ESP Thread Border Router SDK₃

Espressif Thread Border Router solution is based on the combination of Espressif's Wi-Fi and 802.15.4 SoCs, built on the <u>ESP-IDF</u> and open-source <u>OpenThread</u> stack.



ESP-Thread-Border-Router Software Components 3

This solution has obtained the Thread 1.3 <u>Certified Component Certificate</u> issued by the Thread Group, complying with the latest Thread 1.3 standard, and supporting Matter application scenario.

It provides the following features:

- Bi-directional IPv6 Connectivity
- Service Discovery
- Multicast Forwarding

- NAT64
- RCP update
- RF coexistence
- Web GUI
- •••

Table of Contents

- <u>1. Hardware Platforms</u>
 - <u>1.1. Wi-Fi based Thread Border Router</u>
 - o <u>1.2. Ethernet based Thread Border Router</u>
 - <u>1.3. Contents and Packaging</u>
 - <u>1.4. Related Documents</u>
- <u>2. Development Guide</u>
 - o 2.1. Build and Run
 - o <u>2.2. Setting up the Local OTA Server</u>
 - <u>2.3. OTA Update Mechanism</u>
- <u>3. ESP Thread Border Router Codelab</u>
 - o <u>3.1. Bi-directional IPv6 Connectivity</u>
 - o 3.2. Multicast Forwarding
 - <u>3.3. Service Discovery</u>
 - o <u>3.4. NAT64</u>
 - o <u>3.5. WEB GUI</u>
- <u>4. API Reference</u>
 - o <u>4.1. RCP Update</u>
 - 4.2. Border Router HTTP OTA

1. Hardware Platforms₃

The Espressif Thread Border Router supports both Wi-Fi and Ethernet interfaces as backbone link.

1.1. Wi-Fi based Thread Border Router<u>3</u>

The Wi-Fi based ESP Thread Border Router consists of two SoCs:

- The host Wi-Fi SoC, which can be ESP32, ESP32-S and ESP32-C series SoC.
- The radio co-processor (RCP), which is an ESP32-H series SoC. The RCP enables the Border Router to access the 802.15.4 physical and MAC layer.

Espressif provides a Border Router board which integrates the host SoC and the RCP into one module.



ESP Thread Border Router/Zigbee Gateway Board $\underline{\mathfrak{S}}$

1.2. Ethernet based Thread Border Router₃

Similar to the previous Wi-Fi based Thread Border Route setup, but a device with Ethernet interface is required.

Espressif provides a Sub-Ethernet daughter board, which works with the ESP Thread Border Router board to extend Ethernet interface.



ESP Thread Border Router/Zigbee Gateway Sub-Ethernet $\underline{\mathfrak{S}}$

1.3. Contents and Packaging₃

Ordering Information 3

The development board has a variety of variants to choose from, as shown in the table below.

Ordering Code	On-board Module	Flash [A]	PSRAM	Description
ESP Thread BR- Zigbee GW	ESP32-S3-WROOM- 1 and ESP32-H2- MINI-1	4 MB	2 MB	ESP Thread Border Router/Zigbee Gateway Board
ESP Thread BR- Zigbee GW_SUB				ESP Thread Border Router/Zigbee Gateway Sub- Ethernet

<u>A</u>

The flash is integrated in the chip's package.

Retail Orders<u>3</u>

If you order one or several samples, each board comes in an individual package in either antistatic bag or any packaging depending on your retailer.

For retail orders, please go to <u>https://www.espressif.com/en/company/contact/buy-a-</u> <u>sample</u>.

Wholesale Orders<u>3</u>

If you order in bulk, the boards come in large cardboard boxes.

For wholesale orders, please go to <u>https://www.espressif.com/en/contact-us/sales-</u> <u>questions</u>.

1.4. Related Documents₃

• <u>ESP Thread Border Router/Zigbee Gateway Board schematic</u> (PDF)

• ESP Thread Border Router/Zigbee Gateway Sub-Ethernet (PDF)

PreviousNext

2. Development Guide₃

2.1. Build and Run₃

This document contains instructions on building the images for ESP Thread Border Router and CLI device and forming a Thread network with the devices.

2.1.1. Set up the Repositories 3

Clone the <u>esp-idf</u> and the <u>esp-thread-br</u> repository.

```
git clone -b v5.1.1 --recursive https://github.com/espressif/esp-idf.git
cd esp-idf
./install.sh
. ./export.sh
cd ..
git clone --recursive https://github.com/espressif/esp-thread-br.git
```

If you are new to ESP-IDF, please follow the <u>ESP-IDF getting started guide</u> to set up the IDF development environment and get familiar with the IDF development tools.

2.1.2. Build the RCP Image₃

Build the esp-idf/examples/openthread/ot_rcp example. The firmware doesn't need to be explicitly flashed to a device. It will be included in the Border Router firmware and flashed to the ESP32-H2 chip upon first boot.

```
cd $IDF_PATH/examples/openthread/ot_rcp
```

Select the ESP32-H2 as the RCP.

```
idf.py set-target esp32h2
```

The default communication interface on the ESP Thread Border Router board is UARTO with 460800 baudrate, which can be configured in <u>esp_ot_config.h</u>.

idf.py menuconfig
idf.py build

2.1.3. Configure ESP Thread Border Router<u>3</u>

Go to the basic_thread_border_router example folder.

cd esp-thread-br/examples/basic_thread_border_router

The default configuration works as is on ESP Thread Border Router board, the default SoC target is ESP32-S3.

To run the example on other SoCs, please configure the SoC target using command:

idf.py set-target <chip_name>

For any other customized settings, you can configure the project in menuconfig.

idf.py menuconfig

2.1.3.1. Wi-Fi based Thread Border Router 3

By default, it is configured as Wi-Fi based Thread Border Router.

The Wi-Fi SSID and password must be set in menuconfig. The corresponding options are Example Connection Configuration -> WiFi SSID and Example Connection Configuration -> WiFi Password.

The auto start mode is enabled by default, the device will connect to the configured Wi-Fi and form Thread network automatically, and then act as the border router.

Note

The following configuration options are all optional, jump to <u>2.1.4. Build and Run the</u> <u>Thread Border Router</u> if you don't need any customized settings.

2.1.3.2. Ethernet based Thread Border Router<u>3</u>

The border router can also be configured to connect to an Ethernet network. In this case, the daughter board ESP Thread Border Router/Zigbee Gateway Sub-Ethernet is required to extend the Ethernet interface.

The following options need to be set:

- Enable EXAMPLE_CONNECT_ETHERNET
- Disable EXAMPLE_CONNECT_WIFI

The configurations of **EXAMPLE_CONNECT_ETHERNET** as following:

Parameter	Value	Note
Туре	W5500 Module	Mandatory
Stack Size	2048	Customized
SPI Host	SPI2	Mandatory
SPI SCLK	GPIO21	Mandatory
SPI MOSI	GPIO45	Mandatory
SPI MISO	GPIO38	Mandatory
SPI CS	GPIO41	Mandatory
SPI Interrupt	GPIO39	Mandatory
SPI SPEED	36 MHz	Customized
PHY Reset	GPIO40	Mandatory
PHY Address	1	Mandatory

The configuration result would look like this.

```
Espressif IoT Development Framework Configuration
[ ] connect using WiFi interface
[*] connect using Ethernet interface
(2048) emac_rx task stack size
        Ethernet Type (W5500 Module) --->
(2) SPI Host Number
(21) SPI SCLK GPIO number
(45) SPI MOSI GPIO number
(38) SPI MISO GPIO number
(41) SPI CS GPIO number
(36) SPI clock speed (MHz)
(39) Interrupt GPIO number
```

```
(40) PHY Reset GPIO number
(1) PHY Address
[*] Obtain IPv6 address
Preferred IPv6 Type (Local Link Address) --->
```

2.1.3.3. Thread Network Parameters 3

The Thread network parameters could be pre-configured with **OPENTHREAD_NETWORK_xx** options.

2.1.3.4. Communication Interface3

The default communication interface between host SoC and RCP is UART.

In order to use the SPI interface instead,

the **OPENTHREAD_RCP_SPI** and **OPENTHREAD_RADIO_SPINEL_SPI** options should be enabled in **ot_rcp** and **basic_thread_border_router** example configurations, respectively. And set corresponding GPIO numbers in *esp_ot_config.h*.

2.1.3.5. Manual Mode<u>3</u>

Disable **OPENTHREAD_BR_AUTO_START** option if you want to setup the network manually. Then the following CLI commands can be used to connect Wi-Fi and form a Thread network:

```
wifi connect -s <ssid> -p <psk>
dataset init new
dataset commit active
ifconfig up
thread start
```

2.1.3.6. RF External coexistence 3

Enable **EXTERNAL_COEX_ENABLE** option if you want to enable the RF External coexistence.

Note

To enable external coexistence of the Thread Border Router, enable the EXTERNAL_COEX_ENABLE option of \$IDF_PATH/examples/openthread/ot_rcp before building the RCP Image.

2.1.4. Build and Run the Thread Border Router

Build and Flash the example to the host SoC.

idf.py -p \${PORT_TO_BR} flash monitor

The following result will be shown in your terminal:

Wi-Fi Border Router:

```
I (555) cpu start: Starting scheduler on PRO CPU.
I (0) cpu start: Starting scheduler on APP CPU.
I (719) example connect: Start example connect.
I (739) wifi:wifi firmware version: 4d93d42
I (899) wifi:mode : sta (84:f7:03:c0:d1:e8)
I (899) wifi:enable tsf
I (899) example connect: Connecting to xxxx...
I (899) example connect: Waiting for IP(s)
I (5719) example connect: Got IPv6 event: Interface "example netif sta" address:
fe80:0000:0000:86f7:03ff:fec0:d1e8, type: ESP_IP6_ADDR_IS_LINK_LOCAL
I (5719) esp_netif_handlers: example_netif_sta ip: 192.168.1.102, mask: 255.255.255.0, gw:
192.168.1.1
I (5729) example connect: Got IPv4 event: Interface "example netif sta" address: 192.168.1.102
I (5739) example common: Connected to example netif sta
I (5749) example common: - IPv4 address: 192.168.1.102,
I (5749) example common: - IPv6 address: fe80:0000:0000:86f7:03ff:fec0:d1e8, type:
ESP IP6 )
I(5779) OPENTHREAD: [I] Platform-----: RCP reset: RESET POWER ON
I(5809) OPENTHREAD: [N] Platform-----: RCP API Version: 6
I (5919) esp ot br: RCP Version in storage: openthread-esp32/8282dca796-e64ba13fa; esp32h2;
2022-10-10 06:01:35 UTC
I (5919) esp ot br: Running RCP Version: openthread-esp32/8282dca796-e64ba13fa; esp32h2;
2022-10-10 06:01:35 UTC
I (5929) OPENTHREAD: OpenThread attached to netif
I(5939) OPENTHREAD: [I] SrpServer----: Selected port 53535
I(5949) OPENTHREAD: [I] NetDataPublshr: Publishing DNS/SRP service unicast (ml-eid, port:53535)
```

Ethernet Border Router:

I (793) cpu_start: Starting scheduler on PRO CPU. I (793) cpu_start: Starting scheduler on APP CPU. I (904) system_api: Base MAC address is not set I (904) system_api: read default base MAC address from EFUSE I (924) esp_eth.netif.netif_glue: 70:b8:f6:12:c5:5b I (924) esp_eth.netif.netif_glue: ethernet attached to netif I (2524) ethernet_connect: Waiting for IP(s). I (2524) ethernet_connect: Ethernet Link Up I (3884) ethernet_connect: Got IPv6 event: Interface "example_netif_eth" address: fe80:0000:0000:72b8:f6ff:fe12:c55b, type: ESP_IP6_ADDR_IS_LINK_LOCAL I (3884) esp_netif_handlers: example_netif_eth ip: 192.168.8.148, mask: 255.255.255.0, gw: 192.168.8.1 I (3894) ethernet_connect: Got IPv4 event: Interface "example_netif_eth" address: 192.168.8.148 I (3904) example_common: Connected to example_netif_eth

```
I (3904) example_common: - IPv4 address: 192.168.8.148,
I (3914) example_common: - IPv6 address: fe80:0000:0000:72b8:f6fI(3944) OPENTHREAD:[I]
Platform-----: RCP reset: RESET_POWER_ON
I(3974) OPENTHREAD:[N] Platform-----: RCP API Version: 6
I(4144) OPENTHREAD:[I] Settings-----: Read NetworkInfo {rloc:0x4400,
extaddr:129f848762f1c578, role:leader, mode:0x0f, version:4, keyseq:0x0, ...
I(4154) OPENTHREAD:[I] Settings-----: ... pid:0x18954426, mlecntr:0x7da7, maccntr:0x7d1c,
mliid:2874d9fa90dc8093}
I (4194) OPENTHREAD: OpenThread attached to netif
```

2.1.5. Build and Run the Thread CLI Device

Build the esp-idf/examples/openthread/ot_cli example and flash the firmware to another ESP32-H2 devkit.

cd \$IDF_PATH/examples/openthread/ot_cli

idf.py -p \${PORT_TO_ESP32_H2} flash monitor

2.1.6. Attach the CLI Device to the Thread Network³

First acquire the Thread network dataset on the Border Router:

dataset active -x

The network data will be printed on the Border Router:

```
> dataset active -x
0e0800000000000000000000000000000300001335060004001fffe00208dead00beef00cafe0708fdfaeb6813db063b05101122
33445566778899aabbccddeeff00030f4f70656e5468726561642d34396436010212340410104810e2315100afd6bc
9215a6bfac530c0402a0f7f8
Done
```

Commit the dataset on the CLI device with the acquired dataset:

Set the network data active on the CLI device:

dataset commit active

Set up the network interface on the CLI device:

ifconfig up

Start the thread network on the CLI device:

thread start

The CLI device will become a child or a router in the Thread network:

```
> dataset set active
0e0800000000010000000300001335060004001 \\ ffe00208 \\ dead00 \\ beef00 \\ cafe0708 \\ fdfaeb6813 \\ db063 \\ b05101122 \\ bd063 \\ bd05101122 \\ bd063 \\ bd06
 33445566778899aabbccddeeff00030f4f70656e5468726561642d34396436010212340410104810e2315100afd6bc
9215a6bfac530c0402a0f7f8
Done
> dataset commit active
Done
> ifconfig up
Done
I (1665530) OPENTHREAD: netif up
 > thread start
I(1667730) OPENTHREAD: [N] Mle-----: Role disabled -> detached
Done
> I(1669240) OPENTHREAD: [N] Mle-----: RLOC16 5800 -> fffe
I(1669590) OPENTHREAD: [N] Mle------ Attempt to attach - attempt 1, AnyPartition
I(1670590) OPENTHREAD: [N] Mle-----: RLOC16 fffe -> 6c01
I(1670590) OPENTHREAD: [N] Mle------ Role detached -> child
```

2.2. Setting up the Local OTA Server³

This document contains instructions on setting up a self-signed HTTPS OTA server on the local host and configuring the Border Router to trust the server.

Generating the Server Certificate3

Create a new directory ota_server_storage to host the server. Then generate the certificate in this directory:

```
mkdir ota_server_storage && cd ota_server_storage
openssl req -x509 -newkey rsa:2048 -keyout ca_key.pem -out ca_cert.pem -days 365 -nodes
Note that when prompted for the Common Name (CN), the name entered shall match the
hostname running the server.
```

The Border Router OTA image will be automatically generated when building the basic_thread_border_router example. Copy it to the previously created directory ota_server_storage:

cp esp-thread-br/examples/basic_thread_border_router/build/br_ota_image ota_server_storage

Rebuilding the Border Router Firmware with the New Certificate³

The certificate server_certs/ca_cert.pem in the basic_thread_border_router example shall be replaced with the generated certificate. Perform a fullclean and build the application again:

cp path/to/ota_server_storage/ca_cert.pem server_certs/ca_cert.pem

idf.py fullclean

idf.py flash monitor

To download the image from the server and run the following command on the Border Router:

ota download https://\${HOST_URL}:8070/br_ota_image

PreviousNext

2.3. OTA Update Mechanism₃

The ESP Thread Border Router supports building the RCP image into the Border Router image. Upon boot the RCP image will be automatically downloaded to the RCP if the Border Router detects the RCP chip failure to boot.

Hardware Prerequisites

To perform OTA update, the following devices are required:

- An ESP Thread Border Router
- A Linux Host machine

In addition to the UART connection to the RCP chip, two extra GPIO pins are required to control the RESET and the BOOT pin of the RCP. For ESP32-H2 the BOOT pin is GPIO8.

Download and Update the Border Router Firmwares

Create a HTTPS server providing the OTA image file, excute the follow command on your Linux Host machine:

The private key and the certificate shall be accpetable for the Border Router. If they are self-signed, make sure to add the public key to the trusted key set of the Border Router.

Now the image can be downloaded on the Border Router:

ota download https://\${HOST_URL}:8070/br_ota_image

- Tips 1: For optimizing the firmware of border router, CONFIG_COMPILER_OPTIMIZATION_SIZE and CONFIG_NEWLIB_NANO _FORMAT are enabled by default.
- Tips 2: If the OTA function is enabled, it is recommended to optimize the ot_rcp firmware size before building the OTA image. Please refer to ot_rcp README for detailed steps.

After downloading the Border Router will reboot and update itself with the new firmware. The RCP will also be updated if the firmware version changes.

The RCP Image Rollback Mechanism³

The RCP image is stored in a configurable SPIFFS partition under a configurable prefix. Prefix *ot_rcp* will be used in as an example.

Two folders *ot_rcp_0* and *ot_rcp_1* will be used to store the current and the backup RCP image. The RCP image will be stored in *ot_rcp_0* upon first boot. An extra key-value pair will be added in the nvs for storing the current RCP image index.

When applying the RCP update, the RCP image with the current index will be downloaded. After the reboot, the Border Router will detect the status of the RCP. If the RCP fails to boot, the backup image will be flashed to the RCP to revert the change.

When downloading the OTA image from the server, the current RCP image will be marked as the backup image and the previous backup image will be overridden. After the download completes, the current image index will be updated.

The OTA Image File Structure³

The OTA image is a single file containing the meta data of the RCP image, the RCP bootloader, the RCP partition table and the firmwares.

The image can be generated with

script basic_thread_border_router/create_ota_image.py. By default this file is called automatically during build and packs the *ot_rcp* example image into the Border Router firmware.

The RCP image header is defined as the following diagram:

0xff	Header size	0
File Type 0	File size	File offset
File Type 1	File size	File offset

•••

The file type is defined as the following table:

0	RCP version
1	RCP flash arguments
2	RCP bootloader
3	RCP partition table
4	RCP firmware
5	Border Router firmware

3. ESP Thread Border Router Codelab₃

3.1. Bi-directional IPv6 Connectivity₃

The ESP Thread Border Router allows the devices on the Wi-Fi/Ethernet and the Thread network to reach each other with IPv6 addresses.

Hardware Prerequisites3

To perform bi-directional IPv6 connectivity, the following devices are required:

- An ESP Thread Border Router
- A Thread CLI device
- A Linux Host machine

The Border Router and the Linux Host machine shall be connected to the same Wi-Fi network and the Thread device shall join the Thread network formed by the Border Router.

Validate the IPv6 Connectivity₃

The Linux Host machine needs to be configured to accept the router advertisements from the Border Router:

Setting <a>accept_ra to 2 allows all RAs to be accepted:

sudo sysctl -w net/ipv6/conf/wlan0/accept_ra=2
Setting accept_ra_rt_info_max_plen to 128 allows all kinds of prefix length in RAs to be
accepted:

sudo sysctl -w net/ipv6/conf/wlan0/accept_ra_rt_info_max_plen=128

A global address will be assigned to the Thread CLI device. The Linux Host machine can reach the Thread device with an address that can be obtained by running the command *ipaddr* on the CLI device.

ipaddr

The command would produce the similar output:

```
> ipaddr
fd66:afad:575f:1:744d:573e:6e60:188a
fd87:8205:4651:27c8:0:ff:fe00:0
fd87:8205:4651:27c8:e65a:3138:745a:df06
fe80:0:0:0:2433:db2e:62c:b2e4
Done
```

The Linux Host reachable address is fd66:afad:575f:1:744d:573e:6e60:188a, you can ping

this address from Linux Host using the following command:

ping fd66:afad:575f:1:744d:573e:6e60:188a

You well get these output:

\$ ping fd66:afad:575f:1:744d:573e:6e60:188a

```
PING fd66:afad:575f:1:744d:573e:6e60:188a(fd66:afad:575f:1:744d:573e:6e60:188a) 56 data bytes
64 bytes from fd66:afad:575f:1:744d:573e:6e60:188a: icmp_seq=1 ttl=254 time=187 ms
64 bytes from fd66:afad:575f:1:744d:573e:6e60:188a: icmp_seq=2 ttl=254 time=167 ms
```

3.2. Multicast Forwarding

Multicast Forwarding allows reaching devices on the Thread and Wi-Fi network in the same multicast group from both sides.

Hardware Prerequisites<u>3</u>

To perform Multicast Forwarding, the following devices are required:

- An ESP Thread Border Router
- A Thread CLI device
- A Linux Host machine

The Border Router and the Linux Host machine shall be connected to the same Wi-Fi network and the Thread device shall join the Thread network formed by the Border Router.

Limits on Multicast Groups

Note that to forward packets between the Thread and the Wi-Fi network, the multicast group scope shall be at least admin-local(ff04). Link-local and realm-local multicast packets will not be forwarded.

Reaching the Multicast Group from the Wi-Fi Network via ICMP³

First, join the multicast group on the Thread CLI device:

mcast join ff04::123

Now you can ping Thread CLI device on your Linux Host:

ping -I wlan0 -t 64 ff04::123

The output similar to shown below may indicate that the device can be reached on the Wi-Fi network via the multicast group:

\$ ping -I wlan0 -t 64 ff04::123
PING ff04::123(ff04::123) from fdde:ad00:beef:cafe:2eea:7fff:fe37:b4fb wlan0: 56 data bytes
64 bytes from fd66:afad:575f:1:744d:573e:6e60:188a: icmp_seq=1 ttl=254 time=132 ms

Reaching the Multicast Group from the Wi-Fi Network via UDP³

First, join the multicast group and create a UDP socket on the Thread CLI device:

```
> mcast join ff04::123
Done
> udp open
Done
> udp bind :: 5083
Done
```

Use the following python script, which is named multicast_udp_client.py, to send UDP

messages to the Thread CLI device via the multicast group:

```
import socket
import time
data = b'hello'
group = 'ff04::123'
REMOTE_PORT = 5083
network_interface = 'wlan0' # Change to the actual interface name
sock = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_BINDTODEVICE, network_interface.encode())
sock.setsockopt(socket.IPPROT0_IPV6, socket.IPV6_MULTICAST_HOPS, 32)
sock.sendto(data, (group, REMOTE_PORT))
```

On the Linux Host machine, you need to start a UDP client by running this script:

python3 multicast_udp_client.py

sock.close()

Network₃

On the Thread CLI device, the message will be printed:

```
> mcast join ff04::123
Done
> udp open
Done
> udp bind :: 5083
Done
5 bytes from fd11:1111:1122:2222:4a9e:272e:6a50:cf61 56024 hello
Reaching the Multicast Group from the Thread
```

Use the following python script, which is named multicast_udp_server.py, to join an admin-local group and set up a UDP server on the Linux machine:

On the Linux Host machine, you need to start a UDP server by running this script:

```
python3 multicast_udp_server.py
```

After launching the script and the Thread CLI device will be able to send UDP messages to the Linux Host via the multicast group:

udp open udp send ff04::123 5083 hello

You will get a result **Done** after executing each of the commands, the expected output is

below:

> udp open Done > udp send ff04::123 5083 hello Done

On the Linux Host machine, the message will be printed:

\$ python3 multicast_udp_server.py b'hello' ('fd66:afad:575f:1:744d:573e:6e60:188a', 49154, 0, 0)

3.3. Service Discovery₃

The ESP Thread Border Router allows devices on the Thread and Wi-Fi/Ethernet network to discover services published on both the networks.

Hardware Prerequisites

To perform service discovery, the following devices are required:

- An ESP Thread Border Router
- A Thread CLI device
- A Linux Host machine

The Border Router and the Linux Host machine shall be connected to the same Wi-Fi network and the Thread device shall join the Thread network formed by the Border Router.

Publishing Services from the Thread Network³

Thread uses the Service Registration Protocol (SRP) to register services. First, publish the service on the Thread CLI device.

Using this command line to set host name:

srp client host name thread-device
Using this command line to set host address, you can set default address with auto:

srp client host address auto

Using this command line to set parameter of the service:

srp client service add thread-service _test._tcp,_sub1,_sub2 12345 1 1 0778797a3d58595a
Using this command line to enable the service:

srp client autostart enable

You will get a result **Done** after executing each of the commands, the expected output is below:

```
> srp client host name thread-device
Done
> srp client host address auto
Done
> srp client service add thread-service _test._tcp,_sub1,_sub2 12345 1 1 0778797a3d58595a
Done
> srp client autostart enable
Done
```

You can execute this command on the Border Router to resolve the service:

srp server service

The service can be found on the Border Router:

> srp server service

```
thread-service._test._tcp.default.service.arpa.
    deleted: false
    subtypes: _sub2,_sub1
    port: 12345
    priority: 1
    weight: 1
    ttl: 7200
    TXT: [xyz=58595a]
    host: thread-device.default.service.arpa.
    addresses: [fd66:afad:575f:1:744d:573e:6e60:188a]
Done
```

The service can also be found on the Wi-Fi with the Linux Host machine by using this command:

avahi-browse -rt _test._tcp

The expected output is below:

\$	avahi-browse -rt _testtcp		
+	enp1s0 IPv6 thread-service	_testtcp	local
+	enp1s0 IPv4 thread-service	_testtcp	local
=	enp1s0 IPv6 thread-service	_testtcp	local
	<pre>hostname = [thread-device.local]</pre>		
	address = [fd66:afad:575f:1:744d:573e:6e60:188a]		
	port = [12345]		
	<pre>txt = ["xyz=XYZ"]</pre>		
=	enp1s0 IPv4 thread-service	_testtcp	local
	<pre>hostname = [thread-device.local]</pre>		
	address = [fd66:afad:575f:1:744d:573e:6e60:188a]		
	port = [12345]		
	txt = ["xyz=XYZ"]		

Publishing Services from the Wi-Fi Network³

First publish the service on the Linux Host machine with mDNS:

avahi-publish-service wifi-service _test._tcp 22222 test=1 dn="aabbbb"

If the service is established, you will get this output on your Linux Host machine:

\$ avahi-publish-service wifi-service _test._tcp 22222 test=1 dn="aabbbb"
Established under name 'wifi-service'

Then get the Border Router's Mesh-Local Endpoint Identifier, and configure it on the Thread end device. On the Border Router:

ipaddr mleid

You will get:

```
> ipaddr mleid
fdde:ad00:beef:0:f891:287:866:776
Done
```

On the Thread CLI device:

```
dns config fdde:ad00:beef:0:f891:287:866:776
```

You will get:

```
> dns config fd9b:347f:93f7:1:1003:8f00:bcc1:3038
Done
```

The service can be resolved on the Thread CLI device by executing this command:

```
dns service wifi-service _test._tcp.default.service.arpa.
```

The expected output on the Thread CLI device is below:

```
> dns config fdde:ad00:beef:0:f891:287:866:776
Done
> dns service wifi-service _test._tcp.default.service.arpa.
DNS service resolution response for wifi-service for service _test._tcp.default.service.arpa.
Port:22222, Priority:0, Weight:0, TTL:120
Host:FA001388.default.service.arpa.
HostAddress:fd33:1cc4:a6ec:2e0:2eea:7fff:fe37:b4fb TTL:120
TXT:[test=31, dn=616162626262] TTL:120
```

Done

3.4. NAT64<u>3</u>

The ESP Thread Border Router supports NAT64 which allows Thread devices to visit the IPv4 Internet.

Hardware Prerequisites

To perform NAT64, the following devices are required:

- An ESP Thread Border Router
- A Thread CLI device

The Thread device shall join the Thread network formed by the Border Router.

Visiting the IPv4 HTTP Servers³

For visiting HTTP servers with domain names, the DNS server shall be first configured on the Thread CLI device:

dns64server 8.8.8.8

Then you can use curl <website> command to get the data form the specific website(for
example http://www.espressif.com):

curl http://www.espressif.com

The Thread device will first resolve the host with UDP packets sent to the IPv4 DNS server. Then retrieve the page from the IPv4 HTTP server via TCP. The expected output is below:

```
> dns64server 8.8.8.8
Done
> curl http://www.espressif.com
Done
> I (22289) HTTP_CLIENT: Body received in fetch header state, 0x3fcca6b7, 183
<html>
<head><title>301 Moved Permanently</title></head>
<body bgcolor="white">
<center><h1>301 Moved Permanently</h1></center>
<hr><center><loudFront</center>
</body>
</html>
```

3.5. WEB GUI₃

The ESP Thread Border Router is equipped with a user-friendly graphical user interface (GUI) that enables users to easily discover, configure, and monitor Thread networks through the web server.

To access the Web GUI, simply enter the local IPv4 address for the ESP Thread Border Router in your browser window with port 80 and the path index.html.



Prerequisites³

To perform web gui, the follow device are required:

- An ESP Thread Border Router
- A Linux host machine with browser

Enable the **CONFIG_OPENTHREAD_BR_START_SERVER** option to enable the Web Server feature.

The Thread Border Router and the Linux Host machine shall be connected to the same Wi-Fi network that has access to the Internet.

When the ESP Thread Border Router starts up, it will print the website's access address to terminal of the Linux host.

such as:

otbr_web:	<=====server	start======>
otbr_web:	http://192.168.200.98:80/index.	html
otbr_web:	<======================================	>

All REST APIs can be accessed by visiting the IPv4 address of the Thread-Border-Router-Board using the HTTP on port 80 with the API field specified.

Thread REST APIs₃

The ESP Thread Border Router server provides the REST APIs that are compatible with <u>ot-br-posix</u>

```
The Thread REST APIs field
include /diagnostics, /node, /node/rloc, /node/rloc16, /node/ext-
address, /node/state, /node/network-name, /node/leader-data, /node/num-of-router, /node/ext-
panid and /node/active-dataset-tlvs.
```

Entering this link to the browser of Linux machine:

```
http://192.168.200.98:80/node
```

The following feedback result will display on the browser:

```
{
    "NetworkName": "OpenThread-4c68",
    "ExtPanId": "f4f9437404558d34",
    "ExtAddress": "caf97e6ee990b047",
    "RlocAddress": "fd12:cb40:859f:287e:0:ff:fe00:3800",
    "LeaderData": {
        "PartitionId": 61563841,
        "Weighting": 64,
        "DataVersion": 211,
        "StableDataVersion": 110,
        "LeaderRouterId": 14
    },
    "State": 4,
    "Rloc16": 14336,
    "NumOfRouter": 1
}
```

The access method for other APIs is similar to the one described above.

Web GUI REST APIs₃

The web server of ESP Thread Border Router provides the avaiable_network API to discover the all available Thread networks.

Entering this link to the browser of Linux machine:

http://192.168.200.98:80/available_network

The feedback result may appear as follows:

```
{
    "error":
                    0,
    "result":
                    [{
            "id":
                    1,
            "nn":
                    "OpenThread",
            "ep":
                    "dead00beef00cafe",
            "pi":
                    "0xa06d",
                    "5a1ee78f873814fc",
            "ha":
            "ch":
                    11,
            "ri":
                    -35,
            "li":
                    229
       }, {
"id":
":
                    2,
            "nn":
                    "GRL",
            "ep":
                    "000db8000000000",
            "pi":
                    "0xfacf",
            "ha":
                    "166e0a000000003",
            "ch":
                    17,
            "ri":
                    -70,
            "li":
                    51
       }, {
    "id":
    "."
                    3,
            "nn": "NEST-PAN-3DDF",
            "ep": "4500ddd4f9c1597d",
            "pi": "0x3ddf",
            "ha":
                    "9e517ed148e81409",
            "ch":
                    20,
            "ri":
                    -39,
            "li": 209
        }],
    "message":
                    "Networks: Success"
}
```

The web server of ESP Thread Border Router provides the get_properties API to check the Thread network status.

Entering this link to the browser of Linux machine:

http://192.168.200.98:80/get_properties

The feedback result may appear as follows:

```
{
    "error": 0,
    "result": {
        "IPv6:LinkLocalAddress": "fe80:0:0:0:c8f9:7e6e:e990:b047",
        "IPv6:RoutingLocalAddress": "fd12:cb40:859f:287e:0:ff:fe00:3800",
        "IPv6:MeshLocalAddress": "fd12:cb40:859f:287e:a8b5:c617:396b:a4c2",
```

```
"IPv6:MeshLocalPrefix": "fd12:cb40:859f:287e::/64",
        "Network:Name": "OpenThread-4c68",
        "Network:PANID": "0x1254",
"Network:PartitionID": "61563841",
        "Network:XPANID": "f4f9437404558d34",
        "OpenThread:Version": "openthread-esp32/f4446d8819-091f68ed7; esp32s3; 2023-05-
05 13:05:02 UTC",
        "OpenThread:Version API":
                                       "292",
        "RCP:State": "leader",
        "OpenThread:PSKc": "e66d93364793c33985280abb639c214c",
        "RCP:Channel": "12",
"RCP:EUI64": "6055f9f72eebfeff",
"RCP:TxPower": "10 dBm",
"RCP:Version": "openthread-esp32/f4446d8819-091f68ed7; esp32h2; 2023-05-04
08:35:37 UTC",
        "WPAN service": "associated"
    },
                     "Properties: Success"
    "message":
}
```

The web server of ESP Thread Border Router provides the node_information API to otbain the Thread node information.

Entering this link to the browser of Linux machine:

```
http://192.168.200.98:80/node_information
```

The feedback result may appear as follows:

```
{
   "error":
                  0,
   "result":
                 {
       "NetworkName": "OpenThread-4c68",
       "ExtPanId": "f4f9437404558d34",
       "ExtAddress": "caf97e6ee990b047",
                         "fd12:cb40:859f:287e:0:ff:fe00:3800",
       "RlocAddress":
       "LeaderData":
                          {
           "PartitionId": 61563841,
          "Weighting": 64,
          "DataVersion": 225,
           "StableDataVersion":
                               124,
          "LeaderRouterId":
                                14
       },
       "State":
                4,
       "Rloc16": 14336,
       "NumOfRouter":
                         1
   },
   "message": "Get Node: Success"
}
```

The web server of ESP Thread Border Router provides the topology API to retrieve information about the relationship between Thread networks.

Entering this link to the browser of Linux machine:

http://192.168.200.98:80/topology

The feedback result may appear as follows:

```
{
                     "error":
                                                     0,
          "result":
                                        [{
                                "ExtAddress":
                                                                       "caf97e6ee990b047",
                                "Rloc16":
                                                                           14336,
                                 "Mode": {
                                           "RxOnWhenIdle":
                                                                                             1,
                                           "RxUnwhenze":
"DeviceType":
                                                                                                 1,
                                           "NetworkData":
                                                                                             1
                                "ParentPriority": 0,
                                          "LinkQuality3": 0,
"LinkQuality2": 0,
"LinkQuality1": 0,
"LeaderCost": 0,
"IdSequence": 131,
                                           "ActiveRouters": 1,
"SedBufferSize": 1280,
                                           "SedDatagramCount": 1
                               },
"Route":
                                                                           {
                                           "IdSequence":
                                                                                                 131,
                                           "RouteData":
                                                                                                 [{
                                                                 "RouteId": 14,
                                                                 "LinkQualityOut": 0,
                                                                 "LinkQualityIn": 0,
"LinkQualityIn": 1
                                                                                                                      0,
                                                      }]
                                 },
                                   LeaderData": {
                                           "PartitionId":
                                                                                             61563841,
                                           "Weighting":
                                                                                                64,
                                                                                           229,
                                           "DataVersion":
                                           "StableDataVersion":
                                                                                                                      128,
                                           "LeaderRouterId": 14
                                },
                                 "NetworkData":
"08040b02cca60b0e8001010d09380000000500000e1003140040fd634dc9496e000105043800f10007021140030f0
040 f df 4 f 94374048 d 340103380000 b 1981015 d 0 d 143800 f d 12 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f d 6 c b 40859 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f 4 c 4 1730060 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f 4 c 4 1730060 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f 4 a 8000 f 287 e a 8b 5 c 6 17396 b a 4 c 2 d 11 f 0 3130060 f 4 a 8000 f 4 a
34dc9496e0002000000001033800e0",
                                "IP6AddressList":
                                                                                       "fd12:cb40:859f:287e:0:ff:fe00:fc11",
                                                                                       "fd63:4dc9:496e:1:9967:1ba3:5fbf:f2e6",
                                                                                       "fd12:cb40:859f:287e:0:ff:fe00:fc10",
                                                                                       "fd12:cb40:859f:287e:0:ff:fe00:fc38",
                                                                                       "fd12:cb40:859f:287e:0:ff:fe00:fc00",
                                                                                       "fd12:cb40:859f:287e:0:ff:fe00:3800",
                                                                                       "fd12:cb40:859f:287e:a8b5:c617:396b:a4c2",
                                                                                       "fe80:0:0:0:c8f9:7e6e:e990:b047"
                                                                                       ],
                                 "MACCounters": {
                                           "IfInUnknownProtos":
                                                                                                                       0,
                                           "IfInErrors": 0,
                                                                                                 0,
                                           "IfOutErrors":
                                           "IfInUcastPkts": 13,
```

```
"IfInBroadcastPkts": 56,
"IfInDiscards": 0,
"IfOutUcastPkts": 0,
"IfOutBroadcastPkts": 201,
"IfOutDiscards": 0
},
"ChildTable": [],
"ChannelPages": "00"
}],
"message": "Topology: Success"
}
```

The web server provides an HTTP_POST entry that allows users to configure the Border Router to use either networkKeyType or pskdType for joining other networks.

The JSON format of join_network API appears as follow:

```
{
    "credentialType": "networkKeyType",
    "networkKey" : "00112233445566778899aabbccddeeff",
    "pskd" : "12345678",
    "prefix" : "fd11:22::",
    "defaultRoute" : 1,
    "index" : 1
}
```

Note that the network to be joined MUST be the networks scanned by the available_network API, the index indicates the sequence of available networks.

The web server provides an HTTP_POST entry that allows users to configure the Border Router to use the parameter provided by user for forming a Thread network.

The JSON format of **form_network** API appears as follow:

```
{
    "networkName" : "OpenThread-0x99",
    "networkKey" : "00112233445566778899aabbccddeeff",
    "panId" : "0x1234",
    "channel" : 16,
    "extPanId" : "111111122222222",
    "passphrase" : "j01Nme",
    "prefix" : "fd11:22::",
    "defaultRoute" : 1
}
```

The web server provides an HTTP_POST entry that allows users to configure the Border Router for setting current Thread network.

The JSON format of add_prefix API appears as follow:

```
{
    "prefix": "fd11:22::",
    "defaultRoute": 1
}
```

The JSON format of delete_prefix API appears as follow:



Web GUI Application Introduction3

ESP Thread Border Router Web GUI provides practical functions including Thread network discovery, network formation, network settings, status query and network.

Discover<u>3</u>

By clicking the scan button, you can discover for the available Thread networks. The networks will be shown in the table with their network name, channel, extended panid, panid, Mac address, txpower and so on.

			Discove	er Thread N	etworl	K		
			nere, you can scan for		to join one of them.			
Availab	Network Name	S: Scan Complete	PANID	Mac Address	Channel	dBm	1.01	Action
1	OpenThread-960e	949fcd4261ec8a01	0x1254	ae601ba903bfb4f6	12	-19	255	Join
2	OpenThread-960e	949fcd4261ec8a01	0x1254	3a18494adc1a95ae	12	-28	255	Join
3	OpenThread-e86d	dada001234004321	0x198f	1212121212121212	21	-30	255	Join
4	OpenThread-540d	dead00beef00cafe	0x1234	8abc033497b2eb5e	22	-48	163	Join
5	OpenThread-540d	dead00beef00cafe	0x1234	728be6d01cfa6aed	22	-42	193	Join
6	MyHome2087051184	7b5585df74d9435d	0x8df9	360aaf51cf278eae	25	-55	127	Join
7	MyHome2087051184	7b5585df74d9435d	0x8df9	f2a2a7bf31419bfb	25	-49	158	Join
8	MyHome2087051184	7b5585df74d9435d	0x8df9	56fd960a233d94bb	25	-44	183	Join
9	OpenThread-9ac0	17afaebbaca636b2	0x9ac0	065d5267c2d15c69	26	-56	122	Join
10	OpenThread-9ac0	17afaebbaca636b2	0x9ac0	fa6ea1ab3769e89a	26	-56	122	Join
11	OpenThread-9ac0	17afaebbaca636b2	0x9ac0	3ee98a564217a739	26	-58	112	Join
								SCAN

Join<u>3</u>

You can select an available network to join by clicking the join button. Enter the relevant information into the pop-up dialog, submit it, and the result will be displayed for you after a moment.

Availat	ole Thread Network	s: Scan Comple	Join Informat Credential Type: Network Key:	ion		×		
No.	Network Name	Extended PAN ID	1234567890abedef1234	567890abedef		Bm	LQI	Action
1	OpenThread 960a	949fcd4261oc9p01	Prefix:				255	
1	Open meau-sobe	5451004201600001	fd11:22::				200	Join
2	OpenThread-960e	949fcd4261ec8a01	Default Route: 🗹	_		8	255	Join
3	OpenThread-e86d	dada001234004321			Cancel Su	bmit ₀	255	Join
4	OpenThread-540d	dead00beef00cafe	0x1234	8abc033497b2eb5e	22	-48	163	Join
5	OpenThread-540d	dead00beef00cafe	0x1234	728be6d01cfa6aed	22	-42	193	Join
6	MyHome2087051184	7b5585df74d9435d	0x8df9	360aaf51cf278eae	25	-55	127	Join
7	MyHome2087051184	7b5585df74d9435d	0x8df9	f2a2a7bf31419bfb	25	-49	158	Join
8	MyHome2087051184	7b5585df74d9435d	0x8df9	56fd960a233d94bb	25	-44	183	Join
9	OpenThread-9ac0	17afaebbaca636b2	0x9ac0	065d5267c2d15c69	26	-56	122	Join
10	OpenThread-9ac0	17afaebbaca636b2	0x9ac0	fa6ea1ab3769e89a	26	-56	122	Join
	OpenThread-9ac0	17afaebbaca636b2	0x9ac0	3ee98a564217a739	26	-58	112	loip

Form<u></u>3

You can form a Thread network in this section. First, you need to fill network's parameters in the following table. Then click the Form Network button to submit the message. The server will validate the network information and form the network on success.

Settings<u>s</u>

The IPv6 network prefix for Thread can be set in the Settings section. To add it, click Add, and to delete it, click Delete.

	Inread Network Settings	
	Here, setting the default options for the ESP-OpenThread network can make your Thread device more flexible.	
ESP-OpenThread Settings.		
On-Mesh Prefix		
fd11:22::		
Default Route		
Add Delete		
	Delete Prefix: Success	

Status<u>3</u>

By clicking the **OverView** bar, the properties of Thread network will been displayed in the corresponding section.

The statu	s of the ESP-OpenThread network, include	ad Network Status	S	
verView IPv6 Network Open	Thread RCP WPAN	[OpenThread Parameter]	[RCP Information]	[WPAN Status]
Ink Local Address 80:0:0:0:c8f9:7e6e:e990:b047 outing Local Address 25:dd2d:725b:8ca6:0:ff:fe00:7400 lesh Local Address 25:dd2d:725b:8ca6:a8b5:c617:396b;a4c2 lesh Local Prefix Particle 20:c90	Name OpenThread-b732 PANID 0x1234 Partition ID 1278166863 Extended PANID	Version openthread-esp32/f4446d8819-091f68ed7; esp32s3; 2023-05-04 11:45:45 UTC Version API 292 Role leader	Channel 15 IEEE EUI-64 6055f9f72eebfeff TxPower 10 dBm Version	Service associated
(dd2d:7250:86ab::/64	αεάσυσεετυθζατε	PSKC 104810e2315100afd6bc9215a6bfac53	opentnread-esp32/14446d8819- 091f68ed7; esp32h2; 2023-05- 04 08:35:37 UTC	

Topology<u></u>

By clicking the **Start Topology** button, the topology of the current Thread node will be intuitively drawn and displayed.

Thread Network Topology The network topology structure of Thread will be shown in a more intuitive and detailed manner here.							
Network Name: OpenThread-960e	Leader: 0x0	Router Number: 3	Reload Topology				
			 Selected Router Child 				

4. API Reference<u>3</u>

For manipulating the Thread network, the OpenThread API shall be used. The OpenThread API documentation can be found at the <u>OpenThread official website</u>.

For interacting with the ESP OpenThread port and Border Router library, the espopenthread API shall be used. The esp-openthread API docs can be found at <u>ESP-IDF</u> <u>API reference page</u>.

The additional APIs are listed below:

4.1. RCP Update₃

The RCP update component updates the RCP from a local file on the host.

API reference<u>s</u>

Header File<u>3</u>

• <u>components/esp_rcp_update/include/esp_rcp_update.h</u>

Functions3

esp_err_t esp_rcp_update_init(const esp_rcp_update_config_t *update_config)3

This function initializes the RCP update process.

Parameters update config – **[in]** The RCP update specific config

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_ARG If the RCP type is not supported.

esp_err_t esp_rcp_update(void)3

This function triggers an RCP firmware update.

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_STASTE If the RCP update is not initialized.
- ESP_ERR_NOT_FOUND RCP firmware not found in storage.

const char *esp_rcp_get_firmware_dir(void)<u>3</u>

This function acquires the RCP image base directory.

Note

The real RCP image directory should be suffixed the update sequence.

int8_t esp_rcp_get_update_seq(void)3

This function retrieves the update image sequence.

The current update image sequence will be used to update the RCP.

int8_t esp_rcp_get_next_update_seq(void)3

This function retrieves the next update image sequence.

The next update image sequence will be used for the downloaded image.

void esp_rcp_reset(void)3

This function resets the RCP.

esp_err_t esp_rcp_submit_new_image(void)3

This function marks the downloaded image as valid.

The image in the next update image sequence will then be used for RCP update.

Returns

- ESP_OK
- ESP_ERR_INVALID_STASTE If the RCP update is not initialized.

esp_err_t esp_rcp_mark_image_verified(bool verified)3

This function marks previously downloaded image as valid.

Returns

- ESP_OK
- ESP_ERR_INVALID_STASTE If the RCP update is not initialized.

esp_err_t esp_rcp_mark_image_unusable(void)

This function marks previously downloaded image as unusable.

Returns

- ESP_OK
- ESP_ERR_INVALID_STASTE If the RCP update is not initialized.

esp_err_t esp_rcp_load_version_in_storage(char *version_str, size_t size)<u>3</u>

This function loads the RCP version in the current update image.

Parameters

- version_str [out] The RCP version string output.
- size [in] Size of version_str.

Returns

- ESP_OK
- ESP_ERR_INVALID_STASTE If the RCP update is not initialized.
- ESP_ERR_NOT_FOUND RCP version not found in update image.

void esp_rcp_update_deinit(void)3

This function deinitializes the RCP update process.

Structures 3

struct esp_rcp_update_config_t3

The RCP update config for OpenThread.

Public Members

esp_rcp_type_t rcp_type<u>3</u>

RCP type

int uart_rx_pin<u></u> ວັ

UART rx pin

int uart_tx_pin<u></u>3

UART tx pin

int uart_port<u>3</u>

UART port

int uart_baudrate<u>3</u>

UART baudrate

int reset_pin \mathfrak{I}

RESET pin

int boot_pin<u></u>ຽ

Boot mode select pin

uint32_t update_baudrate \mathfrak{S}

Baudrate when flashing the firmware

char firmware_dir[RCP_FIRMWARE_DIR_SIZE]3

The directory storing the RCP firmware

target_chip_t target_chip $\underline{\Im}$

The target chip type

Macros<u>3</u>

RCP_FIRMWARE_DIR_SIZE<u></u> RCP_FILENAME_MAX_SIZE<u></u> RCP_URL_MAX_SIZE<u></u> **Enumerations**

enum esp_rcp_type_t<u>3</u>

Values:

enumerator RCP_TYPE_INVALID<u>S</u> enumerator RCP_TYPE_ESP32H2_UART<u>S</u>

4.2. Border Router HTTP OTA₃

The ESP Thread Border Router HTTP OTA component provides helper functions to download firmware from the web server.

API reference<u>s</u>

Header File<u>3</u>

• <u>components/esp_br_http_ota/include/esp_br_http_ota.h</u>

Functions₃

esp_err_t esp_br_http_ota(esp_http_client_config_t *http_config)3

This function performs Border Router OTA by downloading from a HTTPS server.

Parameters

http_config - [in] The HTTP server download config

Returns

- ESP_OK
- ESP_FAIL
- ESP_ERR_INVALID_STASTE If the RCP update is not initialized.
- ESP_ERR_INVALID_ARG If the http config is NULL or does not contain an url.

Macros<u>3</u>

OTA_MAX_WRITE_SIZE