BC66&BC66-NA QuecOpen RIL Application Note

NB-IoT Module Series

Rev. BC66&BC66-NA_QuecOpen_RIL_Application_Note_V1.0

Date: 2020-06-19

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

COPYRIGHT

About the Document

Revision History

Version	Date	Author	Description
1.0	2020-06-19	Allan LIANG/ Sheffer GU	Initial

Contents

Table Index

1 Introduction

In QuecOpen® solution, Quectel BC66 and BC66-NA modules support RIL mechanism. QuecOpen® RIL is an open-source layer through which you can call APIs easily to send AT commands and get the response of AT commands from the return value of corresponding APIs. Also, you can add new APIs to send new AT commands.

This document introduces the RIL mechanism of BC66 and BC66-NA modules in QuecOpen® solution, as well as how to implement QuecOpen® RIL function.

2 QuecOpen® RIL Overview

QuecOpen RIL is an API function module serving to send AT commands from App to the core. The figure below illustrates the architecture of QuecOpen RIL.

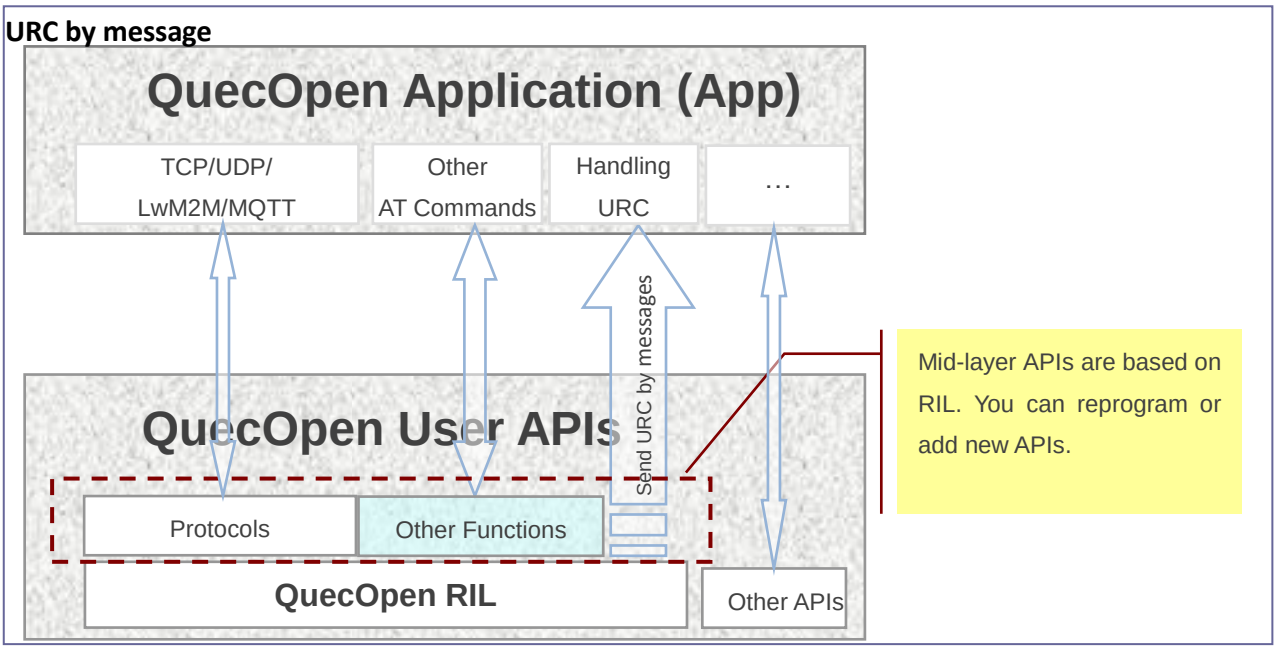

Figure 1: Architecture of QuecOpen® RIL

Based on QuecOpen RIL, you can simply call a RIL API to send an AT command to the core and get the AT command response through the return value of the API. Through RIL APIs, the App realizes the AT command sending function, which greatly simplifies application development.

In QuecOpen RIL, the URC is reported to App (to the main task by default) by a system message MSG_ID_URC_INDICATION. They can also be reported to a customized task. See ***Chapter 4.2.1*** for the structure definition of URCs.

3 QuecOpen® RIL APIs

QuecOpen RIL mainly provides two APIs and two system messages for the upper-layer application.

Table 1: QuecOpen® RIL Service Types

RIL Service Type	RIL Service	Description
System message	MSG_ID_RIL_READY	This message is sent to the main task indicating that the main task can prepare to call *Ql_RIL_Initialize* to initialize RIL when the RIL task is ready during booting.
System message	MSG_ID_URC_INDICATION	This message is sent to the main task when a URC needs to be reported to the App. In URC messages, *param1* indicates the type of URC, and *parame2* indicates the data carried in the specified URC. See ***Chapter 4.2.1*** for the structure definition of URCs.
API	*Ql_RIL_Initialize*	After the main task receives the MSG_ID_RIL_READY message, the App has to call this function to initialize RIL, that is, to execute some initial parameters and the initial AT commands. See ***Table 2*** for the default Initial AT commands defined by the array *g_InitCmds* in SDK.
API	*Ql_RIL_SendATCmd*	This function sends an AT command and calls a callback function to handle the response of the AT command.

3.1. Ql_RIL_Initialize

This function initializes RIL. After the main task receives the MSG_ID_RIL_READY message, the App has to call this function to initialize RIL, that is, to execute some initial parameters and the initial AT commands defined by *g_InitCmds*.

- **Prototype**

```
extern void Ql_RIL_Initialize(void)
```

- **Parameter**

None

- **Return Value**

None

NOTES

1. The App must call this function first before sending any AT command through QuecOpen RIL.
2. This function is subject to a function in the library, so “extern” must be added to its declaration.

3.2. Ql_RIL_SendATCmd

This function realizes the communication between the App and the core, allowing the App to send AT commands to the core. Before the function completes, it calls the callback function *atRsp_callBack* to handle the response of the AT command. You can store the parsing results of AT command responses in a space that the parameter *userData* points to.

- **Prototype**

```
extern s32 Ql_RIL_SendATCmd(char* atCmd, u32 atCmdLen, Callback_ATResponse atRsp_callBack, void* userData, u32 timeOut)
```

- **Parameter**

atCmd:
[In] AT command string to be sent.

atCmdLen:
[In] The length of AT command string.

atRsp_callBack:
[In] Callback function for handling the response of AT command. This parameter can be set to NULL so as to call the default callback function, when the AT command responses are unconcerned. See ***Chapter 3.3*** for details of the default callback function.

userData:
[Out] Used to transfer data to *atRsp_callBack* or get data from *atRsp_callBack*.

timeOut:
[In] Timeout for the AT command. Default timeout value: 180000. Unit: millisecond.

- **Return Value**

0 Succeed in executing AT command, and the AT command response is **OK**.
-1 Failed in executing AT command, or the AT command response is **ERROR**.
-2 Indicates that the sending of AT command times out.
-3 Indicates that the AT command is sending.
-4 Indicates invalid input parameter(s).
-5 Indicates RIL is not initialized. The App must call *Ql_RIL_Initialize* to initialize RIL after the main task receives the MSG_ID_RIL_READY message.

3.2.1. Callback_ATResponse

- **Prototype**

```
typedef s32 (*Callback_ATResponse)(char* line, u32 len, void* userdata);
```

- **Parameter**

line:
[In] Return value of the AT command.

len:
[In] The length of the response of AT command.

userData:
[In] User data.

- **Return Value**

-1 Failed in executing the AT command.
0 Succeed in executing the AT command.
1 Indicates you have to continue to wait for the AT command response.

3.3. Default_atRsp_callback

If the callback function *atRsp_callBack* in *Ql_RIL_SendATCmd* is set to NULL, then the default callback

function *Default_atRsp_callback* will be called. This default callback function only handles simple responses of AT commands.

The following codes are the implementation of the 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, "+CMEE ERROR:")    //Fail
            || Ql_RIL_FindString(line, len, "+CME ERROR:")     //Fail
            || Ql_RIL_FindString(line, len, "+CMS ERROR:"))    //Fail
            || Ql_RIL_FindString(line, len, "+CIS ERROR:"))    //Fail
    {
        return  RIL_ATRSP_FAILED;
    }
    return RIL_ATRSP_CONTINUE;    //Continue to wait
}
```

NOTE

You can customize the callback function so as to realize function parsing for complex AT commands. The AT commands already available in QuecOpen RIL can also be modified according to specific application demands.

4 QuecOpen® RIL Implementation

4.1. RIL Initialization

When the programs start to run, the RIL task automatically sends the MSG_ID_RIL_READY message to the main task. After the main task receives the message, the App must call *Ql_RIL_Initialize* to initialize RIL. After RIL is initialized successfully, the App can call *Ql_RIL_SendATCmd* to send an AT command to the core.

The following codes show how to initialize RIL in App.

```
void proc_main_task(s32 taskId)
{
    s32 ret;
    ST_MSG msg;

    //Start message loop of this task
    while(TRUE)
    {
        Ql_OS_GetMessage(&msg);
        switch(msg.message)
        {
#ifdef __OCPU_RIL_SUPPORT__
        case MSG_ID_RIL_READY:
            APP_DEBUG("<-- RIL is ready -->\r\n");
            ret = Ql_RIL_Initialize();
            break;
#endif
        default:
            break;
        }
    }
}
```

The array *g_InitCmds* in SDK defines all initial AT commands available in QuecOpen RIL. You can delete or update the existing AT commands and/or add new AT commands to meet specific application

demands. The default initial AT commands are listed as below.

Table 2: List of Default Initial AT Commands

Initial AT Commands	Description
AT+CMEE=1	Enables the use of final result code **+CME ERROR: <err>** as an indication of an error relating to the functionality of the MT. When enabled, MT related errors cause **+CME ERROR: <err>** final result code instead of the regular **ERROR** final result code. **ERROR** is returned normally when an error is related to syntax, invalid parameters or TA functionality.
AT+CEREG=1	Enables network registration unsolicited result code **+CEREG: <stat>**
AT+QCFG=?	Queries supported system settings

4.2. URC Programming

4.2.1. URC Structure Definition

In QuecOpen RIL mechanism, all URCs are reported to the App through the system message MSG_ID_URC_INDICATION. In URC messages, *param1* indicates the type of URC (see ***Appendix A*** for common URC types), and *parame2* indicates the data carried in the specified URC. *param2* can be either the data or a pointer.

The structure of URC is defined as follows:

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

Table 3: URC Structure Parameters

Parameter	Parameter Type	Description
message	u32	System message
param1	u32	URC type
param2	u32	Data carried in the URC

srcTaskId	u32	ID of the task that receives the message

4.2.2. URC Types

In QuecOpen RIL, URCs are classified into system URCs and AT URCs.

- System URCs, such as IP address reporting URC and network status indication URC, indicate the various status of the module.
- AT URCs serve some specific AT commands such as **AT+QIOPEN** (see the example below).

```
AT+QIOPEN          //Send the AT command.
OK                 //This the AT command response and the executing of AT command has completed.

+QIOPEN:xxx        //The actual result of the AT command is reported asynchronously to App by the URC.
```

4.3. RIL API Creating

QuecOpen SDK includes some common RIL APIs, including LwM2M and MQTT APIs. You can add new APIs based on the current RIL APIs.

For instance, if you want to add a new API to acquire the IMSI of an USIM card, follow the steps below:

Step 1: Define a new API *RIL_SIM_GetIMSI*. You can create a new file *ril_sim.c* in the *ril\src* directory of SDK to store this function.

- **Prototype**

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

- **Parameter**

imsi:
[Out] IMSI of an USIM card.

- **Return Value**

0	Succeed in getting the IMSI.
Other values	Failed in getting the IMSI.

Step 2: Implement *RIL_SIM_GetIMSI*. See the complete code as below.

```
//
//Implementation for RIL_SIM_GetIMSI
//
s32 RIL_SIM_GetIMSI(char* imsi)
{
    char strAT[] = "AT+CIMI\0";
    if (NULL == imsi)
    {
        return RIL_AT_INVALID_PARAM;
    }
    return Ql_RIL_SendATCmd(strAT, Ql_strlen(strAT), ATRsp_IMSI_Handler,(void*)imsi, 0);
}
```

Step 3: Implement callback function *ATRsp_IMSI_Handler*.

```
//
//Implementation for ATRsp_IMSI_Handler
//Note: All AT responses string will be passed into the callback line by line
//
static s32 ATRsp_IMSI_Handler(char* line, u32 len, void* param)
{
    char* p1 = NULL;
    char* p2 = NULL;
    char* strImsi = (char*)param;

    p1 =   Ql_RIL_FindString(line, len, "OK");
    if (p1)
    {
        return   RIL_ATRSP_SUCCESS;
    }
    p1 = Ql_RIL_FindLine(line, len, "+CME ERROR:");
    if (p1)
    {
        return   RIL_ATRSP_FAILED;
    }
    p1 = Ql_RIL_FindString(line, len, "\r\n");
    if (p1)
    {
        p2 = Ql_strstr(p1 + 2, "\r\n");
        if (p2)
        {
            Ql_memcpy(strImsi, p1 + 2, p2 - p1 -2);
            return RIL_ATRSP_CONTINUE;
        }else{
```

```
                return RIL_ATRSP_FAILED;
            }
        }
        return RIL_ATRSP_CONTINUE;      //Wait for the next line of response
        }
```

Implement the callback function based on the responses of AT commands. Refer to *Quectel_BC66&BC66-NA_AT_Commands_Manual* for the detailed information of AT commands and the corresponding responses.

Step 4: Call the new API. As shown below, the App now can call the new API to acquire the IMSI of the USIM card.

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

s32 ret;
char strImsi[20];
ret = RIL_SIM_GetIMSI(strImsi);
if (RIL_AT_SUCCESS == ret)
{
    APP_DEBUG("The IMSI is: %s.\r\n", strImsi);
}else{
    APP_DEBUG ("Fail to get IMSI!\r\n");
}
```

5 Appendix A Common URC Types

Table 4: List of Common URC Types

URC Type	Definition
URC_SYS_INIT_STATE_IND	Module initial state indication
URC_SIM_CARD_STATE_IND	USIM card state indication
URC_EGPRS_NW_STATE_IND	Network registration state indication
URC_CFUN_STATE_IND	Module functionality level indication
URC_END	Undefined URC

NOTES

1. See *RIL.h* in QuecOpen SDK for detailed description of the URCs.
2. The App does not handle URC_END URC by default. When necessary, add a corresponding API in *ril_urc.c* of SDK.

6 Appendix B References

Table 5: Related Document

SN	Document Name	Remarks
[1]	Quectel_BC66&BC66-NA_AT_Commands_Manual	BC66&BC66-NA AT Commands Manual

Table 6: Terms and Abbreviations

Abbreviation	Description
API	Application Program Interface
App	Application
IMSI	International Mobile Subscriber Identity
IP	Internet Protocol
LwM2M	Lightweight Machine to Machine
MQTT	Message Queuing Telemetry Transport
MT	Mobile Terminal
NB-IoT	Narrowband Internet of Things
RIL	Radio Interface Layer
SDK	Software Development Kit
URC	Unsolicited Result Code
USIM	Universal Subscriber Identity Module