Lab 3: IEEE 802.15.4 & Thread

The 802.15.4 lab will give you experience developing a wireless device using the IEEE 802.15.4 spec.

## Hardware Tools

We will be using the [nRF52840DK](https://www.nordicsemi.com/Products/Development-hardware/nrf52840-dk) (left) and the [nrf52840 Dongle](https://www.nordicsemi.com/Products/Development-hardware/nrf52840-dongle) (right). Your group will have one nRF52840DK board, and two dongles. You should not assign devices to group members. You should share hardware as a group.



## Software development environment

We will be using Visual Studio Code along with the nRF Connect extension. With this extension, you can build and create firmware images, download and debug onto the hardware kits directly.

## Documentation and Resources

### nRF Connect

* How to flash, test, and debug on nRF Connect: <https://nrfconnect.github.io/vscode-nrf-connect/get_started/quick_debug.html>
* nrf\_802154 API Documentation:
  + <https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrfxlib/nrf_802154/doc/api.html>
* IEEE 802.15.4 - 2006 Specification:
  + <https://ieeexplore.ieee.org/document/1700009>

### OpenThread

* + Command descriptions:<https://github.com/openthread/openthread/blob/main/src/cli/README.md>
  + OpenThread guides:<https://openthread.io/guides/>
  + There are a lot of examples online for OpenThread, I recommend [googling](http://google.com).

If you want to start over, you can run ot factoryreset on your device.

[Hardware Tools](#_8h6s0jyq1f3v)

[Software development environment](#_loes9siyliil)

[Documentation and Resources](#_jun4d6iqxp7n)

[nRF Connect](#_tmw577mqovh1)

[OpenThread](#_wwcveu8czo7f)

[Pre-Lab](#_kx9ekwihp0k0)

[Pitfall: Check your SDK Version](#_tewi21er4p3p)

[A. Install the timer app](#_56mhtggjkrpx)

[B. Modifying the Application](#_9mqn8tw0if37)

[C. Installing the Thread CLI app](#_ug3aqhqyqnfi)

[D. Interact with the CLI Thread app](#_16jqiq7cfhyh)

[Lab 3A (Week 1)](#_5632twpbgbai)

[Pitfall: Do not copy or download the lab manual](#_74twdpz3hdph)

[E. Setting up a Receiver and Sender](#_ur32624ws8s4)

[Receiver setup](#_rszrec6ofjfl)

[Sender setup](#_pmesptlpn3ii)

[Verify packet reception](#_q4fsix53asg8)

[F. Filtering out packets not meant for your receiver](#_z4uj8swuv8rf)

[G. Getting acknowledgements from the receiver](#_f3nkklwbkft)

[H. Reducing the packet size](#_gkn5mx7hy9g2)

[I. Low Power Listening (LPL)](#_v4k7cfz97ib5)

[Packet Trains](#_vbvy047w7ics)

[Receive Windows](#_5qz3n2ocly2p)

[Implementation](#_24tknpduc30b)

[Transmitter](#_4c7e5ksg48w9)

[Receiver](#_w6pej9s1sa0o)

[Statistics](#_yol1zeqbgpcp)

[Troubleshooting](#_rnt5tzpu8lz5)

[Analysis](#_sr108al5xwj)

[Deliverables](#_ijn872arbwxg)

[EC 0. OPTIONAL: Lowest Power Listener](#_o9og2848uo9l)

[Lab 3B (Week 2)](#_4c0p7v2zkp2o)

[Introduction](#_fz7lxhcvyn8j)

[J. Installing the Thread CLI App](#_bpyvyzy8jq20)

[K. Join the class Thread network](#_krh1pblt6f7l)

[L. Send UDP messages between your devices](#_72byesj0dw6f)

[M. UDP Server](#_mcar9ix4i4p0)

[Starter Code](#_9bxrlhjek0fc)

[UDP Protocol](#_9varp4r76ppx)

[Thread Device](#_6m7n2fl3r0tq)

[Deliverables](#_vk1zi9or11z1)

[EC1. OPTIONAL: Ping outside network](#_w6mats2i302)

[EC2. OPTIONAL: Visualize the network topology](#_hhzct96mv8r1)

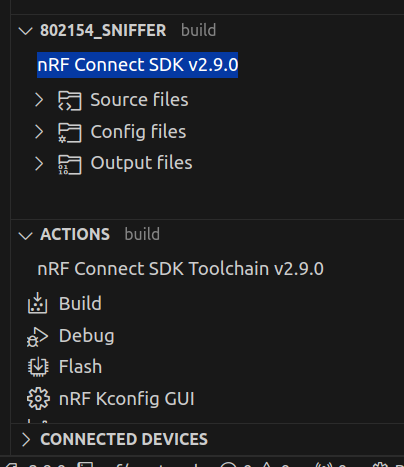
[EC3. OPTIONAL: LED Service](#_4x6de2rjianu)

# 

# Pre-Lab

First, download the Pre-Lab 3 Worksheet document to fill in as you work.

Also, accept the invite for this lab’s GitHub classroom assignment. You’ll have one repo per group which will have some starter code for this lab.

<https://classroom.github.com/a/pxKgQU-4> 

## Pitfall: Check your SDK Version

Make sure your **NRF SDK version is >= v2.9.0**. You can check this in the NRF Connect side panel in VSCode. If that doesn’t work, try checking in the NRF Connect Desktop app or looking at the version name in the /ncs/ folder on your computer.

## A. Install the timer app

Go to the timer subdirectory in your new GitHub repo and flash the timer app.

## B. Modifying the Application

Edit the timer app to create a second timer. Make the two timer intervals different and have each timer print a different message.

**TASK 1: Show the terminal output with the two timers printing periodically.**

**TASK 2: Include your modified code in the prelab.**

## C. Installing the Thread CLI app

Use Nordic’s nRF Connect extension for VS Code.

* Open the nRF Connect extension on VS Code.
* Choose “Create a new application” from the side bar in the “Welcome” pane. Then choose “Copy a sample” and use nrf/samples/openthread/cli. Open your copy of the sample.
* Add a build configuration.
* Flash the app to the board.

## D. Interact with the CLI Thread app

The CLI Thread app is a full implementation of Thread with a command line interface to run Thread commands.

* Use the nRF Connect connected devices pane to open a console to the connected board.  
  
* All Thread commands are under the “ot” (for “OpenThread”) namespace. That is, all commands are formatted as ot<space>command.

**TASK 3: Run the ot help command in the CLI app and copy or screenshot the output.**

# 

# Lab 3A (Week 1)

Make a copy of the postlab template to fill in. You will submit one postlab as a group.

We will be working with the IEEE 802.15.4 radio on the nRF52840DK in this lab. We will be using Nordic’s [nRF 802.15.4 Radio Driver](https://github.com/nrfconnect/sdk-nrfxlib/tree/main/nrf_802154), a set of libraries to work with the 802.15.4 radio on the nRF52 and other SOCs.

## Pitfall: Do not copy or download the lab manual

We make bugfixes as the lab runs, so check back on the Google Doc for the most up to date information. This will make sure you don’t waste time debugging irrelevant problems.

## 

## E. Setting up a Receiver and Sender

One of the boards will be the 802.15.4 receiver and the other will be the 802.15.4 sender.

### Receiver setup

* Pull the github repo and add the nrf-802154-recv application to nRF Connect on VS Code.
* Familiarize yourself with what the receiver code (main.c) is doing.
* Build the code, Flash the code to the nRF52840DK. Monitor the serial port of the board using nRF Terminal.

### Sender setup

* Add the nrf-802154-send application to nRF Connect.
* Familiarize yourself with the sender code.
  + Can you spot where the actual packet is created?
* Before flashing the code, let’s make some changes to the packet!
  + Change the 1st byte of the payload to your group number.
  + **TASK 1: What did you set the 1st byte of the payload to?**
* Flash the code. You probably want to flash it to your dongle. If you want to see the sender serial output, swap roles and put the receiver on the dongle.

### Verify packet reception

* On the receiver board’s serial, you should see your packets starting to appear.
* Do you see packets from other groups? Why?

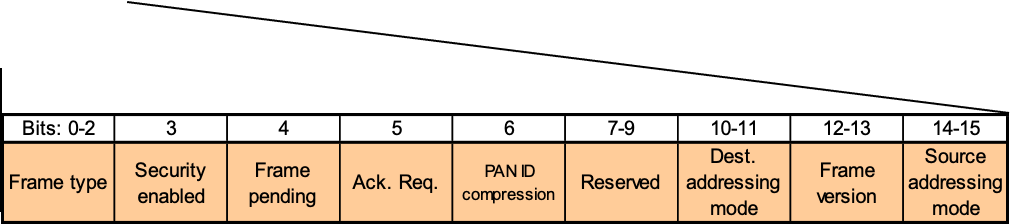
**TASK 2**: Prove that you can receive the packets with the updated payload.

* A screenshot of serial output works here

## 

## F. Filtering out packets not meant for your receiver

IEEE 802.15.4 packet format:



By inspecting the packet headers, we can look at how we can customize our communication to avoid the communication from other groups.

**TASK 1: But first, let’s make sure we understand 802.15.4 transmission. The preamble is 4 bytes. How long (in time) does it take to transmit the preamble?**

Now, let’s try to fix this so we no longer receive other group’s packets! To filter out packets and reduce traffic, do the following:

* Change one or more bytes of the receiver’s extended address. Make sure you update the same in the sender’s packet!
* **TASK 2: What is the updated extended address of your receiver?**
* Change the channel to a channel of your choice. **Remember**: 802.15.4 has channels 11-26 at 2.4GHz. Channel 11 is used by default by the nrf\_802154 library.
* **TASK 3: What nrf\_802154\_ function did you use to change the channel?**
* **TASK 4: Which channel did you use?**
* Use the destination’s PAN ID in the sender packet instead of using the broadcast PAN ID.

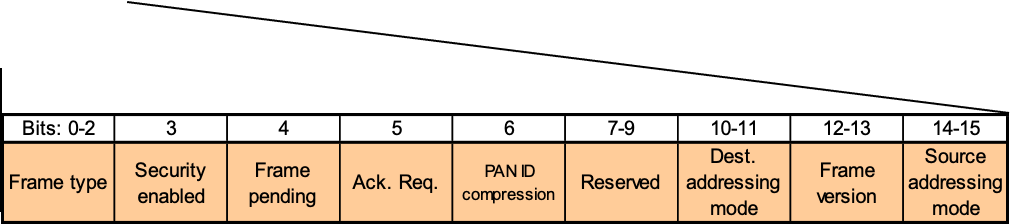
**TASK 5**: Prove you can receive packets with the updated address, channel, and PAN id changes.

* A screenshot of serial output works here.

## G. Getting acknowledgements from the receiver

In the current implementation, the sender does not wait for an acknowledgement from the receiver. So there is no way to ensure that the receiver actually received our packet. Let’s see if we can get an ACK from the receiver when it receives the packet.

* The transmitter can request the receiver to send an acknowledgement. To do so, you need to set the “Ack. Request” bit in the **Frame Control** field of the packet.
  + Can you check the [IEEE 802.15.4 spec](https://ieeexplore.ieee.org/document/1700009) to figure out which bit to set?
* Set the Ack. Request bit to 1 in the Frame Control field of your packet.
* **TASK 1: Include a screenshot of the updated sender packet showing acknowledgements being requested.**
* Try sending the packet again. What do you see?
  + You should get an error on the sender (in **nrf\_802154\_transmit\_failed**). This is because the sender did not receive an ACK packet.
* The nrf\_802154 library has a feature to automatically acknowledge received packets. Check the [API docs](https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrfxlib/nrf_802154/doc/api.html) and turn this feature on in the receiver.



**TASK 2:** Demonstrate that the sender transmissions are successful

* A screenshot of serial output works here.

## 

## H. Reducing the packet size

Now that we have our basic transmission going, let’s try to reduce the size of our packet. IEEE 802.15.4 supports “short addresses”, which let devices be addressed using just 2 bytes rather than using the 8 byte extended address.

**TASK 1: By how many bytes did you reduce the packet size by switching to short addresses?**

* Look through the [API documentation](https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrfxlib/nrf_802154/doc/api.html) and set the short address of the receiver. You can set your favorite two bytes for the address (anything except 0xFFFF which indicates a broadcast address).
* You need to update the packet at the sender to ensure you’re addressing the receiver. Modify the packet to update the destination address.
* **TASK 2: What did you change the sender’s and receiver’s addresses to?**
* The Frame Control field indicates which type of addressing mode you’re using (short address or extended address). This lets receivers know how to parse the packet.
  + Check the spec to figure out how to set the destination addressing mode to short address in the Frame Control field.
* Update the PHY Header length (1st byte) of the packet to reflect the changes you’ve done to the packet. PHR = length of the MAC header + payload + FCS.



**TASK 2:** Set short address for the receiver

* Include your updated sender code and receiver code.
* A GitHub link to a specific commit or lines of code works here.

# 

## I. Low Power Listening (LPL)

*(Note: If you’re moving fast through the lab, you might get through this part too; this is just where we expect you to have to transition from “lab time” to “labs are the homework in this class” time).*

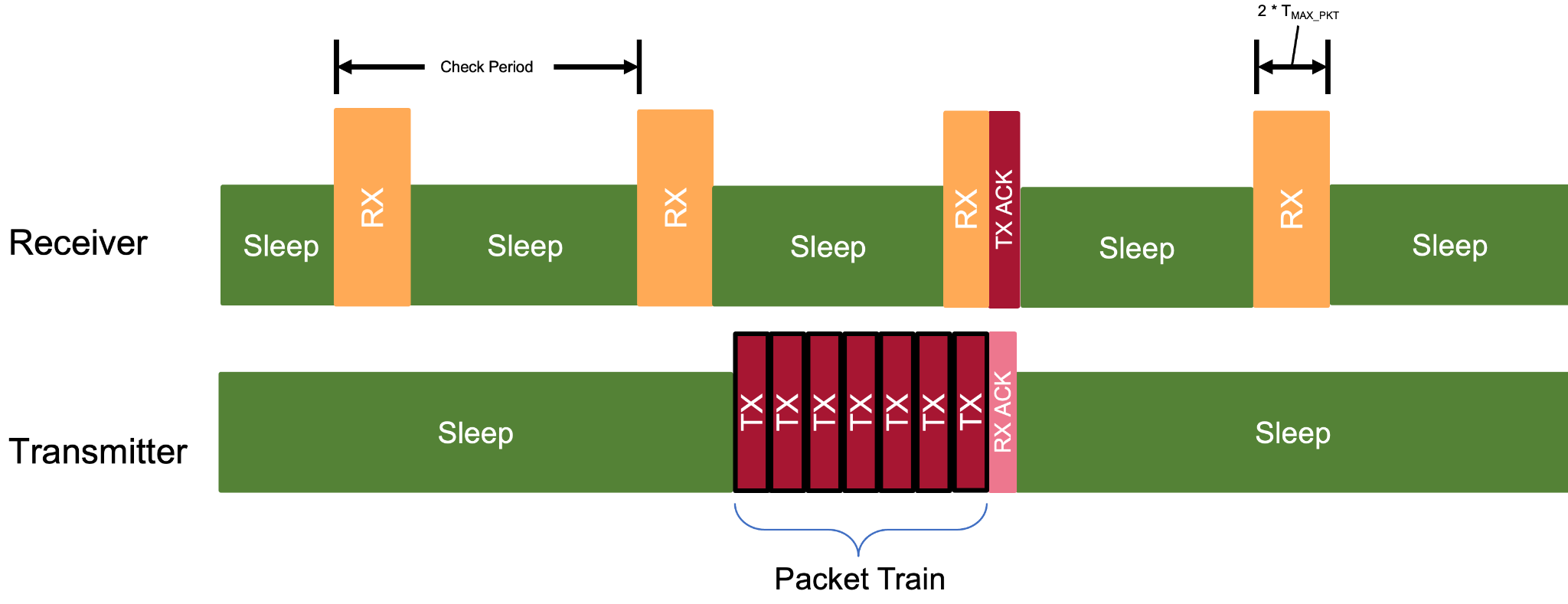
In lecture we discussed how the "base" MAC protocol in the IEEE 802.15.4 specification includes a simple, contention-based MAC that allows network-specific MAC layers to be built above. This led to a variety of academic research, which includes Low Power Listening (LPL) protocols that optimize energy for transmitters and receivers.

In this part, your task is to implement a version of LPL to send and receive data between two devices.

### Packet Trains

Key to LPL is having transmitters transmit for a long enough duration that receivers, which are periodically sampling the wireless channel, are able to detect the packet being transmitted.

The original B-MAC protocol implemented these long transmissions with the 802.15.4 preamble mechanism. However, in practice, it is not straightforward to get modern 802.15.4 radios to send long preambles.

Instead, we will implement the long transmit behavior with packet trains. That is, the transmitter will send the same packet multiple times back-to-back to emulate a long transmit period.

### Receive Windows

For a receiver to receive transmitted packets, the receiver must enter listen mode long enough to receive a packet from the ‘train’. To do this, we will have the receiver stay active for two times the duration of a transmission of the longest 802.15.4 packet. If the receiver does not receive a packet, it can disable receive mode and return to sleep.

### Implementation

You must implement the transmitter and receiver. The check period (the interval between receiver receive windows) is known and hard coded (meaning it is compiled into the application) on both the transmitter and receiver.

You should have a variable or `#define` in your code so that you can change the check period.

### Transmitter

This device must initiate a transmission in response to `Button1` being pressed.

It must send the same packet multiple times back-to-back enough times such that the total transmit time is as long as the check period. You will have to calculate how many transmissions are required depending on the packet length.

The transmitter should send a packet that includes specific source and destination addresses. It should request an acknowledgement. The payload of the packet should be some ASCII string that includes your group name/number.

If your transmitter receives an ACK it should stop transmitting the packet train.

### Receiver

The receiver should normally be in sleep mode with receive disabled.

It should periodically enable receive mode according to the time of the check period. It should stay active in receive mode until either:

1) it receives a packet, or

2) the time sufficient to send the longest possible 802.15.4 packet twice has elapsed.

After the receive window the receiver should go back to sleep (i.e. disable receive mode).

When the receiver receives a packet it should acknowledge it (automatically) and print the contents to the terminal (as in the lab).

### Statistics

Every 15 seconds, the receiver should query the internal statistics maintained by the 802.15.4 software stack and print the results.

Example code to query and print statistics:

nrf\_802154\_stats\_t stats;

nrf\_802154\_stats\_get(&stats);

LOG\_INF("[Receiver Stats] CCA fails: %llu", stats.counters.cca\_failed\_attempts);

LOG\_INF("[Receiver Stats] RX Events: %llu", stats.counters.received\_energy\_events);

LOG\_INF("[Receiver Stats] Frames received: %llu", stats.counters.received\_frames);

The sender should print a count of how many packets it sent before getting an ACK, or if the packet train failed.

The receiver should print every time it starts to receive and sleep, the data of the packet it receives, and the statistics as described in the above code block.

### Troubleshooting

* Zephyr Fault / MPU Fault / Board crashing / Weird prints
  + **Pitfall 1:** It’s possible that there are too many log statements that are overflowing Zephyr’s buffer for log statements. Make sure that the log level config option CONFIG\_LOG\_DEFAULT\_LEVEL is set to 3 and **not** 4 (setting it to 4 will print out many debug messages which will quickly fill up the buffer). If that doesn’t help, try to increase the size of Zephyr’s internal log buffer with the config option CONFIG\_LOG\_BUFFER\_SIZE. Try setting it to something in the few thousands. To set these config options, set/modify them in your project’s prj.conf.
  + **Pitfall 2:** Your board may show weird behavior such as deadlock or corrupted prints because an interrupt comes in during some critical logic. In Zephyr, timer interrupts come in at the highest priority. These interrupts immediately transfer execution, so it’s likely you can have two different points of execution modifying the radio at the same time. This conflict can lead to many weird behaviors. We recommend two options for dealing with your interrupt handlers potentially running at the same time. (1) It’s common in operating systems to split handling an interrupt into two parts. The interrupt handler sets a global flag that the interrupt was seen and returns. In your main(), you repeatedly check if any interrupt flags have been marked and then change the state of your board accordingly. (2) The clean approach that Zephyr recommends is to use [Workqueue Threads — Zephyr Project Documentation](https://docs.zephyrproject.org/latest/kernel/services/threads/workqueue.html). This handles all the synchronization for you.

### Analysis

You should run your receiver and transmitter to successfully send and receive 5 packets using the following check period settings:

* Check period of 500 ms.
* Check period of 800 ms.

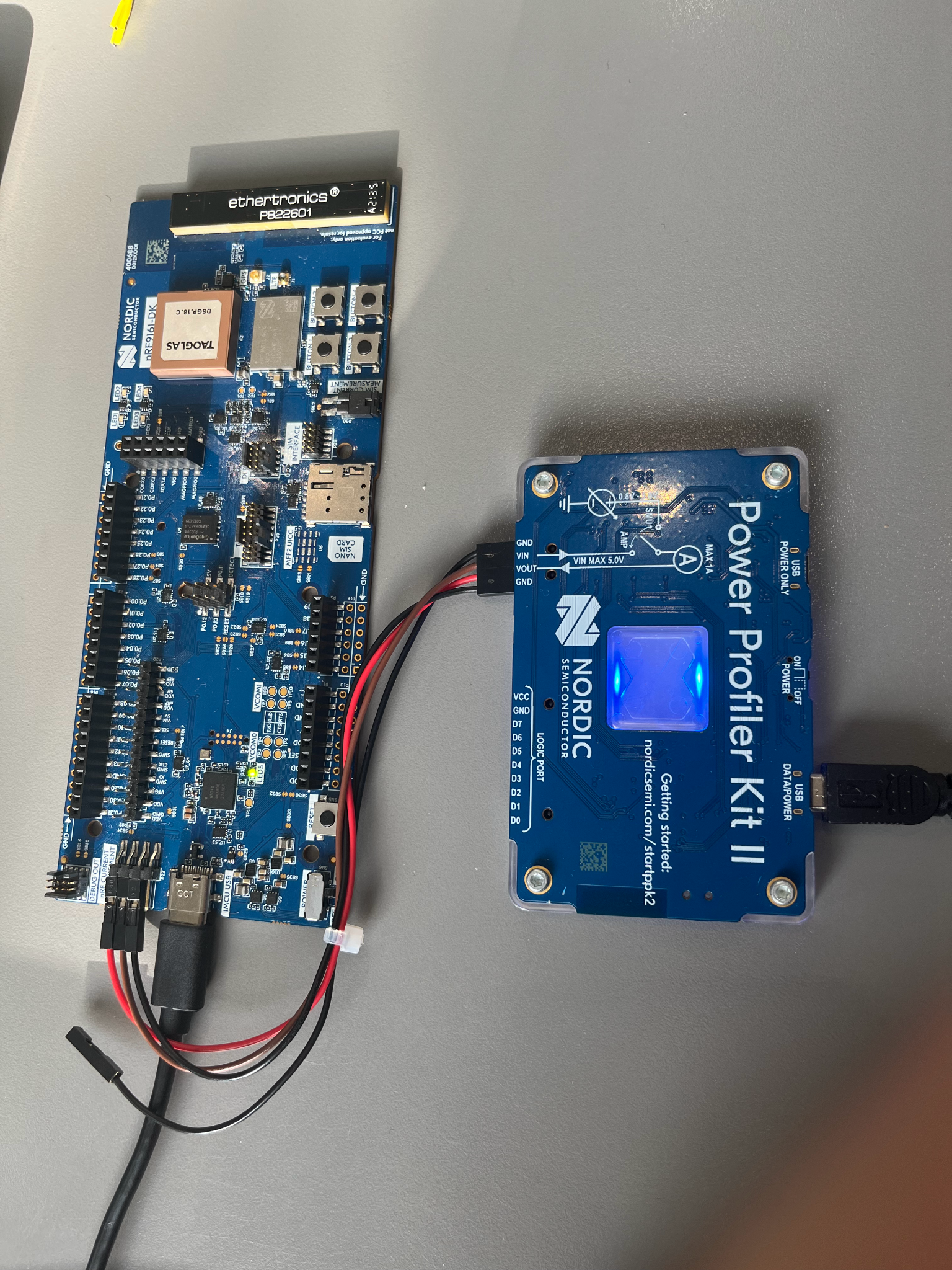
### Deliverables

* **TASK 1:** Code for the transmitter and receiver.
  + A GitHub link to the two main.c files is fine
* **TASK 2:** The terminal outputs from both the transmitter and receiver for the 10 packets using both check periods.
  + This should be four separate text files.
  + A GitHub link to these text files is fine
* **TASK 3:** A short (paragraph or two) description of your results. How do you know your receiver was correctly duty cycling to save energy? Did this version of LPL work? Did you run into any challenges?
* **CHECKOFF:** We will test your two boards with each other and against our reference implementation. Make sure to #define the check period and parameters of the network so you can easily change it.
  + We will change your check window to a large value to verify the board does not receive packets while sleeping.
  + We will make your check window reasonable to make sure it works correctly on both the receiver and sender. The sender should print how long the packet train was.

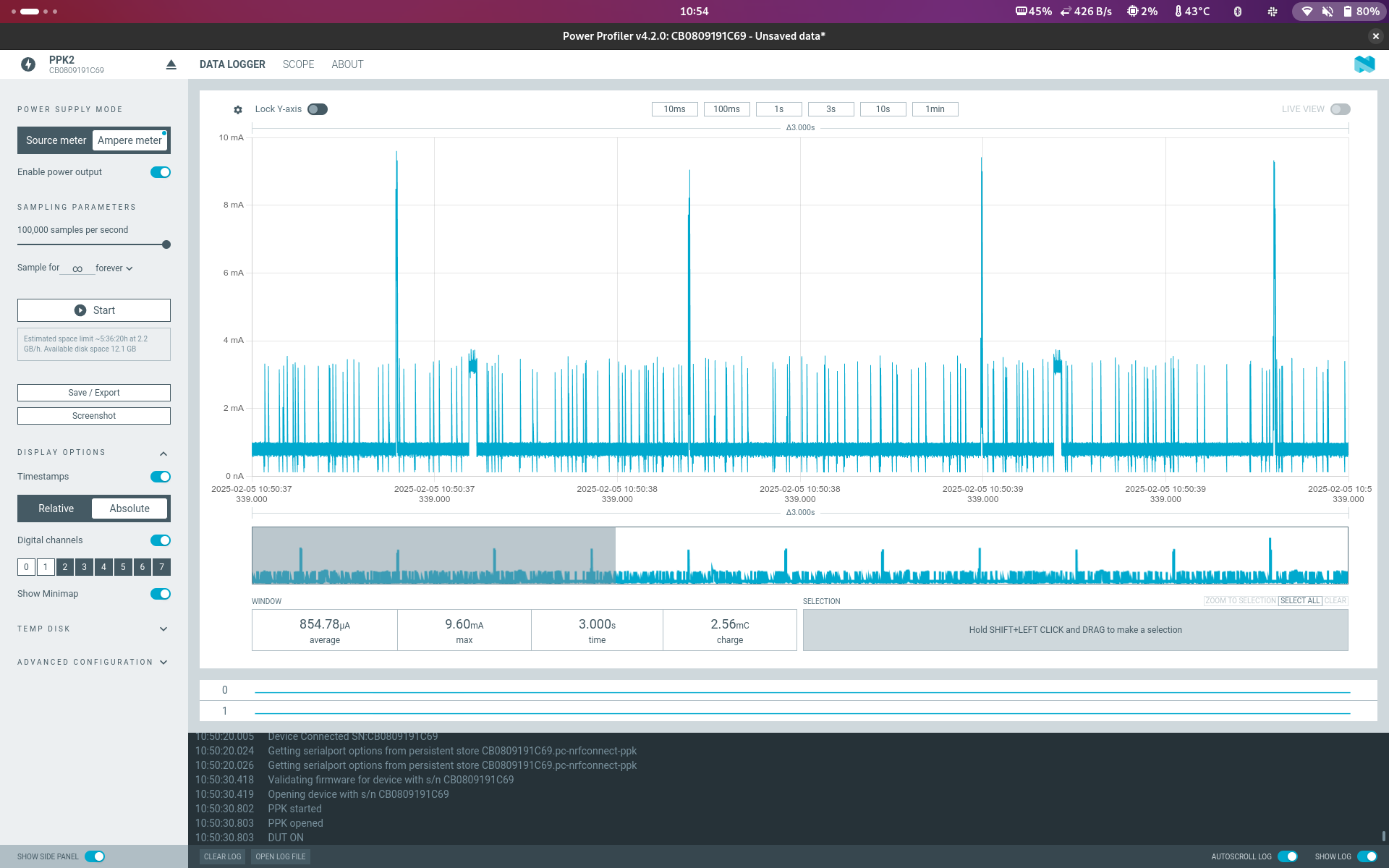
# 

## EC 0. OPTIONAL: Lowest Power Listener

From doing the LPL lab, you have learned that even though our machines seem to be constantly connected to the network, the radios are almost always off (or in some low power mode). If you are live streaming Lofi Girl while working on your prelab, the illusion of “real-time” is a lie, as all network streams are split up into packets. The obvious question might come up: how low power is LPL?



We set up the [Nordic Power Profiler Kit II](https://www.nordicsemi.com/Products/Development-hardware/Power-Profiler-Kit-2) by connecting some jumper cables to our NRF52840dk board. This allows us to sample the power usage of the board and graph it. You can see the following graph shows time on the x axis and current draw in milliamps on the y axis. The spikes to 10mA show when our LPL is in receive mode. The board draws ~1/20th the power when the radio is sleeping. The high frequency spikes to 3mA are probably background tasks happening in Zephyr. The starter code which didn’t sleep the radio would look like it is constantly drawing several mA. This makes a huge difference for battery life. A small electronics device like an airtag will only have about 200mAh of energy available in its coin cell battery.



There is nothing to do for this extra credit section. If there is interest from multiple groups, we will set up a competition leaderboard for which group can make the lowest power listener.

# Lab 3B (Week 2)

## 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
* 2 nRF52840DK (with micro USB cable)
* 1 nRF52840 Dongle

## J. Installing the Thread CLI App

Make sure you have the CLI app installed on both nRF52840dk boards.

## 

## K. Join the class Thread network

We have a Thread network running in the lab which you will be using. You will join using the network key b159079cee5211a9fe37289ac93a0f08.

**Note:** The thread network will be available in CSE 3219 until this lab is due. You’ll need to work in the lab to use the network.

**Note**: Many commands are async, and take a while to complete. You might see output right away, and you might be able to enter another command, before the previous command has actually completed. “Done” typically means the command was accepted/started (not that it has completed).

You should join the network using the following conceptual steps (the exact commands you need to figure out):

* Enable interface (using ifconfig).
* **Join** the network using the preshared key. This may take a couple of tries…
* Enable thread.
* Verify the router state.

You should do this on both boards.

Once you are connected:

* List the device’s IP addresses.
* Run ot router table.
* Run ot child table.
* Use one device to ping your other device.

**TASK 1:** List the IP addresses of both connected boards and include them in your report.

**TASK 2 :** Run the ot router table command in the CLI app and copy or screenshot the output.

**TASK 3:** Run the ot child table command in the CLI app and copy or screenshot the output.

**TASK 4:** Run the ot neighbor table command in the CLI app and copy or screenshot the output.

**TASK 5:** Run a ping and show the output.

### 

## L. Send UDP messages between your devices

Here are details on the ot udp command with examples: [openthread/src/cli/README\_UDP.md at main](https://github.com/openthread/openthread/blob/main/src/cli/README_UDP.md)

* Send UDP packets between your boards. You can use any port(s) you want.

**TASK 1:** Show evidence of sending UDP messages between your devices.

## 

## M. UDP Server

For this part you will develop a networked Thread device that acts as a UDP server. Your device should receive commands via UDP packets and send responses with UDP packets.

### Starter Code

To get you up and running, we’ve added starter code to the repo for this lab. It’s under the directory child-udpserver <https://github.com/ucsd-wxiot-wi25/15.4-thread-starter/tree/main/child-udpserver>

Copy over this code to your team’s repo that you’ve been using for the previous sections.

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

### UDP Protocol

Your device should support the following commands:

* on
  + Action: turn one or more of the LEDs on the board on.
  + Response: “ok” if the command succeeded, or “fail” otherwise.
* off
  + Action: turn one or more of the LEDs on the board off.
  + Response: “ok” if the command succeeded, or “fail” otherwise.
* name
  + Action: None
  + Response: Your group’s group name.

Your UDP server should run on port 4501.

### Thread Device

Your device will join the network using the same network key as the lab. Your device must be configured to be a Sleepy End Device (SED) **not** a router or REED. You will probably want to adjust the SED poll period to make it shorter so the device checks in with the router more frequently. Here’s some documentation on [Sleepy End Device types in Thread](https://docs.nordicsemi.com/bundle/ncs-2.9.0/page/nrf/protocols/thread/sed_ssed.html). Note that the device must be a Minimal Thread Device (MTD) first. We’ve already set this up for you in the starter code in the prj.conf file.

Your device must automatically connect when powered on, and require no manual configuration commands (i.e. you cannot rely on the CLI, although you may use it for testing).

The network is running in CSE 3219

### Deliverables

* **TASK 1**: Your code for your UDP device.
  + A GitHub link is fine here
* **TASK 2:** Terminal output showing your UDP device working (with all three commands).
  + You can use the cli tool to send the commands if you want.
  + You will need to have both devices join the network inside CSE 3219

Tips

* The OpenThread CoAP server [example](https://github.com/nrfconnect/sdk-nrf/blob/main/samples/openthread/coap_server/src/coap_server.c) is a decent template for a starting point. The [udp example](https://github.com/nrfconnect/sdk-nrf/blob/main/samples/net/udp/src/main.c) is *not* very helpful. For UDP code, you really can use the same functions as on a mac/linux/wsl machine.
  + Here’s a pretty good example of how to write a UDP server and client: <https://www.tack.ch/unix/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.
* Receiving UDP datagrams is typically a blocking function. One way to handle this is to use a kernel worker (k\_work). You can implement a function with the signature  
   static void my\_worker(struct k\_work \*item) { }  
  and then use k\_work\_init() and k\_work\_submit().
* I suggest leaving the shell enabled so you can use the ot commands.
* You might want to develop and test with the device as a Minimal Thread Device (not a Sleepy End Device).
* Many OpenThread features are available through the configurations in prf.conf. You will want to use these. You can [search](https://developer.nordicsemi.com/nRF_Connect_SDK/doc/2.6.0/kconfig/) for relevant configs.
* Once you have a board connected to the thread network, it will store the saved settings in its dataset. You can view the dataset with ot dataset active.

Troubleshooting

* If you run into a build error about a missing “libopenthread-cli-mtd.a” file you can build it from source by adding this flag to your prj.conf:

CONFIG\_OPENTHREAD\_SOURCES=y

## 

## ~~EC1. OPTIONAL: Ping outside network~~

~~The border router supports network address translation (~~[~~NAT64~~](https://openthread.io/codelabs/openthread-border-router-nat64)~~) to support IPv6 → IPv4 connections. This allows nodes to do DNS lookups and ping IPv4 devices. This section is worth extra credit if you want to complete it.~~

* ~~Do a DNS lookup for metro.sysnet.ucsd.edu from one of your Thread devices. This is a server in the server room in Rice.~~
* ~~Ping the resulting IPv6 address (which is really just encoding an IPv4 address).~~
* ~~Do a DNS lookup for a AAAA record (IPv6 address) from one of your Thread devices. Ping this IPv6 address.~~

**~~TASK:~~** ~~Show the metro.sysnet.ucsd.edu DNS lookup.~~

**~~TASK:~~** ~~Show the ping result.~~

**~~TASK:~~** ~~Show the DNS lookup for the AAAA record and the ping attempt. Why does the ping fail?~~

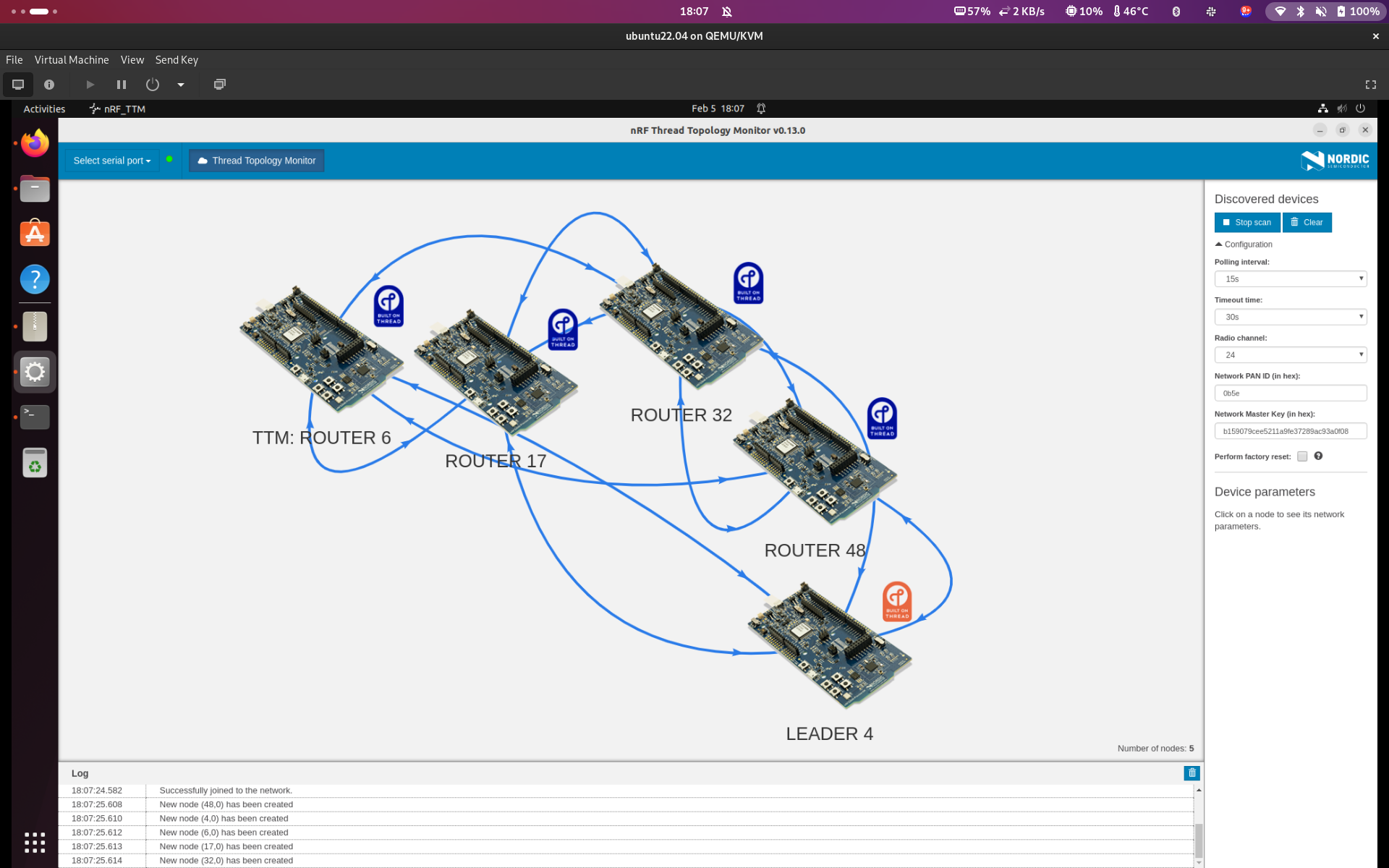
## 

## EC2. OPTIONAL: Visualize the network topology

We’ll inspect the network topology. This section is worth extra credit if you want to complete it.

* The tool to do this is the “nRF Thread Topology Monitor”: [nRF Thread topology monitor - Downloads - nordicsemi.com](https://www.nordicsemi.com/Products/Development-tools/nRF-Thread-topology-monitor/Download)

Note that this works on Windows or Linux (tested on Ubuntu, but probably others too).  
  
It does NOT work on MacOS. You will need to create a VM with Linux if everyone in your group has a Macbook. It’s not going to work at all on ARM (MacOS M1/M2) though.

* You’ll need to load the hex file to the nRF dongle. Use the hex which says 52840 and “mbr”.
* Start up the tool. When you open the tool, you’ll need to connect to your board.
* You’ll need to enter some details from the Dataset: Channel, PAN ID, and Network Key.
  + **Channel**: 24
  + **PAN ID**: 0b5e
  + **Network Key**: b159079cee5211a9fe37289ac93a0f08
* When I got this working, it took a couple of tries and some patience. It takes like 20 seconds for the tool to join your network if the parameters are right. It’s easy to mess up one of the numbers in the Network Key too. If things aren’t working: close the tool, power cycle the board, and then pull it all back up and try again.
* This is what it should look like

**TASK:** Take a screenshot of your network.

**TASK:** Change one of your devices to a different role (child or router) (e.g. ot routereligible disable). Take a screenshot of the changed topology and circle the difference.

## 

## EC3. OPTIONAL: LED Service

If you complete the previous extra credit and want more to do, implement the control device as another OpenThread node. This device must also be a SED. It must find the LED service using service discovery.

It should use the buttons to send UDP messages.

* Button 1: Send “on”
* Button 2: Send “off”
* Button 3: Send “name”

It should print all responses to the serial port.

Deliverables:

1. The additional code.
2. A video showing the button presses controlling the other board with the three commands. There is no Gradescope submission entry for the EC, so you can email it to any TA.