WIoT - Postlab

Lab 2b: BLE Connections

What to submit?

Please use this document as a template, add your responses directly, and export it as a PDF to Gradescope. Each group should submit one postlab.

Group name: Group 14
Team member names: Hudson Burke, Yann Donastien, Zach King

A: BLE Peripheral

[1pt] What is the initial value of the characteristic in the provided code?

c5ec4501

[1pt] What did you set the service UUID to?

5253FF4B-E47C-4EC8-9792-69FDF4923B0E

[5pts] Include your updated code:

```
#include <stdbool.h>
#include <zephyr/types.h>
#include <drivers/sensor.h>
#include <stddef.h>
#include <string.h>
#include <errno.h>
#include <sys/printk.h>
#include <sys/byteorder.h>
#include <kernel.h>

#include <bluetooth/bluetooth.h>
#include <bluetooth/hci.h>
#include <bluetooth/conn.h>
#include <bluetooth/uuid.h>
#include <bluetooth/gatt.h>

#define LAB2_SERVICE_UUID BT_UUID_128_ENCODE(0x5253FF4B, 0xE47C, 0x4EC8, 0x9792,
0x69FDF4923B0E)

uint32_t count;

static ssize_t characteristic_read(struct bt_conn *conn, const struct
bt_gatt_attr *attr, void *buf, uint16_t len, uint16_t offset);

// Global value that saves state for the characteristic.
uint32_t characteristic_value = 0x7;

// Set up the advertisement data.
#define DEVICE_NAME "Group14"
#define DEVICE_NAME_LEN (sizeof(DEVICE_NAME) - 1)

static const struct bt_data ad[] = {
    BT_DATA(BT_DATA_NAME_COMPLETE, DEVICE_NAME, DEVICE_NAME_LEN),
    BT_DATA_BYTES(BT_DATA_UUID128_ALL, LAB2_SERVICE_UUID)
};

// Setup the the service and characteristics.
BT_GATT_SERVICE_DEFINE(lab2_service,
```

```c
    BT_GATT_PRIMARY_SERVICE(
        BT_UUID_DECLARE_128(LAB2_SERVICE_UUID)
    ),
    BT_GATT_CHARACTERISTIC(BT_UUID_DECLARE_16(0x000a), BT_GATT_CHRC_READ,
                          BT_GATT_PERM_READ, characteristic_read, NULL,
&characteristic_value),
);


// Callback when a client reads the characteristic.
//
// Documented under name "bt_gatt_attr_read_chrc()"
static ssize_t characteristic_read(struct bt_conn *conn,
                                   const struct bt_gatt_attr *attr,
                                   void *buf,
                                   uint16_t len,
                                   uint16_t offset)
{
    // The `user_data` corresponds to the pointer provided as the last "argument"
    // to the `BT_GATT_CHARACTERISTIC` macro.
    count++;
    uint32_t *value = attr->user_data;
    printk("%d\n", count);
    // Need to encode data into a buffer to send to client.
    uint8_t out_buffer[4] = {0};
    out_buffer[0]=(count >> 24) & 0xFF;
    out_buffer[1]=(count >> 16) & 0xFF;
    out_buffer[2]=(count >> 8)  & 0xFF;
    out_buffer[3]=(count >> 0)  & 0xFF;

    // User helper function to encode the output data to send to
    // the client.
    return bt_gatt_attr_read(conn, attr, buf, len, offset, out_buffer, 4);
}


// Setup callbacks when devices connect and disconnect.
static void connected(struct bt_conn *conn, uint8_t err)
{
    if (err) {
        printk("Connection failed (err 0x%02x)\n", err);
    } else {
        printk("Connected\n");
    }
}
```

```c
static void disconnected(struct bt_conn *conn, uint8_t reason)
{
    printk("Disconnected (reason 0x%02x)\n", reason);
}

BT_CONN_CB_DEFINE(conn_callbacks) = {
    .connected = connected,
    .disconnected = disconnected,
};


static void bt_ready(int err)
{
    if (err) {
        printk("Bluetooth init failed (err %d)\n", err);
        return;
    }

    printk("Bluetooth initialized\n");

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

    printk("Advertising successfully started\n");
}

void main(void)
{
    int err;
    count = 0;
    err = bt_enable(bt_ready);
    if (err) {
        printk("Bluetooth init failed (err %d)\n", err);
        return;
    }
}
```

B: BLE Central

[5pts] Include your updated code:

```
#include <zephyr/types.h>
#include <stddef.h>
#include <errno.h>
#include <zephyr.h>
#include <sys/printk.h>

#include <bluetooth/bluetooth.h>
#include <bluetooth/hci.h>
#include <bluetooth/conn.h>
#include <bluetooth/uuid.h>
#include <bluetooth/gatt.h>
#include <sys/byteorder.h>

#define LAB2_SERVICE_UUID BT_UUID_128_ENCODE(0x5253FF4B, 0xE47C, 0x4EC8, 0x9792,
0x69FDF4923B0E)
#define LAB2_SERVICE_CHARACTERISTIC_UUID 0x000a

static void start_scan(void);

static struct bt_conn *default_conn;

static struct bt_uuid* search_service_uuid =
BT_UUID_DECLARE_128(LAB2_SERVICE_UUID);
static struct bt_uuid* search_characteristic_uuid =
BT_UUID_DECLARE_16(LAB2_SERVICE_CHARACTERISTIC_UUID);
static struct bt_gatt_discover_params discover_params;
static struct bt_gatt_read_params read_params;

// Callback after reading characteristic value.
static uint8_t read_func(struct bt_conn *conn, uint8_t err,
                   struct bt_gatt_read_params *params,
                   const void *data, uint16_t length)
{
    if (err) {
        printk("read failed (err %d)\n", err);
    }

    uint8_t* buf = (uint8_t*) data;

    if (length == 4) {
        uint32_t val = (((uint32_t) buf[0]) << 24) |
                       (((uint32_t) buf[1]) << 16) |
                       (((uint32_t) buf[2]) <<  8) |
                       (((uint32_t) buf[3]) <<  0);

        printk("Read: 0x%x\n", val);
    }

    return BT_GATT_ITER_STOP;
}

static uint8_t discover_func(struct bt_conn *conn,
                 const struct bt_gatt_attr *attr,
                 struct bt_gatt_discover_params *params)
{
    int err;

    if (!attr) {
        printk("Discover complete\n");
        (void)memset(params, 0, sizeof(*params));
        return BT_GATT_ITER_STOP;
    }

    // printk("[ATTRIBUTE] handle %u\n", attr->handle);

    if (bt_uuid_cmp(discover_params.uuid, BT_UUID_DECLARE_128(LAB2_SERVICE_UUID))
== 0) {
        // printk("Found service\n");

        discover_params.uuid = search_characteristic_uuid;
        discover_params.start_handle = attr->handle + 1;
        discover_params.type = BT_GATT_DISCOVER_CHARACTERISTIC;

        err = bt_gatt_discover(conn, &discover_params);
        if (err) {
            printk("Discover failed (err %d)\n", err);
        }
    }
    else if (bt_uuid_cmp(discover_params.uuid,
BT_UUID_DECLARE_16(LAB2_SERVICE_CHARACTERISTIC_UUID)) == 0) {
        // printk("Found characteristic\n");

        read_params.func = read_func;
        read_params.handle_count = 1;
        read_params.single.handle = bt_gatt_attr_value_handle(attr);
        read_params.single.offset = 0U;
```

```c
        err = bt_gatt_read(conn, &read_params);
        if (err) {
            printk("Read failed (err %d)\n", err);
        }
    }

    return BT_GATT_ITER_STOP;
}

static void connected(struct bt_conn *conn, uint8_t conn_err)
{
    char addr[BT_ADDR_LE_STR_LEN];
    int err;

    bt_addr_le_to_str(bt_conn_get_dst(conn), addr, sizeof(addr));

    if (conn_err) {
        printk("Failed to connect to %s (%u)\n", addr, conn_err);

        bt_conn_unref(default_conn);
        default_conn = NULL;

        start_scan();
        return;
    }

    printk("Connected: %s\n", addr);

    if (conn == default_conn) {
        discover_params.uuid = search_service_uuid;
        discover_params.func = discover_func;
        discover_params.start_handle = BT_ATT_FIRST_ATTTRIBUTE_HANDLE;
        discover_params.end_handle = BT_ATT_LAST_ATTTRIBUTE_HANDLE;
        discover_params.type = BT_GATT_DISCOVER_PRIMARY;

        err = bt_gatt_discover(default_conn, &discover_params);
        if (err) {
            printk("Discover failed(err %d)\n", err);
            return;
        }
    }
}

static void disconnected(struct bt_conn *conn, uint8_t reason)
{
    char addr[BT_ADDR_LE_STR_LEN];

    bt_addr_le_to_str(bt_conn_get_dst(conn), addr, sizeof(addr));

    printk("Disconnected: %s (reason 0x%02x)\n", addr, reason);

    if (default_conn != conn) {
        return;
    }

    bt_conn_unref(default_conn);
    default_conn = NULL;

    start_scan();
}

BT_CONN_CB_DEFINE(conn_callbacks) = {
    .connected = connected,
    .disconnected = disconnected,
};

// Called for each advertising data element in the advertising data.
static bool ad_found(struct bt_data *data, void *user_data)
{
    bt_addr_le_t *addr = user_data;

    printk("[AD]: %u data_len %u\n", data->type, data->data_len);

    switch (data->type) {
    case BT_DATA_UUID128_ALL:
        if (data->data_len != 16) {
            printk("AD malformed\n");
            return true;
        }

        struct bt_le_conn_param *param;
        struct bt_uuid uuid;
        int err;

        bt_uuid_create(&uuid, data->data, 16);
        if (bt_uuid_cmp(&uuid, BT_UUID_DECLARE_128(LAB2_SERVICE_UUID)) == 0) {
            printk("Found matching advertisement\n");

            err = bt_le_scan_stop();
            if (err) {
                printk("Stop LE scan failed (err %d)\n", err);
                return false;
            }

            param = BT_LE_CONN_PARAM_DEFAULT;
            err = bt_conn_le_create(addr, BT_CONN_LE_CREATE_CONN, param,
&default_conn);
            if (err) {
                printk("Create conn failed (err %d)\n", err);
                start_scan();
            }
        }

        return false;

    }

    return true;
}

static void device_found(const bt_addr_le_t *addr, int8_t rssi, uint8_t type,
             struct net_buf_simple *ad)
{
    char dev[BT_ADDR_LE_STR_LEN];

    bt_addr_le_to_str(addr, dev, sizeof(dev));
    printk("[DEVICE]: %s, AD evt type %u, AD data len %u, RSSI %i\n",
           dev, type, ad->len, rssi);

    // We're only interested in connectable devices.
    if (type == BT_GAP_ADV_TYPE_ADV_IND ||
        type == BT_GAP_ADV_TYPE_ADV_DIRECT_IND) {
        // Helper function to parse the advertising data (AD) elements
        // from the advertisement. This will call `ad_found()` for
        // each element.
        bt_data_parse(ad, ad_found, (void*) addr);
    }
}

static void start_scan(void)
{
    int err;

    struct bt_le_scan_param scan_param = {
        .type       = BT_LE_SCAN_TYPE_PASSIVE,
```

```c
        .options     = BT_LE_SCAN_OPT_NONE,
        .interval    = BT_GAP_SCAN_FAST_INTERVAL,
        .window      = BT_GAP_SCAN_FAST_WINDOW,
    };

    err = bt_le_scan_start(&scan_param, device_found);
    if (err) {
        printk("Scanning failed to start (err %d)\n", err);
        return;
    }

    printk("Scanning successfully started\n");
}

static void bt_ready(int err)
{
    if (err) {
        printk("Bluetooth init failed (err %d)\n", err);
        return;
    }

    printk("Bluetooth initialized\n");

    start_scan();
}

void main(void)
{
    int err;

    err = bt_enable(bt_ready);

    if (err) {
        printk("Bluetooth init failed (err %d)\n", err);
        return;
    }
}
```