets the TCP socket ACK number.

- **Prototype**

```
s32 Ql_SOC_GetAckNumber (s32 socketId, u64* ackNum)
```

- **Parameters**

socketId:
[In] Socket ID that is returned when calling *Ql_SOC_Create*.

ackNum:
[Out] Pointer to an u64 data type that is the storage space for the TCP ACK number.

- **Return Value**

If no error occurs, this return value will be *SOC_SUCCESS (0)*. Otherwise, a value of *Enum_SocError* is returned.

5.9.3.14. Ql_SOC_SendTo

This function sends data to a specific destination through UDP.

- **Prototype**

```
s32 Ql_SOC_SendTo(s32 socketId, u8* pData, s32 dataLen, u32 remoteIP, u16 remotePort)
```

- **Parameters**

socketId:
[In] Socket ID that is returned when calling *Ql_SOC_Create*.

pData:
[In] Buffer containing the data to be transmitted.

dataLen:
[In] Length of *pData*. It is in bytes.
remoteIP:
[In] Pointer to the address of the target socket.

remotePort:
[In] The target port number.

- **Return Value**

If no error occurs, this function returns the number of bytes actually sent. Otherwise, a value of *Enum_SocError* is returned.

5.9.3.15. Ql_SOC_RecvFrom

This function receives a datagram data through UDP socket.

- **Prototype**

```
s32 Ql_SOC_RecvFrom(s32 socketId, u8* pData, s32 recvLen, u32* remoteIP, u16* remotePort)
```

- **Parameters**

socketId:
[In] Socket ID that is returned when calling *Ql_SOC_Create*.

pData:
[Out] Pointer to the buffer that is used to store the received data.

rcvLen:
[Out] Length of *pData*. It is in bytes.

remoteIP:
[Out] An optional pointer to the buffer that receives the address of the connecting entity.

remotePort:
[Out] An optional pointer to an integer that contains the port number of the connecting entity.

- **Return Value**

If no error occurs, this function returns the number of bytes received. Otherwise, a value of *Enum_SocError* is returned.

5.9.3.16. Ql_SOC_Bind

This function associates a local address with a socket.

- **Prototype**

```
s32 Ql_SOC_Bind(s32 socketId, u16 localPort)
```

- **Parameters**

socketId:
[In] Socket ID that is returned when calling *Ql_SOC_Create*.

localPort:
[In] Socket local port number.

- **Return Value**

If no error occurs, this function returns *SOC_SUCCESS (0)*. Otherwise, a value of *Enum_SocError* is returned.

5.9.3.17. Ql_SOC_Listen

This function places a socket in a state of listening for an incoming connection.

- **Prototype**

```
s32 Ql_SOC_Listen(s32 listenSocketId, s32 maxClientNum)
```

- **Parameters**

listenSocketId:
[In] Socket ID that is returned when calling *Ql_SOC_Create*.

maxClientNum:
[In] Maximum connection number. It limits the maximum length of the request queue. The maximum value is 5.

- **Return Value**

If no error occurs, this function returns *SOC_SUCCESS (0)*. Otherwise, a value of *Enum_SocError* is returned.

5.9.3.18. Ql_SOC_Accept

This function permits an incoming connection attempt to a socket. When the TCP server is started and there is a client coming, the *Callback_Socket_Accept* function will be called. App can call this function in the *Callback_Socket_Accept* function to accept the connection request. The socket ID is allocated by the operating system.

- **Prototype**

```
s32 Ql_SOC_Accept(s32 listenSocketId, u32 * remoteIP, u16* remotePort)
```

- **Parameters**

listenSocketId:
[In] The listen socket ID.

remoteIP:
[Out] An optional pointer to a buffer that receives the address of the connecting entity.

remotePort:
[Out] An optional pointer to an integer that contains the port number of the connecting entity.

- **Return Value**

If no error occurs, this function returns a socket ID, which is greater than or equal to zero. Otherwise, a value of *Enum_SocError* is returned.

5.9.3.19. Ql_IpHelper_GetIPByHostName

This function retrieves host IP corresponding to a host name.

- **Prototype**

```
s32 Ql_IpHelper _GetIPByHostName (
      u8 contextId,
      u8 requestId
      u8 *hostname,
      Callback_IpHelper_GetIpByName callback_getIpByName
      )

typedef void (*Callback_IpHelper_GetIpByName)(u8 contexId, u8 requestId, s32 errCode, u32
ipAddrCnt, u32* ipAddr)
```

- **Parameters**

contextId:
[In] PDP context ID, which can be 0 or 1.

requestId:
[Out] Embedded in response message.

hostname:
[In] Host name.

callback_getIpByName:
[In] This callback is called by core system to notify whether this function retrieves host IP successfully or not.

errCode:
[Out] Error code.

ipAddrCnt:
[Out] Get the number of address.

ipAddr:
[Out] Host IPv4 address.

- **Return Value**

If no error occurs, this return value will be *SOC_SUCCESS* (0). Otherwise, a value of *Enum_SocError* is returned. However, if the *SOC_WOULDBLOCK* is returned, the application will have to wait till the *callback_getipByName* is called to know whether this function retrieves host IP successfully or not.

5.9.3.20. Ql_IpHelper_ConvertIpAddr

This function checks whether an IP address is valid or not. If yes, each segment of the IP address string will be converted into an integer to be stored in *ipaddr* parameter.

- **Prototype**

```
s32 Ql_IpHelper_ConvertIpAddr(u8 *addressstring, u32* ipaddr)
```

- **Parameters**

addressstring:
[In] IP address string.

ipaddr:
[Out] Pointer to u32 data type. Each byte stores the IP digit converted from the corresponding IP string.

- **Return Value**

The possible return values are as follows:

SOC_SUCCESS: indicates the IP address string is valid.
SOC_ERROR: indicates the IP address string is invalid.
SOC_INVAL: indicates invalid argument.

5.9.4. Possible Error Codes

The error codes are enumerated in the *Enum_SocError* as follows.

```
typedef enum
{
    SOC_SUCCESS                 = 0,
    SOC_ERROR                   = -1,
```

```
    SOC_WOULDBLOCK          = -2,
    SOC_LIMIT_RESOURCE      = -3,     //Limited resource
    SOC_INVALID_SOCKET      = -4,     //Invalid socket
    SOC_INVALID_ACCOUNT     = -5,     //Invalid account ID
    SOC_NAMETOOLONG         = -6,     //Address is too long
    SOC_ALREADY             = -7,     //Operation is already in progress
    SOC_OPNOTSUPP           = -8,     //Operation is not supported
    SOC_CONNABORTED         = -9,     //Software caused connection abortion
    SOC_INVAL               = -10,    //Invalid argument
    SOC_PIPE                = -11,    //Broken pipe
    SOC_NOTCONN             = -12,    //Socket is not connected
    SOC_MSGSIZE             = -13,    //MSG is too long
    SOC_BEARER_FAIL         = -14,    //Bearer is broken
    SOC_CONNRESET           = -15,    //TCP half-write close, i.e., FINED
    SOC_DHCP_ERROR          = -16,
    SOC_IP_CHANGED          = -17,
    SOC_ADDRINUSE           = -18,
    SOC_CANCEL_ACT_BEARER   = -19     //Cancel the activation of bearer
} Enum_SocErrCode;
```

5.9.5. Example

Please refer to the examples *example_tcpclient.c* and *example_udpclient.c* in *SDK\example*.

5.10. DFOTA APIs

QuecOpen provides DFOTA function that can upgrade App remotely. This chapter defines and describes related API functions, and demonstrates how to program with DFOTA.

5.10.1. Usage

Please refer to *Quectel_M25-QuecOpen_DFOTA_Application_Note* for the complete application solution.

5.10.2. API Functions

5.10.2.1. Ql_DFOTA_Init

This function initializes DFOTA related functions.

- **Prototype**

```
s32 Ql_DFOTA_Init(ST_FotaConfig * pFotaCfg)
```

- **Parameters**

pFotaCfg:
[In] Initialize DFOTA configurations including watchdog.

- **Return**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_PARAML: indicates parameter error.
QL_ERR_DFOTA_INIT_FAIL: indicates it fails to initialize DFOTA parameters.

5.10.2.2. Ql_DFOTA_WriteData

This function writes the delta firmware package in file system, and only allows sequential writing mechanism. Authentication mechanism is executed during writing.

- **Prototype**

```
void Ql_DFOTA_WriteData(const unsigned char *buffer, u32 Length)
```

- **Parameters**

buffer:
[In] Point to the start address of buffer.

Length:
[In] The length of delta firmware package that can be written at one time (unit: byte). It is recommended to be 512 bytes.

- **Return**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_PARAM: indicates parameter error.
QL_ERR_DFOTA_FAT_FAIL: indicates it fails to write delta firmware package in flash.

5.10.2.3. Ql_DFOTA_Finish

This function compares calculated checksum with image checksum in the header after the whole image is written.

- **Prototype**

```
void Ql_DFOTA_Finish(void)
```

- **Parameters**

Void.

- **Return**

QL_RET_OK: indicates this function is executed successfully.
QL_ERR_DFOTA_CHECK_FAIL: indicates upgrade package is unavailable.

5.10.2.4. Ql_DFOTA_Update

This function reboots the module and starts firmware or App upgrade via DFOTA.
.

- **Prototype**

```
void Ql_DFOTA_Update(void)
```

- **Parameters**

Void.

- **Return**

QL_RET_OK: indicates this function is executed successfully.
QL_ERR_DFOTA_UPGRADE_FAIL: indicates it fails to upgrade via DFOTA.

5.11. Debug APIs

The head file *ql_trace.h* must be included so that the debug functions can be called. All examples in QuecOpen SDK show the proper usages of these APIs.

5.11.1. Usage

Both application and system debug messages will be outputted through debug port with a special format. The *"CoolWatcher"* tool provided by Quectel can be used to capture and analyze these messages. If needed, please contact Quectel Technical Support team.

5.11.2. API Functions

5.11.2.1. Ql_Debug_Trace

This function formats and prints a series of characters and values through the debug port. This function is the same as that of standard "sprintf".

- **Prototype**

```
s32 Ql_Debug_Trace (char *fmt, ... )
```

- **Parameters**

%type:
A character that determines whether the associated argument is interpreted as a character, a string, or a number.

format:
Pointer to a null-terminated multibyte string that specifies how to interpret the data. The maximum string length is 512 bytes. It is a format-control string. The format specification is listed as follows:

Table 7: Format Specification for String Print

Character	Type	Output Format
c	int	Specifies a single-byte character.
d	int	Signed decimal integer.
o	int	Unsigned octal integer.
x	int	Unsigned hexadecimal integer, using "abcdef".
f	double	Float point digit.
p	Pointer to void	Prints the address of the argument in hexadecimal digits.

- **Return Value**

Number of characters printed.

NOTES

1. The string to be printed should not exceed the maximum number of bytes allowed in the buffer. Otherwise, a buffer overrun may occur.
2. The maximum number of characters allowed to be outputted is 512.
3. To print a 64-bit integer, please first convert it to characters using *Ql_sprintf()*.

5.12. RIL APIs

QuecOpen RIL-related API functions respectively implement the corresponding AT command's function. APIs can be called to send AT commands and get the response when APIs return. Please refer to *Quectel_QuecOpen_RIL_Application_Note* for QuecOpen RIL mechanism.

NOTE

The APIs defined in this chapter work normally only after calling *Ql_RIL_Initialize()*, and *Ql_RIL_Initialize()* is used to initialize RIL option after App receives the message *MSG_ID_RIL_READY*.

5.12.1. AT APIs

The API functions in this chapter are declared in header file *ril.h*.

5.12.1.1. Ql_RIL_SendATCmd

This function is used to send AT command with the result being returned synchronously. Before this function returns, the responses for AT command will be handled in the callback function *atRsp_callback*, and the paring results of AT responses can be stored in the space that the parameter *userData* points to. All AT responses string will be passed into the callback line by line. So the callback function may be called for times.

- **Prototype**

```
s32 Ql_RIL_SendATCmd(char*    atCmd,
                     u32 atCmdLen,
                     Callback_ATResponse atRsp_callback,
                     void* userData,
                     u32 timeout
                     );
typedef s32 (*Callback_ATResponse)(char* line, u32 len, void* userdata);
```

- **Parameter**

atCmd:
[In] AT command string.

atCmdLen:
[In] The length of AT command string.

atRsp_callBack:
[In] Callback function for handling the response of AT command.

userData:
[Out] Used to transfer the user parameters.

timeout:
[In] Timeout for the AT command. Unit: ms. If it is set to 0, RIL uses the default timeout time (3 min).

- **Return Value**

RIL_AT_SUCCESS: indicates the AT command is executed successfully, and the response is **OK**.
RIL_AT_FAILED: indicates it fails to execute the AT command or the response is **ERROR**.
RIL_AT_TIMEOUT: indicates AT command sending times out.
RIL_AT_BUSY: indicates the AT command is being sent.
RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, and the module needs to wait for *MSG_ID_RIL_READY* and then call *Ql_RIL_Initialize()* to initialize RIL.

- **Default Callback Function**

If this callback parameter is set to NULL, a default callback function will be called. But the default callback function only handles the simple AT response. Please refer to *Default_atRsp_callback* in *ril_atResponse.c*.

The following codes are the implementation for default callback function.

```
s32 Default_atRsp_callback(char* line, u32 len, void* userdata)
{
    if (Ql_RIL_FindLine(line, len, "OK")) //Find <CR><LF>OK<CR><LF>, <CR>OK<CR>，<LF>OK<LF>
    {
        return  RIL_ATRSP_SUCCESS;
    }
    else if (Ql_RIL_FindLine(line, len, "ERROR")                //Find <CR><LF>ERROR<CR><LF>,
<CR>ERROR<CR>，<LF>ERROR<LF>
            || Ql_RIL_FindString(line, len, "+CME ERROR:")  //Fail
            || Ql_RIL_FindString(line, len, "+CMS ERROR:")) //Fail
    {
```

```
        return   RIL_ATRSP_FAILED;
    }
    return RIL_ATRSP_CONTINUE;                          //Continue to wait.
}
```

5.12.2. Telephony APIs

This chapter defines telephony-related API functions that are implemented based on QuecOpen RIL. These APIs imeplement the equivalent functions as AT commands **ATD**, **ATA**, **ATH**.

The API functions in this chapter are declared in header file *ril_telephony.h*.

Please call *RIL_AUD_SetChannel()*/*RIL_AUD_GetChannel()* to set/get the voice channels (normal/headset/handfree) and call *RIL_AUD_SetVolume()*/*RIL_AUD_GetVolume()*, which are defined in *ril_audio.h*, to set/get the volume.

5.12.2.1. RIL_Telephony_Dial

This function dials a specified number.

- **Prototype**

```
s32 RIL_Telephony_Dial(u8 type, char* phoneNumber, s32* result)
```

- **Parameters**

type:
[In] Dialing type. It must be 0 and only supports voice call.

phoneNumber:
[In] Phone number. Null-terminated string.

result:
[Out] Result for dialing. One value of *Enum_CallState*.

- **Return Value**

RIL_AT_SUCCESS: indicates the AT command is executed successfully, and the response is **OK**.
RIL_AT_FAILED: indicates it fails to execute the AT command or the response is **ERROR**.
RIL_AT_TIMEOUT: indicates AT command sending times out.
RIL_AT_BUSY: indicates the AT command is being sent.
RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, and the module needs to wait for *MSG_ID_RIL_READY* and then call *Ql_RIL_Initialize()* to initialize RIL.

5.12.2.2. RIL_Telephony_Answer

This function answers a coming call.

- **Prototype**

```
s32 RIL_Telephony_Answer(s32 *result)
```

- **Parameters**

result:
[Out] Result for dialing. One value of *Enum_CallState.*

- **Return Value**

RIL_AT_SUCCESS: indicates the AT command is executed successfully, and the response is **OK**.
RIL_AT_FAILED: indicates it fails to execute the AT command or the response is **ERROR**.
RIL_AT_TIMEOUT: indicates AT command sending times out.
RIL_AT_BUSY: indicates the AT command is being sent.
RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, and the module needs to wait for *MSG_ID_RIL_READY* and then call *Ql_RIL_Initialize()* to initialize RIL.

5.12.2.3. RIL_Telephony_Hangup

This function hangs up the current call.

- **Prototype**

```
s32 RIL_Telephony_Hangup(void)
```

- **Parameters**

Void.

- **Return Value**

RIL_AT_SUCCESS: indicates the AT command is executed successfully, and the response is **OK**.
RIL_AT_FAILED: indicates it fails to execute the AT command or the response is **ERROR**.
RIL_AT_TIMEOUT: indicates AT command sending times out.
RIL_AT_BUSY: indicates the AT command is being sent.
RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, and the module needs to wait for *MSG_ID_RIL_READY* and then call *Ql_RIL_Initialize()* to initialize RIL.

5.12.3. SMS APIs

This chapter defines short message related API functions that are implemented based on QuecOpen RIL. These APIs implement the same functions as AT commands **AT+CMGR**, **AT+CMGS**, **AT+CMGD**, etc. The API functions in this chapter are declared in header file *ril_sms.h*.

5.12.3.1. RIL_SMS_ReadSMS_Text

This function reads a short message in text format with a specified index.

- **Prototype**

```
s32 RIL_SMS_ReadSMS_Text(u32 uIndex, LIB_SMS_CharSetEnum eCharset, ST_RIL_SMS_TextInfo* pTextInfo)
```

- **Parameters**

uIndex:
[In] The SMS index in current SMS storage.
eCharset:
[In] Character set. One value of *LIB_SMS_CharSetEnum*.

pTextInfo:
[In] Pointer of SMS information of text format.

- **Return Value**

RIL_AT_SUCCESS: indicates the AT command is executed successfully, and the response is **OK**.
RIL_AT_FAILED: indicates it fails to execute the AT command or the response is **ERROR**.
RIL_AT_TIMEOUT: indicates AT command sending times out.
RIL_AT_BUSY: indicates the AT command is being sent.
RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, and the module needs to wait for *MSG_ID_RIL_READY* and then call *Ql_RIL_Initialize()* to initialize RIL.

5.12.3.2. RIL_SMS_ReadSMS_PDU

This function reads a short message in PDU format with a specified index.

- **Prototype**

```
s32 RIL_SMS_ReadSMS_PDU(u32 uIndex, ST_RIL_SMS_PDUInfo* pPDUInfo)
```

- **Parameters**

uIndex:
[In] SMS index in current SMS storage.

pPDUInfo:
[In] Pointer of "ST_RIL_SMS_PDUInfo" structure.

- **Return Value**

RIL_AT_SUCCESS: indicates the AT command is executed successfully, and the response is **OK**.
RIL_AT_FAILED: indicates it fails to execute the AT command or the response is **ERROR**.
RIL_AT_TIMEOUT: indicates AT command sending times out.
RIL_AT_BUSY: indicates the AT command is being sent.
RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, and the module needs to wait for *MSG_ID_RIL_READY* and then call *Ql_RIL_Initialize()* to initialize RIL.

5.12.3.3. RIL_SMS_SendSMS_Text

This function sends a short message in text format.

- **Prototype**

```
s32 RIL_SMS_SendSMS_Text(char* pNumber, u8 uNumberLen, LIB_SMS_CharSetEnum eCharset, u8* pMsg, u32 uMsgLen,u32 *pMsgRef)
```

- **Parameters**

pNumber:
[In] Pointer of phone number.

uNumberLen:
[In] The length of phone number.

eCharset:
[In] Character set. One value of *LIB_SMS_CharSetEnum*.

pMsg:
[In] Pointer of message content.

uMsgLen:
[In] The length of message content.

pMsgRef:
[Out] Pointer of message reference number.

- **Return Value**

RIL_AT_SUCCESS: indicates the AT command is executed successfully, and the response is **OK**.
RIL_AT_FAILED: indicates it fails to execute the AT command or the response is **ERROR**.
RIL_AT_TIMEOUT: indicates AT command sending times out.
RIL_AT_BUSY: indicates the AT command is being sent.
RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, and the module needs to wait for *MSG_ID_RIL_READY* and then call *Ql_RIL_Initialize()* to initialize RIL.

5.12.3.4. RIL_SMS_SendSMS_PDU

This function sends a short message of PDU format.

- **Prototype**

```
s32 RIL_SMS_SendSMS_PDU(char* pPDUStr,u32 uPDUStrLen,u32 *pMsgRef)
```

- **Parameters**

pPDUStr:
[In] Pointer of PDU string.

uPDUStrLen:
[In] The length of PDU string.

pMsgRef:
[Out] Pointer of message reference number.

- **Return Value**

RIL_AT_SUCCESS: indicates the AT command is executed successfully, and the response is **OK**.
RIL_AT_FAILED: indicates it fails to execute the AT command or the response is **ERROR**.
RIL_AT_TIMEOUT: indicates AT command sending times out.
RIL_AT_BUSY: indicates the AT command is being sent.
RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, and the module needs to wait for *MSG_ID_RIL_READY* and then call *Ql_RIL_Initialize()* to initialize RIL.

5.12.3.5. RIL_SMS_DeleteSMS

This function deletes one or more short messages in current SMS storage with a specified rule.

- **Prototype**

```
s32 RIL_SMS_DeleteSMS(u32 uIndex,Enum_RIL_SMS_DeleteFlag eDelFlag)
```

- **Parameters**

uIndex:
[In] The index number of SMS message.

eDelFlag:
[In] Delete flag. One value of *Enum_RIL_SMS_DeleteFlag*.

- **Return Value**

RIL_AT_SUCCESS: indicates the AT command is executed successfully, and the response is **OK**.
RIL_AT_FAILED: indicates it fails to execute the AT command or the response is **ERROR**.
RIL_AT_TIMEOUT: indicates AT command sending times out.
RIL_AT_BUSY: indicates the AT command is being sent.
RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, and the module needs to wait for *MSG_ID_RIL_READY* and then call *Ql_RIL_Initialize()* to initialize RIL.

5.12.4. (U)SIM APIs

The API functions in this chapter are declared in *ril_sim.h*.

5.12.4.1. RIL_SIM_GetSimState

This function gets the state of (U)SIM card.

- **Prototype**

```
s32 RIL_SIM_GetSimState(s32* state)
```

- **Parameters**

state:
[Out] (U)SIM card state code. One value of *Enum_SIMState*.

- **Return Value**

RIL_AT_SUCCESS indicates the AT command is executed successfully; or see *Enum_ATSndError*.

5.12.4.2. RIL_SIM_GetIMSI

This function gets the IMSI number of (U)SIM card.

- **Prototype**

```
s32 RIL_SIM_GetIMSI(char* imsi)
```

- **Parameters**

imsi:
[Out] IMSI number. A string of 15 bytes.

- **Return Value**

RIL_AT_SUCCESS indicates the AT command is executed successfully; or see *Enum_ATSndError*.

5.12.4.3. RIL_SIM_GetCCID

This function gets the CCID number of (U)SIM card.

- **Prototype**

```
s32 RIL_SIM_GetCCID(s32* ccid)
```

- **Parameters**

ccid:
[Out] CCID number. A string of 20 bytes.

- **Return Value**

RIL_AT_SUCCESS indicates the AT command is executed successfully; or see *Enum_ATSndError*.

5.12.5. Network APIs

The API functions in this chapter are declared in *ril_network.h*.

5.12.5.1. RIL_NW_GetGSMState

This function gets the GSM network registration state.

- **Prototype**

```
s32 RIL_NW_GetGSMState(s32 *stat)
```

- **Parameters**

stat:
[Out] GSM state.

- **Return Value**

Network registration state code. One value of *Enum_NetworkState*. -1 indicates it fails to get the network state.

5.12.5.2. RIL_NW_GetGPRSState

This function gets the GPRS network registration state.

- **Prototype**

```
s32 RIL_NW_GetGPRSState(s32 *stat)
```

- **Parameters**

stat:
[Out] GPRS network registration state.

- **Return Value**

Network registration state code. One value of *Enum_NetworkState*. -1 indicates it fails to get the network state.

5.12.5.3. RIL_NW_GetSignalQuality

This function gets the signal quality level and bit error rate.

- **Prototype**

```
s32 RIL_NW_GetSignalQuality(u32* rssi, u32* ber)
```

- **Parameters**

rssi:
[Out] Signal quality level. 0~31 or 99. 99 indicates the module is not registered on GSM network.

ber:
[Out] Bit error code of the signal.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_INVALID_PARAMETER: indicates there is an error for input parameters.

5.12.5.4. RIL_NW_SetGPRSContext

This function sets a PDP foreground context.

- **Prototype**

```
s32 RIL_NW_SetGPRSContext(u8 foregroundContext)
```

- **Parameters**

foregroundContext:
[In] Foreground context. A numeric indicates which context will be set as foreground context. The range is 0~1.

- **Return Value**

RIL_AT_SUCCESS: indicates the AT command is executed successfully, and the response is OK.
RIL_AT_FAILED: indicates it fails to execute the AT command or the response is ERROR.
RIL_AT_TIMEOUT: indicates AT command sending times out.
RIL_AT_BUSY: indicates the AT command is being sent.
RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, and the module needs to wait for *MSG_ID_RIL_READY* and then call *Ql_RIL_Initialize()* to initialize RIL.

5.12.5.5. RIL_NW_SetAPN

This function sets the default APN of the module.

- **Prototype**

```
s32 RIL_NW_SetAPN(u8 mode, u8* apn, u8* userName, u8* password)
```

- **Parameters**

mode:
[In] Network mode. 0 indicates CSD (not supported). 1 indicates GPRS.

apn:
[In] APN string.

userName:
[In] User name for APN.

password:
[In] Password for APN.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_INVALID_PARAMETER: indicates there is an error for input parameters.

5.12.5.6. RIL_NW_OpenPDPContext

This function opens/activates the PDP foreground context. The PDP context ID is specified by *RIL_NW_SetGPRSContext()*.

- **Prototype**

```
s32 RIL_NW_OpenPDPContext(void)
```

- **Parameters**

Void.

- **Return Value**

RIL_AT_SUCCESS: indicates the AT command is executed successfully, and the response is **OK**.
RIL_AT_FAILED: indicates it fails to execute the AT command or the response is **ERROR**.
RIL_AT_TIMEOUT: indicates sending AT command timed out.
RIL_AT_BUSY: indicates the AT command is being sent.
RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, and the module needs to wait for *MSG_ID_RIL_READY* and then call *Ql_RIL_Initialize()* to initialize RIL.

5.12.5.7. RIL_NW_ClosePDPContext

This function closes/deactivates the PDP foreground context. The PDP context ID is specified by *RIL_NW_SetGPRSContext()*.

- **Prototype**

```
s32 RIL_NW_ClosePDPContext(void)
```

- **Parameters**

Void.

- **Return Value**

RIL_AT_SUCCESS: indicates the AT command is executed successfully, and the response is **OK**.
RIL_AT_FAILED: indicates it fails to execute the AT command or the response is **ERROR**.
RIL_AT_TIMEOUT: indicates AT command sending times out.
RIL_AT_BUSY: indicates the AT command is being sent.
RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, and the module needs to wait for *MSG_ID_RIL_READY* and then call *Ql_RIL_Initialize()* to initialize RIL.

5.12.5.8. RIL_NW_GetOperator

This function gets the network operator that the module is registered to.

- **Prototype**

```
s32 RIL_NW_GetOperator(char* operator)
```

- **Parameters**

operator:
[Out] A string with maximum 16 characters, which indicates the network operator that the module is registered to.

- **Return Value**

RIL_AT_SUCCESS: indicates the AT command is executed successfully, and the response is **OK**.
RIL_AT_FAILED: indicates it fails to execute the AT command or the response is **ERROR**.
RIL_AT_TIMEOUT: indicates AT command sending times out.
RIL_AT_BUSY: indicates the AT command is being sent.
RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, and the module needs to wait for *MSG_ID_RIL_READY* and then call *Ql_RIL_Initialize()* to initialize RIL.

5.12.6. GSM Location APIs

The API functions in this chapter are declared in *ril_location.h*.

5.12.6.1. RIL_GetLocation

This function retrieves the longitude and latitude of the current place of the module.

- **Prototype**

```
s32 RIL_GetLocation(CB_LocInfo cb_loc)
typedef void(*CB_LocInfo)(s32 result,ST_LocInfo* loc_info);
```

- **Parameters**

cb_loc:
Pointer to a callback function that tells the location information.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_INVALID_PARAMETER: indicates there is an error for input parameters.

5.12.7. System APIs

The API functions in this chapter are declared in header file *ril_system.h*.

5.12.7.1. RIL_GetPowerSupply

This function queries the battery balance and battery voltage.

- **Prototype**

```
s32 RIL_GetPowerSupply(u32* capacity, u32* voltage)
```

- **Parameters**

capacity:
[Out] Battery balance. A percentage ranging from 1~100.

voltage:
[Out] Battery voltage. Unit: mV.

- **Return Value**

RIL_AT_SUCCESS: indicates the AT command is executed successfully, and the response is **OK**.
RIL_AT_FAILED: indicates it fails to execute the AT command or the response is **ERROR**.
RIL_AT_TIMEOUT: indicates AT command sending times out.
RIL_AT_BUSY: indicates the AT command is being sent.

RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, and the module needs to wait for *MSG_ID_RIL_READY* and then call *Ql_RIL_Initialize()* to initialize RIL.

5.12.7.2. RIL_GetIMEI

This function retrieves the IMEI number of the module.

- **Prototype**

```
s32 RIL_GetIMEI(char* imei)
```

- **Parameters**

imei:
[Out] Buffer to store the IMEI number. The length of the buffer should be at least 15 bytes.

- **Return Value**

RIL_AT_SUCCESS: indicates the AT command is executed successfully, and the response is **OK**.
RIL_AT_FAILED: indicates it fails to execute the AT command or the response is **ERROR**.
RIL_AT_TIMEOUT: indicates AT command sending times out.
RIL_AT_BUSY: indicates the AT command is being sent.
RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, and the module needs to wait for *MSG_ID_RIL_READY* and then call *Ql_RIL_Initialize()* to initialize RIL.

5.12.8. Audio APIs

5.12.8.1. RIL_AUD_SetChannel

This function sets the audio channel.

- **Prototype**

```
s32 RIL_AUD_SetChannel(Enum_AudChannel audChannel)
```

- **Parameters**

audChannel:
[Out] Audio channel. See *Enum_AudChannel*.

- **Return Value**

RIL_AT_SUCCESS indicates the AT command is executed successfully; or see *Enum_ATSndError*.

5.12.8.2. RIL_AUD_GetChannel

This function gets the audio channel.

- **Prototype**

```
s32 RIL_AUD_GetChannel(Enum_AudChannel *pChannel)
```

- **Parameters**

pChannel:
[Out] Audio channel. See *Enum_AudChannel*.

- **Return Value**

RIL_AT_SUCCESS indicates the AT command is executed successfully; or see *Enum_ATSndError*.

5.12.8.3. RIL_AUD_SetVolume

This function sets the volume level with a specified volume type.

- **Prototype**

```
s32 RIL_AUD_SetVolume(Enum_VolumeType volType, u8 volLevel)
```

- **Parameters**

volType:
[In] Volume type. See *Enum_VolumeType*.

volLevel:
[In] Volume level.

- **Return Value**

RIL_AT_SUCCESS indicates the AT command is executed successfully; or see *Enum_ATSndError*.

5.12.8.4. RIL_AUD_GetVolume

This function gets the volume level with a specified volume type.

- **Prototype**

```
s32 RIL_AUD_GetVolume(Enum_VolumeType volType, u8* pVolLevel)
```

- **Parameters**

volType:
[In] Volume type. See *Enum_VolumeType*.

pvolLevel:
[Out] Volume level.

- **Return Value**

RIL_AT_SUCCESS indicates the AT command is executed successfully; or see *Enum_ATSndError*.

5.12.8.5. RIL_AUD_RegisterPlayCB

This function registers a callback function that will be invoked to indicate the playing result.

If a feedback (end indication or error code) for playing is needed when calling APIs *RIL_AUD_PlayFile*, please call this API to register a callback function before calling playing API.

- **Prototype**

```
typedef void (*RIL_AUD_PLAY_IND)(s32 errCode);
s32 RIL_AUD_RegisterPlayCB(RIL_AUD_PLAY_IND audCB)
```

- **Parameters**

audCB:
[In] The callback function for playing.

errCode:
[Out] Error code for audio playing, which is defined in **AT+QAUDPLAY**.

- **Return Value**

RIL_AT_SUCCESS indicates the AT command is executed successfully; or see *Enum_ATSndError*.

5.12.8.6. RIL_AUD_PlayFile

This function plays a specified audio file.

- **Prototype**

```
s32 RIL_AUD_PlayFile(char* filePath, bool isRepeated)
```

- **Parameters**

filePath:
[In] Source code file name with file path.

isRepeated:
[In] Repeat play mode.

- **Return Value**

RIL_AT_SUCCESS indicates the AT command is executed successfully; or see *Enum_ATSndError*.

5.12.8.7. RIL_AUD_StopPlay

This function stops playing the audio file.

- **Prototype**

```
s32 RIL_AUD_StopPlay(void)
```

- **Parameters**

Void.

- **Return Value**

RIL_AT_SUCCESS indicates the AT command is executed successfully; or see *Enum_ATSndError*.

5.12.8.8. RIL_AUD_StartRecord

This function starts to record with a spedifed audio format. The recording data will be recorded into a specified file in UFS.

- **Prototype**

```
s32 RIL_AUD_StartRecord(char* fileName, Enum_AudRecordFormat format)
```

- **Parameters**

fileName:
[In] Name of the file, which is used to store the recorded data.

format:
[In] Recording data format. One value of *Enum_AudRecordFormat*.

- **Return Value**

RIL_AT_SUCCESS indicates the AT command is executed successfully; or see *Enum_ATSndError*.

5.12.8.9. RIL_AUD_StopRecord

This function stops recording.

- **Prototype**

```
s32 RIL_AUD_StopRecord(void)
```

- **Parameters**

Void.

- **Return Value**

RIL_AT_SUCCESS indicates the AT command is executed successfully; or see *Enum_ATSndError*.

5.12.8.10. RIL_AUD_GetRecordState

This function gets the current state of recorder.

- **Prototype**

```
s32 RIL_AUD_GetRecordState(u8* pState)
```

- **Parameters**

pState:
[Out] Recording state. 0 indicates the recorder is in idle state; 1 indicates the recorder is recording.

- **Return Value**

RIL_AT_SUCCESS indicates the AT command is executed successfully; or see *Enum_ATSndError*.

5.12.8.11. Ql_AUD_PlayBuf

This function plays audio data and supports audio data in the fomat of .mp3, .amr, or .wav.

- **Prototype**

```
s32 Ql_AUD_PlayBuf(u8 *databuf,u32 len,QlPlayBuf_Format format,u8 loop,u8 pVolume ,u8 channel)
```

- **Parameters**

databuf:
[In] Audio data in a specific format.

len:
[In] Audio data length.

format:
[In] Data format.

loop:
[In] Whether to loop. 1 indicates to loop; 0 indicates to play once.

pVolume:
[In] Volume of the audio being played. Range: 0-100.

channel:
[In] Channel type. 0 indicates the general channel; 2 indicates the speaker channel.

- **Return Value**

RIL_AT_SUCCESS indicates the function succeeds.

5.12.8.12. Ql_AUD_StopPlayBuf

This function stops playing audio.

- **Prototype**

```
void Ql_AUD_StopPlayBuf(void)
```

- **Parameters**

Void.

- **Return Value**

Void.

6 Appendix A References

Table 8: Reference Documents

SN	Document Name	Remark
[1]	Quectel_M25-QuecOpen_Hardware_Design	M25-QuecOpen Hardware Design
[2]	Quectel_M25-QuecOpen_QFlash_User_Guide	M25-QuecOpen QFlash User Guide
[3]	Quectel_M25-QuecOpen_DFOTA_Application_Note	M25-QuecOpen DFOTA Application Note
[4]	Quectel_QuecOpen_RIL_Application_Note	QuecOpen RIL Application Note

Table 9: Abbreviations

Abbreviation	Description
ACK	Acknowledgement
ADC	Analog-to-digital Converter
API	Application Programming Interface
APN	Access Point Name
App	QuecOpen Application
CCID	Circuit Card Identity
CHAP	Challenge Handshake Authentication Protocol
Core	Core System; QuecOpen Operating System
CPU	central processing unit
CSD	Circuit Switched Data
DCB	Data Center Bridging
DFOTA	Delta Firmware Upgrade Over-The-Air

DNS	Domain Name System
EINT	External Interrupt Input
FMP	Find Me Profile
GCC	GNU Compiler Collection
GPIO	General Purpose Input Output
GPRS	General Packet Radio Service
GPS	Global Positioning System
GSM	Global System for Mobile communications
I/O	Input/Output
IIC	Inter-Integrated Circuit
IMSI	International Mobile Subscriber Identification Number
IP	Internet Protocol
KB	Kilobytes
M2M	Machine-to-Machine
MB	Megabytes
MCU	Micro Control Unit
MIMT	Man-in-the-Middle Attack
PAP	Password Authentication Protocol
PDP	Packet Data Protocol
PDU	Protocol Data Unit
RAM	Random-Access Memory
RIL	Radio Interface Layer
RTC	Real Time Clock
SDK	Software Development Kit
SMS	Short Messaging Service
SPI	Serial Peripheral Interface

TCP	Transfer Control Protocol
TLV	Type-Length-Value
UART	Universal Asynchronous Receiver and Transmitter
UDP	User Datagram Protocol
UFS	Universal Flash Storage
UID	User Identification
URC	Unsolicited Result Code
(U)SIM	(Universal) Subscriber Identity Module
WTD	Watchdog

Table 10: Format Map of Properties and Permission

Properties	**Format Map**
Default	0
Broadcast	1
Read	2
Write without response	4
Write	8
Notify	16
Indicate	32
Signed write	64
Extended properties	128
Permission	**Format Map**
Read	1
Read with encrypted protection	2
Read with MITM protection	4
Write	16

Write with encrypted protection	32
Write with MITM protection	64
Signed write	128
Signed write with MITM protection	256