BC35-G&BC28 OpenCPU Series User Guide

NB-IoT Module Series

Rev. BC35-G&BC28-OpenCPU_Series_User_Guide_V1.0

Date: 2018-08-19

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/salesupport.aspx

For technical support, or to report documentation errors, please visit:
http://www.quectel.com/support/techsupport.aspx
Or Email to: Support@quectel.com

GENERAL NOTES

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

About the Document

History

Revision	Date	Author	Description
1.0	2018-08-19	Gary TANG/Fonda Fang	Initial

Contents

Table Index

Figure Index

1 Introduction

OpenCPU is an embedded development solution for M2M applications where NB-IoT 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 NB-IoT modules to run without external MCU. It can be widely used in M2M field, such as air monitor devices, smart meters, street light, wearable devices, etc.

2 OpenCPU Platform

2.1. System Architecture

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

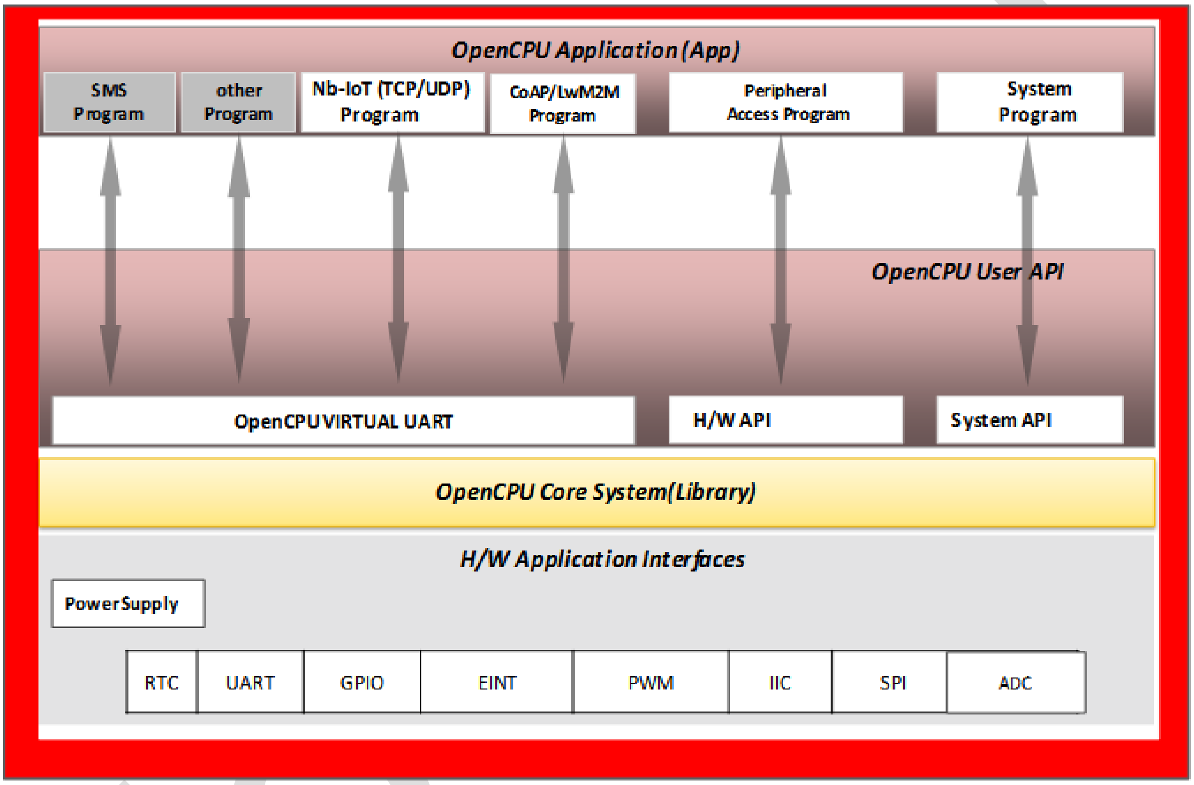

Figure 1: The Fundamental Principle of OpenCPU Software Architecture

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

OpenCPU Core System is a combination of hardware and software of NB-IoT module. It has built-in ARM Cortex-M0 processor, and has been built over LiteOS operating system, which has the characteristics of micro-kernel, real-time, multi-tasking, etc.
OpenCPU User API functions are designed for accessing to hardware resources, radio communications resources , or external devices. All API functions are introduced in ***Chapter 3***.

2.2. Open Resources

2.2.1. Processor

The module contains 3 processor cores: Application Processor, Protocol Processor and Security Processor.
Security Processor : ARM Cortex-M0 core @ 51.75 MHz
Protocol Processor : ARM Cortex-M0 core @ 51.75 MHz
Application Processor :ARM Cortex-M0 core @ 51.75 MHz

2.2.2. Memory Schemes

The module builds in 352KB flash and 64KB onchip SRAM. User application code and core code shared the space.

2.3. Interfaces

2.3.1. UART

OpenCPU provides 3 UART ports: MAIN UART, DEBUG UART and UART3. The MAIN UART and DEBUG UART are also named as UART1 and UART2 respectively.

OpenCPU also provides a VIRTUAL UART ports, which is used for application program to communicate with Core System through AT commands.

UART1, UART2 and UART3 is a 3-wire interface. UART2 has debug function that can debug the Core System.

2.3.2. GPIO

There are 19 I/O pins that can be configured for general purpose I/O. All pins can be accessed by OpenCPU by API functions.

2.3.3. EINT

OpenCPU supports external interrupt input. There are up to 19 I/O pins that can be configured for external interrupt input. But the EINT cannot be used for the purpose of highly frequent interrupt detection, which causes module’s unstable working. The EINT pins can be accessed by API functions.

2.3.4. ADC

There is an analogue input pin that can be configured for ADC. The sampling period and count can be configured by an API.

2.3.5. IIC

OpenCPU series module provides a hardware IIC interface. The IIC interface is multiplexed with GPIOs.

2.3.6. SPI

OpenCPU series module provides a hardware SPI interface. The SPI interface is multiplexed with GPIOs.

2.4. Development Environment

2.4.1. SDK

OpenCPU SDK provides the resources as follows for developers:

- Compile environment.
- Development guide and other related documents.
- A set of header files that defines all API functions and type declaration.
- Static library.
- Source code for examples.
- Download tool for firmware package file.
- Debug tool

Customers may get the latest SDK package from sales channel.

2.4.2. Editor

Any text editor is available for editing codes, such as Source Insight, Visual Studio and even Notepad.

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

2.4.3. Compiler & Compiling

2.4.3.1. Complier

OpenCPU uses **arm-none-eabi-gcc** as the compiler.

2.4.3.2. Compiling

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

```
Scons_new.bat
Scons_new.bat -c
```

2.4.3.3. Compiling Output

In command-line, some compiler processing information will be outputted during compiling.
Therefore, if there exists any compiling error during compiling, please check the error line number and the error hints.

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

```
scons: Building targets ...
Compiling src\custom\private\sys_config.c ...
scons: done building targets.
Neul firmware package updater v3.22.0.14
Adding .\build_scons\arm\application.bin...
Adding .\build_scons\arm\sha256\application.sha256...
Application finished.
```

2.4.4. Download

The document ***Quectel_BC35-G&BC28_Firmware_Upgrade_User_Guide_V1.0*** introduces the download tool and the way to use it to download firmware package.

2.4.5. How to Program

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

2.4.5.1. Program Composition

OpenCPU program consists of the aspects as follows.

Table 1: OpenCPU Program Composition

Item	Description
.h files	Declarations for variables, functions and macros.
.c files	Source code implementations.
.a file	Static library
SConscript files	Define the destination object files and directories to compile.
SConstruct,.py files	Compile script.
.fwpkg file	Standard firmware package file, which contains core image.
messages.xml	Contains a set of message definitions for resoluting debug log file.

2.4.5.2. Program Framework

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

```
/***************************************************************
* Main Task
***************************************************************/
void main_task( void *unused)
{
    UNUSED(unused);
    uint32 cnt = 0;
    ql_wait_for_at_init();    //wait for modem ok

    if(ql_uart_init(UART_PORT1) != QOCPU_RET_OK)
    {
        QDEBUG_NORMAL("uart port1 init error");
    }
    if( ql_uart_open(UART_PORT1, 9600, uart1_recieve_handle) != QOCPU_RET_OK )
    {
        QDEBUG_NORMAL("open uart1 error");
    }

    APP_DEBUG("\r\n<-- OpenCPU: Multitask Example -->\r\n");
    // Start message loop of this task
    while(1)
    {
        APP_DEBUG("\r\n<-- Main task has run %d times-->\r\n",cnt++);

        //Add you own code here
```

```
            osDelay(5000);
            osThreadYield();
        }
}
```

The *main_task* function is the entrance of Embedded Application for developers.

2.4.5.3. Sconscript

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

\src\SConscript needs to be maintained. This sciprt mainly includes:

Preprocessor definitions
Definitions for the paths that include files
Source code directories and files to compile
Library files to link

The macros of Quectel example are appended in the script. If you want to compile and run Quectel example, you can uncomment the macro of the example, one example one time. User macro alse can be appended to control application source code compile.

```
 ####################################################################################
 # You can add your own macros here
 # If you want to compile and run Quectel example, you can uncomment the macro of the example, one
example one time
 if "_QUECTEL_OPEN_CPU_" in env['CPPDEFINES']:
     #env.Append( CPPDEFINES=["__EXAMPLE_MULTITASK__"])          # multitask example
     #env.Append( CPPDEFINES=["__EXAMPLE_ADC__"])                # adc example
     #env.Append( CPPDEFINES=["__EXAMPLE_ATC_PIPE__"])           # atc pipe example
     #env.Append( CPPDEFINES=["__EXAMPLE_UART__"])               # uart example
     #env.Append( CPPDEFINES=["__EXAMPLE_EINT__"])               # eint example
     #env.Append( CPPDEFINES=["__EXAMPLE_SPI__"])                # spi example
     #env.Append( CPPDEFINES=["__EXAMPLE_I2C__"])                # i2c example
     #env.Append( CPPDEFINES=["__EXAMPLE_GPIO__"])               # gpio example
     env.Append( CPPDEFINES=["__EXAMPLE_KV__"])                  # kv example
     #env.Append( CPPDEFINES=["__EXAMPLE_AT_UDP__"])             # at udp example
     #env.Append( CPPDEFINES=["__EXAMPLE_AT_TCP__"])             # at tcp example
     #env.Append( CPPDEFINES=["__EXAMPLE_AT_IOT__"])             # at iot example
     #env.Append( CPPDEFINES=["__CUSTOMER__"])                   # for customer
 ####################################################################################
```

2.4.5.4. How to Add User Source Code Files

User can put new .c files in *\src\custom\private* directory and put .h files in *src\custom\public* directory, then the newly added .c files will be compiled automatically.

2.4.5.5. How to Add User Task

Suppose new main_task function implimented in customer.c file and declared in customer.h file.
Step 1: Added line to ql_task_cfg.h file to include customer.h.

```
#include " customer.h "
```

Step 2: Added task definition to custom_tasks[], include task name, task stack size, task priortiy and task main function pointer.

```
custom_task_definition_t    custom_tasks[] =
{
    {{"main_task", 0, NULL, 0, NULL, (500), CUSTOM_TASK_PRIORITY_0, 0, 0},
      (osThreadFunc_t)main_task, NULL, (custom_CreateQueueFunction_t)NULL},

    {{"END", 0, NULL, 0, NULL, 0, 0, 0, 0}, (osThreadFunc_t)NULL, NULL, NULL},
};
```

Step 3: Compile and run.

3 API Functions

3.1. System API Functions

The header file *cmsis_os2.h* declares system-related API functions. These functions are essential to any customers' applications. Make sure the header file is included when using these functions.

OpenCPU provides interfaces that support multitasking, message, mutex and semaphore mechanism functions. These interfaces are used for multitask programming.

3.1.1. Usage

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

3.1.1.1. Message Quene

Developers can call *osMessageQueueGet* to retrieve a message from a message queue.

Developers can call *osMessageQueuePut* to send messages to quene.

Step 1: Create a message quene. Developers can call *osMessageQueueNew* to create a message quene.

Step 2: Send message. Call osMessageQueuePut send a message to a queen.

Step 3: Get message. Call osMessageQueueGet to retrieve a message from a message queue.

3.1.1.2. 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 the code that accesses the memory is executed. After writing to the shared memory, the task releases the mutex object.

Step 1: Create a mutex. Developers can call *osMutexNew* to create a mutex.
Step 2: Get the specified mutex. If developers want to use mutex mechanism for programming, they can call osMutexAcquire to get the specified mutex.
Step 3: Give the specified mutex. Developers can call *osMutexRelease* to release the specified mutex.

3.1.1.3. 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 waiting for the semaphore object and is incremented each time a task releases the semaphore. When the count reaches zero, no more tasks can successfully wait for the semaphore object state to be signaled. The state of a semaphore is set 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 *osSemaphoreNew* to create a semaphore.
Step 2: Get the specified emaphore. If developers want to use semaphore mechanism for programming, they can call *osSemaphoreAcquire* to get the specified semaphore.
Step 3: Give the specified semaphore. Developers can call *osSemaphoreRelease* to release the specified semaphore.

3.1.1.4. Timer

The system provides maximum 12 software timers for application.

Step 1: Create a timer. Developers can call *osTimerNew* to create a timer.
Step 2: Start a timer. Call *osTimerStart* to start a timer.
Step 3: Stop a timer. Call *osTimerStop* to stop a running timer.

3.1.1.5. Backup Critical Data

OpenCPU has designed 10 blocks of system storage space to backup critical user data. The 10 blocks can be access by keys CUSTOM_KV_ID0~ CUSTOM_KV_ID9.

Developers may call *neul_kv_set* to backup data, call *neul_kv_get* to read back data from backup space, and call *neul_kv_erase_key* to erase back data from backup space

3.1.2. API Functions

3.1.2.1. osDelay

Wait for a specified time period in millisec.

- **Prototype**

```
osStatus_t osDelay (uint32_t ticks)
```

- **Parameters**

ticks :
[in]   time delay value. Unit: ms.

- **Return Value**

status code that indicates the execution status of the function.

3.1.2.2. osMessageQueueNew

Create and Initialize a Message Queue object.

- **Prototype**

```
osMessageQueueId_t osMessageQueueNew (    uint32_t        msg_count,
                                          uint32_t        msg_size,
                                          const osMessageQueueAttr_t *       attr
                                         )
```

- **Parameters**

msg_count:
[In] Maximum number of messages in queue.

msg_size :
[In] maximum message size in bytes.

attr :
[In] message queue attributes; NULL: default values.

- **Return Value**

Message queue ID for reference by other functions or NULL in case of error.

3.1.2.3. osMessageQueuePut

Put a Message into a Queue or timeout if Queue is full.

- **Prototype**

```
osStatus_t osMessageQueuePut (   osMessageQueueId_t        mq_id,
                                 const void *   msg_ptr,
                                 uint8_t    msg_prio,
                                 uint32_t        timeout
                                )
```

- **Parameters**

mq_id :
[In] Message queue ID obtained by osMessageQueueNew.

msg_ptr :
[In] Pointer to buffer with message to put into a queue .

msg_prio:
[In] Message priority.

timeout:
[In] CMSIS_RTOS_TimeOutValue or 0 in case of no time-out.

- **Return Value**

Status code that indicates the execution status of the function.

3.1.2.4. osMessageQueueGet

Get a Message from a Queue or timeout if Queue is empty.

- **Prototype**

```
osStatus_t osMessageQueueGet (  osMessageQueueId_t      mq_id,
                                void *      msg_ptr,
                                uint8_t *       msg_prio,
                                uint32_t        timeout
                               )
```

- **Parameters**

mq_id :
[In] Message queue ID obtained by osMessageQueueNew.

msg_ptr :
[Out] Pointer to buffer with message to get from a queue.

msg_prio:
[Out] Pointer to buffer for message priority or NULL.

timeout:

[In] CMSIS_RTOS_TimeOutValue or 0 in case of no time-out.

- **Return Value**

Status code that indicates the execution status of the function.

3.1.2.5. osMessageQueueDelete

Delete a Message Queue object.

- **Prototype**

```
osStatus_t osMessageQueueDelete ( osMessageQueueId_t     mq_id )
```

- **Parameters**

mq_id :
[In] Message queue ID obtained by osMessageQueueNew.

- **Return Value**

Status code that indicates the execution status of the function.

3.1.2.6. osMutexNew

Create and Initialize a Mutex object.

- **Prototype**

```
osMutexId_t osMutexNew  (const osMutexAttr_t *        attr  )
```

- **Parameters**

attr :
[In] Mutex attributes; NULL: default values.

- **Return Value**

Mutex ID for reference by other functions or NULL in case of error.

3.1.2.7. osMutexAcquire

Acquire a Mutex or timeout if it is locked.

- **Prototype**

```
osStatus_t osMutexAcquire (    osMutexId_t  mutex_id,
                               uint32_t     timeout
                          )
```

- **Parameters**

mutex_id :
[In] Mutex ID obtained by osMutexNew.

timeout :
[In] CMSIS_RTOS_TimeOutValue or 0 in case of no time-out..

- **Return Value**

Status code that indicates the execution status of the function.

3.1.2.8. osMutexRelease

Release a Mutex that was acquired by osMutexAcquire.

- **Prototype**

```
osStatus_t osMutexRelease (   osMutexId_t  mutex_id )
```

- **Parameters**

mutex_id :
[In] Mutex ID obtained by osMutexNew. .

- **Return Value**

Status code that indicates the execution status of the function.

3.1.2.9. osMutexDelete

Delete a Mutex object.

- **Prototype**

```
osStatus_t osMutexDelete ( osMutexId_t      mutex_id )
```

- **Parameters**

mutex_id :
[In] Mutex ID obtained by osMutexNew.

- **Return Value**

status code that indicates the execution status of the function.

3.1.2.10. osSemaphoreNew

Create and Initialize a Semaphore object.

- **Prototype**

```
osSemaphoreId_t osSemaphoreNew (    uint32_t        max_count,
                                    uint32_t        initial_count,
                                    const osSemaphoreAttr_t *       attr
                                )
```

- **Parameters**

max_count :
[In] Maximum number of available tokens .

initial_count:
[In] Initial number of available tokens.

attr :
[In] Semaphore attributes; NULL: default values.

- **Return Value**

Semaphore ID for reference by other functions or NULL in case of error .

3.1.2.11. osSemaphoreAcquire

Acquire a Semaphore token or timeout if no tokens are available.

- **Prototype**

```
osStatus_t osSemaphoreAcquire (    osSemaphoreId_t       semaphore_id,
                                   uint32_t       timeout
                               )
```

- **Parameters**

semaphore_id :
[In] Semaphore ID obtained by osSemaphoreNew.

timeout :
[In] CMSIS_RTOS_TimeOutValue or 0 in case of no time-out.

- **Return Value**

Status code that indicates the execution status of the function.

3.1.2.12. osSemaphoreRelease

Release a Semaphore token that was acquired by osSemaphoreAcquire.

- **Prototype**

```
osStatus_t osSemaphoreRelease (  osSemaphoreId_t       semaphore_id )
```

- **Parameters**

semaphore_id :
[In] Semaphore ID obtained by osSemaphoreNew.

- **Return Value**

Status code that indicates the execution status of the function.

3.1.2.13. osSemaphoreDelete

Delete a Semaphore object.

- **Prototype**

```
osStatus_t osSemaphoreDelete ( osSemaphoreId_t    semaphore_id )
```

- **Parameters**

semaphore_id :
[In] Semaphore ID obtained by osSemaphoreNew.

- **Return Value**

Status code that indicates the execution status of the function.

3.1.2.14. osTimerNew

Create and Initialize a timer.

- **Prototype**

```
osTimerId_t osTimerNew   (   osTimerFunc_t       func,
                             osTimerType_t       type,
                             void *       argument,
                             const osTimerAttr_t *     attr
                           )
```

- **Parameters**

func :
[In] Start address of a timer call back function .

type :
[In] osTimerOnce for one-shot or osTimerPeriodic for periodic behavior.

argument :
[In] Argument to the timer call back function.

attr :
[In] Timer attributes; NULL: default values.

- **Return Value**

Timer ID for reference by other functions or NULL in case of error.

3.1.2.15. osTimerStart

Start or restart a timer.

- **Prototype**

```
osStatus_t osTimerStart   (     osTimerId_t     timer_id,
                                uint32_t    ticks
                              )
```

- **Parameters**

timer_id :
[In] Timer ID obtained by osTimerNew.

ticks :
[In] Time ticks value of the timer.

- **Return Value**

Status code that indicates the execution status of the function.

3.1.2.16. osTimerStop

Stop a timer.

- **Prototype**

```
osStatus_t osTimerStop (    osTimerId_t  timer_id  )
```

- **Parameters**

timer_id :
[In] Timer ID obtained by osTimerNew.

- **Return Value**

Status code that indicates the execution status of the function.

3.1.2.17. osTimerDelete

Delete a timer.

- **Prototype**

```
osStatus_t osTimerDelete  (    osTimerId_t     timer_id  )
```

- **Parameters**

timer_id :
[In] Timer ID obtained by osTimerNew.

- **Return Value**

Status code that indicates the execution status of the function.

3.1.2.18. osTimerIsRunning

Check if a timer is running.

- **Prototype**

```
uint32_t osTimerIsRunning (   osTimerId_t    timer_id )
```

- **Parameters**

timer_id :
[In] Timer ID obtained by osTimerNew.

- **Return Value**

0 not running, 1 running.

3.1.2.19. neul_kv_set

Store a key value pair.

- **Prototype**

```
NEUL_RET neul_kv_set    (    neul_kv_key       key,
                             const uint8 *     kvalue,
                             uint16    kvalue_length
                        )
```

- **Parameters**

key :
[In] Key to associate key kvalue to.

kvalue :
[In] Value to store.

kvalue_length :
[In] Length in bytes of kvalue.

- **Return Value**

NEUL_RET_OK or an error code

3.1.2.20. neul_kv_get

Get a value associated with a specific key.

- **Prototype**

```
NEUL_RET neul_kv_get    (    neul_kv_key       key,
                             uint16    kvalue_max_length,
                             uint16 *  kvalue_length,
                             uint8 *   kvalue
                        )
```

- **Parameters**

key :
[In] Key of the value to get.

kvalue_max_length :
[In] Maximum length (in bytes) allowed to copy in the kvalue buffer if the key is found.

kvalue_length:
[Out] Length of kvalue in bytes.

kvalue :
[Out] Value associated with the provided key and group.

- **Return Value**

NEUL_RET_OK or an error code

3.1.2.21. neul_kv_erase_key

Erase an stored value given its key and group.

- **Prototype**

```
NEUL_RET neul_kv_erase_key ( neul_kv_key    key    )
```

- **Parameters**

key :
[In] Key of the key to erase.

- **Return Value**

NEUL_RET_OK or an error code.

3.1.2.22. irzalloc

Allocate a block of memory from internal RAM.

- **Prototype**

```
void* irzalloc ( size_t   size )
```

- **Parameters**

size :
[In] Size of the block to allocate.

- **Return Value**

Pointer to allocated memory, or NULL.

3.1.2.23. irfree

Free a block of memory.

- **Prototype**

```
void irfree (    void *     buf  )
```

- **Parameters**

buf :
[In] Pointer to allocated memory.

- **Return Value**

None.

3.2. Hardware Interface API Functions

3.2.1. UART

3.2.1.1. UART Overview

In OpenCPU, UART ports include physical UART ports and virtual UART port. The physical UART ports can be connected to external devices, and the virtual UART port is used to communicate between application and the core system.

The main UART port has low power function. When used this function in main UART, it's bandrate can not exceed 57600, otherwise it will be a normal uart. Other hardware uarts have three-wire interfaces.

OpenCPU provides one virtual UART port that are used for communication between App and Core. This virtual port is designed according to the features of physical serial interface.

The working chart for UARTs is shown as below:

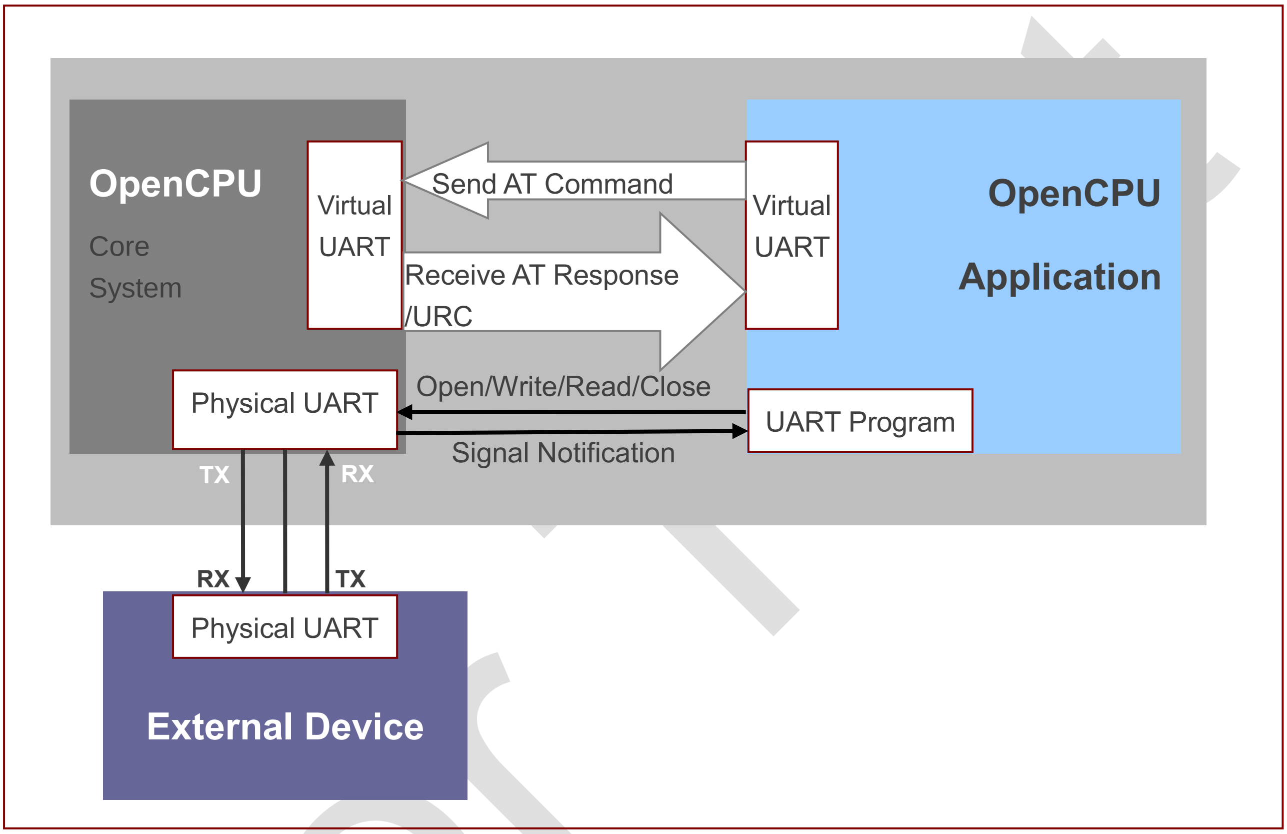

Figure 2: The Working Chart of UART

3.2.1.2. UART Usage

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

Step 1: Program the UART's callback function.
Step 2: Call ql_uart_init to initialize the the special UART port.
Step 3: Call ql_uart_open to open the special UART port.
Step 3: Call ql_uart_write to write data to the specified UART port. When the number of bytes actually sent is less than that to be sent, application should stop sending data.
Step 4: Deal with the UART's notification in the callback function. When data received, an interrupt occurs, then the callback function is called deal with the data in the UART RX buffer.

NOTES

The receive buffer size of physic UART is 2048 bytes. The receive buffer and send buffer size of virtual UART are 2763 bytes respectively.

3.2.1.3. API Functions

3.2.1.3.1. ql_wait_for_at_init

This function is wait for the syterm virtual UART initialized complete. If you want use the virtual UART port, before you initialize the virtual UART port in your application, you need call this function to wait for the system virtual UART initialized complete.

- **Prototype**

```
void  ql_wait_for_at_init(void)
```

- **Parameters**

None

- **Return Value**

None.

3.2.1.3.2. ql_uart_init

This function initializes the prammeters for the the specified serial port.

- **Prototype**

```
QOCPU_RET ql_uart_init ( uart_port    port)

The Enum of the uart port name:
typedef enum{
                    UART_PORT1,
                    UART_PORT3,
                    VIRTUAL_PORT,
               }uart_port
```

- **Parameters**

port :

[In] Port name.

- **Return Value**

QOCPU_RET_OK for executed success. Otherwise, the return value is an error code. To get extended error information, please refer to the ERROR CODES in header file *Qocpu_error.h*.

3.2.1.3.3. ql_uart_open

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

- **Prototype**

```
QOCPU_RET ql_uart_open (   uart_port port,
                           uint32 bandrate,
                           uart_recieve_data_callback uart_receive_call_back
                         )

The prototype of uart_recieve_call_back::
typedef void(*uart_recieve_data_callback)(uint8 *,uint32)
```

- **Parameters**

port:
[In] Port name.

baudrate:
[In] The baud rate of the UART to be opened.
The physical UART supports baud rates are as follows: 2400bps, 4800bps, 7200bps, 9600bps, 19200bps, 38400bps, 57600bps, 115200bps, 230400bps and 460800bps. The parameter does not take effect for VIRTUAL_PORT, so just set it to 0.

- **Return Value**

QOCPU_RET_OK for executed success. Otherwise, the return value is an error code. To get extended error information, please refer to the ERROR CODES in header file *Qocpu_error.h*.

3.2.1.3.4. ql_uart_write

This function is used to send data to the specified UART port. When the number of bytes actually sent is less than that to be sent, application should stop sending data. You must send the next data wait for the function return.

- **Prototype**

```
QOCPU_RET ql_uart_write (    uart_port port,
                              uint8 *buff,
                              uint32 buff_len
                             )
```

- **Parameters**

port :
[In] Port name.

buff :
[In] Pointer to data to write.

buff_len :
[In] The length of the data to write. For VIRTUAL_UART, the maximum length that can be written at one time is 2763 bytes which cannot be modified programmatically in application.

- **Return Value**

QOCPU_RET_OK for executed success. Otherwise, the return value is an error *code*. To get extended error information, please refer to the ERROR CODES in header file *Qocpu_error.h*.

3.2.1.3.5. ql_uart_close

This function closes the specified UART port.

- **Prototype**

```
void  ql_uart_close  (  uart_port  port)
```

- **Parameters**

port:
[In] Port name.

- **Return Value**

None.

3.2.1.4. Example

This chapter gives the example of how to use the UART port API functions, the details please refer to **example_atc_pipe.c** and **example_uart.c**.

```
//the uart1 receive data callback function
static void uart1_recieve_handle(uint8 *buffer,uint32 len)
```

```
{
        //process codes
}

//the virtual uart receive data callback function
static void vuart_recieve_handle(uint8 * buffer, uint32 len)
{
        //process codes
}

Void main (void)
{
       if(ql_uart_init(UART_PORT1) != QOCPU_RET_OK)
       {
              //you can add err handle
       }
       if(ql_uart_init(VIRTUAL_PORT)!=QOCPU_RET_OK)
       {
              //you can add err handle
       }

       if( ql_uart_open(UART_PORT1, 9600, uart1_recieve_handle) != QOCPU_RET_OK )
       {
              //you can add err handle
       }
       if(ql_uart_open(VIRTUAL_PORT, 9600, vuart_recieve_handle) != QOCPU_RET_OK )
       {
              //you can add err handle
       }
}
```

3.2.2. GPIO

3.2.2.1. GPIO Overview

There are 19 I/O pins that can be designed for general purpose I/O. All pins can be accessed under OpenCPU by API functions.

3.2.2.2. GPIO List

Table 2: BC35 Multiplexing Pins

Pin No.	Pin Name	RESET	MODE1	MODE2	MODE3	MODE4
1	PINNAME_SPI_CS	I/PN	SPI_CS	GPIO	EINT	
5	PINNAME_SPI_SO	I/PN	SPI_SO	GPIO	EINT	
6	PINNAME_SPI_CLK	I/PN	SPI_CLK	GPIO	EINT	
7	PINNAME_SPI_SI	I/PN	SPI_SI	GPIO	EINT	
8	PINNAME_UART_TX3	I/PN	UART_TX3	GPIO	EINT	
9	PINNAME_UART_RX3	I/PN	UART_RX3	GPIO	EINT	
10	PINNAME_SPI_CS2	I/PN	SPI_CS2	GPIO	EINT	
11	PINNAME_SPI_SO2	I/PN	SPI_SO2	GPIO	EINT	
12	PINNAME_SPI_CLK2	I/PN	SPI_CLK2	GPIO	EINT	
13	PINNAME_SPI_SI2	I/PN	SPI_SI2	GPIO	EINT	
17	PINNAME_PIO_20	I/PN	GPIO	EINT		
19	PINNAME_DBG_RXD	I/PN	DBG_RXD	GPIO	EINT	
27	PINNAME_DTR_GPIO	I/PN	DTR	GPIO	EINT	
31	PINNAME_CTS_AUX	I/PN	CTS	GPIO	EINT	
32	PINNAME_RTS_AUX	I/PN	RTS	GPIO	EINT	
35	PINNAME_I2C_SCL	I/PN	I2C_SCL	GPIO	EINT	
36	PINNAME_I2C_SDA	I/PN	I2C_SDA	GPIO	EINT	
37	PINNAME_SIM_DETECT	I/PN	GPIO	GPIO	EINT	

.

Table 3: BC28 Multiplexing Pins

Pin No.	Pin Name	RESET	MODE1	MODE2	MODE3	MODE4
3	PINNAME_SPI_SO	I/PN	SPI_SO	GPIO	EINT	

Pin No.	Pin Name	RESET	MODE1	MODE2	MODE3	MODE4
4	PINNAME_SPI_SI	I/PN	SPI_SI	GPIO	EINT	
5	PINNAME_SPI_CLK	I/PN	SPI_CLK	GPIO	EINT	
6	PINNAME_SPI_CS	I/PN	SPI_CS	GPIO	EINT	
19	PINNAME_DTR	I/PN	DTR	GPIO	EINT	
21	PINNAME_USIM_DETECT	I/PN	USIM_DETECT	GPIO	EINT	
22	PINNAME_CTS	I/PN	CTS	GPIO	EINT	
23	PINNAME_RTS	I/PN	RTS	GPIO	EINT	
28	PINNAME_UART_RXD3	I/PN	RXD3	GPIO	EINT	
29	PINNAME_UART_TXD3	I/PN	TXD3	GPIO	EINT	
30	PINNAME_GPIO1	I/PN	GPIO1	EINT		
31	PINNAME_GPIO2	I/PN	GPIO2	EINT		
32	PINNAME_GPIO3	I/PN	GPIO3	I2C_SDL	EINT	
33	PINNAME_GPIO4	I/PN	GPIO4	I2C_SDA	EINT	
38	PINNAME_DBUG_RXD	I/PN	DEBUG_RXD	GPIO	EINT	

.

- The “MODE1” defines the original status of pin in standard module.
- “RESET” column defines the default status of every pin after system is powered 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.
- “PN” means no internal pull

3.2.2.3. GPIO Initial Configuration

In OpenCPU, call GPIO related API functions to initialize GPIOs, please refer to ***Chapter 3.3.2.5***.

.

3.2.2.4. GPIO Usage

The following shows how to use the multifunctional GPIOs:

Step 1: GPIO initialization. *Call ql_gpio_init* function sets the specified pin as the GPIO function, and initializes the configurations, which includes direction, level and pull selection.

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

Step 3: Release the pin. If developers do not want use this pin no longer, and need to use this pin for other purposes (such as PWM, EINT), they must call *ql_gpio_uninit* to release the pin first. This step is optional.

3.2.2.5. API Functions

3.2.2.5.1. ql_gpio_init

This function enables the GPIO function of the specified pin, and initializes the configurations, which includes direction, level and pull selection.

- **Prototype**

```
QOCPU_RET ql_gpio_init   (      Enum_PinName pinName,
                                 Enum_PinDirection dir,
                                 Enum_PinLevel   level
                          )
```

- **Parameters**

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

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

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

- **Return Value**

QOCPU_RET_OK for success. Otherwise, the return value is an error code. To get extended error information, please refer to the ERROR CODES in header file *Qocpu_error.h*.

3.2.2.5.2. ql_gpio_get_level

This function gets the level of the specified GPIO.

- **Prototype**

```
bool ql_gpio_get_level ( Enum_PinName pinName)
```

- **Parameters**

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

- **Return Value**

Return the level of the specified GPIO. True means high level, and false means low level.

3.2.2.5.3. ql_gpio_set_level

This function sets the level of the specified GPIO.

- **Prototype**

```
QOCPU_RET ql_gpio_set_level(PinName pinName, PinLevel level)
```

- **Parameters**

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

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

- **Return Value**

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

3.2.2.5.4. ql_gpio_pull_config

This function sets the pull type of the specified GPIO.

- **Prototype**

```
Void    ql_gpio_pull_config ( Enum_PinName pinName, Enum_Gpio_Pull pull_type)
```

- **Parameters**

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

pull_type :
[In] Pull selection. One value of Enum_Gpio_Pull .

- **Return Value**

None.

3.2.2.5.5. ql_gpio_uninit

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

- **Prototype**

```
QOCPU_RET ql_gpio_uninit(PinName pinName)
```

- **Parameters**

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

- **Return Value**

QOCPU_RET_OK for executed success. Otherwise, the return value is an error code. To get extended error information, please refer to the ERROR CODES in header file *Qocpu_error.h*.

3.2.2.6. Example

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

```
if(ql_gpio_init(PINNAME_PIO_20, PINDIRECTION_IN, PINLEVEL_NONE ) != QOCPU_RET_OK)
{
    QDEBUG_ERROR("gpio init err ") ;
    return false;
}

ql_gpio_pull_config(PINNAME_PIO_20, PIN_PULL_DOWN);
```

3.2.3. EINT

3.2.3.1. EINT Overview

OpenCPU module has up to 19 external interrupt pins, please refer to ***Chapter*** 3.2.2.2. for details. The interrupt trigger mode support level-triggered mode and edge-triggered mode. The software debounce for external interrupt sources is used to minimize the possibility of false activations. External interrupt has higher priority, so frequent interrupt is not allowed.

NOTE

The interrupt has a higher priority, so you can not stay in the the callback function for long time.

3.2.3.2. EINT Usage

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

Step 1: *Call ql_gpio_init* function sets the specified pin as the intput pin.
Step 2: Call ql_gpio_pull_config to set internal pull state, if need.
Step 3: Call ql_gpio_isq_callback_register function to set the interrupt callback function and enable the interrupt.

3.2.3.3. API Functions

3.2.3.3.1. ql_gpio_isq_callback_register

This function registers an specifies the interrupt handler and enable the interrupt.

- **Prototype**

```
void ql_gpio_isq_callback_register (   Enum_PinName      pinName,
                                       Enum_Interrupt_Mode     type,
                                       Ql_GPIO_CALLBACK   qi_isq_callback
                                     )

typedef void (*Ql_GPIO_CALLBACK)(PIN pin);
```

- **Parameters**

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

type :
[in] EINT interrupt triger mode. One value of Enum_Interrupt_Mode.

qi_isq_callback :
[In] The interrupt handler.

- **Return Value**

None.

3.2.3.3.2. ql_gpio_isq_callback_unregister

This function release the EINT pin as a general GPIO_PIN. Disable the interrupt and clear the callback handle.

- **Prototype**

```
void ql_gpio_isq_callback_unregister   ( Enum_PinName   pinName)
```

- **Parameters**

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

- **Return Value**

None.

3.2.3.4. Example

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

```
void extern_isq_isr_call_back(PIN pin)//the gpio interrupt callback function need user to program
{
    //process code
}

if(ql_gpio_init(exten_isq_pin, PINDIRECTION_IN, PINLEVEL_NONE ) != QOCPU_RET_OK)
{
        QDEBUG_ERROR("eint interrupt pin clain err\r\n") ;
        return false;
}

ql_gpio_pull_config(exten_isq_pin, PIN_PULL_DOWN);
ql_gpio_isq_callback_Register(exten_isq_pin,PIN_INTERRUPT_ANY_EDGE,extern_isq_isr_call_back);
```

3.2.4. ADC

3.2.4.1. ADC Overview

The module provides a 10-bit ADC interface for application support. This ADC is available in active and standby modes of operation and is accessible via MUXBUS<1>.

The ADC is capable of performing single measurements as well as continuous sampling with a programmable sampling rate. This peripheral is shared with the Protocol Core. It should not be used for continuous sampling over extended periods of time as not making it available to the Protocol Core this will affect radio performance.

3.2.4.2. ADC Usage

The following steps tell how to use the ADC function:

Step 1: Call *aio_func_trim_adc* function to adjusts trim registers.
Step 2: Call *aio_func_calibrate_adc* function to calibrate the ADC and set the internal ADC calibra tion tables.
Step 3: Call *aio_func_read_aiopin* function to read analogue voltage.

3.2.4.3. API Functions

3.2.4.3.1. aio_func_trim_adc function

This function adjusts trim registers.

- **Prototype**

```
AIO_FUNC_RET aio_func_trim_adc(void)
```

- **Parameters**

None.

- **Return Value**

AIO_MANAGER_RET_OK for success, or error code.

3.2.4.3.2. aio_func_calibrate_adc

This function calibrates the ADC and set the internal ADC calibration tables.

- **Prototype**

```
AIO_FUNC_RET aio_func_calibrate_adc(void)
```

- **Parameters**

None.

- **Return Value**

AIO_FUNC_RET_OK for success, or error code.

3.2.4.3.3. aio_func_read_aiopin

This function reads the analogue voltage present on the AIO<0> or AIO<1> pins, displayed in mv.

- **Prototype**

```
AIO_FUNC_RET aio_func_read_aiopin(uint32 *voltage, uint8 aio_pin_number)
```

- **Parameters**

voltage :
[Out] voltage reading in mv.

aio_pin_number :
[in] AIO<n> pin number, 0 or 1.

- **Return Value**

AIO_FUNC_RET_OK for success, error for any read failure reason

3.2.4.4. Example

The following example demonstrates the use of ADC sampling.

NOTE: you can read relative code in example_adc.c

```
NEUL_RET read_adc(uint32* mV)
{
    if (aio_func_trim_adc() != AIO_FUNC_RET_OK)
    {
        return   NEUL_RET_ERROR;
    }

    if (aio_func_calibrate_adc() != AIO_FUNC_RET_OK)
    {
```

```
        return NEUL_RET_ERROR;
    }

    if (aio_func_read_aiopin(mV, AIO_RESOURCE_AIO_PIN_1) != AIO_FUNC_RET_OK)
    {
        return NEUL_RET_ERROR;
    }

    return NEUL_RET_OK;
}
```

3.2.5. IIC

3.2.5.1. IIC Overview

The module provides a hardware IIC interface. The IIC interface can be simulated by GPIO pins, which can be any two GPIOs in the GPIO list in ***Chapter*** 3.3.2.2**.** Therefore, one or more IIC interfaces are possible.

3.2.5.2. IIC Usage

The following steps tell how to work with IIC function:

Step 1: Call *i2c_init* function to initialize I2C interface.
Step 2: Call *i2c_claim* function to claim an I2C channel, including the specified GPIO pins for I2C and an I2C channel number.
Step 3: Call *i2c_activate* function to config parameters that the slave device needs.
Step 4: Call *i2c_master_send_data* function to write slave address and write data to the specified slave registers.
Step 5: Read data from slave device. Developer can call *i2c_master_send_data* function firstly to write slave address, secondly call *i2c_master_receive_data* function to read data from the specified slave registers.
Step 6: Call *i2c_deactivate* and *i2c_release* function to release the specified I2C channel. This step is optional.

3.2.5.3. API Functions

3.2.5.3.1. i2c_init

This function initializers the I2C Interface.

- **Prototype**

```
void  i2c_init(void)
```

- **Parameters**

None

Return Value
None.

3.2.5.3.2. i2c_deinit

This function de-initializers the I2C Interface.

- **Prototype**

```
void i2c_deinit(void)
```

- **Parameters**

None.

- **Return Value**

None.

3.2.5.3.3. i2c_claim

This function claims the specified I2C channel.

- **Prototype**

```
I2C_RET i2c_claim(I2C_BUS* bus, I2C_PIN i2c_pin)
```

- **Parameters**

bus :
[in] The specified I2C channel.one value of I2C_BUS

i2c_pin :
[in] The specified pins used by I2C channel. The value is pin_map[Enum_PinName];

- **Return Value**

I2C_RET_OK if claimed successfully, I2C_RET_ERROR otherwise.

3.5.5.3.4. i2c_release

This function releases the specified I2C channel.
.

- **Prototype**

```
I2C_RET i2c_release(I2C_BUS bus)
```

- **Parameters**

bus:
[in] The specified I2C channel.

- **Return Value**

I2C_RET_OK if released successfully, I2C_RET_ERROR otherwise.

3.2.5.3.5. i2c_activate

This function activates the specified I2C channel.

- **Prototype**

```
I2C_RET  i2c_activate(I2C_BUS  bus,  I2C_CONFIGURATION  config)
```

- **Parameters**

bus:
[in] The specified I2C channel.

config :
[in] I2C configuration parameter.

- **Return Value**

I2C_RET_OK if configured successfully, I2C_RET_ERROR otherwise

3.2.5.3.6. i2c_deactivate

This function de-activates the specified I2C channel.

- **Prototype**

```
I2C_RET  i2c_deactivate(I2C_BUS  bus)
```

- **Parameters**

bus:
[in] The specified I2C channel.

- **Return Value**

I2C_RET_OK if deactivated successfully, I2C_RET_ERROR otherwise

3.2.5.3.7. i2c_master_send_data

This function is used to send data by the I2C master in block mode.

- **Prototype**

```
I2C_RET i2c_master_send_data (    I2C_BUS i2c_bus,
                                  uint16 addr,
                                  const uint8 *pdata,
                                  uint8 num_of_bytes
                                )
```

- **Parameters**

i2c_bus:
[in] The specified I2C channel.

addr:
[in] 7 bits or 10 bits I2C slave address.

pdata:
[in] I2C transmit data buffer pointer.

num_of_bytes:
[in] I2C transmit data number bytes.

- **Return Value**

I2C_RET_OK if sent data successfully, I2C_RET_ERROR otherwise

NOTE

Every time, use the *i2c_master_send_data* function, developer must place all the contents want to send into a buffer, including the address of the I2C slave and the data you want to send.

3.2.5.3.8. i2c_master_receive_data

This function is used to receive data by the I2C master in block mode.

- **Prototype**

```
I2C_RET i2c_master_receive_data (   I2C_BUS i2c_bus,
                                    uint16  addr,
                                    uint8  *pdata,
                                    uint8  num_of_bytes
                                  )
```

- **Parameters**

i2c_bus:
[in] The specified I2C channel.

addr:
[in] 7bits or 10bits slave address.

pdata:
[Out] I2C receive data buffer pointer.

num_of_bytes:
[Out] I2C receive data number bytes.

- **Return Value**

I2C_RET_OK if read data successfully, I2C_RET_ERROR otherwise

3.2.5.4. Example

The following example code demonstrates the use of IIC interface.

```
static I2C_RET at24c02_i2c_init (void)
{
    I2C_PIN i2c_pin_name = { SCL_I2C_PINNAME, SDA_I2C_PINNAME};
    I2C_CONFIGURATION i2c_config = {
                                    I2C_BUS_MODE_MASTER,
                                    i2c_config_addr_type_7bit,
                                    I2C_Config_Half_Time_100kbit
                                  };

    I2C_datatype.i2c_bus = I2C_bus;
    I2C_datatype.i2c_pin = i2c_pin_name;
    I2C_datatype.i2c_conf = i2c_config;

    i2c_init();

    if (i2c_claim(&I2C_datatype.i2c_bus, I2C_datatype.i2c_pin) != I2C_RET_OK)
    {
```

```
            QDEBUG_NORMAL("<---I2C claim failed ! --->");
            return I2C_RET_ERROR;
    }
    else
    {
            QDEBUG_NORMAL("<---I2C claim succeed ! --->");
    }

    if (i2c_activate(I2C_datatype.i2c_bus, I2C_datatype.i2c_conf) != I2C_RET_OK)
    {
            QDEBUG_NORMAL("<---I2C activate failed ! --->");
            return I2C_RET_ERROR;
    }
    else
    {
            QDEBUG_NORMAL("<---I2C activate succeed ! --->");
    }

    return I2C_RET_OK;
}
```

3.2.6. SPI

3.2.6.1. SPI Overview

Module provides a hardware SPI interface can be specified by GPIO pins. The interface includes the master operation, programmable clock bit rate and pre-scaler, separate transmit and receive FIFO memory buffers programmable data frame size from 4 to 16 bits. In addition, SPI-specific features are supported with Full duplex, four-wire synchronous transfers, programmable clock polarity and phase.

3.2.6.2. SPI Usage

The following steps tell how to use the SPI interface:

Step 1: Call *spi_init* function to initialize SPI interface.
Step 2: Call *spi_claim* function to claim SPI channel including the specified pins for SPI and SPI BUS.
Step 3: Call *spi_activate* function to config some parameters for the SPI channel, including the clock polarity and clock phase.
Step 4: Call *spi_send_data* function to write data to the specified slave bus.
Step 5: Call *spi_recv_data* function to read data from the specified slave bus.
Step 6: Call *spi_send_data* and *spi_recv_data* function to read and write data at one time for SPI full-duplex communication.
Step 7: Call *spi_deactivate* and *spi_release* function to release the SPI channel. This step is optional.

NOTE

Realize the function of SPI peripherals to ensure the voltage field open. example: In Neul_io_bank.c
#define PMU_VDD_IO_BANK_L1_DEFAULT_LEVEL PMU_VDD_IO_LEVEL_3V0

3.2.6.3. API Functions

3.2.6.3.1. spi_init

This function initializers the SPI interface.

- **Prototype**

```
void spi_init(void)
```

- **Parameters**

None.

- **Return Value**

None.

3.2.6.3.2. spi_deint

This function de-initializers the SPI interface.

- **Prototype**

```
void spi_deinit (void)
```

- **Parameters**

None.

- **Return Value**

None.

3.2.6.3.3. spi_claim

This function claims the specified SPI channel.

- **Prototype**

```
SPI_RET spi_claim(SPI_BUS* bus, SPI_PIN spi_pin)
```

```
The enum of the SPI_BUS is defined as follows.
typedef enum
{
       SPI_BUS0,
       SPI_BUS1,
       SPI_BUS_MAX_NUM
}SPI_BUS;

The SPI_PIN structure is defined as follows.
typedef struct
{
       SPI_INTERFACE   interface;
       PIN             clk_pin; // these pin value is pin_map[pin_name]
       PIN             csb_pin;
       PIN             mosi_pin;
       PIN             miso_pin;
}SPI_PIN;
```

- **Parameters**

bus :

[in] The specified SPI channel.

spi_pin :

[in] The pins assign to the specified SPI channel.

- **Return Value**

SPI_RET_OK if claimed successfully, SPI_RET_ERROR otherwise.

3.2.6.3.4. spi_release

This function releases the specified SPI channel and associated pins.

- **Prototype**

```
SPI_RET spi_release(SPI_BUS bus)
```

- **Parameters**

bus :

[in] The SPI channel to release.

- **Return Value**

SPI_RET_OK if SPI channel is released successfully, SPI_RET_ERROR otherwise.

3.2.6.3.5. spi_activate

This function activates the specified SPI channel.

- **Prototype**

```
SPI_RET spi_activate(SPI_BUS bus, SPI_CONFIGURATION config)

The SPI_CONFIGURATION structure is defined as follows.
typedef struct
{
    uint8               data_size;
    uint8               clk_div;
    SPI_CLK_MODE        clk_mode;
}SPI_CONFIGURATION;
```

- **Parameters**

bus :
[in] The SPI channel to activate.

config :
[in] The SPI configuration parameters.

- **Return Value**

SPI_RET_OK if SPI channel is activated successfully, SPI_RET_ERROR otherwise.

3.2.6.3.6. spi_deactivate

This function deactivates the specified SPI channel.

- **Prototype**

```
SPI_RET spi_deactivate(SPI_BUS bus)
```

- **Parameters**

bus:
[in] The SPI channel to deactivate.

- **Return Value**

SPI_RET_OK if SPI channel is deactivated successfully, SPI_RET_ERROR otherwise.

3.2.6.3.7. spi_send_data

This function is used to send data by the SPI master in block mode.

- **Prototype**

```
SPI_RET spi_send_data (SPI_BUS bus,
                       uint8* cmd_buff,
                       uint16 cmd_len,
                       uint8* data_buff,
                       uint16 data_len,
                       SPI_CALLBACK callback)

typedef void(* SPI_CALLBACK)(SPI_RET)
```

- **Parameters**

bus :
[in] The specified SPI channel.

cmd_buff :
[in] Command buffer address.

cmd_len :
[in] Command length

data_buff :
[in] Data buffer address.

data_len :
[in] Data length.

callback:
[in] Callback function

- **Return Value**

SPI_RET_OK if sent data successfully, SPI_RET_ERROR otherwise.

3.2.6.3.8. spi_recv_data

This function can receive data by the SPI master in block mode.

- **Prototype**

```
SPI_RET spi_recv_data   (   SPI_BUS bus,
                            uint8* cmd_buff,
                            uint16 cmd_len,
                            uint8* data_buff,
                            uint16 data_len,
                            SPI_CALLBACK callback,
                            bool ignore_rx_while_tx
                        )
```

- **Parameters**

bus :
[in] The specified SPI channel..

cmd_buff :
[in] Command buffer address.

cmd_len :
[in] Command length.

data_buff :
[Out] Data buffer address.

data_len :
[Out] Data length.

callback :
[in] Callback function.

ignore_rx_while_tx :
[in] The flag indicates that if ignore received data while transmit.

- **Return Value**

SPI_RET_OK if read data successfully, SPI_RET_ERROR otherwise.

3.2.6.4. Example

The following example shows the use of the SPI interface.

```
static void spi_pin_config(void)
{
    exspi.bus = SPI_BUS0;

    //to config spi
    exspi.pin.clk_pin = spi_pin_clk;
    exspi.pin.csb_pin = spi_pin_cs;
```

```
    exspi.pin.miso_pin = spi_pin_sdo;
    exspi.pin.mosi_pin = spi_pin_sdi ;
    exspi.pin.interface = SPI_INTERFACE_SINGLE_UNIDIR;

    exspi.spi_config.data_size = spi_data_size;
    exspi.spi_config.clk_div = spi_clock_div;
    exspi.spi_config.clk_mode = spi_clock_mode;
}

static bool w25q_spi_init(void)
{
    if (non_os_is_driver_initialised(DRIVER_INIT_SPI) == true)
    {
        QDEBUG_ERROR("spi driver has init\r\n") ;

        //If we are inited then we are safe to use, so return true
        return true;
    }
    if((spi_claim(&(exspi.bus), exspi.pin))!=SPI_RET_OK)
    {
        QDEBUG_ERROR("spi clain err\r\n") ;

        return   false;
    }
    if(spi_activate(exspi.bus,exspi.spi_config)!=SPI_RET_OK)
    {
        QDEBUG_ERROR("spi active err\r\n") ;

        return   false;
     }
     non_os_set_driver_initalised(DRIVER_INIT_SPI, true);

     return true;
}
```

4 Appendix A References

Table 3: Reference Documents

SN	Document Name
[1]	Quectel_BC35-G&BC28_AT_Commands_Manual
[2]	Quectel_BC35-G_硬件设计手册_V1.0
[3]	Quectel_BC28_硬件设计手册_V1.0

Table 4: Abbreviations

Abbreviation	Description
ACK	Acknowledgement
ADC	Analog-to-digital Converter
API	Application Programming Interface
App	OpenCPU Application
Core	Core System; OpenCPU Operating System
CSD	Circuit Switched Data
DNS	Domain Name System
EINT	External Interrupt Input
FOTA	Firmware Over the Air
GCC	GNU Compiler Collection
GPIO	General Purpose Input Output
IIC	Inter-Integrated Circuit
IMSI	International Mobile Subscriber Identification Number
I/O	Input/Output
KB	Kilobytes
M2M	Machine-to-Machine

MB	Megabytes
MCU	Micro Control Unit
PWM	Pulse Width Modulation
RAM	Random-Access Memory
ROM	Read-Only Memory
RTC	Real Time Clock
SDK	Software Development Kit
SMS	Short Messaging Service
SPI	Serial Peripheral Interface
SPP	Sequential Packet Protocol
SSP	Secure Simple Pairing
TCP	Transfer Control Protocol
UART	Universal Asynchronous Receiver and Transmitter
UDP	User Datagram Protocol
UID	User Identification
URC	Unsolicited Result Code
(U)SIM	(Universal) Subscriber Identity Module
WTD	Watchdog