Exercise 7

This exercise demonstrates the usage of nRF52's Bluetooth radio. We take one of the provided examples and do some tweaking of our own. To make the most out of the exercise you should download the Android/iOS apps *nRF Connect* and *nRF Toolbox*. We won't go through every step of setting up the BLE, but the template project has been documented quite well and is fairly simple to study.

Softdevice

In Nordic's terms SoftDevice means the Bluetooth protocol stack implemented in the SoC. It is a pre-compiled binary image provided by Nordic Semiconductor and has an API of its own. We use SoftDevice S132 as our Bluetooth stack as it is stable and supports *Bluetooth Low Energy* Controller and Host protocols.

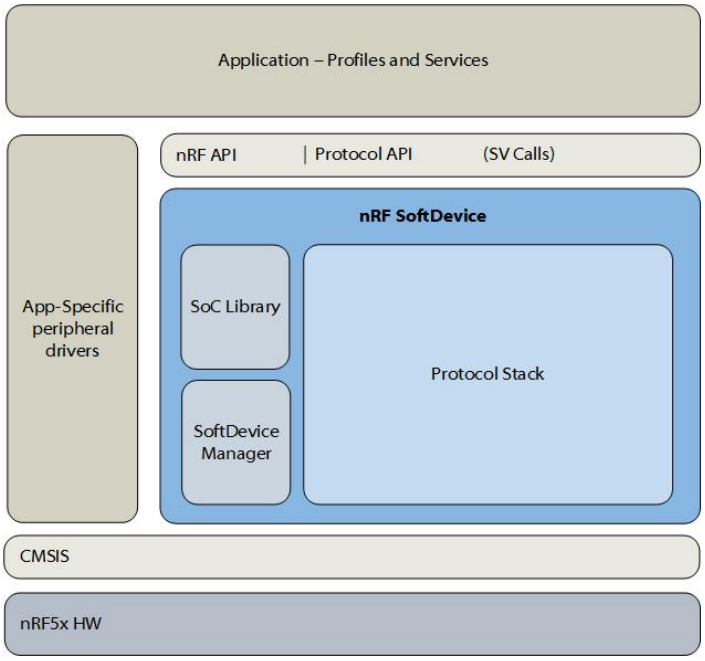

Figure 1: nRF software architecture showing the SoftDevice.

Before we can flash our own program into the board, we have to program the Softdevice using *nRFgo Studio*. Here select your device under nRF52 development boards and select the *Program SoftDevice* tab on the right. Select the correct .hex file from *<SDK>\components\softdevice\s132 \hex* and flash it to the board. First you might have to erase the board.

General Attribute Profile (GATT)

To make BLE more flexible and platform independent, the *Bluetooth Special Interest Group* has defined the GATT hierarchy. GATT is defined in the *Bluetooth Core Specification* as follows: "*The GATT Profile specifies the structure in which profile data is exchanged. This structure defines basic elements such as services and characteristics, used in a profile*." It describes how to bundle, present and transfer data using BLE. It defines both client and server roles.

GATT profile includes services, which are collections of information, e.g. values of sensors. Examples for services are: Battery Service, Heart Rate Service and Health Thermometer Service. You can also make your own custom services if needed.

At the lowest level of GATT are the characteristics. A characteristic defines the actual values and data of the service. E.g. a Battery service has only one mandatory characteristic: the Battery level characteristic, which simply is the battery charge percentage between 0 and 100. Each service has at least one mandatory characteristic it has to implement and some have optional characteristics as well.

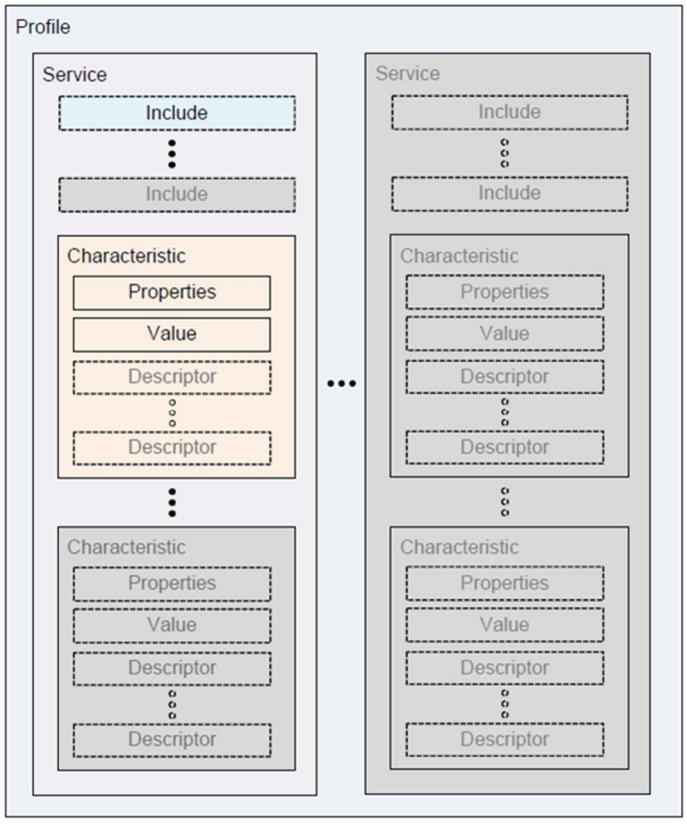

Figure 2: A diagram showing the GATT hierarchy.

General Access Profile (GAP)

GAP is the profile that controls connections and advertising in Bluetooth. The main roles for devices are defined by GAP: *Central* and *Peripheral*. Central devices are usually larger systems with a lot of memory and processing power (e.g laptops, phones) and peripherals are usually small, low energy devices (e.g. IoT tags, health monitoring wristbands). There are two ways of advertising in BLE: the *Advertising Data* payload and the *Scan Response* payload. These can contain up to 31 bytes of data and can be used for small data transfer and broadcasting. You can define the advertising interval by yourself. Short interval means high responsiveness but high power consumption.

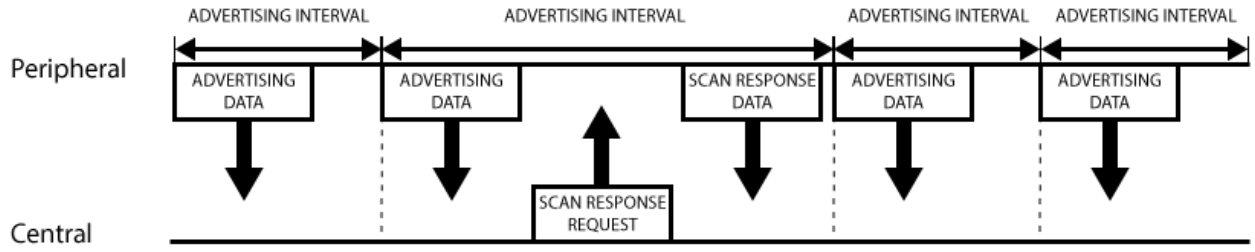

Figure 3: A diagram showing the timing of BLE advertising.

Programming

In this exercise we use the BLE template project from *<SDK>\examples\ble_peripheral \ble_app_template.* Our aim is to add a Health Thermometer Service functionality to it and be able to display simulated body (or ambient) temperature on our smartphone. The HTS also requires Battery Level Service to be implemented. You have to enable both services in *sdk_config.h*. To save program space also disable UART and logging capabilities. Change the device name to e.q. nRF_<your_name>.

```
#include "nrf_temp.h"
#include "ble_hts.h"
#include "ble_bas.h"

// Determines if temperature type is given as characteristic (1) or as a field of
measurement (0)
#define TEMP_TYPE_AS_CHARACTERISTIC     0

BLE_HTS_DEF(m_hts); // Macro for defining a ble_hts instance
BLE_BAS_DEF(m_bas); // Macro for defining a ble_bas instance

// Flag to keep track of when an indication confirmation is pending
static bool m_hts_meas_ind_conf_pending = false;

volatile uint32_t hts_counter; // hold dummy hts data

// Function declarations
static void on_hts_evt(ble_hts_t * p_hts, ble_hts_evt_t * p_evt);
static void temperature_measurement_send(void);
```

In the routine *services_init()* we need to initialize both *ble_hts* and *ble_bas* services. Remember to enable the services in *sdk_config.h*. Also disable the memory hungry logging capabilities as the µVision Evaluation license can only flash a 32 kB program.

```
ret_code_t       err_code;

ble_hts_init_t   hts_init;
ble_bas_init_t   bas_init;

// Initialize Health Thermometer Service
memset(&hts_init, 0, sizeof(hts_init));

hts_init.evt_handler                 = on_hts_evt;
hts_init.temp_type_as_characteristic = TEMP_TYPE_AS_CHARACTERISTIC;
hts_init.temp_type                   = BLE_HTS_TEMP_TYPE_BODY;

// Here the sec level for the Health Thermometer Service can be changed/increased.
hts_init.ht_meas_cccd_wr_sec = SEC_JUST_WORKS;
hts_init.ht_type_rd_sec      = SEC_OPEN;

err_code = ble_hts_init(&m_hts, &hts_init);
APP_ERROR_CHECK(err_code);

// Initialize Battery Service.
memset(&bas_init, 0, sizeof(bas_init));

// Here the sec level for the Battery Service can be changed/increased.
bas_init.bl_rd_sec        = SEC_OPEN;
bas_init.bl_cccd_wr_sec   = SEC_OPEN;
bas_init.bl_report_rd_sec = SEC_OPEN;

bas_init.evt_handler          = NULL;
bas_init.support_notification = true;
bas_init.p_report_ref         = NULL;
bas_init.initial_batt_level   = 100;

err_code = ble_bas_init(&m_bas, &bas_init);
APP_ERROR_CHECK(err_code);
```

We also need to add a button event to the event handler.

```
case BSP_EVENT_KEY_0:
    if (m_conn_handle != BLE_CONN_HANDLE_INVALID)
    {
        temperature_measurement_send();
    }
    break;
```

As well as a function for sending the temperature via BLE.

```
static void temperature_measurement_send(void)
{
    ble_hts_meas_t hts_meas; //Health Thermometer Service measurement structure
    ret_code_t     err_code;

    if (!m_hts_meas_ind_conf_pending)
    {
        generate_temperature(&hts_meas);

        err_code = ble_hts_measurement_send(&m_hts, &hts_meas);

        switch (err_code)
        {
            case NRF_SUCCESS:
                // Measurement was successfully sent, wait for confirmation.
                m_hts_meas_ind_conf_pending = true;
                break;

            case NRF_ERROR_INVALID_STATE:
                // Ignore error.
                break;

            default:
                APP_ERROR_HANDLER(err_code);
                break;
        }
    }
}
```

And a function for handling HTS events.

```
/*
 * Function for handling the Health Thermometer Service events.
 */
static void on_hts_evt(ble_hts_t * p_hts, ble_hts_evt_t * p_evt)
{
    switch (p_evt->evt_type)
    {
        case BLE_HTS_EVT_INDICATION_ENABLED:
            // Indication has been enabled, send a single temperature measurement
            temperature_measurement_send();
            break;

        case BLE_HTS_EVT_INDICATION_CONFIRMED:
            m_hts_meas_ind_conf_pending = false;
            break;

        default:
            // No implementation needed.
            break;
    }
}
```

HTS requires the temperature to be sent in a specific format with time stamp information, so here we simulate the actual body temperature sensor with a custom function.

```
/*
 * Function for generating a dummy temperature information packet.
 */
static void generate_temperature(ble_hts_meas_t * p_meas)
{
    static ble_date_time_t time_stamp = { 2018, 16, 10, 16, 15, 0 };

    uint32_t celciusX100;

    p_meas->temp_in_fahr_units = false;
    p_meas->time_stamp_present = true;
    p_meas->temp_type_present  = (TEMP_TYPE_AS_CHARACTERISTIC ? false : true);

    celciusX100 = 2000+hts_counter++; // one unit is 0.01 Celcius

    p_meas->temp_in_celcius.exponent = -2;
    p_meas->temp_in_celcius.mantissa = celciusX100;
    p_meas->temp_in_fahr.exponent    = -2;
    p_meas->temp_in_fahr.mantissa    = (32 * 100) + ((celciusX100 * 9) / 5);
    p_meas->time_stamp               = time_stamp;
    p_meas->temp_type                = BLE_HTS_TEMP_TYPE_FINGER;

    // update simulated time stamp
    time_stamp.seconds += 27;
    if (time_stamp.seconds > 59)
    {
        time_stamp.seconds -= 60;
        time_stamp.minutes++;
        if (time_stamp.minutes > 59)
        {
            time_stamp.minutes = 0;
        }
    }
}
```

Verifying functionality

After compiling and flashing the code, you can open the nRF Connect app and connect to your device. The app shows you the services and characteristics we defined earlier. You can tap on *Show log* to observe the BLE transactions. Next open nRF Toolbox and select HTM. The displayed temperature should update on button press.

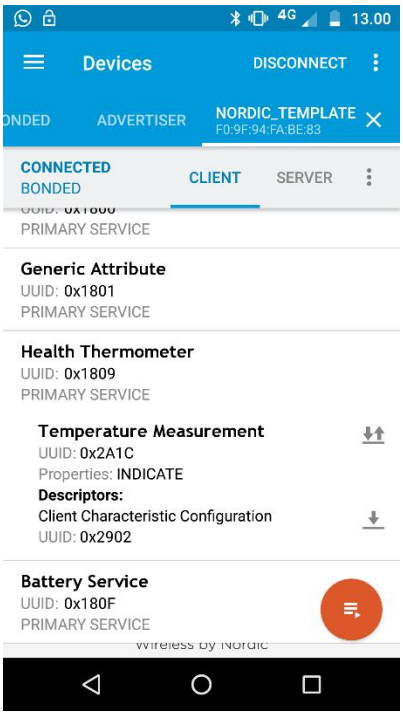

Figure 4: nRF Connect showing the advertised services.