Lab: Thread

Introduction

The purpose of this lab is to get you some hands-on experience with Thread. This will come in a couple of different forms:

- Create a network of thread devices
- Send data over IP between devices in a network
- Write an application using UDP

Goals

- Understand steps to create a Thread network and join it
- Explore UDP communication over a Thread link
- Consider the topology of a mesh network
- Write embedded applications that use traditional UDP communication

Equipment

- Computer
- nRF52840DK + USB cable (3 total for the group)

Github Classroom

- https://classroom.github.com/a/RwmRKAzG

Partners

- This lab should be done with **your group of three**

Table of Contents

List of Checkoffs

- Section 3.1: Demonstrate that you can receive Acknowledgement packets

- Section 5.1: Show your Thread network details
- Section 5.2: Show your IP addresses and understand them

- Section 6.1: Show my Thread network details
- Section 6.2: Draw the topology of the mesh network

- Section 7.1: Get data from the UDP servers.
- Section 7.2: Which UDP server is the child node?

- Section 8.1: Demonstrate your working LED control application

Lab Setup

1. VSCode and nRF Connect

We're using the same tools as the BLE lab, so they should already be good. If something about your setup needs to be re-done, see the instructions for the BLE lab.

HOWEVER: this lab has been tested with nRF Connect version 2.9.1. If you're using a different version, there may be issues.

CHECKOFF: None. Continue to the next section.

2. Create Your Lab Git Repo

I'll want to look at the code you wrote, so I need to give you somewhere to put it. Github classroom makes private repos for each student team so you can get the starter code and upload your own modifications. I can access all student repos, but you can only access your own.

- There is a github classroom link on the first page of this document. Click it!
- Pick a team name
  - Unless someone else already started it, in which case, join their team name
- Generally, do what github classroom says
- At the end, it should create a new private repo that you have access to for your code
  - Be sure to commit your code to this repo often during class!
- The repo link might 404. If so, you first have to go to https://github.com/nu-cs433-student and join the organization
- Clone the repo locally on your computer
  - If you're on Windows: git BASH does a good job https://gitforwindows.org/
    - Make sure there are no space characters in the entire path to the repo. Probably put it somewhere like "C:/nrf_apps/REPONAME" if you have a space in your username.
  - If you're on MacOS or Linux, you can clone the repo from command line
  - We'll be using VSCode for everything, and it has a mechanism for working with git too, so you could use that: https://code.visualstudio.com/docs/sourcecontrol/overview#_cloning-a-repository

CHECKOFF: None. Continue to the next section.

Communicate with Raw 802.15.4

3. Send and receive packets

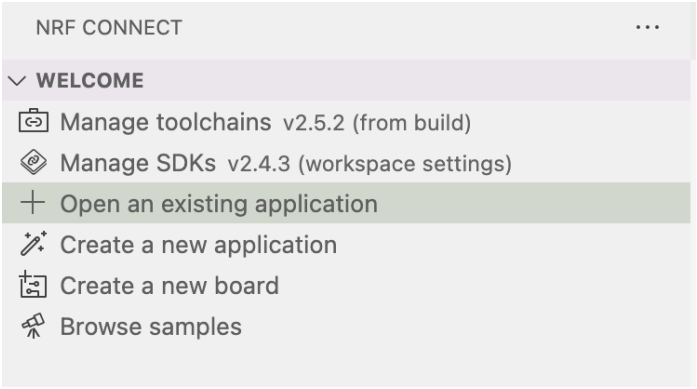
There are three apps for you that already do this:

- `nrf-154-send` - This sends data packets
- `nrf-154-recv` - This receives data packets destined for it
- `nrf-154-promiscuous` - This receives all 15.4 packets on the channel

- First, modify the source and destination in both the sender and receiver so they're unique to your team. Otherwise you'll all receive each other's packets.
  - Generate at least the first 16 bits at random.
  - You could also change the channel if you want. Channels 11-25 are valid.

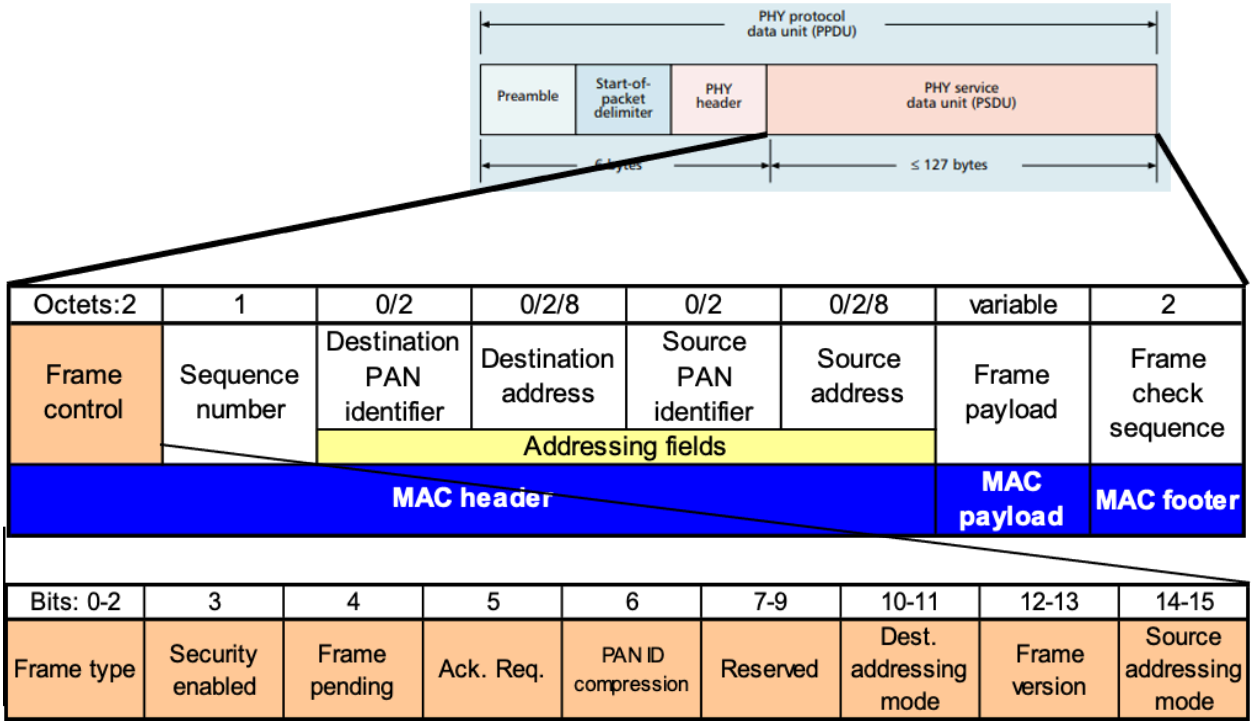
- Now, load these apps onto your three boards and observe the behavior. Check that you can see the packet data arriving on both the receiver and the promiscuous board.

- Open the one of those applications in VSCode using "Open an existing application"



- Add a build configuration for it, same as before:
  - Board as **nrf52840dk/nrf52840**
  - Base Configuration as **prj.conf**
  - **DO NOT SKIP THIS. IT'S VERY IMPORTANT TO GET IT RIGHT.**
  - The first time especially, this can take *forever* to complete
- Build the code, Flash it to the nRF52840DK

- Now, update the Sender to require acknowledgements. To do so, you'll need to modify the Frame Control header to enable the Ack Required bit.
  - The two bytes in the Frame Control header are placed in little-endian order in the packet.
  - Note that bits are listed backwards for some reason in this diagram.0



- For better documentation about what those fields mean, see here: https://onlinedocs.microchip.com/oxy/GUID-4B3D771E-5647-4191-AD45-C897B0D23071-en-US-3/GUID-46A20735-4885-446D-9C0A-461B67A126BF.html

- Our default packet sets the Frame Type to be Data, and sets the Destination and Source addressing modes to be “Address field contains a 64-bit extended address”.

- When you make the change correctly, the receiver will still receive the packet, but the sender will have an error, since it is not receiving an acknowledgement.

- Update the receiver to automatically acknowledge incoming packets. This is easy: there’s a function call where we set auto acknowledgements to false already in the code. You should make that true.

  - This should make the sender succeed again! Also, you should see the acknowledgement packet on your promiscuous scanner!

- Change the sequence number on the sender, and you should observer the sequence number acknowledged change to match

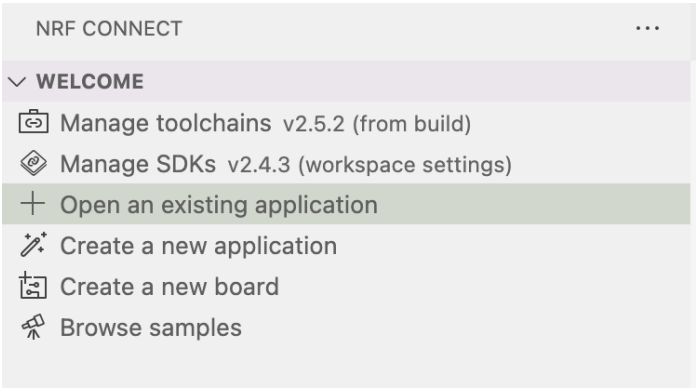
1. **CHECKOFF:** Demonstrate that you can receive acknowledgement packets.
    - Point out the acknowledgement packet from your promiscuous scanner
    - Prove to me that it's your acknowledgement, and not someone else's

Create a Thread Network

4. Upload the Thread CLI Application

We'll start by using the Thread CLI (Command Line Interface) application. It provides a text-based interface for sending commands that manage and introspect a Thread network. We'll have this same command-line interface available on all of our applications for this lab.

- Open the "thread-cli" application in VSCode using "Open an existing application"



- Add a build configuration for it, same as before:
  - Board as **nrf52840dk/nrf52840**
  - Base Configuration as **prj.conf**
  - **DO NOT SKIP THIS. IT'S VERY IMPORTANT TO GET IT RIGHT.**
  - The first time especially, this can take *forever* to complete
- Build the code, Flash it to the nRF52840DK

- Monitor board output
  - To view board print statements, you'll need to open a serial terminal. In the application panel on the left, under "CONNECTED DEVICES", you should see a serial number for your board, then in a dropdown from that, one or more serial devices. The little "plug" icon when you hover over one of the serial devices should open up a serial port to it.
  - If it asks you for settings: 115200, 8n1, and rtscts:off are correct (115200 baudrate, 8 data bits with no parity bits and one stop bit, and no request-to-send/clear-to-send)
  - Hit the reset button to see print output.

- Interact with the CLI
  - Either hit the enter key a few times, or hit the reset button on the board to see a prompt

- Type `help` to see all commands
- Type `ot help` to see all commands relevant to OpenThread

CHECKOFF: None. Continue to the next section.

5. Building the Network

Once you have at least two boards with the Thread CLI loaded onto them, you can build your own network.

I recommend connecting to each board on a separate computer. That'll make it easier to remember which console is which board and will allow you to add some physical distance between the boards.

We're going to be using the Commissioning process, which requires that a board be authenticated and authorized to become part of the network. One of your boards will have at least three roles: Thread Leader, Commissioner, and Router. The other board will either be a Router or an End Device, depending on your configuration. Commissioning in Thread is described here (we'll be doing "on-mesh" commissioning without a Border Router): https://docs.nordicsemi.com/bundle/ncs-2.9.1/page/nrf/protocols/thread/overview/commissioning.html

- Here is a guide to the CLI commands: https://github.com/openthread/openthread/blob/main/src/cli/README.md

  A warning, not all commands will work. Depending on the configurations used in the `prj.conf` file, different options are enabled on the board and it will be able to respond to different commands.

  You'll need to look through a bunch of those, as they'll be very important. All of them are behind the "ot" command to start with, even though they don't show it in the examples. So something like "`ot scan`", not just "`scan`".

- Generally, to get started, you should follow major steps 2 ("Disabling the Thread Network") through 4 ("Adding the Joiner") here: https://docs.nordicsemi.com/bundle/ncs-2.9.1/page/nrf/protocols/thread/overview/commissioning.html#configuring_on-mesh_thread_commissioning

  Two immediate exceptions:
  a. FIRST, the commissioner should set a password OTHER than N0RD1C. You don't want to use the same password as everyone else in the class. It has to follow the Joining Device Credential creation rules (At least six characters, no `I`, `O`, `Q`, or `Z`.)
  b. SECOND, You might need to make the second device join your first network rather than some other team's network. Try doing nothing special about this first and see if things just work.
     - If they don't work, try this: https://github.com/openthread/openthread/blob/main/src/cli/README_DATASET.md#attach-to-existing-network

    - I also believe you can do this by setting the `panid` value on your new device to match the `panid` for the leader. (The leader will automatically set a `panid`) But I'm not 100% sure.

A couple of notes for problems I ran into:

a. The commissioner step will fail until your device is a thread leader. If it doesn't work, give it 20 seconds or so. You can check the thread state with the `ot state` command.

b. When running "ot commissioner joiner add…", the goal is to allow a specific device to join the network if it has the key passphrase. So the address there is the EUI64 from the device that is joining the network.

c. When you actually join with the secondary device, it should work the first time. I ran into an issue though where it totally failed the second time I tried it though. It turns out the device caches the PAN ID from the last network it connected to. You can set the PAN ID to 0xffff to overwrite this and get joining working again (after stopping thread and doing an `ot ifconfig down`)
    - Alternatively, you can "Erase and Flash" the device by clicking the little chip icon that appears next to the Flash command in VSCode. That'll clear the Thread network config from Flash and give you a fresh board.

d. Once you add the joiner to the commissioner you have about two minutes to then commission the new device before it times out. You can change this with an additional timeout argument to `ot commissioner joiner add`
    - 15000000 (fifteen million) worked well for me

e. Generally, it just takes a bit for everything to start up. After connecting the joiner successfully, it still takes up to 30 seconds for it to actually be a functional network member (as checked with the `ot state` command). And then another minute or so before it decides to upgrade itself to a router.

f. If you're having problems with devices not seeming to join the network correctly, make sure that thread is started on the board `ot thread start`

- Get all three of your boards connected together into a single network.

- Make one of the three boards into a child device.

  All boards by default act as routers. Not right away as the join: remember that all thread

devices join as the child of an existing router. But they quickly (10-60 seconds) upgrade to be routers.

You can make a board become and stay and End Device (child) though. To do so, take a look at the `ot routereligible` and the `ot state` commands.

1. **CHECKOFF:** Show me details of the network you created (screenshots are fine)
    a. The network Dataset (on the commissioner)
        i. Other devices can run `ot dataset active`
    b. The Router Table on the commissioner
        i. Should be two routers (one is the leader/commissioner)
    c. The Child Table on the commissioner
        i. Should be one child

2. **CHECKOFF**: Get IP Address details and understand them
    a. Show the IP addresses of all three devices
    b. Why are there all these IP addresses, and what do they mean?

My Thread Network

6. Investigate My Network

For some more interesting insights, we'd need a larger network. Good news! I have a network of several (more than 5) devices set up in the area around my office: Tech L368. You should be able to connect to the network and determine things about how it works.

Notes:

1. This involves physically traveling to Tech, so feel free to skip this for now if you're in lab on a Friday and go to the next major section.
2. Connecting to a Thread network is rather finicky. So don't do this section until you've done the previous one and have experience setting up Thread networks.

Instructions:

- To connect, you'll use the same CLI application we've been using previously. Here's that guide again: https://github.com/openthread/openthread/blob/main/src/cli/README.md

- I have already set up a commissioner, with the password: NUCS433
    - So you'll do the `ot joiner start NUCS433` command on your board
        - Make sure that your network interface is up first

    - Make sure to either Erase-and-Flash it to clear the prior network's credentials, or you'll have to manually set the PAN ID to 0xFFFF with `ot panid 0xffff` to clear the credentials

    - It'll still take a few seconds, but you should be able to connect to the network. If you're having serious issues, let me know. There's a small chance that the network crashes and you can't join anymore, although I've kept it stably running for several days straight without issue so far.

- Once you've joined, snoop around the network a little to figure out how it connects.
    - A particularly useful tool for doing so is the `ot meshdiag` family of commands. There are several subcommands there that let you inspect topology, router tables, and children for any node on the network.
    - These commands only work on `thread-cli` out of the box. To enable them for other apps, add `CONFIG_OPENTHREAD_MESH_DIAG=y` to `prj.conf`

1. **CHECKOFF:** Show me details of my network (screenshots are fine)
    a. The network Dataset
    b. How many devices are routers and how many are children
    c. IP addresses for all of the routers AND children

2. **CHECKOFF:** Draw the topology of the mesh network
    a. Show which routers connect to which other routers
    b. Show which routers are parents to which children
    c. Show which device is your own connected device
    d. Draw in any clean and readable way you like: hand-drawn, diagramming software, powerpoint, MS paint, etc.

7. Communicate with UDP Servers

In addition to just existing, two of my devices are running IPv6 UDP servers. You can send data to them with a CLI command and get responses back.

- ➔ UDP Server 1: fd32:bb13:60ea:6ae1:1fb:55eb:3663:b435
- ➔ UDP Server 2: fd32:bb13:60ea:6ae1:25a6:b047:2460:9a00
- ➔ Port: 6060 (for both)

- To send UDP messages, you can use the `ot udp` family of commands.
    - Be careful not to mess up the IPv6 address. You should be able to copy-paste it.
    - The UDP servers accept a number of commands. If it's not a valid command, it should respond with the list of commands. So just start by sending whatever data you want.
    - One of the UDP servers is a child node, with a polling period of 10 seconds. So it will respond with up to 10 seconds of latency.

1. **CHECKOFF:** Get data from the UDP servers. For each server, determine what the list of commands are and get me the data from each command.
    a. Make sure you've gotten all the responses from both servers.

2. **CHECKOFF:** Which UDP server is the child node? (identify it by IP address)

Thread Application

8. LED Control Application

If you haven't realized yet, this class is going to be a big adventure in using buttons to blink LEDs via much too complicated of methods. So, let's do it with thread and UDP.

The overall goal is that you will have three devices. One commissioner/leader/router with no special code, one child which is also a UDP server, and one router which sends UDP packets. Whenever a button is pressed on the router, it should send a UDP packet to the server informing it that a button was pushed and *which* button it was. When the server receives a UDP packet, it should toggle the corresponding LED on or off (whatever the opposite of the current state is). So each time you press a button, the corresponding LED should turn either on or off.

- The commissioner/leader can be set up as you've done previously with the `thread-cli` app.

- There is starter code for the two applications named `router-base` and `child-udpserver`
    - The relevant bits are the code in `main.c` and the `prj.conf` settings.

    - Note that these applications *also* have the Thread CLI installed, so you can configure their Thread network settings via that.

    - For the child, you can configure the polling period in the `prj.conf` file. This will determine the latency of its responses. Here's where you can look up information on `prj.conf` settings: https://docs.nordicsemi.com/bundle/ncs-2.9.1/page/kconfig/index.html

- First, you should figure out how to write an IPv6 UDP server that works on a regular computer (before trying to get it working on an embedded device).
    - If you've never used the socket interface in C before, slides 49-66 here explain some of the details: lecture18_networks.pdf

    - Here's a pretty good example of how to write a UDP server and client: https://www.tack.ch/programming/network/sockets/udpv6.shtml
        - However, you should hard-code a Port number rather than let it choose. And the `getsockname()` call isn't all that helpful.
        - You'll also want to modify it to run forever by putting `recvfrom()` in a while loop.

    - Try this locally first!! If you've got a Linux or MacOS machine you can run it. You could also go onto Moore and run it. You can test the server side alone with the Netcat command: `nc -6 -u ::1 PORT` (then type a message and hit enter)

- Next, you'll want to port that server to your child application. The POSIX socket function calls *should* just work as-is without changing them. (Which is a pretty darn cool feature of Zephyr.)

    - The `initialize_server()` function will hold the server startup code.

    - The `run_server()` function will hold the while loop and your `recvfrom()` call.

    - The `k_work` mechanism that's already set up runs a function in a dedicated thread. That way it can run indefinitely in parallel to whatever the other code is doing (in this case, keeping Thread running). https://docs.zephyrproject.org/latest/kernel/services/threads/workqueue.html

    - You can test your child UDP server with the `ot udp` family of commands from your commissioner, just like you did in the prior step with the UDP servers on my Thread network.

- Finally, you'll want to port the UDP client to your router application. Again, the code *should* "just work" (easier said than done).

    - Since the UDP client implementation doesn't need to block forever, you can call those functions directly from the button callback and don't need any special mechanism for it.

    - For the Router, you'll need an IP address for the child. You should boot the child first, figure out what its IP address is, and then hardcode that IP address in the router code.

- Remember how buttons and LEDs work from your BLE implementation to wrap up the whole thing.

2. **CHECKOFF:** Demonstrate your working LED control application

    - Explain the code you wrote to make this work as well

    - Was anything particularly challenging to get working?