Hi3861 V100 / Hi3861L V100 RPL

Development Guide

Issue 01

Date 2020-04-30

HiSilicon (Shanghai) Technologies Co., Ltd.

Address:	New R&D Center, 49 Wuhe Road, Bantian, Longgang District, Shenzhen 518129 P. R. China
Website:	https://www.hisilicon.com/en/
Email:	support@hisilicon.com

About This Document

Purpose

This document describes the application programming interfaces (APIs) in the RPL module of Hi3861 V100 and provides development samples.

Related Versions

The following table lists the product versions related to this document.

Product Name	Version
Hi3861	V100
Hi3861L	V100

Intended Audience

This document is intended for:

- Software development engineers
- Technical support engineers

Symbol Conventions

The symbols that may be found in this document are defined as follows.

Symbol	Description
⚠ DANGER	Indicates a hazard with a high level of risk which, if not avoided, will result in death or serious injury.
⚠ WARNING	Indicates a hazard with a medium level of risk which, if not avoided, could result in death or serious injury.

Symbol	Description
⚠ CAUTION	Indicates a hazard with a low level of risk which, if not avoided, could result in minor or moderate injury.
NOTICE	Indicates a potentially hazardous situation which, if not avoided, could result in equipment damage, data loss, performance deterioration, or unanticipated results. NOTICE is used to address practices not related to personal injury.
NOTE	Supplements the important information in the main text. NOTE is used to address information not related to personal injury, equipment damage, and environment deterioration.

Change History

Issue	Date	Change Description
01	2020-04-30	This issue is the first official release.
00B01	2020-04-10	This issue is the first draft release.

Contents

1 API Description

1.1 Overview

RPL stands for IPv6 Routing Protocol for Low Power and Lossy Networks. It is a lossy network routing protocol with low power consumption, and is used in a network in which both an internal link and a router are limited. Processor functions, memory, and system power consumption (battery power supply) of a router in the network may be greatly limited. The network connection also features a high packet loss rate, a low data transmission rate, and instability.

NOTE

The APIs described in this document are the original external APIs of the RPL protocol. Currently, the upper-layer self-networking module of the Wi-Fi software has called these APIs and encapsulated them as function calling APIs. This helps you directly use the self-networking function. If you need to use these original APIs in the RPL protocol for development, refer to this document, especially the code samples.

Table 1-1 lists the RPL functional APIs provided by the current protocol stack.

Table 1-1 RPL APIs

API Name	Description
rpl_init	Initializes the RPL module.
rpl_deinit	Deinitializes the RPL module.
rpl_start	Starts the RPL module.
rpl_stop	Stops the RPL module
rpl_route_entry_count	Obtains the number of routes in the RPL.
rpl_route_entry_get	Obtains the routing table in the RPL.
rpl_rank_get	Obtains the **rank** value of an RPL node.
rpl_rank_threshold_set	Sets the **rank** threshold of an RPL node.
l3_event_msg_callback_reg	Registers an upper-layer callback function.
lwip_nat64_init	Initializes NAT64.
lwip_nat64_deinit	Deinitializes NAT64.
netifapi_dhcp_clients_info_get	Obtains information about the DHCP client.
netifapi_dhcp_clients_info_free	Frees information about the DHCP client of the corresponding API.

1.2 rpl_init

Prototype	int rpl_init(char *name, uint8_t len);
Description	Initializes the RPL (for example, allocating memory).
Argument	• **name**: netif name • **len**: length of the netif name (unit: byte)

Return	• Success: A positive integer (context index) is returned. • Failure: The value **–1** is returned.
Error Code	-
Availability Since	nStack_N500 1.0.2

1.3 rpl_deinit

Prototype	int rpl_deinit(int ctx);
Description	Deinitializes the RPL (for example, freeing the memory).
Argument	• **ctx**: context index, which is the value returned by **rpl_init**
Return	• Success: The value **0** is returned. • Failure: The value **–1** is returned.
Error Code	-
Availability Since	nStack_N500 1.0.2

1.4 rpl_start

Prototype	int rpl_start(int ctx, uint8_t mode, uint8_t inst_id, const rpl_config_t *cfg, const rpl_ip6_prefix_t *prefix);
Description	Starts the RPL.

Argument	• **ctx**: context index, which is the value returned by **rpl_init**
	• **mode**: network access role of the current node. The value options are as follows:
	  – **RPL_MODE_MBR**: The node accesses the network as an MBR role.
	  – **RPL_MODE_MG**: The node accesses the network as an MG role. (It is not used by the subsequent parameters.)
	• **inst_id**: instance ID
	• **cfg**: DODAG config (not used currently)
	• **prefix**: DODAG prefix (IPv6 address)

Return	• Success: The value **0** is returned. • Failure: The value **–1** is returned.
Error Code	-

Availability Since	nStack_N500 1.0.2

1.5 rpl_stop

Prototype	int rpl_stop(int ctx);
Description	Stops the RPL. This operation clears the routing table and stops the timer.
Argument	• **ctx**: context index, which is the value returned by **rpl_init**
Return	• Success: The value **0** is returned. • Failure: The value **–1** is returned.
Error Code	-
Availability Since	nStack_N500 1.0.2

1.6 rpl_route_entry_count

Prototype	int rpl_route_entry_count(int ctx, uint16_t *cnt);
Description	Obtain the number of routes.
Argument	• **ctx**: context index, which is the value returned by **rpl_init** • **cnt**: number of routes (output argument)
Return	• Success: The value **0** is returned. • Failure: The value **–1** is returned.
Error Code	-
Availability Since	nStack_N500 1.0.2

1.7 rpl_route_entry_get

Prototype	int rpl_route_entry_get(int ctx, rpl_route_entry_t *rte, uint16_t cnt);
Description	Obtains the routing table.

Argument	● **ctx**: context index, which is the value returned by **rpl_init** ● **rte**: memory of the copied routing table (output argument) ● **cnt**: number of entries in the routing table memory **rte**
Return	● Success: A positive integer (number of copied routes) is returned. ● Failure: The value **–1** is returned.
Error Code	-
Availability Since	nStack_N500 1.0.2

1.8 rpl_rank_get

Prototype	int rpl_rank_get(uint16_t *rank);
Description	Obtains the **rank** value of a node.
Argument	● **rank**: rank value for storage (output argument)
Return	● Success: The value **0** is returned. ● Failure: The value **–1** is returned.
Error Code	-
Availability Since	nStack_N500 1.0.2

1.9 rpl_rank_threshold_set

Prototype	int rpl_rank_threshold_set(uint16_t rank_threshold);
Description	Sets the **rank** threshold of a node.
Argument	● **rank_threshold**: rank threshold of the node
Return	● Success: The value **0** is returned. ● Failure: The value **–1** is returned.
Error Code	-
Availability Since	nStack_N500 1.0.2

1.10 l3_event_msg_callback_reg

Prototype	void l3_event_msg_callback_reg(enum l3_event_msg_type evt_type, app_callback_fn app_callback);
Description	Registers the upper-layer callback function.
Argument	● **evt_type**: type of the event reported in the RPL ● **app_callback**: callback function corresponding to the event type
Return	-
Error Code	-
Availability Since	nStack_N500 1.0.2

1.11 lwip_nat64_init

Prototype	int lwip_nat64_init(char *name, uint8_t len);
Description	Initializes NAT64.
Argument	● **name**: network API name ● **len**: length of the API name (unit: byte)
Return	● Success: The value **0** is returned. ● Failure: The value **–1** is returned.
Error Code	-
Availability Since	nStack_N500 1.0.2

1.12 lwip_nat64_deinit

Prototype	int lwip_nat64_deinit(void);
Description	Deinitializes NAT64, including clearing the NAT64 table and stopping the DHCP proxy.
Argument	-
Return	● Success: The value **0** is returned. ● Failure: The value **–1** is returned.

Error Code	-
Availability Since	nStack_N500 1.0.2

1.13 netifapi_dhcp_clients_info_get

Prototype	netifapi_dhcp_clients_info_get(struct netif *netif, struct dhcp_clients_info **clis_info);
Description	Obtains information about a DHCP client.
Argument	• **netif**: network API corresponding to the DHCP client • **clis_info**: obtained DHCP client information (output argument)
Return	• Success: **ERR_OK** (the value is 0) is returned. • Failure: A negative value is returned.
Error Code	-
Availability Since	nStack_N500 1.0.2

1.14 netifapi_dhcp_clients_info_free

Prototype	netifapi_dhcp_clients_info_free(struct netif *netif, struct dhcp_clients_info **clis_info);
Description	Frees the DHCP client information of the corresponding API.
Argument	• **netif**: network API corresponding to the DHCP client • **clis_info**: dhcp_clients_info information obtained by calling **netifapi_dhcp_clients_info_get**
Return	• Success: **ERR_OK** (the value is 0) is returned. • Failure: A negative value is returned.
Error Code	-
Availability Since	nStack_N500 1.0.2

2 Development Guidance

2.1 Development Restrictions

The RPL module has the following restrictions:

- A node can store a maximum of 128 downstream routing tables.
- Each DAG node supports a maximum of four parent nodes.
- Each RPL instance supports one DODAG.
- The DIO message supports only one prefix.
- A DAO message supports a maximum of nine unicast targets.
- In the RPL protocol, the MBR module needs to use an IPv4 address to communicate with a router. However, a mesh network does not use IPv4 for communication, but uses IPv6 for data and mechanism synchronization.

2.2 Code Samples

Sample 1:

1. Initialize and start an RPL network by calling **rpl_init** and **rpl_start**.
2. Start the DHCP service and enable the NAT64 function.
3. Disable the DHCP service, exit NAT64, and clear the configuration.
4. Disable the RPL network and free related resources.

```
#include "lwip/lwip_ripple_api.h"
#include "lwip/netif.h"
#include "lwip/nat64_api.h"
#include "lwip/netifapi.h"
#include "hi_at.h"
#define WIFI_IFNAME_MAX_SIZE  16
#define RPL_MODE_MBR 1 /* Root mode */
#define RPL_MODE_MG 2  /* Router mode */
hi_u8 g_rpl_prefix[8] = {0xFD, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
hi_void hi_rpl_process_example()
```



```
{
    rpl_ip6_prefix_t prefix = {0};
    hi_s32 ctx;
    char ifname[WIFI_IFNAME_MAX_SIZE + 1] = "test_wifi_rpl";

    if (memcpy_s(prefix.prefix, sizeof(prefix.prefix), g_rpl_prefix, 8) != EOK) {
        return;
    }
    prefix.len = 8;

    ctx = rpl_init(ifname, strlen(ifname));
    if (ctx < 0) {
        hi_at_printf("rpl_init fail.\r\n");
        return;
    }

    if (rpl_start(ctx, RPL_MODE_MG, 99, NULL, &prefix) != 0) {
        hi_at_printf("rpl_start fail!\r\n");
        return;
    }
    hi_at_printf("start rpl example succ!\r\n");
...
...
    struct netif *lwip_netif = NULL;
    lwip_netif = netif_find("wlan0");
    if (lwip_netif == NULL) {
        return;
    }
    /* here we start dhcp first, and then init nat64 */
    netifapi_dhcp_start(lwip_netif);
    if (lwip_nat64_init("wlan0", 5) != 0) {
        hi_at_printf("nat64_init fail!\r\n");
        return;
    } else {
        hi_at_printf("nat64_init succ!\r\n");
    }
...
...
    /* here we stop dhcp first, and then deinit nat64 */
    netifapi_dhcp_stop(lwip_netif);
    if (lwip_nat64_deinit() != 0) {
    hi_at_printf("nat64_deinit fail!\r\n");
        return;
    }
...
...
    ret = rpl_deinit(ctx);
    if (ret == -1) {
        hi_at_printf("rpl deinit fail!\r\n");
        return;
    }
    hi_at_printf("mesh mr stop success!\r\n");
    return;
}
```

Sample 2

- Call **rpl_route_entry_count** to obtain the number of routes based on the received **ctx**.
- Call **rpl_route_entry_get** to store the obtained entries to the structure array **rte**.


```
#include "lwip_ripple_api.h"
#include "hi_at.h"
int mesh_rpl_get_entry_example(hi_s32 ctx, hi_u8 sock_type)
{
    hi_u16 item_count = 0;
    /* here we get the rpl route entry count */
    if (rpl_route_entry_count(ctx, &item_count) != ERR_OK) {
        hi_at_printf("rpl_route_entry_count failed. nodeCount:%d\r\n", item_count);
        return -1;
    } else {
        hi_at_printf("rpl_route_entry_count succ. nodeCount : %d\r\n", item_count);
    }

    rpl_route_entry_t rte[10] = {0};
    rpl_route_entry_t *rte_ptr = rte;
    /* here we get the rpl route entry */
    if (rpl_route_entry_get(ctx, rte_ptr, item_count) < 0) {
        hi_at_printf("rpl_route_entry_get failed.\r\n");
        return -1;
    } else {
        hi_at_printf("rpl_route_entry_get succ.\r\n");
    }
    return 0;
}
```

Sample 3: Obtain the **rank** value of a node.

```
#include "lwip/lwip_ripple_api.h"
#include "hi_at.h"
int rpl_rank_get_example(void)
{
    hi_u16 my_rank = 0;
    int ret;
    ret = rpl_rank_get(&my_rank);
    if (ret == -1) {
        hi_at_printf("rpl_rank_get failed!\r\n");
        return -1;
    } else {
        hi_at_printf("rpl_rank_get succ. my_rank is %d\r\n", my_rank);
    }
    return 0;
}
```

Sample 4: Register an L3 event. When the event occurs, execute the corresponding callback function by calling **l3_event_msg_callback_reg**.

```
#include <stdio.h>
#include "lwip/l3event.h"
#include "hi_at.h"
static hi_void mesh_lwip_clean_parent_callback(hi_u8 type, hi_void *para)
{
    (hi_void) para;
    hi_at_printf("mesh_lwip_clean_parent_callback event L3_EVENT_PARENT_CLEAN\n");
    return;
}
static hi_void mesh_lwip_rout_change_callback(hi_u8 type, hi_void *para)
{
    (hi_void) para;
    hi_at_printf("mesh_lwip_rout_change_callback event L3_EVENT_ROUTE_CAHNGE\n");
    return;
}
static hi_void mesh_lwip_join_rpl_callback(hi_u8 type, hi_void *para)
{
```



```
    (hi_void) para;
    hi_at_printf("mesh_lwip_join_rpl_callback event L3_EVENT_MSG_RPL_JOIN_SUCC\n");
    return;
}
void l3_event_register_example(void)
{
    /* here we register these three L3_EVENT_MSG, and when L3_EVENT_MSG happend, the
callback function will be executed */
    l3_event_msg_callback_reg(L3_EVENT_MSG_PARENT_CLEAR,
mesh_lwip_clean_parent_callback);
    l3_event_msg_callback_reg(L3_EVENT_MSG_ROUTE_CHANGE,
mesh_lwip_route_change_callback);
    l3_event_msg_callback_reg(L3_EVENT_MSG_RPL_JOIN_SUCC, mesh_lwip_join_rpl_callback);
}
```

Sample 5: Obtain the DHCP client information and free the client.

```
#include "lwip/netif.h"
#include "lwip/netifapi.h"
#include "hi_at.h"
hi_void hi_get_dhcp_clients_example(void)
{
    struct netif *netif = netifapi_netif_find("wlan0");
    if (netif == NULL) {
        return;
    }
    struct dhcp_clients_info *clis_info = NULL;
    if (netifapi_dhcp_clients_info_get(netif, &clis_info) != ERR_OK) {
        hi_at_printf("get dhcp clients fail!\r\n");
        return;
    }
    hi_at_printf("get dhcp clients success!\r\n");
    if (clis_info != HI_NULL) {
        if (netifapi_dhcp_clients_info_free(netif, &clis_info)!= ERR_OK) {
            hi_at_printf("free dhcp clients fail!\r\n");
            return;
        }
    hi_at_printf("free dhcp clients success!\r\n");
    }
}
```


A Terminology

- RPL: IPv6 Routing Protocal for Low-Power and Lossy Networks (LLN).
- RPL instance: a set of one or more DODAGs that share an RPL instance ID.
- DAG: Directed Acyclic Graph, having the property that all edges are oriented in such a way that no cycles exist.
- Destination-Oriented DAG (DODAG): a DAG rooted at a single destination, that is, at a single DAG root (the DODAG root) with no outgoing edges.
- DADAG parent: parent node of a node in the DODAG, that is, the next hop of the node to the root path of the DODAG.
- DIO: DODAG Information Object that carries information that allows a node to discover a RPL Instance, learn its configuration parameters, select a DODAG parent set, and maintain the DODAG to construct an uplink route.
- DAO: Destination Advertisement Object which is used to propagate destination information upward along the DODAG to construct a downlink route.
- DIS: DODAG Information Solicitation which is used to solicit a DIO from an RPL node. A node may use DIS to probe its neighborhood for nearby DODAGs.
- GACK: private message, that is, a message sent from the DODAG root node to the destination node. The message carries acknowledgment information indicating that the DAO message has reached the DODAG root node.

