BC66&BC66-NA QuecOpen Quick Start Guide

NB-IoT Module Series

Rev. BC66&BC66-NA_QuecOpen_Quick_Start_Guide_V1.1

Date: 2020-06-03

Status: Released

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

Quectel Wireless Solutions Co., Ltd.
Building 5, Shanghai Business Park Phase III (Area B), No.1016 Tianlin Road, Minhang District, Shanghai 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

Revision History

Version	Date	Author	Description
1.0	2019-06-27	Allan LIANG	Initial
1.1	2020-06-03	Allan LIANG	1. Updated “OpenCPU” into “QuecOpen”. 2. Completed and optimized the description of how to compile SDK in Chapter 4. 3. Updated some descriptions in Chapter 8 and Chapter 10.

Contents

Table Index

Figure Index

1 Introduction

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

This document introduces how to use the QuecOpen® SDK package to quickly start the development of BC66 and BC66-NA modules in QuecOpen® solution. Additionally, this document introduces matters needing attention for the application design and programming.

2 QuecOpen® Related Documents

The following documents are necessary during QuecOpen application design and programming.

Table 1: QuecOpen® Related Documents

Document Name	Description
Quectel_BC66&BC66-NA_QuecOpen_User_Guide	Describes the detailed APIs relating to the development of BC66 QuecOpen and BC66-NA QuecOpen.
Quectel_BC66&BC66-NA_QuecOpen_RIL_Application_Note	Describes how to program AT commands with RIL APIs, and how to get the response of AT commands when the API returns.
Quectel_BC66&BC66-NA_QuecOpen_DFOTA_Application_Note	Describes how to use DFOTA feature in BC66 QuecOpen and BC66-NA QuecOpen.
Quectel_BC66_QuecOpen_Hardware_Design	Describes how to design hardware in BC66 QuecOpen.
Quectel_BC66-NA_QuecOpen_Hardware_Design	Describes how to design hardware in BC66-NA QuecOpen.
Quectel_BC66&BC66-NA_QuecOpen_Genie_Log_Capture_Guide	Describes how to capture Genie log in BC66 QuecOpen and BC66-NA QuecOpen.

3 Preparations

Before QuecOpen application design and programming, it is necessary to confirm whether the following software and hardware components listed in this chapter have been prepared well.

3.1. Host System

The host operating system must be one of the following ones:

- Windows XP
- Windows Vista
- Windows 7 32-bit or 64-bit
- Windows 10 32-bit or 64-bit

3.2. Hardware

- Quectel BC66 QuecOpen module or BC66-NA QuecOpen module
- Quectel BC66-TE-B/BC66-NA-TE-B
- Accessories such as USB cable, etc.

Please contact Quectel Technical Supports (support@quectel.com) to get the required hardware if needed.

3.3. QuecOpen SDK

- QuecOpen SDK package

Please contact Quectel Technical Supports for the QuecOpen SDK package.

- App download tool

The QFlash tool is available in *tools* folder in the SDK package.

4 Compilation

This chapter introduces how to compile SDK in command line.

4.1. Development Environment

4.1.1. SDK

Please contact Quectel Technical Supports to obtain the latest SDK package. BC66 QuecOpen/BC66-NA QuecOpen SDK package provides the following resources for developers:

- Compiling environment.
- Development guide and other related documents.
- A set of header files that defines all API functions and type declaration.
- Source code examples.
- Open source code for RIL.
- App download tool.

4.1.2. Editor

Text editors are available for editing codes, such as Source Insight, Visual Studio or 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++ programs, and provides syntax highlighting, code navigation and customizable keyboard shortcuts.

4.1.3. Programming Language

C programming language.

4.2. Compiling and Downloading

4.2.1. Compiler

GCC can be used to compile the source codes in QuecOpen solution, and the recommended compiler

version is *arm-none-eabi-gcc v4.8.3*. The GCC compiler is provided under *tools* directory by default, and no additional installation is needed. Directly open batch processing window in MS-DOS to compile the codes.

4.2.2. Compiling Commands

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

```
make clean    //Clear the existing App
make new      //Compile a new App
```

Before compiling a new App, clear the existing App first so as to avoid incomplete file compilation.

4.2.3. Compiling Output

In a command line, some compilation information will be output during compiling. All WARNINGs and ERRORs are saved in *build\gcc\build.log*. Therefore, if there exists any compilation error during compiling, check *build.log* for the error line number and the error hints.

For example, in line 195 of *example_at.c*, the semicolon is intentionally omitted, as shown below:

```
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 compilation error will be shown in build.log as follows:

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

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

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

4.2.4. Download

QFlash tool provided in the SDK package is typically used to download the application bin files. Refer to ***document [1]*** for more details about the tool and its usage.

4.3. How to Program

By default, the source code files are stored in *custom* directory. A new folder can be created in the root directory to manage a custom project.

4.3.1. Program Composition

The composition of a QuecOpen program is described as follows.

Table 2: QuecOpen Program Composition

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

4.3.2. Program Framework

The following codes are the least codes that comprise a QuecOpen 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:
            {
                APP_DEBUG("<-- 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 sending AT commands.
                Demo_SendATCmd();
                break;
            }
            case MSG_ID_URC_INDICATION:
            {
                switch (msg.param1)
                {
                    case URC_SYS_INIT_STATE_IND:
                        APP_DEBUG ("<-- Sys Init Status %d -->\r\n", msg.param2);
                    break;

                    case URC_SIM_CARD_STATE_IND:
                        APP_DEBUG ("<-- SIM Card Status:%d -->\r\n", msg.param2);
                    break;

                    case URC_EGPRS_NW_STATE_IND:
                        APP_DEBUG ("<-- EGPRS Network Status:%d -->\r\n", msg.param2);
                    break;

                    default:
                        APP_DEBUG ("<-- Other URC: type=%d\r\n", msg.param1);
                    break;
                }
                break;
            }
            default:
            break;
        }
    }
}
```

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

Ql_OS_GetMessage is an important function that handles the messages received from the task. Be noted

that this function may be a blocking API if there is no incoming message.

MSG_ID_RIL_READY indicates the RIL task is ready. After receiving this message, call *Ql_RIL_Initialize* immediately to prepare for calling the RIL API.

MSG_ID_URC_INDICATION is a system message to indicate that a new URC is coming.

4.3.3. Makefile

In QuecOpen solution, the default project file is loaded to *makefile* and is ready for use. In custom projects, the added files should be loaded to *makefile*. Refer to ***Chapters 4.3.4*** and ***4.3.5*** for details about how to add files.

Before compiling programs according to native conditions, it is necessary to change some settings of *makefile*, such as the compiler environment path. The content of *makefile* mainly includes:

- Path of GCC compiler
- Preprocessor definitions
- Definitions of hearder file paths
- Source code path and files to be compiled
- Library files to be linked

4.3.4. How to Add a .c File

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

4.3.5. How to Add a Directory

Generally, a project not only needs to add .c or .h files, but also needs to build different folders according to project functions. After building a folder for a specific project function, the function folder needs to be added to *makefile*. Follow the steps below to add a new folder (take the folder *tracker* as an example):

1. Add a new directory for .c and .h files of *tracker* in *make\gcc\gcc_makefile*.

```
#-------------------------------------------------------------------------------
# Configure the include directories
#-------------------------------------------------------------------------------
INCS =  -I $(ENV_INC)
INCS += -I ./           \
        -I include      \
        -I ril/inc      \
        -I custom/config   \
        -I tracker/inc


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

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

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

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

5 Download

5.1. Download through TE-B

If Quectel TE-B kit is used, download the firmware or App bin file via the main UART (UART0) on the TE-B.

5.2. Download through Customer Devices

If the module has been soldered to the motherboard of a customer device, download the firmware or App bin file through the main UART port (RXD, TXD).

5.3. Download in Mass Production

In order to improve the production efficiency, Quectel provides multi-port download tools to download firmware to several modules synchronously. Consult Quectel Technical Supports for the tool if needed.

6 Debug

During the QuecOpen application development process, the main debugging method is to trace the log printed out through UART.

BC66 QuecOpen and BC66-NA QuecOpen modules provide three UART ports:

- Main UART (UART0)
- Debug UART (UART2)
- Auxiliary UART (UART1)

When the module encounters problems such as abnormal reboot, dump and network issues, users can capture Genie log through debug UART, auxiliary UART or debug USB port. For more details, please refer to ***document [2]***.

7 QuecOpen® SDK Introduction

After unzipping the SDK package, take *BC66_QuecOpen_NB1_SDK_V1.5* as an example, the typical directory hierarchy will be shown as below:

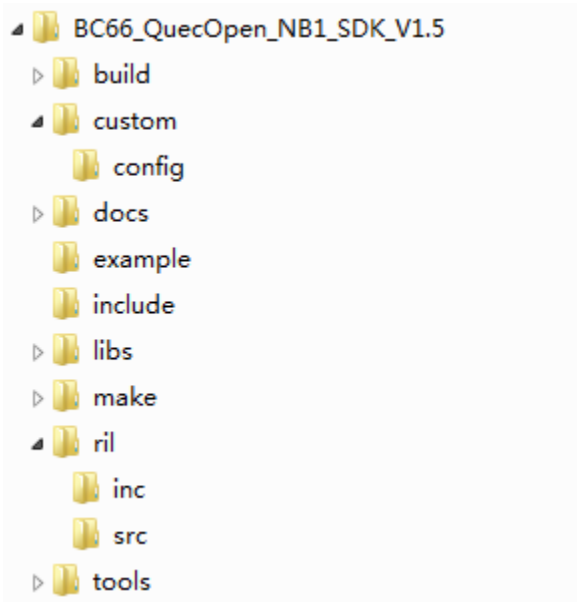

Figure 1: Directory Hierarchy of BC66_QuecOpen_NB1_SDK

Table 3: BC66_QuecOpen_NB1_SDK Directory Description

Directory Name	Description
BC66_QuecOpen_NB1_SDK_V1.5	Root directory of QuecOpen SDK package.
build	Compilation information and intermediate files storage path.
custom	Custom development files directory, including some essential configuration files.
docs	QuecOpen development files storage path.
example	Examples for supported functions; can be compiled independently.
include	Common header files.

libs	Link library storage path, including linker script.
make	Project makefile storage path.
ril	QuecOpen RIL source code storage path. Users can use RIL APIs to program AT commands.
tools	Tools storage path, including compiler and downloading tools.

8 Create Custom Projects

By default, the directory *custom* is designed as the root directory for custom projects. In this directory, a program file *main.c* is available to demonstrate the usage of QuecOpen UART ports and AT commands.

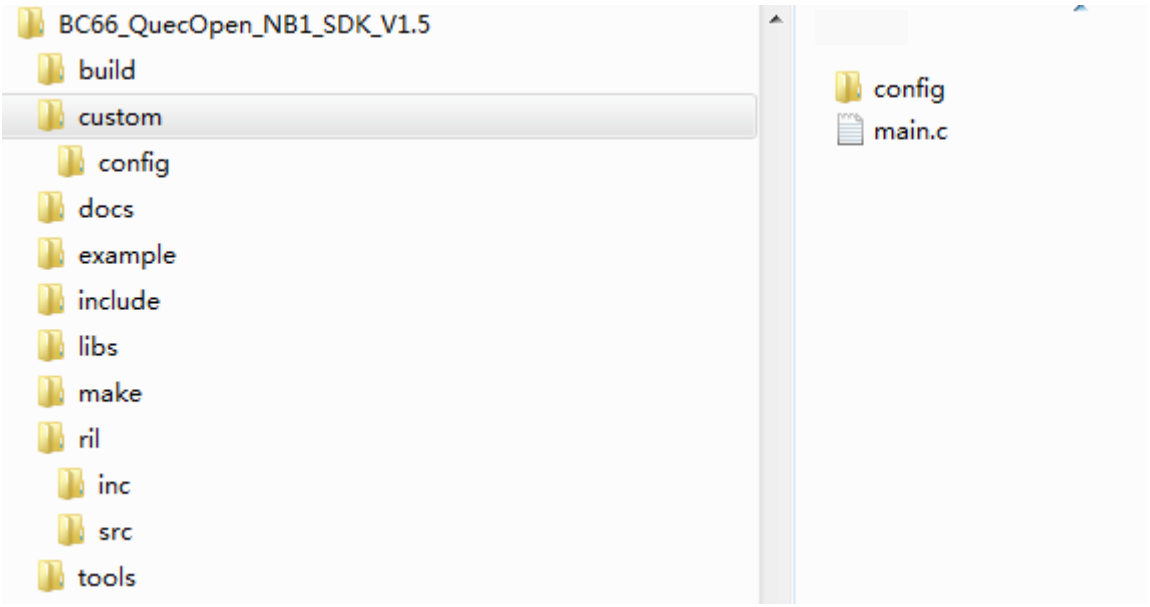

Figure 2: Custom Directory

In the *custom* directory, new custom files or sub-directories can be added. For specific programming guides, please refer to ***Chapter 4.3***.

All source code files are managed by *makefile* which is located in *make\gcc\gcc_makefile*. Users can decide which source code files in the project directory will be compiled. For specific programming guides, please refer to ***Chapter 4.3***.

There is a default project implementation that *main.c* in the SDK is added to *makefile*. The default codes in *main.c* demonstrates the usage of QuecOpen UART ports and AT commands, and users can also debug functions through modifying the codes.

9 Quick Programming

This chapter demonstrates how to control the pin level of a GPIO to drive the LED light, that is, to turn on/off the LED light as programmed. Through this, users will get a quick understanding on the basic software framework of QuecOpen SDK.

Please use the sample codes in this chapter to overwrite *custom\main.c*, or delete *main.c* and then create a new .c file to control the pin level of GPIO, thus realizing the on/off control of the LED light.

9.1. GPIO Control

9.1.1. Confirm Header Files to be Included

To confirm the header files (.h files) to be included, requirements of the application have to be confirmed first. For instance, in the demonstration application, the requirement is to realize the on/off control of the LED light by changing the level of the GPIO periodically.

- Firstly, to control a GPIO, the definitions of GPIO related API prototypes and variables, which are included in *ql_gpio.h*, are needed.
- Secondly, “changing the pin level of the GPIO periodically” means a timer is needed. The related definitions are included in *ql_timer.h*.
- Finally, the messages of the timer and UART ports need to be processed in the application, so *ql_system.h* is necessary. In addition, it is necessary to print some log messages to debug the program, and the relevant header files are *ql_stdlib.h*, *ql_uart.h* and *ql_trace.h*. All return values of API functions are defined in *ql_error.h*.

Therefore, the header files to be included are:

```
#include "ql_stdlib.h"
#include "ql_trace.h"
#include "ql_error.h"
#include "ql_system.h"
#include "ql_uart.h"
#include "ql_gpio.h"
#include "ql_timer.h"
```

9.1.2. Control GPIO

The NETLIGHT pin is already connected to an LED on the BC66-TE-B/BC66-NA-TE-B. Thus the on/off control of the LED can be achieved by controlling NETLIGHT directly.

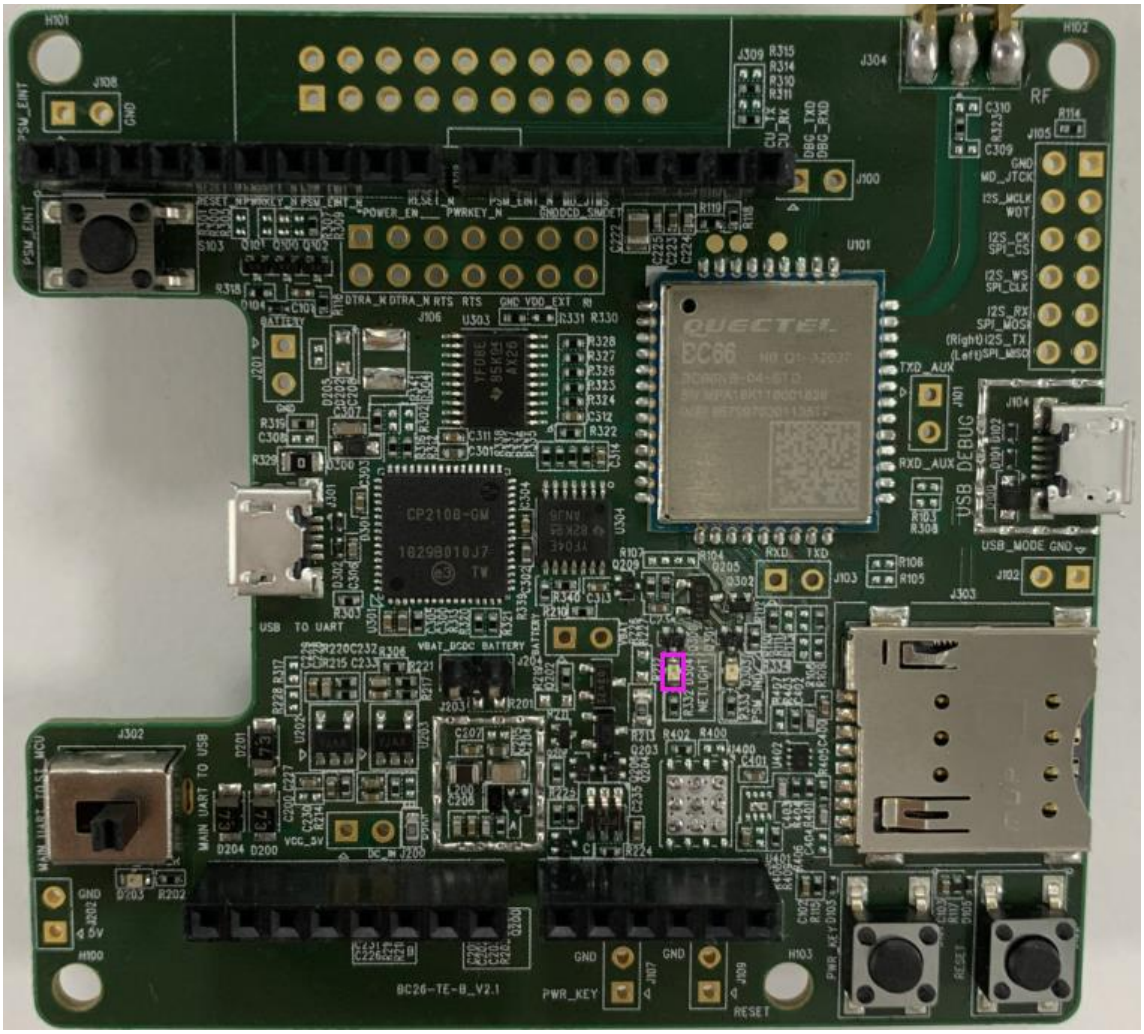

Figure 3: NETLIGHT on TE-B

Step 1: Configure GPIO pin definition in the program.

```
//Define GPIO pin
static Enum_PinName m_gpioPin = PINNAME_NETLIGHT;
```

Step 2: Initialize GPIO as follows:

- Initialize “input/output” status to "PINDIRECTION_OUT".
- Initialize "initial level" status to "PINLEVEL_LOW".
- Initialize “pull up and down” status to "PINPULLSEL_PULLUP".

```
//Initialize GPIO
ret = Ql_GPIO_Init(m_gpioPin, PINDIRECTION_OUT, PINLEVEL_LOW, PINPULLSEL_PULLUP);
if (QL_RET_OK == ret)
{
    APP_DEBUG ("<-- Initialize GPIO successfully -->\r\n");
}else{
    APP_DEBUG ("<-- Fail to initialize GPIO pin, cause=%d -->\r\n", ret);
}
```

Afterwards, there is a need to start a timer and periodically change the level of the GPIO to realize LED blinking. The steps are illustrated in the subsequent chapter.

9.1.3. Define the Timer

In this demo, the program defines a timer with the timeout of 500 ms. It means the LED will be on for 500 ms and off for 500 ms.

Step 1: Define the timer ID, timer inverval (timeout value) and corresponding timer interrupt handler.

```
//Define a timer and the handler
static u32 m_myTimerId = 2019;
static u32 m_nInterval = 500;     // 500 ms
static void Callback_OnTimer(u32 timerId, void* param);
```

Step 2: Register and start the defined timer.

```
//Register and start the timer
Ql_Timer_Register(m_myTimerId, Callback_OnTimer, NULL);
Ql_Timer_Start(m_myTimerId, m_nInterval, TRUE);
```

Step 3: Implement interrupt handler of the defined timer.

```
static void Callback_OnTimer(u32 timerId, void* param)
{
    s32 gpioLvl = Ql_GPIO_GetLevel(m_gpioPin);
    if (PINLEVEL_LOW == gpioLvl)
    {
        // Set GPIO to high level, then LED is light
        Ql_GPIO_SetLevel(m_gpioPin, PINLEVEL_HIGH);
        APP_DEBUG ("<-- Set GPIO to high level -->\r\n");
    }else{
        // Set GPIO to low level, then LED is dark
        Ql_GPIO_SetLevel(m_gpioPin, PINLEVEL_LOW);
        APP_DEBUG ("<-- Set GPIO to low level -->\r\n");
```

```
    }
}
```

9.1.4. Complete Sample Code

So far, all programming is completed, and the complete sample code is provided as follow:

```
#include "ql_stdlib.h"
#include "ql_trace.h"
#include "ql_error.h"
#include "ql_system.h"
#include "ql_gpio.h"
#include "ql_timer.h"
#include "ql_uart.h"

//Define APP DEBUG
    #define DEBUG_ENABLE 1
    #if DEBUG_ENABLE > 0
    #define DEBUG_PORT    UART_PORT0
    #define DBG_BUF_LEN     512
    static char DBG_BUFFER[DBG_BUF_LEN];
    #define APP_DEBUG(FORMAT,...) {\
        Ql_memset(DBG_BUFFER, 0, DBG_BUF_LEN);\
        Ql_sprintf(DBG_BUFFER,FORMAT,##__VA_ARGS__); \
        if (UART_PORT2 == (DEBUG_PORT)) \
        {\
            Ql_Debug_Trace(DBG_BUFFER);\
        } else {\
            Ql_UART_Write((Enum_SerialPort)(DEBUG_PORT),(u8*)(DBG_BUFFER),
Ql_strlen((const char *)(DBG_BUFFER)));\
        }\
    }
    #else
    #define APP_DEBUG(FORMAT,...)
    #endif

    static Enum_SerialPort m_myUartPort    = UART_PORT0;

//Define GPIO pin
static Enum_PinName m_gpioPin = PINNAME_NETLIGHT;

//Define a timer and the handler
static u32 m_myTimerId = 2019;
static u32 m_nInterval = 500;     // 500ms
```

```
static void Callback_OnTimer(u32 timerId, void* param);
static void CallBack_UART_Hdlr(Enum_SerialPort port, Enum_UARTEventType msg, bool level, void*
customizedPara);
/************************************************************************/
/* The entrance procedure for this example application                  */
/************************************************************************/
void proc_main_task(s32 taskId)
{
    s32 ret;
    ST_MSG msg;

        //Register & open UART port
        ret = Ql_UART_Register(m_myUartPort, CallBack_UART_Hdlr, NULL);
        if (ret < QL_RET_OK)
        {
            Ql_Debug_Trace("Fail to register serial port[%d], ret=%d\r\n", m_myUartPort, ret);
        }
        ret = Ql_UART_Open(m_myUartPort, 115200, FC_NONE);
        if (ret < QL_RET_OK)
        {
            Ql_Debug_Trace("Fail to open serial port[%d], ret=%d\r\n", m_myUartPort, ret);
    }
    APP_DEBUG("QuecOpen: LED Blinking by NETLIGH\r\n");

    //Initialize GPIO
    ret = Ql_GPIO_Init(m_gpioPin, PINDIRECTION_OUT, PINLEVEL_LOW, PINPULLSEL_PULLUP);
    if (QL_RET_OK == ret)
    {
        APP_DEBUG ("<-- Initialize GPIO successfully -->\r\n");
    }else{
        APP_DEBUG ("<-- Fail to initialize GPIO pin, cause=%d -->\r\n", ret);
    }

    //Register and start the timer
    Ql_Timer_Register(m_myTimerId, Callback_OnTimer, NULL);
    Ql_Timer_Start(m_myTimerId, m_nInterval, TRUE);

    //Start message loop of this task
    while(TRUE)
    {
        Ql_OS_GetMessage(&msg);
        switch(msg.message)
        {
        default:
```

```
                break;
            }
        }
}

static void Callback_OnTimer(u32 timerId, void* param)
{
    s32 gpioLvl = Ql_GPIO_GetLevel(m_gpioPin);
    if (PINLEVEL_LOW == gpioLvl)
    {
        //Set GPIO to high level, then LED is ON
        Ql_GPIO_SetLevel(m_gpioPin, PINLEVEL_HIGH);
        APP_DEBUG ("<-- Set GPIO to high level -->\r\n");
    }else{
        //Set GPIO to low level, then LED is OFF
        Ql_GPIO_SetLevel(m_gpioPin, PINLEVEL_LOW);
        APP_DEBUG ("<-- Set GPIO to low level -->\r\n");
    }
}

    static void CallBack_UART_Hdlr(Enum_SerialPort port, Enum_UARTEventType msg, bool level, void* customizedPara)
    {
}
```

9.1.5. Run App Bin

The complete code can be copied to *custom\main.c* to overwrite the existing code, and then compile and download the App bin to the module to run the App.

When the application is running, users will see that D304 LED on the TE-B is blinking periodically and the main UART outputs the following debug information:

```
[2019-01-23_13:14:08:624]QuecOpen: LED Blinking by NETLIGH
[2019-01-23_13:14:08:624]<-- Initialize GPIO successfully -->
[2019-01-23_13:14:09:120]<-- Set GPIO to high level -->
[2019-01-23_13:14:09:620]<-- Set GPIO to low level -->
[2019-01-23_13:14:10:120]<-- Set GPIO to high level -->
[2019-01-23_13:14:10:620]<-- Set GPIO to low level -->
[2019-01-23_13:14:11:120]<-- Set GPIO to high level -->
[2019-01-23_13:14:11:621]<-- Set GPIO to low level -->
[2019-01-23_13:14:12:120]<-- Set GPIO to high level -->
[2019-01-23_13:14:12:621]<-- Set GPIO to low level -->
[2019-01-23_13:14:13:121]<-- Set GPIO to high level -->
```

10 Matters Needing Attention

10.1. Light Sleep Mode

When the module is in Light Sleep mode, before sending any data, it is necessary to send a packet of data with two bytes (such as **AT**) through the main UART (UART0) to wake up the module first.

10.2. Deep Sleep Mode

When the module is in Deep Sleep mode, the CPU will be powered off and only the RTC is running. When the module is woken up from Deep Sleep mode, programs will be reloaded and the code will be resumed.

To avoid loss of important data after the module enters Deep Sleep mode, API functions (*Ql_Flash_Write* and *Ql_Flash_Read*) can be called to save and read data within 4 KB.

10.3. UART

BC66 QuecOpen or BC66-NA QuecOpen module provides three UART ports with the default baud rate of 115200 bps. The data buffer size of each UART port is 1400 bytes, so do not send any data exceeding 1400 bytes to the module at a time. If the data exceeds 1400 bytes, divide it into several sub-packets with each within 1400 bytes, and then send the sub-packets one by one.

When the event of *EVENT_UART_READY_TO_READ* in the UART callback is received, the application should call *Ql_UART_Read* to read all the received data from the UART buffer.

10.4. Timer

In QuecOpen solution, there are two kinds of timers: common timer and fast timer. QuecOpen solution supports 90 common timers and 6 fast timers in total, and additionally supports a microsecond fast timer.

The common timer may be delayed due to task blocking, therefore a common timer is a task timer. While the fast timer does not belong to any task and is triggered by interrupts, so the fast timer is more real-time. Be noted to prevent interrupt overload, otherwise the system may be abnormal, otherwise the system may be abnormal.

10.5. Dynamic Memory

*Ql_MEM_Alloc c*an be called to specify the size of the dynamic memory and *Ql_MEM_Free* can be called to release the memory. The maximum size of the dynamic memory for an application is 300 KB.

11 Appendix A References

Table 4: Related Documents

SN	Document Name	Remark
[1]	Quectel_QFlash_User_Guide	Describes how to use QFlash
[2]	Quectel_BC66&BC66-NA_QuecOpen_Genie_Log_Capture_Guide	Describes how to capture Genie log in BC66 QuecOpen and BC66-NA QuecOpen
[3]	Quectel_BC66&BC66-NA_QuecOpen_User_Guide	Describes the detail API interfaces related to QuecOpen development

Table 5: Terms and Abbreviations

Abbreviation	Description
API	Application Program Interface
App	Application
DOS	Disk Operating System
GCC	GNU Compiler Collection
GPIO	General Purpose Input/Output
LED	Light Emitting Diode
RTC	Real Time Clock
SDK	Software Development Kit
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus