# BeagleConnect™ Technology

BeagleBoard.org Foundation
Oct 29, 2025

# Table of contents


**1 Overview** **3**
- 1.1 Greybus . . . 3
- 1.2 BeagleConnect™ Technology Mission . . . 4
- 1.3 Why should you use BeagleConnect™? . . . 5
- 1.4 What's next? . . . 5
- 1.5 Contributions . . . 6

**2 BeaglePlay + BeagleConnect Freedom** **7**
- 2.1 Architecture . . . 7
- 2.2 Demo . . . 8
- 2.3 Components Involved . . . 8
- 2.4 Demo . . . 8
  - 2.4.1 Using Pre-built Images . . . 8
  - 2.4.2 Build Images from Source . . . 9
- 2.5 Conclusion . . . 13

**3 BeagleBone + BeagleConnect Freedom** **15**
- 3.1 Software architecture . . . 15
- 3.2 TODO items . . . 15
- 3.3 Associated pre-work . . . 15
- 3.4 User experience concerns . . . 15
- 3.5 BeagleConnect™ Greybus demo using BeagleConnect™ Freedom . . . 17
  - 3.5.1 Introduction . . . 17
  - 3.5.2 Flash BeagleConnect™ Freedom node device with Greybus firmware . . . 17
  - 3.5.3 Trying for different add-on boards . . . 21
  - 3.5.4 Observe the node device . . . 21
  - 3.5.5 Rebuilding from source . . . 23
  - 3.5.6 Ping Pong . . . 26
  - 3.5.7 Assign a Static Address . . . 28
  - 3.5.8 Greybus . . . 28
  - 3.5.9 Blinky! . . . 29
  - 3.5.10 Read I2C Registers . . . 31
- 3.6 Conclusion . . . 31

**Important:** Currently under development

BeagleConnect™ is a revolutionary technology virtually eliminating low-level software development for IoT and IIoT applications, such as building automation, factory automation, home automation, and scientific data acquisition.

While numerous IoT and IIoT solutions available today provide massive software libraries for microcontrollers supporting a limited body of sensors, actuators and indicators as well as libraries for communicating over various networks, BeagleConnect™ simply eliminates the need for these libraries by shifting the burden into the most massive and collaborative software project of all time, the Linux kernel.

These are the tools used to automate things in scientific data collection, data science, mechatronics, and IoT.

BeagleConnect™ technology solves:

- The need to write software to add a large set of diverse devices to your system,
- The need to maintain the software with security updates,
- The need to rapidly prototype using off-the-shelf software and hardware without wiring,
- The need to connect to devices using long-range, low-power wireless, and
- The need to produce high-volume custom hardware cost-optimized for your requirements.

# Chapter 1

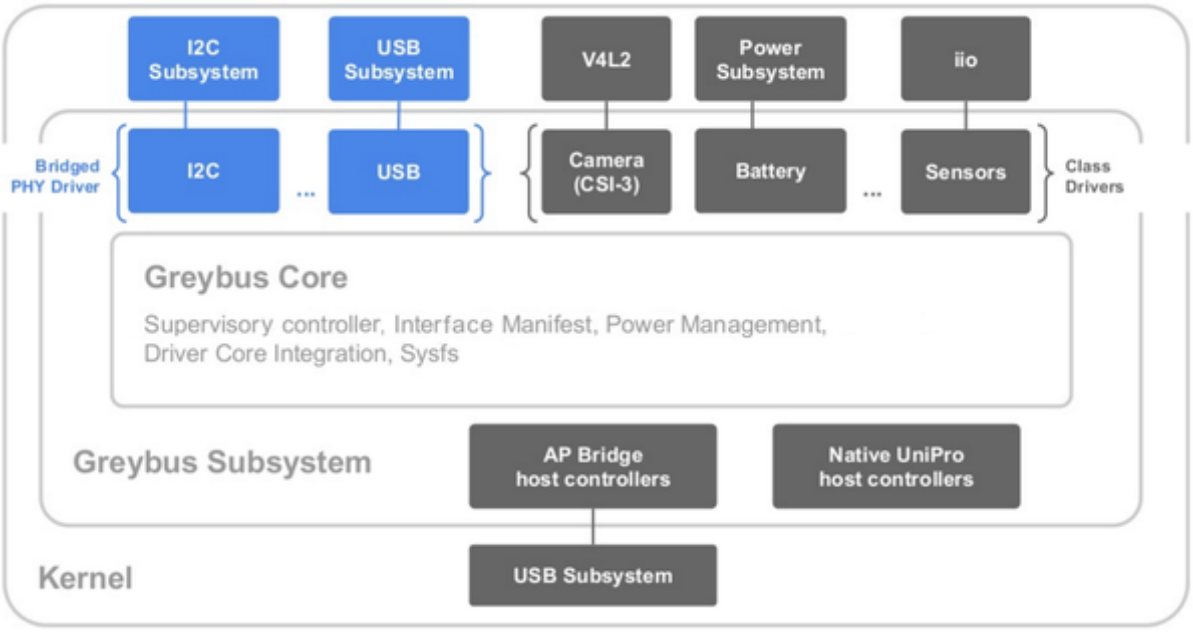
# Overview

## 1.1 Greybus

I will be taking information from the Greybus LWN article. So feel free to check it out.

Greybus was initially designed for Google's Project Ara smartphone (which is discontinued now), but the first (and only) product released with it is Motorola's Moto Mods. It was initially merged for potential use by kernel components that need to communicate in a platform-independent way.

The Greybus specification provides device discovery and description at runtime, network routing and housekeeping, and class and bridged PHY protocols, which devices use to talk to each other and to the processors. The following figure shows how various parts of the kernel interact with the Greybus subsystem.




There are three main entities in the Greybus network:

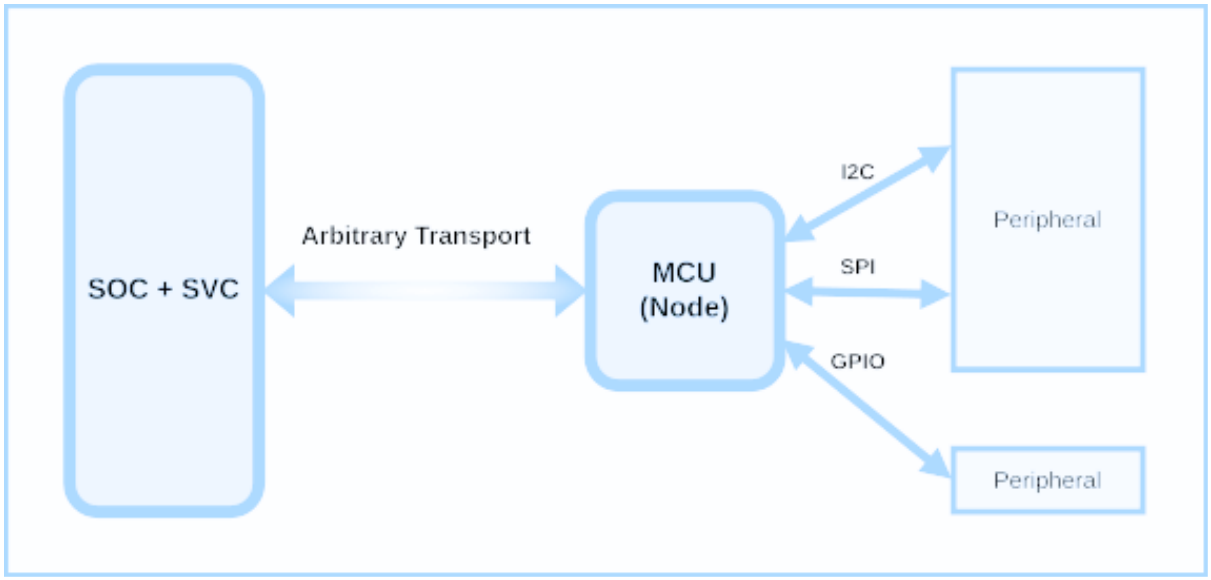
1. **AP:** It refers to the host CPUs, i.e., CPUs running Linux in most cases. It is responsible for administrating the Greybus network via the SVC.
2. **SVC:** The SVC represents an entity within the Greybus network that configures and controls the Greybus (UniPro) network, mostly based on the instructions from the AP. All module insertion and removal events are first reported to the SVC, which in turn informs the AP about them using the SVC protocol.
3. **Module:** A module is the physical hardware entity that can be connected or disconnected statically (before powering the system on) or dynamically (while the system is running) from the Greybus network.

Once the modules are connected to the Greybus network, the AP and the SVC enumerate the modules and fetch per-interface manifests to learn about their capabilities.

While Greybus is a great protocol, the implementation is tightly coupled with the UniPro transport. This makes it challenging to use Greybus in other modes of transport.

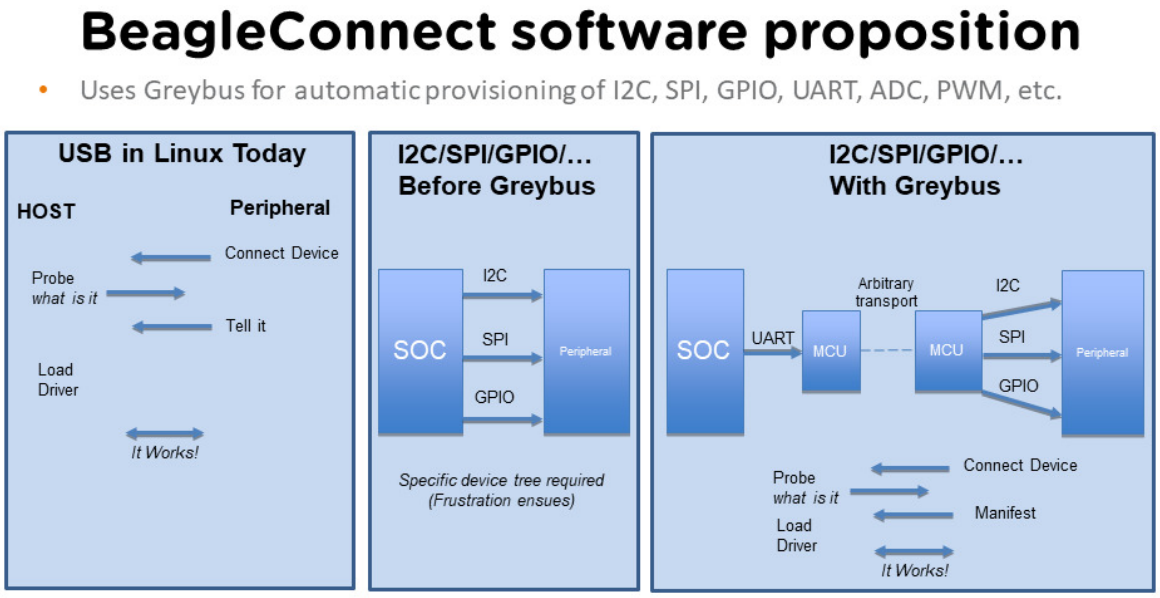
## 1.2 BeagleConnect™ Technology Mission

BeagleConnect™ Technology aims to use Greybus outside of the traditional Greybus network. This includes using transports other than UniPro (such as 6lowpan), using embedded devices running Zephyr RTOS as modules, emulating SVC in co-processor, etc. This makes BeagleConnect™ much more flexible than what traditional greybus seems to support. Here is a diagram of the general BeagleConnect™ setup:




The SVC is either emulated in userspace software in the SOC (gbridge) or in a co-processor (e.g., in BeaglePlay). The arbitrary transport can be anything from 6lowpan (for long range) to ethernet or optical cables (for max speed). Finally, greybus nodes such as BeagleConnect™ Freedom running Greybus Zephyr firmware allow the use of mikroBUS which opens a host of Plug and Play possibilities for peripherals.

## 1.3 Why should you use BeagleConnect™?




1. **Open-source:** The Greybus Spec is open-source and a part of the Linux kernel. This makes it easy to use and personalize for your use case. Being part of the Linux Kernel also provides it a level of reliability that most similar solutions lack.
2. **Network agnostic:** BeagleConnect™ allows Greybus to be network agnostic. This means it can be used over networks like 6lowpan, which has incredible wireless range, or over optical networks for high-throughput, low-latency use cases.
3. **Rapid Prototyping:** Any device (e.g., mikroBUS add-on boards) connected to the greybus node can be accessed from the Linux host. In this setup, only the Linux host needs to have device drivers. We remove the need to write drivers for the OS our node (the device with which peripheral is actually connected) runs on (e.g. Zephyr RTOS Project, Nuttx, etc). This allows being able to prototype devices by just creating a Linux driver instead of having to write drivers for each individual embedded OS.
4. **Star topology IoT and IIoT networks:** Greybus was designed to be low level and allow hot-plugging of remote devices. This means a greybus network does not need to use bulky protocols like REST and data formats like JSON. This in turn allows using relatively low-powered device as nodes.
5. **Use of Existing Infrastructure:** Since BeagleConnect™ devices show up as normal Linux devices, they work with existing local device management software. This eliminates need for propritory and custom solutions to monitor devices. Instead Linux host can directly read peripherals on nodes using standard Linux tools such as iio_readdev.
6. **Infinite Customization:** With support for mikroBUS add-on boards, capabilities of BeagleConnect™ nodes can be expanded dramaticically with little to no fiddling.

---

**Note:** The above is just a glimpse of what BeagleConnect™ can do. Many more use cases can be explored. If you have any ideas, feel free to reach out to us.

---

## 1.4 What's next?

BeagleConnect™ is still in its early stages. We are working on making it more robust and easy to use. We are trying to provide a complete experience for testing BeagleConnect™ Technology in our BeaglePlay and

BeagleConnect™ boards.

We are looking for more people to join us in improving BeagleConnect™ technology. Feel free to reach out to us at Discord or BeagleBoard Forum.

## 1.5 Contributions

- Greybus LWN article

# Chapter 2

# BeaglePlay + BeagleConnect Freedom

BeaglePlay and BeagleConnect Freedom are the first boards with the aim to provide seamless BeagleConnect™ Technology support over 6lowpan network which can have a range upto 1km. The support for mikroBUS add-on boards on BeagleConnect Freedom provide endless possibilties of peripherals. Let us go over some of the internal details that might be useful for developers who would like to get involved.

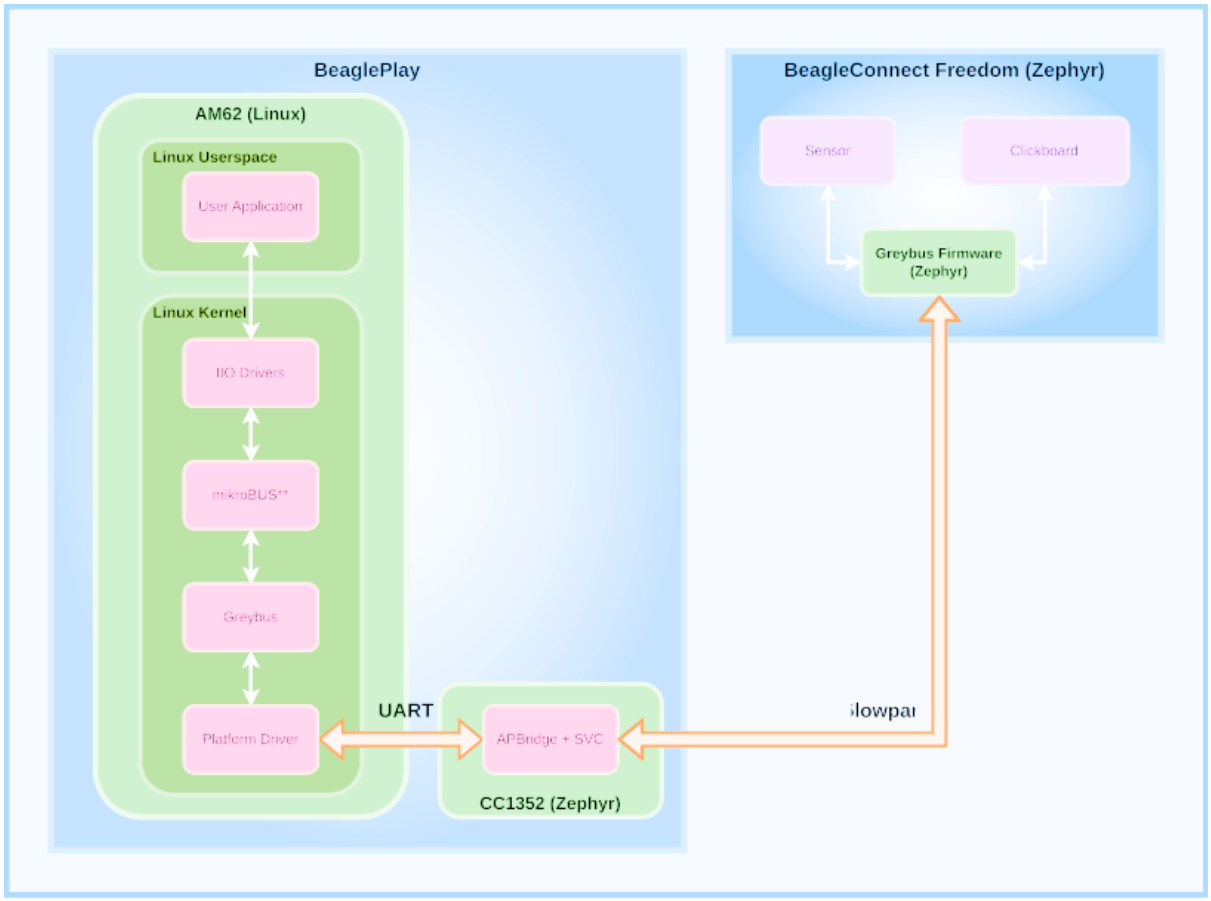
## 2.1 Architecture

**Note:** This section assumes that you are familiar with terminology introduced in *Overview*.

BeaglePlay single-board computer contains 2 processors, an AM62x running Debian Linux and a CC1352P7 co-processor. The AM62x processor acts as the AP in Greybus architecture while CC1352P7 acts as the SVC. The sub-1 ghz networking present in CC1352P7 is used as transport. This means all greybus messages between AP and Node are routed through CC1352P7.

BeagleConnect Freedom serves as the Greybus module, running ZephyrRTOS. It has 2 *mikroBUS* ports which enables compatibility with over 1,000 mikroBUS add-on sensors, acutators, indicators and additional connectivity and storage options.

Here is a visual representation of the architecture:




## 2.2 Demo

**Important:** The current setup is in heavy development. In case of any problems feel free to reach out to us at Discord or BeagleBoard Forum.

greybus-host

Here is a video of BeaglePlay + BeagleConnect Freedom in action:

## 2.3 Components Involved

- gb-beagleplay Linux Driver: Mainline Linux Kernel since v6.7.0
- mikroBUS Linux Driver: Out of tree
- greybus-node-firmware: Out of tree
- greybus-host-firmware: Out of tree

## 2.4 Demo

### 2.4.1 Using Pre-built Images

The pre-built images for both BeaglePlay and BeagleConnect Freedom are available at here.

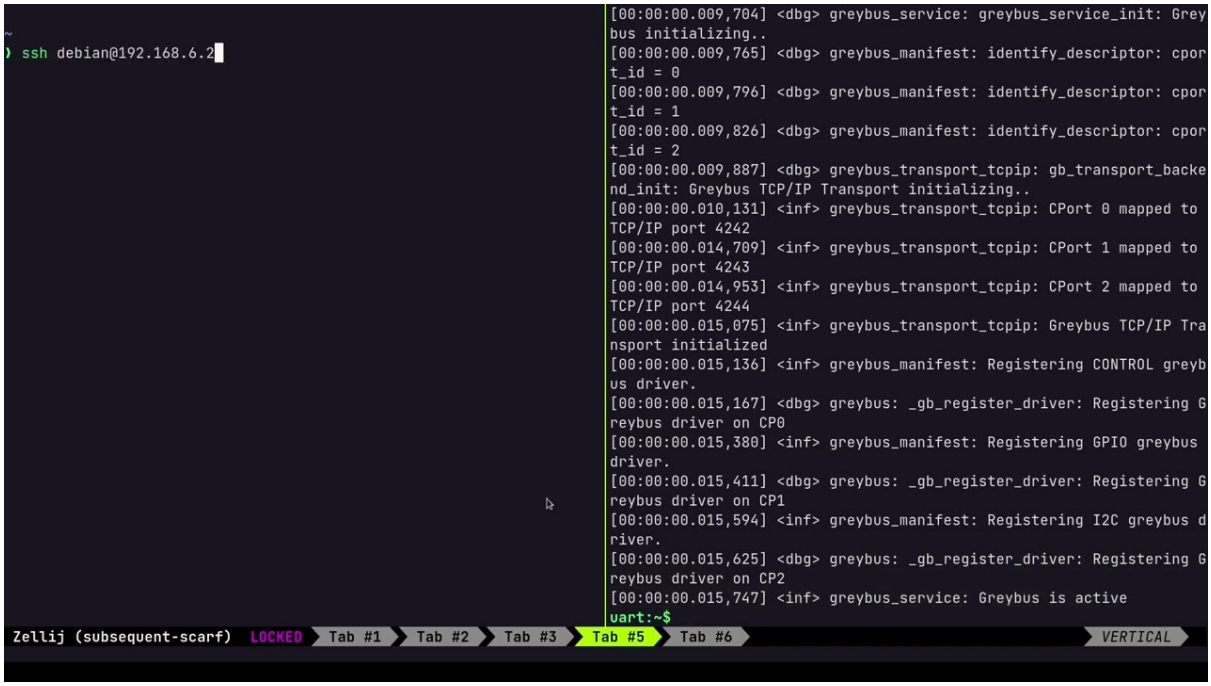



Fig. 2.1: https://youtu.be/O5coD55JvGU

### 2.4.2 Build Images from Source

**Note:** The following steps are for building the images from source. If you want to use pre-built images, you can skip this section.

**Todo:** Use upstream Zephyr. The current support in Zephyr upstream has some performance problems which are being worked on. For now, we are using a custom fork based on Zephyr v3.4

#### Setup Zephyr

**Note:** Checkout Zephyr Getting Started Guide for more up to date instructions.

1. Install the required packages:

```
sudo apt install --no-install-recommends git cmake ninja-build gperf \
  ccache dfu-util device-tree-compiler wget \
  python3-dev python3-pip python3-setuptools python3-tk python3-wheel xz-
↪utils file \
  make gcc gcc-multilib g++-multilib libsdl2-dev libmagic1 python3-venv
```

2. Create a new virtual environment:

```
python3 -m venv ~/zephyrproject/.venv
```

3. Activate the virtual environment:

```
source ~/zephyrproject/.venv/bin/activate
```

4. Install west:

```
pip install west
```

5. Get the Zephyr source code:

```
west init -m https://openbeagle.org/ayush1325/zephyr.git --mr demo-new ~/
↪zephyrproject
cd ~/zephyrproject
west update
```

6. Export a Zephyr CMake package. This allows CMake to automatically load boilerplate code required for building Zephyr applications.

```
west zephyr-export
```

7. Zephyr's scripts/requirements.txt file declares additional Python dependencies. Install them with pip.

```
pip install -r ~/zephyrproject/zephyr/scripts/requirements.txt
```

8. Download and verify the Zephyr SDK bundle:

```
cd ~/.local/opt
wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v0.
↪16.5-1/zephyr-sdk-0.16.5-1_linux-x86_64.tar.xz
wget -O - https://github.com/zephyrproject-rtos/sdk-ng/releases/download/
↪v0.16.5-1/sha256.sum | shasum --check --ignore-missing
```

9. Extract the Zephyr SDK bundle archive:

```
tar xf zephyr-sdk-0.16.5-1_linux-x86_64.tar.xz
rm zephyr-sdk-0.16.5-1_linux-x86_64.tar.xz
```

---

**Note:** If trying to build on BeaglePlay, use *zephyr-sdk-0.16.5-1_linux-aarch64_minimal.tar.xz* instead of full Zephyr SDK.

---

1. Run the Zephyr SDK bundle setup script:

```
cd zephyr-sdk-0.16.5-1
./setup.sh
```

2. Install udev rules, which allow you to flash most Zephyr boards as a regular user:

```
sudo cp ~/zephyr-sdk-0.16.5-1/sysroots/x86_64-pokysdk-linux/usr/share/
↪openocd/contrib/60-openocd.rules /etc/udev/rules.d
sudo udevadm control --reload
```

3. Install cc1352-flasher

```
pip install cc1352-flasher
```

### Build and Flash BeagleConnect Freedom

1. Build Greybus for node

```
west build -b beagleconnect_freedom modules/greybus/samples/subsys/
↪greybus/net/ -p -- -DOVERLAY_CONFIG=overlay-802154-subg.conf
```

2. Connect Beagleconnect Freedom and flash the firmware

```
west flash
```

### Build and Flash BeaglePlay CC1352

1. Build Greybus for host

```
west build -b beagleplay_cc1352 modules/greybus-host/ -p
```

2. Start BeaglePlay with bcfserial overlay. If you are using USB to UART cable to connect to BeaglePlay, you can select *BeaglePlay eMMC disable BCFSERIAL* option. Else run the following command and reboot.

```
sed -i '5d' /boot/firmware/extlinux/extlinux.conf
sed -i '5idefault BeaglePlay eMMC disable BCFSERIAL' temp2
```

3. Copy compiled image to BeaglePlay:

```
scp build/zephyr/zephyr.bin debian@beagleplay.local:~/greybus/zephyr/
↪zephyr.bin
```

4. Install cc1352-flasher on BeaglePlay

```
pip install cc1352-flasher
```

5. Flash the firmware

```
cc1352-flasher --play ~/greybus
```

6. Enable bcfserial overlay. (Skip this step if you used Uboot menu in step 2):

```
sed -i '5d' /boot/firmware/extlinux/extlinux.conf
sed -i '5idefault BeaglePlay eMMC (default)' temp2
```

7. Blacklist bcfserial Linux driver. This is required only in 5.x kernels:

```
sed -i '28s/$/ modprobe.blacklist=mikrobus/' /boot/firmware/extlinux/
↪extlinux.conf
```

8. Reboot

### BeaglePlay Driver

**Note:** This section is only required for 5.x kernels.

1. Clone the driver:

```
git clone https://git.beagleboard.org/gsoc/greybus/beagleplay-greybus-
↪driver.git
cd beagleplay-greybus-driver
```

2. Install kernel headers:

```
sudo apt install linux-headers-$(uname -r)
```

3. Build *gb-beagleplay* driver:

```
debian@BeaglePlay:~/beagleplay-greybus-driver$ make
make -C /lib/modules/5.10.168-ti-arm64-r111/build M=/home/debian/
↪beagleplay-greybus-driver modules
make[1]: Entering directory '/usr/src/linux-headers-5.10.168-ti-arm64-
↪r111'
  CC [M]  /home/debian/beagleplay-greybus-driver/gb-beagleplay.o
  MODPOST /home/debian/beagleplay-greybus-driver/Module.symvers
  CC [M]  /home/debian/beagleplay-greybus-driver/gb-beagleplay.mod.o
  LD [M]  /home/debian/beagleplay-greybus-driver/gb-beagleplay.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.10.168-ti-arm64-r111
↪'
```

4. Load the driver:

```
sudo insmod gb-beagleplay.ko
```

5. Check *iio_info*. Sensors from beagleconnect freedom should show up here:

```
debian@BeaglePlay:~$ iio_info
Library version: 0.24 (git tag: v0.24)
Compiled with backends: local xml ip usb
IIO context created with local backend.
Backend version: 0.24 (git tag: v0.24)
Backend description string: Linux BeaglePlay 5.10.168-ti-arm64-r111
↪#1bullseye SMP Tue Sep 26 14:22:20 UTC 2023 aarch64
IIO context has 2 attributes:
        local,kernel: 5.10.168-ti-arm64-r111
        uri: local:
IIO context has 2 devices:
        iio:device0: adc102s051
                2 channels found:
                        voltage1:  (input)
                        2 channel-specific attributes found:
                                attr  0: raw value: 4068
                                attr  1: scale value: 0.805664062
                        voltage0:  (input)
                        2 channel-specific attributes found:
                                attr  0: raw value: 0
                                attr  1: scale value: 0.805664062
                No trigger on this device
        iio:device1: hdc2010
                3 channels found:
                        temp:  (input)
                        4 channel-specific attributes found:
                                attr  0: offset value: -15887.515151
                                attr  1: peak_raw value: 28928
                                attr  2: raw value: 28990
                                attr  3: scale value: 2.517700195
                        humidityrelative:  (input)
                        3 channel-specific attributes found:
                                attr  0: peak_raw value: 43264
                                attr  1: raw value: 41892
                                attr  2: scale value: 1.525878906
                        current:  (output)
                        2 channel-specific attributes found:
                                attr  0: heater_raw value: 0
                                attr  1: heater_raw_available value: 0 1
                No trigger on this device
```

## 2.5 Conclusion

While BeagleConnect™ technology is still in development, we are excited to see the possibilities it brings to the table. We are continuously working on improving the technology and adding more features. Fee free to reach out to us at Discord or BeagleBoard Forum.

# Chapter 3

# BeagleBone + BeagleConnect Freedom

**Important:** This demo was the old way of doing things. Anyone new and inexperienced should look at beagleplay + beagleconnect freedom instead.

## 3.1 Software architecture

## 3.2 TODO items

**Todo:** Click Board plug-ins for node-red for the same 100 or so Click Boards

**Todo:** BeagleConnect™ Freedom System Reference Manual and FAQs

## 3.3 Associated pre-work

- Click Board support for Node-RED can be executed with native connections on PocketBeagle+TechLab and BeagleBone Black with mikroBUS Cape
- Device tree fragments and driver updates can be provided via https://bbb.io/click
- The Kconfig style provisioning can be implemented for those solutions, which will require a reboot. We need to centralize edits to /boot/uEnv.txt to be programmatic. As I think through this, I don't think BeagleConnect is impacted, because the Greybus-style discovery along with Click EEPROMS will eliminate any need to edit /boot/uEnv.txt.

## 3.4 User experience concerns

- Make sure no reboots are required
- Plugging BeagleConnect into host should trigger host configuration
- Click EEPROMs should trigger loading whatever drivers are needed and provisioning should load any new drivers
- Userspace (spidev, etc.) drivers should unload cleanly when 2nd phase provisioning is completed

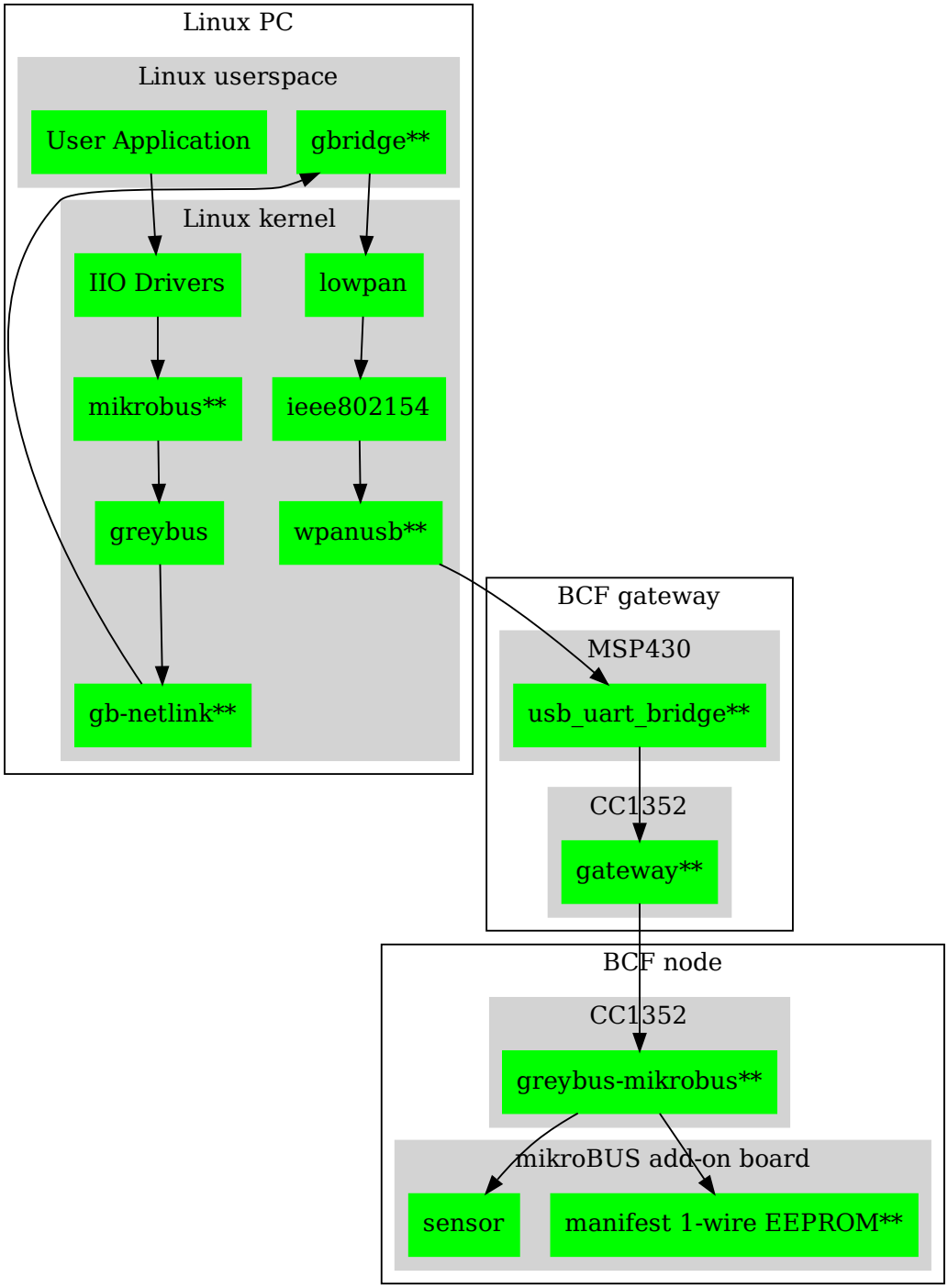



Fig. 3.1: BeagleConnect Software Architecture Diagram

## 3.5 BeagleConnect™ Greybus demo using BeagleConnect™ Freedom

BeagleConnect™ Freedom runs a subGHz IEEE 802.15.4 network. This BeagleConnect™ Greybus demo shows how to interact with GPIO, I2C and mikroBUS add-on boards remotely connected over a BeagleConnect™ Freedom.

This section starts with the steps required to use Linux embedded computer (BeagleBone Green Gateway) and the Greybus protocol, over an IEEE 802.15.4 wireless link, to blink an LED on a Zephyr device.

### 3.5.1 Introduction

*Why??*

Good question. Blinking an LED is kind of the Hello, World of the hardware community. In this case, we're less interested in the mechanics of switching a GPIO to drive some current through an LED and more interested in how that happens with the Internet of Things (IoT).

There are several existing network and application layers that are driven by corporate heavyweights and industry consortiums, but relatively few that are community driven and, more specifically, even fewer that have the ability to integrate so tightly with the Linux kernel.

The goal here is to provide a community-maintained, developer-friendly, and open-source protocol for the Internet of Things using the Greybus Protocol, and blinking an LED using Greybus is the simplest proof-of-concept for that. All that is required is a reliable transport.

1. Power a BeagleConnect Freedom that has not yet been programmed via a USB power source, not the BeagleBone Green Gateway. You'll hear a click every 1-2 seconds along with seeing 4 of the LEDs turn off and on.
2. In an isolated terminal window, `sudo beagleconnect-start-gateway`
3. `sensortest-rx.py`

Every 1-2 minutes, you should see something like:

```
('fe80::3111:7a22:4b:1200%lowpan0', 52213, 0, 13)  '2l:7.79;'
('fe80::3111:7a22:4b:1200%lowpan0', 52213, 0, 13)  '4h:43.75;4t:23.11;'
```

The value after "2l:" is the amount of light in lux. The value after "4h:" is the relative humidity and after "4t:" is the temperature in Celsius.

### 3.5.2 Flash BeagleConnect™ Freedom node device with Greybus firmware

#TODO: How can we add a step in here to show the network is connected without needing gbridge to be fully functional?

Do this from the BeagleBone® Green Gateway board that was previously used to program the BeagleConnect™ Freedom gateway device:

1. Disconnect the BeagleConnect™ Freedom **gateway** device
2. Connect a new BeagleConnect™ Freedom board via USB
3. `sudo systemctl stop lowpan.service`
4. `cc2538-bsl.py /usr/share/beagleconnect/cc1352/greybus_mikrobus_beagleconnect.bin /dev/ttyACM0`
5. After it finishes programming successfully, disconnect the BeagleConnect Freedom node device
6. Power the newly programmed BeagleConnect Freedom node device from an alternate USB power source
7. Reconnect the BeagleConnect Freedom **gateway** device to the BeagleBone Green Gateway

8. sudo systemctl start lowpan.service
9. sudo beagleconnect-start-gateway

```
debian@beaglebone:~$ sudo beagleconnect-start-gateway
[sudo] password for debian:
setting up wpanusb gateway for IEEE 802154 CHANNEL 1(906 Mhz)
ping6: Warning: source address might be selected on device other than␣
↪lowpan0.
PING 2001:db8::1(2001:db8::1) from ::1 lowpan0: 56 data bytes
64 bytes from 2001:db8::1: icmp_seq=2 ttl=64 time=185 ms
64 bytes from 2001:db8::1: icmp_seq=3 ttl=64 time=40.9 ms
64 bytes from 2001:db8::1: icmp_seq=4 ttl=64 time=40.9 ms
64 bytes from 2001:db8::1: icmp_seq=5 ttl=64 time=40.6 ms

--- 2001:db8::1 ping statistics ---
5 packets transmitted, 4 received, 20% packet loss, time 36ms
rtt min/avg/max/mdev = 40.593/76.796/184.799/62.356 ms
debian@beaglebone:~$ iio_info
Library version: 0.19 (git tag: v0.19)
Compiled with backends: local xml ip usb serial
IIO context created with local backend.
Backend version: 0.19 (git tag: v0.19)
Backend description string: Linux beaglebone 5.14.18-bone20 #1buster PREEMPT␣
↪Tue Nov 16 20:47:19 UTC 2021 armv7l
IIO context has 1 attributes:
    local,kernel: 5.14.18-bone20
IIO context has 3 devices:
    iio:device0: TI-am335x-adc.0.auto (buffer capable)
        8 channels found:
            voltage0:  (input, index: 0, format: le:u12/16>>0)
            1 channel-specific attributes found:
                attr  0: raw value: 1412
            voltage1:  (input, index: 1, format: le:u12/16>>0)
            1 channel-specific attributes found:
                attr  0: raw value: 2318
            voltage2:  (input, index: 2, format: le:u12/16>>0)
            1 channel-specific attributes found:
                attr  0: raw value: 2631
            voltage3:  (input, index: 3, format: le:u12/16>>0)
            1 channel-specific attributes found:
                attr  0: raw value: 817
            voltage4:  (input, index: 4, format: le:u12/16>>0)
            1 channel-specific attributes found:
                attr  0: raw value: 881
            voltage5:  (input, index: 5, format: le:u12/16>>0)
            1 channel-specific attributes found:
                attr  0: raw value: 0
            voltage6:  (input, index: 6, format: le:u12/16>>0)
            1 channel-specific attributes found:
                attr  0: raw value: 0
            voltage7:  (input, index: 7, format: le:u12/16>>0)
            1 channel-specific attributes found:
                attr  0: raw value: 1180
        2 buffer-specific attributes found:
                attr  0: data_available value: 0
                attr  1: watermark value: 1
    iio:device1: hdc2010
        3 channels found:
            humidityrelative:  (input)
            3 channel-specific attributes found:
                attr  0: peak_raw value: 52224
                attr  1: raw value: 52234
```

(continues on next page)

(continued from previous page)

```
                attr  2: scale value: 1.525878906
            current:  (output)
            2 channel-specific attributes found:
                attr  0: heater_raw value: 0
                attr  1: heater_raw_available value: 0 1
            temp:  (input)
            4 channel-specific attributes found:
                attr  0: offset value: -15887.515151
                attr  1: peak_raw value: 25600
                attr  2: raw value: 25628
                attr  3: scale value: 2.517700195
    iio:device2: opt3001
        1 channels found:
            illuminance:  (input)
            2 channel-specific attributes found:
                attr  0: input value: 79.040000
                attr  1: integration_time value: 0.800000
        2 device-specific attributes found:
                attr  0: current_timestamp_clock value: realtime

                attr  1: integration_time_available value: 0.1 0.8
debian@beaglebone:~$ dmesg | grep -e mikrobus -e greybus
[  100.491253] greybus 1-2.2: Interface added (greybus)
[  100.491294] greybus 1-2.2: GMP VID=0x00000126, PID=0x00000126
[  100.491306] greybus 1-2.2: DDBL1 Manufacturer=0x00000126,␣
↪Product=0x00000126
[  100.737637] greybus 1-2.2: excess descriptors in interface manifest
[  102.475168] mikrobus:mikrobus_port_gb_register: mikrobus gb_probe , num␣
↪cports= 2, manifest_size 192
[  102.475206] mikrobus:mikrobus_port_gb_register: protocol added 3
[  102.475214] mikrobus:mikrobus_port_gb_register: protocol added 2
[  102.475239] mikrobus:mikrobus_port_register: registering port mikrobus-1
[  102.475400] mikrobus_manifest:mikrobus_state_get: mikrobus descriptor not␣
↪found
[  102.475417] mikrobus_manifest:mikrobus_manifest_attach_device: parsed␣
↪device 1, driver=opt3001, protocol=3, reg=44
[  102.494516] mikrobus_manifest:mikrobus_manifest_attach_device: parsed␣
↪device 2, driver=hdc2010, protocol=3, reg=41
[  102.494567] mikrobus_manifest:mikrobus_manifest_parse:  (null) manifest␣
↪parsed with 2 devices
[  102.494592] mikrobus mikrobus-1: registering device : opt3001
[  102.495096] mikrobus mikrobus-1: registering device : hdc2010
debian@beaglebone:~$
```

#TODO: update the below for the built-in sensors

#TODO: can we also handle the case where these sensors are included and recommend them? Same firmware?

#TODO: the current demo is for the built-in sensors, not the Click boards mentioned below

Currently only a limited number of add-on boards have been tested to work over Greybus, simple add-on boards without interrupt requirement are the ones that work currently. The example is for Air Quality 2 Click and Weather Click attached to the mikroBUS ports on the device side.

/var/log/gbridge will have the gbridge log, and if the mikroBUS port has been instantiated successfully the kernel log will show the devices probe messages:

#TODO: this log needs to be updated

```
greybus 1-2.2: GMP VID=0x00000126, PID=0x00000126
greybus 1-2.2: DDBL1 Manufacturer=0x00000126, Product=0x00000126
greybus 1-2.2: excess descriptors in interface manifest
mikrobus:mikrobus_port_gb_register: mikrobus gb_probe , num cports= 3,␣
```

(continues on next page)

(continued from previous page)

```
 ↪manifest_size 252
mikrobus:mikrobus_port_gb_register: protocol added 11
mikrobus:mikrobus_port_gb_register: protocol added 3
mikrobus:mikrobus_port_gb_register: protocol added 2
mikrobus:mikrobus_port_register: registering port mikrobus-0
mikrobus_manifest:mikrobus_manifest_attach_device: parsed device 1,␣
 ↪driver=bme280, protocol=3, reg=76
mikrobus_manifest:mikrobus_manifest_attach_device: parsed device 2,␣
 ↪driver=ams-iaq-core, protocol=3, reg=5a
mikrobus_manifest:mikrobus_manifest_parse:  Greybus Service Sample␣
 ↪Application manifest parsed with 2 devices
mikrobus mikrobus-0: registering device : bme280
mikrobus mikrobus-0: registering device : ams-iaq-core
```

#TODO: bring in the GPIO toggle and I2C explorations for greater understanding

### Flashing via a Linux Host

If flashing the Freedom board via the BeagleBone fails here's a trick you can try to flash from a Linux host.

Use `sshfs` to mount the Bone's files on the Linux host. This assumes the Bone is plugged in the USB and appears at `192.168.7.2`:

```
host$ cd
host$ sshfs 192.168.7.2:/ bone
host$ cd bone; ls
bin   dev  home    lib         media  opt   root  sbin  sys  usr
boot  etc  ID.txt  lost+found  mnt    proc  run   srv   tmp  var
host$ ls /dev/ttyACM*
/dev/ttyACM1
```

The Bone's files now appear as local files. Notice there is already a `/dev/ACM*` appearing. Now plug the Connect into the Linux host's USB port and run the command again.

```
host$ ls /dev/ttyACM*
/dev/ttyACM0  /dev/ttyACM1
```

The `/dev/ttyACM` that just appeared is the one associated with the Connect. In my case it's `/dev/ttyACM0`. That's what I'll use in this example.

Now change directories to where the binaries are and load:

```
host$ cd ~/bone/usr/share/beagleconnect/cc1352;ls
greybus_mikrobus_beagleconnect.bin     sensortest_beagleconnect.dts
greybus_mikrobus_beagleconnect.config  wpanusb_beagleconnect.bin
greybus_mikrobus_beagleconnect.dts     wpanusb_beagleconnect.config
sensortest_beagleconnect.bin           wpanusb_beagleconnect.dts
sensortest_beagleconnect.config

host$ ~/bone/usr/bin/cc2538-bsl.py sensortest_beagleconnect.bin /dev/ttyACM0
8-bsl.py sensortest_beagleconnect.bin /dev/ttyACM0
Opening port /dev/ttyACM0, baud 50000
Reading data from sensortest_beagleconnect.bin
Cannot auto-detect firmware filetype: Assuming .bin
Connecting to target...
CC1350 PG2.0 (7x7mm): 352KB Flash, 20KB SRAM, CCFG.BL_CONFIG at 0x00057FD8
Primary IEEE Address: 00:12:4B:00:22:7A:10:46
    Performing mass erase
Erasing all main bank flash sectors
    Erase done
Writing 360448 bytes starting at address 0x00000000
```

(continues on next page)

(continued from previous page)

```
Write 104 bytes at 0x00057F988
    Write done
Verifying by comparing CRC32 calculations.
    Verified (match: 0x0f6bdf0f)
```

Now you are ready to continue the instructions above after the cc2528 command.

### 3.5.3 Trying for different add-on boards

See mikroBUS over Greybus for trying out the same example for different mikroBUS add-on boards/ on-board devices.

### 3.5.4 Observe the node device

Connect BeagleConnect Freedom node device to an Ubuntu laptop to observe the Zephyr console.

#### Console (`tio`)

In order to see diagnostic messages or to run certain commands on the Zephyr device we will require a terminal open to the device console. In this case, we use tio due how its usage simplifies the instructions.

1. Install `tio` `sudo apt install -y tio`
2. Run `tio` `tio /dev/ttyACM0`

   To exit `tio` (later), enter `ctrl+t, q`.

#### The Zephyr Shell

After flashing, you should observe the something matching the following output in `tio`.

```
uart:~$ *** Booting Zephyr OS build 9c858c863223  ***
[00:00:00.009,735] <inf> greybus_transport_tcpip: CPort 0 mapped to TCP/IP␣
↪port 4242
[00:00:00.010,131] <inf> greybus_transport_tcpip: CPort 1 mapped to TCP/IP␣
↪port 4243
[00:00:00.010,528] <inf> greybus_transport_tcpip: CPort 2 mapped to TCP/IP␣
↪port 4244
[00:00:00.010,742] <inf> greybus_transport_tcpip: Greybus TCP/IP Transport␣
↪initialized
[00:00:00.010,864] <inf> greybus_manifest: Registering CONTROL greybus␣
↪driver.
[00:00:00.011,230] <inf> greybus_manifest: Registering GPIO greybus driver.
[00:00:00.011,596] <inf> greybus_manifest: Registering I2C greybus driver.
[00:00:00.011,871] <inf> greybus_service: Greybus is active
[00:00:00.026,092] <inf> net_config: Initializing network
[00:00:00.134,063] <inf> net_config: IPv6 address: 2001:db8::1
```

The line beginning with `***` is the Zephyr boot banner.

Lines beginning with a timestamp of the form `[H:m:s.us]` are Zephyr kernel messages.

Lines beginning with `uart:~$` indicates that the Zephyr shell is prompting you to enter a command.

From the informational messages shown, we observe the following.

- Zephyr is configured with the following link-local IPv6 address `fe80::3177:a11c:4b:1200`
- It is listening for (both) TCP and UDP traffic on port 4242

However, what the log messages do not show (which will come into play later), are 2 critical pieces of information:

1. **The RF Channel**: As you may have guessed, IEEE 802.15.4 devices are only able to communicate with each other if they are using the same frequency to transmit and receive data. This information is part of the Physical Layer.
2. The PAN identifier: IEEE 802.15.4 devices are only be able to communicate with one another if they use the same PAN ID. This permits multiple networks (PANs) on the same frequency. This information is part of the Data Link Layer.

If we type `help` in the shell and hit Enter, we're prompted with the following:

```
Please press the <Tab> button to see all available commands.
You can also use the <Tab> button to prompt or auto-complete all commands or␣
↪its subcommands.
You can try to call commands with <-h> or <--help> parameter for more␣
↪information.
Shell supports following meta-keys:

Ctrl+a, Ctrl+b, Ctrl+c, Ctrl+d, Ctrl+e, Ctrl+f, Ctrl+k, Ctrl+l, Ctrl+n,␣
↪Ctrl+p, Ctrl+u, Ctrl+w
Alt+b, Alt+f.
Please refer to shell documentation for more details.
```

So after hitting Tab, we see that there are several interesting commands we can use for additional information.

```
uart:~$
clear       help        history     ieee802154  log         net
resize      sample      shell
```

### Zephyr Shell: IEEE 802.15.4 commands

Entering `ieee802154 help`, we see

```
uart:~$ ieee802154 help
ieee802154 - IEEE 802.15.4 commands
Subcommands:
ack              :<set/1 | unset/0> Set auto-ack flag
associate        :<pan_id> <PAN coordinator short or long address (EUI-64)>
disassociate     :Disassociate from network
get_chan         :Get currently used channel
get_ext_addr     :Get currently used extended address
get_pan_id       :Get currently used PAN id
get_short_addr   :Get currently used short address
get_tx_power     :Get currently used TX power
scan             :<passive|active> <channels set n[:m:...]:x|all> <per-channel
                 duration in ms>
set_chan         :<channel> Set used channel
set_ext_addr     :<long/extended address (EUI-64)> Set extended address
set_pan_id       :<pan_id> Set used PAN id
set_short_addr   :<short address> Set short address
set_tx_power     :<-18/-7/-4/-2/0/1/2/3/5> Set TX power
```

We get the missing Channel number (frequency) with the command `ieee802154 get_chan`.

```
uart:~$ ieee802154 get_chan
Channel 26
```

We get the missing PAN ID with the command `ieee802154 get_pan_id`.

```
uart:~$ ieee802154 get_pan_id
PAN ID 43981 (0xabcd)
```

#### Zephyr Shell: Network Commands

Additionally, we may query the IPv6 information of the Zephyr device.

```
uart:~$ net iface

Interface 0x20002b20 (IEEE 802.15.4) [1]
========================================
Link addr : CD:99:A1:1C:00:4B:12:00
MTU       : 125
IPv6 unicast addresses (max 3):
        fe80::cf99:a11c:4b:1200 autoconf preferred infinite
        2001:db8::1 manual preferred infinite
IPv6 multicast addresses (max 4):
        ff02::1
        ff02::1:ff4b:1200
        ff02::1:ff00:1
IPv6 prefixes (max 2):
        <none>
IPv6 hop limit           : 64
IPv6 base reachable time : 30000
IPv6 reachable time      : 16929
IPv6 retransmit timer    : 0
```

And we see that the static IPv6 address (`2001:db8::1`) from `samples/net/sockets/echo_server/prj.conf` is present and configured. While the statically configured IPv6 address is useful, it isn't 100% necessary.

### 3.5.5 Rebuilding from source

#TODO: revisit everything below here

#### Prerequisites

- Zephyr environment is set up according to the Getting Started Guide
  - Please use the Zephyr SDK when installing a toolchain above
- Zephyr SDK is installed at ~/zephyr-sdk-0.11.2 (any later version should be fine as well)
- Zephyr board is connected via USB

#### Cloning the repository

This repository utilizes git submodules to keep track of all of the projects required to reproduce the ongoing work. The instructions here only cover checking out the `demo` branch which should stay in a tested state. ongoing development will be on the `master` branch.

Note: The parent directory ~ is simply used as a placeholder for testing. Please use whatever parent directory you see fit.

#### Clone specific tag

```
cd ~
git clone --recurse-submodules --branch demo https://github.com/jadonk/
↪beagleconnect
```

### Zephyr

**Add the Fork** For the time being, Greybus must remain outside of the main Zephyr repository. Currently, it is just in a Zephyr fork, but it should be converted to a proper Module (External Project). This is for a number of reasons, but mainly there must be:

- specifications for authentication and encryption
- specifications for joining and rejoining wireless networks
- specifications for discovery

Therefore, in order to reproduce this example, please run the following.

```
cd ~/beagleconnect/sw/zephyrproject/zephyr
west update
```

**Build and Flash Zephyr** Here, we will build and flash the Zephyr greybus_net sample to our device.

1. Edit the file `~/.zephyrrc` and place the following text inside of it

```
export ZEPHYR_TOOLCHAIN_VARIANT=zephyr
export ZEPHYR_SDK_INSTALL_DIR=~/zephyr-sdk-0.11.2
```

1. Set up the required Zephyr environment variables via

```
source zephyr-env.sh
```

1. Build the project

```
BOARD=cc1352r1_launchxl west build samples/subsys/greybus/net --pristine \
--build-dir build/greybus_launchpad -- -DCONF_FILE="prj.conf overlay-802154.
↪conf"
```

1. Ensure that the last part of the build process looks somewhat like this:

```
...
[221/226] Linking C executable zephyr/zephyr_prebuilt.elf
Memory region         Used Size  Region Size  %age Used
        FLASH:      155760 B     360360 B     43.22%
    FLASH_CCFG:          88 B         88 B    100.00%
            SRAM:       58496 B         80 KB     71.41%
        IDT_LIST:         184 B          2 KB      8.98%
[226/226] Linking C executable zephyr/zephyr.elf
```

1. Flash the firmware to your device using

```
BOARD=cc1352r1_launchxl west flash --build-dir build/greybus_launchpad
```

### Linux

Warning: If you aren't comfortable building and installing a Linux kernel on your computer, you should probably just stop here. I'll assume you know the basics of building and installing a Linux kernel from here on out.

**Clone, patch, and build the kernel** For this demo, I used the 5.8.4 stable kernel. Also, I've applied the `mikrobus` kernel driver, though it isn't strictly required for greybus.

Note: The parent directory ~ is simply used as a placeholder for testing. Please use whatever parent directory you see fit.

TODO: The patches for gb-netlink will eventually be applied here until pushed into mainline.

```
cd ~
git clone --branch v5.8.4 --single-branch git://git.kernel.org/pub/scm/linux/
↪kernel/git/stable/linux.git
cd linux
git checkout -b v5.8.4-greybus
git am ~/beagleconnect/sw/linux/v2-0001-RFC-mikroBUS-driver-for-add-on-
↪boards.patch
git am ~/beagleconnect/sw/linux/0001-mikroBUS-build-fixes.patch
cp /boot/config-`uname -r` .config
yes "" | make oldconfig
./scripts/kconfig/merge_config.sh .config ~/beagleconnect/sw/linux/mikrobus.
↪config
./scripts/kconfig/merge_config.sh .config ~/beagleconnect/sw/linux/atusb.
↪config
make -j`nproc --all`
sudo make modules_install
sudo make install
```

Reboot and select your new kernel.

**Probe the IEEE 802.15.4 Device Driver** On the Linux machine, make sure the `atusb` driver is loaded. This should happen automatically when the adapter is inserted or when the machine is booted while the adapter is installed.

```
$ dmesg | grep -i ATUSB
[    6.512154] usb 1-1: ATUSB: AT86RF231 version 2
[    6.512492] usb 1-1: Firmware: major: 0, minor: 3, hardware type: ATUSB␣
↪(2)
[    6.525357] usbcore: registered new interface driver atusb
...
```

We should now be able to see the IEEE 802.15.4 network device by entering `ip a show wpan0`.

```
$ ip a show wpan0
36: wpan0: <BROADCAST,NOARP,UP,LOWER_UP> mtu 123 qdisc fq_codel state␣
↪UNKNOWN group default qlen 300
    link/ieee802.15.4 3e:7d:90:4d:8f:00:76:a2 brd ff:ff:ff:ff:ff:ff:ff:ff
```

But wait, that is not an IP address! It's the hardware address of the 802.15.4 device. So, in order to associate it with an IP address, we need to run a couple of other commands (thanks to wpan.cakelab.org).

### Set the 802.15.4 Physical and Link-Layer Parameters

1. First, get the phy number for the `wpan0` device

```
$ iwpan list
    wpan_phy phy0
    supported channels:
        page 0: 11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26
    current_page: 0
    current_channel: 26,  2480 MHz
    cca_mode: (1) Energy above threshold
    cca_ed_level: -77
```

(continues on next page)

(continued from previous page)

```
    tx_power: 3
    capabilities:
        iftypes: node,monitor
        channels:
            page 0:
                [11]  2405 MHz, [12]  2410 MHz, [13]  2415 MHz,
                [14]  2420 MHz, [15]  2425 MHz, [16]  2430 MHz,
                [17]  2435 MHz, [18]  2440 MHz, [19]  2445 MHz,
                [20]  2450 MHz, [21]  2455 MHz, [22]  2460 MHz,
                [23]  2465 MHz, [24]  2470 MHz, [25]  2475 MHz,
                [26]  2480 MHz
        tx_powers:
                3 dBm, 2.8 dBm, 2.3 dBm, 1.8 dBm, 1.3 dBm, 0.7 dBm,
                0 dBm, -1 dBm, -2 dBm, -3 dBm, -4 dBm, -5 dBm,
                -7 dBm, -9 dBm, -12 dBm, -17 dBm,
        cca_ed_levels:
                -91 dBm, -89 dBm, -87 dBm, -85 dBm, -83 dBm, -81 dBm,
                -79 dBm, -77 dBm, -75 dBm, -73 dBm, -71 dBm, -69 dBm,
                -67 dBm, -65 dBm, -63 dBm, -61 dBm,
        cca_modes:
            (1) Energy above threshold
            (2) Carrier sense only
            (3, cca_opt: 0) Carrier sense with energy above threshold␣
↪(logical operator is 'and')
            (3, cca_opt: 1) Carrier sense with energy above threshold␣
↪(logical operator is 'or')
        min_be: 0,1,2,3,4,5,6,7,8
        max_be: 3,4,5,6,7,8
        csma_backoffs: 0,1,2,3,4,5
        frame_retries: 3
        lbt: false
```

1. Next, set the Channel for the 802.15.4 device on the Linux machine

```
sudo iwpan phy phy0 set channel 0 26
```

1. Then, set the PAN identifier for the 802.15.4 device on the Linux machine `sudo iwpan dev wpan0 set pan_id 0xabcd`
2. Associate the `wpan0` device to a new, 6lowpan network interface

```
sudo ip link add link wpan0 name lowpan0 type lowpan
```

1. Finally, set the links up for both `wpan0` and `lowpan0`

```
sudo ip link set wpan0 up
sudo ip link set lowpan0 up
```

We should observe something like the following when we run `ip a show lowpan0`.

```
ip a show lowpan0
37: lowpan0@wpan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1280 qdisc noqueue␣
↪state UNKNOWN group default qlen 1000
    link/6lowpan 9e:0b:a4:e8:00:d3:45:53 brd ff:ff:ff:ff:ff:ff:ff:ff
    inet6 fe80::9c0b:a4e8:d3:4553/64 scope link
    valid_lft forever preferred_lft forever
```

### 3.5.6 Ping Pong

### Broadcast Ping

Now, perform a broadcast ping to see what else is listening on `lowpan0`.

```
$ ping6 -I lowpan0 ff02::1
PING ff02::1(ff02::1) from fe80::9c0b:a4e8:d3:4553%lowpan0 lowpan0: 56 data␣
↪bytes
64 bytes from fe80::9c0b:a4e8:d3:4553%lowpan0: icmp_seq=1 ttl=64 time=0.099␣
↪ms
64 bytes from fe80::9c0b:a4e8:d3:4553%lowpan0: icmp_seq=2 ttl=64 time=0.125␣
↪ms
64 bytes from fe80::cf99:a11c:4b:1200%lowpan0: icmp_seq=2 ttl=64 time=17.3␣
↪ms (DUP!)
64 bytes from fe80::9c0b:a4e8:d3:4553%lowpan0: icmp_seq=3 ttl=64 time=0.126␣
↪ms
64 bytes from fe80::cf99:a11c:4b:1200%lowpan0: icmp_seq=3 ttl=64 time=9.60␣
↪ms (DUP!)
64 bytes from fe80::9c0b:a4e8:d3:4553%lowpan0: icmp_seq=4 ttl=64 time=0.131␣
↪ms
64 bytes from fe80::cf99:a11c:4b:1200%lowpan0: icmp_seq=4 ttl=64 time=14.9␣
↪ms (DUP!)
```

Yay! We have pinged (pung?) the Zephyr device over IEEE 802.15.4 using 6LowPAN!

### Ping Zephyr

We can ping the Zephyr device directly without a broadcast ping too, of course.

```
$ ping6 -I lowpan0 fe80::cf99:a11c:4b:1200
PING fe80::cf99:a11c:4b:1200(fe80::cf99:a11c:4b:1200) from␣
↪fe80::9c0b:a4e8:d3:4553%lowpan0 lowpan0: 56 data bytes
64 bytes from fe80::cf99:a11c:4b:1200%lowpan0: icmp_seq=1 ttl=64 time=16.0 ms
64 bytes from fe80::cf99:a11c:4b:1200%lowpan0: icmp_seq=2 ttl=64 time=13.8 ms
64 bytes from fe80::cf99:a11c:4b:1200%lowpan0: icmp_seq=3 ttl=64 time=9.77 ms
64 bytes from fe80::cf99:a11c:4b:1200%lowpan0: icmp_seq=5 ttl=64 time=11.5 ms
```

### Ping Linux

Similarly, we can ping the Linux host from the Zephyr shell.

```
uart:~$ net ping --help
ping - Ping a network host.
Subcommands:
--help  :'net ping [-c count] [-i interval ms] <host>' Send ICMPv4 or ICMPv6
        Echo-Request to a network host.
$ net ping -c 5 fe80::9c0b:a4e8:d3:4553
PING fe80::9c0b:a4e8:d3:4553
8 bytes from fe80::9c0b:a4e8:d3:4553 to fe80::cf99:a11c:4b:1200: icmp_seq=0␣
↪ttl=64 rssi=110 time=11 ms
8 bytes from fe80::9c0b:a4e8:d3:4553 to fe80::cf99:a11c:4b:1200: icmp_seq=1␣
↪ttl=64 rssi=126 time=9 ms
8 bytes from fe80::9c0b:a4e8:d3:4553 to fe80::cf99:a11c:4b:1200: icmp_seq=2␣
↪ttl=64 rssi=128 time=13 ms
8 bytes from fe80::9c0b:a4e8:d3:4553 to fe80::cf99:a11c:4b:1200: icmp_seq=3␣
↪ttl=64 rssi=126 time=10 ms
8 bytes from fe80::9c0b:a4e8:d3:4553 to fe80::cf99:a11c:4b:1200: icmp_seq=4␣
↪ttl=64 rssi=126 time=7 ms
```

### 3.5.7 Assign a Static Address

So far, we have been using IPv6 Link-Local addressing. However, the Zephyr application is configured to use a statically configured IPv6 address as well which is, namely `2001:db8::1`.

If we add a similar static IPv6 address to our Linux IEEE 802.15.4 network interface, `lowpan0`, then we should expect to be able to reach that as well.

In Linux, run the following

```
sudo ip -6 addr add 2001:db8::2/64 dev lowpan0
```

We can verify that the address has been set by examining the `lowpan0` network interface again.

```
$ ip a show lowpan0
37: lowpan0@wpan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1280 qdisc noqueue␣
↪state UNKNOWN group default qlen 1000
    link/6lowpan 9e:0b:a4:e8:00:d3:45:53 brd ff:ff:ff:ff:ff:ff:ff:ff
    inet6 2001:db8::2/64 scope global
    valid_lft forever preferred_lft forever
    inet6 fe80::9c0b:a4e8:d3:4553/64 scope link
    valid_lft forever preferred_lft forever
```

Lastly, ping the statically configured IPv6 address of the Zephyr device.

```
$ ping6 2001:db8::1
PING 2001:db8::1(2001:db8::1) 56 data bytes
64 bytes from 2001:db8::1: icmp_seq=2 ttl=64 time=53.7 ms
64 bytes from 2001:db8::1: icmp_seq=3 ttl=64 time=13.1 ms
64 bytes from 2001:db8::1: icmp_seq=4 ttl=64 time=22.0 ms
64 bytes from 2001:db8::1: icmp_seq=5 ttl=64 time=22.7 ms
64 bytes from 2001:db8::1: icmp_seq=6 ttl=64 time=18.4 ms
```

Now that we have set up a reliable transport, let's move on to the application layer.

### 3.5.8 Greybus

Hopefully the videos listed earlier provide a sufficient foundation to understand what will happen shortly. However, there is still a bit more preparation required.

#### Build and probe Greybus Kernel Modules

Greybus was originally intended to work exclusively on the UniPro physical layer. However, we're using RF as our physical layer and TCP/IP as our transport. As such, there was need to be able to communicate with the Linux Greybus facilities through userspace, and out of that need arose gb-netlink. The Netlink Greybus module actually does not care about the physical layer, but is happy to usher Greybus messages back and forth between the kernel and userspace.

Build and probe the gb-netlink modules (as well as the other Greybus modules) with the following:

```
cd ${WORKSPACE}/sw/greybus
make -j`nproc --all`
sudo make install
../load_gb_modules.sh
```

#### Build and Run Gbridge

The gbridge utility was created as a proof of concept to abstract the Greybus Netlink datapath among several reliable transports. For the purposes of this tutorial, we'll be using it as a TCP/IP bridge.

To run `gbridge`, perform the following:

```
sudo apt install -y libnl-3-dev libnl-genl-3-dev libbluetooth-dev libavahi-
↪client-dev
cd gbridge
autoreconf -vfi
GBNETLINKDIR=${PWD}/../greybus \
./configure --enable-uart --enable-tcpip --disable-gbsim --enable-netlink --
↪disable-bluetooth
make -j`nproc --all`
sudo make install
gbridge
```

### 3.5.9 Blinky!

Now that we have set up a reliable TCP transport, and set up the Greybus modules in the Linux kernel, and used Gbridge to connect a Greybus node to the Linux kernel via TCP/IP, we can now get to the heart of the demonstration!

First, save the following script as `blinky.sh`.

```
#!/bin/bash

# Blinky Demo for CC1352R SensorTag

# /dev/gpiochipN that Greybus created
CHIP="$(gpiodetect | grep greybus_gpio | head -n 1 | awk '{print $1}')"

# red, green, blue LED pins
RED=6
GREEN=7
BLUE=21

# Bash array for pins and values
PINS=($RED $GREEN $BLUE)
NPINS=${#PINS[@]}

for ((;;)); do
    for i in ${!PINS[@]}; do
        # turn off previous pin
        if [ $i -eq 0 ]; then
            PREV=2
        else
            PREV=$((i-1))
        fi
        gpioset $CHIP ${PINS[$PREV]}=0

        # turn on current pin
        gpioset $CHIP ${PINS[$i]}=1

        # wait a sec
        sleep 1
    done
done
```

Second, run the script with root privileges: `sudo bash blinky.sh`

The output of your minicom session should resemble the following.

```
$ *** Booting Zephyr OS build zephyr-v2.3.0-1435-g40c0ed940d71  ***
[00:00:00.011,932] <inf> net_config: Initializing network
[00:00:00.111,938] <inf> net_config: IPv6 address: fe80::6c42:bc1c:4b:1200
```

(continues on next page)

(continued from previous page)

```
[00:00:00.112,121] <dbg> greybus_service.greybus_service_init: Greybus␣
↪initializing..
[00:00:00.112,426] <dbg> greybus_transport_tcpip.gb_transport_backend_init:␣
↪Greybus TCP/IP Transport initializing..
[00:00:00.112,579] <dbg> greybus_transport_tcpip.netsetup: created server␣
↪socket 0 for cport 0
[00:00:00.112,579] <dbg> greybus_transport_tcpip.netsetup: setting socket␣
↪options for socket 0
[00:00:00.112,609] <dbg> greybus_transport_tcpip.netsetup: binding socket 0␣
↪(cport 0) to port 4242
[00:00:00.112,640] <dbg> greybus_transport_tcpip.netsetup: listening on␣
↪socket 0 (cport 0)
[00:00:00.112,823] <dbg> greybus_transport_tcpip.netsetup: created server␣
↪socket 1 for cport 1
[00:00:00.112,823] <dbg> greybus_transport_tcpip.netsetup: setting socket␣
↪options for socket 1
[00:00:00.112,854] <dbg> greybus_transport_tcpip.netsetup: binding socket 1␣
↪(cport 1) to port 4243
[00:00:00.112,854] <dbg> greybus_transport_tcpip.netsetup: listening on␣
↪socket 1 (cport 1)
[00:00:00.113,037] <inf> net_config: IPv6 address: fe80::6c42:bc1c:4b:1200
[00:00:00.113,250] <dbg> greybus_transport_tcpip.netsetup: created server␣
↪socket 2 for cport 2
[00:00:00.113,250] <dbg> greybus_transport_tcpip.netsetup: setting socket␣
↪options for socket 2
[00:00:00.113,281] <dbg> greybus_transport_tcpip.netsetup: binding socket 2␣
↪(cport 2) to port 4244
[00:00:00.113,311] <dbg> greybus_transport_tcpip.netsetup: listening on␣
↪socket 2 (cport 2)
[00:00:00.113,494] <dbg> greybus_transport_tcpip.netsetup: created server␣
↪socket 3 for cport 3
[00:00:00.113,494] <dbg> greybus_transport_tcpip.netsetup: setting socket␣
↪options for socket 3
[00:00:00.113,525] <dbg> greybus_transport_tcpip.netsetup: binding socket 3␣
↪(cport 3) to port 4245
[00:00:00.113,555] <dbg> greybus_transport_tcpip.netsetup: listening on␣
↪socket 3 (cport 3)
[00:00:00.113,861] <inf> greybus_transport_tcpip: Greybus TCP/IP Transport␣
↪initialized
[00:00:00.116,149] <inf> greybus_service: Greybus is active
[00:00:00.116,546] <dbg> greybus_transport_tcpip.accept_loop: calling poll
[00:45:08.397,399] <dbg> greybus_transport_tcpip.accept_loop: poll returned 1
[00:45:08.397,399] <dbg> greybus_transport_tcpip.accept_loop: socket 0␣
↪(cport 0) has traffic
[00:45:08.397,491] <dbg> greybus_transport_tcpip.accept_loop: accepted␣
↪connection from [2001:db8::2]:39638 as fd 4
[00:45:08.397,491] <dbg> greybus_transport_tcpip.accept_loop: spawning␣
↪client thread..
[00:45:08.397,735] <dbg> greybus_transport_tcpip.accept_loop: calling poll
[00:45:08.491,363] <dbg> greybus_transport_tcpip.accept_loop: poll returned 1
[00:45:08.491,363] <dbg> greybus_transport_tcpip.accept_loop: socket 3␣
↪(cport 3) has traffic
[00:45:08.491,455] <dbg> greybus_transport_tcpip.accept_loop: accepted␣
↪connection from [2001:db8::2]:39890 as fd 5
[00:45:08.491,455] <dbg> greybus_transport_tcpip.accept_loop: spawning␣
↪client thread..
[00:45:08.491,699] <dbg> greybus_transport_tcpip.accept_loop: calling poll
[00:45:08.620,056] <dbg> greybus_transport_tcpip.accept_loop: poll returned 1
[00:45:08.620,086] <dbg> greybus_transport_tcpip.accept_loop: socket 2␣
↪(cport 2) has traffic
[00:45:08.620,147] <dbg> greybus_transport_tcpip.accept_loop: accepted␣
```

(continues on next page)

(continued from previous page)

```
↪connection from [2001:db8::2]:42422 as fd 6
[00:45:08.620,147] <dbg> greybus_transport_tcpip.accept_loop: spawning␣
↪client thread..
[00:45:08.620,422] <dbg> greybus_transport_tcpip.accept_loop: calling poll
[00:45:08.679,504] <dbg> greybus_transport_tcpip.accept_loop: poll returned 1
[00:45:08.679,534] <dbg> greybus_transport_tcpip.accept_loop: socket 1␣
↪(cport 1) has traffic
[00:45:08.679,595] <dbg> greybus_transport_tcpip.accept_loop: accepted␣
↪connection from [2001:db8::2]:48286 as fd 7
[00:45:08.679,595] <dbg> greybus_transport_tcpip.accept_loop: spawning␣
↪client thread..
[00:45:08.679,870] <dbg> greybus_transport_tcpip.accept_loop: calling poll
...
```

### 3.5.10 Read I2C Registers

The SensorTag comes with an opt3001 ambient light sensor as well as an hdc2080 temperature & humidity sensor.

First, find which i2c device corresponds to the SensorTag:

```
ls -la /sys/bus/i2c/devices/* | grep "greybus"
lrwxrwxrwx 1 root root 0 Aug 15 11:24 /sys/bus/i2c/devices/i2c-8 -> ../../../
↪devices/virtual/gb_nl/gn_nl/greybus1/1-2/1-2.2/1-2.2.2/gbphy2/i2c-8
```

On my machine, the i2c device node that Greybus creates is `/dev/i2c-8`.

Read the ID registers (at the i2c register address 0x7e) of the opt3001 sensor (at i2c bus address 0x44) as shown below:

```
i2cget -y 8 0x44 0x7e w
0x4954
```

Read the ID registers (at the i2c register address 0xfc) of the hdc2080 sensor (at i2c bus address 0x41) as shown below:

```
i2cget -y 8 0x41 0xfc w
0x5449
```

## 3.6 Conclusion

The blinking LED can and poking i2c registers can be a somewhat anticlimactic, but hopefully it illustrates the potential for Greybus as an IoT application layer protocol.

What is nice about this demo, is that we're using Device Tree to describe our Greybus Peripheral declaratively, they Greybus Manifest is automatically generated, and the Greybus Service is automatically started in Zephyr.

In other words, all that is required to replicate this for other IoT devices is simply an appropriate Device Tree overlay file.

The proof-of-concept involving Linux, Zephyr, and IEEE 802.15.4 was actually fairly straight forward and was accomplished with mostly already-upstream source.

For Greybus in Zephyr, there is still a considerable amount of integration work to be done, including * converting the fork to a proper Zephyr module * adding security and authentication * automatic detection, joining, and rejoining of devices.

Thanks for reading, and we hope you've enjoyed this tutorial.