Table of Contents

1. Openthread 2
1.1. Creating a Thread Network with the ESP Thread Border Router and ESP32H2 2
1.1.1. Example programs 2
1.1.2. Forming a mesh network 2
1.2. Connecting the STM32WB55 to the Thread Network 3
1.2.1. Flashing the necessary firmware 3
1.2.2. Example programs 3
1.2.3. Joining the network 3
1.3. Sending messages between Thread Nodes via CLI 4
1.4. Code samples 4
1.4.1. Reacting to device role changes 4
1.4.2. Sending and receiving UDP messages 5

1. Openthread

1.1. Creating a Thread Network with the ESP Thread Border Router and ESP32H2

The following guide describes the necessary steps:
https://openthread.io/codelabs/esp-openthread-hardware#8

1.1.1. Example programs

The example programs allow to configure and operate devices via the command line interface (CLI).

For the ESP Thread Border Router, two applications should be built: ot_rcp (device target - esp32h2) and basic_thread_border_router (device target - esp32s3).
For the ESP32H2, only one application should be built: ot_cli (device target - esp32h2).

Before building the software, the menuconfig shoud be used to:
- enable the commissioner feature for ESP Thread Border Router (basic_thread_border_router)
Component config > OpenThread > Enable Commissioner
- enable the joiner feature for ESP32H2
Component config > OpenThread > Enable Joiner

1.1.2. Forming a mesh network

We create a dataset on the ESP Thread Border Router using the following commands:

```
dataset init new
dataset commit active
```

The entire dataset can be printed using the `dataset` command. It is possible to print individual parts by typing their names in the console (for example: `networkkey`, `pskc`, `panid`, `eui64`).

In the next step, we bring up the IPv6 interface and the Thread protocol:

```
ifconfig up
thread start
```

After a while, the device should change its status to “leader”. We can use the `state` command to check it.

In order to connect the ESP32H2 to the network, we can use the network key from the dataset generated on the ESP Thread Border Router:

```
dataset networkkey <networkkey>
```

```
dataset commit active
```

After that, we bringe up the IPv6 interface and the Thread protocol with `ifconfig up` and `thread start`. After a while, the device should change its status to “child” and then automatically to “router”.

1.2. Connecting the STM32WB55 to the Thread Network

1.2.1. Flashing the necessary firmware

The appropriate binary file (stm32wb5x_Thread_FTD_fw.bin) must be flashed for the Wireless Coprocessor using a tool like the STM32CubeProgrammer.
It is available in github repository at the following link:
https://github.com/STMicroelectronics/STM32CubeWB/blob/master/Projects/STM32WB_Copro_Wireless_Binaries/STM32WB5x/stm32wb5x_Thread_FTD_fw.bin
The detailed procedure to change the Wireless Coprocessor binary (including install addresses) is available in the same repository:
https://github.com/STMicroelectronics/STM32CubeWB/blob/master/Projects/STM32WB_Copro_Wireless_Binaries/STM32WB5x/Release_Notes.html

1.2.2. Example programs

For the STM32WB55, the Thread_CLI_Cmd application can be found in the STM example programs and then built and flashed.

To establish a serial connection and send CLI commands, it is necessary to use an external application (e.g. PuTTY).

1.2.3. Joining the network

In addition to using the network key, it is possible to use the secure commissioning procedure to connect to the Thread network.

To connect the STM32WB55 to the network using commissioning we must enter the pskc and panid generated on the ESP Thread Border Router and bring up the IPv6 interface.

```
dataset pskc <pskc>
dataset panid <panid>
dataset commit active
ifconfig up
```

On the ESP Thread Border Router, we have to start the commissioner and specify the eui64 of the device that we want to join, along with the joiner credential:

```
commissioner start
```

```
commissioner joiner add <eui64> <joiner credential>
```

The Joiner Credential is a device-specific string of all uppercase alphanumeric characters (0-9 and A-Y, excluding I, O, Q and Z for readability), with a length between 6 and 32 characters.

On the STM32WB55, we start the joiner role with the previously set joiner credential:

```
joiner start <joiner credential>
```

After a while, we should get confirmation of a successful authentication.

1.3. Sending messages between Thread Nodes via CLI

The `ipaddr` command can be used to print the IPv6 address of the device on which the program is running.
To ping the selected node we use the appropriate command and Mesh-Local EID of the board we want to ping: `ping <ML-EID>`.

In order to send a message between two Thread devices with UDP we have to open udp and bind it on selected port on one board:

```
udp open
udp bind :: <port>
```

Then send a message from the second board to MLE-ID address of the first board and previously selected port:

```
udp open
udp send <ML-EID> <port> <message>
```

1.4. Code samples

The following guide covers the basics of using the OpenThread API:
https://openthread.io/codelabs/openthread-apis#0
The official documentation can be found at the following link: https://openthread.io/reference

To make OpenThread API calls, it is necessary to initialize a static instance of the OpenThread library. In the ESP32H2 example, such an instance is returned by the `esp_openthread_get_instance()` function. In the STM32WB55 example, by default the OpenThread instance is initialized in the `APP_THREAD_DeviceConfig(void)` function.

1.4.1. Reacting to device role changes

If we want to react whenever OpenThread instance state changes, we need to add following callback registration to our code:

```
otSetStateChangedCallback(esp_openthread_get_instance(),
handleNetifStateChanged, esp_openthread_get_instance());
```

We also need to define a function that will be called after each state change:

```
void handleNetifStateChanged(uint32_t aFlags, void *aContext)
{
   if ((aFlags & OT_CHANGED_THREAD_ROLE) != 0)
   {
       otDeviceRole changedRole = otThreadGetDeviceRole(aContext);

       switch (changedRole)
       {
       case OT_DEVICE_ROLE_LEADER:
       case OT_DEVICE_ROLE_ROUTER:
           gpio_set_level(GREEN_LED, 1);
           gpio_set_level(YELLOW_LED, 0);
           gpio_set_level(RED_LED, 0);
           break;

       case OT_DEVICE_ROLE_CHILD:
           gpio_set_level(GREEN_LED, 0);
           gpio_set_level(YELLOW_LED, 1);
           gpio_set_level(RED_LED, 0);
           break;

       case OT_DEVICE_ROLE_DETACHED:
       case OT_DEVICE_ROLE_DISABLED:
           gpio_set_level(GREEN_LED, 0);
           gpio_set_level(YELLOW_LED, 0);
           gpio_set_level(RED_LED, 1);
           break;
        }
    }
}
```

The code above was written for the ESP32H2 platform, but in the Thread_CLI_Cmd example on STM32WB55 there exist similar functionality by default.

1.4.2. Sending and receiving UDP messages

When using UDP, we can define the port we want to use and also declare static variables such as socket, address or message payload.

```
#define UDP_PORT    1212

static otUdpSocket sUdpSocket;
static const char UDP_DEST_ADDR[] = "ff03::1";
static const char UDP_PAYLOAD[]   = "Hello World!";
```

The following function can be used to initialize UDP:

```
void initUdp(otInstance *aInstance)
{
    otSockAddr  listenSockAddr;

    memset(&sUdpSocket, 0, sizeof(sUdpSocket));
    memset(&listenSockAddr, 0, sizeof(listenSockAddr));

    listenSockAddr.mPort    = UDP_PORT;

    otUdpOpen(aInstance, &sUdpSocket, handleUdpReceive, aInstance);
    otUdpBind(aInstance, &sUdpSocket, &listenSockAddr, OT_NETIF_THREAD);
}
```

We also define a function that will be called after receiving a message:

```
void handleUdpReceive(void *aContext, otMessage *aMessage, const
otMessageInfo *aMessageInfo)
{
    OT_UNUSED_VARIABLE(aContext);
    OT_UNUSED_VARIABLE(aMessage);
    OT_UNUSED_VARIABLE(aMessageInfo);

    gpio_set_level(UDP_LED, 1);
    vTaskDelay(100);
    gpio_set_level(UDP_LED, 0);
    vTaskDelay(100);
    gpio_set_level(UDP_LED, 1);
    vTaskDelay(100);
    gpio_set_level(UDP_LED, 0);
}
```

To send a UDP message we can use the following function:

```
void sendUdp(otInstance *aInstance)
{
    otError       error = OT_ERROR_NONE;
    otMessage *   message;
    otMessageInfo messageInfo;
    otIp6Address  destinationAddr;

    memset(&messageInfo, 0, sizeof(messageInfo));

    otIp6AddressFromString(UDP_DEST_ADDR, &destinationAddr);
    messageInfo.mPeerAddr    = destinationAddr;
    messageInfo.mPeerPort    = UDP_PORT;

    message = otUdpNewMessage(aInstance, NULL);
    otEXPECT_ACTION(message != NULL, error = OT_ERROR_NO_BUFS);
```

```
    error = otMessageAppend(message, UDP_PAYLOAD, sizeof(UDP_PAYLOAD));
    otEXPECT(error == OT_ERROR_NONE);

    error = otUdpSend(aInstance, &sUdpSocket, message, &messageInfo);

 exit:
    if (error != OT_ERROR_NONE && message != NULL)
    {
        otMessageFree(message);
    }
}
```

Here is an example of calling the above function inside a button interrupt from an STM32WB55 board.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if(GPIO_Pin == BUTTON_SW1_PIN) {
      sendUdp(PtOpenThreadInstance);
  } else {
      __NOP();
  }
}
```

From:
https://dokuwiki.zmitac.aei.polsl.pl/ - **RAU6 Dokuwiki**

Permanent link:
https://dokuwiki.zmitac.aei.polsl.pl/en:edu:zmitac:s2:aih:projekt:studenci:openthread

Last update: **2024/09/18 20:58**