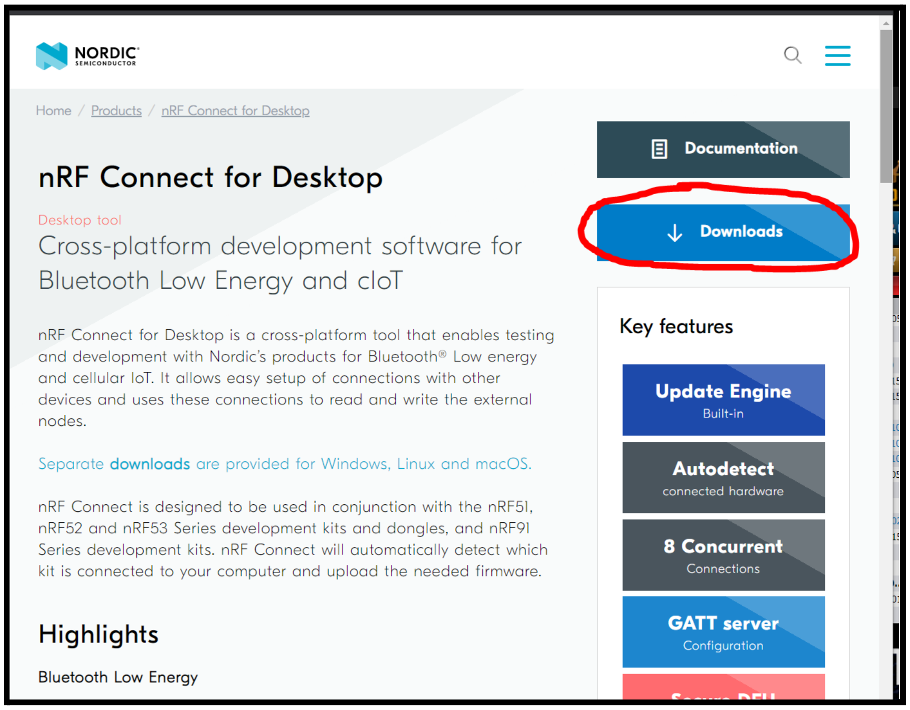
Table of Contents

Setting Up The Environment

The easiest way to set up the environment on windows is to install the **nRF Connect Software Development Kit**. This will install all the necessary components to run and work the Zephyr based projects. The nRF Connect SDK will also download the samples which makes them easy to try. The Thingy 52 code discussed down below was written with **nRF Connect SDK v1.5.1**.

To install the nRF Connect SDK v1.5.1 or any of the versions use the following steps:

1. Install nRF Connect for Desktop for Nordic Semiconductor
    a. https://www.nordicsemi.com/Products/Development-tools/nrf-connect-for-desktop
    b. Click downloads on the page



Home / Products / nRF Connect for Desktop

nRF Connect for Desktop

Desktop tool

Cross-platform development software for Bluetooth Low Energy and cIoT

nRF Connect for Desktop is a cross-platform tool that enables testing and development with Nordic's products for Bluetooth® Low energy and cellular IoT. It allows easy setup of connections with other devices and uses these connections to read and write the external nodes.

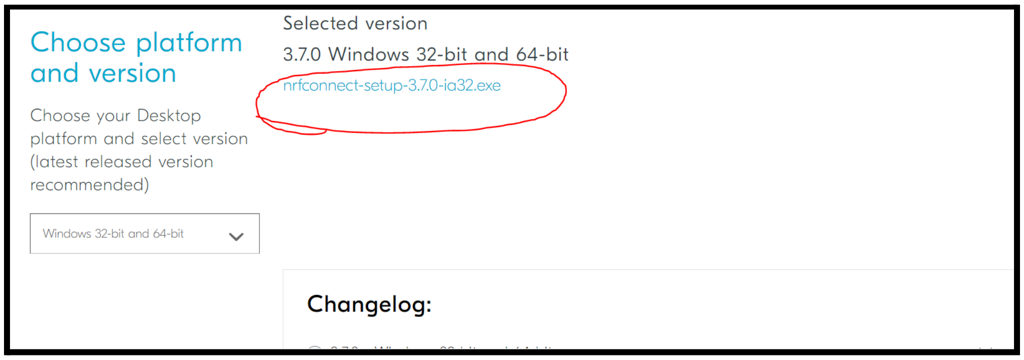
Separate downloads are provided for Windows, Linux and macOS.

nRF Connect is designed to be used in conjunction with the nRF51, nRF52 and nRF53 Series development kits and dongles, and nRF91 Series development kits. nRF Connect will automatically detect which kit is connected to your computer and upload the needed firmware.

Highlights

Bluetooth Low Energy

    c. Click this link to download the executable



Choose platform and version

Choose your Desktop platform and select version (latest released version recommended)

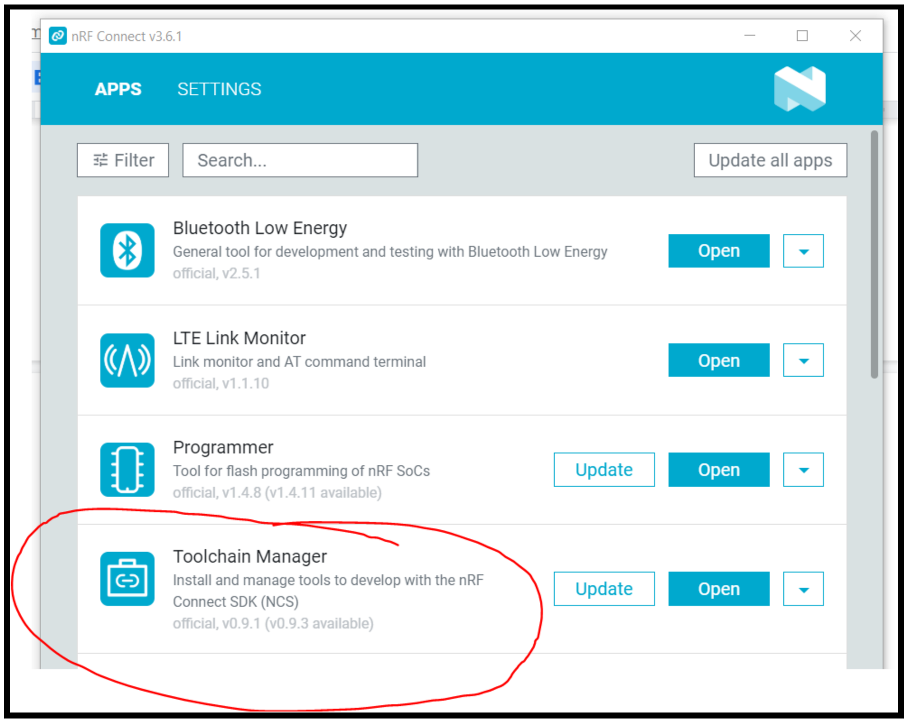
Selected version

3.7.0 Windows 32-bit and 64-bit

nrfconnect-setup-3.7.0-ia32.exe

Changelog:

    d. Once you have completed the installation for nRF Connect. Install the **Toolchain Manager**


nRF Connect v3.6.1

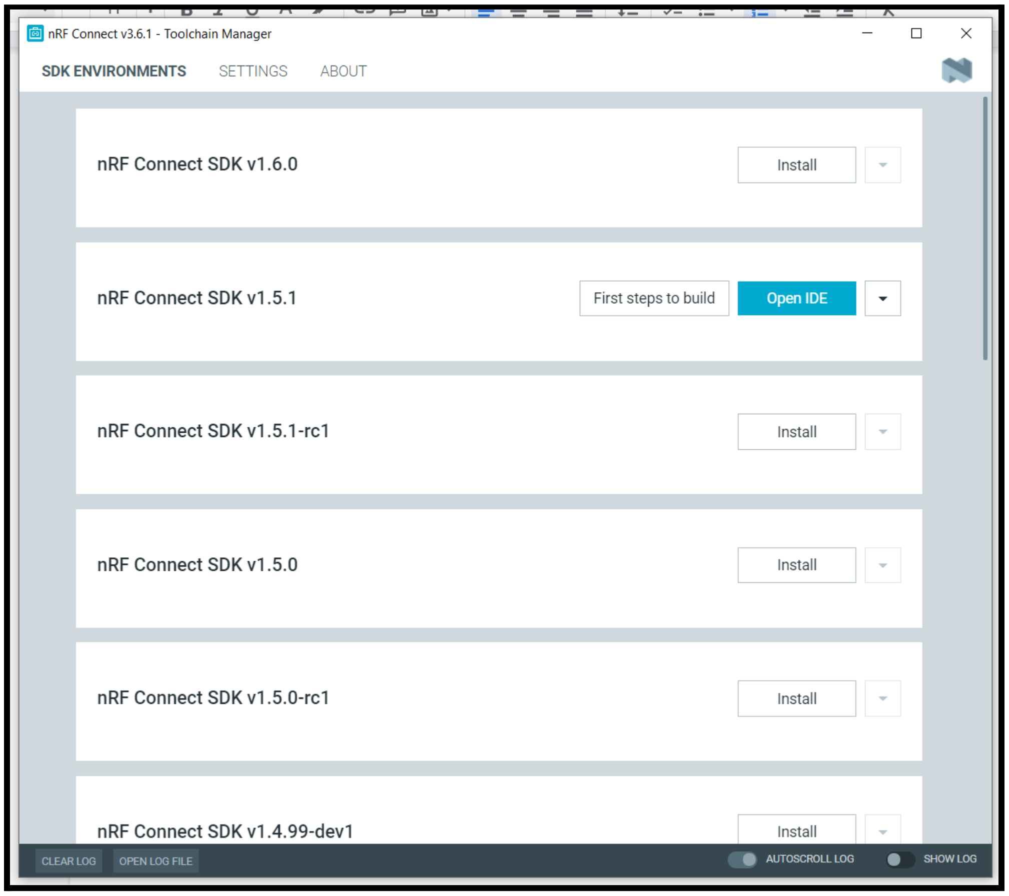
Bluetooth Low Energy
General tool for development and testing with Bluetooth Low Energy
official, v2.5.1

LTE Link Monitor
Link monitor and AT command terminal
official, v1.1.10

Programmer
Tool for flash programming of nRF SoCs
official, v1.4.8 (v1.4.11 available)

Toolchain Manager
Install and manage tools to develop with the nRF Connect SDK (NCS)
official, v0.9.1 (v0.9.3 available)

e. Then you will be able to install the different nRF Connect SDK versions


nRF Connect v3.6.1 - Toolchain Manager

nRF Connect SDK v1.6.0 — Install

nRF Connect SDK v1.5.1 — First steps to build — Open IDE

nRF Connect SDK v1.5.1-rc1 — Install

nRF Connect SDK v1.5.0 — Install

nRF Connect SDK v1.5.0-rc1 — Install

nRF Connect SDK v1.4.99-dev1 — Install

f. Follow the installation instructions to install the SDK

Connecting the Thingy: 52 and the Segger J-Link

1. To connect the Thingy 52 to the Segger J-Link, simply use the 10 pin connector provided to connect the two devices. Below is a diagram of the Thingy 52 and where to connect the 10 pin connector on the device.

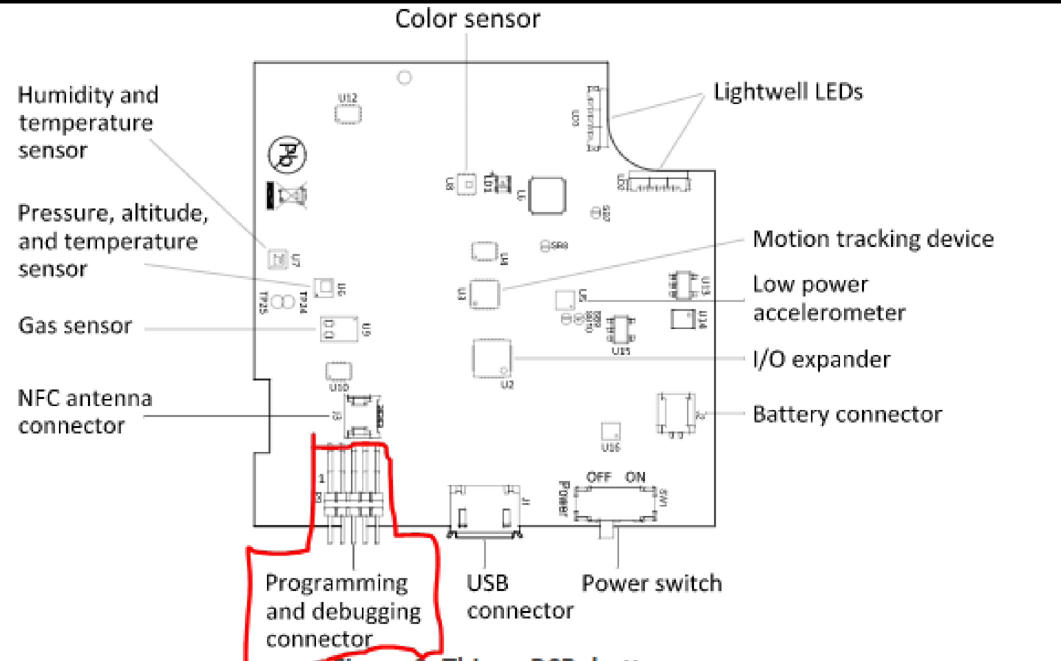
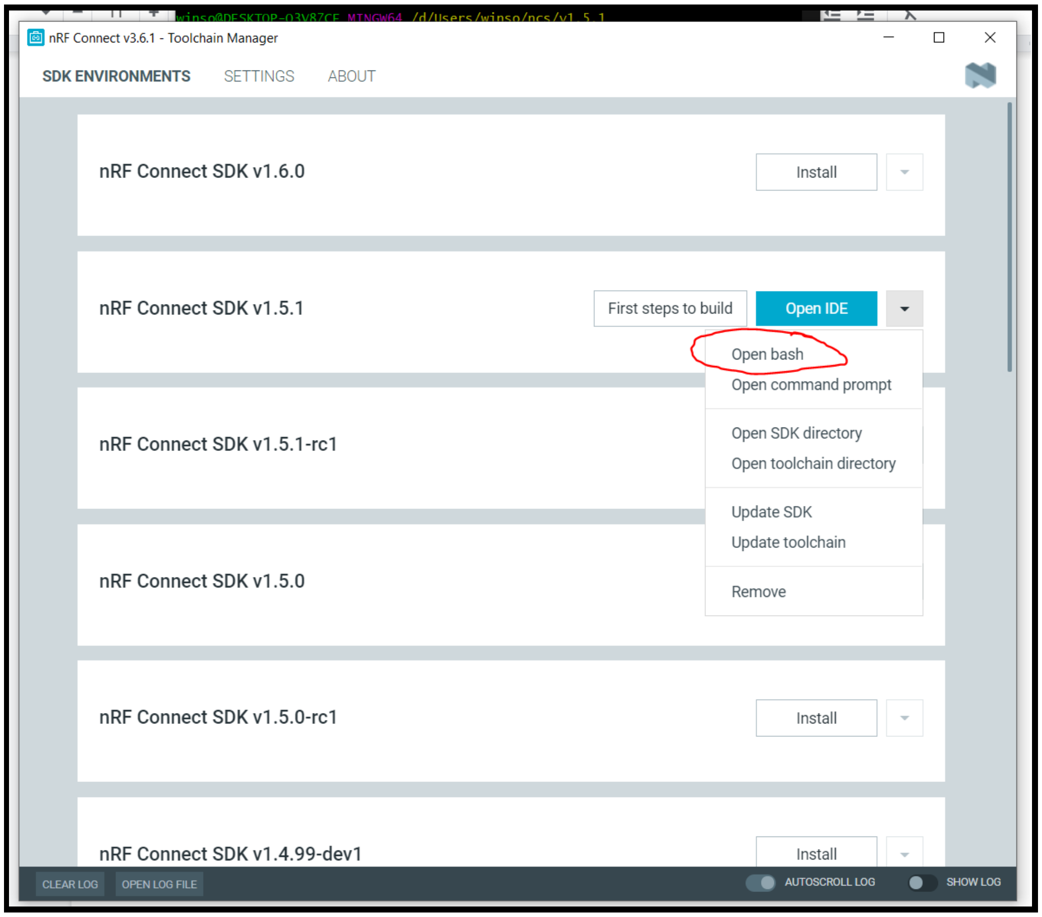


Figure 2. Thingy PCB, bottom

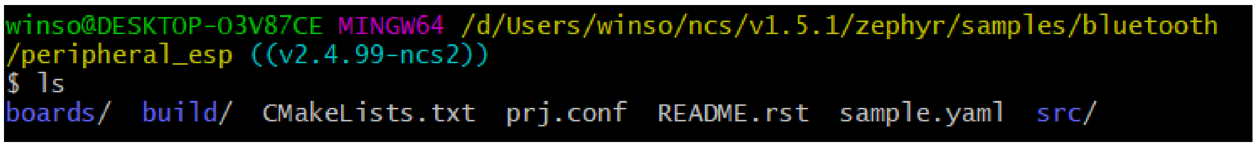
2. Connect the Segger J-Link to the computer via micro-USB.

Compiling and Flashing the Code

1. Open the terminal by clicking the drop down for an SDK in the toolchain manager and clicking **open bash.**



2. To compile the code simply run **west build -p -b [platform] [folder]** in the terminal. For us our platform is `thingy52_nrf5283` and the folder must contain the necessary files such as the CMakeList.txt, the prj.conf, and the source files.The bluetooth peripheral environment sensor profile sample folder's contents are displayed below.



```
winso@DESKTOP-03V87CE MINGW64 /d/Users/winso/ncs/v1.5.1/zephyr/samples/bluetooth
/peripheral_esp ((v2.4.99-ncs2))
$ ls
boards/  build/  CMakeLists.txt  prj.conf  README.rst  sample.yaml  src/
```

   To run this folder from the zephyr directory, we would run `west build -b thingy52_nrf52832 samples/sensor/peripheral_esp`

3. If it has built successfully and the Thingy 52 is connected via the J-Link and is on, we should be able to use the `west flash` command in the terminal to flash what we just compiled onto the Thingy 52.

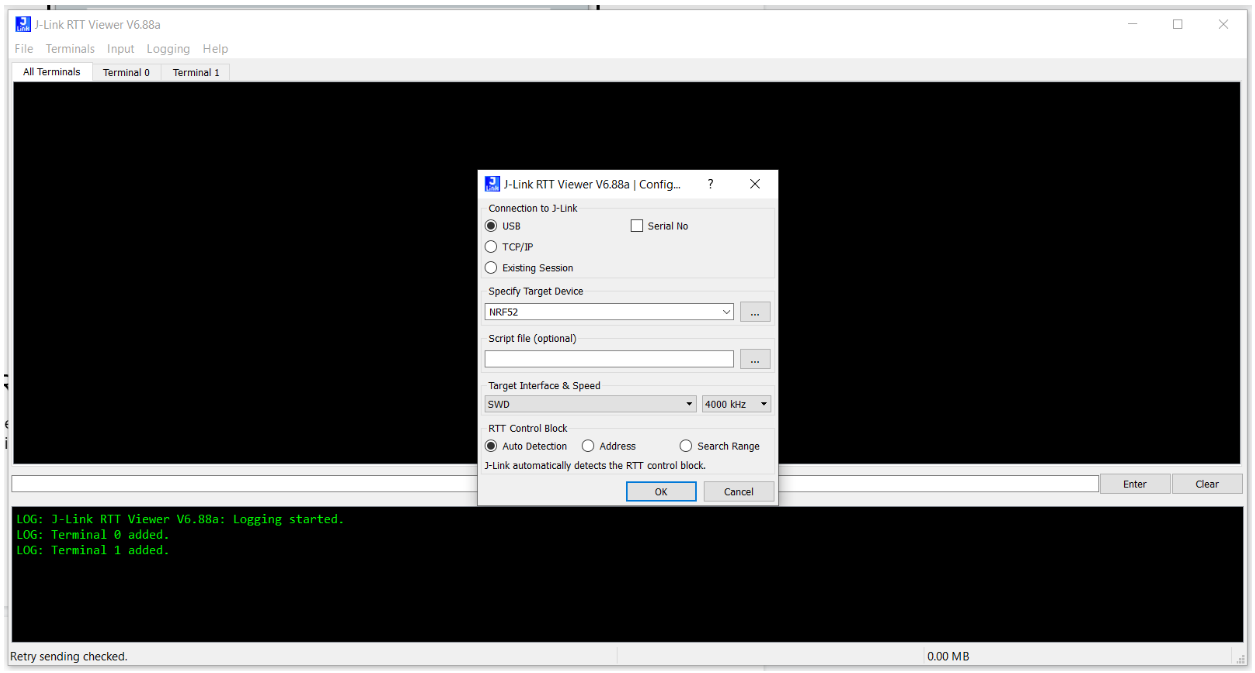
RTT (For Debugging and Tracing)

Segger J-Link supports Real Time Tracing which can be enabled by using the prj.conf file. Simply include these 3 lines into the prj.conf file.

1. `CONFIG_USE_SEGGER_RTT=y`
2. `CONFIG_RTT_CONSOLE=y`

3. `CONFIG_UART_CONSOLE=n`

You should be able to use RTT Viewer to now view the console using these settings


J-Link RTT Viewer V6.88a
File Terminals Input Logging Help
All Terminals Terminal 0 Terminal 1
Enter Clear
OK Cancel

J-Link RTT Viewer V6.88a | Config...

Connection to J-Link
- [✓ USB]
- Serial No
- TCP/IP
- Existing Session

Specify Target Device: NRF52

Script file (optional):

Target Interface & Speed: SWD, 4000 kHz

RTT Control Block
- [✓ Auto Detection]
- Address
- Search Range

J-Link automatically detects the RTT control block.

```
LOG: J-Link RTT Viewer V6.88a: Logging started.
LOG: Terminal 0 added.
LOG: Terminal 1 added.
```

Retry sending checked. 0.00 MB

If you have enabled RTT correctly, you should be able to view printf and printk statements from your code.

For more information use this link:

https://docs.zephyrproject.org/latest/guides/tools/nordic_segger.html

Thingy 52 Code

Notes: The code was written with the nRF Connect SDK v.1.5.1

Github: https://github.com/winsonrbi/thingy_code

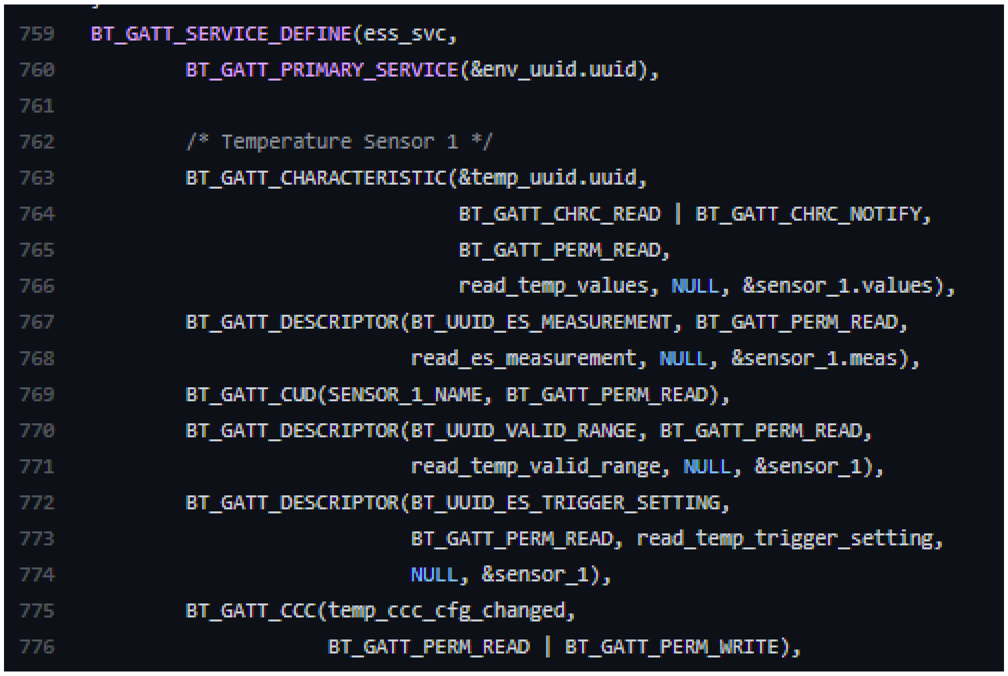
Coding the Bluetooth

For general information about bluetooth services, characteristics, and attributes

- http://www.blesstags.eu/2018/08/services-characteristics-descriptors.html
- If you are not familiar with bluetooth services I would recommend reading this and familiarizing yourself with this topic as they are used extensively in the code.

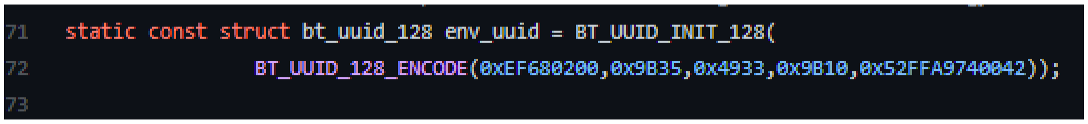
You can declare a bluetooth service struct using the **`BT_GATT_SERVICE_DEFINE`** macro, then declare the primary service using **`BT_GATT_PRIMARY_SERVICE`** macro. Then to define the characteristics for the service you can use **`BT_GATT_CHARACTERISTIC`** macro.

To read more about these macros you can use this link (https://docs.zephyrproject.org/latest/reference/bluetooth/gatt.html) and if you would like to see how to use these macros, you can look at the code or study some of the Zephyr bluetooth samples.



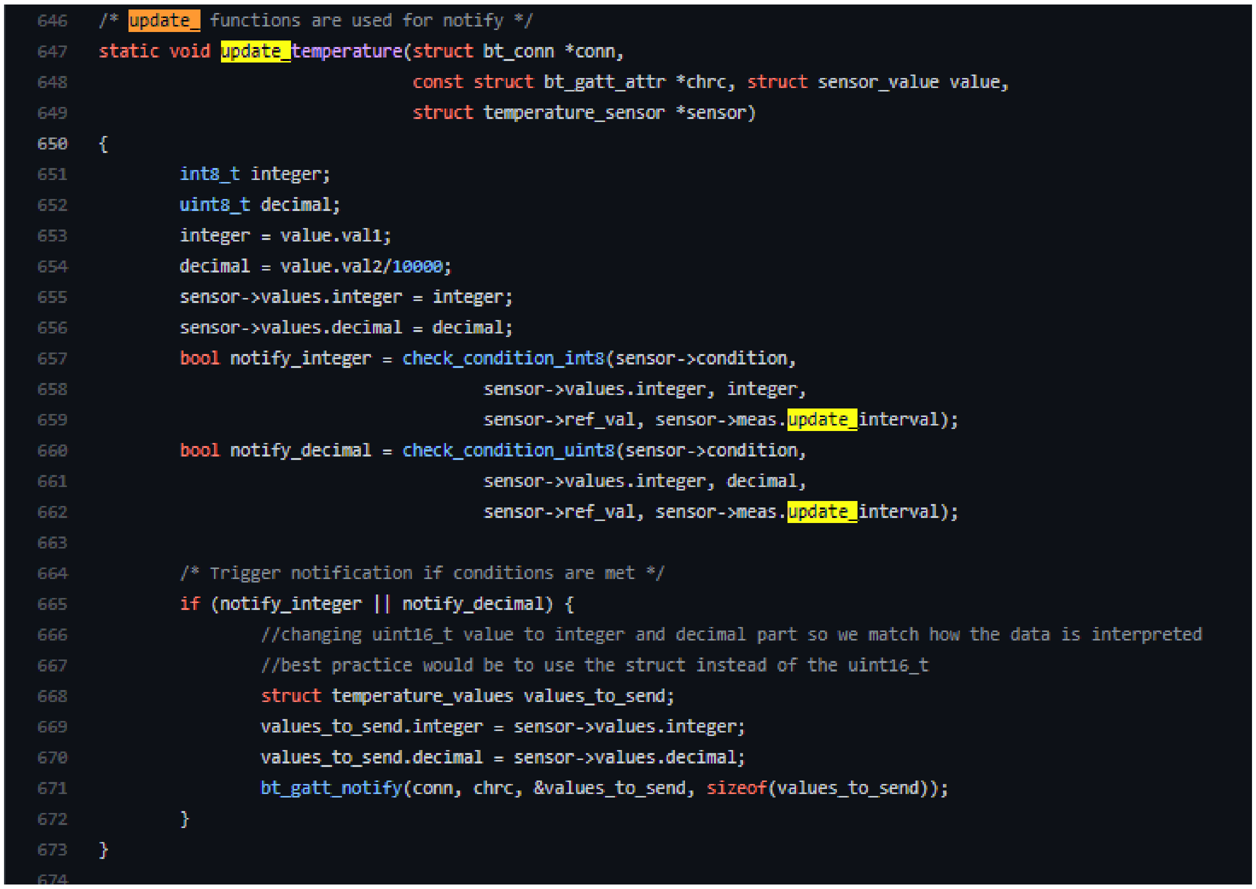
```
BT_GATT_SERVICE_DEFINE(ess_svc,
	BT_GATT_PRIMARY_SERVICE(&env_uuid.uuid),

	/* Temperature Sensor 1 */
	BT_GATT_CHARACTERISTIC(&temp_uuid.uuid,
			       BT_GATT_CHRC_READ | BT_GATT_CHRC_NOTIFY,
			       BT_GATT_PERM_READ,
			       read_temp_values, NULL, &sensor_1.values),
	BT_GATT_DESCRIPTOR(BT_UUID_ES_MEASUREMENT, BT_GATT_PERM_READ,
			   read_es_measurement, NULL, &sensor_1.meas),
	BT_GATT_CUD(SENSOR_1_NAME, BT_GATT_PERM_READ),
	BT_GATT_DESCRIPTOR(BT_UUID_VALID_RANGE, BT_GATT_PERM_READ,
			   read_temp_valid_range, NULL, &sensor_1),
	BT_GATT_DESCRIPTOR(BT_UUID_ES_TRIGGER_SETTING,
			   BT_GATT_PERM_READ, read_temp_trigger_setting,
			   NULL, &sensor_1),
	BT_GATT_CCC(temp_ccc_cfg_changed,
		    BT_GATT_PERM_READ | BT_GATT_PERM_WRITE),
```

Defining 128 bit UUID's:



```
static const struct bt_uuid_128 env_uuid = BT_UUID_INIT_128(
		BT_UUID_128_ENCODE(0xEF680200,0x9B35,0x4933,0x9B10,0x52FFA9740042));
```

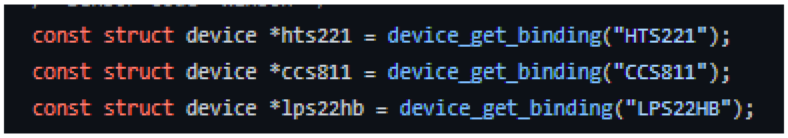
We will use **`BT_GATT_NOTIFY`** to send notifications. This macro requires an attribute from the struct created by the **`BT_GATT_SERVICE_DEFINE`** macro. The name of the struct is declared by the first parameter. The struct is created and attributes are added to this struct based on the macros from above when defining the service. For example, each call of the **`BT_GATT_CHARACTERISTIC`** macro adds two attribute entries to the struct. Depending on what attribute you are trying to update, you will have to pass in the corresponding bluetooth attribute. Below we have code that shows how to use the **`BT_GATT_NOTIFY`** macro, depending on if the condition of when we notify is true, we use the **`BT_GATT_NOTIFY`** macro to send a notification for the attribute.



```
/* update_ functions are used for notify */
static void update_temperature(struct bt_conn *conn,
                               const struct bt_gatt_attr *chrc, struct sensor_value value,
                               struct temperature_sensor *sensor)
{
        int8_t integer;
        uint8_t decimal;
        integer = value.val1;
        decimal = value.val2/10000;
        sensor->values.integer = integer;
        sensor->values.decimal = decimal;
        bool notify_integer = check_condition_int8(sensor->condition,
                                        sensor->values.integer, integer,
                                        sensor->ref_val, sensor->meas.update_interval);
        bool notify_decimal = check_condition_uint8(sensor->condition,
                                        sensor->values.integer, decimal,
                                        sensor->ref_val, sensor->meas.update_interval);

        /* Trigger notification if conditions are met */
        if (notify_integer || notify_decimal) {
                //changing uint16_t value to integer and decimal part so we match how the data is interpreted
                //best practice would be to use the struct instead of the uint16_t
                struct temperature_values values_to_send;
                values_to_send.integer = sensor->values.integer;
                values_to_send.decimal = sensor->values.decimal;
                bt_gatt_notify(conn, chrc, &values_to_send, sizeof(values_to_send));
        }
}
```

Accessing the Sensors

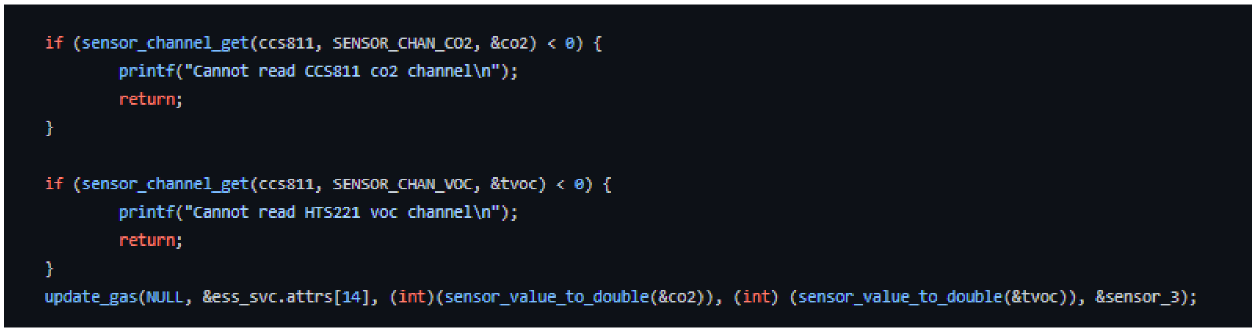
We must define the device struct for each sensor using `device_get_binding` the parameter is the label defined in the devicetree, these labels can be found here for the Thingy:52: https://github.com/zephyrproject-rtos/zephyr/blob/main/boards/arm/thingy52_nrf52832/thingy52_nrf52832.dts



```
const struct device *hts221 = device_get_binding("HTS221");
const struct device *ccs811 = device_get_binding("CCS811");
const struct device *lps22hb = device_get_binding("LPS22HB");
```

Reading the Sensor's Value

To access the device's sensor value we simply have to use the `sensor_sample_fetch` and `sensor_channel_get` functions. Below we have an example use of these functions in the ess_simulate function. The ess_simulate function fetches all sensor values and pass them to update functions for each sensor.



```
if (sensor_channel_get(ccs811, SENSOR_CHAN_CO2, &co2) < 0) {
        printf("Cannot read CCS811 co2 channel\n");
        return;
}

if (sensor_channel_get(ccs811, SENSOR_CHAN_VOC, &tvoc) < 0) {
        printf("Cannot read HTS221 voc channel\n");
        return;
}
update_gas(NULL, &ess_svc.attrs[14], (int)(sensor_value_to_double(&co2)), (int) (sensor_value_to_double(&tvoc)), &sensor_3);
```

Recommendations For The Current Code

One recommendation I might have is perhaps trying to put the application into sleep mode based on the lowest common denominator of all sensors.

The color sensor is currently not implemented and still needs to be done if it is necessary.

Change name of **`ess_simulate`** function to maybe something like **`update`**

Helpful Resources

Zephyr Thingy 52 Documentation
- https://docs.zephyrproject.org/2.4.0/boards/arm/thingy52_nrf52832/doc/index.html
- Contains basic information on how to develop for the Thingy 52 board using Zephyr

Zephyr Samples Documentation
- https://docs.zephyrproject.org/2.4.0/samples/index.html
- Contains documentation on the samples from the Zephyr project, contains samples for sensors and bluetooth as well as other things. Some samples are **not** compatible with certain boards such as the Thingy:52.

Zephyr Github Repository
- https://github.com/zephyrproject-rtos/zephyr
- Contains the code for the zephyr project, I want to note this should not need to be cloned since the samples and zephyr support should be included when we install the nRF Connect SDK

Nordic Thingy 52 Firmware Architecture
- https://nordicsemiconductor.github.io/Nordic-Thingy52-FW/documentation/firmware_architecture.html
- Contains the bluetooth UUID for all the bluetooth services and the correct data format to send for each bluetooth service