Zephyr: Bluetooth Low Energy (BLE)

Medical Electrical Equipment (BME590L)

Mark L. Palmeri, M.D., Ph.D.

April 10, 2023

Duke | BME

What is Bluetooth?

- Bluetooth Low Energy (BLE) is a wireless serial communication protocol.
- Part of BT 4.0 core; very distinct from previous version of BT.
- Standardized via the BT Special Interest Group (SIG)
- "Only" ubiquitous communication protocol between device:OS[1].
- Designed for technology in smart homes, health, sport and fitness.
- "Low energy" - months/years of power on single battery (e.g., CR2032)
- Small size & low cost

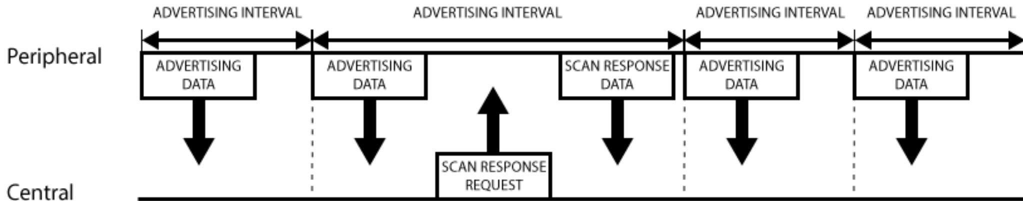
[1] iOS, Android, OS X, Windows, Linux

Generic Access Profile (GAP)

- **Peripheral Device:** more resource-constrained (power, size, horsepower) device
- **Central Device:** more resource-equipped device that will do most processing and data storage

Device Advertising

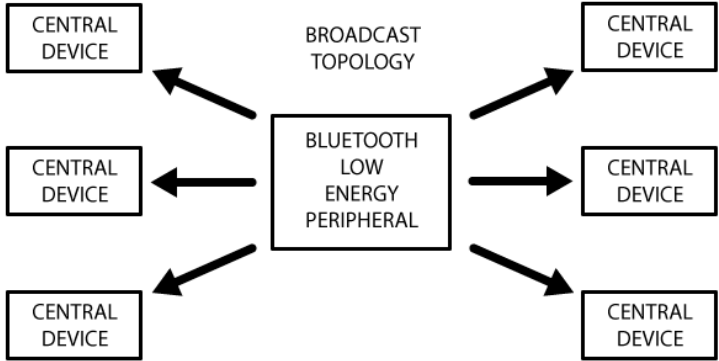
- 31 bytes of data
- Can configure advertising interval (frequency $\propto$ energy consumption)
- Optional: Scan Response Request



- Once a connection is established, advertising ceases and GATT operations take over.

Duke | BME

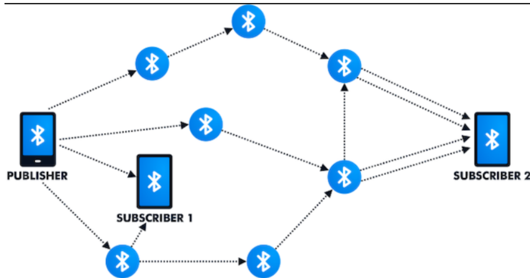
Broadcast Topology



Mesh Profiles

- BLE peripherals can communicate with one another to pass information to other devices (that might otherwise be out of range).
- Common example: lights in a building
- Zephyr has its own standard protocol:

`https://launchstudio.bluetooth.com/ListingDetails/95153`

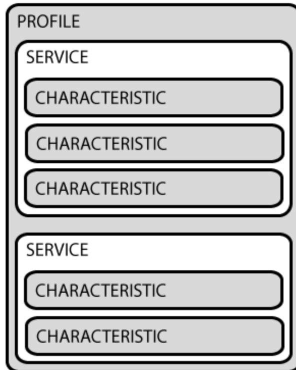


Duke | BME

Generic Attribute (GATT) Profiles

- API used by all BLE devices.
- **Client:** initiates GATT commands/requests & accepts responses (e.g., smartphone)
- **Server:** receives GATT commands/requests & returns responses (e.g., wearable sensor)
- **Characteristic:** Data value communicated between client/server
- **Service:** Collection of related characteristics (similar to a structure or class/object)
- **Descriptor:** optional characteristic metadata (e.g., value units)

GATT Profiles



https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt

Universally Unique Identifier (UUID)

- Identifiers are used to identify all **attributes**.[2]
- UUID are specified by the BT SIG. They are 128-bits, but 16- or 32- bits represent the unique information.
- Can be used to encode information, such as manufacturer, project ID, firmware revision, etc.
- Online generators exist to create valid UUIDs.[3]
- Example: `b562bf0c-0039-418e-9756-8b271b33d5be`
- The "base" UUID can be set for a device and then incremented for custom services not in the standard GATT.

[2] Services, characteristics, & descriptors.
[3] `https://www.uuidgenerator.net/`

Duke | BME

GATT Workflow

- UUIDs can be *discovered* or explicitly *found*.
- Service characteristics can be *discovered*.
- Data can be read (server $\rightarrow$ client) using either a UUID or **handle** (similar to an "alias").
- Data can be written to the server using the handle (with or without response).
- Data streams are limited / dictated by the Maximum Transfer Unit (MTU), which can be extended, but with tradeoffs.

Duke | BME

What are notifications?

- A **notification** can be sent by the server when a characteristic is available for the client (e.g., data stored in the characteristic has been updated). The client is responsible for then reading the characteristic.
- Notifications avoid the client having to periodically read a characteristic to determine if a value has been updated or is available.
- Notifications are similar to an ISR for the client to know to send a read request to the server.
- An **indication** is a notification that requires a response from the client.

prf.conf (BT)

```
# BLE
CONFIG_BT=y
CONFIG_BT_PERIPHERAL=y
CONFIG_BT_DEVICE_NAME="YourDeviceName"
CONFIG_BT_DEVICE_APPEARANCE=0
CONFIG_BT_MAX_CONN=1
CONFIG_BT_LL_SOFTDEVICE=y
CONFIG_BT_BAS=y  // Battery Service GATT
CONFIG_BT_SETTINGS=y
CONFIG_SETTINGS_RUNTIME=y
CONFIG_SETTINGS=y
CONFIG_SETTINGS_NONE=y
```

prf.conf (DIS)

```
CONFIG_BT_DIS=y  // Device Information Service GATT
CONFIG_BT_DIS_SETTINGS=y
CONFIG_BT_DIS_STR_MAX=25  // allow strings to be 25 char (or longer if desired)
CONFIG_BT_DIS_MODEL="YourDeviceModel"
CONFIG_BT_DIS_MANUF="YourManufacturerName"
CONFIG_BT_DIS_SERIAL_NUMBER=y
CONFIG_BT_DIS_SERIAL_NUMBER_STR="0001"  // serial number
CONFIG_BT_DIS_FW_REV=y
CONFIG_BT_DIS_FW_REV_STR="0.9.0"  // semantic version numbering
CONFIG_BT_DIS_HW_REV=y
CONFIG_BT_DIS_HW_REV_STR="0.9.0"  // semantic version numbering
CONFIG_BT_DIS_SW_REV=y
CONFIG_BT_DIS_SW_REV_STR="0.9.0"  // semantic version numbering
CONFIG_BT_DIS_PNP=y  // plug-n-play (vendor ID)
CONFIG_BT_DIS_PNP_VID_SRC=1
CONFIG_BT_DIS_PNP_VID=0x02DF  // vendor ID
```

Libraries to Include

```
#include <zephyr/bluetooth/bluetooth.h>
#include <zephyr/bluetooth/uuid.h>
#include <zephyr/bluetooth/gatt.h>
#include <zephyr/bluetooth/hci.h> // host controller interface
#include <zephyr/bluetooth/services/bas.h>  // battery service GATT
#include <zephyr/settings/settings.h>
```

Duke | BME

UUID

```c
/* UUID of the Remote Service */
// Project ID: 065 (3rd entry)
// MFG ID = 0x02DF (4th entry)
#define BT_UUID_REMOTE_SERV_VAL \
        BT_UUID_128_ENCODE(0xe9ea0000, 0xe19b, 0x0065, 0x02DF, 0xc7907585fc48)

/* UUID of the Data Characteristic */
#define BT_UUID_REMOTE_DATA_CHRC_VAL \
        BT_UUID_128_ENCODE(0xe9ea0001, 0xe19b, 0x0065, 0x02DF, 0xc7907585fc48)

/* UUID of the Message Characteristic */
#define BT_UUID_REMOTE_MESSAGE_CHRC_VAL \
        BT_UUID_128_ENCODE(0xe9ea0002, 0xe19b, 0x0065, 0x02DF, 0xc7907585fc48)
```

UUID

- Define custom service and two characteristics
- "Data" characteristic (server sends data)
- "Message" characteristic (server receives messages)

```
#define BT_UUID_REMOTE_SERVICE
↪                              BT_UUID_DECLARE_128(BT_UUID_REMOTE_SERV_VAL)
#define BT_UUID_REMOTE_DATA_CHRC
↪                      BT_UUID_DECLARE_128(BT_UUID_REMOTE_DATA_CHRC_VAL)
#define BT_UUID_REMOTE_MESSAGE_CHRC
↪              BT_UUID_DECLARE_128(BT_UUID_REMOTE_MESSAGE_CHRC_VAL)
```

BLE Data Structures

```c
// capture notifications state
enum bt_data_notifications_enabled {
    BT_DATA_NOTIFICATIONS_ENABLED,
    BT_DATA_NOTIFICATIONS_DISABLED,
};

// create a struct of all BLE callbacks
struct bt_remote_srv_cb {
    void (*notif_changed)(enum bt_data_notifications_enabled status);
    void (*data_rx)(struct bt_conn *conn, const uint8_t *const data, uint16_t
     ↪  len);
};
```

Duke | BME

Preprocessor Definitions

```
static K_SEM_DEFINE(bt_init_ok, 1, 1); // blocking thread semaphore

#define DEVICE_NAME CONFIG_BT_DEVICE_NAME
#define DEVICE_NAME_LEN (sizeof(DEVICE_NAME)-1)
#define BLE_DATA_POINTS 600 // limited by MTU

static struct bt_remote_srv_cb remote_service_callbacks;
enum bt_data_notifications_enabled notifications_enabled;
```

Duke | BME

Preprocessor: Advertising & Scan Response

```
/* Advertising data */
static const struct bt_data ad[] = {
    BT_DATA_BYTES(BT_DATA_FLAGS, (BT_LE_AD_GENERAL | BT_LE_AD_NO_BREDR)),
    BT_DATA(BT_DATA_NAME_COMPLETE, DEVICE_NAME, DEVICE_NAME_LEN)
};
/* Scan response data */
static const struct bt_data sd[] = {
    BT_DATA_BYTES(BT_DATA_UUID128_ALL, BT_UUID_REMOTE_SERV_VAL),
};
```

Function Declarations

```
static ssize_t read_data_cb(struct bt_conn *conn, const struct bt_gatt_attr
↪   *attr, void *buf, uint16_t len, uint16_t offset);
void data_ccc_cfg_changed_cb(const struct bt_gatt_attr *attr, uint16_t value);
static ssize_t on_write(struct bt_conn *conn, const struct bt_gatt_attr *attr,
↪   const void *buf, uint16_t len, uint16_t offset, uint8_t flags);
void bluetooth_set_battery_level(int level, int nominal_batt_mv);
uint8_t bluetooth_get_battery_level(void);
```

Setup BLE Services

```
BT_GATT_SERVICE_DEFINE(remote_srv,
BT_GATT_PRIMARY_SERVICE(BT_UUID_REMOTE_SERVICE),
GATT_CHARACTERISTIC(BT_UUID_REMOTE_DATA_CHRC,
        BT_GATT_CHRC_READ | BT_GATT_CHRC_NOTIFY,
        BT_GATT_PERM_READ,
        read_data_cb, NULL, NULL),
GATT_CCC(data_ccc_cfg_changed_cb, BT_GATT_PERM_READ | BT_GATT_PERM_WRITE),
GATT_CHARACTERISTIC(BT_UUID_REMOTE_MESSAGE_CHRC,
        BT_GATT_CHRC_WRITE_WITHOUT_RESP,
        BT_GATT_PERM_WRITE,
        NULL, on_write, NULL),
);
```

Duke | BME

BLE Callbacks

```
void data_ccc_cfg_changed_cb(const struct bt_gatt_attr *attr, uint16_t value)
{
    bool notif_enabled = (value == BT_GATT_CCC_NOTIFY);
    LOG_INF("Notifications: %s", notif_enabled? "enabled":"disabled");

    notifications_enabled = notif_enabled?
    ↪  BT_DATA_NOTIFICATIONS_ENABLED:BT_DATA_NOTIFICATIONS_DISABLED;

    if (remote_service_callbacks.notif_changed) {
        remote_service_callbacks.notif_changed(notifications_enabled);
    }
}
```

BLE Callbacks

```
static ssize_t read_data_cb(struct bt_conn *conn, const struct bt_gatt_attr
↪   *attr, void *buf, uint16_t len, uint16_t offset)
{
    return bt_gatt_attr_read(conn, attr, buf, len, offset, &compliance_data,
    ↪   sizeof(compliance_data));
}

static ssize_t on_write(struct bt_conn *conn, const struct bt_gatt_attr *attr,
↪   const void *buf, uint16_t len, uint16_t offset, uint8_t flags)
{
    LOG_INF("Received data, handle %d, conn %p", attr->handle, (void *)conn);

    if (remote_service_callbacks.data_rx) {
        remote_service_callbacks.data_rx(conn, buf, len);
    }
    return len;
}
```

BLE Callbacks

```
void on_sent(struct bt_conn *conn, void *user_data)
{
    ARG_UNUSED(user_data);
    LOG_INF("Notification sent on connection %p", (void *)conn);
}

void bt_ready(int ret)
{
    if (ret) {
        LOG_ERR("bt_enable returned %d", ret);
    }

    // release the thread once initialized
    k_sem_give(&bt_init_ok);
}
```

Duke | BME

BLE Callbacks

```
int send_data_notification(struct bt_conn *conn, uint8_t *value, uint16_t length)
{
    int ret = 0;

    struct bt_gatt_notify_params params = {0};
    const struct bt_gatt_attr *attr = &remote_srv.attrs[2];

    params.attr = attr;
    params.data = &value;
    params.len = length;
    params.func = on_sent;

    ret = bt_gatt_notify_cb(conn, &params);

    return ret;
}
```

Initialize BLE Connection

```
int bluetooth_init(struct bt_conn_cb *bt_cb, struct bt_remote_srv_cb *remote_cb)
{
    LOG_INF("Initializing Bluetooth");

    if ((bt_cb == NULL) | (remote_cb == NULL)) {
        return -NRFX_ERROR_NULL;
    }
    bt_conn_cb_register(bt_cb);
    remote_service_callbacks.notif_changed = remote_cb->notif_changed;
    remote_service_callbacks.data_rx = remote_cb->data_rx;

    int ret = bt_enable(bt_ready);
    if (ret) {
        LOG_ERR("bt_enable returned %d", ret);
        return ret;
    }
```

Initialize BLE Connection (continued)

```
    // hold the thread until Bluetooth is initialized
    k_sem_take(&bt_init_ok, K_FOREVER);

    if (IS_ENABLED(CONFIG_BT_SETTINGS)) {
        settings_load();
    }

    ret = bt_le_adv_start(BT_LE_ADV_CONN, ad, ARRAY_SIZE(ad), sd,
    ↪   ARRAY_SIZE(sd));
    if (ret) {
        LOG_ERR("Could not start advertising (ret = %d)", ret);
        return ret;
    }

    return ret;
}
```

Battery Level GATT

```
// normalized_level comes from an ADC reading of the battery voltage

// this function populates the BAS GATT with information
err = bt_bas_set_battery_level((int)normalized_level);
if (err) {
    LOG_ERR("BAS set error (err = %d)", err);
}

// this function retrieves BAS GATT with information
battery_level =  bt_bas_get_battery_level();
```

https://docs.zephyrproject.org/apidoc/latest/group__bt__bas.html

Duke | BME

main.c: Declarations

```
/* Declarations */
static struct bt_conn *current_conn;
void on_connected(struct bt_conn *conn, uint8_t ret);
void on_disconnected(struct bt_conn *conn, uint8_t reason);
void on_notif_changed(enum bt_data_notifications_enabled status);
void on_data_rx(struct bt_conn *conn, const uint8_t *const data, uint16_t len);
```

main.c: Data Structures

```
struct bt_conn_cb bluetooth_callbacks = {
    .connected = on_connected,
    .disconnected = on_disconnected,
};

struct bt_remote_srv_cb remote_service_callbacks = {
    .notif_changed = on_notif_changed,
    .data_rx = on_data_rx,
};
```

Duke | BME

main.c: Callbacks

```
void on_data_rx(struct bt_conn *conn, const uint8_t *const data, uint16_t len)
{
    uint8_t temp_str[len+1];
    memcpy(temp_str, data, len);
    temp_str[len] = 0x00; // manually append NULL character at the end

    LOG_INF("BT received data on conn %p. Len: %d", (void *)conn, len);
    LOG_INF("Data: %s", temp_str);
}

void on_connected(struct bt_conn *conn, uint8_t ret)
{
    if (ret) { LOG_ERR("Connection error: %d", ret); }
    LOG_INF("BT connected");
    current_conn = bt_conn_ref(conn);
}
```

main.c: Callbacks

```c
void on_disconnected(struct bt_conn *conn, uint8_t reason)
{
    LOG_INF("BT disconnected (reason: %d)", reason);
    if (current_conn) {
        bt_conn_unref(current_conn);
        current_conn = NULL;
    }
}

void on_notif_changed(enum bt_data_notifications_enabled status)
{
    if (status == BT_DATA_NOTIFICATIONS_ENABLED) {
        LOG_INF("BT notifications enabled");
    }
    else {
        LOG_INF("BT notifications disabled");
    }
}
```

main.c: main()

```
/* Initialize Bluetooth */
int err = bluetooth_init(&bluetooth_callbacks, &remote_service_callbacks);
if (err) {
    LOG_ERR("BT init failed (err = %d)", err);
}

// do stuff, write to "data" array

// send a notification that "data" is ready to be read...
err = send_data_notification(current_conn, data, 1);
if (err) {
    LOG_ERR("Could not send BT notification (err: %d)", err);
}
else {
    LOG_INF("BT data transmitted.");
}
```

References

- Adafruit: Introduction to Bluetooth Low Energy
- Wikipedia: BLE
- BLE: Mesh Networking
- Zephyr: Battery Service (BAS)
- Nordic DevAcademy: BLE Fundamentals