Table of Contents

1. Zephyr BLE peripheral 2
1.1. Battery service 3
1.2. Device Information Service 3
1.3. Health thermometer service 3
1.4. Adding the possibility of reading the characteristic 4
1.5. Tasks to do during the laboratory 6

1. Zephyr BLE peripheral

In this example, we will observe BLE behaviour in peripheral (server) mode. In Zephyr OS there are some software examples of different BLE devices. One of them is “peripheral_ht” available at the path:

```
C:\Users\...\.platformio\packages\framework-
zephyr\samples\bluetooth\peripheral_ht
```

This example uses the following services:

- Battery service (BAS)
- Device information service
- Health thermometer service

Battery service allows reading the characteristic with simulated data stored in the variable that is decremented every second. The device information service presents two characteristics which allow reading two text strings. The health thermometer service reads temperature data from the nRF52840 built-in temperature sensor and sends measurements to the client while connected. All services are presented in the advertising frame created with an “ad” structure similar to the one from the beacon example.

```
static const struct bt_data ad[] = {
    BT_DATA_BYTES(BT_DATA_FLAGS, (BT_LE_AD_GENERAL | BT_LE_AD_NO_BREDR)),
    BT_DATA_BYTES(BT_DATA_UUID16_ALL,
              BT_UUID_16_ENCODE(BT_UUID_HTS_VAL),
              BT_UUID_16_ENCODE(BT_UUID_DIS_VAL),
              BT_UUID_16_ENCODE(BT_UUID_BAS_VAL)),
};
```

Services are encoded as 16-bit UUID values which can be found in Bluetooth SiG documentation [1)].

Tab. 1: UUIDs of services used in the example

UUID constant	UUID value	UUID name
BT_UUID_HTS_VAL	0x1809	Health Thermometer
BT_UUID_DIS_VAL	0x180a	Device Information
BT_UUID_BAS_VAL	0x180f	Battery

Advertising is started with the “bt_le_adv_start()” function call:

```
err = bt_le_adv_start(BT_LE_ADV_CONN_NAME, ad, ARRAY_SIZE(ad), NULL, 0);
    if (err) {
        printk("Advertising failed to start (err %d)\n", err);
        return;
    }
```

Notice that in this example there is no additional scan response information provided while calling the advertising start function. Modification of parameters of this function results in changes in advertising

information only. If you want to add another service it must be implemented also in another part of the software.

1.1. Battery service

Battery service is the standard service intended to return the battery level expressed in percentages. The battery service is fully implemented in ZephyrOS, so we need to activate it only. It is done in “prj.conf” file with the line:

```
CONFIG_BT_BAS=y
```

Simple functions:

```
uint8_t battery_level = bt_bas_get_battery_level();
bt_bas_set_battery_level(battery_level);
```

can be used to read and set the value of the characteristic responsible for the current battery percentage.

1.2. Device Information Service

Device Information (DIS) Service is enabled with the configuration option:

```
CONFIG_BT_DIS=y
```

It is implemented in Zephyr OS and automatically handled. By default, it has two active characteristics: Model Number String, and Manufacturer Number String. Default strings can be replaced by setting the options in prj.conf:

```
CONFIG_BT_DIS_MODEL="My model"
CONFIG_BT_DIS_MANUF="Producer"
```

1.3. Health thermometer service

In the example, there is also a health thermometer service that is not fully implemented in Zephyr OS as a battery service is. The part of the software responsible for the definition, and for handling of this service is written in the “hts.c” file. The built-in thermometer is not enabled by default and is not enabled in the example files. It can be switched on in the prj.conf file by setting the option:

```
CONFIG_TEMP_NRF5=y
```

And now let us explain how the service is handled. To define the service the macro “BT_GATT_SERVICE_DEFINE” is used:

```
/* Health Thermometer Service Declaration */
```

```
BT_GATT_SERVICE_DEFINE(hts_svc,
    BT_GATT_PRIMARY_SERVICE(BT_UUID_HTS),
    BT_GATT_CHARACTERISTIC(BT_UUID_HTS_MEASUREMENT, BT_GATT_CHRC_INDICATE,
                   BT_GATT_PERM_NONE, NULL, NULL, NULL),
    BT_GATT_CCC(htmc_ccc_cfg_changed,
            BT_GATT_PERM_READ | BT_GATT_PERM_WRITE),
    /* more optional Characteristics */
);
```

BT_GATT_CHARACTERISTIC macro has the following list of parameters

```
BT_GATT_CHARACTERISTIC(_uuid,_props,_perm,_read,_write,_user_data)
```

The meaning of parameters are as follows:

- _uuid – Characteristic attribute uuid.
- _props – Characteristic attribute properties (what appears as available for the client: read, write, indicate etc.)
- _perm – Characteristic Attribute access permissions (if the client can really read or write the characteristic data.)
- _read – Characteristic Attribute read callback (address of the function called while the client wants to read the data).
- _write – Characteristic Attribute write callback (address of the write function).
- _user_data – Characteristic Attribute user data (optional pointer to the characteristic data).

The indication mechanism allows the server to send data to the client only when needed (for example periodically or on value change). The client must enable indication by setting the descriptor with UUID 0x2902 to 0x0002. This can be done in the nRF Connect application by tapping the two arrows symbol, as in figure 1. Three-arrows symbol visible in BAS service means notification, as in figure 2, which can be enabled by setting the same descriptor to 0x0001.



Fig. 1: Indication symbol in nRF Connect application



Fig. 2: Notification symbol in nRF Connect application

The notification value is updated by “bt_gatt_indicate(NULL, &ind_params)” function call, where the structure “ind_params” holds the attributes of the health thermometer service, pointers to callback functions executed at the beginning and the end of the indication, and a new value to be sent to the client. The attributes “ind_params.attr” that hold the UUID of the service connect this notification structure to the chosen service (here “hts_svc”).

1.4. Adding the possibility of reading the characteristic

In the example, the health temperature service allows an indicating mode of data reading only, enabled with the flag “BT_GATT_CHRC_INDICATE”. To enable reading the value from the characteristic there is a need to do four things:

- add property BT_GATT_CHRC_READ (if you want to allow both indicate and read you need to “or” both flags with the “pipe” operator “|”)
- modify permissions with BT_GATT_PERM_READ
- write the callback function that calls bt_gatt_attr_read() (example function in the following code)
- register callback function within the BT_GATT_CHARACTERISTIC macro using the “_read” parameter.

Notice that the temperature data which is sent in the original example by indications is encoded in the following format:

```
bool            temp_in_fahr_units
bool            time_stamp_present
bool            temp_type_present
ieee_float32_t  temp_in_celcius
ieee_float32_t  temp_in_fahr
ble_date_time_t time_stamp
uint8_t     temp_type
```

All boolean flags are set to 0 (the first byte which is sent is 0x00), so the only data is the temperature expressed as a 32-bit floating point number encoded as an 8-bit exponent and 24-bit mantissa (ieee_float32_t).



In the ieee_float32_t format, the exponent is encoded as a signed 8-bit binary 10-based number. The mantissa is encoded as a signed 24-bit binary number. This format is incompatible with single or double-precision formats used in microprocessors, so conversion is required.

<m 18> temperature = mantissa ^ (10 * exponent)</m> An example of the callback function which simulates the thermometer value can look like this:

```
static ssize_t read_ht(struct bt_conn *conn,
               const struct bt_gatt_attr *attr, void *buf,
               uint16_t len, uint16_t offset)
{
    uint8_t temp_buffer[5];
    double temp_read;
    uint32_t mantissa;
    uint8_t exponent;

    temp_read = 36.6;    //constant temperature for reading
    temp_buffer[0] = 0x00;    //flags for temerature data formatting

        //conversion from double to 4 bytes
    //mantissa is 24 bits signed binary
    //exponent is 8 bits signed 10-base binary
    mantissa = (uint32_t)(temp_read * 100);
    exponent = (uint8_t)-2;
    sys_put_le24(mantissa, (uint8_t *)&temp_buffer[1]);
    temp_buffer[4] = exponent;
```

```
    return bt_gatt_attr_read(conn, attr, buf, len, offset, &temp_buffer,
sizeof(temp_buffer));
}
```

1.5. Tasks to do during the laboratory

1. Import the peripheral_ht example.
2. Connect to the exposed server with the nRF Connect application (or similar)
3. Observe notification and indication mechanisms in BAS and health temperature services
4. Modify the program to enable the reading possibility of the health temperature (value can be simulated)

[1)]
https://bitbucket.org/bluetooth-SIG/public/src/main/assigned_numbers/uuids/service_uuids.yaml

From:
https://dokuwiki.zmitac.aei.polsl.pl/ - **RAU6 Dokuwiki**

Permanent link:
https://dokuwiki.zmitac.aei.polsl.pl/en:edu:zmitac:s2:aih:lab:nrf52zephyrbleperipheral?rev=1705155466

Last update: **2024/01/13 15:17**