MC60-OpenCPU

User Guide

GSM/GPRS/GNSS Module Series

Rev. MC60-OpenCPU_User_Guide_V1.1

Date: 2016-10-09

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

Quectel Wireless Solutions Co., Ltd.
Office 501, Building 13, No.99, Tianzhou Road, 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/salesupport.aspx

For technical support, or to report documentation errors, please visit:
http://www.quectel.com/support/techsupport.aspx
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	2016-07-22	Hyman DING	Initial
1.1	2016-10-09	Ablaze LU	1. Added three multiplexing pins as GPIO port: PINNAME_SIM2_CLK, PINNAME_SIM2_DATA and PINNAME_SIM2_RST (Table 6) 2. Newly opened five GPIOs: PINNAME_GPIO0, PINNAME_GPIO1, PINNAME_GPIO2, PINNAME_GPIO3 and PINNAME_GPIO4 (Table 6)

Contents

Table Index

Figure Index

1 Introduction

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

2 OpenCPU Platform

2.1. System Architecture

The following figure shows the fundamental principle of OpenCPU software architecture.

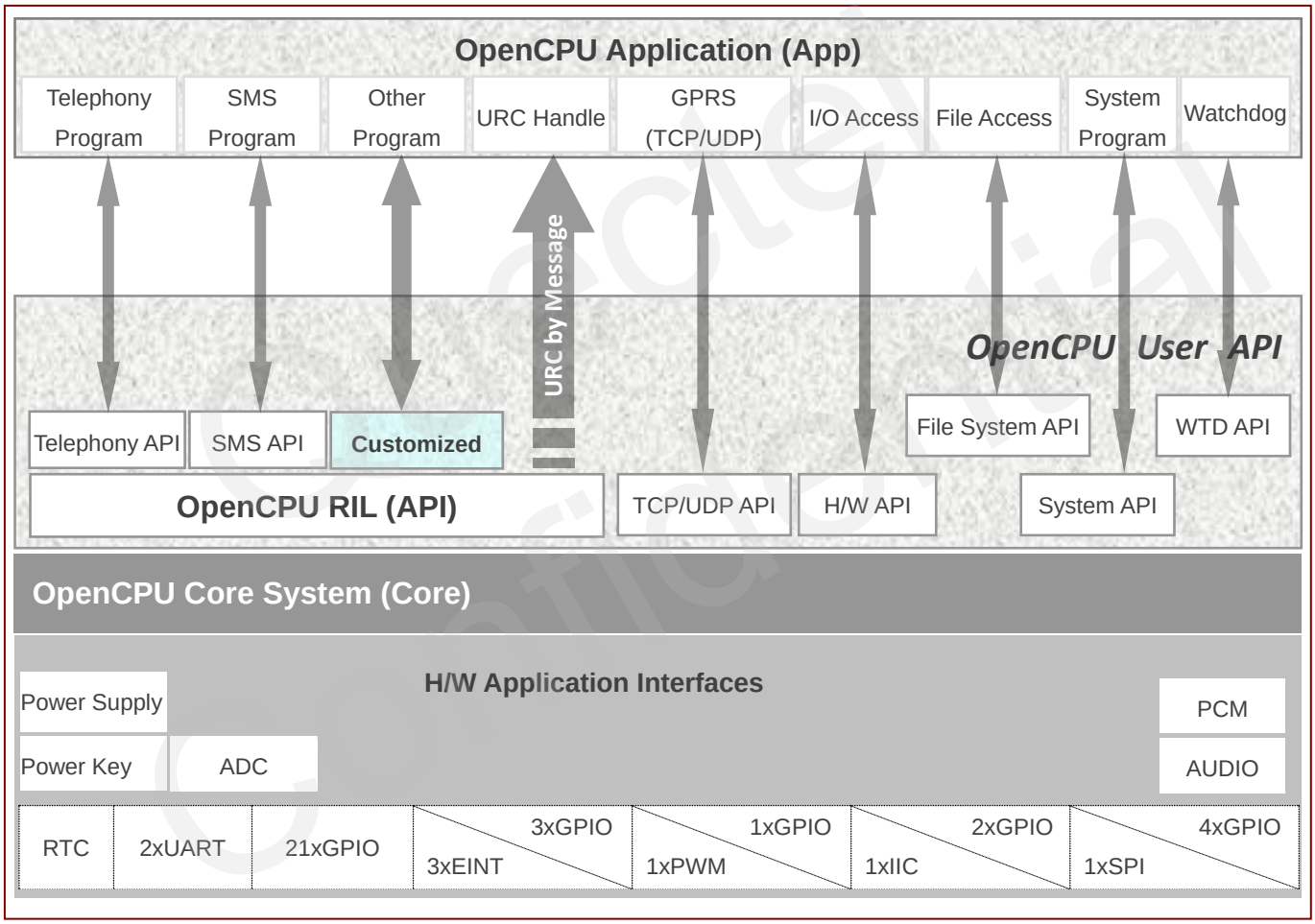

Figure 1: The Fundamental Principle of OpenCPU Software Architecture

PWM, EINT, IIC, SPI are multiplexing interfaces with GPIOs.

OpenCPU Core System is a combination of hardware and software of GSM/GPRS/GNSS module. It has built-in ARM7EJ-S processor, and has been built over Nucleus operating system which has the characteristics of micro-kernel, real-time, multi-tasking, etc.

OpenCPU User APIs are designed for access to hardware resources, radio communications resources, user file system, or external devices. All APIs are introduced in ***Charpter 5***.

OpenCPU RIL is an open source layer, which enables developers to simply call API to send AT and get the response when API returns. Additionally, developers can easily add a new API to implement an AT command. Please also refer to document ***Quectel_OpenCPU_RIL_Application_Note***.

In OpenCPU RIL, all URC messages of 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.

2.2. Open Resources

2.2.1. Processor

32-bit ARM7EJ-STM RISC 260MHz.

2.2.2. Memory Scheme

MC60-OpenCPU module builds in 4MB flash and 4MB RAM.

- User App Code Space: 320KB space available for image bin.
- RAM Space: 100KB static memory and 500KB dynamic memory.
- User File System Space: 120KB available.

2.3. Interfaces

2.3.1. Serial Interfaces

OpenCPU provides 2 UART ports: MAIN UART and DEBUG UART. They are also named as UART1 and UART2, respectively. Please refer to ***Chapter 5.7.1*** for software API functions.

UART1 is a 9-pin serial interface with RTS/CTS HW handshake. UART2 is a 3-wire interface. UART2 has debug function that can debug the Core System. Please refer to ***Chapter 5.12*** for details.

2.3.2. GPIO

There are 21 I/O pins that can be configured for general purpose I/O. All pins can be accessed under OpenCPU by API functions. Please refer to ***Chapter 5.7.2*** for details.

2.3.3. EINT

OpenCPU supports external interrupt input. There are three I/O pins that can be configured for external interrupt input. But the EINT cannot be used for the purpose of highly frequent interrupt detection, which causes module unstably working. The EINT pins can be accessed by APIs. Please refer to ***Chapter 5.7.3*** for details.

2.3.4. PWM

There is one I/O pin that can be configured for PWM. There are 32K and 13M clock sources that are available. The PWM pin can be configured and controlled by APIs. Please refer to ***Chapter 5.7.4*** for details.

2.3.5. ADC

There's an analogue input pins that can be configured for ADC. The sampling period and count can be configured by an API. Please refer to ***Chapter 5.7.5***.

Please refer to the ***document [2]*** for the characteristics of ADC interface.

2.3.6. IIC

MC60 OpenCPU provides a hardware IIC interface. Please refer to ***Chapter 5.7.6*** for programming API functions.

2.3.7. SPI

MC60 OpenCPU provides a hardware SPI interface. The SPI interface is multiplexing with PCM interface. And also both of them are multiplexing with GPIOs. Please refer to ***Chapter 5.7.7*** for programming API functions.

2.3.8. Power Key

In OpenCPU, App can catch the behavior that power key is pressed down or released. Then developers may redefine the behavior of pressing power key. Please also refer to ***Chapters 4.3.1***, ***5.4.2.2*** and ***5.4.2.3***.

2.4. Development Environment

2.4.1. SDK

OpenCPU SDK provides the resources as follows for developers:

- Compile 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.
- Pack tool for FOTA upgrade.

Customers may get the latest SDK package from sales channel.

2.4.2. Editor

Any text editor is 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 & Compiling

2.4.3.1. Complier

OpenCPU uses GCC as the compiler, and the compiler edition is “Sourcery CodeBench”. The document ***Quectel_OpenCPU_GCC_Installation_Guide*** tells the ways of establishing GCC environment.

2.4.3.2. Compiling

In OpenCPU, compiling commands are executed in command line. The compiling and clean commands are defined as follows.

```
make clean
make new
```

2.4.3.3. Compiling Output

In command-line, some compiler processing information will be output during compiling. All WARNINGs

and ERRORs are recorded in \SDK\build\gcc\build.log.

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

For example, in line 195 in example_at.c, the semicolon is missed intentionally.

```
194     // Handle the response...
195     Ql_Debug_Trace("<-- Send 'AT+GSN' command, Response:%s -->\r\n\r\n", ATResponse)
196     if (0 == ret)
```

When compiling this example program, a compiling error will be thrown out. In build.log, it goes like this:

```
example/example_at.c:196:5: error: expected ';' before 'if'
make.exe[1]: *** [build\gcc\obj/example/example_at.o] Error 1
make: *** [all] Error 2
```

If there is no any compiling error during compiling, the prompt for successful compiling is given.

```
--------------------------------------------------
- GCC Compiling Finished Sucessfully.
- The target image is in the 'build\gcc' directory.
--------------------------------------------------
```

2.4.4. Download

The document ***Quectel_QFlash_User_Guide*** introduces the download tool and the way to use it to download application bin.

2.4.5. How to Program

By default, the “custom” directory has been designed to store the customer source code files in SDK.

2.4.5.1. Program Composition

OpenCPU program consists of the aspects as follows.

Table 1: OpenCPU 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 compile.

2.4.5.2. Program Framework

The following codes are the least codes that comprise an OpenCPU Embedded Application.

```
/**
 * The entrance of this application
 */
void proc_main_task(s32 taskId)
{
    ST_MSG msg;

    //Start message loop of this task
    while (1)
    {
        Ql_OS_GetMessage(&msg);
        switch(msg.message)
        {
        case MSG_ID_RIL_READY:
            {
                Ql_Debug_Trace("<-- RIL is ready -->\r\n");

                //Before using the RIL feature, you must initialize it by calling the following API.
                //After receiving the 'MSG_ID_RIL_READY' message.
                Ql_RIL_Initialize();

                //Now you can start to send AT commands.
                Demo_SendATCmd();
                break;
            }
        case MSG_ID_URC_INDICATION:
        {
            //Ql_Debug_Trace("<-- Received URC: type: %d, -->\r\n", msg.param1);
            switch (msg.param1)
            {
            case URC_SYS_INIT_STATE_IND:
                Ql_Debug_Trace("<-- Sys Init Status %d -->\r\n", msg.param2);
                break;
            case URC_SIM_CARD_STATE_IND:
                Ql_Debug_Trace("<-- SIM Card Status:%d -->\r\n", msg.param2);
                break;
```

```
            case URC_GSM_NW_STATE_IND:
                Ql_Debug_Trace("<-- GSM Network Status:%d -->\r\n", msg.param2);
                break;
            case URC_GPRS_NW_STATE_IND:
                Ql_Debug_Trace("<-- GPRS Network Status:%d -->\r\n", msg.param2);
                break;
            case URC_CFUN_STATE_IND:
                Ql_Debug_Trace("<-- CFUN Status:%d -->\r\n", msg.param2);
                break;
            case URC_COMING_CALL_IND:
                {
                    ST_ComingCall* pComingCall = (ST_ComingCall*)msg.param2;
                    Ql_Debug_Trace("<-- Coming call, number:%s, type:%d -->\r\n",
pComingCall->phoneNumber, pComingCall->type);
                    break;
                }
            case URC_CALL_STATE_IND:
                switch (msg.param2)
                {
                case CALL_STATE_BUSY:
                    Ql_Debug_Trace("<-- The number you dialed is busy now -->\r\n");
                    break;
                case CALL_STATE_NO_ANSWER:
                    Ql_Debug_Trace("<-- The number you dialed has no answer -->\r\n");
                    break;
                case CALL_STATE_NO_CARRIER:
                    Ql_Debug_Trace("<-- The number you dialed cannot reach -->\r\n");
                    break;
                case CALL_STATE_NO_DIALTONE:
                    Ql_Debug_Trace("<-- No Dial tone -->\r\n");
                    break;
                default:
                    break;
                }
                break;
            case URC_NEW_SMS_IND:
                Ql_Debug_Trace("<-- New SMS Arrives: index=%d\r\n", msg.param2);
                break;
            case URC_MODULE_VOLTAGE_IND:
                Ql_Debug_Trace("<-- VBatt Voltage Ind: type=%d\r\n", msg.param2);
                break;
            default:
                Ql_Debug_Trace("<-- Other URC: type=%d\r\n", msg.param1);
                break;
```

```
            }
            break;
        }
        //
        //Case other user message ID...
        //
        default:
            break;
        }
    }
}
```

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

Ql_OS_GetMessage is an important system function that the Embedded Application receives messages from message queue of the task.

MSG_ID_RIL_READY is a system message that RIL module sends to main task.

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

2.4.5.3. Makefile

In OpenCPU, the compiler compiles program according to the definitions in makefile. The profile of makefile has been pre-designed and is ready for use. However, developers need to change some settings before compiling program according to native conditions, such as compiler environment path.

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

Environment path definition of compiler
Preprocessor definitions
Definitions for include search paths
Source code directories and files to compile
Lib files to link

2.4.5.4. How to Add .c File

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

2.4.5.5. How to Add Directory

If developers need to add new directory in “custom”, please follow the steps below.

First, add the new directory name in variable “SRC_DIRS” in \SDK\make\gcc\gcc _makefile\gcc_makefile, and define the source code files to compile.

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

Secondly, define the source code files to compile in the new directory.

```
SRC_SYS=$(wildcard custom/config/*.c)
SRC_SYS_RIL=$(wildcard ril/src/*.c)
SRC_EXAMPLE=$(wildcard example/*.c)
SRC_CUS=$(wildcard custom/*.c)



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

3 Basic Data Types

3.1. Required Header

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

3.2. Required Header

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;
float	Floating-point variable. This variable is declared in math.h.

4 System Configuration

In the \SDK\custom\config directory, developers can reconfigure the application according to requirements, such as heap memory size, add tasks and stack size of tasks, GPIO initial status. All configuration files for developers are named as prerfix “custom_”.

Table 3: System Config File List

Config File	Description
custom_feature_def.h	OpenCPU features enabled. Now only include RIL. Developers generally don’t need to change this file.
custom_gpio_cfg.h	Configurations for GPIO initial status
custom_heap_cfg.h	Definition of heap size
custom_task_cfg.h	Multitask configuration
custom_sys_cfg.c	Other system configurations, including power key, speciified GPIO pin for external watchdog, and setting working mode of debug port.

4.1. Configuration for Tasks

OpenCPU supports multitask processing. Developers only need to simply follow suit to add a record in “*custom_task_cfg.h*” file to define a new task. OpenCPU supports one main task, and maximum TEN subtasks.

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

Developers should avoid calling these functions: *Ql_Sleep()*, *Ql_OS_TakeSemaphore()* and *Ql_OS_TakeMutex()*. These functions will block the task, which will make the task cannot fetch message from the message queue. If the message queue is filled up, the system will automatically reboot unexpectedly.

4.2. Configuration for GPIO

In OpenCPU, there are two ways to initialize GPIOs. One is to configure initial GPIO list in "*custom_gpio_cfg.h*"; the other way is to call GPIO related API (Please refer to ***Chapter 5.7.2***) to initialize after App starts. But the former one is earlier than the latter one on time sequence. The following figure shows the time sequence relationship.

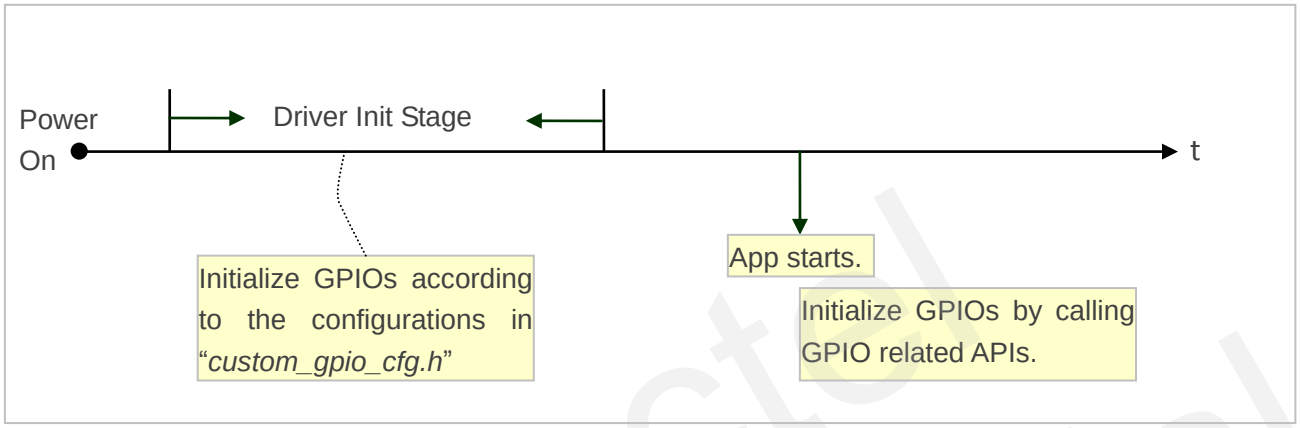

Figure 2: Time Sequence for GPIO Initialization

4.3. Configuration for Customizations

All customization items are configured in TLV (Type-Length-Value) in "*custom_sys_cfg.c*". Developers may change App's features by changing the value.

```
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 Item

Item	Type(T)	Length(L)	Default Value	Possible Value	Description
App Enable	SYS_CONFIG_APP_ENABLE_ID	4	APP_ENABLE	APP_ENABLE APP_DISABLE	App enable config
PWRKEY Pin Config	SYS_CONFIG_PWRKEY_DATA_ID	2	TRUE TRUE	TRUE/FALSE	Power on/off working mode, see ***Chapter 4.3.1***
GPIO for WTD Config	SYS_CONFIG_WATCHDOG_DATA_ID	8	PINNAME_GPIO0	One value of Enum_PinName	GPIO for feeding WTD, see ***Chapter 4.3.2***
Working Mode for Debug Port	SYS_CONFIG_DEBUG_MODE_ID	4	BASIC_MODE	BASIC_MODE ADVANCE_MODE	Application mode or debug mode

4.3.1. Power Key Configuration

```
static const ST_PowerKeyCfg pwrkeyCfg =
{
    TRUE,    //Working mode for power-on on PWRKEY pin
    /*
    Module automatically powers on when feeding a low level to POWER_KEY pin.

    When set to FALSE, the callback that Ql_PwrKey_Register registers will be trigged. Application must call Ql_LockPower() to lock power supply, or module will lose power when the level of PWRKEY pin goes high.
    */

    TRUE,    //Working mode for power-off on PWRKEY pin
    /*
    Module automatically powers off when feeding a low level to POWER_KEY pin.

    When set to FALSE, the callback that Ql_PwrKey_Register registers will be trigged.
    Application may do post processing before switches off the module.
    */
};
```

For example, if the “pwrKeyCfg” is configured as follows.

```
static const ST_PowerKeyCfg pwrkeyCfg =
{
    FALSE,   //Working mode for power-on on PWRKEY pin
    FALSE,   //Working mode for power-off on PWRKEY pin
};
```

When switching on/off the module by feeding a low level to POWER_KEY, the callback in application will be triggered. The example codes are shown below.

```
…
//Register a callback function for pressing POWER KEY.
Ql_PwrKey_Register((Callback_PowerKey_Ind)callback_pwrKey_ind);
…
//Callback definition
void Callback_PowerKey_Hdlr(s32 param1, s32 param2)
{
    Ql_Debug_Trace("<-- Power Key: %s, %s -->\r\n",
        param1==POWER_OFF ? "Power Off":"Power On",
        param2==KEY_DOWN ? "Key Down":"Key Up"
        );
    if (POWER_ON==param1)
    {
        Ql_Debug_Trace("<-- App Lock Power Key! -->\r\n");
        Ql_LockPower();
    }
    else if (POWER_OFF==param1)
    {
        //Post processing before power down
        //...
        //Power down
        Ql_PowerDown();
    }
 }
```

4.3.2. GPIO for External Watchdog

When an external watchdog is adopted to monitor the application, the module has to feed the watchdog in the whole period that the module is in power up, including the boot course, App active course, and App upgrade course.

Table 5: Participants for Feeding External Watchdog

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

Therefore, developers just need to specify which GPIO is designed to feed the external watchdog.

```
static const ST_ExtWatchdogCfg wtdCfg = {
    PINNAME_CTS,     //Specify a pin which connects to the external watchdog, other GPIO is ok.
    PINNAME_END      //Specify another pin for watchdog if needed.
};
```

4.3.3. Debug Port Working Mode Config

The serial debug port (UART2) may work as a common serial port (BASIC_MODE), or a special debug port (ADVANCE_MODE) that can debug some issues during application.

Usually developers don't need to use ADVANCE_MODE without the requirements from support engineer. If needed, please refer to the document ***Catcher_Operation_UGD*** for the usage of the special debug mode.

```
static const ST_DebugPortCfg debugPortCfg = {
    BASIC_MODE          //Set the serial debug port (UART2) to a common serial port.
    //ADVANCE_MODE      //Set the serial debug port (UART2) to a special debug port.
};
```

5 API Functions

5.1. System API

The "*ql_system.h*" file declares system-related APIs. These functions are essential to any customer's applications. Make sure to include the header file.

OpenCPU supports multitasking, message, mutex, semaphore and event mechanism. These interfaces are used for multitask programming. The example "example_multitask.c" in OpenCPU SDK shows the proper usages of these API functions.

5.1.1. Usage

This section introduces some important operations and the API function in system-level programming.

5.1.1.1. Receive Message

Developers can call *Ql_OS_GetMessage* to retrieve a message from the current task's message queue. The message can be a system message, and also can be a customized message.

5.1.1.2. Send Message

Developers can call *Ql_OS_SendMessage* to send messages to other tasks. To send message, developers have to define a message ID. In OpenCPU, user message ID must bigger than 0x1000.

Step 1: Define message ID

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

Step 2: Send 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, and non-signaled when it is owned. Only one task at a time can own a mutex object. For example, to prevent two tasks from writing to shared memory at the same time, each task waits for ownership of a mutex object before executing the code that accesses the memory. After writing to the shared memory, the task releases the mutex object.

Step 1: Create mutex. Developers can call *Ql_OS_CreateMutex* to create a mutex.
Step 2: Get mutex. If developers want to use mutex mechanism for programming, they can call *Ql_OS_TakeMute* to get the specified mutex ID.
Step 3: Give Mutex. Developers can 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 completes a 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 become signaled. The state of a semaphore is set to signaled when its count is greater than zero and non-signaled when its count is zero.

Step 1: Create Semaphore. Developers can call *Ql_OS_CreateSemaphore* to create a semaphore.
Step 2: Get Semaphore. If developers want to use semaphore mechanism for programming, they can call *Ql_OSTakeSemaphore* to get the specified semaphore ID.
Step 3: Give Semaphore. Developers can 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 to signaled by use of the *Ql_OS_SetEvent* function.

5.1.1.6. Backup Critical Data

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

Developers may call *Ql_Userdata_Backup()* to backup data, and call *Ql_Userdata_Read()* to read back data from backup space.

5.1.2. API Functions

5.1.2.1. Ql_Reset

This function resets the system.

- **Prototype**

```
void Ql_Reset(s32 resetType)
```

- **Parameters**

resetType:
[in] Must be 0.

- **Return Value**

None.

5.1.2.2. Ql_Sleep

This function suspends the execution of the current task until the time-out interval elapses. The sleep time should not exceed 500 ms, because if the task is suspended too long, and it may receives too many messages to be crushed.

- **Prototype**

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

- **Parameters**

msec:
[in] The time interval for which execution is to be suspended in milliseconds.

- **Return Value**

None.

5.1.2.3. Ql_GetUID

This function gets the module 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] Point to the buffer which is used to store the UID. Need at least 20 bytes length of buffer.

len:
[in] The “ptrUID” buffer length. The value must be less than or equal the buffer size that “ptrVer” point.

- **Return Value**

If the ptrUID is null, this function will return *QL_RET_ERR_INVALID_PARAMETER*. If this function read 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 no more than 35 characters string, and it ends with ‘\0’.

- **Prototype**

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

- **Parameters**

ptrVer:
[in] Point to the buffer which is used to store the version ID of the core. Need at least 35 bytes of the buffer.

len:
[in] The “ptrVer” buffer length. The value must be less than or equal the buffer size that “ptrVer” point.

- **Return Value**

The return value is the length of version ID of the core if this function succeeds. Otherwise, the return value is an error code. To get extended error information, please refer to ERROR CODES.

5.1.2.5. Ql_GetSDKVer

This function gets the version ID of the SDK. The SDK version ID is no more than 20 characters string, and it is end with ‘\0’.

- **Prototype**

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

- **Parameters**

ptrVer:
[in] Point to the buffer which is used to store the version ID of the SDK. Need at least 20 bytes of the buffer.

len:
[in] The “ptrVer” buffer length. The value must be less than or equal to the buffer size that “ptrVer” point.

- **Return Value**

The return value is the length of version ID if this function succeeds. Otherwise, the return value is an error code. To get extended error information, please refer to ERROR CODES.

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**

None.

- **Return Value**

Number of milliseconds.

5.1.2.7. Ql_OS_GetMessage

This function retrieves a message from the current task's message queue. When there is no message in the task’s message queue, the task is 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] Point to a “ST_MSG” object.

- **Return Value**

QL_RET_OK.

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.

param1:
[in] User data.

param2:
[in] User data.

- **Return Value**

OS_SUCCESS: send message succeeds.

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 return a valid handle also. 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 created mutex, 0 indicates failure.

5.1.2.10. Ql_OS_TakeMutex

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

- **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 the 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 return a valid handle also. 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 max count of semaphore.

- **Return Value**

A handle of created semaphore. 0 indicates failure.

5.1.2.13. Ql_OS_TakeSemaphore

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

- **Prototype**

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

- **Parameters**

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

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

- **Return Value**

OS_SUCCESS: the operation is done successfully.
OS_SEM_NOT_AVAILABLE: the semaphore is unavailable immediately.

5.1.2.14. Ql_OS_CreateEvent

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

- **Prototype**

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

- **Parameters**

evtName:
[in] Event name.

- **Return Value**

An event ID identifies that this event is unique.

5.1.2.15. Ql_OS_WaitEvent

This function waits until the specified type of event is in the signaled state. Developers can specify different types of events for 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. Please refer to "*Enum_EventFlag*".

- **Return Value**

Zero indicates success; nonzero indicates failure.

5.1.2.16. Ql_OS_SetEvent

This function sets the 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. Please refer to *Enum_EventFlag*.

- **Return Value**

Zero indicates success; nonzero indicates failure.

5.1.2.17. Ql_OS_GiveSemaphore

This function releases an instance of the 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_SetLastErrorCode

This function sets error code.

- **Prototype**

```
s32 Ql_SetLastErrorCode(s32 errCode)
```

- **Parameters**

errCode:
[in] Error code.

- **Return Value**

QL_RET_OK: indicates success.
QL_RET_ERR_FATAL: fail to set error code.

5.1.2.19. Ql_GetLastErrorCode

This function retrieves the calling task's last-error code value.

- **Prototype**

```
s32 Ql_GetLastErrorCode(void)
```

- **Parameters**

None.

- **Return Value**

The return value is the calling task's last-error code.

5.1.2.20. Ql_OS_GetCurrenTaskLeftStackSize

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

- **Prototype**

```
u32 Ql_OS_GetCurrenTaskLeftStackSize(void)
```

- **Parameters**

None.

- **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 Code

The frequent error-codes, which APIs in multitask programming could return, 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. Example

1. Mutex example:

```
static int    s_iMutexId = 0;

//Create the 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 after 3 seconds.
    Ql_Sleep(3000);

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

2. Semaphore example:

```
static int s_iSemaphoreId = 0; //Defined 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 indicate the task should
    wait infinitely or return immediately.
    Ql_OS_TakeMutex(s_iSemMutex);
    s_iTestSemNum--; //One semaphore is be used
    Ql_OS_GiveMutex(s_iSemMutex);

    Ql_Sleep(3000);

    //3 seconds later, release the semaphore.
    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 API

OpenCPU module provides time-related APIs including setting local time, getting local time, converting the calendar time into seconds or converting seconds into the calendar time, etc.

5.2.1. Usage

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

5.2.2. API Functions

Time struct 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 example: the time zone of Beijing is East Area 8, timezone=8; the time zone of Washington is West Zone 5, timezone=-5.

5.2.2.1. Ql_SetLocalTime

Set the current local date and time.

- **Prototype**

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

- **Parameter**

datetime:
[in] Point to the ST_Time object.

- **Return Value**

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

5.2.2.2. Ql_GetLocalTime

Get the current local date and time.

- **Prototype**

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

- **Parameter**

dateTime:
[Out] Point to the ST_Time object.

- **Return Value**

If succeeds, return the current local date and time. NULL means failure.

5.2.2.3. Ql_Mktime

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

- **Prototype**

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

- **Parameter**

dateTime:
[in] Point to the ST_Time object.

- **Return Value**

Return the total seconds.

5.2.2.4. Ql_MKTime2CalendarTime

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

- **Prototype**

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

- **Parameter**

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

pOutDateTime:
[Out] Point to the ST_Time object.

- **Return Value**

If succeeds, return the current local date and time. NULL means operation failure.

5.2.3. Example

The following code shows 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 1, 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 1, 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 API

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

5.3.1. Usage

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

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

5.3.2. API Functions

5.3.2.1. Ql_Timer_Register

Register a Common Timer. Each task supports 10 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)
```

- **Parameter**

timerId:
[in] Timer ID. Must ensure that the ID is the only one under openCPU task. Of course, the ID that registered by *Ql_Timer_RegisterFast* also cannot be the same with it.

callback_onTimer:
[Out] Notify the customer when the timer arrives.

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

- **Return Value**

QL_RET_OK: indicates registered successfully.
QL_RET_ERR_PARAM: indicates parameter error.
QL_RET_ERR_INVALID_TIMER: indicates the timer invalid.
QL_RET_ERR_TIMER_FULL: indicates all timers are used up.

5.3.2.2. Ql_Timer_RegisterFast

Register a Fast Timer. Only support one 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)
```

- **Parameter**

timerId:
[in] Timer ID. Must ensure that the ID is not the same with the ID that registered by *Ql_Timer_Register*.

callback_onTimer:
[Out] Notify the customer when the timer arrives.

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

- **Return Value**

QL_RET_OK: indicates registered successfully.
QL_RET_ERR_PARAM: indicates parameter error.
QL_RET_ERR_INVALID_TIMER: indicates the timer invalid.
QL_RET_ERR_TIMER_FULL: indicates all timers are used up.

5.3.2.3. Ql_Timer_Start

Start up the specified timer. When start or stop a specified timer in a task, the task must be the same as the one that registers the timer.

- **Prototype**

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

- **Parameter**

timerId:
[in] Timer ID. The timer ID must be registered.

interval:
[in] Set the interval of the timer, unit: ms.

If you start a Common Timer, the interval must be more than or equal to 1ms. If you start a Fast Timer, the interval must be an integral multiple of 10ms.

autoRepeat:
[in] TRUE or FALSE, indicates that the timer is executed once or repeatedly.

- **Return Value**

QL_RET_OK: indicates started successfully.
QL_RET_ERR_PARAM: indicates parameter error.

QL_RET_ERR_INVALID_TIMER: indicates the timer invalid.
QL_RET_ERR_INVALID_TASK_ID: indicates the current task is not the one that registers the timer.

5.3.2.4. Ql_Timer_Stop

Stop the specified timer. When start or stop a specified timer in a task, the task must be the same as the one that registers the timer.

- **Prototype**

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

- **Parameter**

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

- **Return Value**

QL_RET_OK: indicates stopped successfully.
QL_RET_ERR_PARAM: indicates parameter error.
QL_RET_ERR_INVALID_TIMER: indicates the timer invalid.
QL_RET_ERR_INVALID_TASK_ID: indicates the current task is not the one that registers the timer.

5.3.3. Example

The following code shows how to register a Common Timer and how to 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 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 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 API

Power management contains the power-related operations, such as power down, power key control and low power consumption enabling/disabling.

5.4.1. Usage

5.4.1.1. Power on/off

Developers may call *Ql_PowerDown* function to power off the module when PWRKEY pin has not been short-circuited to ground. And this action will reset the module when PWRKEY pin has been short-circuited to ground.

5.4.1.2. Sleep Mode

The *Ql_ SleepEnable* function can enable the sleep mode of module. The the module enters sleep mode when it is idle.

The timeout of timer, coming call, coming SMS, GPRS data and an interrupt event can wake up the module from sleep mode. The *Ql_SleepDisable* function 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 call this API to power down the module, the module will complete the network anti-registration first. So power off the module will need more time.

- **Prototype**

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

- **Parameters**

pwrDwnType:
[in] Action types of this function.
1=Normal power off

- **Return Value**

None.

5.4.2.2. Ql_LockPower

When getting the control right of power key, application must call *Ql_LockPower* to lock power supply, or the module will lose power when the level of PWRKEY pin goes high. Please also refer to ***Chapter 4.3.1***.

- **Prototype**

```
void Ql_LockPower(void);
```

- **Parameters**

None.

- **Return Value**

None.

5.4.2.3. Ql_PwrKey_Register

This function registers the callback for PWRKEY indication. The callback function will be triggered when the power key is pressed down or released (including power on and power off). The configuration for power key in sys_config.c should be set to FALSE. Or else, the callback will not be triggered. Please refer to ***Chapter 4.31.***

- **Prototype**

```
s32 Ql_PwrKey_Register(Callback_PowerKey_Ind callback_pwrKey)
typedef void    (*Callback_PowerKey_Ind)(s32 param1, s32 param2)
```

- **Parameters**

Callback_pwrKey:
[in] Callback function for PWRKEY indication.

param1:
[Out] One value of *Enum_PowerKeyOpType*.

param2:
[Out] One value of *Enum_KeyState*.

- **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 ERROR CODES.

5.4.2.4. Ql_SleepEnable

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

- **Prototype**

```
s32 Ql_ SleepEnable()
```

- **Parameters**

None.

- **Return Value**

QL_RET_OK: indicates this function succeeds.
Ql_RET_NOT_SUPPORT: function not supported in currently used version.

5.4.2.5. Ql_SleepDisable

This function disables the sleep mode of module.

- **Prototype**

```
s32 Ql_SleepDisable()
```

- **Parameters**

None.

- **Return Value**

QL_RET_OK: indicates this function succeeds.
Ql_RET_NOT_SUPPORT: this function is not supported.

5.4.3. Example

The following sample codes show how to enter into and quit 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 into sleep mode
      }else{
             SYS_DEBUG( DBG_Buffer,"DTR set to high=%d    Sleep \r\n", level);
            Ql_SleepEnable(); //Quit from sleep mode
      }
}
```

5.5. Memory API

OpenCPU O.S supports dynamic memory management. *Ql_MEM_Alloc* and *QL_MEM_Free* functions are used to allocate and release the dynamic memory.

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

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

5.5.1. Usage

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

Step 2: If the memory block is not needed any more, please call *Ql_MEM_Free()* to free the memoryblock which 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 the specified size in the memory heap.

- **Prototype**

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

- **Parameter**

Size:
[in] Number of bytes of memory 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 which is allocated by *Ql_MEM_Alloc*.

- **Prototype**

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

- **Parameters**

Ptr:
[in] Previously allocated memory block to be free.

- **Return Value**

None.

5.5.3. Example

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

```
char *pch=NULL;

//Allocate 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 memory
```

```
Ql_MEM_Free(pch);
pch=NULL;
```

5.6. File System API

OpenCPU supports user file system, and provides a set of complete API functions to create, access and delete files and directories. This section describes these APIs and the usage.

The storage can be flash (UFS) and RAM (RAM file). The RAM file doesn't support directory structure.

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 doesn't support directory structure. The customers can select the storage location according to their own needs. When you want to create/open a file or directory, you must use a relative path. For example, if you want to create a file in the root of the UFS, you can set like this, such as “filename.ext”.

- The *Ql_FS_GetTotalSpace* function is used to obtain the amount of total space on Flash.
- The *Ql_FS_GetFreeSpace* function is used to obtain the amount of free space on Flash.
- The *Ql_FS_GetSize* function is used to get the size, in bytes, of the specified file.
- The *Ql_FS_Open* function is used to create or open a file. You must define the file's opening and access mode. If you want to know its usage, please refer to the detailed descriptions of this function.
- The *Ql_FS_Read* and *Ql_FS_Write* functions are used to read or write a file, you must ensure that the file has been opened.
- The *Ql_FS_Seek* and *Ql_FS_GetFilePosition* functions are used to set or get the position of the file pointer, you must ensure that the file has been opened.
- The *Ql_FS_Truncate* function is used to truncate the specified file to zero length.
- The *Ql_FS_Delete* and *Ql_FS_Check* functions are used to delete or check a file.
- The *Ql_FS_CreateDir*, *Ql_FS_DeleteDir* and *Ql_FS_CheckDir* functions are used to create, delete or check a specified directory.
- The *Ql_FS_FindFirst*, *Ql_FS_FindNext* and *Ql_FS_XDelete* functions are used to traverse all files and directories in the specified directory. These three functions are usually used together.
- The *Ql_FS_XDelete* function is multi-functional. It can be used to delete a specified file or an empty directory. You can also delete all files and directories in the specified directory by recursive way.
- The *Ql_FS_XMove* function is used to move or copy a file or folder.
- The *Ql_FS_Format* function is used to format the UFS.

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 name of the file. The name is limited to 252 characters. You must use a relative path, such as "filename.ext" or "dirname\filename.ext".

flag:
[in] A u32 that defines the file's opening and access mode. The possible values are shown as follow:

QL_FS_READ_WRITE: can read and write.
QL_FS_READ_ONLY:read only.
QL_FS_CREATE: opens the file, if it exists. If the file does not exist, the function creates the file.
QL_FS_CREATE_ALWAYS: creates a new file. If the file has already exisited, the function overwrites the file and clears the existing attributes.

- **Return Value**

If the function succeeds, the return value specifies a file handle. If the function fails, the return value is an error code.

QL_RET_ERR_PARAM: indicates parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: indicates filename too long.
QL_RET_ERR_FILEOPENFAILED: indicates open file failed.

5.6.2.2. Ql_FS_OpenRAMFile

This function opens or creates a file with a specified name in the RAM. You need to add prefix "RAM:" in the front of the file. You can create 15 files at most.

- **Prototype**

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

- **Parameters**

lpFileName:
[in] The file name. The name is limited to 252 characters. You must use a relative path, such as “RAM: filename.ext”.

flag:
[in] A u32 that defines the file’s opening and access mode. The possible values are shown as follow:

QL_FS_READ_WRITE: can read and write.
QL_FS_READ_ONLY: read only.
QL_FS_CREATE: opens the file, if it exists. If the file does not exist, the function creates the file.
QL_FS_CREATE_ALWAYS: creates a new file. If the file has already existed, the function overwrites the file and clears the existing attributes.

ramFileSize:
[in] The size of the specified file that you want to create.

- **Return Value**

If the function succeeds, the return value specifies a file handle. If the function fails, the return value is an error code.

QL_RET_ERR_PARAM: indicates parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: indicates filename too long.
QL_RET_ERR_FILEOPENFAILED: indicates open file failed.

5.6.2.3. Ql_FS_Read

Read data from the specified file, starting at the position indicated by the file pointer. After the reading operation has been 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] A handle to the file to be read, which is the return value of the function *Ql_FS_Open*.

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

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

numberOfBytesRead:
[Out] The number of bytes has been read. Sets this value to zero before doing taking action or checking errors

- **Return Value**

QL_RET_OK: success.
QL_RET_ERR_FILEREADFAILED: read file failed.

5.6.2.4. Ql_FS_Write

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

- **Prototype**

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

- **Parameters**

fileHandle:
[in] A handle to the file to be written, which is the return value of the function *Ql_FS_Open*.

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

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

numberOfBytesToWritten:
[Out] Point to the number of bytes written by the function call.

- **Return Value**

QL_RET_OK: success.
QL_RET_ERR_FILEDISKFULL: file disk is full.
QL_RET_ERR_FILEWRITEFAILED: write file failed.

5.6.2.5. Ql_FS_Seek

This function repositions the pointer in the previously open file.

- **Prototype**

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

- **Parameters**

fileHandle:
[in] File handle, which is the return value of the function *Ql_FS_Open*.

offset:
[in] Number of bytes to move the file pointer.

whence:
[in] Pointer movement mode. 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: success.
QL_RET_ERR_FILESEEKFAILED: file seek failed.

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] File handle, which is the return value of the function *Ql_FS_Open*.

- **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: fail to operate 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 the function *Ql_FS_Open*.

- **Return Value**

QL_RET_OK: success.
QL_RET_ERR_FILEFAILED: fail to operate file.

5.6.2.8. Ql_FS_Flush

Force any 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 the function *Ql_FS_Open*.

- **Return Value**

None.

5.6.2.9. Ql_FS_Close

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 the function *Ql_FS_Open*.

- **Return Value**

None.

5.6.2.10. Ql_FS_GetSize

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

- **Prototype**

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

- **Parameters**

lpFileName:
[in] The name of the file. The name is limited to 252 characters. You must use a relative path, such as "filename.ext" or "dirname\filename.ext".

- **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: parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: filename too long.
QL_RET_ERR_FILEFAILED: fail to operate file.

5.6.2.11. Ql_FS_Delete

This function deletes an existing file.

- **Prototype**

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

- **Parameters**

lpFileName:
[in] The name of the file. The name is limited to 252 characters. You must use a relative path, such as "filename.ext" or "dirname\filename.ext".

- **Return Value**

QL_RET_OK: success.
QL_RET_ERR_PARAM: parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: filename too long.
QL_RET_ERR_FILEFAILED: fail to operate file.

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 is limited to 252 characters. You must use a relative path, such as "filename.ext" or "dirname\filename.ext".

- **Return Value**

QL_RET_OK: success.
QL_RET_ERR_PARAM: parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: filename too long.
QL_RET_ERR_FILEFAILED: fail to operate file.
QL_RET_ERR_FILENOTFOUND: file 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 is limited to 252 characters. You must use a relative path, such as "filename.ext" or "dirname\filename.ext".

newLpFileName:
[in] The new name of the file. The new name is different from the existing names. The name is limited to

252 characters. You must use a relative path, such as “filename.ext”, “dirname\filename.ext”.

- **Return Value**

QL_RET_OK: success.
QL_RET_ERR_PARAM: parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: filename too long.
QL_RET_ERR_FILEFAILED: fail to operate file.

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 is limited to 252 characters. You must use a relative path, such as “dirname1” or “dirname1\dirname2”.

- **Return Value**

QL_RET_OK: success.
QL_RET_ERR_PARAM: parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: filename too long.
QL_RET_ERR_FILEFAILED: fail to operate file.

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 is limited to 252 characters. You must use a relative path, such as “dirname1” or “dirname1\dirname2”.

- **Return Value**

QL_RET_OK: success.
QL_RET_ERR_PARAM: parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: filename too long.
QL_RET_ERR_FILEFAILED: fail to operate file.

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 is limited to 252 characters. You must use a relative path, such as “dirname1” or “dirname1\dirname2”.

- **Return Value**

QL_RET_OK: success.
QL_RET_ERR_PARAM: parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: filename too long.
QL_RET_ERR_FILEFAILED: fail to operate file.
QL_RET_ERR_FILENOTFOUND: file not found.

5.6.2.17. Ql_FS_FindFirst

Search a directory for a file or subdirectory whose name matches the specified file name.

- **Prototype**

```
s32 Ql_FS_FindFirst(char *lpPath, char *lpFileName, u32 fileNameLength, u32 *fileSize, bool *isDir)
```

- **Parameters**

lpPath:
[in] Pointer to a null-terminated string that specifies a valid directory or path.

lpFileName:
[in] Pointer to a null-terminated string that specifies a valid file name, which can contain wildcard characters, such as * and ?.

fileNameLength:
[in] The maximum number of bytes to be received of the name.

fileSize:
[Out] A pointer to the variable which represents the size specified by the file.

isDir:
[Out] A pointer to the variable which represents the type specified by the file.

- **Return Value**

If the function succeeds, the return value is a search handle that can be used in a subsequent call to the *Ql_FindNextFile* or *Ql_FindClose* function.

If the function fails, the return value is an error code:

QL_RET_ERR_PARAM: parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: filename too long.
QL_RET_ERR_FILEFAILED: fail to operate file.
QL_RET_ERR_FILENOMORE: no more files.

5.6.2.18. Ql_FS_FindNext

This function is used to find the next file continuously according to the handle which is a return value of *Ql_FS_FindFirst* function.

- **Prototype**

```
s32 Ql_FS_FindNext(s32 handle, char *lpFileName, u32 fileNameLength, u32 *fileSize, bool *isDir)
```

- **Parameters**

handle:
[in] The handle is a return value of *Ql_FS_FindFirst* function.

lpFileName:
[in] Pointer to a null-terminated string that specifies a valid file name, which can contain wildcard characters, such as * and ?.

fileNameLength:
[in] The maximum number of bytes to be received of the name.

fileSize:
[Out] A pointer to the variable which represents the size specified by the file.

isDir:
[Out] A pointer to the variable whose type is specified by the file.

- **Return Value**

QL_RET_OK: success.
QL_RET_ERR_PARAM: parameter error.
QL_RET_ERR_FILEFAILED: fail to operate file.
QL_RET_ERR_FILENOMORE: file not found.

5.6.2.19. Ql_FS_FindClose

This function closes the specified search handle.

- **Prototype**

```
void Ql_FS_FindClose(s32 handle)
```

- **Parameters**

handle:
[in] Find handle, returned by a previous call of the *Ql_FS_FindFirst* function.

- **Return Value**

None.

5.6.2.20. Ql_FS_XDelete

This function deletes a file or directory.

- **Prototype**

```
s32 Ql_FS_XDelete(char* lpPath, u32 flag)
```

- **Parameters**

lpPath:
[in] File path to be deleted.

flag:
[in] A u32 that defines the file's opening and access mode.

The possible values are shown as follow:

QL_FS_FILE_TYPE
QL_FS_DIR_TYPE
QL_FS_RECURSIVE_TYPE

- **Return Value**

QL_RET_OK: success.
QL_RET_ERR_PARAM: parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: filename too long.
QL_RET_ERR_FILENOTFOUND: file not found.
QL_RET_ERR_PATHNOTFOUND: path not found.
QL_RET_ERR_GET_MEM: fail to get memory.
QL_RET_ERR_GENERAL_FAILURE: general faillure.

5.6.2.21. Ql_FS_XMove

This function provides a facility to move or copy a file or folder.

- **Prototype**

```
s32 Ql_FS_XMove(char* lpSrcPath, char* lpDestPath, u32 flag)
```

- **Parameters**

lpSrcPath:
[in] Source path to be moved or copied.

lpDestPath:
[in] Destination path.

flag:
[in] A u32 that defines the file's opening and access mode.

The possible values are shown as follows:

QL_FS_MOVE_COPY
QL_FS_MOVE_KILL
QL_FS_MOVE_OVERWRITE

- **Return Value**

QL_RET_OK: success.
QL_RET_ERR_PARAM: parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: filename too long.
QL_RET_ERR_FILENOTFOUND: file not found.

QL_RET_ERR_PATHNOTFOUND: path not found.
QL_RET_ERR_GET_MEM: fail to get memory.
QL_RET_ERR_FILE_EXISTS: file existed.
QL_RET_ERR_GENERAL_FAILURE: general failture.

5.6.2.22. Ql_FS_ GetFreeSpace

This function obtains the amount of free space on Flash.

- **Prototype**

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

- **Parameters**

storage:
[in] The type of storage, which is 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_UNKOWN: unkown error.

5.6.2.23. Ql_FS_GetTotalSpace

This function obtains the amount of total space on Flash.

- **Prototype**

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

- **Parameters**

storage:
[in] The type of storage, which is one value of *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_UNKOWN: unkown error.

5.6.2.24. Ql_FS_Format

This function formats the UFS.

- **Prototype**

```
s32 Ql_FS_Format(u32 storage)
```

- **Parameters**

storage:
[in] The format storage, which is one value of *Enum_FSStorage*.

- **Return Value**

QL_RET_OK: success.
QL_RET_ERR_PARAM: parameter error.
QL_RET_ERR_FILENAMETOOLENGTH: filename too long.
QL_RET_ERR_FILENOTFOUND: file not found.
QL_RET_ERR_PATHNOTFOUND: path not found.
QL_RET_ERR_GET_MEM: fail to get memory.
QL_RET_ERR_GENERAL_FAILURE: general failture.

5.6.3. Example

The following codes show how to use the file system functions.

```
#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.
    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.
    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);

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

//The file "test.txt" rename 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);

//Creates 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 dir.
ret=Ql_FS_CreateDir(DIR_NAME);
Ql_Debug_Trace("Ql_FS_CreateDir(%s)=%d\r\n", DIR_NAME, ret);

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

//Check the dir. exist or not
ret=Ql_FS_DeleteDir(DIR_NAME);
Ql_Debug_Trace("Ql_FS_DeleteDir(%s)=%d\r\n", DIR_NAME, ret);

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

//List all files and directories under the root of the UFS..
Ql_memset(strBuf,0,100);
findfile=Ql_FS_FindFirst(LPPATH, strBuf, 100, &filesize, &isdir);
Ql_Debug_Trace("\r\nLater:strBuf=[%s]",strBuf);
if(findfile < 0)
{
    Ql_Debug_Trace("Failed Ql_FS_FindFirst(%s)=%d\r\n", LPPATH, findfile);
}else{
    Ql_Debug_Trace("Sueecss Ql_FS_FindFirst(%s)\r\n", LPPATH);
}
ret=findfile;
while(ret >=0)
{
```

```
    Ql_Debug_Trace("filesize(%d),isdir(%d),Name(%s)\r\n", filesize, isdir, strBuf);
    ret=Ql_FS_FindNext(findfile, strBuf, 100, &filesize, &isdir);
    if(ret !=QL_RET_OK)
    break;
}
Ql_FS_FindClose(findfile);

//Copy the file "test.txt" to the dir "DIR\\".
ret=Ql_FS_XMove(FILE_NAME, DIR_NAME, QL_FS_MOVE_COPY);
Ql_Debug_Trace("Ql_FS_XMove(%s.%s,%x)=%d\r\n", FILE_NAME, DIR_NAME,
QL_FS_MOVE_COPY, ret);

//List all files and directories under the dir "DIR\\".
Ql_memset(strBuf,0,100);
findfile=Ql_FS_FindFirst(LPPATH2, strBuf, 100, &filesize, &isdir);
Ql_Debug_Trace("\r\nLater:strBuf=[%s]",strBuf);
if(findfile<0)
{
    Ql_Debug_Trace("Failed Ql_FS_FindFirst(%s)=%d\r\n", LPPATH2, findfile);
}else{
    Ql_Debug_Trace("Sueecss Ql_FS_FindFirst(%s)\r\n", LPPATH2);
}

ret=findfile;
while(ret>=0)
{
    Ql_Debug_Trace("filesize(%d),isdir(%d),Name(%s)\r\n", filesize, isdir, strBuf);
    ret=Ql_FS_FindNext(findfile, strBuf, 100, &filesize, &isdir);
    if(ret !=QL_RET_OK)
        break;
}
Ql_FS_FindClose(findfile);

//Delete all files and directories under the root of the UFS by recursive way.
ret=Ql_FS_XDelete(XDELETE_PATH,QL_FS_FILE_TYPE
    |QL_FS_DIR_TYPE|QL_FS_RECURSIVE_TYPE);
Ql_Debug_Trace("\r\nQl_FS_XDelete(%s,%x)=%d\r\n",XDELETE_PATH,
    QL_FS_RECURSIVE_TYPE, ret);

Ql_memset(strBuf,0,100);
Findfile=Ql_FS_FindFirst(LPPATH, strBuf, 100, &filesize, &isdir);
Ql_Debug_Trace("Later:strBuf=[%s]",strBuf);
if(findfile < 0)
{
```

```
        Ql_Debug_Trace("Failed Ql_FS_FindFirst(%s)=%d\r\n", LPPATH, findfile);
    }else{
        Ql_Debug_Trace("Sueecss Ql_FS_FindFirst(%s)\r\n", LPPATH);
    }
    ret=findfile;
    while(ret>=0)
    {
        Ql_Debug_Trace("filesize(%d),isdir(%d),Name(%s)\r\n", filesize, isdir, strBuf);
        ret=Ql_FS_FindNext(findfile, strBuf, 100, &filesize, &isdir);
        if(ret !=QL_RET_OK)
            break;
     }
     Ql_FS_FindClose(findfile);
}
```

5.7. Hardware Interface API

5.7.1. UART

5.7.1.1. UART Overview

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

One of the physical UART ports has hardware handshaking function, and others have three-wire interfaces.

OpenCPU provides two virtual UART ports that are used for communication between App and Core. These virtual ports are designed according to the features of physical serial interface. 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 for UARTs is shown below:

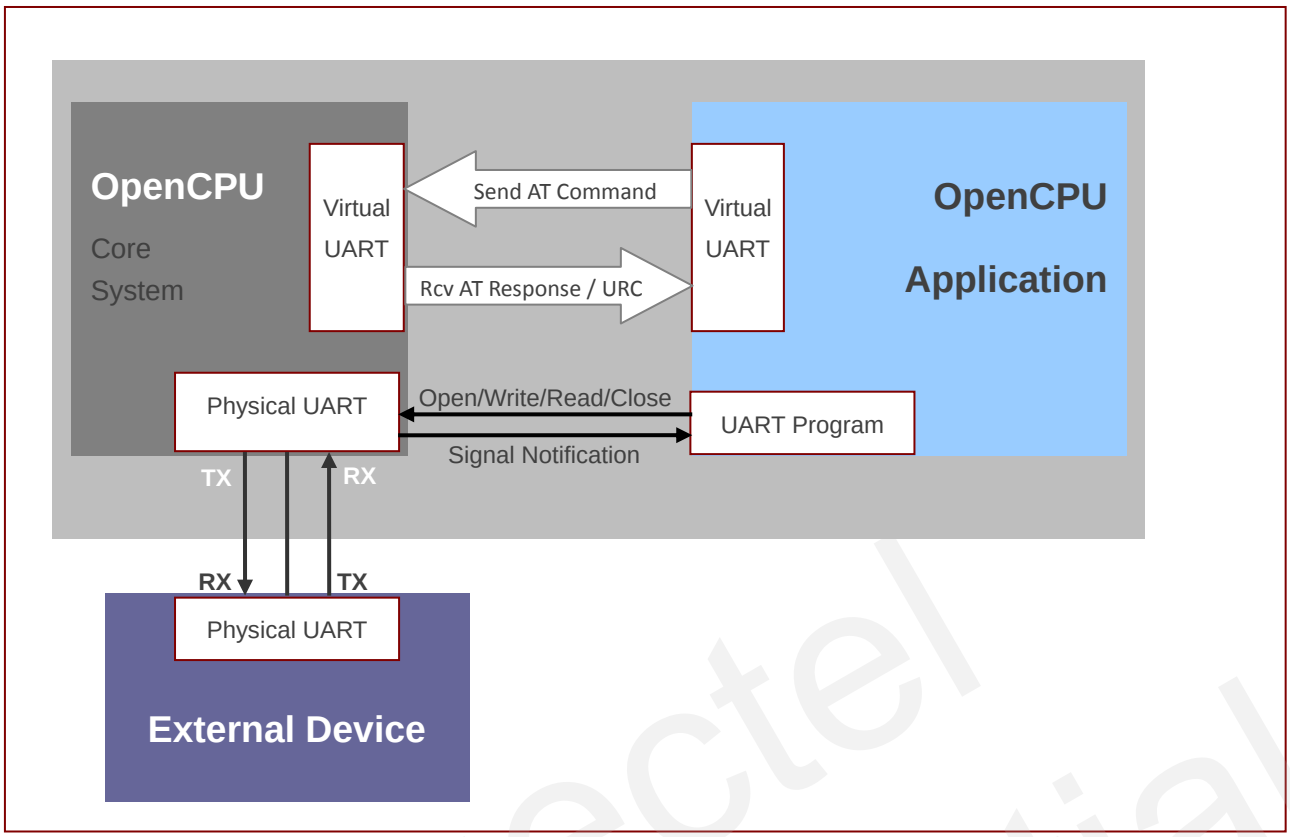

Figure 3: The Working Chart of UART

5.7.1.2. UART Usage

For physical UART or virtual UART initialization and usage, you can accomplish by following simple steps.

Step 1: Call *QI_UART_Register* to register the UART's callback function.
Step 2: Call *QI_UART_Open* to open the special UART port.
Step 3: Call *QI_UART_Write* to write data to the specified UART port. When the number of bytes actually sent is less than that to send, application should stop sending data, and application will receive an event EVENT_UART_READY_TO_WRITE later in callback function. After receiving this event application can continue to send data, and previously unsent data should be resent.
Step 4: In the callback function, deal with the UART's notification. If the notification type is EVENT_UART_READY_TO_READ, developers should 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. QI_UART_Register

This function registers the callback function for the the specified serial 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] The pointer of the UART callback function.

event:
[Out] Indication the event type of UART call back, 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 indicats the related pin's current level otherwise this parameter has no meaning, just ignore it.

customizePara:
[in] Customized parameter. If not used, just set 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 ERROR CODES.

5.7.1.3.2. Ql_UART_Open

This function opens a specified UART port with the specified flow control mode. Which task calls this function, which task will own the specified UART port.

- **Prototype**

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

```
typedef enum {
    FC_NONE=1,    //None Flow Control
    FC_HW,        //Hardware Flow Control
```

```
    FC_SW           //Software Flow Control
} Enum_FlowCtrl;
```

- **Parameters**

port:
[in] Port name.

baudrate:
[in] The baud rate of the UART to be open.
The physical UART's baud rate supports 75, 150, 300, 600, 1200, 2400, 4800, 7200, 9600, 14400, 19200, 28800, 38400, 57600, 115200, 230400, 460800. The parameter does not take effect on the VIRTUAL_PORT1 and VIRTUAL_PORT2, so just set to 0.

flowCtrl:
[in] Please refer to *Enum_flowCtrl*, for the physical UART ports. Only UART_PORT1 supports hardware flow control (FC_HW).

- **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 ERROR CODES.

5.7.1.3.3. Ql_UART_OpenEx

This function opens a specified UART port with the specified DCB parameters. Which task calls this function, which task 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 setting, including baud rate, data bits, stop bits, parity, and flow control. Only physical serial port1 (UART_PORT1) supports hardware flow control. And this parameter doesn't take effect on the VIRTUAL_PORT1 and VIRTUAL_PORT2, so just set 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 ERROR CODES.

5.7.1.3.4. Ql_UART_Write

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

- **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 1023 bytes which cannot be modified programmatically in application.

- **Return Value**

Number of bytes actually written. If this function fails to write data, a negative number will be returned. To get extended information, please refer to ERROR CODES.

5.7.1.3.5. Ql_UART_Read

This function reads data from the specified UART port. When the UART callback is invoked, and the notification is EVENT_UART_READY_TO_READ, developers should 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 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] Point to buffer for the read data.

readLen:
[in] The length of the data to be read. The max data length of the receive buffer for physical UART buffer is 3584 bytes, and 1023 bytes for virtual UART. The buffer size cannot be modified programmatically in application.

- **Return Value**

Number of bytes actually read. If 'readLen' equal to the actual read length, users needs continue to read the UART until the actual read length is less than the 'readLen'. To get extended information Please refer to ERROR CODES.

5.7.1.3.6. Ql_UART_SetDCBConfig

This function sets the parameters of the specified UART port. This function 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,    //None Flow Control
    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] The pointer to the UART DCB struct. Including baud rate, databits, stopbits 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 ERROR CODES.

5.7.1.3.7. Ql_UART_GetDCBConfig

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

- **Prototype**

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

- **Parameter**

port:
[in] Port name.

dcb:
[in] The specified UART port's current DCB configration parameters, including baud rate, databits, stopbits 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 ERROR CODES.

5.7.1.3.8. Ql_UART_ClrRxBuffer

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

- **Prototype**

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

- **Parameter**

port:
[in] Port name.

- **Return Value**

None.

5.7.1.3.9. Ql_UART_ClrTxBuffer

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

- **Prototype**

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

- **Parameter**

port:
[in] Port name.

- **Return Value**

None.

5.7.1.3.10. Ql_UART_GetPinStatus

This function gets the pin status (includes RI, DCD, DTR) of the virtual UART port. It does not work for the physical UART ports

- **Prototype**

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

```
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 UART.
UART_PIN_DCD,           //DCD read operator is only valid on the virtual UART.
                        //DCD set operatir is invalid both on virtual and physical UART.
} Enum_UARTPinType;
```

- **Parameters**

port:
[in] Virtual UART port name.

pin:
[in] Pin name, one value of *Enum_UARTPinType*.

- **Return Value**

If >=0, indicates success, and the returned special pin level value. 0: low level, 1: high level.
If <=0, indicates failure.

5.7.1.3.11. Ql_UART_SetPinStatus

This function sets the pin level status of the virtual UART port. It doesn't 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: low level, 1: 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 ERROR CODES.

5.7.1.3.12. Ql_UART_SendEscap

This function notifies the virtual serial port to quit 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 ERROR CODES.

5.7.1.3.13. Ql_UART_Close

This function closes the specified UART port.

- **Prototype**

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

- **Parameter**

port:
[in] Port name.

- **Return Value**

None.

5.7.1.4. Example

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

```
//Write the call back function, for dealing with the UART notifications.
static void CallBack_UART_Hdlr(Enum_SerialPort port, Enum_UARTEventType msg, bool level, void*
customizedPara);    //Call back
{
 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 write 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 call back function
s32 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 21 I/O pins that can be designed for general purpose I/O. All pins can be accessed under

OpenCPU by API functions.

5.7.2.2. GPIO List

Table 6: Multiplexing Pins

PIN No	PIN NAME	RESET	MODE1	MODE2	MODE3	MODE4
7	PINNAME_SD_CMD	I/PD	SD_CMD	GPIO		
8	PINNAME_SD_CLK	I/PD	SD_CLK	GPIO		
9	PINNAME_SD_DATA	I/PD	SD_DATA	GPIO		
10	PINNAME_SIM2_CLK	I/PD	SIM2_CLK	GPIO		
11	PINNAME_SIM2_DATA	I/PD	SIM2_DATA	GPIO		
12	PINNAME_SIM2_RST	I/PD	SIM2_RST	GPIO		
35	PINNAME_RI	I/PD	RI	GPIO	I^2C_SCL	
36	PINNAME_DCD	I/PD	DCD	GPIO	I^2C_SDA	
37	PINNAME_DTR	I/PD	DTR	GPIO	EINT	SIM_PRESENCE
38	PINNAME_CTS	I/PU	CTS	GPIO	EINT	
39	PINNAME_RTS	I/PU	RTS	GPIO		
47	PINNAME_NETLIGHT	I/PD	NETLIGHT	GPIO	PWM_OUT	EINT
57	PINNAME_GPIO0	I/PD	GPIO			
58	PINNAME_GPIO1	I/PD	GPIO			
59	PINNAME_PCM_CLK	HO/-	PCM_CLK	GPIO	SPI_CS	
60	PINNAME_PCM_OUT	I/PD	PCM_OUT	GPIO	SPI_MOSI	
61	PINNAME_PCM_ SYNC	I/PD	PCM_SYNC	GPIO	SPI_MISO	
62	PINNAME_PCM_IN	I/PU	PCM_IN	GPIO	SPI_CLK	
63	PINNAME_GPIO2	I/PD	GPIO			
64	PINNAME_GPIO3	I/PD	GPIO			
65	PINNAME_GPIO4	I/PD	GPIO			

- The “MODE1” defines the original status of pin in standard module.
- “RESET” column defines the default status of every pin after system powers on.
- “I” means input.
- “O” means output.
- “HO” means high output.
- “PU” means internal pull-up circuit.
- “PD” means internal pull-down circuit.
- “EINT” means external interrupt input.
- “PWM_OUT” means PWM output function.

NOTES

1. If PINNAME_SD_CMD, PINNAME_SD_CLK and PINNAME_SD_DATA are designed as SD interface, please don’t configure these pins in customer application.
2. If PINNAME_SIM2_DATA, PINNAME_SIM2_RST, PINNAME_SIM2_CLK are designed as SIM card interface, please don’t configure these pins in customer application.

5.7.2.3. GPIO Initial Configuration

In OpenCPU, there are two ways to initialize GPIOs. One is to configure initial GPIO list in “*custom_gpio_cfg.h*”, please refer to ***Chapter 4.3***; the other way is to call GPIO related API to initialize after App starts.

The following codes show the PINNAME_NETLIGHT, PINNAME_PCM_IN and PINNAME_PCM_OUT pins initial Configuration in “*custom_gpio_cfg.h*” file.

```
    /*------------------------------------------------------------------------------------------------------------------------
    { Pin Name              |          Direction        |        Level          |    Pull Selection          }
    *-----------------------------------------------------------------------------------------------------------------------*/
#if 1//If needed, config GPIOs here
GPIO_ITEM(PINNAME_NETLIGHT,    PINDIRECTION_OUT,    PINLEVEL_LOW,    PINPULLSEL_PULLDOWN)
GPIO_ITEM(PINNAME_PCM_IN,      PINDIRECTION_OUT,    PINLEVEL_LOW,    PINPULLSEL_PULLDOWN)
GPIO_ITEM(PINNAME_PCM_OUT,     PINDIRECTION_OUT,    PINLEVEL_LOW,    PINPULLSEL_PULLUP)
#else if 0
…
#endif
```

5.7.2.4. GPIO Usage

The following shows how to use the multifunctional GPIOs:

Step 1: GPIO initialization. Call *Ql_GPIO_Init* function sets the specified pin as the GPIO function, and

initializes the configurations, including direction, level and pull selection.

Step 2: GPIO control. When the pin is initialized as GPIO, the developers can call the GPIO related APIs to change the GPIO level.

Step 3: Release the pin. If you don't want use this pin no longer, and need to use this pin for other purposes (such as PWM, EINT), you must 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 the specified pin, and initializes the configurations, including direction, level and pull selection.

- **Prototype**

```
s32 Ql_GPIO_Init(PinName pinName,PinDirection dir,PinLevel    level ,PinPullSel pullsel)
```

- **Parameters**

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

dir:
[in] The initial direction of GPIO, one value of *Enum_PinDirection*.

pullsel:
[in] The initial level of GPIO, one value of *Enum_PinLevel*.

level:
[in] Pull selection, one value of *Enum_PinPullSel*.

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates this function fails.

5.7.2.5.2. Ql_GPIO_GetLevel

This function gets the level of the specified GPIO.

- **Prototype**

```
s32 Ql_GPIO_GetLevel(PinName pinName)
```

- **Parameters**

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

- **Return Value**

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

5.7.2.5.3. Ql_GPIO_SetLevel

This function sets the level of the specified GPIO.

- **Prototype**

```
s32 Ql_GPIO_SetLevel(PinName pinName, 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: this function succeeds. Negative integer indicates this function fails.

5.7.2.5.4. Ql_GPIO_GetDirection

This function gets the direction of the specified GPIO.

- **Prototype**

```
s32 Ql_GPIO_GetDirection(PinName pinName)
```

- **Parameters**

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

- **Return Value**

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

5.7.2.5.5. Ql_GPIO_SetDirection

This function sets the direction of the specified GPIO.

- **Prototype**

```
s32 Ql_GPIO_SetDirection(PinName pinName,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: this function succeeds. Negative integer indicates this function fails.

5.7.2.5.6. Ql_GPIO_GetPullSelection

This function gets the pull selection of the specified GPIO.

- **Prototype**

```
s32 Ql_GPIO_GetPullSelection(PinName pinName)
```

- **Parameters**

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

- **Return Value**

Return the pull selection of the specified GPIO, one value of *Enum_PinPullSel*.

5.7.2.5.7. Ql_GPIO_SetPullSelection

This function sets the pull selection of the specified GPIO.

- **Prototype**

```
s32 Ql_GPIO_SetPullSelection(PinName pinName,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: this function succeeds. Negative integer indicates this function fails.

5.7.2.5.8. Ql_GPIO_Uninit

This function releases the specified GPIO which was initialized by calling *Ql_GPIO_Init* previously. After releasing, the GPIO can be used for other purpose.

- **Prototype**

```
s32 Ql_GPIO_Uninit(PinName pinName)
```

- **Parameters**

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

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates that this function fails.

5.7.2.6. Example

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

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

OpenCPU module has three external interrupt pins, please refer to ***Chapter 5.7.2.2*** for details. The interrupt trigger mode just supports level-triggered mode. The software debounce for external interrupt sources in order to minimize the possibility of false activations. External interrupt have higher priority, so frequent interruption is not allowed. It's strongly recommended that the interrupt frequency is not more than 2, and too frequent interrupt will cause that other tasks cannot be scheduled, which probably leads to unexpected exception.

NOTE

The interrupt response time is 50ms by default, and can be re-programmed to a bigger value in OpenCPU. However, it's strongly recommended that the interrupt frequency cannot 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 interruption function:

Step 1: Register an external interrupt function. You must choose one external interrupt pin and use *Ql_EINT_Register* (or *Ql_EINT_RegisterFast*) API to register an interrupt handler function.
Step 2: Initialize the interrupt configurations. Call *Ql_EINT_Init* function to config the software debounce time, set level-triggered interrupt mode.
Step 3: Interrupt handle. The interrupt callback function will be called if the level has changed. Developers can process something in the handler.
Step 4: Mask the interrupt. When you do not want external interrupt you can use the *Ql_EINT_Mask* function to disable the external interrupt, and you can call the *Ql_EINT_Unmask* function to enable the external interrupt.
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(PinName eintPinName, Callback_EINT_Handle callback_eint,void* customParam)
typedef void (*Callback_EINT_Handle)(PinName eintPinName, 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 to NULL.

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates that 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 don't add any task schedule in the interrupt handler which cannot consume much CPU time. Or else, system exceptions or resetting may be caused.

- **Prototype**

```
s32 Ql_EINT_RegisterFast(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.

pinLevel:
[in] The EINT pin level value, one value of *Enum_PinLevel.*

customParam:
[in] Customized parameter, if not used, just set to NULL.

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates this function fails.

5.7.3.3.3. Ql_EINT_Init

Initialize an external interrupt function.

- **Prototype**

```
s32 Ql_EINT_Init(PinName eintPinName,EintType eintType,u32 hwDebounce,u32  swDebounce, bool autoMask)
```

- **Parameters**

eintPinName:
[in] EINT pin name, one value of *Enum_PinName* that has the interrupt function.

eintType:
[in] Interrupt type, level-triggered or edge-triggered. Now, only level-triggered interrupt is supported.

hwDebounce:
[in] Hardware debounce. Unit: in 10ms. Not supported now.

swDebounce:
[in] Software debounce. Unit: in 10ms. The minimum value for this parameter is 5, which means the minimum software debounce time is 5*10ms=50ms.

autoMask:
[in] Whether auto mask the external interrupt after the interrupt happened. 0 means not, and 1 means yes.

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates this function fails.

5.7.3.3.4. Ql_EINT_Uninit

This function releases the specified EINT pin.

- **Prototype**

```
s32 Ql_EINT_Uninit(PinName eintPinName)
```

- **Parameters**

eintPinName:
[in] EINT pin name.

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates that this function fails.

5.7.3.3.5. Ql_EINT_GetLevel

This function gets the level of the specified EINT pin.

- **Prototype**

```
s32 Ql_EINT_GetLevel(PinName eintPinName)
```

- **Parameters**

eintPinName:
[in] EINT pin name.

- **Return Value**

1 means high level, and 0 means low level.

5.7.3.3.6. Ql_EINT_Mask

This function masks the specified EINT pin.

- **Prototype**

```
void Ql_EINT_Mask(PinName eintPinName)
```

- **Parameters**

eintPinName:
[in] EINT pin name.

- **Return Value**

None.

5.7.3.3.7. Ql_EINT_Unmask

This function unmasks the specified EINT pin.

- **Prototype**

```
void Ql_EINT_Unmask(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.

```
void eint_callback_handle(Enum_PinName eintPinName, Enum_PinLevel pinLevel, void* customParam)
{
    s32 ret;
    if(PINNAME_DTR==eintPinName) //Extern interrput from which pin
    {
        ret=Ql_EINT_GetLevel(eintPinName); //Get the pin level if you need.

        //You need unmask the interrupt again, because PINNAME_DTR pin interrupt initialized as auto
mask.
        Ql_EINT_Unmask(eintPinName);
        if(*((s32*)customParam) >= 3)
        {
          //If don't want the interrupt you can mask it now!!!
            Ql_EINT_Mask(eintPinName);
        }
    }
    else if(PINNAME_SIM_PRESENCE==eintPinName)
    {
        ret=Ql_EINT_GetLevel(eintPinName);
        Ql_Debug_Trace("\r\n<--Ql_EINT_GetLevel pin(%d) levle(%d)-->\r\n",eintPinName,ret);

        //Ql_EINT_Unmask(eintPinName); not need, initialization this interrupt is not auto mask.
        if(*((s32*)customParam) >= 3)
        {
            //If don't want the interrupt you can mask it now!!!
            Ql_EINT_Mask(PINNAME_SIM_PRESENCE);
       }
    }
    *((s32*)customParam) +=1;
}

void API_TEST_eint(void)
```

```
{
    s32 ret;

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

    //Initialization some parameters; auto mask is false.
    ret=Ql_EINT_Init(PINNAME_SIM_PRESENCE, EINT_LEVEL_TRIGGERED, 0,5,0);
    Ql_Debug_Trace("\r\n<--pin(%d) Ql_EINT_Init ret=%d-->\r\n",PINNAME_SIM_PRESENCE,ret);

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

    //Initialization some parameters; auto maks is true.
    ret=Ql_EINT_Init( PINNAME_DTR, EINT_LEVEL_TRIGGERED, 0, 5,1);
}
```

5.7.4. PWM

5.7.4.1. PWM Overview

OpenCPU module has one PWM pin，Please refer to ***Chapter 5.7.2.2*** for details. The PWM have two clock sources: one is 32K (the exact value is 32768Hz) and the other is 13M. When the module is in the sleep mode, the 13M clock source will be disabled, but the 32K clock source works normally.

5.7.4.2. PWM Usage

The following steps show how to use the PWM function:

Step 1: Initialize a PWM pin. Call *Ql_PWM_Init* function to config the PWM duty cycle and frequency.
Step 2: PWM waveform control. Call *Ql_PWM_Output* to switch on/off the PWM waveform output.
Step 3: Release the PWM pin. Call *Ql_PWM_Uninit* to release the PWM pin. This step is optional.

5.7.4.3. API Functions

5.7.4.3.1. Ql_PWM_Init

This function initializes the PWM pin.

- **Prototype**

```
s32 Ql_PWM_Init(PinName pwmPinName,PwmSource pwmSrcClk,PwmSourceDiv pwmDiv,u32 lowPulseNum,u32 highPulseNum)
```

- **Parameters**

pwmPinName:
[in] Pin name, only can be PINNAME_NETLIGHT.

pwmSrcClk:
[in] PWM clock source, one value of *Enum_PwmSource*.

pwmDiv:
[in] Clock source divide, one value of *Enum_PwmSourceDiv*.

lowPulseNum:
[in] Set the number of clock cycles to stay at low level. The result of lowPulseNum plushighPulse Num is less than 8193.

highPulseNum:
[in] Set the number jof clock cycles to stay at high level. The result of lowPulseNum plus highPulseNum is less than 8193.

NOTES

1. PWM Duty cycle=highPulseNum/(lowPulseNum+highPulseNum).
2. PWM frequency=(pwmSrcClk / pwmDiv)/(lowPulseNum+highPulseNum).

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates this function fails.

5.7.4.3.2. Ql_PWM_Uninit

This function releases a PWM pin.

- **Prototype**

```
s32 Ql_PWM_Uninit(PinName pwmPinName)
```

- **Parameters**

pwmPinName:
[in] Pin name, one value of *Enum_PinName*.

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates this function fails.

5.7.4.3.3. Ql_PWM_Output

This function switches on/off the PWM waveform output.

- **Prototype**

```
s32 Ql_PWM_Output(PinName pwmPinName,bool pwmOnOff)
```

- **Parameters**

pwmPinName:
[in] Pin name, one value of *Enum_PinName*.

pwmOnOff:
[in] PWM enable. Control the PWM waveform output or disable.

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates this function fails.

5.7.4.4. Example

This following sample codes show how to use the PWM.

```
void API_TEST_pwm(void)
{
    s32 ret;

    //Initialization of some parameters.
    ret=Ql_PWM_Init(PINNAME_NETLIGHT, PWMSOURCE_32K, PWMSOURCE_DIV4, 500, 500);
    Ql_Debug_Trace("\r\n<--pin(%d) Ql_PWM_Init ret=%d-->\r\n",PINNAME_NETLIGHT,ret);

    //PWM waveform output.
    ret=Ql_PWM_Output(PINNAME_NETLIGHT, 1);
    Ql_Debug_Trace("\r\n<--pin(%d) Ql_PWM_Output start ret=%d-->\r\n",PINNAME_NETLIGHT,ret);

    Ql_Sleep(3000);
    //PWM waveform stop.
    ret=Ql_PWM_Output(PINNAME_NETLIGHT, 0);
    Ql_Debug_Trace("\r\n<--pin(%d) Ql_PWM_Output stop ret=%d-->\r\n",PINNAME_NETLIGHT,ret);
```

```
    //Release the pin if you do not use it.
    ret=Ql_PWM_Uninit(PINNAME_NETLIGHT);
    Ql_Debug_Trace("\r\n<--pin(%d) Ql_PWM_Uninit stop ret=%d-->\r\n",PINNAME_NETLIGHT,ret);
}
```

5.7.5. ADC

5.7.5.1. ADC Overview

OpenCPU module provides an analogue input pin that can be used to detect the external voltage. Pleae refer to ***document [2]*** for the pin definitions and ADC hardware characteristics. The voltage range that can be detected is 0~2800mV.

5.7.5.2. ADC Usage

The following steps tell the use of the ADC function:

Step 1: Register an ADC sampling function. Call *Ql_ADC_Register* function to register a callback function which will be when the module output the ADC value.

Step 2: ADC sampling parameter initialization. Call *Ql_ADC_Init* function to set the sample counts and the interval of each sample.

Step 3: Start/stop ADC sampling. Use *Ql_ADC_Sampling* function with an enable parameter to start ADC sampling, and then ADC callback function will be invoked cyclically to report the ADC value. Again call this API function with a disable parameter may stop the ADC sampling.

5.7.5.3. API Functions

5.7.5.3.1. Ql_ADC_Register

This function registers an ADC callback function. The callback function will be called when the module output the ADC value.

- **Prototype**

```
s32 Ql_ADC_Register(ADCPin adcPin,Callback_ADC callback_adc,void *customParam)
typedef void (*Callback_ADC)(ADCPin adcPin, u32 adcValue, void *customParam)
```

- **Parameters**

adcPin:
[in] ADC pin name, one value of *Enum_ADCPin*.

callback_adc:
[in] Callback funtion, will be called when the module output the ADC value.

customParam:
[in] Customized parameter, if not used, just set to NULL.

adcValue :
[in] It is the average value of ADC sampling. The range is 0~2800 mV.

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates this function fails.

5.7.5.3.2. Ql_ADC_Init

This function initializes the configurations for ADC, including sampling count and the interval of each sampling. The ADC callback function will be called when the module output the ADC value, and the value is the average of the sampling value.

- **Prototype**

```
s32 Ql_ADC_Init(ADCPin adcPin,u32 count,u32 interval)
```

- **Parameters**

adcpin:
[in] ADC pin name, one value of *Enum_ADCPin*.

count:
[in] Internal sampling times for each reporting ADC value. The minimum is 5.

interval:
[in] Interval of each internal sampling; unit is ms. The minimum is 200 (ms). This means the ADC Report frequency must be less than 1 Hz.

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates this function fails.

5.7.5.3.3. Ql_ADC_Sampling

This function switches on/off ADC sampling.

- **Prototype**

```
s32 Ql_ADC_Sampling(ADCPin adcPin,bool enable)
```

- **Parameters**

adcPin:
[in] ADC pin name, one value of *Enum_ADCPin*.

enable:
[in] Sample control. 1: start to sample 0: stop sampling.

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates this function fails.

5.7.5.4. Example

The following example demonstrates the use of the ADC sampling.

```
void ADC_callback_handle(Enum_ADCPin adcPin, u32 adcValue, void *customParam)
{
    s32 ret;
    if (PIN_ADC0==adcPin )
    {
        if( *((s32*)customParam) >= 4)
        {
            //Stop ADC0 sampling, if you do not need.
            ret=Ql_ADC_Sampling(PIN_ADC0, 0);
        }
    }
  *((s32*)customParam) +=1;
}
void API_TEST_adc(void)
{
    s32 ret;

    //Register ADC0 callback function.
    ret=Ql_ADC_Register(PIN_ADC0, ADC_callback_handle, (void * )&ADC0customParam);

    //Set the internal sampling times, and the internal sampling interval.
    ret=Ql_ADC_Init(PIN_ADC0, 5, 200);//So the ADC0 reports the ADC value at frequency of 1
Hz.(5*200ms).
    ret=Ql_ADC_Sampling(PIN_ADC0, 1); //Start to sample
}
```

5.7.6. IIC

5.7.6.1. 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.

5.7.6.2. IIC Usage

The following steps tell how to work with IIC function:

Step 1: Initialize IIC interface. Call *QI_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 *QI_IIC_Config* to config parameters that the slave device needs. Please refer to the API decription for extended information.

Step 3: Read data from slave. Developers can use *QI_IIC_Read* function to read data from the specified slave. The following figure shows the data exchange direction.

Step 4: Write data to slave. Developers can use *QI_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. Developers can use *QI_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. Developers can use *QI_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 *QI_IIC_Uninit* function to release the specified IIC channel.

5.7.6.3. API Functions

5.7.6.3.1. 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 No, the range is 0~254.

pinSCL:
[in] IIC SCL pin.

pinSDA:
[in] IIC SDA pin.

IICtype:
[in] IIC type. 0 means the IIC communication is simulated by pins, and 1 means IIC contronller.

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates this function fails.

5.7.6.3.2. 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 No. The No is specified by *Ql_IIC_Init* function.

isHost :
[in] Must be ture, just support host mode.

slaveAddr:
[in] Slave address.

speed:
[in] Just for IIC controller, and the parameter can be ingored if you use simulation IIC

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates this function fails.

5.7.6.3.3. Ql_IIC_Write

This function writes data to specified slave through IIC interface.

- **Prototype**

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

- **Parameters**

chnnlNo:
[in] IIC channel No. The No is specified by *Ql_IIC_Init* function.

slaveAddr:
[in] Slave address.

pData:
[in] Setting value to slave.

Len:
[in] Number of bytes to write. If IICtype=1, then 1<len<8. Because our IIC contronller at most supports 8 bytes at most for one time transaction

- **Return Value**

If there is no error, return the length of the write data. Negative integer indicates that this function fails.

5.7.6.3.4. Ql_IIC_Read

This function reads data from specified slave through IIC interface.

- **Prototype**

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

- **Parameters**

chnnlNo:
[in] IIC channel No. The No is specified by *Ql_IIC_Init* function.

slaveAddr:
[in] Slave address.

pBuffer:
[Out] Read buffer of reading the specified register from slave.

Len:
[Out] Number of bytes to read. If IICtype=1, then 1<len<8. Because our IIC contronller at most support 8 bytes at most for one time transaction.

- **Return Value**

If no error, return the length of the read data. Negative integer indicates this function fails.

5.7.6.3.5. Ql_IIC_WriteRead

This function reads data from the specified register (or address) of the 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 No. The No 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 write. If IICtype=1, 1<wrtLen<8.

pBuffer:
[Out] Read buffer of reading the specified register from slave.

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

- **Return Value**

If there is no error, return the length of the read data. Negative integer indicates that this function fails.

5.7.6.3.6. Ql_IIC_Uninit

This function releases the pins.

- **Prototype**

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

- **Parameters**

chnnlNo:
[in] IIC channel No. The No is specified by *Ql_IIC_Init* function.

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates this function fails.

5.7.6.4. 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");

    //Simultion IIC test
    ret=Ql_IIC_Init(0,PINNAME_GPIO0,PINNAME_GPIO1,0);

    //Simultion 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_GPIO8,PINNAME_GPIO9,1);

    //IIC controller speed 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);
}
```

5.7.7. SPI

5.7.7.1. SPI Overview

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

5.7.7.2. 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: Configuration. Call *Ql_SPI_Config* function to config some 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: Read data. Call *Ql_SPI_Read* function to read bytes from the specified slave bus.
Step 5: Write and read. The *Ql_SPI_WriteRead* function is used for SPI full-duplex communication that can read and write data at one time.
Step 6: Release SPI interface. Invoke *Ql_SPI_Uniti* function to release the SPI PINs. This step is optional.

5.7.7.3. API Functions

5.7.7.3.1. 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 No; the range is 0~254

pinClk:
[in] SPI CLK pin.

pinMiso:
[in] SPI MISO pin.

pinMosi:
[in] SPI MOSI pin.

spiType:
[in] SPI type, the type must be zero.

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates this function fails.

5.7.7.3.2. 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 No. The No is specified by *Ql_SPI_Init* function.

isHost:
[in] Must be true, not support salve mode.

cpol:
[in] Clock Polarity. More information please refer to the SPI standard protocol.

cpha:
[in] Clock Phase. More information please refer to the SPI standard protocol.

clkSpeed:
[in] SPI speed, not supported right now. The input argument will be ignored.

- **Return Value**

If there is no error, return the length of the write data. Negative integer indicates that this function fails

5.7.7.3.3. 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 No. The No is specified by *Ql_SPI_Init* function.

pData:
[in] The setting value to slave.

len:
[in] Number of bytes to write.

- **Return Value**

If there is no error, return the length of the write data. Negative integer indicates that this function fails.

5.7.7.3.4. Ql_SPI_Read

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

- **Prototype**

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

- **Parameters**

chnnlNo:
[in] SPI channel No. The No is specified by *Ql_SPI_Init* function.

pBuffer:
[Out] Read buffer of reading from slave.

rdLen:
[Out] Number of bytes to read.

- **Return Value**

If no error, return the length of the read data. Negative integer indicates this function fails.

5.7.7.3.5. Ql_SPI_WriteRead

This function is used for SPI full-duplex communication.

- **Prototype**

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

- **Parameters**

chnnlNo:
[in] SPI channel No. The No is specified by *Ql_SPI_Init* function.

pData:
[in] Setting value to slave.

wrtLen:
[in] Number of bytes to write.

pBuffer:
[Out] Read buffer of reading from slave.

rdLen:
[Out] Number of bytes to read.

NOTES

1. If (wrtLen>rdLen), the other read buffer data will be set 0xff;
2. If (rdLen>wrtLen), the other write buffer data will be set 0xff.

- **Return Value**

If no error, return the length of the read data. Negative integer indicates this function fails.

5.7.7.3.6. Ql_SPI_Uninit

This function releases the SPI pins.

- **Prototype**

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

- **Parameters**

chnnlNo:
[in] SPI channel No. The No is specified by *Ql_SPI_Init* function.

- **Return Value**

QL_RET_OK: this function succeeds. Negative integer indicates this function fails

5.7.7.4. 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];
    Ql_Debug_Trace("\r\n<***********   TEST API Test   ***********>\r\n");

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

    ret=Ql_SPI_Config(1,1,1,1,1000);   //isHost=1, cpol=1, cpha=1, clock=10MHz
    Ql_Debug_Trace("<--Ql_SPI_Config(), SPI channel 1, ret=%d-->",ret);

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

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

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

5.8. GPRS API

5.8.1. Overview

The API functions in this section are declared in “ql_gprs.h”.

The module supports defining and activating 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 OpenCPU 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 function *Ql_GPRS_Register* to register the GPRS’s callback function.

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

Step 3: Activate PDP. Call function *Ql_GPRS_Activate* 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 execute with 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 function *Ql_GPRS_GetLocalIPAddress* to get the local IP address.

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

Step 6: Deactivate. Call function *Ql_GPRS_Deactivate* 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_Activate* 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 execute with 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.The callback functions will be invoked only in the registered task.

- **Prototype**

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

```
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] OpenCPU supports two PDP-contexts to the destination host at a time. This parameter can be 0 or 1.

callback_func:
[in] This callback function is called by OpenCPU to inform Embedded Application whether this function succeeds or not. This function should be implemented by Embedded Application.

customerParam:
[in] One customized parameter that can be passed into the callback functions.

- **Return Value**

The return value is 0 if 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*. This parameter maybe is 0 or 1.

errCode:
[Out] The result code of activating GPRS, 0 indicates succeed in activating GPRS.

customerParam:
[Out] One customized parameter that was passed into when calling *Ql_GPRS_Register*. This parameter may be is 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*. This parameter may be is 0 or 1.

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

customerParam:
[Out] One customized parameter that was passed into when calling *Ql_GPRS_Register*. This parameter may be is NULL.

- **Return Value**

None.

5.8.3.4. Ql_GPRS_Config

This function sets the authentication parameters APN/login/password/authentication to use with a profile ID during PDP activation.

- **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] NULL-terminated APN characters.

apnUserId:
[in] User ID, NULL-terminated characters.

apnPasswd:
[in] Password, NULL-terminated characters.

Authtype:
[in] Authentication method
1 - PAP
2 - CHAP

- **Return Value**

The possible return values are as follows:

SOC_SUCCESS: This function succeeds.
SOC_INVAL: Invalid argument.
SOC_ALREADY: The function is running.

5.8.3.5. Ql_GPRS_Activate

This function actives a PDP context. Depending on the network status, PDP 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 give the activation result.

- **Prototype**

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

- **Parameters**

contextId:
[in] OpenCPU supports two PDP-contexts to the destination host at the same time. This parameter can be 0 or 1.

- **Return Value**

The possible return values are as follows:

GPRS_PDP_SUCCESS: This function succeeds, and activating GPRS succeeds.
GPRS_PDP_WOULDBLOCK: The app should wait, till the callback function is called.
The app gets the information of success or failure in callback function.
The maximum possible time for Activating GPRS is 180s.
GPRS_PDP_INVAL: Invalid argument.
GPRS_PDP_ALREADY: The activating operation is in process.
GPRS_PDP_BEARER_FAIL: Bearer is broken.

- **Example**

The following codes show the activating GPRS processing.

```
{
    s32 ret;
    ret=Ql_GPRS_Activate(0);
    if (GPRS_PDP_SUCCESS==ret)
    {
        //Activate GPRS successfully
    }
    else if (GPRS_PDP_WOULDBLOCK==ret)
    {
        //Activating GPRS, need to wait Callback_GPRS_Actived for the result.
    }
    else if (GPRS_PDP_ALREADY==ret)
    {
        //GPRS has been activating...
```

```
    }else{
        //Fail to activate GPRS, 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 the specified PDP context. The maximum possible time for Activating GPRS is 180s.

This function supports two modes:

- **Non-blocking Mode**

When the "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 the "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 same as *Ql_GPRS_Activate()* functionally.

- **Prototype**

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

- **Parameters**

contextId:
[in] OpenCPU supports two PDP-contexts to the destination host at the same time. This parameter can be 0 or 1.

isBlocking:
[in] Blocking mode. TRUE=blocking mode, FALSE=non-blocking mode.

- **Return Value**

The possible return values are as follows:

GPRS_PDP_SUCCESS: This function succeeds, and activating GPRS succeeds.
GPRS_PDP_INVAL: Invalid argument.
GPRS_PDP_ALREADY: The activating operation is in process.

GPRS_PDP_BEARER_FAIL: 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)
    {
        //Activate GPRS successfully
    }
    else if (GPRS_PDP_ALREADY==ret)
    {
        //GPRS has been activating...
    }else{
        //Fail to activate GPRS, and the error code is in "ret".
        //Developers may retry to activate GPRS, and reset the module after 3 successive failures.
    }
}
```

5.8.3.7. Ql_GPRS_Deactivate

This function deactivates the specified PDP context. Depending on the network status, PDP deactivation will take some time, the longest time is 90s. When the PDP deactivation succeeds or fails, *CallBack_GPRS_Deactived* callback function will be called, and give the activation result.

- **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 Possible Error Codes.

- **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)
    {
        //Deactivating GPRS, need to wait Callback_GPRS_Deactived for the result
    }else{
        //Fail to deactivate GPRS, and the error code is in "ret".
    }
}
```

5.8.3.8. Ql_GPRS_DeactivateEx

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

This function supports two modes:

- **Non-blocking Mode**

When the "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 the "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 same as *Ql_GPRS_Deactivate()* functionally.

- **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] Blocking mode. TRUE=blocking mode, FALSE=non-blocking mode.

- **Return Value**

The possible return values are as follows:

GPRS_PDP_SUCCESS: This function succeeds, and activating GPRS succeeds.
GPRS_PDP_INVAL: Invalid argument.
GPRS_PDP_ALREADY: The activating operation is in process.
GPRS_PDP_BEARER_FAIL: 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 the 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] Point to the buffer that is the storage space for 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, which include the first DNS addess and the second DNS addess.

- **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] Point to the buffer that is the storage space for the primary DNS server's IP address.

secondAddr:
[Out] Point to the buffer that is the storage space for 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 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 API

5.9.1. Overview

Socket program implements the TCP and UDP protocols. In OpenCPU, developers use the API functions 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 section 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. Call function *Ql_SOC_Register* to register the socket-related callback functions.

Step 2: Create socket. Call function *Ql_SOC_Create* to create a socket. The ‘contextId’ argument should be same as *Ql_GPRS_Register* uses, and the ‘socketType’ should be set as ’SOCK_TCP’.

Step 3: Connet to socket. Call *Ql_SOC_Connect* to request a socket connection. The *Callback_Socket_Connect* function will be invoked whether the connection is successful or not.

Step 4: Send data to socket. Call function *Ql_SOC_Send* to send data. After the data is sent out, you can call *Ql_SOC_GetAckNumber* function to check whether the data is received by the server. If *Ql_SOC_Send* retruns *SOC_WOULDBLOCK*, the App must wait for *callback_socket_write* function to send data again.

Step 5: Receive data from socket. When there’s data coming from the socket，the Callback_socket_read function will be invoked to inform App. When received the notificatioin, App may call Ql_SocketRecv to receive the data. App must read out all of the data. Otherwise, the callback function will not be invoked when new data comes.

Step 6: Close socket. Call function *Ql_SOC_Close* to close the socket. App can call function *Ql_SOC_Close* to close the socket. When App receives the notification the server side has closed the socket, App 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. Call function *Ql_SOC_Register* to register the socket-related callback functions.
Step 2: Create Socket. Call function *Ql_SOC_Create* to create a socket.
Step 3: Bind. Call function *Ql_SOC_Bind* to associate a local address with a socket.
Step 4: Listen. Call function *Ql_SOC_Listen* 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 function *Ql_SOC_Accept* to accept the connection request.
Step 6: Send data to socket. Call function *Ql_SOC_Send* to send data to socket. After the data is sent out, you can call *Ql_SOC_GetAckNumber* function to check whether the data is received by the client. When this function retruns *SOC_WOULDBLOCK*, the App has to wait till *callback_socket_write* is invoked, and then App 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 of the data. Otherwise, the callback function will not be invoked when new data comes.
Step 8: Close socket. Call function *Ql_SOC_Close* to close the socket. App can call function *Ql_SOC_Close* to close the socket. When App receives the notification that client side has closed the socket, App has to call *Ql_SOC_Close* to close the socket from the server side.

5.9.2.3. UDP Service Socket Usage

The following steps tell how to work with UDP Server:

Step 1: Register. Call function *Ql_SOC_Register* to register the socket-related callback functions.
Step 2: Create socket. Call function *Ql_SOC_Create* to create a socket. The ‘contextId’ argument should be same as *Ql_GPRS_Register* uses, and the ‘socketType’ should be set as ‘SOCK_UDP’.
Step 3: Bind. Call function *Ql_SOC_Bind* to associate a local address with a socket.
Step 4: Send data to socket. Call function *Ql_SOC_SendTo* to send data. When this function retruns *SOC_WOULDBLOCK*, the App 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 App and App can call *Ql_SocketRecvFrom* to receive the data. App must read out all of the data. Otherwise, the callback function will not be invoked when new data comes.
Step 6: Close socket. Call function *Ql_SOC_Close* to close the socket. App can call function *Ql_SOC_Close* to close the socket.

5.9.3. API Functions

5.9.3.1. Ql_SOC_Register

This function registers callback functions for the 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] The pointer of the socket-related callback function.

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

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.

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.

5.9.3.4. Callback_Socket_Accept

Accept a connection on a socket when the module is a server. This function 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*.

error_code:
[Out] Error code.

customParam:
[Out] Customized parameter.

- **Return Value**

None.

5.9.3.5. Callback_Socket_Read

This function will be invoked when received data from the socket. Then you can 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*.

error_code:
[Out] Error code.

customParam:
[Out] Customized parameter.

- **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 application 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.

- **Return Value**

None.

5.9.3.7. Ql_SOC_Create

This function creates a socket with the specified socket ID on the 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*. This parameter may be is 0 or 1.

socketType :
[in] This parameter is one of *Enum_SocketType*:

```
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: Invalid argument.
SOC_BEARER_FAIL: Bearer is broken.
SOC_LIMIT_RESOURCE: Exceed the maximum socket number.

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 the 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 longest time is 75 seconds, 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 the function succeeds. Otherwise, a value of *Enum_SocError* is returned.The possible return values are as follows:

SOC_SUCCESS: This function succeeds.
SOC_WOULDBLOCK: The application should wait, till the *callback_socket_connect* function is called. The application can get the information of success or failure in callback function.
SOC_INVALID_SOCKET: 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 longest time is 75 seconds, 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:

- **Non-blocking Mode**

When the "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 the “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 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] Blocking mode. TRUE=blocking mode, FALSE=non-blocking mode.

- **Return Value**

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

SOC_SUCCESS: This function succeeds.
SOC_INVALID_SOCKET: Invalid socket.
Other values: error code. Please refer to *Enum_SocErrCode*.

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 you call *Ql_SOC_Send* function to send to many data to the socket buffer, this function will return

SOC_WOULDBLOCK. Then you must stop sending data. After the socket buffer has enough space, the *callback_socket_write* callback function will be called, and you can continue to send the data. This function just sends data to the network, but whether the data is received by the server is unknown. So maybe you need to call *Ql_SOC_GetAckNumber* function 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 send.

dataLen:
[in] Number of bytes to send.

- **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 in the dataLen parameter. Otherwise, a value of *Enum_SocError* is returned.

NOTES

1. The application should call *Ql_SOC_Send* circularly 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, but not *SOC_WOULDBLOCK*, which indicates some error happened to the socket, 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*, 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. You can 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 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] Point to a buffer that is the storage space for the received data.

dataLen:
[Out] Length of pData, in bytes.

- **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* circularly in the *callback_socket_read* function to receive data and do data processing work till the *SOC_WOULDBLOCK* is returned.
2. If this function returns 0, which indicates the server closed the socket, 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, but not *SOC_WOULDBLOCK*, which indicates some errors happened to the socket, 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] Point to an u64 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 the data in pData, 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] Buffer to store the received data.

rcvLen:
[Out] Length of pData, in bytes.

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 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 in which it is 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. Limiting the maximum length of the request queue. The maximum 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 on 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 O.S.

- **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] OpenCPU supports two PDP-contexts to the destination host at a time. This parameter can be 0 or 1.

requestId:
[Out] Embedded in response message.

hostname:
[in] The 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 if fail *ipAddrCnt:*
[Out] Get address number.

ipAddr:
[Out] The 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 integer to store in *ipaddr* parameter.

- **Prototype**

```
s32 Ql_IpHelper_ConvertIpAddr(u8 *addressstring, u32* ipaddr)
```

- **Parameters**

addressstring:
[in] IP address string.

ipaddr:
[Out] Pointer to u32, each byte stores the IP digit converted from the corresponding IP string.

- **Return Value**

The possible return values are as follows:

SOC_SUCCESS: The IP address string is valid.
SOC_ERROR: The IP address string is invalid.
SOC_INVAL: 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 too long
    SOC_ALREADY                 = -7,     //Operation already in progress
    SOC_OPNOTSUPP               = -8,     //Operation 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 exmples “example_tcpclient.c”, example_udpclient.c in the SDK\example\.

5.10. Watchdog API

Pleae refer to the document ***Quectel_OpenCPU_Watchdog_Application_Note*** for the complete introduction of OpenCPU watchdog solution.

5.11. FOTA API

OpenCPU provides FOTA (Firmware Over the Air) function that can upgrade App remotely. The related API functions are defined and described in this section, and demonstrates how to program with FOTA.

5.11.1. Usage

Please refer to the document ***Quectel_OpenCPU_FOTA_Application_Note*** for the complte application solution.

5.11.2. API Functions

5.11.2.1. Ql_FOTA _Init

Initialise FOTA related functions. It is a simple API. Programers only need to pass the simple parameters to this API.

- **Prototype**

```
s32 Ql_FOTA_Init(ST_FotaConfig * pFotaCfg)
```

- **Parameters**

pFotaCfg:
[in] A pointer to ST_FotaConfig.

```
typedef struct tagFotaConfig
{
    s16 Q_gpio_pin1;          //Watchdog GPIO pin 1. If only use one GPIO, you can set others to -1 which means invalid.
    s16 Q_feed_interval1;     //GPIO1 time interval for feed dog.
    s16 Q_gpio_pin2;          //Watchdog GPIO pin 2. If only use one GPIO, you can set others to -1 which means invalid.
    s16 Q_feed_interval2;     //GPIO 2 time interval for feed dog.
    s32 reserved1;            //Reserve 1, must be zero
    s32 reserved2;            //Reserve 2, must be zero
}ST_FotaConfig;
```

- **Return Value**

QL_RET_OK: indicates the function succceeds.
QL_RET_ERR_PARAM: indicates parameter error.
Ql_RET_NOT_SUPPORT: indicates not support this function.
Ql_RET_ERR_RAWFLASH_UNKNOW: indicates unkown error.

5.11.2.2. Ql_FOTA_WriteData

This function writes the delta data of applications to the special space in the module.

- **Prototype**

```
s32 Ql_FOTA_WriteData(s32 length, s8* buffer)
```

- **Parameters**

length:
[in] The length of writing (Unit: Bytes). Recommend to be 512 bytes

buffer:
[in] A pointer to the data buffer.

- **Return Value**

QL_RET_OK: indicates this function successes.
QL_RET_ERR_PARAM: indicates parameter error.
Ql_RET_NOT_SUPPORT: indicates not support this function.
Ql_RET_ERR_UNKOWN: indicates unkown error.
Ql_RET_ERR_RAWFLASH_OVERRANGE: indicates over flash range.
Ql_RET_ERR_RAWFLASH_UNIITIALIZED: indicates uninitialized before writing or reading flash.
Ql_RET_ERR_RAWFLASH_UNKNOW: indicates unkown error.
Ql_RET_ERR_RAWFLASH_INVLIDBLOCKID: indicates block ID invalid.
Ql_RET_ERR_RAWFLASH_PARAMETER: indicates parameter error.
Ql_RET_ERR_RAWFLASH_ERASEFlASH: indicates erasen flash failure.
Ql_RET_ERR_RAWFLASH_WRITEFLASH: indicates writen flash failure.
Ql_RET_ERR_RAWFLASH_READFLASH: indicates read flash failure.
Ql_RET_ERR_RAWFLASH_MAXLENGATH: indicates the data length too long.

5.11.2.3. Ql_FOTA_ReadData

This function reads data from the data region which *Ql_FOTA_WriteData* writes to. If developers need to check the whole data package after writing, this API can read back the data.

- **Prototype**

```
s32 Ql_FOTA_ReadData(u32 offset, u32 len, u8* pBuffer)
```

- **Parameters**

offset:
[in] The offset value to the data region.

len:
[in] The length to read (Unit: Byte) . Recommend to be 512 bytes.

pBuffer:
[Out] Point to the buffer to store read data.

- **Return Value**

QL_RET_ERR_PARAM: indicates parameter error.

If succceeds, returns the real read number of bytes.

5.11.2.4. Ql_FOTA_Finish

Compare calculated checksum with image checksum in the header after the whole image is written.

- **Prototype**

```
s32 Ql_FOTA_Finish(void)
```

- **Parameters**

None.

- **Return Value**

QL_RET_OK: indicates this function successed.
Ql_RET_NOT_SUPPORT: indicates not support this function.
Ql_RET_ERR_UNKOWN: indicates unknown error.
Ql_RET_ERR_RAWFLASH_OVERRANGE: indicates over flash range.
Ql_RET_ERR_RAWFLASH_UNIITIALIZED: indicates uninitialized before writing or reading flash.
Ql_RET_ERR_RAWFLASH_UNKNOW: indicates unknown error.
Ql_RET_ERR_RAWFLASH_INVLIDBLOCKID: indicates block ID invalid.
Ql_RET_ERR_RAWFLASH_PARAMETER: indicates parameter error.
Ql_RET_ERR_RAWFLASH_ERASEFlASH: indicates erase flash failure.
Ql_RET_ERR_RAWFLASH_WRITEFLASH: indicates written flash failure.
Ql_RET_ERR_RAWFLASH_READFLASH: indicates read flash failure.
Ql_RET_ERR_RAWFLASH_MAXLENGATH: indicates the data length too long.

5.11.2.5. Ql_FOTA_Update

Starts FOTA Update.

- **Prototype**

```
s32 Ql_FOTA_Update(void);
```

- **Parameters**

None.

- **Return Value**

QL_RET_OK: indicates this function succeeds.
QL_RET_ERR_INVALID_OP: indicates invalid operation.
Ql_RET_NOT_SUPPORT: indicates not support this function.
Ql_RET_ERR_RAWFLASH_PARAMETER: indicates parameter error.

Ql_RET_ERR_RAWFLASH_ERASEFlASH: indicates erasen flash failure.
Ql_RET_ERR_RAWFLASH_WRITEFLASH: indicates writen flash failure.

5.11.3. Example

The following code shows how to use FOTA function.

```
static ST_FotaConfig       FotaConfig;
static u8 g_AppBinFile[64]="appbin.bin"; //File name in file system
#define READ_SIZE 512
int StartAppUpdate()
{
      int iRet=-1;
      int iFileSize=0;
      int iReadSize=0;
      int iReadLen=0;
      int hFile=-1;
      char buf[512];
      char *p=NULL;
      static int s_iSizeRem=0;

      //1. Initialize some parameters
      Ql_memset((void *)(&FotaConfig), 0, sizeof(ST_FotaConfig));    //Do not enable watch_dog
      FotaConfig.Q_gpio_pin1=0;
      FotaConfig.Q_feed_interval1=100;
      FotaConfig.Q_gpio_pin2=26;
      FotaConfig.Q_feed_interval2=500;

      //2. Begin to check the Bin file.
      iRet=Ql_FS_GetSize((u8 *)g_AppBinFile); //Get the size of upgrade file from file system.
      if(iRet <QL_RET_OK)
      {
            //The file does not exist
            return -1;
       }

       iRet=Ql_FS_Open((u8 *)g_AppBinFile, QL_FS_READ_WRITE|QL_FS_CREATE);
       if(iRet <0)
       {
            //Open file failed.
              return -1;
       }
       hFile=iRet;//Get file handle
```

```
/*Write App bin to flash*/
iRet=Ql_FOTA_Init(&FotaConfig);      //Initialise the upgrade operation
if(QL_RET_OK !=iRet)
{
      return -1;
}
Ql_Debug_Trace("Ql_Fota_Init OK!\r\n");

while(iFileSize > 0)
{
      Ql_memset(buf, 0, sizeof(buf));
      if (iFileSize <=READ_SIZE)
{
      iReadSize=iFileSize;
}
else
{
       iReadSize=READ_SIZE;
}
iRet=Ql_FS_Read(hFile, buf, iReadSize, &iReadLen);    //Read upgrade data from file system.
if(QL_RET_OK != iRet)
{
      Ql_Debug_Trace("Read file failed!(iRet = %x)\r\n", iRet);
      return -1;
}
//Write upgrade data to FOTA Cache region.
iRet=Ql_FOTA_WriteData(iReadSize,(s8*)buf);
if(QL_RET_OK !=iRet)
{
      Ql_Debug_Trace("Fota write file failed!(iRet=%d)\r\n", iRet);
      return -1;
}else
{
      s_iSizeRem +=iReadSize;
}
      iFileSize -= iReadLen;
      Ql_Sleep(5);              //Sleep 5 ms for outputing catcher log!!!
}
Ql_FS_Close(hFile);

iRet=Ql_FOTA_Finish();      //Finish the upgrade operation ending with calling this API.
iRet=Ql_FOTA_Update();     //Update flag fields in the FOTA Cache.
if(QL_RET_OK != iRet)       //If this function succeeds, the module will automatically restart.
{
```

```
            Ql_Debug_Trace("[max] Ql_Fota_Update failed!(iRet=%d)\r\n", iRet);
            return -1;
        }
return 0;
}
```

Please refer to the “example_fota_ftp.c”, “example_fota_http.c” for the complete sample code in SDK\example\.

5.12. Debug API

The head file ql_trace.h must be included so that the debug functions can be called. All examples in OpenCPU SDK show the proper usages of these APIs.

5.12.1. Usage

There are two working modes for UART2 (DEBUG port): BASE_MODE and ADVANCE_MODE. Developers can config the working mode of UART2 by the “debugPortCfg” variable in the “*custom_sys_cfg.c*” file.

```
static const ST_DebugPortCfg debugPortCfg = {
BASIC_MODE              //Set the serial debug port (UART2) to a common serial port.
    //ADVANCE_MODE     //Set the serial debug port (UART2) to a special debug port.
};
```

Under basic mode, application debug messages will be output as text through UART2 port. The UART2 port works as common serial port with RX, TX and GND. In this case, UART2 can be used as common serial port for application.

Under ADVANCE_MODE, both application debug messages and system debug messages will be output through UART2 port with special format. The “Catcher Tool” provided by Quectel can be used to capture and analyze these messages. Usually developers don’t need to use ADVANCE_MODE without the requirements from support engineer. If needed, please refer to the document ***Catcher_Operation_UGD*** for the usage of the special debug mode.

5.12.2. API Functions

5.12.2.1. Ql_Debug_Trace

This function formats and prints a series of characters and values through the debug serial port (UART2). Its function is same as the standard “sprintf”.

- **Prototype**

```
s32 Ql_Debug_Trace (char *fmt, ... )
```

- **Parameter**

format:
Point to a null-terminated multibyte string which specifies how to interpret the data. The maximum string length is 512 bytes.

Format-control string. A format specification has the following form:

%type
type, a character that determines whether the associated argument is interpreted as a character, a string, or a number.

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 must not be larger than the maximum number of bytes allowed in buffer; otherwise, a buffer overrun can occur.
2. The maximum allowed number of characters to be output is 512.
3. To print a 64-bit integer, please first convert it to characters using *Ql_sprintf()*, and then print the characters of the 64-bit integer.

5.13. RIL API

OpenCPU RIL related API functions respectively implement the corresponding AT command's function. Developers can simply call API to send AT commands and get the response when API returns. You can refer to the document ***Quectel_OpenCPU_RIL_Application_Note*** for OpenCPU RIL mechanism.

NOTE

The APIs defined in this section 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.

5.13.1. AT API

The API functions in this section are declared in "ril.h".

5.13.1.1. 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 customer's parameter.

timeOut:
[in] Timeout for the AT command, unit in ms. If it is set to 0, RIL uses the default timeout time (3min).

- **Return Value**

RIL_AT_SUCCESS: succeed in executing AT command, and the response is OK.
RIL_AT_FAILED: fail to execute AT command, or the response is ERROR.
RIL_AT_TIMEOUT: indicates sending AT is timeout.
RIL_AT_BUSY: indicates sending AT.
RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, need 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
}
```

5.13.2. Telephony API

This section defines telephony related API functions that are implemented based on OpenCPU RIL. These APIs imeplement the equivalent functions as AT commands: **ATD**, **ATA**, **ATH**.

The API functions in this section are declared in “ril_telephony.h”.

To set/get the voice channel (normal/headset/handfree), you can call *RIL_AUD_SetChannel()*/*RIL_AUD_GetChannel()*. To set/get the volume, you can call *RIL_AUD_SetVolume()*/*RIL_AUD_GetVolume()*, which are defined in “ril_audio.h”.

5.13.2.1. RIL_Telephony_Dial

This function dials a specified number.

- **Prototype**

```
s32 RIL_Telephony_Dial(u8 type, char* phoneNumber, s32* result);
```

- **Parameter**

type:
[in] Must be 0; just support voice call.

phoneNumber:
[in] Phone number, null-terminated string.

result:
[Out] Result for dial, one value of *Enum_CallState*.

- **Return Value**

RIL_AT_SUCCESS: send AT successfully.
RIL_AT_FAILED: send AT failed.
RIL_AT_TIMEOUT: send AT timeout.
RIL_AT_BUSY: sending AT.
RIL_AT_INVALID_PARAM: invalid input parameter.
RIL_AT_UNINITIALIZED: RIL is not ready, need to wait for MSG_ID_RIL_READY and then call *Ql_RIL_Initialize* to initialize RIL.

5.13.2.2. RIL_Telephony_Answer

This function answers a coming call.

- **Prototype**

```
s32 RIL_Telephony_Answer(s32 *result);
```

- **Parameter**

result:
[Out] Result for dial, one value of *Enum_CallState*.

- **Return Value**

RIL_AT_SUCCESS: send AT successfully.
RIL_AT_FAILED: send AT failed.
RIL_AT_TIMEOUT: send AT timeout.
RIL_AT_BUSY: sending AT.
RIL_AT_INVALID_PARAM: invalid input parameter.
RIL_AT_UNINITIALIZED: RIL is not ready, need to wait for MSG_ID_RIL_READY and then call *Ql_RIL_Initialize* to initialize RIL.

5.13.2.3. RIL_Telephony_Hangup

This function hangs up the current call.

- **Prototype**

```
s32 RIL_Telephony_Hangup(void);
```

- **Parameter**

None.

- **Return Value**

RIL_AT_SUCCESS: send AT successfully.
RIL_AT_FAILED: send AT failed.
RIL_AT_TIMEOUT: send AT timeout.
RIL_AT_BUSY: sending AT.
RIL_AT_INVALID_PARAM: invalid input parameter.
RIL_AT_UNINITIALIZED: RIL is not ready, need to wait for MSG_ID_RIL_READY and then call *Ql_RIL _Initialize* to initialize RIL.

5.13.3. SMS API

This section defines short message related API functions that are implemented bassed on OpenCPU RIL. These APIs imeplement the same functionas as AT commands: **AT+CMGR**, **AT+CMGS**, **AT+CMGD**, etc.

The API functions in this section are declared in “ril_sms.h”.

5.13.3.1. RIL_SMS_ReadSMS_Text

This function reads a short message of text format with the specified index.

- **Prototype**

```
s32 RIL_SMS_ReadSMS_Text(u32 uIndex, LIB_SMS_CharSetEnum eCharset, ST_RIL_SMS_TextInfo* pTextInfo);
```

- **Parameter**

uIndex:
[in] The SMS index in current SMS storage

eCharset:
[in] Character set enum value

pTextInfo:
[in] The pointer of TEXT SMS info

- **Return Value**

RIL_AT_SUCCESS: send AT successfully.
RIL_AT_FAILED: send AT failed.
RIL_AT_TIMEOUT: send AT timeout.
RIL_AT_BUSY: sending AT.
RIL_AT_INVALID_PARAM: invalid input parameter.
RIL_AT_UNINITIALIZED: RIL is not ready, need to wait for MSG_ID_RIL_READY and then call *Ql_RIL_Initialize* to initialize RIL.

5.13.3.2. RIL_SMS_ReadSMS_PDU

This function reads a short message of PDU format with the specified index.

- **Prototype**

```
s32 RIL_SMS_ReadSMS_PDU(u32 uIndex, ST_RIL_SMS_PDUInfo* pPDUInfo);
```

- **Parameter**

index:
[in] The SMS index in current SMS storage

pduInfo:
[in] The pointer of “ST_RIL_SMS_PDUInfo” data

- **Return Value**

RIL_AT_SUCCESS: send AT successfully.
RIL_AT_FAILED: send AT failed.
RIL_AT_TIMEOUT: send AT timeout.
RIL_AT_BUSY: sending AT.
RIL_AT_INVALID_PARAM: invalid input parameter.
RIL_AT_UNINITIALIZED: RIL is not ready, need to wait for MSG_ID_RIL_READY and then call *Ql_RIL_Initialize* to initialize RIL.

5.13.3.3. RIL_SMS_SendSMS_Text

This function sends a short message of text format.

- **Prototype**

```
s32 RIL_SMS_SendSMS_Text(char* pNumber, u8 uNumberLen, LIB_SMS_CharSetEnum eCharset, u8* pMsg, u32 uMsgLen,u32 *pMsgRef);
```

- **Parameter**

pNumber:
[in] The pointer of phone number.

uNumberLen:
[in] The length of phone number.

eCharset
[in] CharSet, its value is same as *LIB_SMS_CharSetEnum*.

pMsg:
[in] The pointer of message content.

uMsgLen:
[in] The length of message content.

pMsgRef:
[Out] The pointer of message reference number.

- **Return Value**

RIL_AT_SUCCESS: send AT successfully.
RIL_AT_FAILED: send AT failed.

RIL_AT_TIMEOUT: send AT timeout.
RIL_AT_BUSY: sending AT.
RIL_AT_INVALID_PARAM: invalid input parameter.
RIL_AT_UNINITIALIZED: RIL is not ready, need to wait for MSG_ID_RIL_READY and then call *Ql_RIL_Initialize* to initialize RIL.

5.13.3.4. 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);
```

- **Parameter**

pPDUStr:
[in] The pointer of PDU string.

uPDUStrLen:
[in] The length of PDU string.

pMsgRef:
[Out] The pointer of message reference number.

- **Return Value**

RIL_AT_SUCCESS: send AT successfully.
RIL_AT_FAILED: send AT failed.
RIL_AT_TIMEOUT: send AT timeout.
RIL_AT_BUSY: sending AT.
RIL_AT_INVALID_PARAM: invalid input parameter.
RIL_AT_UNINITIALIZED: RIL is not ready, need to wait for MSG_ID_RIL_READY and then call *Ql_RIL_Initialize* to initialize RIL.

5.13.3.5. RIL_SMS_DeleteSMS

This function deletes one short message or more messages in current SMS storage with the specified rule.

- **Prototype**

```
s32 RIL_SMS_DeleteSMS(u32 uIndex,Enum_RIL_SMS_DeleteFlag eDelFlag);
```

- **Parameter**

index:
[in] The index number of SMS message.

flag:
[in] Delete flag, which is one value of *Enum_RIL_SMS_DeleteFlag*.

- **Return Value**

RIL_AT_SUCCESS: send AT successfully.
RIL_AT_FAILED: send AT failed.
RIL_AT_TIMEOUT: send AT timeout.
RIL_AT_BUSY: sending AT.
RIL_AT_INVALID_PARAM: invalid input parameter.
RIL_AT_UNINITIALIZED: RIL is not ready, need to wait for MSG_ID_RIL_READY and then call *Ql_RIL_Initialize* to initialize RIL.

5.13.4. SIM Card API

The API functions in this section are declared in “ril_sim.h”.

5.13.4.1. RIL_SIM_GetSimState

This function gets the state of SIM card.

- **Prototype**

```
s32 RIL_SIM_GetSimState(s32* state);
```

- **Parameter**

state:
[Out] SIM card State code, one value of *Enum_SIMState*.

- **Return Value**

RIL_AT_SUCCESS: this function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

5.13.4.2. RIL_SIM_GetIMSI

This function gets the IMSI number of SIM card.

- **Prototype**

```
s32 RIL_SIM_GetIMSI(char* imsi);
```

- **Parameter**

imsi:
[Out] IMSI number, a string of 15-byte.

- **Return Value**

RIL_AT_SUCCESS: this function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

5.13.4.3. RIL_SIM_GetCCID

This function gets the CCID number of SIM card.

- **Prototype**

```
s32 RIL_SIM_GetCCID(s32* ccid);
```

- **Parameter**

state:
[Out] CCID number, a string of 20-byte.

- **Return Value**

RIL_AT_SUCCESS: this function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

5.13.5. Network API

The API functions in this section are declared in “ril_network.h”.

5.13.5.1. RIL_NW_GetGSMState

This function gets the GSM network register state.

- **Prototype**

```
s32 RIL_NW_GetGSMState(s32 *stat);
```

- **Parameter**

stat:
[Out]GSM State.

- **Return Value**

One value of *Enum_NetworkState*: network register state code. -1: fail to get the network state.

5.13.5.2. RIL_NW_GetGPRSState

This function gets the GPRS network register state.

- **Prototype**

```
s32 RIL_NW_GetGPRSState(s32 *stat);
```

- **Parameter**

stat:
[Out]GPRS State.

- **Return Value**

One value of *Enum_NetworkState*: network register state code. -1: fail to get the network state.

5.13.5.3. RIL_NW_GetSignalQuality

This function gets the signal quality level and bit error rate.

- **Prototype**

```
s32 RIL_NW_GetSignalQuality(u32* rssi, u32* ber);
```

- **Parameter**

rssi:
[Out] Signal quality level, 0~31 or 99. 99 indicates that the module is not registered on GSM network.

ber:
[Out] The bit error code of signal.

- **Return Value**

QL_RET_OK: indicates success.
QL_RET_ERR_INVALID_PARAMETER: indicates something wrong for input parameters.

5.13.5.4. RIL_NW_SetGPRSContext

This function sets a PDP foreground context.

- **Prototype**

```
s32 RIL_NW_SetGPRSContext(u8 foregroundContext);
```

- **Parameter**

foregroundContext
[in] Anumeric indicates which context will be set as foreground context. The range is 0-1.

- **Return Value**

RIL_AT_SUCCESS: send AT successfully.
RIL_AT_FAILED: send AT failed.
RIL_AT_TIMEOUT: send AT timeout.
RIL_AT_BUSY: sending AT.
RIL_AT_INVALID_PARAM: invalid input parameter.
RIL_AT_UNINITIALIZED: RIL is not ready, need to wait for MSG_ID_RIL_READY and then call *Ql_RIL_Initialize* to initialize RIL.

5.13.5.5. RIL_NW_SetAPN

This function sets the default APN of module.

- **Prototype**

```
s32 RIL_NW_SetAPN(u8 mode, u8* apn, u8* userName, u8* possword);
```

- **Parameter**

mode:
[in] 0 for CSD, 1 for GPRS.

apn:
[in] APN string.

userName:
[in] User name.

password:
[in] Password for APN.

- **Return Value**

QL_RET_OK: indicates success.
QL_RET_ERR_INVALID_PARAMETER: indicates something wrong for input parameters.

5.13.5.6. 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);
```

- **Parameter**

None.

- **Return Value**

RIL_AT_SUCCESS: send AT successfully.
RIL_AT_FAILED: send AT failed.
RIL_AT_TIMEOUT: send AT timeout.
RIL_AT_BUSY: sending AT.
RIL_AT_INVALID_PARAM: invalid input parameter.
RIL_AT_UNINITIALIZED: RIL is not ready, need to wait for MSG_ID_RIL_READY and then call *Ql_RIL_Initialize* to initialize RIL.

5.13.5.7. 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);
```

- **Parameter**

None.

- **Return Value**

RIL_AT_SUCCESS: send AT successfully.
RIL_AT_FAILED: send AT failed.

RIL_AT_TIMEOUT: send AT timeout.
RIL_AT_BUSY: sending AT.
RIL_AT_INVALID_PARAM: invalid input parameter.
RIL_AT_UNINITIALIZED: RIL is not ready, need to wait for MSG_ID_RIL_READY and then call *Ql_RIL_Initialize* to initialize RIL.

5.13.5.8. RIL_NW_GetOperator

This function gets the network operator that the module is registered to.

- **Prototype**

```
s32 RIL_NW_GetOperator(char* operator);
```

- **Parameter**

operator:
[Out] A string with max 16 characters, which indicates the network operator that the module registered to.

- **Return Value**

RIL_AT_SUCCESS: send AT successfully.
RIL_AT_FAILED: send AT failed.
RIL_AT_TIMEOUT: send AT timeout.
RIL_AT_BUSY: sending AT.
RIL_AT_INVALID_PARAM: invalid input parameter.
RIL_AT_UNINITIALIZED: RIL is not ready, need to wait for MSG_ID_RIL_READY and then call *Ql_RIL_Initialize* to initialize RIL.

5.13.6. GSM location API

The API functions in this section are declared in “ril_location.h”.

5.13.6.1. RIL_GetLocation

This function retrieves the longitude and latitude of the current place where the module is.

- **Prototype**

```
s32 RIL_GetLocation(CB_LocInfo cb_loc);
typedef void(*CB_LocInfo)(s32 result,ST_LocInfo* loc_info);
```

- **Parameter**

cb_loc:
Pointer to a callback function that tells the location information.

- **Return Value**

QL_RET_OK: indicates success.
QL_RET_ERR_INVALID_PARAMETER: indicates something wrong for input parameters.

5.13.7. Secure data API

The API functions in this section are declared in “ril_system.h”.

5.13.7.1. Ql_SecureData_Store

This function can be used to store some critical user data to prevent them from losing.

NOTES

1. OpenCPU has been designed with 13 blocks of system storage space to backup critical user data. Developers may specify the first parameter index [1-13] to specify different storage block. Among the storage blocks, 1~8 blocks can store 50 bytes for each block, 9~12 blocks can store 100 bytes for each block, and the 13th block can store 500 bytes.
2. User should not call this API function frequently, which is not good for life cycle of flash.

- **Prototype**

```
s32 Ql_SecureData_Store(u8 index , u8* pData, u32 len);
```

- **Parameter**

index:
[in] The index of the secure data block. The range is: 1~13.

pData:
[in] The data to be backed up. In 1~8 groups, every group can save 50 bytes at most. In 9~12 groups, every group can save 100 bytes at most. If index is 13, the user data can save 500 bytes at most.

len:
[in] The length of the user data. If the index is (1~8), then len<=50. If the index is (9~12), then len<=100. If the index is 13, then len<=500.

- **Return Value**

QL_RET_OK: this function succeeds.
QL_RET_ERR_PARAM: invalid paramter.
QL_RET_ERR_GET_MEM: the heap memory is no enough.

5.13.7.2. Ql_SecureData_Read

This functin reads secure data which is previously stored by *Ql_SecureData_Store*.

- **Prototype**

```
s32 Ql_SecureData_Read(u8 index, u8* pBuffer, u32 len);
```

- **Parameter**

index:
[in] The index of the secure data block. The range is: 1~13.

len:
[in] The length of the user data. If the index is (1~8), then len<=50. If the index is (9~12), then len<=100. If the index is 13, then len<=500.

- **Return Value**

If this function succeeds, the real read length is returned.

QL_RET_ERR_PARAM: invalid paramter.
QL_RET_ERR_GET_MEM: the heap memory is no enough.
Ql_RET_ERR_UNKOWN: unknown error.

5.13.8. System API

The API functions in this section are declared in “ril_system.h”.

5.13.8.1. RIL_QuerySysInitStatus

Queries the initialization status of the module.

- **Prototype**

```
s32 RIL_QuerySysInitStatus(s32* SysInitStatus);
```

- **Parameter**

SysInitStatus
[Out] System initialization status. 0/1/2/3, the initialization status value, is one value of *Enum_SysInitState*. Please refer to **AT+QINISTAT** in AT Commands Manual document for the meanings.

- **Return Value**

RIL_AT_SUCCESS: send AT successfully.
RIL_AT_FAILED: send AT failed.
RIL_AT_TIMEOUT: send AT timeout.
RIL_AT_BUSY: sending AT.
RIL_AT_INVALID_PARAM: invalid input parameter.
RIL_AT_UNINITIALIZED: RIL is not ready, need to wait for MSG_ID_RIL_READY and then call *Ql_RIL_Initialize* to initialize RIL.

5.13.8.2. RIL_GetPowerSupply

This function queries the battery balance, and the battery voltage.

- **Prototype**

```
s32 RIL_GetPowerSupply(u32* capacity, u32* voltage);
```

- **Parameter**

capacity:
[Out] Battery balance, a percent, ranges from 1 to 100.

voltage:
[Out] Battery voltage, unit in mV.

- **Return Value**

RIL_AT_SUCCESS: send AT successfully.
RIL_AT_FAILED: send AT failed.
RIL_AT_TIMEOUT: send AT timeout.
RIL_AT_BUSY: sending AT.
RIL_AT_INVALID_PARAM: invalid input parameter.
RIL_AT_UNINITIALIZED: RIL is not ready, need to wait for MSG_ID_RIL_READY and then call *Ql_RIL_Initialize* to initialize RIL.

5.13.8.3. RIL_GetIMEI

This function retrieves the IMEI number of module.

- **Prototype**

```
s32 RIL_GetIMEI(char* imei);
```

- **Parameter**

imei:
[Out] buffer to store the IMEI number. The length of buffer should be at least 15-byte..

- **Return Value**

RIL_AT_SUCCESS: send AT successfully.
RIL_AT_FAILED: send AT failed.
RIL_AT_TIMEOUT: send AT timeout.
RIL_AT_BUSY: sending AT.
RIL_AT_INVALID_PARAM: invalid input parameter.
RIL_AT_UNINITIALIZED: RIL is not ready, need to wait for MSG_ID_RIL_READY and then call *Ql_RIL_Initialize* to initialize RIL.

5.13.9. Audio API

5.13.9.1. RIL_AUD_SetChannel

This function sets the audio channel.

- **Prototype**

```
s32 RIL_AUD_SetChannel(Enum_AudChannel audChannel);
```

- **Parameter**

audChannel:
[Out] Audio channel, see *Enum_AudChannel*.

- **Return Value**

RIL_AT_SUCCESS: This function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

5.13.9.2. RIL_AUD_GetChannel

This function gets the audio channel.

- **Prototype**

```
s32 RIL_AUD_GetChannel(Enum_AudChannel *pChannel);
```

- **Parameter**

pChannel:
[Out] Audio channel, refer to *Enum_AudChannel*.

- **Return Value**

RIL_AT_SUCCESS: this function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

5.13.9.3. RIL_AUD_SetVolume

This function sets the volume level with the specified volume type.

- **Prototype**

```
s32 RIL_AUD_SetVolume(Enum_VolumeType volType, u8 volLevel);
```

- **Parameter**

volType:
[in] Volume type, see *Enum_VolumeType*.

volLevel:
[in] Volume level.

- **Return Value**

RIL_AT_SUCCESS: this function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

5.13.9.4. RIL_AUD_GetVolume

This function gets the volume level with the specified volume type.

- **Prototype**

```
s32 RIL_AUD_GetVolume(Enum_VolumeType volType, u8* pVolLevel);
```

- **Parameter**

volType:
[in] Volume type, see *Enum_VolumeType*.

pvolLevel:
[in] Volume level.

- **Return Value**

RIL_AT_SUCCESS, this function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

5.13.9.5. RIL_AUD_RegisterPlayCB

This function registers a callback function that will be invoked to indicate the playing result.

If you want to get a feedback (end indication or error code) for playing when calling APIs *RIL_AUD_PlayFile* and *RIL_AUD_PlayMem*, you can 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);
```

- **Parameter**

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: this function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

5.13.9.6. RIL_AUD_PlayFile

This function plays the specified audio file.

- **Prototype**

```
s32 RIL_AUD_PlayFile(char* filePath, bool isRepeated);
```

- **Parameter**

filePath:
[in] Source file name with filepath.

isRepeated:
[in] Repeat play mode.

- **Return Value**

RIL_AT_SUCCESS: this function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

5.13.9.7. RIL_AUD_StopPlay

This function stops playing the audio file.

- **Prototype**

```
s32 RIL_AUD_StopPlay(void);
```

- **Parameter**

None.

- **Return Value**

RIL_AT_SUCCESS: this function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

5.13.9.8. RIL_AUD_PlayMem

This function plays the specified audio data in RAM.

- **Prototype**

```
s32 RIL_AUD_PlayMem(u32 mem_addr, u32 mem_size, u8 aud_format, bool repeat);
```

- **Parameter**

mem_addr:
[in] RAM address of audio data.

mem_size:
[in] Size of audio data.

aud_format:
[in] Audio data format.

repeat:
[in] Play circularly or not.

- **Return Value**

RIL_AT_SUCCESS: this function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

5.13.9.9. RIL_AUD_StopPlayMem

This function stops playing the audio file.

- **Prototype**

```
s32 RIL_AUD_StopPlayMem(void);
```

- **Parameter**

None.

- **Return Value**

RIL_AT_SUCCESS: this function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

5.13.9.10. RIL_AUD_StartRecord

This function starts to record with the spedifed audio format. The recording data will be recorded into the specified file in UFS.

- **Prototype**

```
s32 RIL_AUD_StartRecord(char* fileName, Enum_AudRecordFormat format);
```

- **Parameter**

fileName:
[in] File name, which is used to store record data.

format:
[in] Record data format, one value of *Enum_AudRecordFormat*.

- **Return Value**

RIL_AT_SUCCESS: this function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

5.13.9.11. RIL_AUD_StopRecord

This function stops recording.

- **Prototype**

```
s32 RIL_AUD_StopRecord(void);
```

- **Parameter**

None.

- **Return Value**

RIL_AT_SUCCESS: this function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

5.13.9.12. RIL_AUD_GetRecordState

This function gets the current state of recorder.

- **Prototype**

```
s32 RIL_AUD_GetRecordState(u8* pState);
```

- **Parameter**

pState:
[Out] Recording state. 0 indicates the recorder is in idle state; 1 indicates the recorder is in recording.

- **Return Value**

RIL_AT_SUCCESS: this function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

5.14. GNSS API

5.14.1.1. RIL_GPS_Open

This function powers on/off GNSS.

- **Prototype**

```
s32 RIL_GPS_Open(u8 op);
```

- **Parameter**

op:
[in] Operation. 0: indicates power off GNSS; 1: indicates power on GNSS.

- **Return Value**

RIL_AT_SUCCESS: this function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

5.14.1.2. RIL_GPS_Read

This function queries the navigation information.

- **Prototype**

```
s32 RIL_GPS_Read(u8 *item, u8 *rdBuff);
```

- **Parameter**

item:
[in] Item, pointer to the query item.

rdBuff:
[in] Pointer to buffer, which stores the navigation information.

- **Return Value**

RIL_AT_SUCCESS: this function succeeds. Or, please refer to the definition of *Enum_ATSndError*.

6 Appendix A References

Table 8: Reference Documents

SN	Document Name
[1]	Quectel_MC60_AT_Commands_Manual
[2]	Quctel_MC60-OpenCPU_Hardware_Design
[3]	Quectel_QFlash_User_Guide
[4]	Quectel_OpenCPU_FOTA_Application_Note
[5]	OpenCPU_GCC_Installation_Guide
[6]	Quectel_OpenCPU_RIL_Application_Note
[7]	Quectel_OpenCPU_Watchdog_Application_Note
[8]	Quectel_OpenCPU_Security_Data_Application_Note

Table 9: Abbreviations

Abbreviation	Description
API	Application Programming Interface
App	OpenCPU Application
Core	Core System, OpenCPU Operating System
FOTA	Firmware Over the Air
GPIO	General Purpose Input Output
KB	Kilobytes
MB	Megabytes
MCU	Micro Control Unit

OS	Operating System
RAM	Random-Access Memory
RIL	Radio Interface Layer
ROM	Read-Only Memory
SDK	Software Development Kit
TCP/IP	Transfer Control Protocol / Internet Protocol
UART	Universal Asynchronous Receiver and Transmitter