

# Introducing Bluetooth Low Energy

Kolbjørn Austreng

## i Introduction

Wired protocols are all well and nice - and although CAN is a relatively old protocol now partly superseded by CAN FD (ISO 11898); it is still widely used today. Nevertheless, wireless protocols are becoming increasingly popular wherever they can be used, due to their low cost, and ease of use.

One such protocol - Bluetooth 4.2, or Bluetooth Low Energy (BLE for short), is almost certainly supported by your smart phone; as well as a plethora of different devices that we interact with daily. This makes it an ideal candidate for introducing wireless connectivity to your project - of course provided you can live with the interference present in the 2.4 GHz ISM band.

Many of the students taking this course will previously have had the course "Embedded Systems" (TTK4235), in which the *micro:bit* development board was used during the labs. This is an excellent platform for introducing the Bluetooth Low Energy stack, and we will be using the micro:bit platform in this course as well. TTK4235 is no prerequisite, and it should be straight forward to follow this lab assignment without having gone through that course.

If no one on your group has a micro:bit, you may talk to one of the student assistants. It is also possible to use other hardware, such as an nRF52DK if you happen to have one, but bear in mind that other hardware platforms support different Application Programming Interfaces (APIs) for the Bluetooth stack.

This assignment was developed on Linux. It should be relatively straight forward to follow along with other platforms using the GNU toolchain as well, but keep in mind that you will be slightly more on your own in that case.

## ii Initial Setup of Hardware and Toolchain

The following hardware configuration is only necessary for those of you who have not already done so through TTK4235. However, it might still be necessary to set up the toolchain, as nothing is certain when it comes to the computers at the real time lab.

### ii.a Hardware Configuration

By default, the micro:bit comes preloaded with a custom Device Abstraction Layer (DAL) that allows programming the device through Scratch, Micropython, or JavaScript. This is well for simple demonstrations, but yields no control of the low level hardware. To get around this limitation, we will upgrade the flasher circuit to use the BBC micro:bit J-Link OB Firmware.

1. Download the J-Link OB Firmware from Segger.com<sup>1</sup>.
2. Start with the micro:bit unplugged. While holding down the Reset button (next to the USB connector), connect it to the computer.
3. The micro:bit should now have mounted itself as "MAINTENANCE" on the file system. You can now let go of the Reset button.
4. Move the .hex-file that you downloaded into "MAINTENANCE".
5. The micro:bit should now automatically remount after a few seconds; this time as "MICROBIT".

### ii.b Toolchain Configuration

To compile and flash code for the micro:bit, we need to install a toolchain. For compilation, GCC is a natural choice. Since the heart of the micro:bit is an nRF51822, we will be using J-Link and nrfjprog to flash the board.

1. Call `nrfjprog --version` to see if the toolchain is already set up on the computer. If it is, you may skip the remaining steps.
2. Call `sudo apt install gcc-arm-none-eabi` to install the GCC ARM compiler.
3. Go to [nordicsemi.com](https://www.nordicsemi.com)<sup>2</sup> and download the newest version of the nRF command line tools.

---

<sup>1</sup>[https://www.segger.com/downloads/jlink#BBC\\_microbit](https://www.segger.com/downloads/jlink#BBC_microbit)

<sup>2</sup><https://www.nordicsemi.com/eng/nordic/Products/nRF51822/nRF5x-Command-Line-Tools-Linux64/51386>

4. The download contains the programs `nrfjprog` and `mergehex`, both must be available in the Linux `PATH` variable. This can be done in many ways, here is one:
  - (a) Extract the `nrfjprog`- and `mergehex` folders to the home folder.
  - (b) Add the following string at the bottom of the `~/.bashrc` file:  
`"export PATH=$HOME/nrfjprog:$HOME/mergehex:$PATH"` (do not include the quotation marks)
  - (c) Finally, since `~/.bashrc` is read once when a terminal opens, you need to close and reopen all terminal windows that you might have active.
5. Go to [segger.com](https://www.segger.com)<sup>3</sup> and download the J-Link Software and Documentation pack. You should choose the Linux 64 bit version, DEB installer.
6. Once you have downloaded the DEB package, navigate to the download location with a terminal, and call the following command to install the package: `sudo dpkg -i JLink_Linux[...].deb`

### ii.c Hardware and Toolchain Verification

First off, you should call `nrfjprog --version` to see if the toolchain is set up. If this command complains, you will need to repeat section ii.b.

On Blackboard, you can find an example `.hex`-file, aptly named "example.hex". To flash this file to the micro:bit, navigate to the directory where you have the hex, and call:

```
nrfjprog -f nrf51 --chiperase --program example.hex --reset
```

The micro:bit should now start broadcasting the name "u:bit" over Bluetooth. To verify this, you may use your phone, as described in section iii.

### iii Use your phone for testing

You are free to use whatever to check your Bluetooth connection, but an easy choice is to download the "nRF Connect" app for your smart phone. This lab assignment will use that app.

When you open nRF Connect, your phone will start scanning for advertising packets, as seen in figure 1. You can adjust the RSSI filter (signal strength) to only show devices that are close to you. In the image, this filter is set to  $-80$  dBm, but setting it to around  $-60$  dBm is probably better if

---

<sup>3</sup><https://www.segger.com/downloads/jlink/#J-LinkSoftwareAndDocumentationPack>

there are a lot of Bluetooth devices nearby.

When you connect to the micro:bit, you will see something similar to figure 2. The names "LED Matrix" and "Buttons" will not show up by default, but may be assigned by you, by long-pressing on the *characteristic* and editing the name. Either way, the name of the characteristic is merely cosmetic.

By uploading a byte different than 0x00 to the "LED Matrix" characteristic, the LED matrix should turn on; and it should turn off when you upload 0x00.

The "Buttons" characteristic contains the state of the two micro:bit tactile buttons. You may read their state *once* by pressing the download icon - or you may *subscribe* to state updates by pressing the "notify" icon (The solid line with three arrows pointing down). With subscriptions turned on, you should see the state of the buttons update on the screen as you press- and release the buttons.

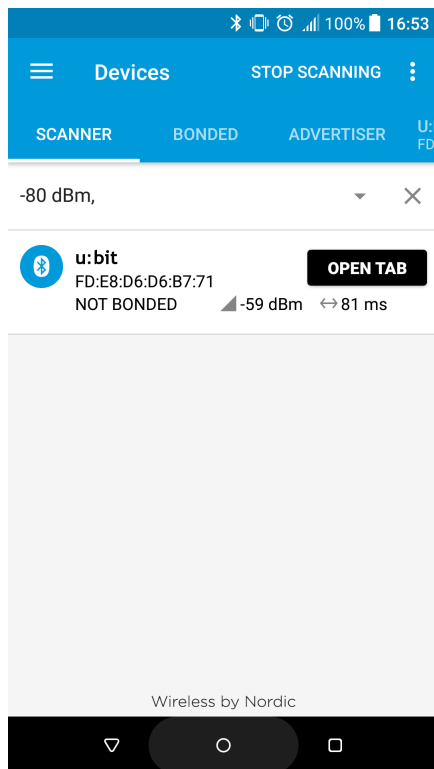


Figure 1: nRF Connect scan.

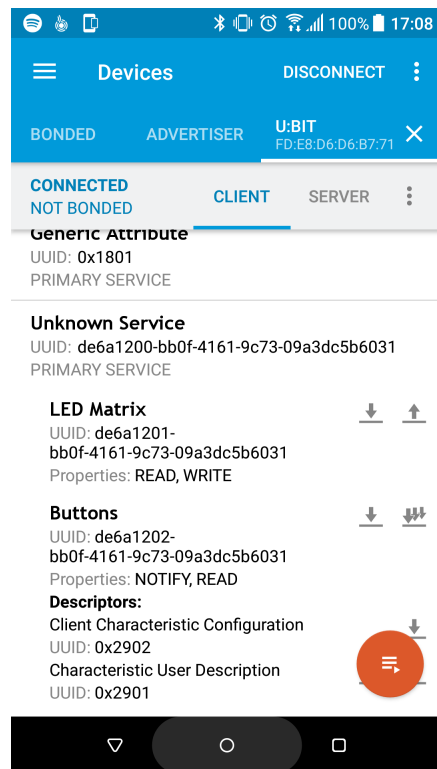


Figure 2: Example services.

## 1 Heads up

Bluetooth Low Energy (BLE) is a messy beast to tackle, so it might be smart to skim appendix A as you go through this lab assignment. Don't fret if you do not get all the details however - some things are better understood when coded, rather than when read.

Finally, an honest word from the author of this assignment: Sometimes, documentation about Bluetooth Low Energy can be a little "Meh". Feeling like there should be an easier way to do Bluetooth is completely normal - just remember to breathe with the stomach, and bite the pillow when the going gets rough.

Do enjoy :)

## 2 First steps - GAP

The gatekeeper of Bluetooth is the *Generic Access Profile*. This is the part of Bluetooth that handles broadcasting, discovery, the setting up- and managing connections, as well as negotiating security levels.

Although GAP is used to establish connections, GAP itself is connectionless, and pretty simple. The bottom line is this: When advertising, a Bluetooth device sends out an *Advertising Packet* of 31 bytes every advertising interval - which can be anything between 20 milliseconds and 10.24 seconds.

In addition to this, anyone observing the advertiser can send a *scan request* to the device. Upon receiving such a scan request, the device *may* respond with a *Scan Response Packet*, which is also 31 bytes and can contain different information than the Advertising Packet. Note that the device is not required to respond to scan requests, however.

GAP happens entirely on the Bluetooth channels 37, 38, and 39, which are reserved for advertising - as illustrated in figure 9 in appendix A. The reason that exactly these channels were chosen for advertising is to avoid most of the interference from WiFi, which is most noticeable on the channels 0-9, 11-20, and 23-32.

To make matters a little more interesting, the Bluetooth 4.2 specification specifies 4 different ways a device can advertise. The nuances of this is not terribly important for this course, but they are listed here for the interested:

- **ADV\_IND:** "Connectable undirected advertising". This is the regular way to advertise. Any other device can request a scan response or a connection.
- **ADV\_DIRECT\_IND:** "Connectable directed advertising". This is used to ask a specific central for a connection. The packet is a broadcast packet, but scanners will ignore it if the peer address does not match them. In addition, the advertiser itself will ignore all scan requests.
- **ADV\_SCAN\_IND:** "Scannable undirected advertising". In this mode, the advertiser will ignore all connection requests, but may respond to scan requests.
- **ADV\_NONCONN\_IND:** "Non-connectable undirected advertising". This is essentially just an information beacon; the advertiser will ignore all scan- and connection requests, and will simply spew out its own Advertising Packets until the end of time.

## 2.1 Advertising Packet Format

Even though the Advertising Packet (and Scan Response Packet) consists of 31 bytes, not all are usable by the application programmer. This is because the packet is subdivided into a number of *AD Structs* - or "Advertising Data Structures". Actually, truth be told, each Advertising Packet may *really* send out 47 bytes over the air each advertising interval, but a lot of this is overhead, as illustrated in figure 3. The squares with just numbers in them denote fields that are necessary, but not of direct interest to us.

Despite the fact that we do not really get to use all the 31 bytes in the advertising data, we must still hand craft the entire field to comply with with the Bluetooth 4.2 spec in our code. In fact, you can see the advertising data used in the example program in listing 1.

```
static uint8_t adv_data[] = {
    6, BLE_GAP_AD_TYPE_COMPLETE_LOCAL_NAME,
    'u', ':', 'b', 'i', 't',
    3, BLE_GAP_AD_TYPE_16BIT_SERVICE_UUID_COMPLETE,
    0x0D, 0xF0
};
uint8_t adv_data_length = 11;
```

Listing 1: The advertising data field from the example hex.

Note that the advertising data does not have to use all the 31 bytes available.

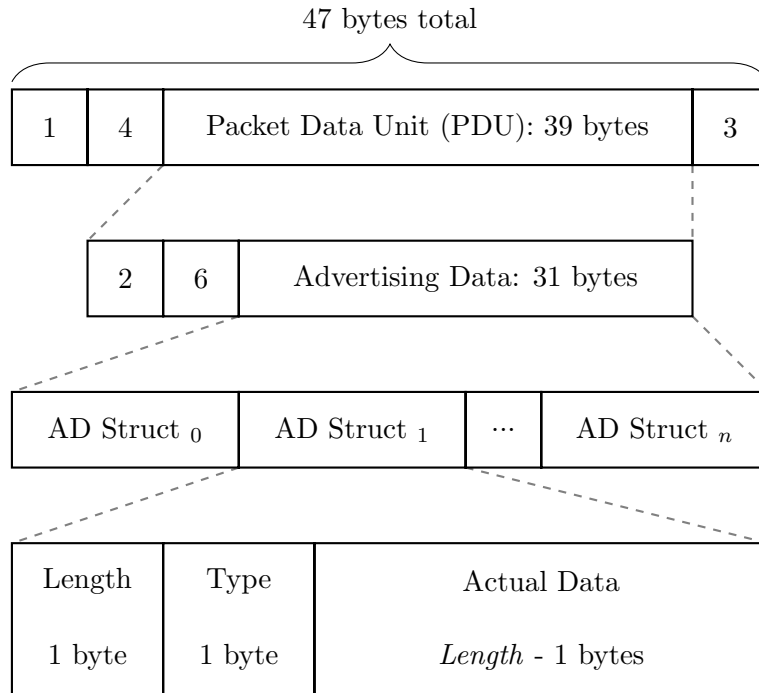


Figure 3: Breakdown of each GAP advertising transmission unit.

We are free to use fewer bytes, and therefore transmit less data over the air each advertising interval. This allows us to reduce power consumption a little if we have a really resource constrained application.

## 2.2 Write the code

You have already been handed out some C-files that you may build on. If you happen to be a masochist, you are of course free to discard this code and start from scratch. Otherwise, follow along.

In the `main` function, you will see a call to `bluetooth_init()`. This is simply a wrapper around two calls to first initialize the Nordic SoftDevice, before enabling the Bluetooth Low Energy stack with some sane parameters. You do not need to worry about this; instead, you will be implementing the `bluetooth_gap_advertise_start()` function, which is responsible for setting up the AD structs (figure 3), and then enabling advertising.

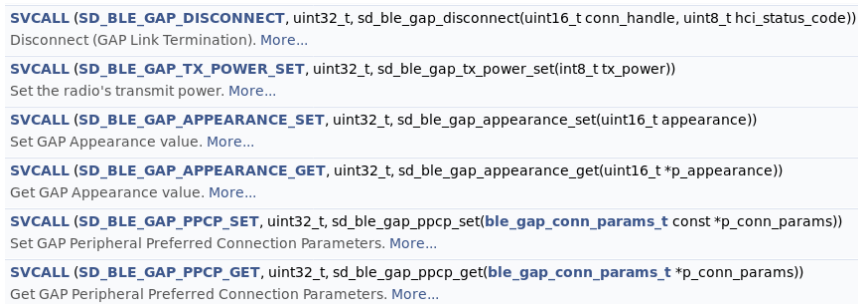
The first step is to open up the documentation for the Nordic SoftDevice. Open up `softdevice/s130_nrf51_2.0.1_API/html/index.html` in a web browser. From there, navigate to the **Modules** tab, then click **Generic Access Profile (GAP)**, and then click on **Functions**.

You will now be greeted with a fairly intimidating looking list of things. If you are feeling a little bit overwhelmed at this point, read section 2.2.1. If you are feeling confident, feel free to skip ahead.

### 2.2.1 What on earth is all this?

Most of the documentation for the Nordic SoftDevices is handed out as specially formatted comments in the header files of the SoftDevice API. You can then generate a more friendly format, such as an HTML web site, using a handy program called **doxygen**. This has already been done for you, but it is pretty easy to do yourselves as well.

Doxygen is very handy for integrating documentation into code, and often leads to more up to date documentation in general. The only flip side is that the output is slightly less human friendly than it would otherwise have been, had it been hand crafted by an actual human being. Do not despair though, let us dissect this cryptic output together:



```
SVCALL (SD_BLE_GAP_DISCONNECT, uint32_t, sd_ble_gap_disconnect(uint16_t conn_handle, uint8_t hci_status_code))
Disconnect (GAP Link Termination). More...

SVCALL (SD_BLE_GAP_TX_POWER_SET, uint32_t, sd_ble_gap_tx_power_set(int8_t tx_power))
Set the radio's transmit power. More...

SVCALL (SD_BLE_GAP_APPEARANCE_SET, uint32_t, sd_ble_gap_appearance_set(uint16_t appearance))
Set GAP Appearance value. More...

SVCALL (SD_BLE_GAP_APPEARANCE_GET, uint32_t, sd_ble_gap_appearance_get(uint16_t *p_appearance))
Get GAP Appearance value. More...

SVCALL (SD_BLE_GAP_PPCP_SET, uint32_t, sd_ble_gap_ppcp_set(ble_gap_conn_params_t const *p_conn_params))
Set GAP Peripheral Preferred Connection Parameters. More...

SVCALL (SD_BLE_GAP_PPCP_GET, uint32_t, sd_ble_gap_ppcp_get(ble_gap_conn_params_t *p_conn_params))
Get GAP Peripheral Preferred Connection Parameters. More...
```

Figure 4: SoftDevice GAP documentation excerpt.

In figure 4, you can see an excerpt of the SoftDevice GAP documentation. Each line starts with the macro **SVCALL** - which stands for "Supervisor Call". This is simply an implementation detail, and not something that we have any control over.

The actual function that we want to call is listed after this macro. For example, in the line **SVCALL(SD\_BLE\_GAP\_TX\_POWER\_SET, uint32\_t [...])**, the function that is available to us is called

```
sd_ble_gap_tx_power_set(int8_t tx_power);
```

Underneath the **SVCALL** line, you can see a short description of what this function does, along with a link that will give you more detailed information about how you should use the function.



Many functions will take a pointer to a custom defined struct, instead of several smaller parameters. This is again due to implementation details. In some such cases, the memory that the pointer points to, must remain available after the function has executed. In those cases, manually allocating memory or marking the variable as static is recommended.

Finally, it might be worth pointing out some of the coding standard used in the SoftDevice public API:

1. There is liberal use of the `const` keyword, to signal the compiler that a variable should not be changed. This is good practice - especially when dealing with things that can easily break, such as Bluetooth code.
2. Pointer variables are always prefixed with `p_`, and in the rare cases of pointer-to-pointer, the prefix `pp_` is used. This is heartily recommended, as it is a quick way to kill off some otherwise hard-to-find bugs.
3. In the documentation of each function, the parameters will be marked with either `[in]`, `[out]`, or `[in,out]`. This tells you whether the function intends to modify the input parameter or not. Parameters marked `[in]` will not be modified, whereas `[out]` and `[in,out]` will.

### 2.2.2 Set the advertising data

The first step you need to do is setting the Advertising Packet data content. In the `bluetooth.c` file, you will find an incomplete function called `bluetooth_gap_advertise_start()`. Your task is to fill out the `static uint8_t adv_data[]` array with data according to one or more AD structs, as illustrated in figure 3.

The function of most interest to you is called `sd_ble_gap_adv_data_set`. You are free to skip the Scan Response Packet, in which case you simply set `p_sr_data` to `NULL`, and `srdlen` to 0.

For a complete list of the AD Struct types that this SoftDevice supports, you may check out [Generic Access Profile \(GAP\) → Defines → GAP Advertising and Scan Response Data format](#).



If you are interested in using both a complete local name, as well as short local name, there is one caveat you need to be aware of: A short local name must be a continuous subset of the complete local name, starting at the beginning of the complete name - as per the Bluetooth Core Specification Supplement version 7. No worries, it took me ages to figure out as well.

### 2.2.3 Start advertising

After you have set the advertising data, you need to activate advertising. Check out the documentation for `sd_ble_gap_adv_start`. Note that this function call takes a pointer to a `ble_gap_adv_params_t` struct.

To get the fields of this struct, follow the link in the documentation of the `sd_ble_gap_adv_start` function. New tabs for each link you have to follow is advisable, but remember to close the ones you no longer need - otherwise you *will* be overrun with information quickly.

Hint: Only the fields `type` and `interval` are of particular interest, while the rest of the struct can be all zeros. A neat way to do this without manually setting each field is using the `memset` function, provided by `<string.h>`:

```
ble_gap_adv_params_t adv_params;
memset(&adv_params, 0, sizeof(adv_params));
// Then set only the interesting fields
```

To get the documentation for `memset`, you may call `man 3 memset` in a terminal, but the long and short of it is this: Fill `sizeof(adv_params)` number of bytes with all zeros, starting at the address of `adv_params`.

You stand free to choose the advertising type, but you are most likely interested in using `BLE_GAP_ADV_TYPE_ADV_IND`. You can find a complete list of advertising types under **Generic Access Profile (GAP) → Defines → GAP Advertising types**.

### 2.3 Print to check your code

All the SoftDevice calls will return 0 (which is the value of  `NRF_SUCCESS` ) upon successful execution. You can print this over UART to the computer by using the provided `ubit_uart_print` function - which works the same way `printf` does, except that `ubit_uart_print` only accepts integers. An example invocation looks like this:

```
ubit_uart_init(); // Call once
ubit_uart_print("My favorite number is %d\n\r", 2);
```

To read the micro:bit UART output, you may use any serial terminal program that you choose, with the baudrate 9600. The micro:bit will most likely mount as `/dev/ttyACM0`. An example using the `picocom` program looks like this:

```
picocom -b 9600 /dev/ttyACM0
```



If you have any other USB UART interfaces connected to the computer, the micro:bit may mount as `/dev/ttyACM1` instead. You can call `dmesg --follow` and then connect the micro:bit if you want to know for sure where it mounts.

## 2.4 Check with your phone

If your program compiles and runs, you may check with your phone to see if there is anything on the air. Use the nRF Connect app and start scanning. Your micro:bit should now broadcast its name, along with any other AD Structs that you might have included.

If you made it, take a moment to savor your victory; writing Bluetooth Low Energy firmware is not a trivial task ;)

Note that if you try to connect to your micro:bit at this point, you will be met with very anticlimactic behavior: The default when a device is attempted connected to, is to stop advertising. However, since we have not implemented any connection acceptance yet, the device will simply turn off advertising and then idle - and then your phone will time out.

Finally, you might have noticed that the nRF Connect app reports a different advertising interval than you specified. For example, if you chose 80 ms in your code, you will typically see an actual advertising interval of around 85 ms. This is because the Bluetooth spec will add a random delay to the specified interval, in order to reduce the likelihood of packet collisions.

## 3 Get your game on - GATT

GAP is cool and all, but the real workhorse of Bluetooth is the GATT; the *Generic Attribute Profile*. Unlike the GAP, which only deals with details regarding interactions between devices, the GATT deals with data transfer.

Since Bluetooth 4.1, it became possible to use the mysterious L2CAP (the *Logical Link Controller and Adaptation Protocol*) for higher throughput connection-oriented data transfer channels. Thus, you *could* use this layer for your data transfer directly. Nevertheless, at the time of writing, Bluetooth data transfer seems synonymous with GATT usage in some shape or form. For those interested, the Bluetooth Low Energy layers are illustrated in figure 8.

It is perfectly normal to be confused about GAP and GATT at this point, but the essence is this: GAP takes care of advertising, connections and security. GATT is a *profile* that uses a *protocol* named ATT (Attribute Protocol). Through a set of black magic, the GATT uses the ATT to provide a means for data transfer.

### 3.1 Protocols and Profiles

It is useful to have at least a high-level understanding of what the Bluetooth spec talks about when it discusses protocols and profiles. Still, a deep knowledge about these things should not be necessary to use Bluetooth Low Energy, so please do not fret if some of the concepts remain fuzzy.

#### 3.1.1 Protocols

In Bluetooth, a *protocol* is a layer that implements low-level details like packet formatting, routing, encoding, multiplexing, and some interface that allows data to be sent between peers.

The ATT (Attribute protocol) is an example of a protocol. The ATT does not have that much to it; it is merely a simple client-server stateless way to exchange data, presented as so-called "attributes". A Bluetooth device can be either a client, a server, or both - regardless of whether it is a master or a slave.

Inside the ATT, you find a set of attributes. Each attribute is a collection of four components:

1. A 16-bit attribute handle.
2. A number known as a Universally Unique Identifier (UUID).
3. A set of read- and write permissions.
4. The value, or the real data of the attribute.

Each time a client wants to read- or write an attribute value, it issues a request to the ATT server. The server must then fully process this single

request, before tending to any other requests.

When a client accesses an attribute, it uses the 16-bit attribute handle to identify which attribute it is trying to access. The UUID is a 128-bit (16 byte) number that describes the type or nature of the attribute data. When reading, the server does minimal checking; the client is responsible for interpreting and understanding the supplied data. This is unlike a write operation - which the server is free to reject if the client data does not fit the expected format.

### 3.1.2 Profiles

A *profile* is a recipe for how to use one or more protocols to serve a certain purpose, and achieve a particular goal. An example of such a profile is the GATT, or the Generic Attribute Profile.

The GATT heavily relies on the ATT, but to further convolute matters more, the GATT introduces some new concepts and terminology. The most important difference the GATT introduces is data organization. In GATT, data is ordered in a hierarchy of so-called *services*. A service is a collection of *characteristics*, and a set of relationships to other services. Characteristics are just conceptually similar pieces of data grouped together. See figure 5 for an illustration of this.

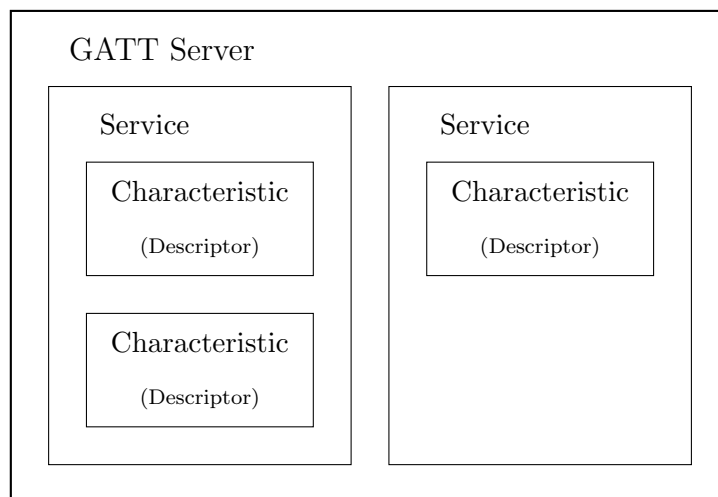


Figure 5: Conceptual illustration of a GATT server.

If you have a background in object-oriented programming, it might be helpful to think of services as *classes*, and characteristics as *member variables*.



The Bluetooth spec might have gone a *little* overboard with all the abbreviations. Remember to breathe with the stomach as you take it all in.

## 3.2 Create a Service

As you can probably guess from figure 5, the first thing we want to do to make GATT useful is to add a Service. Before we do this, we need to do some housekeeping; first off, include "ble\_gatts.h". Secondly, we need to allocate some persistent memory that can keep track of our new service. We can do this by creating a `static` struct at the top of `bluetooth.c`, like so:

```
static struct {  
    uint16_t conn_handle;  
    uint16_t service_handle;  
    ble_gatts_char_handles_t matrix_handles;  
} m_service_ubit;
```

This struct might look cryptic, but it is merely a representation of a - "Service (`m_service_ubit`), which has a single "Characteristic" (`matrix_handles`). The reason for the `_m` prefix on `m_service_ubit` is simply to denote to ourselves that this variable is reachable in the whole `module`. Getting into the habit of using simple prefixes like this is a very neat way of doing some simple sanity checks .

This is well and nice, but we still need to hold our horses a bit. Before we can actually add the service and characteristic, we need to give it some sort of ID, so the Bluetooth Stack knows how to address it. This is done by adding a new entry to the SoftDevice UUID table.

### 3.2.1 Add a Vendor Specific UUID

The Bluetooth spec defines a set of 16 bit UUIDs for the members of the Bluetooth Special Interest group, and some widely adopted services. Sadly, we are not on this list, so we need to create our own, longer, 128 bit UUID and use that instead.

In a terminal, call `uuidgen`. You will get a random string of hexadecimal values, for example "d8ff10af-2e1a-4bcf-b479-7f43a6995846". This is now our base UUID, that we will supply to the SoftDevice.

Since the `uuidgen` command yields a UUID in big endian, and the SoftDevice represents UUIDs in little endian, we technically have to enter our UUID in reverse. For the UUID above, you would enter this in `bluetooth.c`:

```
#define CUSTOM_UUID_BASE {{\n  0x46, 0x58, 0x99, 0xa6, 0x43, 0x7f, 0x79, 0xb4,\n  0xcf, 0x4b, 0x1a, 0x2e, 0xaf, 0x10, 0xff, 0xd8\n}}
```

In this specific case, it would not matter if you entered the UUID the wrong way, since this would simply correspond to a different randomly generated UUID. However, it would matter if you were implementing a "known and registered" service, so it is good to get into the habit of doing it correctly.

Since we will be implementing a single service with a single characteristic, we will need two more UUIDs after we have our base; one for the service, and one for the characteristic. Instead of specifying a full 128-bit UUID for these however, we simply need to supply two 16-bit "modifiers" to the base UUID. In this example, we are going to create a service called "ubit", with a characteristic called "matrix", so add the following to `bluetooth.c`:

```
#define CUSTOM_UUID_SERVICE_UBIT 0xdead\n#define CUSTOM_UUID_CHAR_MATRIX 0xbabe
```

Since we are using "vendor specific" UUIDs, the actual values of the modifiers `CUSTOM_UUID_SERVICE_UBIT` and `CUSTOM_UUID_CHAR_MATRIX` are not important, and you are free to set them to any 16-bit number of your liking, as long as they are not identical.

Now that we have decided what our UUIDs, will be, we must also tell this to the SoftDevice. In `bluetooth_gatts_start()`, you do this by running the following code:

```
ble_uuid128_t base_uuid = CUSTOM_UUID_BASE;\n\nble_uuid_t ubit_service_uuid;\nubit_service_uuid.uuid = CUSTOM_UUID_SERVICE_UBIT;\n\nerr_code = sd_ble_uuid_vs_add(\n    &base_uuid,\n    &ubit_service_uuid.type\n);\n
```

This is all well and fine. We have now added our own vendor specific service UUID to the SoftDevice UUID table. The only catch is of course that we have not added the actual service yet. No reason for despair though, we will do that next.

### 3.2.2 Add a Service to our UUID

We already have space for our service, namely `m_service_ubit`, that we declared earlier. Now we will tell the SoftDevice that this is what we intend to use for the service itself. Look into the `sd_ble_gatts_service_add` function, which you can find documentation for under **Generic Attribute Profile (GATT) Server → Functions**.

The type of the service should be primary. The `p_uuid` parameter should be a pointer to our service UUID. Finally, the `p_handle` parameter should be a pointer to the `service_handle` member of `m_service_ubit`.

It is probably smart to check that this call returns `NRF_SUCCESS` (defined to be 0) before moving on.

### 3.2.3 Add the Characteristic UUID

Now that we have added our service UUID and defined a handle for it, we need to populate that service with a characteristic. The first step is of course adding the UUID for our characteristic. This is done the same way as before:

```
ble_uuid_t matrix_uuid;
matrix_uuid.uuid = CUSTOM_UUID_CHAR_MATRIX;

err_code = sd_ble_uuid_vs_add(
    &base_uuid,
    &matrix_uuid.type
);
```

### 3.2.4 Add the Characteristic

A quick inventory check of what we have done so far: First off, we added our own vendor specific service UUID. Then we added the actual service to the SoftDevice. Now we have added a characteristic UUID to that service, but we have yet to add the "meat" of the characteristic itself. This is done by using the `sd_ble_gatts_characteristic_add` function, so if you are feeling adventurous, feel free to just read the documentation for that call and get going. Otherwise, read on.





The upcoming steps will probably seem completely arbitrary unless you have used a similar API before. Trust me, after a few times it starts making sense :)

A characteristic consists of zero or more *descriptors*, a set of *characteristic metadata*, and some *attribute metadata*. You can be pretty sure we have to deal with all of these. It is very easy to get lost in the process of setting this up, so I will not force you to do it completely on your own - still, you are encouraged to check out the documentation and try to understand what we are doing as you follow along.

First, we are going to set up the characteristic metadata, as such:

```
static uint8_t matrix_char_desc[] = {
    'L', 'E', 'D', ' ', 'M', 'a', 't', 'r', 'i', 'x'
};
ble_gatts_char_md_t matrix_char_md;
memset(&matrix_char_md, 0, sizeof(matrix_char_md));
matrix_char_md.char_props.read = 1;
matrix_char_md.char_props.write = 1;
matrix_char_md.p_char_user_desc = matrix_char_desc;
matrix_char_md.char_user_desc_max_size = 10;
matrix_char_md.char_user_desc_size = 10;
```

You can probably interpret most of this yourself. We create a characteristic that is both read- and writable. Then we describe the characteristic by adding the descriptor "LED Matrix", which is a UTF-8 encoded string that is not null-terminated. Since the string should not be terminated by a NULL, we have to supply the SoftDevice with the length of the descriptor as well. Some descriptors are allowed to be changed during operation, so we also supply a max length. In this case, we do not want the descriptor to be mutable, so we leave the *descriptor metadata* (`p_user_desc_md`) unset. You can see the documentation for the whole characteristic metadata struct by going to [Generic Attribute Profile \(GATT\) Server → Structures → ble\\_gatts\\_char\\_md\\_t](#).

Next up is making the attribute that our characteristic will refer to. Remember that GATT uses ATT behind the scenes, so it probably does not come as a surprise that we have to set some attribute metadata as well:

```
ble_gatts_attr_md_t matrix_attr_md;
memset(&matrix_attr_md, 0, sizeof(matrix_attr_md));
```

```

matrix_attr_md.read_perm.lv = 1;
matrix_attr_md.read_perm.sm = 1;
matrix_attr_md.write_perm.lv = 1;
matrix_attr_md.write_perm.sm = 1;
matrix_attr_md.vloc = BLE_GATTS_VLOC_USER;

```

Here, we are specifying the access restrictions to our attribute in the ATT server. The magic values for `lv` and `sm` set the *Security Mode* and *Level* of authentication needed to modify the attribute. When both are 1, we have an open link, so no security is needed. Even though this assignment will not touch on the security aspects of Bluetooth Low Energy, the Bluetooth spec boasts an impressive set of security considerations, like signing, encryption, and man-in-the-middle-attack protection.

In addition to setting the access restrictions for our attribute, we also tell the SoftDevice where the attribute value resides, that is, the **value location** of the attribute. The value `BLE_GATTS_VLOC_USER` means that the user (us) is responsible for ensuring that the attribute value is always stored in a valid location throughout the entire lifetime of the attribute. We will tend to this next.

Finally, we can create the attribute value. This needs to be reachable in the entire module, so add `static uint8_t m_matrix_attr_value = 0;` to the top of `bluetooth.c`. After that, we simply have to put this value and the metadata variables together, like so:

```

ble_gatts_attr_t matrix_attr;
memset(&matrix_attr, 0, sizeof(matrix_attr));
matrix_attr.p_uuid = &matrix_uuid;
matrix_attr.p_attr_md = &matrix_attr_md;
matrix_attr.init_len = 1;
matrix_attr.max_len = 1;
matrix_attr.p_value = &m_matrix_attr_value;

```

And then, to register our beautiful creation with our SoftDevice overlord, we call `sd_ble_gatts_characteristic_add`:

```

err_code = sd_ble_gatts_characteristic_add(
    m_service_ubit.service_handle,
    &matrix_char_md,
    &matrix_attr,
    &m_service_ubit.matrix_handles
);

```

Congratulations, your Bluetooth application is now approximately 100% cooler than before. However, it is still not connectable. We need to fix that, so read on.

## 4 Serve connections

We are nearly there. All that remains is to react to *connection events* and update the *connection handle* in our service. In the general case, you have to set up an event buffer, fill it up with incoming events, and process each of them in order.

In your case, this is already done for you. At the bottom of your `main` function, simply call `bluetooth_serve_forever()`.

If you inspect that function, you will see that all it essentially does, is listen for `BLE_GAP_EVT_CONNECTED` and `BLE_GAP_EVT_DISCONNECTED`. You can probably guess that these signify connection requests and disconnects through GAP. When these happen, we update the connection handle in our service accordingly.

In addition to this, the infinite loop in `bluetooth_serve_forever` sets and clears the LED matrix on the micro:bit according to the value of our characteristic.



Do not forget to initialize the LED matrix before calling `bluetooth_serve_forever`. Find an appropriate place to call the `init` function.

## 5 Instant gratification

You may now compile and flash your code. On your phone, you should now see something similar to figure 6 when you scan. If you try to connect, you should get something akin to figure 7. Note that we are able to use Unicode letters (encoded as UTF-8) in the GAP advertising packet, as seen in figure 6 with the umlaut `ü`. Even so, the nRF Connect app does not support all Unicode letters, so some of the more esoteric glyphs might cause the app to ignore your advertising packet.

Also note that we can recognize the UUID modifiers that we chose for the service and characteristic, as seen highlighted in figure 7.

You can now download the Characteristic User Description descriptor by clicking the download symbol underneath the descriptors category in figure 7. The descriptor then tells you that this is a characteristic for the LED

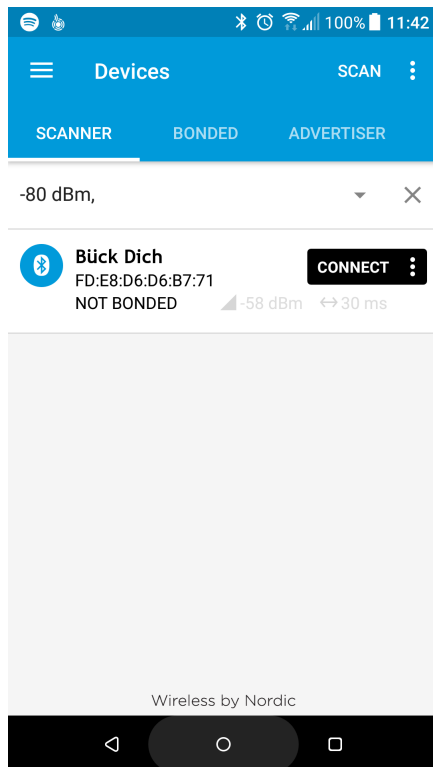


Figure 6: nRF Connect scan.

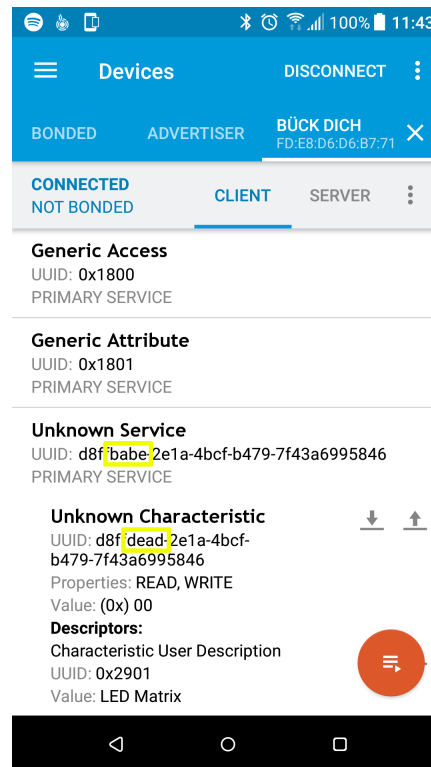


Figure 7: Custom service.

matrix.

As in the example program, you can toggle the LED matrix by writing a byte to the matrix characteristic.

## 6 What now?

Now you are free to integrate the micro:bit into your project however you wish, for example by broadcasting the score over Bluetooth to your phone. The remainder of this section will mainly deal with two things; firstly, I will give some tips on how to add notification support, so you do not have to manually read a characteristic when it changes. Secondly, I will give a few suggestions about how you can interface the micro:bit with the rest of your project.

### 6.1 Notifications

Suppose you want to add notifications for when you press buttons on the micro:bit. This can be done by either adding a new service, or adding a new

characteristic to our already existing service. Since our service is named "ubit" and deals with the micro:bit hardware, it makes the most sense to expand this service.

The first thing you would do, is to add new handles for the new service, by updating the definition of `m_service_ubit`:

```
static struct {
    uint16_t conn_handle;
    uint16_t service_handle;
    ble_gatts_char_handles_t matrix_handles;
    ble_gatts_char_handles_t button_handles;
} m_service_ubit;
```

Since we have two buttons on the micro:bit, it makes sense to use an array of two `uint8_ts` to represent their state - though you stand free to squeeze them both into one byte, if you want to be greedy with the resources:

```
static uint8_t m_button_press_a_b[2] = {0, 0};
static uint8_t m_button_press_a_b_previous[2] = {0, 0};
```

This is analogous to our `static uint8_t m_matrix_attr_value` from earlier. The reason we create two copies, is so we can notify the phone only when the button states update. This way, we do not needlessly transmit the button states unless they have changed since the last time we transmitted.

Adding the rest of the characteristic is similar to how we added the LED matrix characteristic, with some minor tweaks. The most obvious is that we do not want to make the attribute writable, so the attribute metadata will look like this instead:

```
ble_gatts_attr_md_t button_attr_md;
memset(&button_attr_md, 0, sizeof(button_attr_md));
button_attr_md.vloc = BLE_GATTS_VLOC_USER;
button_attr_md.read_perm.sm = 1;
button_attr_md.read_perm.lv = 1;
```

Next, we have to make sure the characteristic metadata matches the attribute metadata. There is one catch though; since the client (the phone) should be able to subscribe and unsubscribe to notifications, we need another piece of metadata describing whether the client wants notifications or not. This is stored in the Client Characteristic Configuration Descriptor, or the aptly named CCCD. The Bluetooth Special Interest Group dodged a bullet by not naming it the Client Characteristic Configuration Property.

This descriptor is managed by the SoftDevice, so we should allocate it to

the stack, rather than to user memory. After that, we need to tell the characteristic metadata to use the CCCD. Do not lose hope if this sounds like made-up gibberish; it should look like this:

```
ble_gatts_attr_md_t button_cccd_md;
memset(&button_cccd_md, 0, sizeof(button_cccd_md));
button_cccd_md.vloc = BLE_GATTS_VLOC_STACK;
button_cccd_md.read_perm.lv = 1;
button_cccd_md.read_perm.sm = 1;
button_cccd_md.write_perm.lv = 1;
button_cccd_md.write_perm.sm = 1;

static uint8_t button_char_desc[] = {
    'B', 'u', 't', 't', 'o', 'n', 's'
};
ble_gatts_char_md_t button_char_md;
memset(&button_char_md, 0, sizeof(button_char_md));
button_char_md.char_props.read = 1;
button_char_md.char_props.notify = 1;
button_char_md.p_char_user_desc = button_char_desc;
button_char_md.char_user_desc_max_size = 7;
button_char_md.char_user_desc_size = 7;
button_char_md.p_cccd_md = &button_cccd_md;
```

If you fill out the rest of the characteristic, and then add it to the SoftDevice by calling `sd_ble_gatts_characteristic_add`, you will see a new characteristic when you connect to the micro:bit (you might have to go to settings and "Refresh services" in the nRF Connect app). Still, the characteristic will not do anything, since we never update the button attribute value. This is next on our list.

### 6.1.1 Updating the client

To send notifications, we need to use the function `sd_ble_gatts_hvx`. HVX stands for "Handle Value X". I think we can all agree that the price for best function name does not go to `sd_ble_gatts_hvx`. Either way, here you can see an example invocation that transmits the state of our buttons:

```
if(m_service_ubit.conn_handle != BLE_CONN_HANDLE_INVALID){
    uint16_t notification_length = 2;

    ble_gatts_hvx_params_t hvx_params;
    memset(&hvx_params, 0, sizeof(hvx_params));
    hvx_params.handle =
        ↪ m_service_ubit.button_handles.value_handle;
```

```

        hvx_params.type = BLE_GATT_HVX_NOTIFICATION;
        hvx_params.p_len = &notification_length;
        hvx_params.p_data = m_button_press_a_b;

        sd_ble_gatts_hvx(
            m_service_ubit.conn_handle,
            &hvx_params
        );
    }

```

You would call this at the bottom of the infinite loop in `bluetooth_serve_forever`. Transmitting only when the button state has changed is simply a matter of comparing `m_button_press_a_b` and `m_button_press_a_b_previous`. This is up to you.

If you implement all this, you have essentially replicated the example program that was handed out, congratulations!

## 6.2 Interfacing to the rest of the project

When interfacing with the rest of your project, there is several ways to approach the problem. The two ways that I find most obvious are described here.

### 6.2.1 Direct connection

When interfacing with the rest of your project, there is several ways to approach the problem. The micro:bit supports both SPI and I<sup>2</sup>C (aka TWI), as well as UART. It is thus possible to connect the micro:bit to either node 1 or node 2, and interface it using one of the above protocols.



The micro:bit runs on 3.3V, while the rest of the project runs on 5V. Get a logic level converter, or a set of high frequency optocouplers if you want to follow this approach.

This would also require you to write your own SPI or I<sup>2</sup>C drivers. For those of you who have taken TTK4235, this should be a cake walk.

### 6.2.2 Indirect connection

Another approach is to simply use the host computer as a middle man. You may then use the micro:bit to communicate with the computer, which will in turn communicate with one of the nodes in your project.

This would require you to extend the UART driver for the micro:bit to also support receiving messages, as the driver handed out is only unidirectional. It would also require you to write a bridging program running on the computer. This bridging program simply has to forward messages between two serial ports. Considering the myriad of serial libraries out there, this task should be dirt simple - for example pySerial for Python, Nerves for Elixir, SerialPort for Ruby, etc.

### **6.2.3 Conclusion**

If you want to interface the micro:bit with the rest of your project using either of these methods, the amount of work should roughly be the same.

Using a direct connection removes the need for a man-in-the-middle, but it places more emphasis on hardware, as you have to correctly convert the logical levels in order to not damage the micro:bit.

Using the computer as a middle man, there is no extra hardware requirements at all, and the problem is reduced to "simply" writing correct code. In conclusion, it depends on what you are most comfortable working with ;)



## A Introduction to BLE

Bluetooth Low Energy, or Bluetooth 4.2 is a version of Bluetooth specifically tailored toward low energy applications, as the name suggests. The first version of BLE (Bluetooth 4.0) saw the light of day in 2010, and since then, BLE has had an adoption rate unparalleled by most other technologies. The most influential factor is likely the cellphone market, helped along by the kin of Samsung and Apple. In addition to this, BLE allows for anyone who wants to use it to tweak it their purposes, while still being able to take advantage of the BLE framework. This is in contrast to Bluetooth Classic, which is only focused on a narrow set of pre-defined use cases.

Interestingly, in the pursuit of lower power consumption, BLE has scaled back on the punch it packs in terms of data throughput. While you should be comfortably able to listen to music over Bluetooth Classic, this is in reality close to impossible over Bluetooth Low Energy.

Even though the radio modulation rate of BLE is 1 Mbps, the protocol itself introduces a lot of overhead. In our case, the nRF51822 can transmit up to 6 data packets par *connection interval* - where a connection interval is simply one data exchange between two devices. Each data packet can contain up to 20 user defined bytes. The shortest possible connection interval is 7.5 ms, and so we are already at a theoretical upper limit of

$$\frac{6 \cdot 20 \cdot 8 \text{ bits}}{7.5 \text{ ms}} = 128 \text{ kbit/s}$$

Add to that the fact that Bluetooth Low Energy will always try to retransmit lost packages, collisions might happen, the radio might be busy, as so on and so forth. In practice, you can expect to see around 40-80 kbps. This is pretty unimpressive in term of raw data throughput, but Bluetooth Low Energy still shines in what it was intended to be good at, namely low power consumption in resource constrained environments.

From a technical point of view, the Bluetooth stack itself has changed over the years. So much so that a Bluetooth Classic cannot talk directly to Bluetooth Low Energy. The different layers of the Bluetooth Classic and Bluetooth Low Energy stacks are illustrated in figure 8. Bluetooth Classic is also often called "BR/EDR", which stands for "Basic Rate/Enhanced Data Rate".

In addition to the stack differences in Bluetooth Classic and Bluetooth Low Energy, the two standards also do radio operation differently. Bluetooth Classic segments the 2.4 GHz ISM band into 79 channels that it uses. Bluetooth Low Energy splits the same band into only 40 channels, as illus-

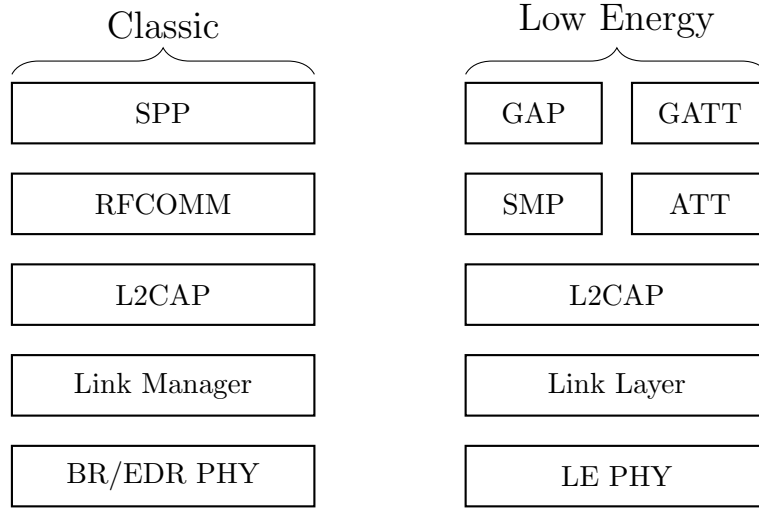


Figure 8: Bluetooth Classic stack and Bluetooth Low Energy stack.

trated in figure 9. Bluetooth Low Energy uses the channels 37, 38, and 39 exclusively for advertising, and the remaining channels for data transmission.

In essence, it is instructive to think of Bluetooth Classic and Bluetooth Low Energy as completely different technologies, merely sharing the same brand name. Even so, there exists *dual mode* Bluetooth devices, that can operate on both Bluetooth Classic, as well as Bluetooth Low Energy. This is the technology you will find in most phones.

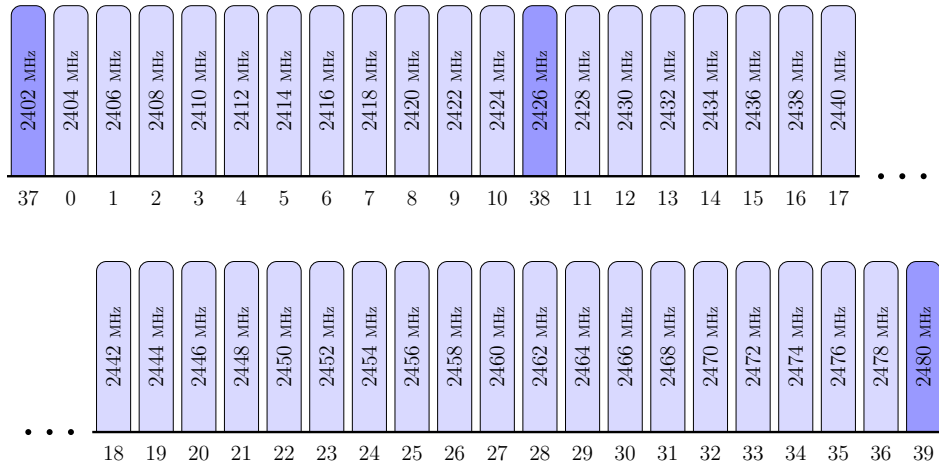


Figure 9: The Bluetooth channel spectrum.

When it comes to network topology, the Bluetooth Low Energy standard

is quite permissive in what it allows you to do. It is possible to simply broadcast to the surrounding world on some, or all, of the three advertising channels - this allows a device to send information to several other devices at the same time. However, it does have limitations; the information exchange is unidirectional only, so a broadcasting device cannot get anything in return.

For fancier things, where you require more than just the capability to spew out advertising packets, there are *connections*. In Bluetooth Low Energy, a connection is a periodic data exchange between *one* central, and *one* peripheral. It is not possible to send to two or more devices at the same time. Still, since Bluetooth 4.1, one physical device can don any combination of roles and connections it wants to. Thus, you can be a central and a peripheral at the same time. These *instances* of centrals or peripherals are then free to connect to other centrals or peripherals.

An important concept in Bluetooth Low Energy is the Generic Attribute Profile, or GATT for short. This is the workhorse behind virtually every BLE data transaction, and is what allows the standard to be so versatile. In essence, the GATT is simply a server-client type of service provided by Bluetooth, that allows for a standard way of organizing and accessing data.