MediaTek LinkIt™ Development Platform for RTOS Radio Interface Layer (RIL) Developer's Guide

Version: 1.0

Release date: 13 October 2017

Document Revision History

Revision	Date	Description
1.0	13 October 2017	Initial release

Table of contents

Lists of tables and figures

1. Introduction

1.1. Overview

MediaTek MT2625 platform provides a modem domain for protocol implementation and an application domain to develop applications with narrowband Internet of Things technology.

Communication with the MT2625 modem is through AT command interface. The user in AP domain sends AT commands to modem through CSCI or an external equipment, such as PC or Host chip. The commands can also communicate through UART and receive command response or unsolicited result codes from the modem. In AP domain, MT2625 provides a middleware layer named Radio Interface Layer (RIL) for more intuitive and convenient way to communicate with modem.

This document guides you through:

- Introduction to the RIL and its software architecture
- Detailed design of the RIL module
- Sample code on how to use the RIL interface

The software architecture on MT2625 application domain is shown in Figure 1.

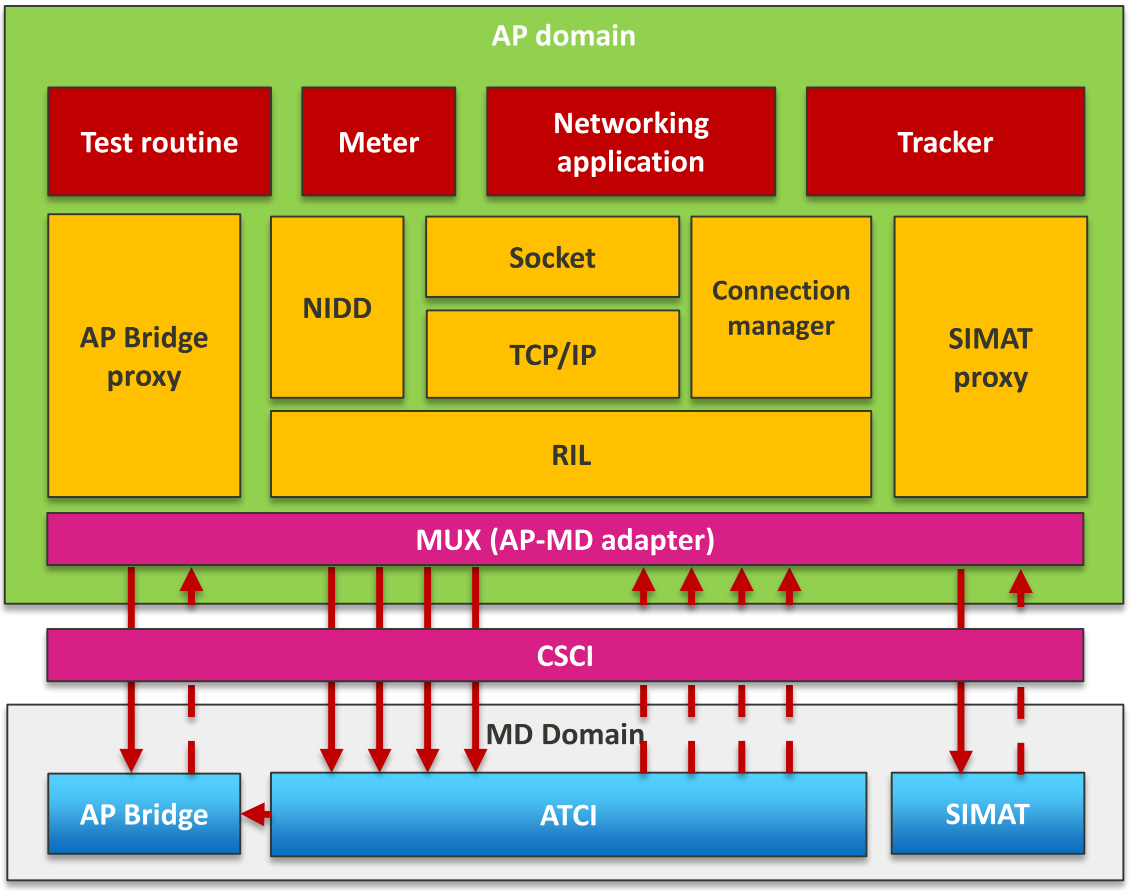

Figure 1. MT2625 software architecture in application domain

The modules with middleware services or applications call RIL interfaces. RIL sends the command data through MUX to modem upon request and receives response from the MUX layer after modem handles the command. The response is then parsed and the result codes and parameters are extracted.

1.2. RIL architecture

The RIL block diagram is shown in Figure 2. There are five sub-modules for RIL, including command sender, listener, response parser, URC parser and event callback registration. The following is the functionality description for each sub-module:

- Command sender
  - Constructs a command string according to user input parameters and calls MUX layer APIs to send data to the modem.
- Listener
  - Uses MUX layer interfaces to monitor each MUX channel, if data is received, pre-checks and parses patterns of a command string, classifies the command data type and notifies the RIL task to start processing.
- Response parser
  - Parses the received command data from the MUX channel according to the MT2625 AT command specification, extracts parameters included in the response and reports to the caller.
- URC parse
  - Parses unsolicited result codes from the modem.
- Event callback registration
  - RIL broadcasts received unsolicited result codes to the user listening to the events. Notifications are sent using callbacks.

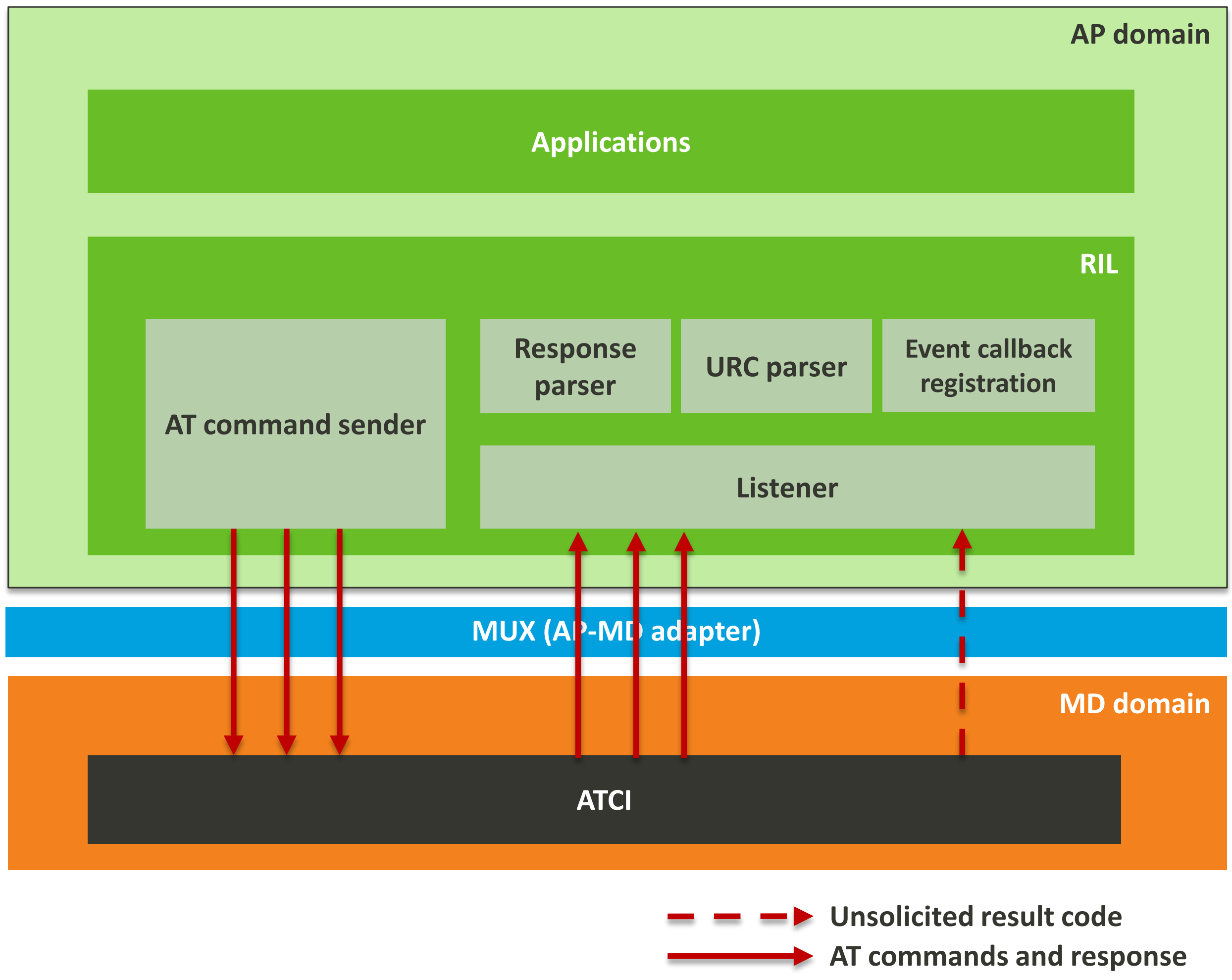

Figure 2. RIL block diagram

The tasks related to RIL are shown in Figure 3. It describes the workflow of using RIL interfaces in AP domain on MT2625 platform:

1) Call RIL APIs to send an AT command to the modem, the API implementation will first send a message with user input parameters to the RIL task.
2) RIL task handles the requested message and constructs an AT command string according to the input parameters. Then it calls the MUX API to send the message to the modem through Cross Side Communication Interface (CSCI).
3) The AT command response or unsolicited result codes (URC) are sent back to AP domain. The MUX task receives it and notifies the RIL (RIL callback is invoked).
4) RIL task parses the received data and determines if it's an AT response or a URC, and dispatches to the right handler.
5) The handler finishes parsing and extracting parameters in a command string, and reports to caller (invokes user callback that was passed when calling RIL interfaces).
6) Callback runs in RIL's call stack. Note that to process heavy operations send the message to the user's own task.

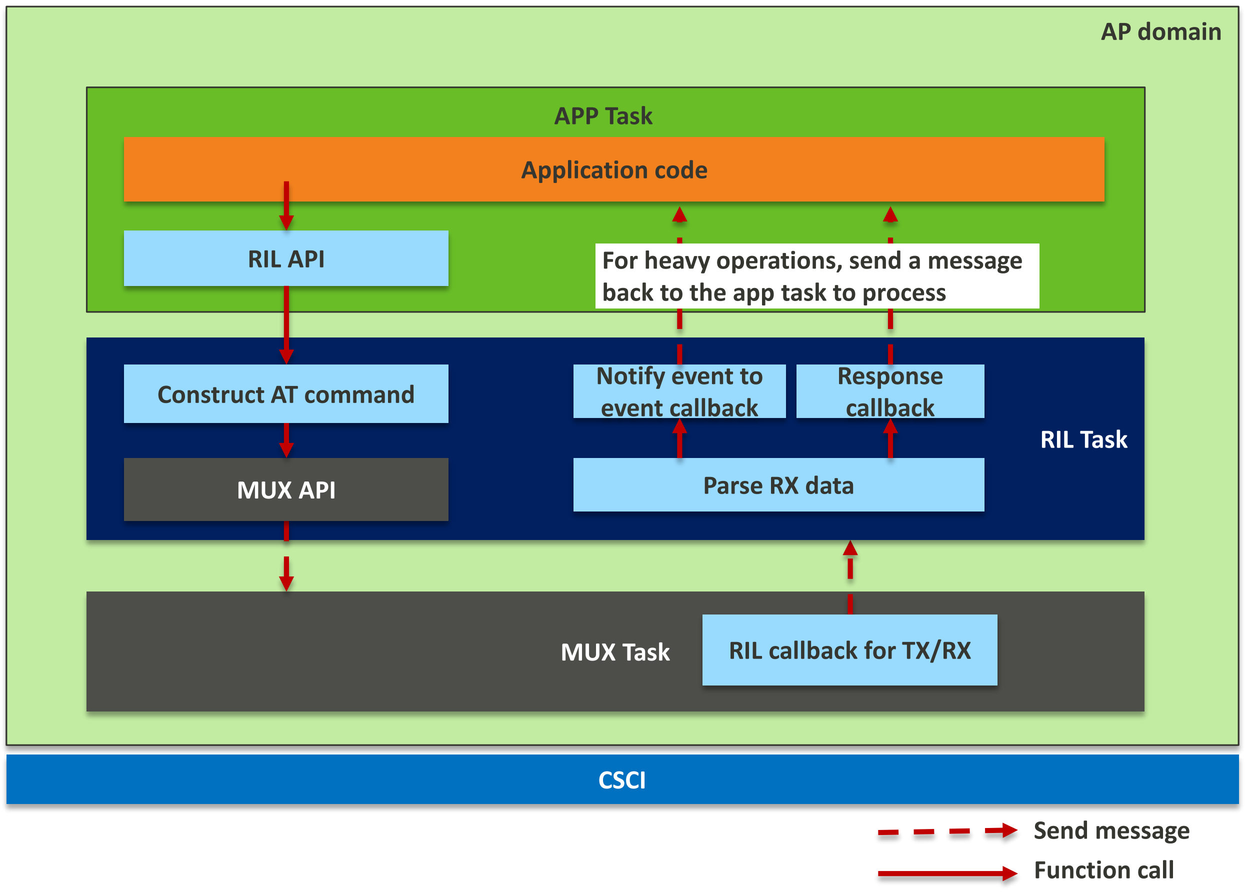

Figure 3. RIL related task workflow

2. RIL Design

This section describes the RIL design, including folder and file structure, channel management and communication between connection manager and lwIP.

2.1. RIL source folder and file structure

This section describes the source code and header file hierarchy for RIL and its usage. The files are all located at `<sdk_root>/middleware/MTK/ril.`

Table 1 provides details on the file/folder structure in the RIL root folder.

Table 1. RIL root folder content

File	Description
`middleware/MTK/ril/module.mk`	RIL module Makefile. To use it, simply include in your project Makefile.
`middleware/MTK/ril/src`	This folder contains all source files for RIL module.
`middleware/MTK/ril/inc`	This folder contains all header files for RIL module.

Table 2 provides details of source and header files in the RIL module.

Table 2. RIL source file description

File	Description
`middleware/MTK/ril/inc/ril.h`	The header file of all exported interfaces from RIL module, including functions, data structures, macros and enumerations. Include this header file to use RIL interfaces.
`middleware/MTK/ril/inc/ril_channel_config.h`	The number of URC channels and AT and Data channel is defined in this file. Configuration modification is prohibited.
`middleware/MTK/ril/src(inc)/ril_cmds_27007.c(.h)`	Commands in 3GPP Technical Specification 27.007.
`middleware/MTK/ril/src(inc)/ril_cmds_27005.c(.h)`	Commands in 3GPP Technical Specification 27.005.
`middleware/MTK/ril/src(inc)/ril_cmds_v250.c(.h)`	Commands in 3GPP Technical Specification v250.
`middleware/MTK/ril/src(inc)/ril_cmds_proprietary.c(.h)`	MediaTek proprietary commands.
`middleware/MTK/ril/src(inc)/ril_setting.c(.h)`	AT command format settings and special commands sendt after power on.
`middleware/MTK/ril/src(inc)/ril_utils.c(.h)`	Utiliy for string parsing, token checking and memory management.
`middleware/MTK/ril/src(inc)/ril_task.c(.h)`	RIL task creation, mesage handling and channel management.

2.2. Channel management

There are serial channels provided by MUX layer to communicate with modem and partial channels managed by RIL. the channels are classified into two types - URC channel and AT & Data channel. The following is functionality description for these two types of channels:

- URC channel:
  - Dedicated to configure URC reporting and receive URC.
- AT & Data channel:
  - Used to send commands without URC reporting. Usually it stays at command mode and enables the AT command string to send and receive, but if the connection manager activates a data channel, it may change to data mode to transfer binary data, such as IP data. At this time, the channel is controlled by the lwIP module instead of RIL. lwIP could retrieve IP data from MUX layer directly, no need to bypass through other modules. Once the data transfer is finished, the connection manager helps to terminate and change this channel back to command mode, so that the RIL can manage this channel again.

One channel is reserved for URC channel dedicated to process URC related commands. The rest are defined as AT & Data channel. Please refer to `<ril_root_path>/inc/ril_channel_config.h` for channel number configuration. The default settings are shown below:

```
/*confgiure the number of channels */
#if defined(MT2503_2625_DUAL)
#define RIL_URC_CHANNEL_NUM    (1)
#define RIL_AT_DATA_CHANNEL_NUM    (2)
#else
#define RIL_URC_CHANNEL_NUM    (1)
#define RIL_AT_DATA_CHANNEL_NUM    (5)
#endif
```

2.3. RIL workflow

2.3.1. Initializing the RIL

RIL registers callback function to MUX layer for each channel, listens to the events reported by MUX, including registration success and failure, data receiving preparation, data receiving completion, data sending completion and more. If registration event is successful, all RIL channels are set to ready state. The flow chart is shown in Figure ***4***.

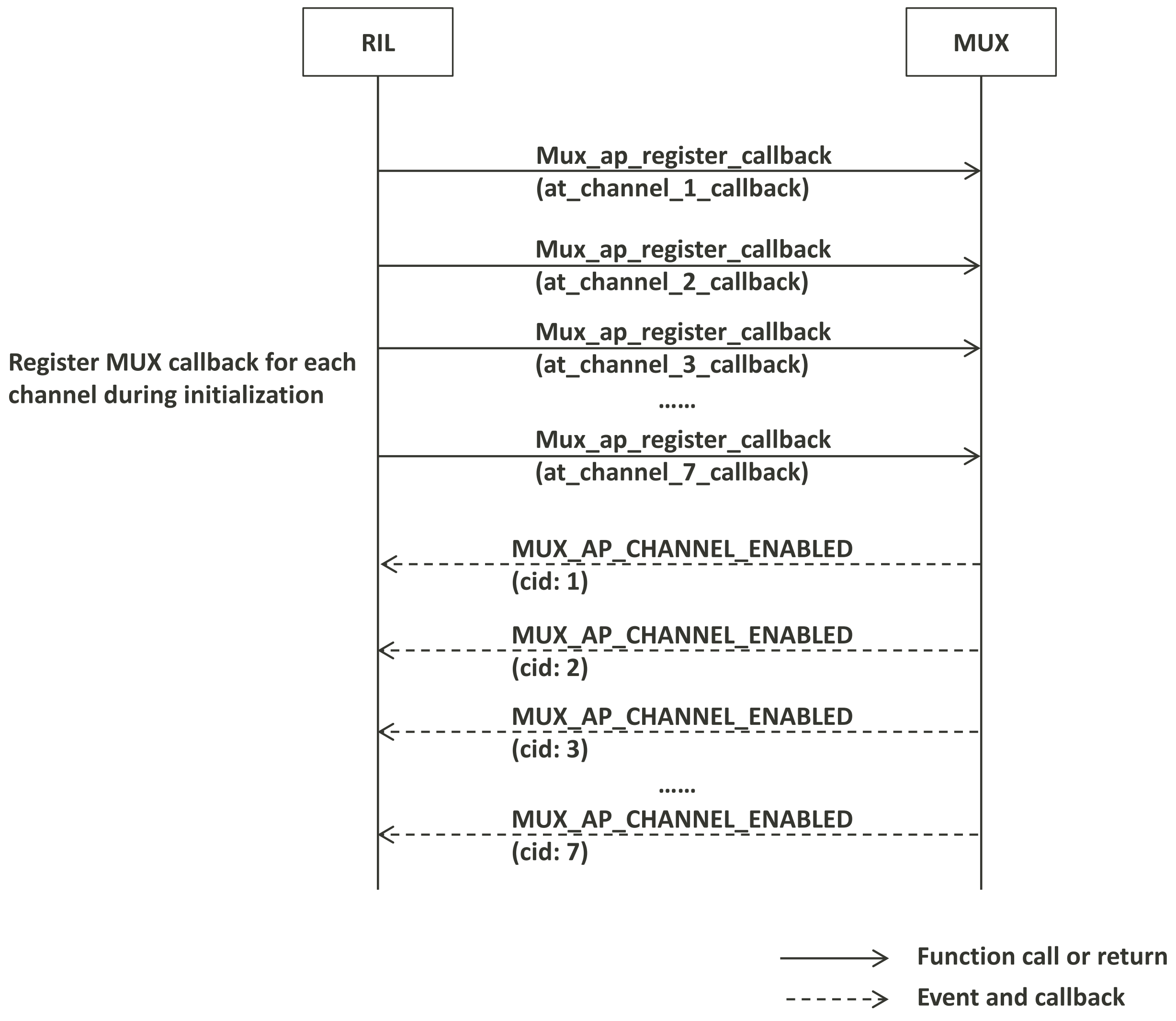
RIL
MUX
Mux_ap_register_callback (at_channel_1_callback)
Mux_ap_register_callback (at_channel_2_callback)
Mux_ap_register_callback (at_channel_3_callback)
……
Mux_ap_register_callback (at_channel_7_callback)
Register MUX callback for each channel during initialization
MUX_AP_CHANNEL_ENABLED (cid: 1)
MUX_AP_CHANNEL_ENABLED (cid: 2)
MUX_AP_CHANNEL_ENABLED (cid: 3)
……
MUX_AP_CHANNEL_ENABLED (cid: 7)
Function call or return
Event and callback

Figure** 4. **RIL channel registration to MUX layer when initializing phase

2.3.2. Sending command flow

AT commands are classified into several groups indicating specific functionality. There is a significant channel restriction on AT command grouping; the commands in specific groups cannot be executed on different channels at the same time. The RIL module holds the commands and only sends the first command to the modem. Only when the response of the first command is received, the next command in the same group will be sent out. Currently there are 10 groups:

- Power Management (PM)
- Mobility Management (MM)
- Short message (SMS)
- SIM (SIM)
- Packet Data (PD)
- Engineering Mode (EM)

- SIM toolkit (STK)
- MUX (MUX)
- N/A (NONE, no specific group, no channel restrictions)

These classifications are defined in `ril_func_group_t` in `ril_cmds_def.h`.

When application calls RIL interfaces to send an AT command, the API creates a message with the input command parameters and sends it to the RIL task. The RIL task checks the function group and channel state and allocates an available channel to transfer the command. The channel is occupied until the response is received. The sending command flow is shown in Figure 5.

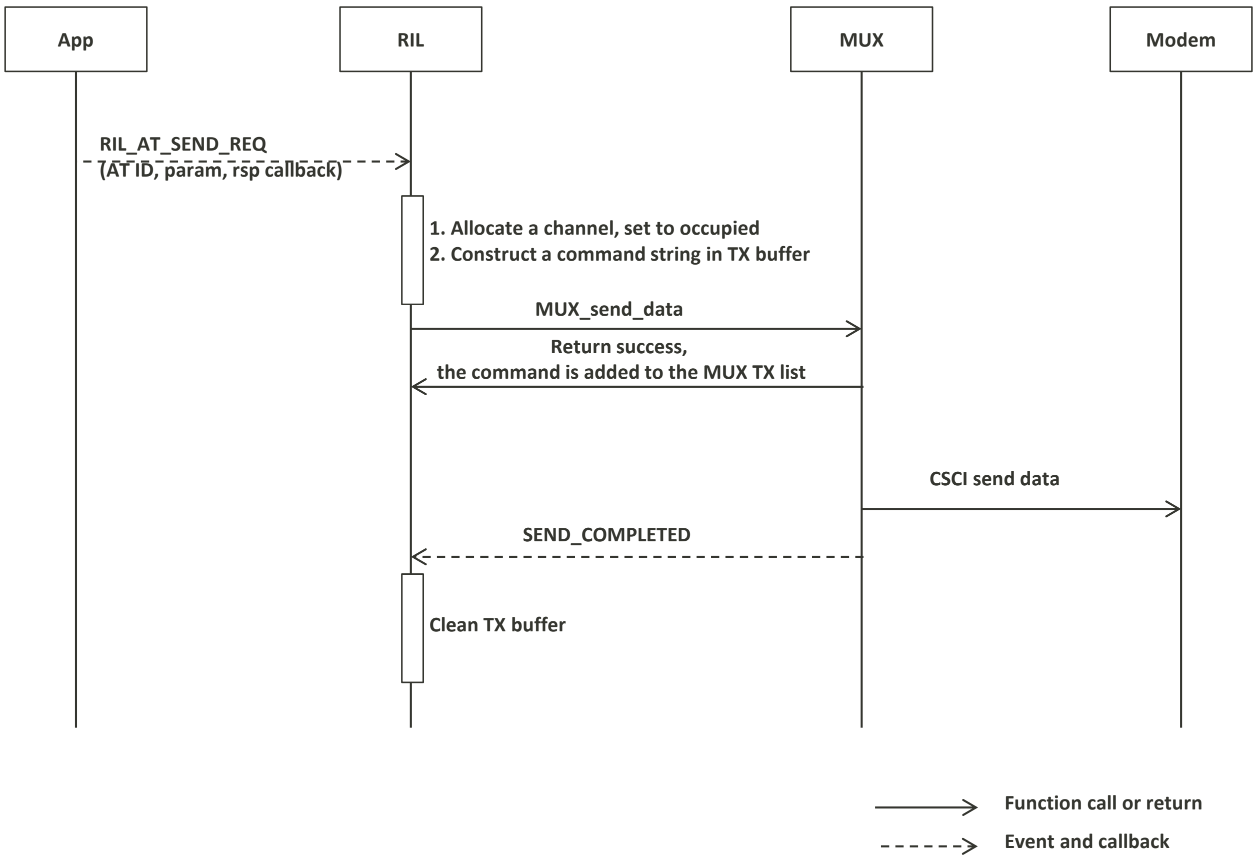
App
RIL
MUX
Modem
RIL_AT_SEND_REQ (AT ID, param, rsp callback)
1. Allocate a channel, set to occupied
2. Construct a command string in TX buffer
MUX_send_data
Return success, the command is added to the MUX TX list
CSCI send data
SEND_COMPLETED
Clean TX buffer
Function call or return
Event and callback

Figure 5. The sending command flow

If the channel or function group is busy, RIL will enqueue the sending request and continue processing other commands. Once an available channel is found or the command function group is ready, a hold request will be dequeued and handled immediately. The sending command flow in channel busy or function group busy case is shown in Figure 6.

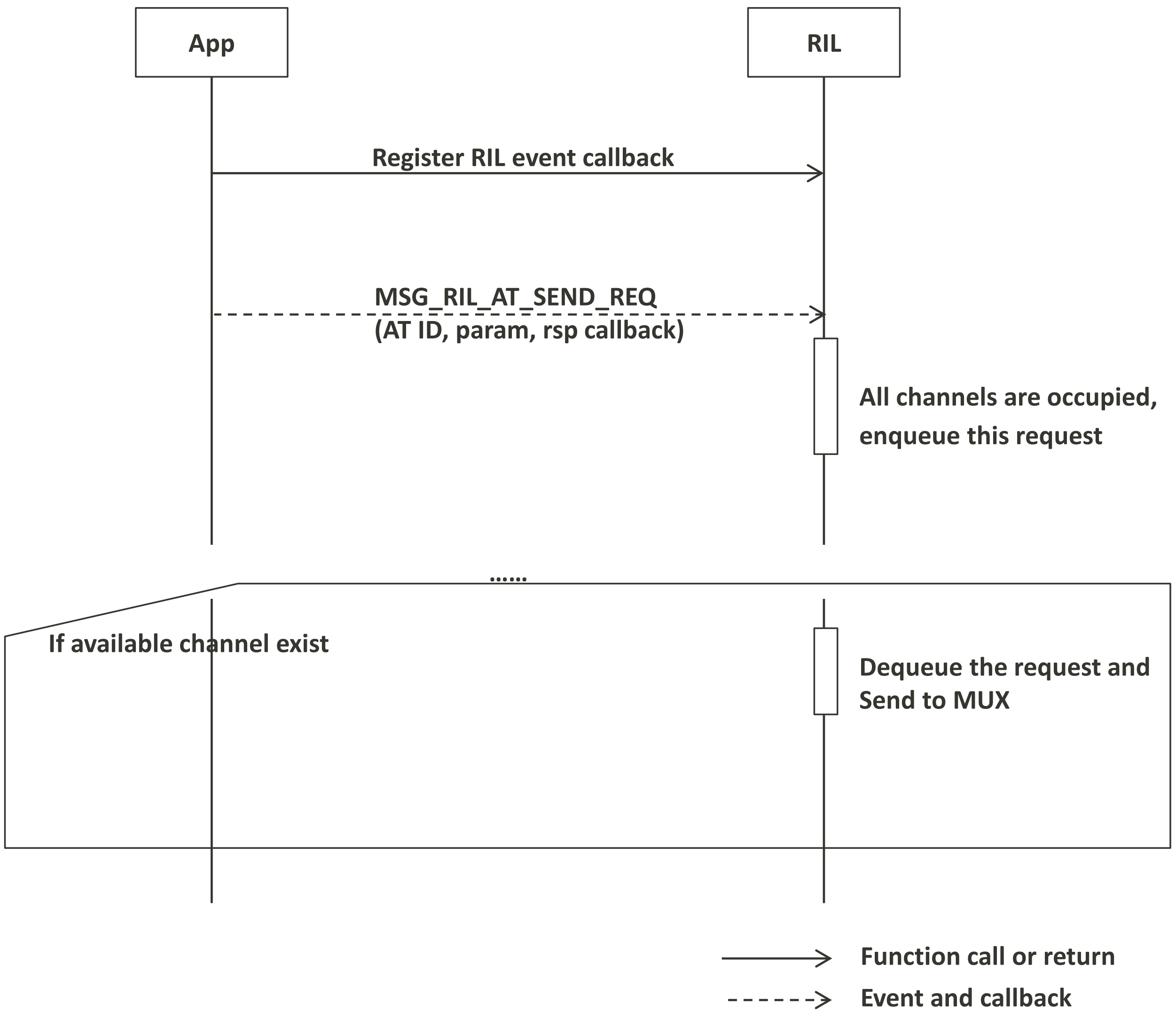
App
RIL
Register RIL event callback
MSG_RIL_AT_SEND_REQ
(AT ID, param, rsp callback)
All channels are occupied, enqueue this request
……
If available channel exist
Dequeue the request and Send to MUX
Function call or return
Event and callback

Figure 6. The sending command flow in channel or function group busy case

2.3.3. Receiving command flow

The receiving command response flow is shown in Figure 7.

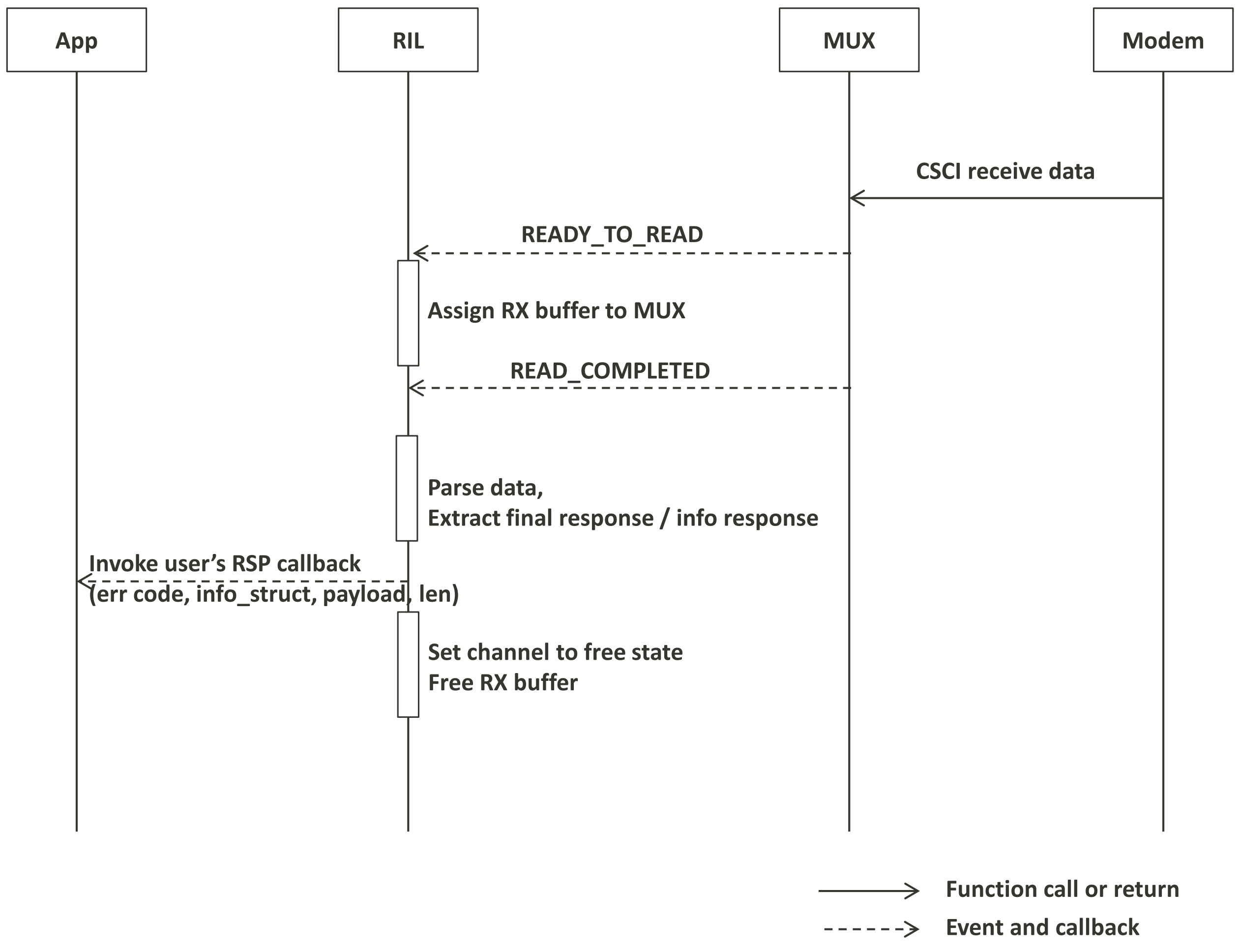

Figure 7. The receiving command response flow

2.3.4. Receiving unsolicited result codes flow

The receiving unsolicited result codes flow is similar to the AT response receiving flow. The URC event notification with extracted parameters is sent to multiple users one by one. The workflow is shown in Figure 8.

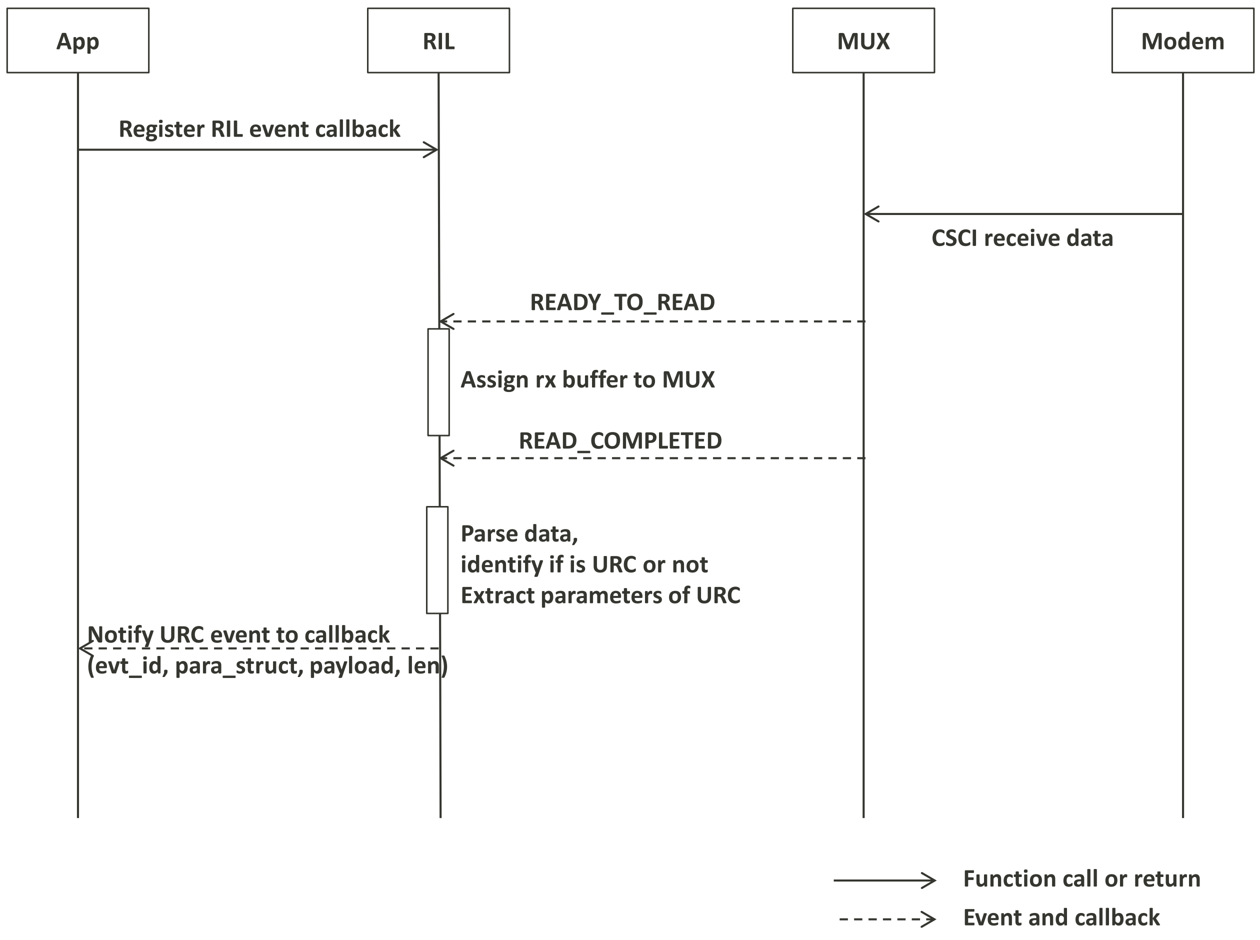
App
RIL
MUX
Modem
Register RIL event callback
CSCI receive data
READY_TO_READ
Assign rx buffer to MUX
READ_COMPLETED
Parse data,
identify if is URC or not
Extract parameters of URC
Notify URC event to callback
(evt_id, para_struct, payload, len)
Function call or return
Event and callback

Figure 8. The receiving unsolicited result codes flow

2.4. General settings

To parse the command string, RIL has to regulate the command string format to conveniently check the token and extract parameters included in the command string. Formatting commands will be sent to the modem during initialization, if necessary. In addition, there are other commands related to special functionality to send to modem after power on. There is a general configuration flow in RIL module that runs automatically after boot up. Commands are classified into two types, format settings and special command settings.

- Format settings
  - AT command formatting configuration is set to the channel. Each channel can have its own formatting settings. Thus, the commands must be sent to each RIL channel, e.g. ATE0, AT+CMEE=1
- Special command settings
  - Turn on or enable specific functionality, e.g. AT+CFUN=1 (set to full modem function), enable URC reporting for indication of wakeup from deep sleep. Send the command only once. Please refer to the following two tables defined in `<ril_root_path>/src/ril_general_setting.c`.

3. How to use RIL interfaces

This section provides sample codes to demonstrate how to send an AT command and receive AT response and listen to URC events through RIL interfaces.

3.1. Sending an AT command

An example to query current network registration status by calling the AT command AT+CEREG is described

Search for "+CEREG" keyword in `ril.h` header file to find the related interfaces, as shown below:

```
typedef struct {
    int32_t n;    /**< Indicates which display mode to configure when "+CEREG"
URC is received. */
    int32_t stat;    /**< Indicates EPS registration status. */
    int32_t tac;    /**< Two byte tracking area code in hexadecimal format. */
    int32_t lac;    /**< String type, two bytes location area code in
hexadecimal format. */
    int32_t ci;    /**< String type, four byte GERAN/UTRAN/E-TRAN cell id in
hexadecimal format. */
    int32_t act;    /**< Access technology of the registered network. */
    int32_t rac;    /**< String type, one byte routing area code in hexadecimal
format. */
    int32_t cause_type;    /**< Integer type, indicates the type of the field
"reject cause". */
    int32_t reject_cause;    /**< Integer type, contains the cause of the
failed registration. */
    int32_t active_time;    /**< One byte in an 8 bit format, indicates the
Active Time value allocated to the UE in E-UTRAN. */
    int32_t periodic_tau;    /**< One byte in an 8 bit format, indicates the
extended periodic TAU value allocated to the UE in E-UTRAN. */
} ril_eps_network_registration_status_rsp_t;



typedef struct {
    int32_t stat;    /**< Indicates EPS registration status. */
    int32_t lac;    /**< String type, two bytes location area code in
hexadecimal format. */
    int32_t ci;    /**< String type, four byte GERAN/UTRAN/E-TRAN cell id in
hexadecimal format. */
    int32_t act;    /**< Access technology of the registered network. */
    int32_t rac;    /**< String type, one byte routing area code in hexadecimal
format. */
```

```
    int32_t cause_type;    /**< Integer type, indicates the type of the field
"reject cause". */
    int32_t reject_cause;    /**< Integer type, contains the cause of the
failed registration. */
    int32_t active_time;    /**< One byte in an 8 bit format, indicates the
Active Time value allocated to the UE in E-UTRAN. */
    int32_t periodic_tau;    /**< One byte in an 8 bit format, indicates the
extended periodic TAU value allocated to the UE in E-UTRAN. */
} ril_eps_network_registration_status_urc_t;


ril_status_t ril_request_eps_network_registration_status(
        ril_request_mode_t mode,
        int32_t n,
        ril_cmd_response_callback_t callback,
        void *user_data);
```

- `ril_eps_network_registration_status_rsp_t`: represents the response parameters of the AT "+CEREG" command.
- `ril_eps_network_registration_status_urc_t`: represents the response parameters of the URC "+CEREG" command.
- `ril_request_eps_network_registration_status`: sends the AT command "+CEREG".

Before using the RIL interfaces, clarify the following items:

1) "AT+CEREG?" should send command with READ MODE.
2) The parameter "n" is omitted in read mode, pass the macro `RIL_OMITTED_INTEGER_PARAMETER` in `ril.h`.
3) Define a response callback to receive parameters following the prototype `ril_cmd_response_callback_t`, and before using the included parameters ensure:
   a) casting "`void *cmd_param`" to the right data structure.
   b) checking if "`cmd_param`" is valid or not. If the modem returns ERROR or other exception error codes, the "`cmd_param`" should be NULL, as there is no parameter to report to the user.

An example code to send a command:

```
/* response callback defintion */
int32_t ril_CEREG_response_callback(ril_cmd_response_t *cmd_response)
{
    switch (cmd_response->cmd_id) {
        case RIL_CMD_ID_CEREG:
            if (cmd_response->res_code == RIL_RESULT_CODE_OK &&
                cmd_response->cmd_param != NULL) {
```

```
                // cast to right data structure.
                ril_eps_network_registration_status_rsp_t *param =
(ril_eps_network_registration_status_rsp_t *)cmd_response->cmd_param;
                printf(”current state: %ld\r\n”, param->stat);
                /* Do something. If heavy handling, please send message to your
own task to process. Because callback runs in RIL’s task callstack, make sure
don’t block RIL task execution. */
                ......
            }
        break;
        ......
        default:
        break;
   }
}

/* app code */
APP_func() {
    ........
    /* send AT+CEREG? to modem */
   ril_status_t ret;
   ret = ril_request_eps_network_registration_status(
            RIL_READ_MODE,
            RIL_OMITTED_PARAMETER_INTEGER,
            ril_CEREG_response_callback,
            NULL);
}
```

3.2. Listening to the URC

An example is provided to listen the network registration status change using “+CEREG”.

First, send an AT command to enable the URC reporting. The command response callback could reuse “`ril_CEREG_response_callback`” defined in section 3.1, "Sending an AT command. The command string to enable URC reporting is “AT+CEREG=2”, as shown in the example code below:

```
App_func()
{
   /* enable +CEREG URC reporting */
   ril_status_t ret;
   ret = ril_request_eps_network_registration_status(
            RIL_EXECUTE_MODE,
            2,
            ril_CEREG_response_callback,
            NULL);
}
```

URC is broadcast to all users listening to this event. The user should register a callback to RIL module before listening. The registration API is defined in `ril.h`, as shown in the function declaration below:

```
ril_status_t ril_register_event_callback(uint32_t group_mask,
ril_event_callback_t callback);
```

The sample code to listen to the "+CEREG" URC:

```
int32_t ril_URC_event_demo_callback(ril_urc_id_t event_id, void *param,
uint32_t param_len)
{
    switch (event_id) {
        case RIL_URC_ID_CEREG:
            /* cast to the right data structure */
            ril_eps_network_registration_status_urc_t *urc_param =
(ril_eps_network_registration_status_urc_t *)param;
            printf("current network stat: %ld\r\n", urc_param->stat);
            /* Do something, if heavy handling, please send message to your
own task to process. Because callback runs in RIL's task callstack, make sure
don't block RIL task execution. */
            ......
        break;
        ......
        default:
        break;
    }
}


App_func()
{
    /* register to urc event callback */
    ril_status_t ret;
    ret = ril_register_event_callback(RIL_GROUP_MASK_ALL,
ril_URC_event_demo_callback);
}
```