BC26-OpenCPU

User Guide

NB-IoT Module Series

Rev. BC26-OpenCPU_User_Guide_V1.0

Date: 2018-09-13

Status: Released

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

Quectel Wireless Solutions Co., Ltd.
7th Floor, Hongye Building, No.1801 Hongmei Road, Xuhui District, Shanghai 200233, China
Tel: +86 21 5108 6236
Email: info@quectel.com

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

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

GENERAL NOTES

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

COPYRIGHT

About the Document

History

Revision	Date	Author	Description
1.0	2018-09-13	Lebron LIU/ Flame YANG	Initial

Contents

Table Index

Figure Index

1 Introduction

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

This document mainly introduces how to use OpenCPU solution on Quectel NB-IoT BC26 module.

2 OpenCPU Platform

2.1. System Architecture

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

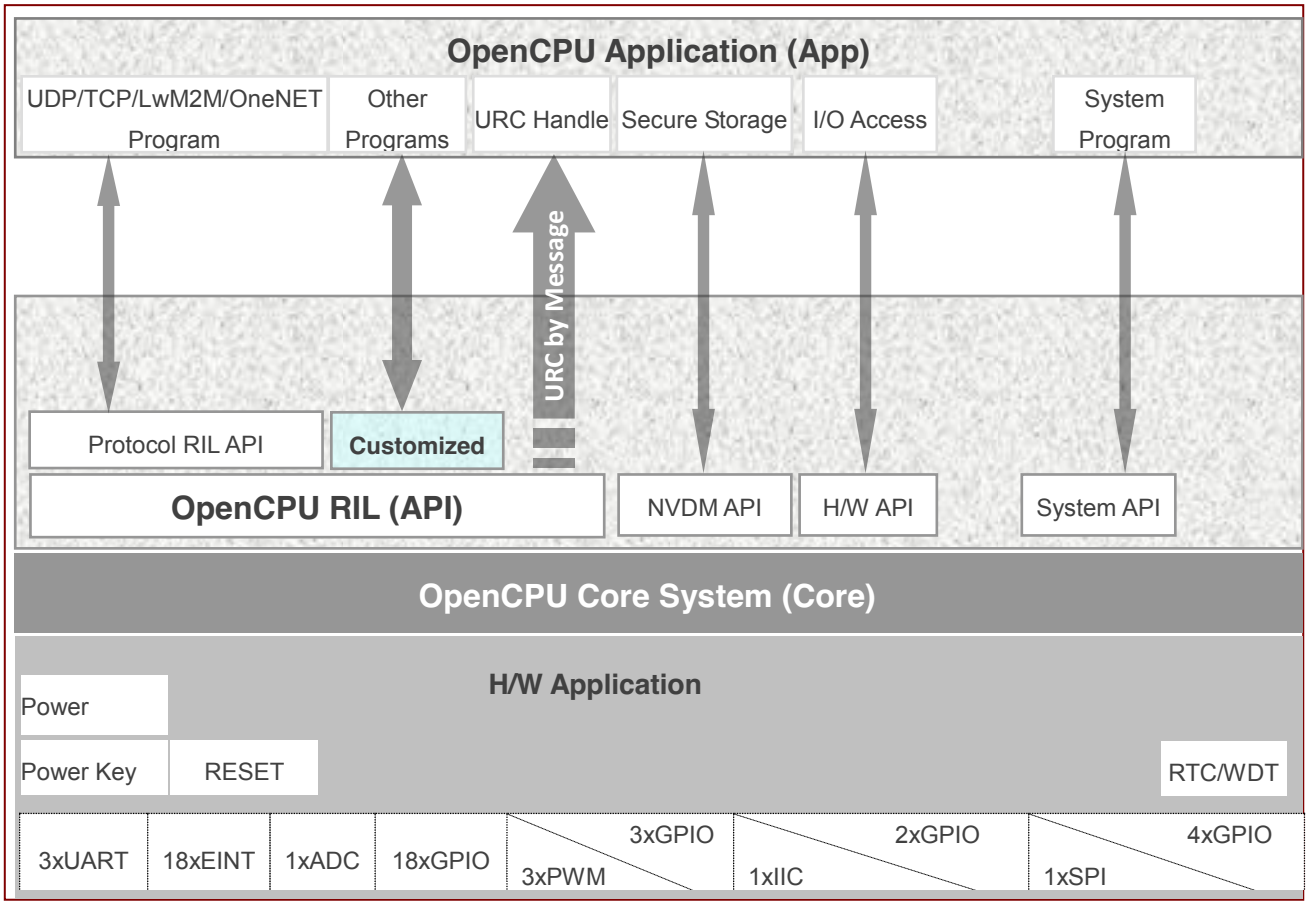


Figure 1: The Fundamental Principle of OpenCPU Software Architecture

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

OpenCPU Core System is a combination of hardware and software of NB-IoT module. It has built-in ARM Cortex-M4 processor, and has been built over FreeRTOS operating system, which has the characteristics of micro-kernel, real-time, multi-tasking, etc.

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

OpenCPU RIL is an open source layer, which enables developer 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. For more details, please refer to document *Quectel_OpenCPU_RIL_Application_Note*.

In OpenCPU RIL, all URC messages of module have already reinterpreted and the result is informed 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 ARM Cortext-M4 RISC 78MHz.

2.2.2. Memory Scheme

BC26-OpenCPU module builds in 4MB flash and 4MB RAM.

- User App Code Space: 200KB space available for image bin
- RAM Space: 100KB static memory and 300KB dynamic memory

2.3. Interfaces

2.3.1. Serial Interfaces

OpenCPU provides three UART ports: UART0 (Main UART), UART1 (Debug UART) and UART2 (Auxiliary UART). Please refer to ***Chapter 5.7.1*** for software API functions.

UART1 and UART2 provide debug function to enable Core system debugging. And UART1 supports hardware flow control. For more details, please refer to ***Chapter 5.8***.

2.3.2. GPIO

There are 18 I/O pins that can be configured into general purpose I/O. All pins are accessible under OpenCPU by API functions. Please refer to ***Chapter 5.7.2*** for details.

2.3.3. EINT

OpenCPU supports external interrupt input. All I/O pins can be configured into external interrupt inputs. But the EINT cannot be used for the purpose of highly frequent interrupt detection so as to avoid the module's unstability. The EINT pins can be accessed by APIs. Please refer to ***Chapter 5.7.3*** for details.

2.3.4. PWM

There are three I/O pins that can be configured into PWM pins. And 32K and 13M clock sources 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 is an analogue input pin that can be configured into ADC. The sampling period and count can be configured by an API. Please refer to ***Chapter 5.7.5*** for more details.

Please refer to *Quectel_BC26-OpenCPU_Hardware_Design* for the characteristics of ADC interface.

2.3.6. IIC

BC26-OpenCPU provides a hardware IIC interface. Please refer to ***Chapter 5.7.6*** for more details.

2.3.7. SPI

BC26-OpenCPU provides a hardware SPI interface. Please refer to ***Chapter 5.7.7*** for more details.

2.4. Development Environment

2.4.1. SDK

OpenCPU SDK provides 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

Customers may get the latest SDK package from Quectel Sales Representatives or Technical Supports.

2.5. 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.

2.5.1. Compiler & Compiling

2.5.1.1. Complier

OpenCPU uses GCC as the compiler, and the compiler edition is *arm-none-eabi-gcc v4.8.3*. The GCC compiler is provided under *tools* directory by default, and no additional installation is needed.

2.5.1.2. Compiling

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

```
make clean
make new
```

2.5.1.3. 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 shown in *build.log*:

```
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 no any compiling error during compiling, the prompt for successfully compiling will be given.

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

2.5.2. Download

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

2.5.3. How to Program

By default, the *custom* directory has been designed to store developers' source code files in SDK.

2.5.3.1. Program Composition

OpenCPU program consists of the aspects below.

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.

2.5.3.2. 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 use the RIL feature, you must initialize it by calling the following API
                //After receive 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_EGPRS_NW_STATE_IND:
                Ql_Debug_Trace("<-- EGPRS 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_NEW_SMS_IND:
                Ql_Debug_Trace("<-- New SMS Arrives: index=%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 which enables the Embedded Application to retrieve 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.

2.5.3.3. Makefile

In OpenCPU, the compiler compiles programs according to the definitions in makefile. The profile of makefile has been pre-designed and is ready for use. However, developers have to change some settings before compiling programs 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

2.5.3.4. How to Add .c File

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

2.5.3.5. How to Add Directory

If developers have to add a new directory in *custom*, please follow the steps below.

1. 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     \
```

2. 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 File

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

3.2. Base Data Type

Table 2: Base Data Type

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

u64	64-bit unsigned integer. This variable is declared as follows: typedef unsigned long lone u64;
float	Floating-point variable. This variable is declared in *math.h*.

4 System Configuration

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

Table 3: System Config File List

Config File	Description
custom_feature_def.h	OpenCPU features enable. Now only include RIL. Developers generally do not need to change this file.
custom_gpio_cfg.h	Configurations for GPIO initialization status
custom_heap_cfg.h	Definition of heap size (no need to configure, customers use system heap by default)
custom_task_cfg.h	Multitask configuration
custom_sys_cfg.c	Other system configurations, including power key, specified GPIO pin for external watchdog, and setting working mode of debug port.

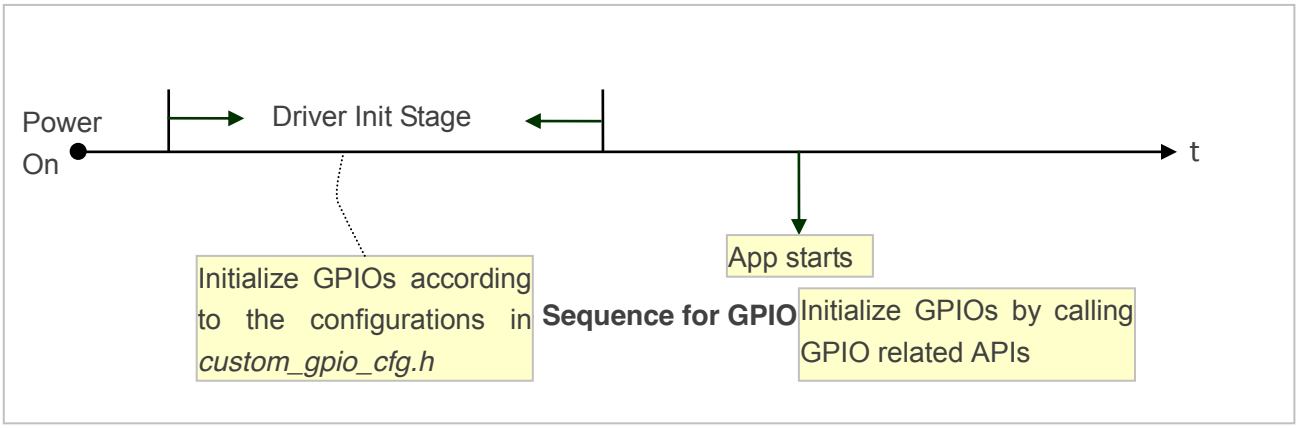
4.1. Configuration of Tasks

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 maximally ten subtasks.

Developers should calling these functions: *Ql_Sleep()*, *Ql_OS_TakeSemaphore()* and *Ql_OS_TakeMutex()* cautionly. These functions will block the task, which makes 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 of GPIOs

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



Sequence for GPIO

4.3. Configuration of Customization Items

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 . PWRKEY and Watchdog configuration features are currently not supported.

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

Table 4: Customization Items

Item	Type(T)	Length (L)	Default Value	Possible Value	Description
App Enabling	SYS_CONFIG_APP_ENABLE_ID	4	APP_ENABLE	APP_ENABLE APP_DISABLE	App enable config
PWRKEY Pin Config	SYS_CONFIG_PWRKEY_DATA_ID	2	TRUE TRUE	TRUE/FALSE	Currently not supported
GPIO for WTD Config	SYS_CONFIG_WATCHDOG_DATA_ID	8	PINNAME_GPIO0	One value of Enum_PinName	GPIO for feeding WTD.
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. 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 during which the module is 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 Runing	App
Upgrading App By DFOTA	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 or another GPIO to connect to the external watchdog.
PINNAME_END         //Specify another pin for watchdog if needed.
};
```

4.3.2. Debug Port Working Mode Configuration

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

Usually developers do not need to use ADVANCE_MODE when there are no requirements from support engineers. If needed, please refer to *Genie Logging Tool* 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 APIs

The header file *ql_system.h* declares system-related APIs. These functions are essential to customers' applications. Please make sure the header file is included when using these functions.

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

5.1.1. Usage

This section introduces some important operations and the API functions 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 or a customized message.

5.1.1.2. Send Message

Developers can call *Ql_OS_SendMessage* or *Ql_OS_SendMessageFromISR* to send messages to other tasks. To send message, developers have to define a message ID. In OpenCPU, user message ID must greater 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);
```

```
Ql_OS_SendMessageFromISR(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. A task can only own one mutex object at a time. For example, to prevent two tasks from being written to shared memory at the same time, each task waits for ownership of a mutex object before executing the code for accessing the memory. After writing to the shared memory, the task releases the mutex object.

Step 1: Create a mutex. Developers can call *Ql_OS_CreateMutex* to create a mutex.
Step 2: Get a specified mutex.. If developers want to use mutex mechanism for programming, they can call *Ql_OS_TakeMute* to get a specified mutex ID.
Step 3: Release a specified mutex. Developers can call *Ql_OS_GiveMutex* to release a 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 wait 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 a semaphore. Developers can call *Ql_OS_CreateSemaphore* to create a semaphore.
Step 2: Get a specified semaphore. If developers want to use semaphore mechanism for programming, they can call *Ql_OSTakeSemaphore* to get a specified semaphore ID.
Step 3: Release a specified semaphore. Developers can call *Ql_OS_GiveSemaphore* to release a 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 2 blocks of system storage space to backup critical user data. Each block can

store 50 bytes data.

Developers may call *Ql_SecureData_Store* to backup data, and call *Ql_SecureData_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)
```

- **Parameter**

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 timeout interval elapses. The sleep time should not be too long, because if the task is suspended too long, and it may receive too many messages to be crushed.

- **Prototype**

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

- **Parameter**

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

- **Return Value**

None.

5.1.2.3. Ql_GetSDKVer

This function gets the version ID of the SDK. The SDK version ID is no more than 20 characters in string format, and is end with '\0'.

- **Prototype**

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

- **Parameter**

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

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

- **Return Value**

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

5.1.2.4. 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 the waiting state.

- **Prototype**

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

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

- **Parameter**

msg:
[In] Pointer to the “ST_MSG” object.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.

5.1.2.5. Ql_OS_SendMessageFromISR

This function sends messages from interrupt routine. The destination task receives messages with *Ql_OS_GetMessage.*

- **Prototype**

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

- **Parameter**

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: indicates the function is executed successfully.

5.1.2.6. 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)
```

- **Parameter**

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: indicates the function is executed successfully.

5.1.2.7. 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(void)
```

- **Return Value**

A handle of created mutex, 0 indicates failure.

5.1.2.8. Ql_OS_TakeMutex

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

- **Prototype**

```
void Ql_OS_TakeMutex(u32 mutexId,u32 block_time)
```

- **Parameter**

mutexid:
[In] Destination mutex to be taken.

Block_time:
[In] The time to be blocked for waiting to take specified mutex.

- **Return Value**

None.

5.1.2.9. Ql_OS_GiveMutex

This function releases an instance of a specified mutex.

- **Prototype**

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

- **Parameter**

mutexid:
[In] Destination mutex to be given.

- **Return Value**

None.

5.1.2.10. 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(u32 maxCount,u32 InitialCount)
```

- **Parameter**

maxCount:
[In] The max count of semaphore.

InitialCount:
[In] The initial count of a specified semaphore.

- **Return Value**

A handle of created semaphore. 0 indicates failure.

5.1.2.11. Ql_OS_TakeSemaphore

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

- **Prototype**

```
u32 Ql_OSTakeSemaphore(u32 semId, u32 block_time)
```

- **Parameter**

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

block_time*:*
[In] The time to be blocked for waiting to take specified semaphore.

- **Return Value**

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

5.1.2.12. Ql_OS_GiveSemaphore

This function releases an instance of a specified semaphore.

- **Prototype**

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

- **Parameter**

semId:
[In] Destination semaphore to be given.

- **Return Value**

None.

5.1.2.13. Ql_OS_CreateEvent

This function creates a specified type of event.

- **Prototype**

```
u32 Ql_OS_CreateEvent(void);
```

- **Return Value**

An event ID that identifies this event is unique.

5.1.2.14. 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,u32 block_time);
```

- **Parameter**

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

evtFlag:
Event flag type. Please refer to *Enum_EventFlag*.

block_time:
The time to be blocked for waiting a specified event .

- **Return Value**

Zero indicates the function is executed successfully and nonzero indicates failed to execute the function.

5.1.2.15. Ql_OS_SetEvent

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

- **Prototype**

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

- **Parameter**

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

evtFlag:
Event flag type. Please refer to *Enum_EventFlag*.

- **Return Value**

Zero indicates the function is executed successfully and nonzero indicates failed to execute the function.

5.1.2.16. Ql_SetLastErrorCode

This function sets an error code.

- **Prototype**

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

- **Parameter**

errCode:
[In] Error code.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_FATAL: indicates failed to set error code.

5.1.2.17. Ql_GetLastErrorCode

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

- **Prototype**

```
s32 Ql_GetLastErrorCode(void)
```

- **Parameter**

None.

- **Return Value**

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

5.1.2.18. Ql_OS_GetCurrenTaskLeftStackSize

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

- **Prototype**

```
u32 Ql_OS_GetCurrenTaskLeftStackSize(void)
```

- **Parameter**

None.

- **Return Value**

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

5.1.2.19. Ql_SecureData_Store

This function can be used to stroe some critical user data to prevent them from losing. OpenCPU has designed 2 blocks of system storage space to backup critical user data. Developer may specify the first parameter index [1-2] to specify different storage block. Each storage block can store 50 bytes data.

- **Prototype**

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

- **Parameter**

index:
[In] the index of the secure data block. The range is: 1~2.

pData:
[In] The data to be backed up. Every storage block can save 50 bytes at most.

len:
[In] The length of the user data. The maximum value of this parameter is 50.

- **Return Value**

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

5.1.2.20. Ql_SecureData_Read

This function reads the user data which is backed up by *Ql_SecureData_Store API*.

- **Prototype**

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

- **Parameter**

index:
[In] The index of the secure data block. The range is 1~2.

len:
[In] The length of the user data. The maximum value of this parameter is 50.

- **Return Value**

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

5.1.3. Possible Error Code

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

```
/************************************************************
* Error Code Definition
 ************************************************************/
typedef enum {
    OS_SUCCESS,
    OS_ERROR,
    OS_Q_FULL,
    OS_Q_EMPTY,
    OS_SEM_NOT_AVAILABLE,
    OS_WOULD_BLOCK,
    OS_MESSAGE_TOO_BIG,
    OS_INVALID_ID,
    OS_NOT_INITIALIZED,
```

```
    OS_INVALID_LENGHT,
    OS_NULL_ADDRESS,
    OS_NOT_RECEIVE,
    OS_NOT_SEND,
    OS_MEMORY_NOT_VALID,
    OS_NOT_PRESENT,
    OS_MEMORY_NOT_RELEASE
} Enum_OS_ErrCode;
```

5.1.4. Examples

1. Mutex Example

```
static int    s_iMutexId = 0;

//Create the mutex first
s_iMutexId = Ql_OS_CreateMutex();

void MutextTest(int iTaskId)    //Two task Run this function at the same time
{

     //Get the mutex
    Ql_OS_TakeMutex(s_iMutexId,0xFFFFFFFF);

     //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_iTestSemInitial = 3; //Set the initial value of semaphore is 3
static int s_iTestSemMaxNum =4; //Set the maximum semaphore number is 4

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

    //Get the mutex
    iRet = Ql_OS_TakeSemaphore(s_iSemaphoreId, 0xFFFFFFFF);//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 APIs

OpenCPU 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 00:00:00 on January 1st, 1970 as a standard point in time.

5.2.2. API Functions

Time struct is defined as below:

```
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, the timezone=-5.

5.2.2.1. Ql_SetLocalTime

This function sets the current local date and time.

- **Prototype**

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

- **Parameter**

datetime:
[In] Pointer to the ST_Time object.

- **Return Value**

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

5.2.2.2. Ql_GetLocalTime

This function gets the current local date and time.

- **Prototype**

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

- **Parameter**

dateTime:
[Out] Pointer to the ST_Time object.

- **Return Value**

If the function is executed successfully, the current local date and time will be returned. NULL means failure.

5.2.2.3. Ql_Mktime

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

- **Prototype**

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

- **Parameter**

dateTime:
[In] Pointer 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 1st, 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 1st, 1970.

pOutDateTime:
[Out] Pointer to the ST_Time object.

- **Return Value**

If the function is executed successfully, the current local date and time will be returned. NULL means operation failure.

5.2.3. Example

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

```
s32 ret;
u64 sec;
ST_Time datetime, *tm;
datetime.year=2013;
datetime.month=6;
```

```
datetime.day=12;
datetime.hour=18;
datetime.minute=12;
datetime.second=13;
datetime.timezone=-8;

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

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

//Get total seconds elapsed since 1970.01.01 00:00:00
sec=Ql_Mktime(tm);
Ql_Debug_Trace("\r\n<--Ql_Mktime,sec=%lld -->\r\n",sec);

//Convert the seconds elapsed since 1970.01.01 00:00:00 to local date and time
tm=Ql_MKTime2CalendarTime(sec, & datetime);
Ql_Debug_Trace("<--%d/%d/%d %d:%d:%d %d -->\r\n",tm->year, tm->month, tm->day, tm->hour, tm
->minute, tm->second, tm->timezone);
```

5.3. Timer APIs

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

Developers use *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 an integral multiple of 10ms.

5.3.2. API Functions

5.3.2.1. Ql_Timer_Register

This function registers a Common Timer. Each task supports 10 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. It must be ensured 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 developers when the timer arrives.

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

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_PARAM: indicates a parameter error.
QL_RET_ERR_INVALID_TIMER: indicates the timer is invalid.
QL_RET_ERR_TIMER_FULL: indicates all timers are used up.

5.3.2.2. Ql_Timer_RegisterFast

This function registers a Fast Timer. It only supports 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. It must be ensured 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 this function is executed successfully.
QL_RET_ERR_PARAM: indicates a parameter error.
QL_RET_ERR_INVALID_TIMER: indicates the timer is invalid.
QL_RET_ERR_TIMER_FULL: indicates all timers are used up.

5.3.2.3. Ql_Timer_Start

This function starts up a 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, which must be registered.

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

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

- **Return Value**

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

QL_RET_ERR_INVALID_TIMER: indicates the timer invalid.
QL_RET_ERR_INVALID_TASK_ID: indicates the current task is not the timer registered task.

5.3.2.4. Ql_Timer_Stop

This function stops a 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 this function is executed successfully.
QL_RET_ERR_PARAM: indicates a parameter error.
QL_RET_ERR_INVALID_TIMER: indicates the timer is 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 codes show how to register 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. RTC/PSM_EINT APIs

RTC/PSM_EINT can be used to wakeup the module from deep sleep mode. When RTC/PSM_EINT event occurred, the registered callback will be called to notify users.

5.4.1. Usage

5.4.1.1. RTC/PSM_EINT Control

Ql_Rtc_RegisterFast function is used to register an RTC event callback.
Ql_Rtc_Start function is used to start an RTC timer.
Ql_Rtc_Stop function is used to stop an RTC timer.
Ql_Psm_Eint_Register function is used to register a PSM_EINT event callback.

5.4.2. API Functions

5.4.2.1. Ql_Rtc_RegisterFast

This function registers an RTC timer with dedicated ID. When the RTC timer expires, the registered callback will be called.

- **Prototype**

```
s32 Ql_Rtc_RegisterFast(u32 rtcId, Callback_Rtc_Func callback_onTimer, void* param)
```

- **Parameter**

rtcId:
[In] RTC timer ID. It must be greater than 0x200. And please make sure that the ID is not the same with the ID that registered by *Ql_Rtc_RegisterFast*.

callback_onTimer:
[Out] Notify the application when the RTC timer expires.

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

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QL_RET_ERR_PARAM: indicates a parameter error.
QL_RET_ERR_INVALID_TIMER: indicates that the timer ID is invalid.
QL_RET_ERR_FATAL: indicates this function is failed.

5.4.2.2. Ql_Rtc_Start

This function starts an RTC timer.

- **Prototype**

```
s32 Ql_Rtc_Start(u32 rtcId, u32 interval, bool autoRepeat);
```

- **Parameter**

rtcId:
[In] Rtc timer ID, which must be registered.

interval:
[In] Set the interval of the timer. Unit: ms. This value must be a multiple of 100ms.

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

- **Return Value**

QL_RET_OK: indicates this function is executed successfully..
QL_RET_ERR_PARAM: indicates a parameter error.
QL_RET_ERR_INVALID_TIMER: indicates that the timer ID is invalid.
QL_RET_ERR_FATAL: indicates this function is failed.

5.4.2.3. Ql_Rtc_Stop

This function stops an RTC timer.

- **Prototype**

```
s32 Ql_Rtc_Stop(u32 rtcId);
```

- **Parameter**

rtcId:
[In] Rtc timer ID, which must be registered.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully..
QL_RET_ERR_PARAM: indicates a parameter error.
QL_RET_ERR_INVALID_TIMER: indicates that the timer ID is invalid.
QL_RET_ERR_FATAL: indicates this function is failed.

5.4.2.4. Ql_Psm_Eint_Register

This function registers a PSM_EINT event notification. When PSM_EINT is triggered (falling edge), the registered callback will be called.

- **Prototype**

```
s32 Ql_Psm_Eint_Register(Callback_Psm_Eint_Func   callback_psm_eint,void* param);
```

- **Parameter**

callback_psm_eint:
[Out] Notify the application when the PSM_EINT pin is triggered.

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

- **Return Value**

QL_RET_OK: indicates this function is executed successfully..
QL_RET_ERR_PARAM: indicates a parameter error.
QL_RET_ERR_INVALID_TIMER: indicates that the timer ID is invalid.
QL_RET_ERR_FATAL: indicates this function is failed.

5.4.3. Example

The following sample codes show how to register an RTC timer and PSM_EINT event callback.

```
ret = Ql_Psm_Eint_Register(callback_psm_eint,NULL);
APP_DEBUG("psm_eint register , ret=%d\r\n",ret);
 // Register & open UART port
ret = Ql_GetPowerOnReason();

APP_DEBUG("power on reason, ret=%d\r\n",ret);
if(ret == QL_SYS_RESET)
{
   ret = Ql_Rtc_RegisterFast(RTC_ID_USER_START,rtc_callback,NULL);
   if (ret < QL_RET_OK)
   {
        APP_DEBUG("RTC register failed, ret=%d\r\n");
   }
   ret = Ql_Rtc_Start(RTC_ID_USER_START, 30*1000, TRUE);
   if (ret < QL_RET_OK)
   {
        APP_DEBUG("RTC start failed, ret=%d\r\n",ret);
   }
}
```

5.5. Power Management APIs

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

5.5.1. Usage

5.5.1.1. Sleep Mode

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

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

5.5.2. API Functions

5.5.2.1. Ql_SleepEnable

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

idle state.

- **Prototype**

```
s32 QI_ SleepEnable(void)
```

- **Parameter**

None.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QI_RET_NOT_SUPPORT: indicates the function is not supported in currently used version.

5.5.2.2. QI_SleepDisable

This function disables the sleep mode of module.

- **Prototype**

```
s32 QI_ SleepDisable(void)
```

- **Parameter**

None.

- **Return Value**

QL_RET_OK: indicates this function is executed successfully.
QI_RET_NOT_SUPPORT: this function is not supported.

5.5.2.3. QI_GetPowerOnReason

This function is used to query the module's power-on reason (reset or wake up from deep sleep).

- **Prototype**

```
s32 QI_ GetPowerOnReason(void)
```

- **Parameter**

None.

- **Return Value**

QL_SYS_RESET: indicates system reset occurs.
QL_DEEP_SLEEP: indicates wake up from deep sleep.

5.5.3. Example

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

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

5.6. Memory APIs

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

The dynamic memory is system heap space. And the maximum available system heap of application is 300KB.

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

5.6.1. 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 memory block that is previously allocated by calling *Ql_MEM_Alloc()*.

5.6.2. API Functions

5.6.2.1. 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 memory bytes to be allocated.

- **Return Value**

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

5.6.2.2. Ql_MEM_Free

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

- **Prototype**

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

- **Parameter**

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

- **Return Value**

None.

5.6.3. Example

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

```
char *pch=NULL;

//Allocate memory
```

```
pch=(char*)QI_MEM_Alloc(1024);
if (pch !=NULL)
{
    QI_Debug_Trace("Successfully apply for memory, pch=0x%x\r\n", pch);
}else{
    QI_Debug_Trace("Fail to apply for memory, size=%d\r\n", 1024);
}
//Free memory
QI_MEM_Free(pch);
pch=NULL;
```

5.7. Hardware Interface APIs

5.7.1. UART

5.7.1.1. UART Overview

In OpenCPU, the physical UART ports can be applied to connect to external devices. The working chart of UARTs is shown below:

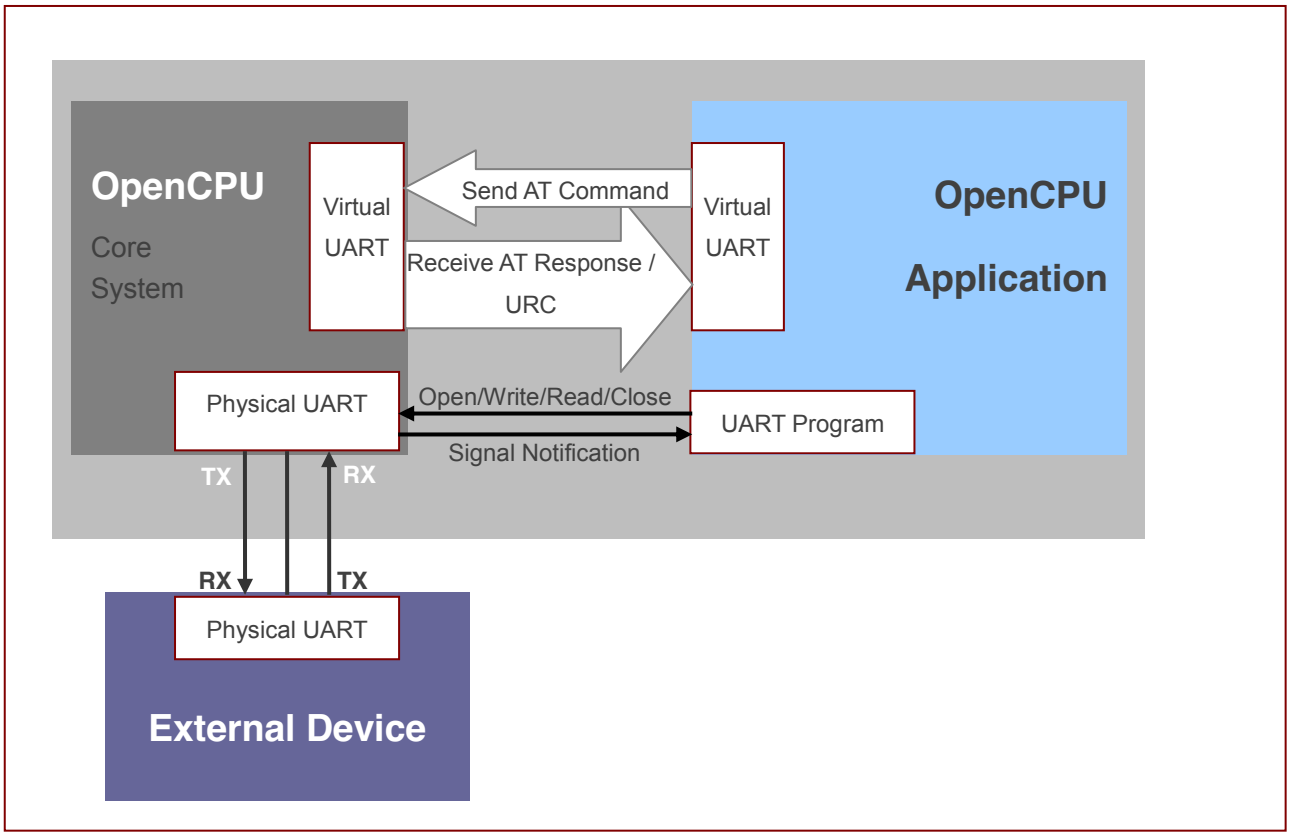


Figure 3: The Working Chart of UARTs

5.7.1.2. UART Usage

The following illustrates a few simple steps for physical UART usage:

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 be 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, the application can continue to send data, and the previously unsent data should be resent.
Step 4: Deal with the UART's notification in the callback function. If the notification type is EVENT_UART_READY_TO_READ, 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 QI_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)
```

- **Parameter**

port:
[In] Port name.

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

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

pinLevel:
[Out] This parameter indicates the related pin's current level if the event type is EVENT_UART_RI_IND or

EVENT_UART_DCD_IND or EVENT_UART_DTR_IND. Otherwise it has no meaning, and please just ignore it.

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

- **Return Value**

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

5.7.1.3.2. Ql_UART_Open

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

- **Prototype**

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

```
typedef enum {
    FC_NONE=1,    // None Flow Control
    FC_HW,        // Hardware Flow Control
    FC_SW         // Software Flow Control
} Enum_FlowCtrl;
```

- **Parameter**

port:
[In] Port name.

baudrate:
[In] The baud rate of the UART to be opened.
The physical UART supports baud rates in unit of bps as follows: 75, 150, 300, 600, 1200, 2400, 4800, 7200, 9600, 14400, 19200, 28800, 38400, 57600, 115200, 230400, 460800, 921600.

flowCtrl:
[In] Please refer to *Enum_flowCtrl* for the physical UART ports. Only UART_PORT1 and UART_PORT2 support 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. The task that calls this function will own the specified UART port.

- **Prototype**

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

- **Parameter**

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 port 1/2 (UART_PORT1/UART_PORT2) support hardware flow control.

- **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 a specified UART port. When the number of bytes actually sent is less than that has been sent, the application should stop sending data, and application (in callback function) will receive an event EVENT_UART_READY_TO_WRITE later. After receiving this event the 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)
```

- **Parameter**

port:
[In] Port name

data:
[In] Pointer to data to write.

writeLen:
[In] The length of the data to be written. The maximum data length of the receive buffer for physical UART

port is 2048 bytes.

- **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 a 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)
```

- **Parameter**

port:
[In] Port name

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

readLen:
[In] The length of the data to be read. The maximum data length of the receive buffer for physical UART port is 1024 bytes. And the buffer size cannot be modified programmatically in application.

- **Return Value**

Number of bytes actually read. If *readLen* equals to the actual read length, developers should 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_Close

This function closes a 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 an example to illustrate how to use the UART port.

```
//Write the call back function, for deal 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;
      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 18 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
3	PINNAME_SPI_MISO	I/PD	SPI_MST0_MISO	GPIO	EINT	
4	PINNAME_SPI_MOSI	I/PD	SPI_MST0_MOSI	GPIO	EINT	
5	PINNAME_SPI_SCLK	I/PD	SPI_MST0_SCLK	GPIO	EINT	
6	PINNAME_SPI_CS	I/PD	SPI_MST0_CS	GPIO	EINT	PWM1
16	PINNAME_NETLIGHT	I/PU	PWM0	GPIO	EINT	
20	PINNAME_RI	I/PD	IIC0_SCL	GPIO	EINT	
21	PINNAME_DCD	I/PD	IIC0_SDA	GPIO	EINT	
22	PINNAME_CTS_AUX	I/PD		GPIO	EINT	
23	PINNAME_RTS_AUX	I/PD	PWM2	GPIO	EINT	
26	PINNAME_GPIO1	I/PD		GPIO	EINT	
28	PINNAME_RXD_AUX	I/PD		GPIO	EINT	
29	PINNAME_TXD_AUX	I/PD		GPIO	EINT	
30	PINNAME_GPIO2	I/PD		GPIO	EINT	
31	PINNAME_GPIO3	I/PD	PWM3	GPIO	EINT	
32	PINNAME_GPIO4	I/PD		GPIO	EINT	
33	PINNAME_GPIO5	I/PD		GPIO	EINT	
38	PINNAME_RX_DBG	I/PD		GPIO	EINT	
39	PINNAME_TX_DBG	I/PD		GPIO	EINT	

- 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" means PWM output function.

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*** and the other way is to call GPIO related API to initialize after App starts.

The following codes show the PINNAME_NETLIGHT, PINNAME_STATUS and PINNAME_GPIO0 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 to set a specified pin as the GPIO function, and then initialize the configurations, including direction, level and pull selection.
Step 2: GPIO control. When the pin is initialized as GPIO, developers can call the GPIO related APIs to change the GPIO level.
Step 3: Release the pin. If developers do not want to use this pin no longer, and need to use this pin for other purpose (such as PWM, EINT), they 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)
```

- **Parameter**

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

5.7.2.5.2. Ql_GPIO_GetLevel

This function gets the level of a specified GPIO.

- **Prototype**

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

- **Parameter**

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

- **Return Value**

Return the level of a 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 a specified GPIO.

- **Prototype**

```
s32 Ql_GPIO_SetLevel(PinName pinName, PinLevel level)
```

- **Parameter**

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

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

- **Return Value**

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

5.7.2.5.4. Ql_GPIO_GetDirection

This function gets the direction of a specified GPIO.

- **Prototype**

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

- **Parameter**

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 a specified GPIO.

- **Prototype**

```
s32 Ql_GPIO_SetDirection(PinName pinName,PinDirection dir)
```

- **Parameter**

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

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

- **Return Value**

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

5.7.2.5.6. Ql_GPIO_SetPullSelection

This function sets the pull selection of a specified GPIO.

- **Prototype**

```
s32 Ql_GPIO_SetPullSelection(PinName pinName,PinPullSel pullSel)
```

- **Parameter**

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

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

- **Return Value**

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

5.7.2.5.7. Ql_GPIO_Uninit

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

- **Prototype**

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

- **Parameter**

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

- **Return Value**

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

5.7.2.6. Example

This chapter gives an example to illustrate 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 eighteen external interrupt pins, and please refer to ***Chapter 5.7.2.2*** for details. The external interrupt have higher priority, so frequent interruption is not allowed. It is strongly recommended that the interrupt frequency is not more than 2, and too frequent interrupt will make other tasks cannot be scheduled, which probably leads unexpected exception.

NOTE

The interrupt response time is 50ms by default, and can be re-programmed to a greater value in OpenCPU. However, it is 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. Developers 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 configure the software debounce time and set level-triggered interrupt mode.

Step 3: Interrupt handle. The interrupt callback function will be called if the level has changed. And developers can process something in the handler.

Step 4: Mask the interrupt. When developers do not want external interrupt they can use the *Ql_EINT_Mask* function to disable the external interrupt, and 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 purpose after it is released. This step is optional.

5.7.3.3. API Functions

5.7.3.3.1. QI_EINT_Register

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

- **Prototype**

```
s32 QI_EINT_Register(PinName eintPinName, Callback_EINT_Handle callback_eint,void* customParam)
typedef void (*Callback_EINT_Handle)(PinName eintPinName, PinLevel pinLevel, void* customParam)
```

- **Parameter**

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

callback_eint:
[In] The interrupt handler.

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

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

- **Return Value**

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

5.7.3.3.2. QI_EINT_RegisterFast

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

- **Prototype**

```
s32 QI_EINT_RegisterFast(PinName eintPinName, Callback_EINT_Handle callback_eint, void* customParam)
```

- **Parameter**

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

callback_eint:
[In] The interrupt handler.

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

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

- **Return Value**

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

5.7.3.3.3. Ql_EINT_Init

This function initializes an external interrupt function.

- **Prototype**

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

- **Parameter**

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 1ms. Not supported now.

swDebounce:
[In] Software debounce. Not supported now.

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

- **Return Value**

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

5.7.3.3.4. Ql_EINT_Uninit

This function releases a specified EINT pin.

- **Prototype**

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

- **Parameter**

eintPinName:
[In] EINT pin name.

- **Return Value**

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

5.7.3.3.5. Ql_EINT_GetLevel

This function gets the level of a specified EINT pin.

- **Prototype**

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

- **Parameter**

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 a specified EINT pin.

- **Prototype**

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

- **Parameter**

eintPinName:
[In] EINT pin name.

- **Return Value**

None.

5.7.3.3.7. Ql_EINT_Unmask

This function unmasks a specified EINT pin.

- **Prototype**

```
void Ql_EINT_Unmask(PinName eintPinName)
```

- **Parameter**

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.

        //Developers need to unmask the interrupt again, because PINNAME_DTR pin interrupt is
initialized as auto mask.
```

```
        Ql_EINT_Unmask(eintPinName);
        if(*((s32*)customParam) >= 3)
        {
            //If developers do not want the interrupt, 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 developers do not want the interrupt, 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);

    //Initialize some parameters and set auto mask to 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);

    //Initialize some parameters and set auto mask to 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 have three PWM pins, and please refer to ***Chapter 5.7.2.2*** for details. The PWM pin has two clock sources: one is 32K (the exact value is 32768Hz) and the other is 13M.

5.7.4.2. PWM Usage

The following steps illustrate how to use the PWM function:

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

5.7.4.3. API Functions

5.7.4.3.1. QI_PWM_Init

This function initializes a PWM pin.

- **Prototype**

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

- **Parameter**

pwmPinName:
[In] PWM pin name and it only can be PINNAME_NETLIGHT.

pwmSrcClk:
[In] PWM clock source. One value of *Enum_PwmSource*.

pwmDiv:
[In] Clock source division. One value of *Enum_PwmSourceDiv*.

lowPulseNum:
[In] Set the number of clock cycles to stay at low level. The result of *lowPulseNum* plus *highPulseNum* is less than 8193.

highPulseNum:
[In] Set the number of 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 indicates 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)
```

- **Parameter**

pwmPinName:
[In] PWM pin name. One value of *Enum_PinName*.

- **Return Value**

QL_RET_OK indicates 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)
```

- **Parameter**

pwmPinName:
[In] PWM pin name. One value of *Enum_PinName*.

pwmOnOff:
[In] PWM waveform output enable/disable control.

- **Return Value**

QL_RET_OK indicates 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;

    //Initialize 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);

    //Enable 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);
        //Disable PWM waveform output.
    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 it is no longer needed.
    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. Please refer to *Quectel_BC26-OpenCPU_Hardware_Design* for the pin definitions and ADC hardware characteristics. The voltage range that can be detected is 0mV~1400mV.

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 invoked after ADC has sampled count times.

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

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. Call this API function again with a disabling 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 outputs 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)
```

- **Parameter**

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

callback_adc:
[In] Callback funtion, which will be called when the module outputs the ADC value .

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

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

- **Return Value**

QL_RET_OK indicates 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 outputs the ADC value, and the value is the average of the sampling value .

- **Prototype**

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

- **Parameter**

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

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

interval:
[In] Interval of each internal sampling. Unit: ms. The minimum value is 200 and this means the ADC report frequency must be less than 1Hz.

- **Return Value**

QL_RET_OK indicates 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)
```

- **Parameter**

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

enable:
[In] Sampling control. 1 means start sampling, and 0 means stop sampling.

- **Return Value**

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

5.7.5.4. Example

The following example demonstrates the use of 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 to sample, if you 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, the each internal sampling interval
    ret=Ql_ADC_Init(PIN_ADC0, 5, 200);//So the ADC0 report the ADC value frequency 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 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 *Ql_IIC_Init* function to initialize an IIC channel, including the specified

GPIO pins for IIC and an IIC channel number.

Step 2: Configure IIC interface. Call *QI_IIC_Config* to configure parameters that the slave device needs. Please refer to the API decription for extend 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. QI_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 QI_IIC_Init(u32 chnnlNo,PinName pinSCL,PinName pinSDA, u32 IICtype)
```

- **Parameter**

chnnlNo:
[In] IIC channel number. 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, while 1 means IIC contronller.

- **Return Value**

QL_RET_OK indicates 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)
```

- **Parameter**

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

isHost:
[In] Must be ture. Only host mode is supported.

slaveAddr:
[In] Slave address.

speed:
[In] Just for IIC controller, and the parameter can be ingored if simulated IIC is used.

- **Return Value**

QL_RET_OK indicates 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)
```

- **Parameter**

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

slaveAddr:
[In] Slave address.

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

len:
[In] Number of bytes to write. If *IICtype*=1,1<*len*<8 because the IIC contronller supports 8 bytes at most for onw-time transmission.

- **Return Value**

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

5.7.6.3.4. Ql_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)
```

- **Parameter**

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

slaveAddr:
[In] Slave address.

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

len:
[Out] Number of bytes to read. If *IICtype*=1,1<*len*<8 because the IIC contronller supports 8 bytes at most for one-time transmission.

- **Return Value**

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

5.7.6.3.5. Ql_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)
```

- **Parameter**

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

slaveAddr:
[In] Slave address.

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

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

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

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

- **Return Value**

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

5.7.6.3.6. Ql_IIC_Uninit

This function releases the IIC pins.

- **Prototype**

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

- **Parameter**

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

- **Return Value**

QL_RET_OK indicates 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 ,do not care the IicSpeed.
    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. The interface can also be simulated by GPIO pins, which can be any GPIO in the GPIO list in ***Chapter 5.7.2.2***.

5.7.7.2. SPI Usage

The following steps illustrate how to use the SPI function:

Step 1: Initialize SPI Interface. Call *Ql_SPI_Init* function to initialize the configurations for a SPI channel, including the specified pins for SPI, SPI type, and SPI channel number.
Step 2: Configure parameters. Call *Ql_SPI_Config* function to configure 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 a 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)
```

- **Parameter**

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

pinClk:
[In] SPI CLK pin.

pinMiso:
[In] SPI MISO pin.

pinMosi:
[In] SPI MOSI pin.

spiType:
[In] SPI type. It must be zero.

- **Return Value**

QL_RET_OK indicates 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)
```

- **Parameter**

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

isHost:
[In] Whether use host mode or not. It must be TRUE and just support host mode.

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

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

clkSpeed:
[In] SPI speed. It is not supported now, so the input argument will be ignored.

- **Return Value**

If no error occurs, the length of the write data will be returned. Negative integer indicates 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)
```

- **Parameter**

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

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

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

- **Return Value**

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

5.7.7.3.4. 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)
```

- **Parameter**

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

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

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

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

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

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 occurs, the length of the read data will be returned. Negative integer indicates this function fails.

5.7.7.3.5. Ql_SPI_Uninit

This function releases the SPI pins.

- **Prototype**

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

- **Parameter**

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

- **Return Value**

QL_RET_OK indicates 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_KBR0,PINNAME_KBR1,PINNAME_KBR2,0);
    Ql_Debug_Trace("\r\n<--SPI channel 1 Ql_SPI_Init ret=%d-->\r\n",ret);

    ret=Ql_SPI_Config(1,1,1,1,0);// isHost=1, cpol=1, cpha=1,
    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_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. Debug APIs

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.

5.8.1. Usage

There are two working modes for UART2 (DEBUG port): BASE MODE and ADVANCE MODE. Developers can configure 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 outputted as text through UART2 port. And the UART2 port works as common serial port with RX, TX and GND. In such case, UART2 can be used as common serial port for application.

Under advance mode, both application debug messages and system debug messages will be outputted

through UART2 port with special format. The “Genie Tool” provided by Quectel can be used to capture and analyze these messages. Usually developers do not need to use ADVANCE_MODE without the requirements from support engineers. If needed, please refer to *Quectel_Genie_Tool_User_Guide* for the usage of the special debug mode.

5.8.2. API Functions

5.8.2.1. Ql_Debug_Trace

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

- **Prototype**

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

- **Parameter**

format:
Pointer to a null-terminated multibyte string that specifies how to interpret the data. The maximum string length is 512 bytes. Format-control string. A format specification has the following form:

%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 outputted is 512.
3. To print a 64-bit integer, please first convert it to characters using *Ql_sprintf()*.

5.9. RIL APIs

OpenCPU RIL related API functions respectively implement the corresponding AT command's function. Developers can simply call APIs to send AT commands and get the response when APIs return. Developers can refer to 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*.

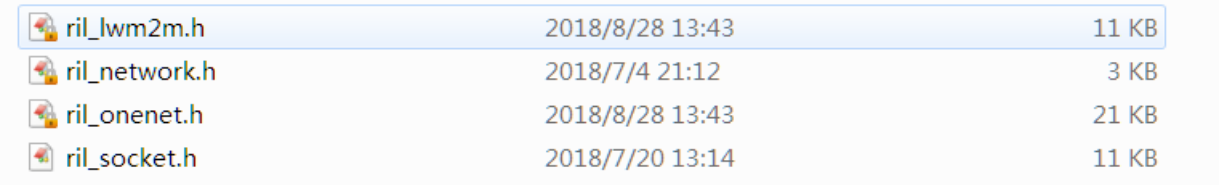
5.9.1. AT APIs

The API functions in this section are declared in header file *ril.h*.

5.9.1.1. 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.

Network/UDP/TCP/LwM2M/OneNET demos based on RIL API are available in *ril_xxx.h* files in *ril\inc* directory.



- **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 users' parameter.

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

- **Return Value**

RIL_AT_SUCCESS: indicates the AT command is executed successfully, and the response is OK.
RIL_AT_FAILED: indicates failed to execute the AT command or the response is ERROR.
RIL_AT_TIMEOUT: indicates sending AT command timed out.
RIL_AT_BUSY: indicates the AT command is being sent.
RIL_AT_INVALID_PARAM: indicates invalid input parameter.
RIL_AT_UNINITIALIZED: indicates RIL is not ready, and module needs to wait for MSG_ID_RIL_READY and then call *Ql_RIL_Initialize()* to initialize RIL.

- **Default Callback Function**

If this callback parameter is set to NULL, a default callback function will be called. But the default callback function only handles the simple AT response. Please refer to *Default_atRsp_callback* in *ril_atResponse.c*.

The following codes are the implementation for default callback function.

```
s32 Default_atRsp_callback(char* line, u32 len, void* userdata)
{
    if (Ql_RIL_FindLine(line, len, "OK")) //Find <CR><LF>OK<CR><LF>, <CR>OK<CR>，<LF>OK<LF>
    {
        return  RIL_ATRSP_SUCCESS;
    }
    else if (Ql_RIL_FindLine(line, len, "ERROR") //Find <CR><LF>ERROR<CR><LF>,
<CR>ERROR<CR>,<LF>ERROR<LF>
            || Ql_RIL_FindString(line, len, "+CME ERROR:") //Fail
            || Ql_RIL_FindString(line, len, "+CMS ERROR:")) //Fail
    {
        return  RIL_ATRSP_FAILED;
    }
    return RIL_ATRSP_CONTINUE;  //Continue to wait
}
```

6 Appendix A References

Table 8: Reference Documents

SN	Document Name
[1]	Quectel_BC26_AT_Commands_Manual
[2]	Quectel_BC26-OpenCPU_Hardware_Design
[3]	Quectel_OpenCPU_RIL_Application_Note
[4]	Quectel_QFlash_User_Guide
[5]	Quectel_Genie_Tool_User_Guide

Table 9: Abbreviations

Abbreviation	Description
ADC	Analog-to-digital Converter
API	Application Programming Interface
App	OpenCPU Application
EINT	External Interrupt Input
DFOTA	Delta Firmware Over the Air
GCC	GNU Compiler Collection
DCB	Data Center Bridging
GPIO	General Purpose Input Output
IIC	Inter-Integrated Circuit
I/O	Input/Output
KB	Kilobytes
M2M	Machine-to-Machine

MB	Megabytes
MCU	Micro Control Unit
PWM	Pulse Width Modulation
RAM	Random-Access Memory
RIL	Radio Interface Layer
RTC	Real Time Clock
SDK	Software Development Kit
SMS	Short Messaging Service
SPI	Serial Peripheral Interface
TCP	Transfer Control Protocol
UART	Universal Asynchronous Receiver and Transmitter
UDP	User Datagram Protocol
URC	Unsolicited Result Code
WTD	Watchdog