OpenCPU RIL

Application Note

Rev. OpenCPU_RIL_Application_Note_V1.2

Date: 2017-07-21

About the Document

History

Revision	Date	Author	Description
1.0	2017-07-01	Chunmao Li	Initial
1.1	2017-07-10	Chunmao Li	1. Inform App of URC message with system message, instead of callback function. 2. Delete Ql_RIL_RegisterURC function.
1.2	2017-07-21	Chunmao Li	Updated the supported module type.

Contents

Table Index

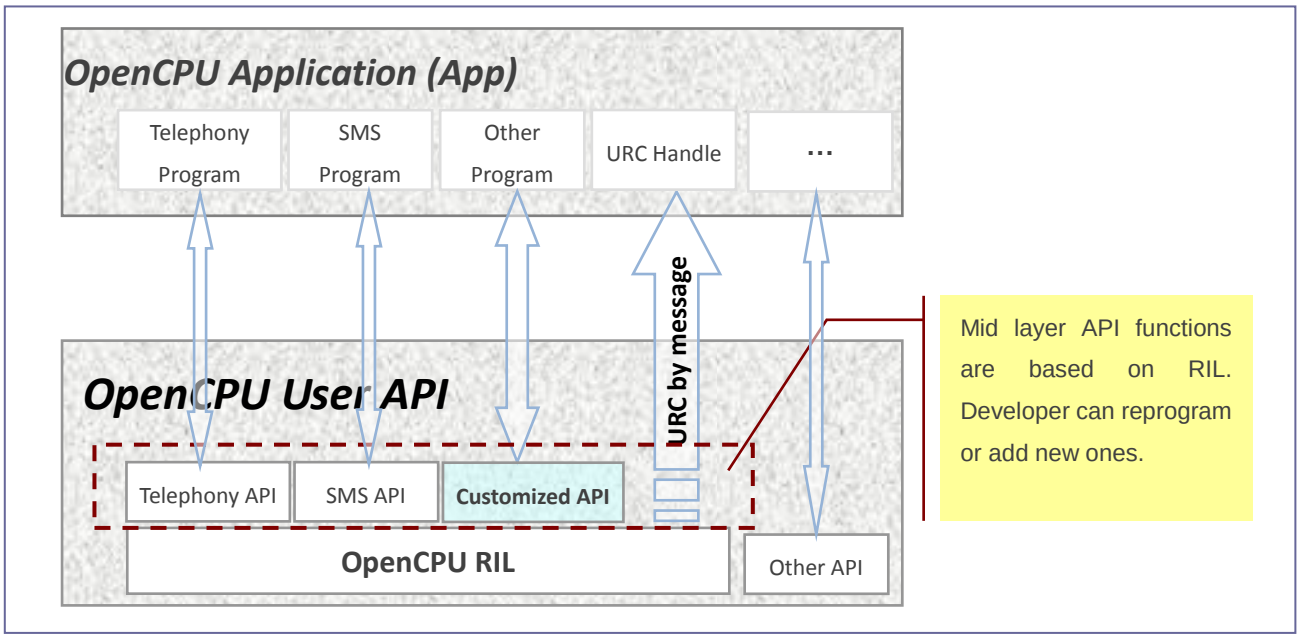
1 Introduction

This document introduces the OpenCPU RIL (Radio Interface Layer) mechanism and how to use it. Through the introduction of this document, developer can realize the value of convenience that OpenCPU RIL brings, especially when they program with AT commands.

This document is applicable to all ZF GSM, WCDMA and LTE modules.

2 About OpenCPU RIL

OpenCPU RIL is a user API function module. It is an open source layer, and serves AT processing. The following diagram shows the RIL framework.



With the OpenCPU RIL, developer can simply call API to send AT commands and get the response when API returns. Based on this, programmer can easily develop the mid layer API functions that serve the upper application.

In OpenCPU module, all URC (Unsolicited Result Code) messages are reported to App by a system message “MSG_ID_URC_INDICATION”. Please see chapter 5 for the definitions of all URCs.

About URC, please see chapter 4.2.

3 OpenCPU RIL Interfaces

OpenCPU RIL mainly provides two API functions and two system messages for the upper application.

RIL Service	Description
Message: MSG_ID_RIL_READY	This message is sent to the main task when RIL is ready during booting.
Message: MSG_ID_URC_INDICATION	This message is sent to the main task when an URC generates. In this message, the parameter1 carries the URC type, and the parameter2 carries the specified URC-related information, which needs to be interpreted from URC type to URC type.
API: Ql_RIL_Initialize	When receives the message, the main task needs to call this function to initialize RIL-related things. The init commands defined in *g_InitCmds* will be executed at this time.
API: Ql_RIL_SendATCmd	This function implements sending AT command with the result being returned synchronously.

3.1. Ql_RIL_Initialize

This function initializes RIL-related functions. When the main task receives the message *MSG_ID_RIL_READY*, App needs to call this function to initialize RIL-related things, including executing some initial AT commands that are defined in the variable *g_InitCmds*.

- **Prototype**

```
void Ql_RIL_Initialize(void);
```

- **Parameter**

None.

- **Return value**

None

3.2. Ql_RIL_SendATCmd

This function is used to send AT command with the result being returned synchronously. Before this function returns, the responses for AT command will be handled in the callback function *atRsp_callback*, and the paring results of AT responses can be stored in the space that the parameter *userData* points to. All AT responses string will be passed into the callback line by line. So the callback function may be called for times.

This function is the critical API that is the interface between OpenCPU RIL and the upper application. The function is defined as below.

- **Prototype**

```
s32 Ql_RIL_SendATCmd(char*    atCmd,
                     u32 atCmdLen,
                     Callback_ATResponse atRsp_callback,
                     void* userData,
                     u32 timeout
                    );
typedef s32 (*Callback_ATResponse)(char* line, u32 len, void* userdata);
```

- **Parameter**

atCmd:

[in]AT command string.

atCmdLen:

[in]The length of AT command string.

atRsp_callBack:

[in]Callback function for handling the response of AT command.

userData:

[out]Used to transfer the customer's parameter.

timeOut:

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

- **Return value**

RIL_AT_SUCCESS, succeed in executing AT command, and the response is OK.
RIL_AT_FAILED, fail to execute AT command, or the response is ERROR.
RIL_AT_TIMEOUT, indicates sending AT is timeout.
RIL_AT_BUSY, indicates sending AT.
RIL_AT_INVALID_PARAM, indicates invalid input parameter.
RIL_AT_UNINITIALIZED, indicates RIL is not ready, need to wait for MSG_ID_RIL_READY and then call Ql_RIL_Initialize() to initialize RIL.

- **Default Callback Function:**

If this callback parameter is set to NULL, a default callback function will be called. But the default callback function only handles the simple AT response. Please see *Default_atRsp_callback* in ril*_atResponse.c.* The following codes are the implementation for default callback function.

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

NOTE

When using RIL API please consider if one RIL API is not return, the other tasks which call RIL API will pending until the previous RIL API return.

4 Work with OpenCPU RIL

4.1. RIL Initialization

When the programs start up, RIL will send *MSG_ID_RIL_READY* message to main task. When received the message, the main task needs to call *Ql_RIL_Initialize* function to initialize RIL, including executing some initial AT commands that are defined in the variable *g_InitCmds*.

Besides, App needs to handle *MSG_ID_URC_INDICATION* message to receive URC messages. In *MSG_ID_URC_INDICATION* message, the parameter1 carries the URC type, and the parameter2 carries the specified URC-related information, which needs to be interpreted from URC type to URC type.

The following codes show how to initialize RIL in application.

```
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:
            Ql_Debug_Trace("<-- RIL is ready -->\r\n");
            ret = Ql_RIL_Initialize();
            break;
#endif
        default:
            break;
        }
    }
}
```

The variable *g_InitCmds* defines all initial AT commands that OpenCPU RIL depends on. Developer may add other AT commands, but should not delete the existing AT commands. The existing initial AT commands are listed as below.

Initial AT Commands	Description
AT+CMEE=1	Report mobile equipment error with numeric values
ATS0=0	No auto-answer the coming call
AT+CREG=1	Network register status indication
AT+CGREG=1	GPRS network status indication
AT+CLIP=1	RING number indication for the coming call
AT+COLP=0	No CLI (Connected Line Identification)

4.2. Program URC

All URC messages are indicated to App via the system message *MSG_ID_URC_INDICATION*. In this message, the parameter1 carries the URC type, and the parameter2 carries the specified URC-related information, which needs to be interpreted from URC type to URC type.

In OpenCPU RIL, URC contains two types: system URC and AT URC.

- System URCs indicate the various status of module.
- AT URC serves some specific AT command.
  For example, the response for some AT command is as below:

```
AT+QABC      (send AT command)

OK           (response1)
+QABC:xxx    (response2) --> this is the final result which is reported by URC.
```

When calling Ql_RIL_SendATCmd() to send such AT command, the return value of Ql_RIL_SendATCmd indicates the response1, and the response2 may be reported via the callback function, especially for some AT commands that the time span between response1 and response2 is very long, such as AT+QHTTPDL, AT+QFTPGET.

By default, OpenCPU demos some basic and necessary URCs, not all URCs. Developer may add new URC handling according to the real requirements. The following sample codes show the basic URCs indications and handling.

```
// receive and handle URCs in the system message MSG_ID_URC_INDICATIO
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:
            Ql_Debug_Trace("<-- RIL is ready -->\r\n");
            ret = Ql_RIL_Initialize();
            break;
        case MSG_ID_URC_INDICATION:
            switch (msg.param1)
            {
            case URC_SYS_INIT_STATE_IND:
                Ql_Debug_Trace("<-- Sys Init Status %d -->\r\n", msg.param2);
                break;
            case URC_SIM_CARD_STATE_IND:
                Ql_Debug_Trace("<-- SIM Card Status:%d -->\r\n", msg.param2);
                break;
            case URC_GSM_NW_STATE_IND:
                Ql_Debug_Trace("<-- GSM Network Status:%d -->\r\n", msg.param2);
                break;
            case URC_GPRS_NW_STATE_IND:
                Ql_Debug_Trace("<-- GPRS Network Status:%d -->\r\n", msg.param2);
                break;
            case URC_CFUN_STATE_IND:
                Ql_Debug_Trace("<-- CFUN Status:%d -->\r\n", msg.param2);
                break;
            case URC_COMING_CALL_IND:
                {
                    ST_ComingCall* pComingCall = (ST_ComingCall*)msg.param2;
                    Ql_Debug_Trace("<-- Coming call, number:%s, type:%d -->\r\n",
pComingCall->phoneNumber, pComingCall->type);
                    break;
                }
            case URC_CALL_STATE_IND:
                switch (msg.param2)
```

```
                {
                case CALL_STATE_BUSY:
                    Ql_Debug_Trace("<-- The number you dialed is busy now -->\r\n");
                    break;
                case CALL_STATE_NO_ANSWER:
                    Ql_Debug_Trace("<-- The number you dialed has no answer -->\r\n");
                    break;
                case CALL_STATE_NO_CARRIER:
                    Ql_Debug_Trace("<-- The number you dialed cannot reach -->\r\n");
                    break;
                case CALL_STATE_NO_DIALTONE:
                    Ql_Debug_Trace("<-- No Dial tone -->\r\n");
                    break;
                default:
                    break;
                }
                break;
            case URC_NEW_SMS_IND:
                Ql_Debug_Trace("<-- New SMS Arrives: index=%d\r\n", msg.param2);
                break;
            case URC_MODULE_VOLTAGE_IND:
                Ql_Debug_Trace("<-- VBatt Voltage Ind: type=%d\r\n", msg.param2);
                break;
            default:
                Ql_Debug_Trace("<-- Other URC: type=%d\r\n", msg.param2);
                break;
            }
            break;
 #endif
        default:
            break;
        }
    }
}
```

In the sample codes above, developer may watch the parameter "*msg.param2"* is interpreted as a different type from URC type to URC type.

For example, when the URC type is *URC_GPRS_NW_STATE_IND*, the parameter "*msg.param2"* carries the GPRS network status information, which is a 32 integer value (one value of *Enum_NetworkState*). When the URC type is *URC_COMING_CALL_IND*, the parameter "*msg.param2"* carries the coming call information, which is a structure (see also the definition of *ST_ComingCall*).

All system URC string from module operating system and the handler are defined in the variable "*m_SysURCHdlEntry*" in the file ril_urc.c. If there's a new system URC that needs to be processed, developer may refer to the implementation method for the existing URC to add the new URC string and the handler in "*m_SysURCHdlEntry*".

All AT commands-related URC string and the handler are defined in the variable "*m_AtURCHdlEntry*" in the file ril_urc.c. If there's a new AT URC that needs to be processed, developer may refer to the implementation method for the existing AT URC to add the new AT URC string and the handler in "*m_ATURCHdlEntry*".

4.3. Develop Mid Layer API

Over OpenCPU RIL, some primary API functions, such as telephony-related and short message-related, have been developed. Developer can program new API to support other functions. For example, to get IMSI of SIM card, here is the steps for implementation below.

- First, define a new API function named "RIL_SIM_GetIMSI"

Developer may create a new file *ril_sim.c* in the directory "SDK\ril\src\" to store this API function.
The prototype of this function is declared as below.

```
s32 RIL_SIM_GetIMSI(/*[in]*/char* pIMSI, /*[in]*/u32 pIMSILen)
```

The parameter *pImsi* is pointer to a buffer to store the IMSI string
The parameter *pIMSILen* indicates the buffer length of *pIMSI*. The buffer length is at least 15bytes.

- Secondly, implement RIL_SIM_GetIMSI

The following is the complete implementation codes for RIL_SIM_GetIMSI().

```
//
// Implementation for RIL_SIM_GetIMSI
//
s32 RIL_SIM_GetIMSI(/*[in]*/char* pIMSI, /*[in]*/u32 pIMSILen)
{
    if (!pIMSI || pIMSILen < 15)
    {
        return QL_RET_ERR_INVALID_PARAMETER;
    }
    Ql_memset(pIMSI, 0x0, pIMSILen);
    return Ql_RIL_SendATCmd("AT+CIMI", Ql_strlen("AT+CIMI"), ATResponse_IMSI_Handler, pIMSI,
0);
}
```

- Thirdly, implement the callback

```
//
// Implementation for ATResponse_IMSI_Handler
//   Note: All AT responses string will be passed into the callback line by line
//
static s32 ATResponse_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, p2 + 2, p2 - p1 -2);
            return RIL_ATRSP_CONTINUE;
        }else{
            return RIL_ATRSP_FAILED;
        }
    }
    return RIL_ATRSP_CONTINUE; // Wait for the next line of response
}
```

Developer should implement the callback function according to the kinds of responses of the AT command (please see document [1]). For example, the following diagram shows the usage and all kinds of responses of AT+CIMI.

AT+CIMI Request International Mobile Subscriber Identity (IMSI)	
Test Command **AT+CIMI=?**	Response **OK**
Execution Command **AT+CIMI**	Response TA returns <IMSI>for identifying the individual SIM which is attached to ME. **<IMSI>** **OK** If error is related to ME functionality: **+CME ERROR: <err>**
Reference GSM 07.07	

- Fourthly, use the new API

Now the new API has been made. Programmer can call the API to get IMSI number. The usage is shown as below.

```
extern s32 RIL_SIM_GetIMSI(char* pIMSI, u32 pIMSILen);

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

5 Appendix

Table 1: URC Definition

URC Message	Description
URC_SYS_INIT_STATE_IND	Module initial state indication during booting.
URC_SIM_CARD_STATE_IND	SIM card state indication
URC_GSM_NW_STATE_IND	GSM/UMTS/LTE network state indication
URC_GPRS_NW_STATE_IND	GPRS network state indication
URC_CFUN_STATE_IND	CFUN state indication
URC_COMING_CALL_IND	Coming call indication
URC_CALL_STATE_IND	Call state indication
URC_NEW_SMS_IND	New SMS indication
URC_MODULE_VOLTAGE_IND	Voltage indication when the power supply to module is not normal.

Table 2: Reference Document

SN	Document Name
[1]	AT Commands Manual of GSM/WCDMA/LTE modules
[2]	OpenCPU_User_Guide