INGCHIPS SDK User Guide

Ingchips Technology Co., Ltd.

Web: http://www.ingchips.com

E-mail: service@ingchips.com

Tel: 010-85160285

Address: Room 803, Building #3, Zijin Digital Park, Haidian District, *Beijing*

Room 316, Tower A, Juxin Building, Xiangke Road #58, *Shanghai*

Room 1009, Shuguang Building, Science Park, Nanshan District, *Shenzhen*

Contents

List of Figures

List of Tables

Welcome

Welcome to use *INGCHIPS* 918xx/9186xx Software Development Kit.

INGCHIPS 918xxx/9186xx are BLE 5.x full feature SoC solutions. This manual will give you an in-depth view on BLE development with 918xx/9186xx from software perspective.

Chapter 1

Introduction

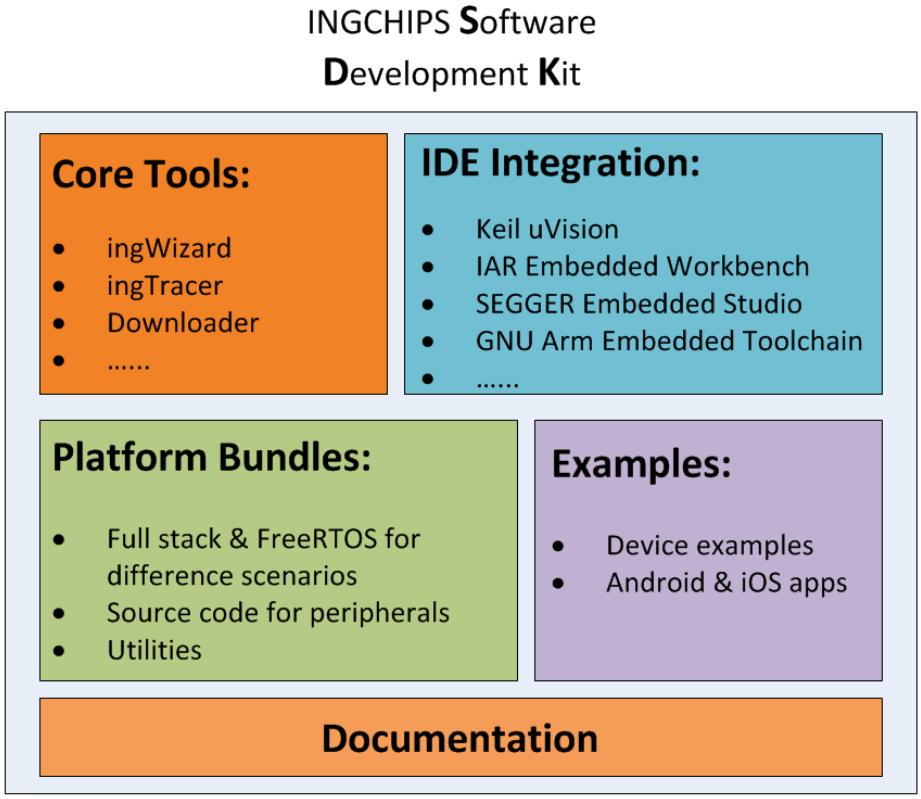

Figure 1.1: SDK Overview

INGCHIPS software development kit has following major components (see Figure 1.1):

1. Core Tools

   Provide project wizard, flash loader and other functionalities. These tools make BLE development easy and seamless.

2. Language & IDE Integration

Support Keil µVision[1], IAR Embedded Workbench[2], Rowley Crossworks for ARM[3], and SEGGER Embedded Studio for ARM[4]. All IDE/Toolchain settings are configured by core tools properly and automatically. GNU Arm Embedded Toolchain[5] is also supported.

Nim[6] is also supported as an alternative of C.

Optionally, new projects can be configured properly to use Visual Studio Code[7] as code editor. For Nim and GNU Arm Embedded Toolchain, building and downloading tasks are created and can be invoked in Visual Studio Code.

3. Platform Bundles

   Provide different bundles for different application scenarios (such as typical, and extension). Each bundle contains full stack & (optional) FreeRTOS binary, and C header files. Source codes for accessing peripherals are also provided.

4. Examples

   Provide a rich set of BLE device examples and corresponding Android and iOS referencing applications.

5. Documentation

   User guide (this document), API reference, and application notes are also provided.

1.1 Scope

This document covers platform overall architecture, core tools and platform APIs.

1.2 Architecture

There are two variants of bundles, one with built-in FreeRTOS (RTOS Bundles), and one without built-in RTOS (“NoOS” Bundles).

1.2.1 RTOS Bundles

ING918xx/ING9186xx software architecture is shown in Figure 1.2. Bootloader is stored in ROM and can’t be modified, while platform and app executable are stored in flash. Platform executable is provided for each bundle. BLE stack, FreeRTOS and some SoC functionality are compiled into this single platform executable. When system starts up, platform executable initializes, then loads the primary app executable.

[1] https://www.keil.com/
[2] https://www.iar.com/iar-embedded-workbench/
[3] https://www.crossworks.com/index.htm/
[4] https://www.segger.com/products/development-tools/embedded-studio/
[5] https://developer.arm.com/open-source/gnu-toolchain/gnu-rm
[6] https://www.nim-lang.org
[7] https://code.visualstudio.com

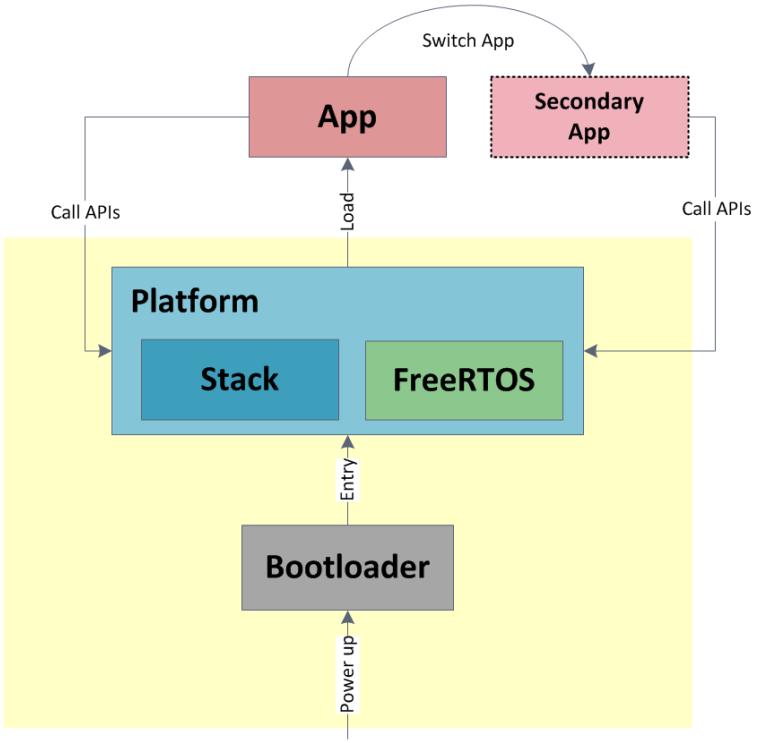

Figure 1.2: Architecture

A secondary app can only be asked to execute programmaticly. It is possible to download several secondary apps, and to switch between them programmaticly. After reset, platform will load the primary app executable as usual. Entry address of the primary app is managed by SDK tools, while entry addresses of secondary apps can be configured manually.

1.2.1.1 Apps built with c

App executable's main function is named `app_main`, where app gets initialized:

```c
int app_main(void)
{
    ...
    return 0;
}
```

`app_main` should always return `0`.

Platform, BLE stack and FreeRTOS APIs are all declared in corresponding `c` header files. To use these APIs, just include the necessary header files.

1.2.1.2 Apps built with `Nim`

App executable's main function is named `appMain`, where app gets initialized:

```
proc appMain*(): int {.exportc noconv.} =
  ...
  return 0;
```

`appMain` should always return `0`.

Thanks to the separation of platform and app, size of the app executable is significantly reduced. There are several benefits of this, not limited to:

- Smaller & cleaner app code base
- Faster downloading & FOTA
- Focus on function development

Flash tools are available for all supported IDEs and installed when installing SDK. App debugging is also quite easy: download platform binary with flash downloader (Section 3.2), then download & debug apps in IDE as usual.

1.2.2 "NoOS" Bundles

When developers want to use other RTOS, or use features that are missing in those RTOS bundles, developers can choose the "NoOS" bundles.

A generic RTOS interface is defined, and developers should provide an implementation of this interface to platform binaries through the returning value of `app_main`:

```
uintptr_t app_main(void)
{
    ...
    return (uintptr_t)os_impl_get_driver();
}
```

1.3 Abbreviations & Terminology

Table 1.1: Abbreviations

Abbreviation	Notes
ATT	Attribute Protocol
BLE	Bluetooth Low Energy
FOTA	Firmware Over-The-Air
IRQ	Interrupt Request
GAP	Generic Access Profile
GATT	Generic Attribute Profile
RAM	Random Access Memory
ROM	Read Only Memory
SDK	Software Development Kit

Table 1.2: Terminology

Terminology	Notes
Flash Memory	An electronic non-volatile computer storage medium
FreeRTOS	A real-time operating system kernel

1.4 References

1. Host API Reference
2. Bluetooth SIG[8]
3. FreeRTOS[9]
4. Mastering the FreeRTOS™ Real Time Kernel[10]

[8] https://www.bluetooth.com/
[9] https://freertos.org
[10] https://www.freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

Chapter 2

Tutorials

Following step-by-step tutorials show the basic usage of core tools and concepts of the SDK.

2.1 Hello World

In this tutorial, we are going to create a device which is advertising its name, “Hello, ”.

Start `ingWizard` from start menu and select menu item `Project -> New Project ...`. This brings up the project wizard. This first page shown by the wizard is `Development Tool` (see Figure 2.1).

2.1.1 `Development Tool` Page

On this page (Figure 2.1):

1. Choose IDE/Toolchain
2. Choose a project name
3. Choose where to store your project

`ingWizard` provides below handy functionality:

- If `Git` is used for software configuration management, select `Setup .gitignore`;
- If `Visual Studio Code` is the preferred code editor, select `Setup Visual Studio Code`.

Then press `Next` to proceed to the next page, `Choose Chip Series`.

2.1.2 `Choose Chip Series` Page

On this page (Figure 2.2), choose the target chip series of the project. Then press `Next` to proceed to the next page, `Choose Project Type`.

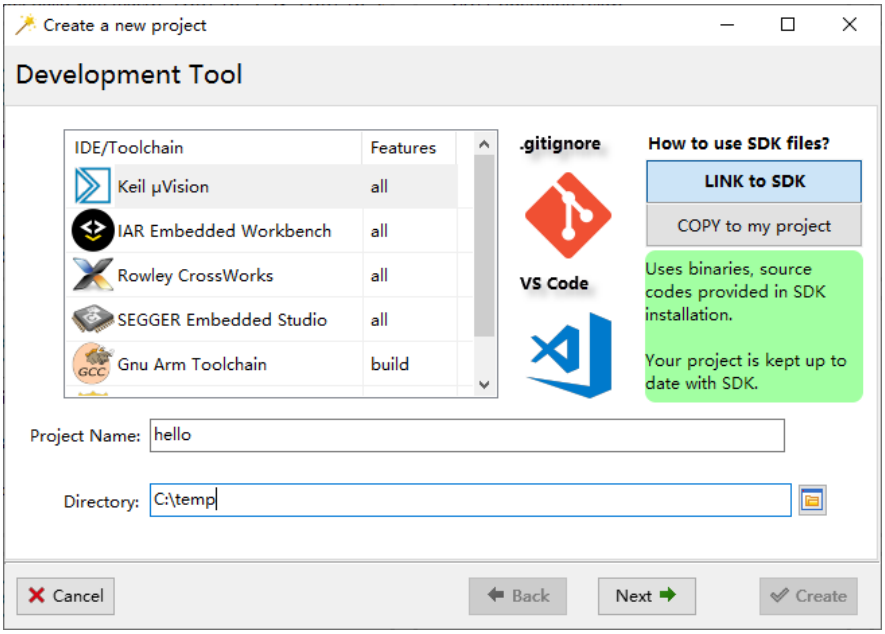

Figure 2.1: Choose Project Type

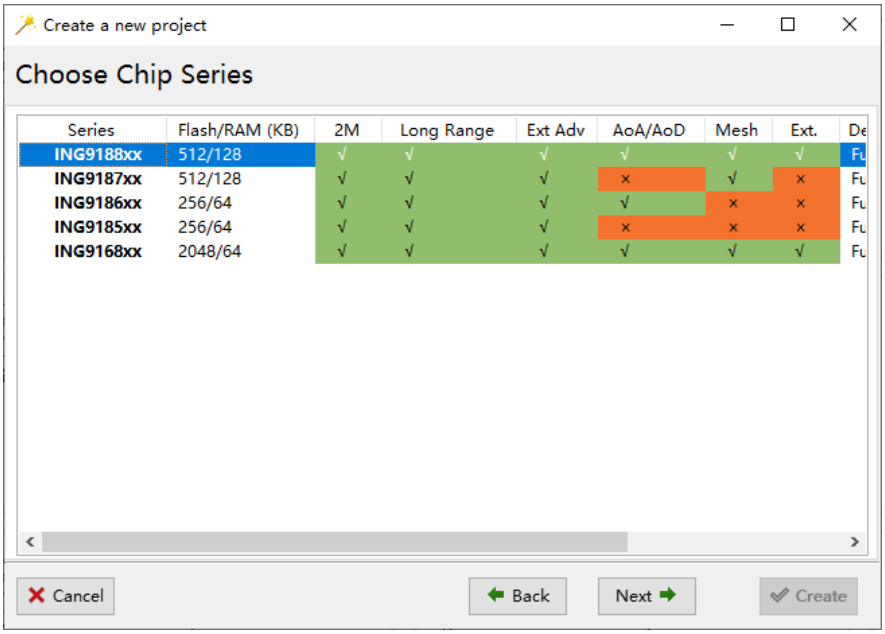

Figure 2.2: Choose Chip Series

2.1.3 `Choose Project Type` Page

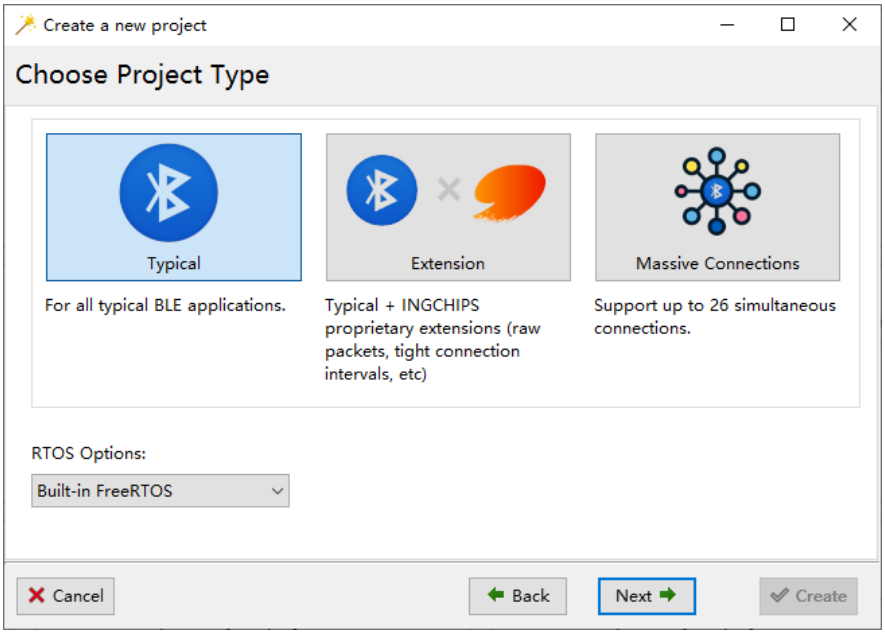

Figure 2.3: Choose Project Type

On this page (Figure 2.3), select `Typical`.

Then press `Next` to proceed to the next page, `Role of Your Device`.

2.1.4 `Role of Your Device` Page

On this page (Figure 2.4), just select Peripheral, and press `Next` to proceed to the next page, `Peripheral Setup`.

2.1.5 `Peripheral Setup` Page

On this page (Figure 2.5), select `Legacy` advertising.

Phones that support BLE 5.x extended advertising are still rare at present (`r Sys.Date()`) even if BLE 5.0 is declared as "supported", so we use legacy advertising for better compatibility. Furthermore, legacy advertising can be changed to BLE 5.x extended advertising by toggling a single bit later.

Click `Setup Advertising Data` button, which will bring up the advertising data editor (Figure 2.6). In the editor, type `name` to quickly search for the GAP advertising item `09 - «Complete Local Name»`, and click `Add` to add it into our device's advertising data.

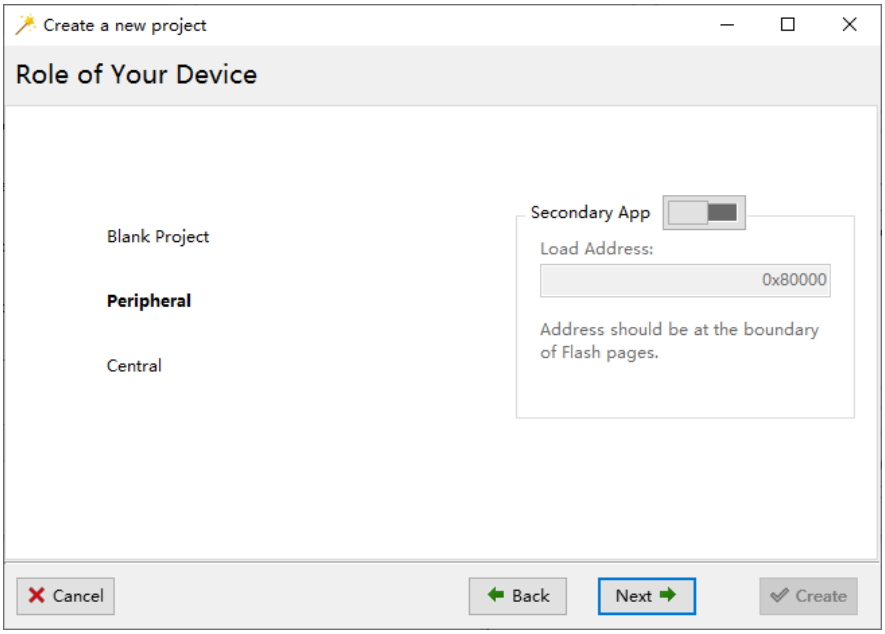

Figure 2.4: Role of Your Device

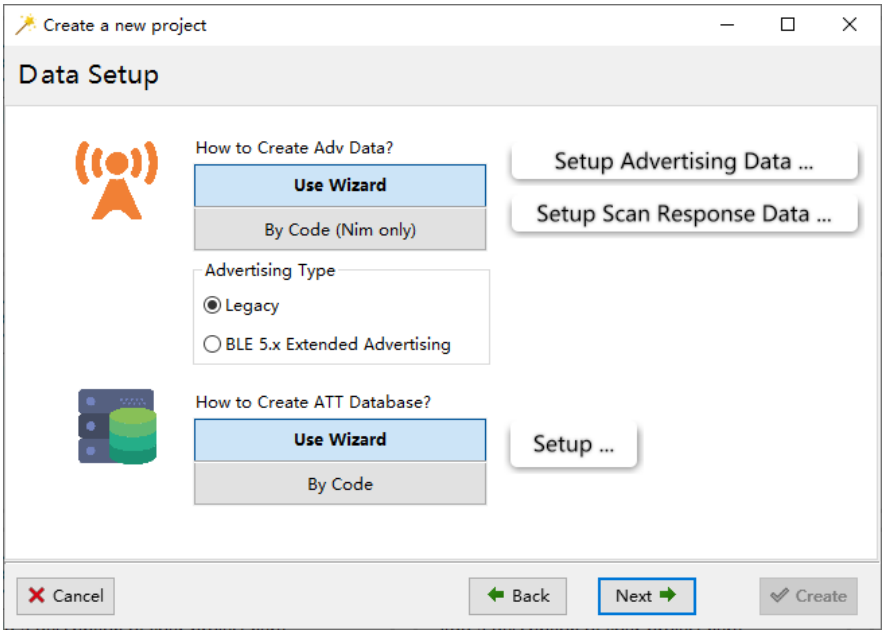

Figure 2.5: Peripheral Setup

Click the newly added `09 - «Complete Local Name»` item, then fill in "Hello, 世界" in the data editor shown below and press Enter. `Data Preview` will be updated and the whole advertising data is shown in raw bytes with a few comments on each item. Obviously, Chinese characters are encoded in `UTF-8` properly.

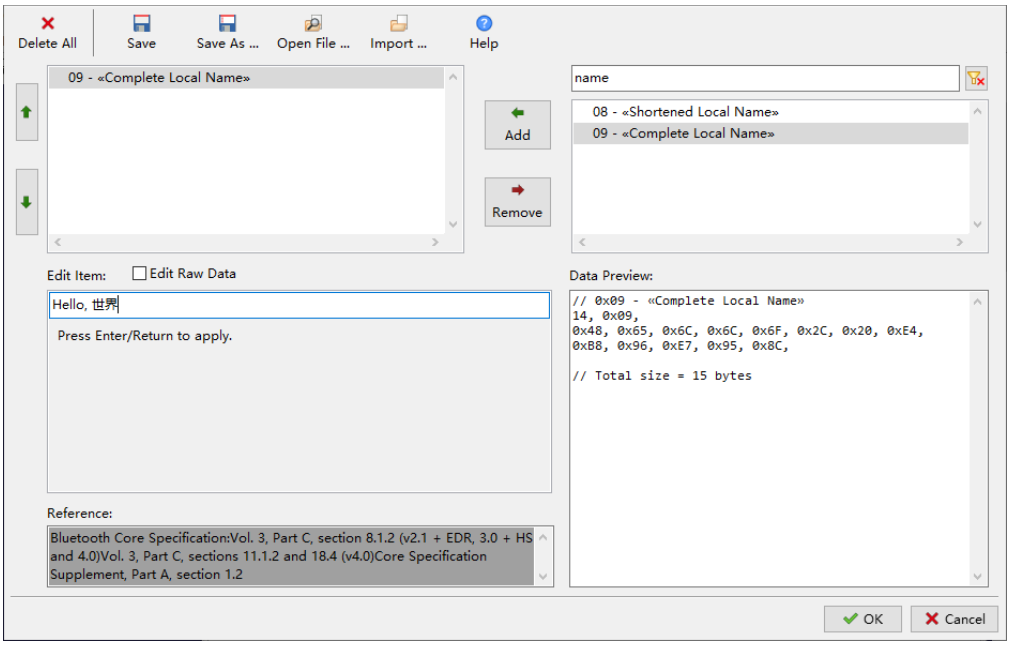

Figure 2.6: Edit Advertising Data

Now, click `OK` to go back to project wizard, and press `Next` to proceed to the next page `Security & Privacy`.

2.1.6 `Security & Privacy` Page

Leave all options as default (Figure 2.7), and press `Next` to proceed to the next page `Firmare Over-The-Air`.

2.1.7 `Firmare Over-The-Air` Page

Leave all options as default (Figure 2.8), and press `Next` to proceed to the last page `Common Functions`.

2.1.8 `Common Functions` Page

On this page (Figure 2.9), we also accept the default settings and press `Create`. Now your project is created (Figure 2.10), and ready for building and downloading.

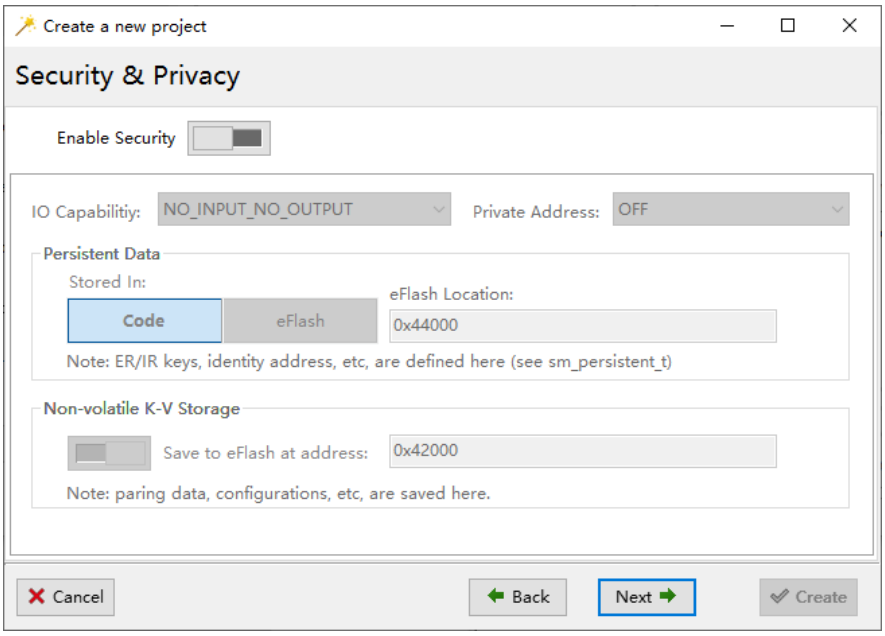

Figure 2.7: Firmare Over-The-Air

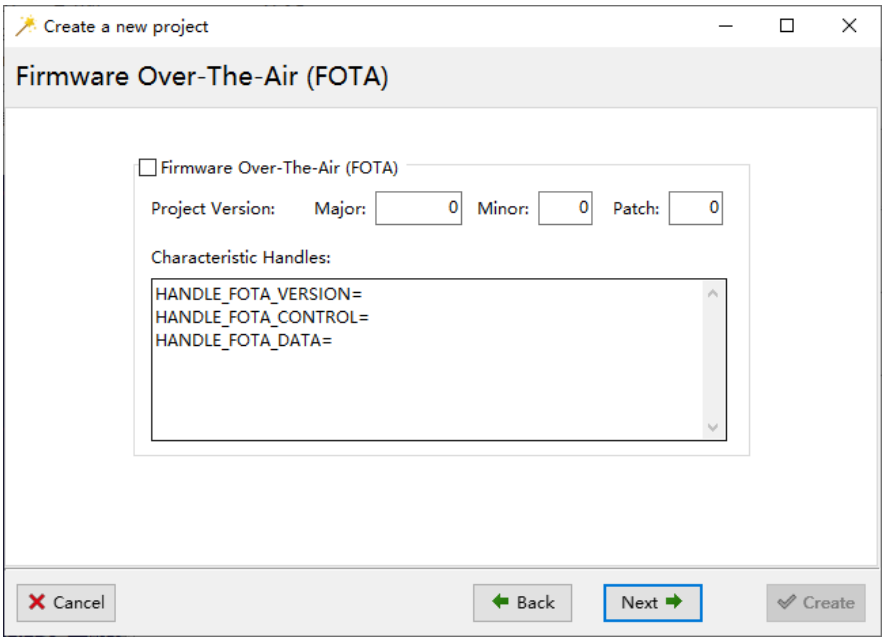

Figure 2.8: Firmare Over-The-Air

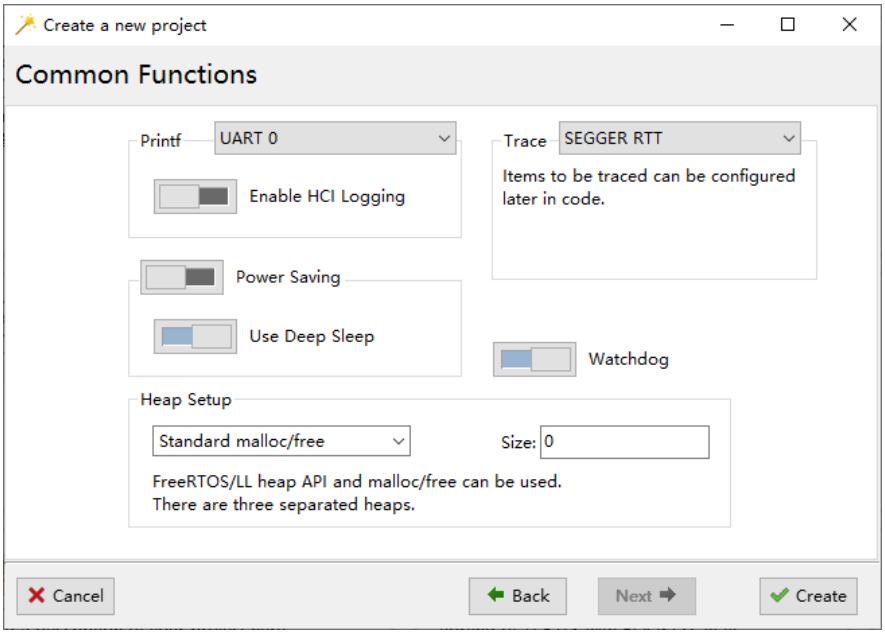

Figure 2.9: Common Functions

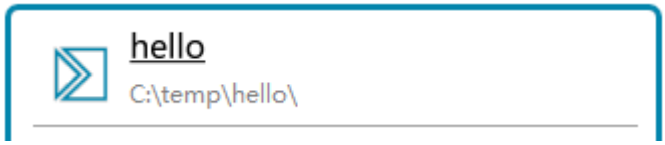

Figure 2.10: ”Hello, ” is Ready

2.1.9 Build your project

Back to the main window of `ingWizard` (Figure 2.10), click on your project to open it. Build your project in IDE.

2.1.10 Download

To download your project, back to `ingWizard` (Figure 2.10), right click on your project, and select `Download to Flash` from the popup menu to start the downloader (Figure 2.11).

All settings in the downloader are ready except the UART port number. In the downloader, configure the correct UART port and then click `Start`.

Once downloaded, check if you can find a device named “Hello, ” by LightBlue, `INGdemo` (Figure 2.12) or other apps. Note that, this device may not be listed in the Bluetooth menu of system settings at present.

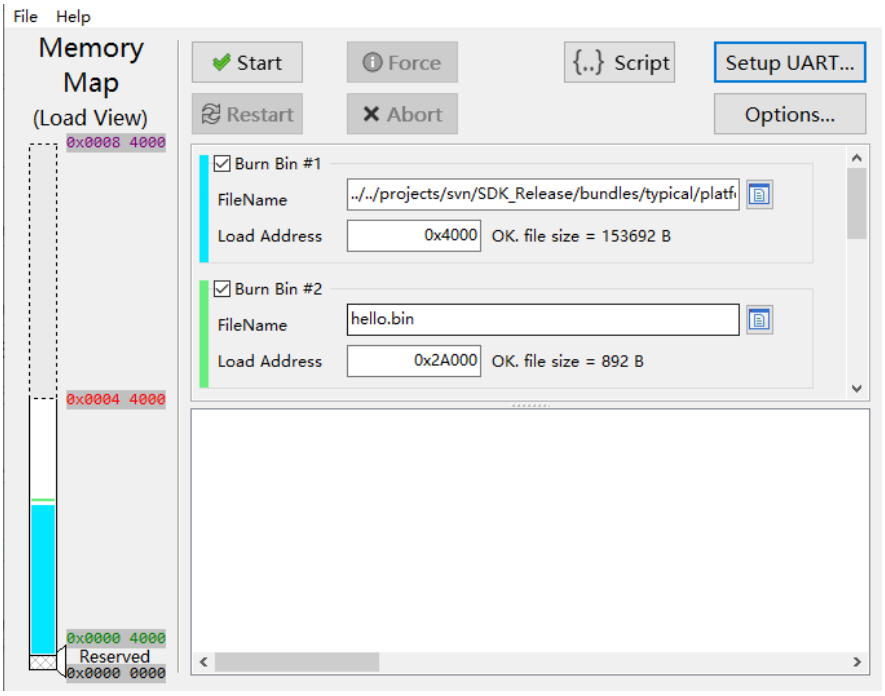

Figure 2.11: Download to Flash

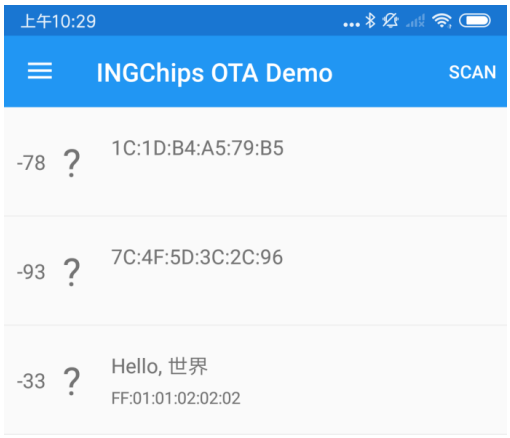

Figure 2.12: Hello,

2.2 iBeacon

In this tutorial, let's make an iBeacon. iBeacon is a protocol developed by Apple[1] and introduced at the Apple Worldwide Developers Conference in 2013. Beacons are a class of Bluetooth low energy (BLE) devices that broadcast their identifier to nearby portable electronic devices. This technology enables smartphones, tablets and other devices to perform actions when in close proximity to an iBeacon device.

Firstly, get a iBeacon scanning app from App Store. We will use an app called `Locate` in this tutorial. `Locate` has a list of preconfigured proximity UUIDs, which includes an all 0s Null UUID.

[1] https://developer.apple.com/ibeacon/

We will use this Null UUID[2].

2.2.1 Setup Advertising Data

There are two items in iBeacon advertising packet.

1. Flags

   Value is fixed to 0x06, i.e. two bits are set, LE General Discoverable Mode & BR/EDR Not Supported.

2. Manufacturer Specific Data

   The contents of this item is shown in Table 2.1

Table 2.1: iBeacon Manufacturer Specific Data

Size in Bytes	Name	Value	Notes
2	Company ID	0x004C	Company ID of Apple, Inc
2	Beacon Type	0x1502	Value defined by Apple
16	Proximity UUID		User defined value
2	Major		Group ID
2	Minor		ID within a group
1	Measured Power	in dBm	Measured by an iPhone 5s at a 1 meter distance

In order to make an iBeacon device, we can just follow the same steps as in the Hello World example, with only on exception that we need to configure the advertising package according to the specification.

In the advertising data editor, add `0x01 - «Flags»` and `0xFF - «Manufacturer Specific Data»`. Click `0x01 - «Flags»`, check `LE General Discoverable Mode` and `BR/EDR Not Supported`. Click `0xFF - «Manufacturer Specific Data»`, then the `Edit as` button, a menu pops up and select `iBeacon ...` (Figure 2.13) to open iBeacon manufacturer specific data editor (Figure 2.14).

Signal power can be set to any reasonable value (such as -50dBm), and we will calibrate it later with the help of the `Locate` app.

2.2.2 Try It

Let's select GNU Arm Embedded Toolchain as our development environment on `Choose Project Type` page, and the wizard will make everything ready (Figure 2.15).

Click on the project to open a console, type `make`[3] to build it. Back to `ingWizard`, follow the same steps to download it. Now we are able to find our newly created iBeacon in `Locate`. (Figure 2.16)

[2]Note that UUID is not allowed to be all 0s in final products.
[3]`Makefile` follows the syntax of GNU `make`.

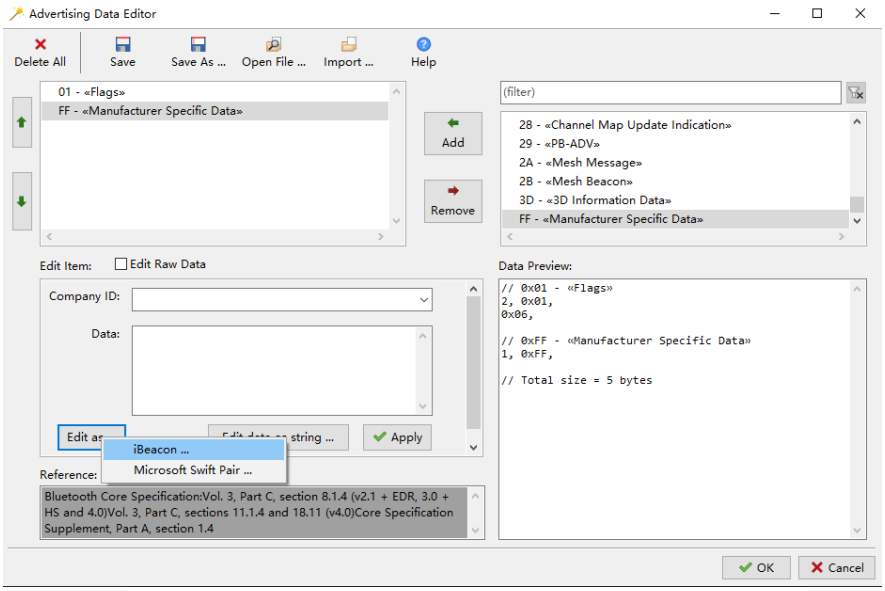

Figure 2.13: Edit iBeacon Advertising Data

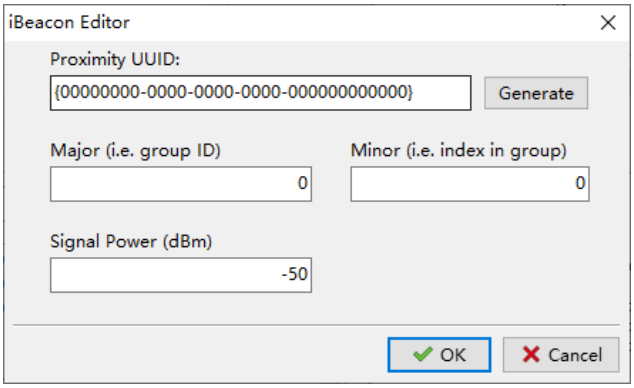

Figure 2.14: Edit iBeacon Manufacturer Specific Data

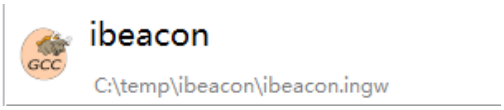

Figure 2.15: iBeacon Ready for GNU Arm Toolchain

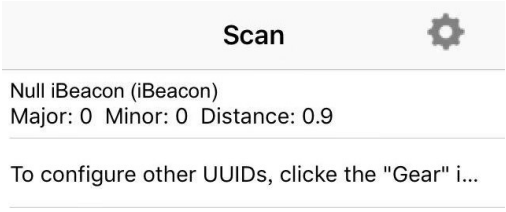

Figure 2.16: iBeacon in Locate app

Tap on our device then we can calibrate signal power or check distance in real-time as shown in Figure 2.17.

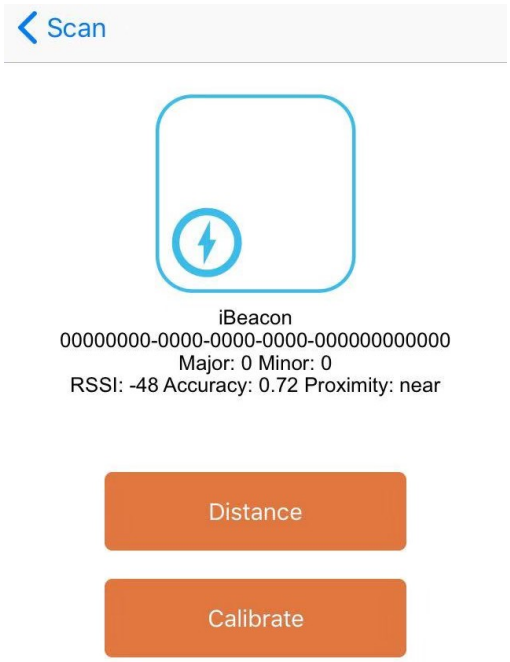

Figure 2.17: iBeacon Detailed Information in Locate app

Once signal power is calibrated, we can right click on our project in `ingWizard`, and select `Edit Data -> Advertising` menu item to edit its advertising data with same editor that we are getting familiar with. After advertising data is updated, rebuild the project and check if the distance is more accurate.

According to the specification, proximity beacons must use a non connectable undirected advertising PDU, using a fixed 100ms advertising interval. In this tutorial, we are not going to touch the code, so advertising parameters are not touched, either. To make these parameters fully meet the specification, please refer to the corresponding host GAP APIs.

2.3 Thermometer

In this tutorial, we are going to make a *serious* BLE device, a thermometer. Bluetooth SIG has already defined a GATT service called Health Thermometer[4]. This SDK contains a reference app called `INGdemo`, which can be deployed to an `Android` or `iOS` device. Using `INGdemo`, we can check Bluetooth devices' advertising data, and if health thermometer service is found in a device, `INGdemo` can connect to it and read temperature.

[4]https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.service.health_thermometer.xml

In this tutorial, you will learn how to:

- Broadcast supported services
- Configure a GATT profile
- Respond to the read request of a GATT characteristic

2.3.1 Setup Advertising Data

Again, we follow the same steps as in the Hello World example, and on the `Peripheral Setup` page, we declare the thermometer service and create a GATT profile. Add following three items into the advertising data:

1. `Flags`

   Value is fixed to 0x06, i.e. two bits are set, LE General Discoverable Mode & BR/EDR Not Supported.

2. `Complete List of 16-bit Service Class UUIDs`

   Add one service `0x1809 - Health Thermometer` as shown in Figure 2.18.

3. `Complete Local Name`

   Let's name our device as "AccurateOne".

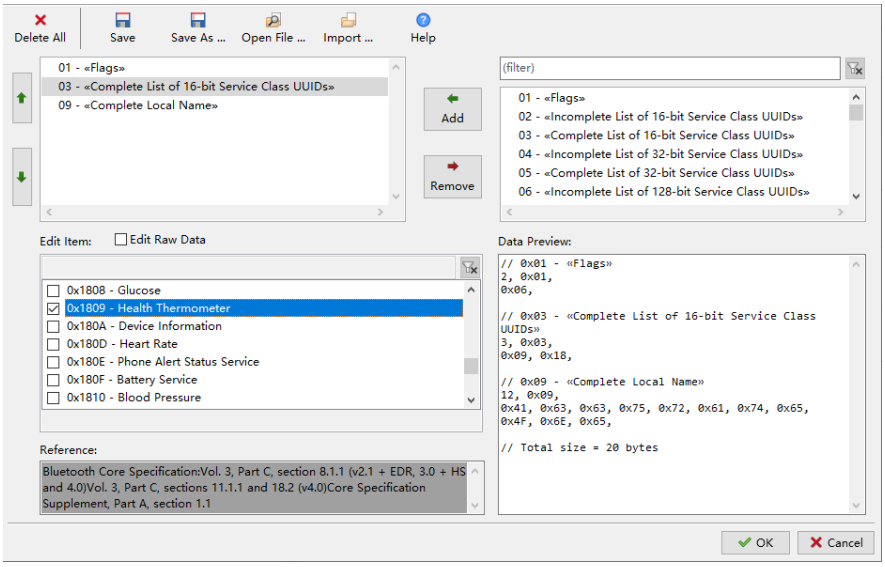

Figure 2.18: Thermometer Advertising Data

2.3.2 Setup GATT Profile

Back to the `Peripheral Setup` page and click `Setup ATT database ...` to open the GATT profile editor. Add two service, General Access (0x1800) and Health Thermometer (0x1809).
Delete all non-mandatory characteristics of General Access service. For Health Thermometer service, keep two characteristics, i.e. temperature measurement and temperature type, and delete the other two.

Next, edit each characteristic's value:

1. Device Name of General Access:

   Right click on the characteristic, select `Edit String Value ...` menu, and set the value to "AccurateOne".

2. Appearance of General Access:

   Right click on the characteristic, select `Help` and the editor will open the corresponding document on Bluetooth SIG website. Find the value for general thermometer (0x0300), then click the `Edit` button and input `0x00, 0x03` into the data field.

3. Temperature Measurement of Health Thermometer

   Check the document on Bluetooth SIG website. click the `Edit` button and input five 0s (`0, 0, 0, 0, 0`) into the data field. Here the first byte contains the flags showing that the following measurement is a `FLOAT` value in units of Celsius. Check `read` and `dynamic` properties (Figure 2.19).

   `FLOAT` type is IEEE-11073 32-bit float. Basically, it has a 24-bit mantissa, and an 8-bit exponent (the most significant byte) in base *10*.

4. Temperature Type of Health Thermometer

   Check the document on Bluetooth SIG website. Set it to any valid value by click the `Edit` button.

2.3.3 Write the Code

After project is created, open `profile.c` in IDE, and the temperature measurement characteristic handling function `att_read_callback` is automatically generated by `ingWizard`.

```
static uint16_t att_read_callback(hci_con_handle_t connection_handle,
                                  uint16_t att_handle, uint16_t offset,
                                  uint8_t * buffer, uint16_t buffer_size)
{
    switch (att_handle)
    {
```

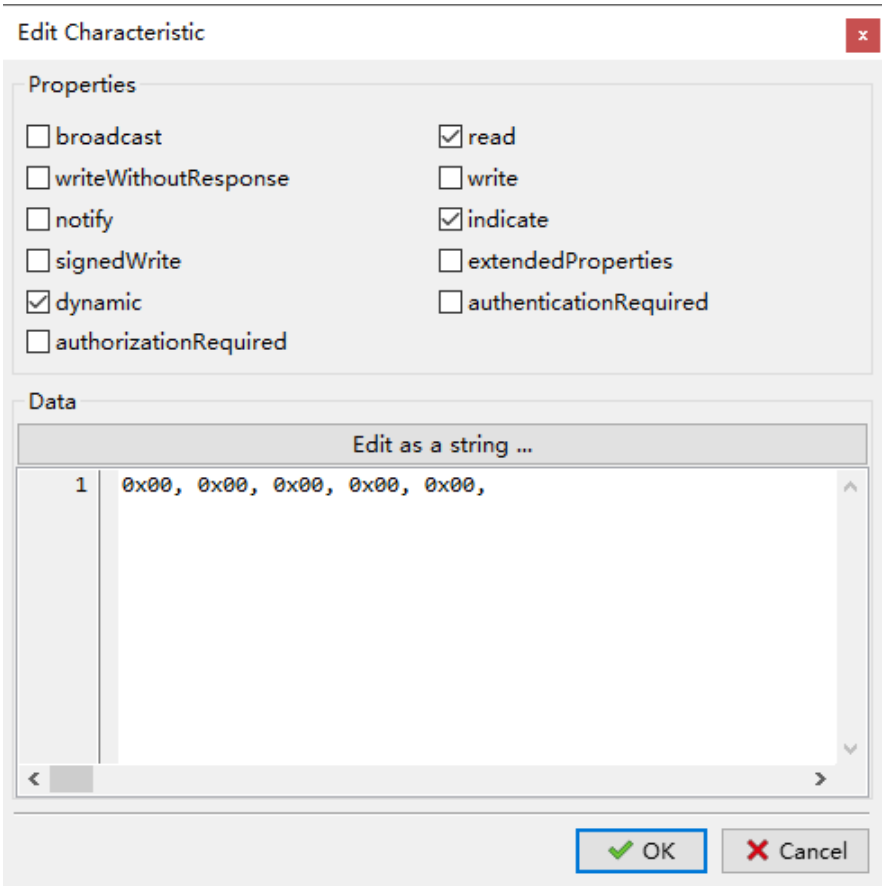

Figure 2.19: Edit Temperature Measurement

```
    case HANDLE_TEMPERATURE_MEASUREMENT:
        if (buffer)
        {
            // add your code
            return buffer_size;
        }
        else
            return 1; // TODO: return required buffer size

    default:

        return 0;
    }
}
```

`att_read_callback` will be called twice or more when app reads a characteristic that has `dynamic` property: one for querying required buffer size, and one for reading data. If data is large, `att_read_callback` might be called more times, each reading a part of data specified by `offset`.

As discussed above, define a temperature measurement type:

```
typedef __packed struct gatt_temperature_meas
{
     uint8 flags;
    sint32 mantissa:24;
    sint32 exponent:8;
} gatt_temperature_meas_t;

static gatt_temperature_meas_t temperature_meas = {0};
```

Now, we can complete the above `case HANDLE_TEMPERATURE_MEASUREMENT` clause:

```
    case HANDLE_TEMPERATURE_MEASUREMENT:
        if (buffer)
        {
            // simulate an "accurate" thermometer
            temperature_meas.mantissa = rand() % 100;
            // output data
            memcpy(buffer, ((uint8 *)&temperature_meas) + offset, buffer_size);
            return buffer_size;
        }
        else
            return sizeof(gatt_temperature_meas_t);
```

Build & download project, then connect to “AccurateOne” device in `INGdemo` app. Check if temperature changes randomly each time `Refresh` button is pressed (Figure 2.20).

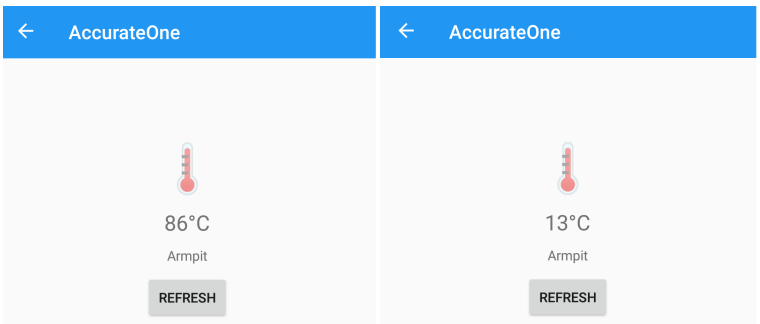

Figure 2.20: Refresh Temperature Measurement

A thermometer (a server) can use notification or indication procedure to notify (without acknowledge) or indicate (with acknowledge) a characteristic value, see [Thermometer with Notification]. In this example, “AccurateOne” does not use these two procedures, and sends its measurement passively.

2.3.4 Notification

2.4 Thermometer with FOTA

In this tutorial, we are going to add Firmware Over-The-Air update feature into our thermometer. This SDK provides a FOTA reference design that is workable out-of-the-box. To make FOTA work, at least three parties are involved, a device, an app, and an HTTP server. The `INGdemo` app is already there, so in this tutorial, we will focus on the device and HTTP server.

2.4.1 Device with FOTA

Follow the same steps as in the previous Thermometer example to create a new project, say “ota”.

When editing advertising data, we can import data created in previous example by clicking `Open File...` button of the editor. Advertising data is stored in `$(ProjectPath)/data/advertising.adv`. Let’s change device’s name to “Clickety Click”.

When editing GATT profile database, we can import data created in previous example by clicking `Open File...` button of the editor. GATT profile data is stored in `$(ProjectPath)/data/gatt.profile`. Select `INGChips Service` from the drop-down menu of `Add Service` button, and add “INGChips FOTA Service”. At present, we are not going to consider security issues, so delete the “FOTA Public Key” characteristics. Next, edit characteristics value of this service:

1. FOTA Version:

   This identifies the full version number of our project. As shown in flash downloader, a whole project is composed by two binaries, one is from SDK bundle, called platform binary, and the other one is built from our project, called the app binary. FOTA version contains two sub-versions, one for each binary. Each sub-version contains three fields:

   - Major: A 16-bit field.
   - Minor: A 8-bit field.
   - Patch: Another 8-bit field.

   Each bundle has its own version (so as the platform binary), using the same numbering scheme, which can be found on `SDK` page of `Environment Options` dialog (use menu item

`Tools` -> `Environment Options` to open this dialog). Suppose platform version is 1.0.1[5], and we would like our app's version to be 1.0.0, then we set this characteristic's value to (Fig 2.21):

```
0x0001, 0, 1  // platform version
0x0001, 0, 0, // app version
```

2. FOTA Control

   This is control point during update. Set its value to `0` (i.e. `OTA_STATUS_DISABLED`), which is the initial status of FOTA.

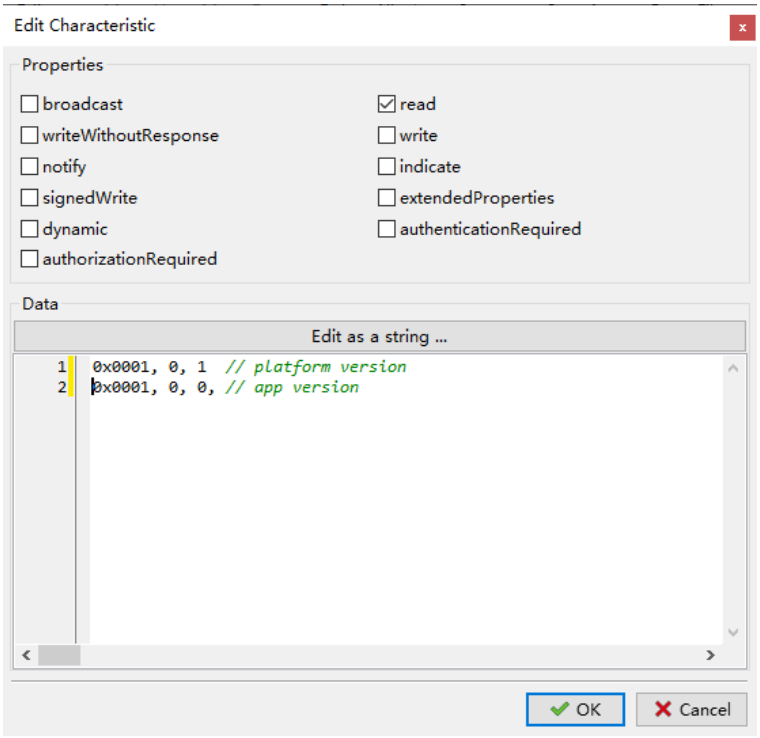

Figure 2.21: Configure FOTA Version

Click `OK` to close GATT profile editor. (Note: do not click `Save`, unless you want to change the file `$(ProjectPath)/data/gatt.profile` that is opened in editor.)

Back to project wizard, press `Next` to proceed to the next page `Firmare Over-The-Air`. On this page, let's check `FOTA`. Note that characteristics handles related to FOTA is generated automatically by inspecting the GATT profile. Then finish remaining steps on project wizard.

Open our brand-new project "ota", copy the code from previous example to make our thermometer respond to `Refresh` in `INGdemo` app.

Next, let's make a new version.

[5]Apps can report a different version in FOTA. It is not reqired to be same as in `Environment Options`.

2.4.2 Make a New Version

New version of our "ota" will have a new name "Barba Trick", and app version number is upgraded to `2.0.0`. These data are saved in advertising and profile data respectively, so right click on the project and use editors to update it. After data is updated, use `Save As ...` to save data to another file in the same directory, for example, update advertising data and save it to `$(ProjectPath)/data/advertising_2.adv`, and updated profile to `$(ProjectPath)/data/gatt_2.profile`.

Use macro `V2` to control the actual advertising and profile data:

```
const static uint8_t adv_data[] = {
#ifndef V2
    #include "../data/advertising.adv"
#else
    #include "../data/advertising_2.adv"
#endif
};

......

const static uint8_t profile_data[] = {
#ifndef V2
    #include "../data/gatt.profile"
#else
    #include "../data/gatt_2.profile"
#endif
};
```

Rebuild the project with macro `V2` defined, copy `ota.bin` and `platform.bin` (in `SDK_DIR/sdk/bundles/typical`) to an empty directory, say `ota_app_v2`.

Create a file named `manifest.json` in `ota_app_v2`, with follow data in it:

```
{
    "platform": {
                "version": [1,0,1],
                "name": "platform.bin",
                "address": 16384
    },
    "app": {
                "version": [2,0,0],
                "name": "ota.bin",
                "address": 163840
```

```
    },
    "entry": 16384,
    "bins":[]
}
```

Those addresses can be found in `Environment Options`. `entry` value is fixed to `0x4000`, i.e. `16384`. Note that `json` do not accept the popular `0xabcd` hexadecimal literals. `INGdemo` can download additional binaries specified by `bins` to device. In this case, we don't have such binaries, so this field is left as an empty array.

Then create a `readme` file for this update with some information about this update in it.

Now the FOTA package is ready. Make a `ota_app_v2.zip` ZIP archive of the whole `ota_app_v2` directory. Note that `ota_app_v2` should not be made into a sub-directory in `ota_app_v2.zip`. Table 2.2 summarize the files in the ZIP archive.

Table 2.2: FOTA Package Summary

File Name	Notes
readme	Some information about this update
manifest.json	Meta information
platform.bin	Platform binary
ota.bin	App binary

Back to IDE, rebuild the project leaving macro `V2` undefined, then download the project.

2.4.3 FOTA Server

`INGdemo` app needs a FOTA server URL, defined in `class Thermometer.FOTA_SERVER`. Move `ota_app_v2.zip` to HTTP server's document directory, and create a `latest.json` file, which contains information about latest version. Its content is:

```
{
    "app": [2,0,0],
    "platform": [1,0,1],
    "package": "ota_app_v2.zip"
}
```

Make sure that these two files can be accessed through URL `(FOTA_SERVER + latest.json)` and `(FOTA_SERVER + ota_app_v2.zip)`.

2.4.4 Try It

Connect to “Clickety Click” in `INGdemo`, click `Update` (Figure 2.22). Since `platform.bin` is up-to-date, only `app.bin` need to be updated, the whole update completes in a short time. Return to the main page, scan again and check if our new version works, a device named “Barba Trick” appearing. Connect to “Barba Trick”, firmware is up-to-date now.

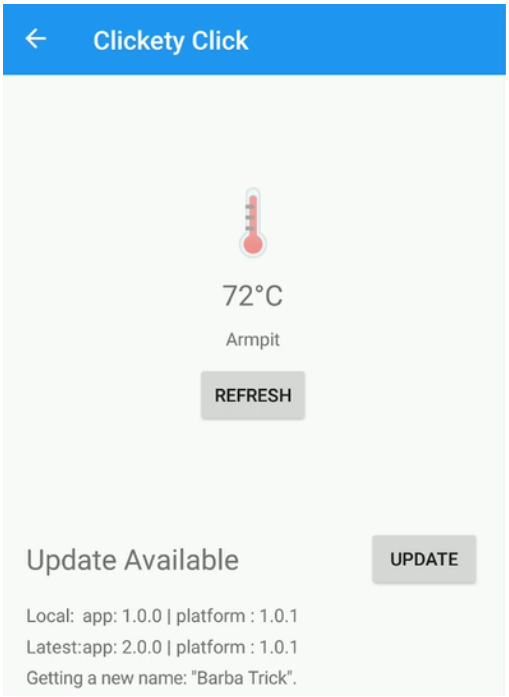

Figure 2.22: Update Available for ”Clickety Click”

This tutorial gives an example on FOTA implementation. Users are free to design a new FOTA solution, from version definition to FOTA service and characteristics. It also possible to develop a dedicated secondary app for FOTA.

Security must be considered.

2.5 iBeacon Scanner

We already know to how to make iBeacon devices. In this tutorial, we are going to create an iBeacon scanner.

A scanner plays a central role in Bluetooth pico network. As always, we create a new project named “iscanner” in `ingWizard` (Fig 2.23). On `Role of Your Device` page, select *Central*. A central device almost always scans for something then performs other actions, and our new project wizard automatically adds codes to start scanning.

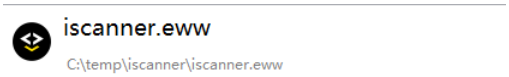

Figure 2.23: ”iscanner” Created for IAR Embedded Workbench

Open this new project in IDE, and navigate to function `user_packet_handler`. We can see there is an event called `HCI_SUBEVENT_LE_EXTENDED_ADVERTISING_REPORT`:

```
    case HCI_SUBEVENT_LE_EXTENDED_ADVERTISING_REPORT:
        {
            const le_ext_adv_report_t *report = decode_hci_le_meta_event(packet,
                      le_meta_event_ext_adv_report_t)->reports;
            // ...
        }
        break;
```

Each time this event is received, we can check if the advertising report contains `0xFF - «Manufacturer Specific Data»`, and if it is an iBeacon packet. With the knowlegde of making an iBeacon device, it is straight forward to define an iBeacon packet type in `C`.

```
typedef __packed struct ibeacon_adv
{
    uint16_t apple_id;
    uint16_t id;
    uint8_t  uuid[16];
    uint16_t major;
    uint16_t minor;
    int8_t   ref_power;
} ibeacon_adv_t;

#define APPLE_COMPANY_ID        0x004C
#define IBEACON_ID              0x1502
```

`__packed` is an extended keyword to specify a data alignment of 1 for a data type. Fortunately, it is supported by both *ARM* and *IAR* compilers. Alternatively, one can use `#pragma pack` directive:

```
#pragma pack (push, 1)
typedef struct ibeacon_adv
{
    ...
} ibeacon_adv_t;
#pragma pack (pop)
```

Before proceeding, let's create a helper function that converts an UUID to a string.

```
const char *format_uuid(char *buffer, uint8_t *uuid)
{
    sprintf(buffer, "{%02X%02X%02X%02X-%02X%02X-%02X%02X-"
                    "%02X%02X-%02X%02X%02X%02X%02X%02X}",
            uuid[0], uuid[1], uuid[2], uuid[3],
            uuid[4], uuid[5], uuid[6], uuid[7], uuid[8], uuid[9],
            uuid[10], uuid[11], uuid[12], uuid[13], uuid[14], uuid[15]);
    return buffer;
}
```

2.5.1 Distance Estimation

The received signal strength indication (RSSI) is reported together with advertising data. Generally, the intensity of electromagnetic waves radiating from a point source is inversely proportional to the square of the distance from the source. The well known equation for free space loss is:

$$Loss = 32.45 + 20log(d) + 20log(f)$$

Where d is in km, f in MHz and $Loss$ in dB. By comparing RSSI and measured power at a distance of 1 meter (`ref_power`), we can grossly estimate the distance between the scanner and beacon using the free space loss equation:

```
double estimate_distance(int8_t ref_power, int8_t rssi)
{
    return pow(10, (ref_power - rssi) / 20.0);
}
```

Now, we are able to make a fully functional iBeancon scanner in less than twenty lines:

```
    uint8_t length;
    ibeacon_adv_t *p_ibeacon;
    char str_buffer[80];
    const le_ext_adv_report_t *report;
    ......
    case HCI_SUBEVENT_LE_EXTENDED_ADVERTISING_REPORT:
        report = decode_hci_le_meta_event(packet,
                        le_meta_event_ext_adv_report_t)->reports;
        p_ibeacon = (ibeacon_adv_t *)ad_data_from_type(report->data_len,
```

```
                    (uint8_t *)report->data, 0xff, &length);

        if ((length != sizeof(ibeacon_adv_t))
            || (p_ibeacon->apple_id != APPLE_COMPANY_ID)
            || (p_ibeacon->id != IBEACON_ID))
            break;

        printf("%s %04X,%04X, %.1fm\n",
                format_uuid(str_buffer, p_ibeacon->uuid),
                p_ibeacon->major, p_ibeacon->minor,
                estimate_distance(p_ibeacon->ref_power, report->rssi));
        break;
```

Use the `Locate` app to transmit iBeacon signal, and check if our device can found it (Figure 2.24). Finaly, since RSSI value fluctuates, one can add a low pass filter on RSSI to make the estimation more stable.

```
[14:00:53.135] {2F234454-CF6D-4A0F-ADF2-F4911BA9FFA6} 0000,0000, 2.8m
[14:00:53.184] {2F234454-CF6D-4A0F-ADF2-F4911BA9FFA6} 0000,0000, 4.0m
[14:00:53.232] {2F234454-CF6D-4A0F-ADF2-F4911BA9FFA6} 0000,0000, 3.5m
[14:00:53.296] {2F234454-CF6D-4A0F-ADF2-F4911BA9FFA6} 0000,0000, 2.8m
```

Figure 2.24: iBeacon Scan Result

Note that the size of this app's binary increases dramatically. This is mainly because that Cortex-M3 don't have a hardware floating-point unit and floating-point operations are all performed by library functions. *Think twice* before using floating-point operations.

2.5.2 Concurrent Advertising & Scanning

As an exercise, we can merge iBeacon project with this one, and check if our device can send iBeacon signals while keeps scanning for other iBeacon devices.

Bluetooth radio uses TDD (Time Division Duplex) topology in which data transmission occur in one direction at one time and data reception occur at another time, and it's impossible to receive its own iBeacon signal.

2.6 Notification & Indication

A server can use notification or indication procedure to notify (without acknowledge) or indicate (with acknowledge) a characteristic value. Now, let's add notification and indication features to our thermometer we have created in a previous tutorial.

To notify or indicate a characteristic value, we use `att_server_notify` and `att_server_indicate` respectively. These two functions are only allowed to be called in `ATT_EVENT_CAN_SEND_NOW` event from `ATT` module, which is requested by `att_server_request_can_send_now_event`. These APIs must be called within the Bluetooth stack (`Host`) task.

Unsolicited notifications and indication may be triggered by a timer or interrupts, i.e. by sources outside of Bluetooth stack task. To call these Bluetooth stack APIs, a inter-task communication mechanism based on RTOS messages is provided.

2.6.1 Inter-task Communication

Use `btstack_push_user_msg` to send a message into Bluetooth stack stack:

```
uint32_t  btstack_push_user_msg(uint32_t msg_id, void *data, const uint16_t len);
```

This message will be passed to your `user_packet_handler` under event ID `BTSTACK_EVENT_USER_MSG`:

```
static void user_packet_handler(uint8_t packet_type, uint16_t channel,
                                uint8_t *packet, uint16_t size)
{
    uint8_t event = hci_event_packet_get_type(packet);
    btstack_user_msg_t *p_user_msg;
    if (packet_type != HCI_EVENT_PACKET) return;

    switch (event)
    {
    // ......
    case BTSTACK_EVENT_USER_MSG:
        p_user_msg = hci_event_packet_get_user_msg(packet);
        user_msg_handler(p_user_msg->msg_id, p_user_msg->data,
                         p_user_msg->len);
        break;
    // ......
    }
}
```

Here, we delegate the handling of the user message to another function `user_msg_handler`. Note that `user_msg_handler` is running in the context of Bluetooth stack task, and we are allowed to call those Bluetooth stack APIs now.

Event `BTSTACK_EVENT_USER_MSG` is broacasted to all `HCI` event callback functions.

2.6.2 Timer

Now let's make our thermometer "AccurateOne" to update its value once per second. Firstly, create a timer in initialization, such as in `app_main` or `setup_profile`.

```
TimerHandle_t app_timer = 0;

uint32_t setup_profile(void *data, void *user_data)
{
    app_timer = xTimerCreate("app",
                             pdMS_TO_TICKS(1000),
                             pdTRUE,
                             NULL,
                             app_timer_callback);
        // ...
}
```

Timer callback function is defined as:

```
#define USER_MSG_ID_REQUEST_SEND            1
static void app_timer_callback(TimerHandle_t xTimer)
{
    if (temperture_notify_enable | temperture_indicate_enable)
        btstack_push_user_msg(USER_MSG_ID_REQUEST_SEND, NULL, 0);
}
```

This timer is started when we get `HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE` in `HCI_EVENT_LE_META`, and stopped when we get `HCI_EVENT_DISCONNECTION_COMPLETE`.

Here `temperture_notify_enable` and `temperture_indicate_enable` are two flags initialized as 0s and set to `1` in `att_write_callback`:

```
static int att_write_callback(hci_con_handle_t connection_handle,
                              uint16_t att_handle, uint16_t transaction_mode,
                              uint16_t offset, uint8_t *buffer, uint16_t buffer_size)
{
    switch (att_handle)
    {
    case HANDLE_TEMPERATURE_MEASUREMENT + 1:
        handle_send = connection_handle;
        switch (*(uint16_t *)buffer)
        {
```

```
        case GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_INDICATION:
            temperture_indicate_enable = 1;
            break;
        case GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION:
            temperture_notify_enable = 1;
            break;
        }
        return 0;
    // ...
    }
}
```

Here we store `connection_handle` to a global variable `handle_send` which will be used later. The last piece of code is to handle message `USER_MSG_ID_REQUEST_SEND` in `user_msg_handler`:

```
static void user_msg_handler(uint32_t msg_id, void *data, uint16_t size)
{
    switch (msg_id)
    {
    case USER_MSG_ID_REQUEST_SEND:
        att_server_request_can_send_now_event(handle_send);
        break;
    }
}
```

And report temperature in `ATT_EVENT_CAN_SEND_NOW`:

```
...
case ATT_EVENT_CAN_SEND_NOW:
    temperature_meas.mantissa = rand() % 100;
    if (temperture_notify_enable)
    {
        att_server_notify(handle_send,
                          HANDLE_TEMPERATURE_MEASUREMENT,
                          (uint8_t*)&temperature_meas,
                          sizeof(temperature_meas));
    }

    if (temperture_indicate_enable)
    {
        att_server_indicate(handle_send,
                          HANDLE_TEMPERATURE_MEASUREMENT,
```

```
                        (uint8_t*)&temperature_meas,
                        sizeof(temperature_meas));
    }
    break;
...
```

Try to rebuild and download the project, and check if the temperature value shown in `INGdemo` changes once per second.

There is a fully functional thermometer example, a.k.a `thermo_ota`, supporting FOTA, notification and indication.

2.7 Throughput

BLE 5.0 introduces a new uncoded PHY with a sampling rate at 2M.

2.7.1 Theoretical Peak Throughput

Maximum payload length is 251 bytes for a Data Physical Channel PDU. Using 2M PHY, it takes 1048 µs to transmit. And an empty Data Physical Channel PDU takes 44 µs to transmit.

To achieve maximum throughput on one direction, length of all PDUs on this direction should be 251 bytes, while on the other direction, all PDUs should be empty. So, the transmission of 251 bytes takes a total duration of

$$1048 + 44 + 150 * 2 = 1392(\mu s)$$

Therefore, the theoretical peak throughput provided by link layer is

$$251 * 8/1392 * 1000000 \approx 1442.528(kbps)$$

For an app working above GATT, I2CAP and ATT all have their own overhead. Typically, GATT has a maximum effective payload of (251 - 7 =) 244 bytes. So, GATT could provide a theoretical peak throughput of

$$244 * 8/1392 * 1000000 \approx 1402.298(kbps)$$

2.7.2 Test Throughput

There are a pair of examples in SDK for throughput testing (Figure 2.25).

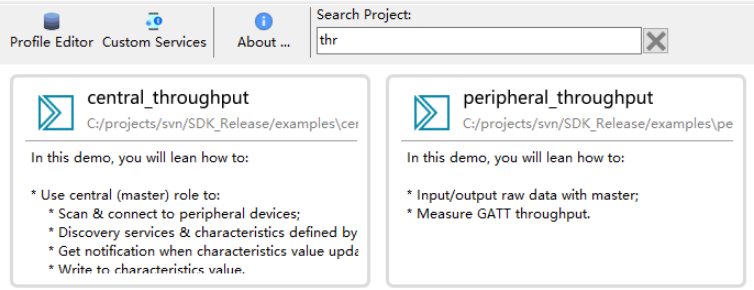

Figure 2.25: Examples for Throughput Testing

2.7.2.1 Test against `INGdemo`

Download `peripheral_throughput`. Use `INGdemo` to connect to `ING Tpt`, and open throughput testing page. On this page, we can test throughput from master to slave, from slave to master, or on both directions simultaneously.

Figure 2.26 shows that using a common low end Android phone with 2M PHY support, we can achieve a 1M+ bps throughput over the air.

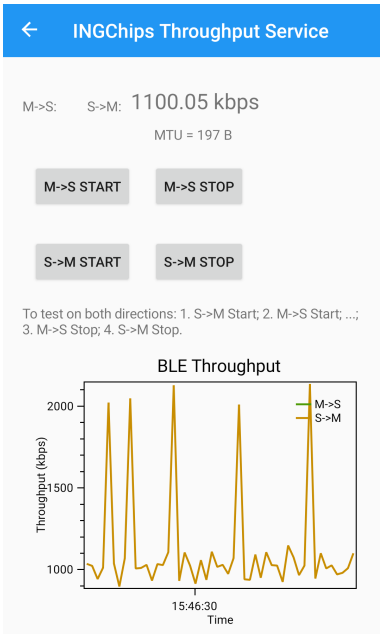

Figure 2.26: Througput on an Android Phone

2.7.2.2 Test against Our Own App

Example `central_throughput` demonstrates the typical procedure for a BLE central device:

1. Scan and connect to a device that has throughput service declared in its advertising
2. Discover throughput service;

3. Discover characteristics of the service;
4. Discover descriptors of characteristics.

`INGChips Throughput Service` has two characteristics.

- Generic Output

  By this characteristic, peripheral device send data to central device.

  This characteristic has a `Client Characteristic Configuration` descriptor.

- Generic Input

  By this characteristic, central device send data to peripheral device.

Download `central_throughput` to another board. This app has a UART command line interface to host computer. Connect to a host computer, type "?" to check supported commands. This app connects to `peripheral_throughput` automatically. Input command `start s->m` or `start m->s` to start testing throughput from peripheral to central, or from central to peripheral, receptively.

```
?
commands:
h/?          show this
start dir    start throughput test on dir
stop  dir    stop throughput test on dir

note: dir = s->m, or m->s
start s->m
```

Figure 2.27: Command interface

Figure 2.28 shows that using two boards, we have achieved a stable throughout at 1.2M+ bps over the air.

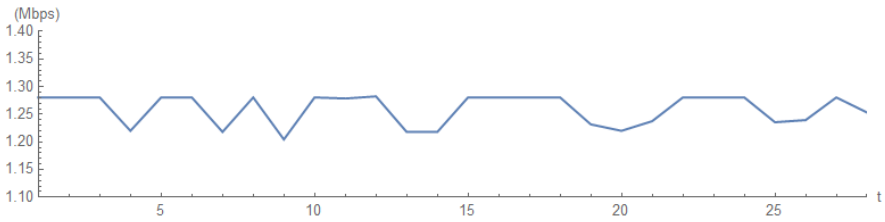

Figure 2.28: Througput Between Boards

This throughput is tested over the air, a little bit lower than theoretical peak value, but much more practical.

2.8 Dual Role & BLE Gateway

In this tutorial, we are going to create a BLE gateway, which collects data from several peripheral devices and reports data to a central device. When collecting data, this gateway is a central device, while reporting data, it is a peripheral device, i.e., our app has two roles.

More specifically, our gateway only supports to collect data from thermometers. Let call it a `smart_meter`.

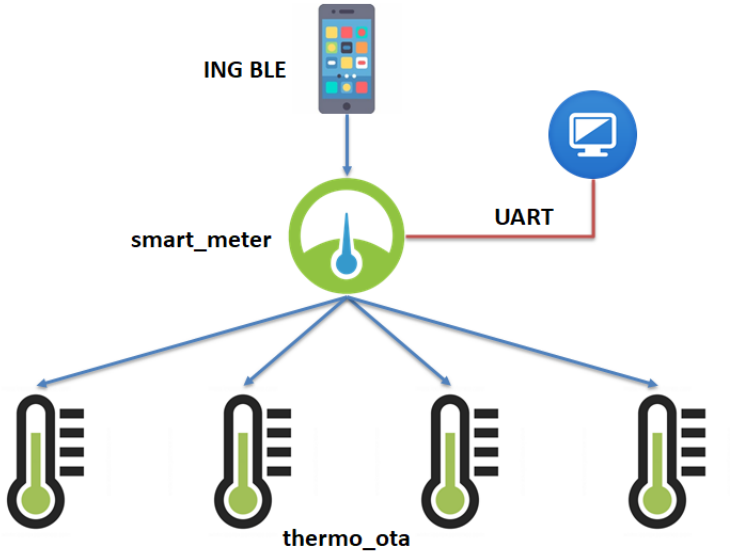

Figure 2.29: Smart Meter Overview

`smart_meter` uses a generic string based output service for report data to a central device, such as the `INGdemo` running on a smart phone. It also has a UART control interface connecting to a host computer.

Checkout the example `peripheral_console` for how to do string input & output.

Full functional `smart_meter` app is also provided as an example. Take this example as an reference while creating your own.

Now, let's create this BLE gateway.

2.8.1 Use `ingWizard` to create a peripheral app

Use GUI editor to edit advertising data, naming our app as "ING Smart Meter".

Use GUI editor to edit GATT Profile. Add `INGChips Console Service` into GATT Profile (Figure 2.30).

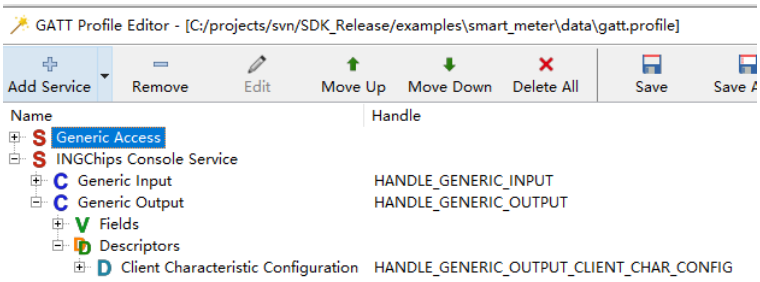

Figure 2.30: Smart Meter GATT Profile

2.8.2 Define Thermometer Data

A thermometer is identified by its device address and `id`. Each thermometer uses its own connection identified by `conn_handle`.

```
typedef struct slave_info
{
    uint8_t     id;
    bd_addr_t   addr;
    uint16_t    conn_handle;
    gatt_client_service_t                   service_thermo;
    gatt_client_characteristic_t            temp_char;
    gatt_client_characteristic_descriptor_t temp_desc;
    gatt_client_notification_t              temp_notify;
} slave_info_t;
```

Define four thermometers.

2.8.3 Scan for Thermometers

Call two GAP APIs to start scanning. Once a device is found, check whether its device address is one of the thermometers. If so, stop scanning and call `gap_ext_create_connection` to connect.

After connection established, if there is any thermometer not connected, then start scanning again.

2.8.4 Discover Services

After connection established, call `gatt_client` APIs to discover its services.

These APIs follow a similar logic like Android, iOS.

2.8.5 Data Handling

Subscribe to thermometer's `Temperature Measurement` characteristic. When a new measurement is received, convert the value into a string and report it to a host computer. If our app is already connected to a central device, forward this information to it through GATT characteristic.

2.8.6 Robustness

To make our app more *robust*:

- If disconnected from a thermometer, then start scanning;
- If disconnected from a central device, then start advertising.

2.8.7 Prepare Thermometers

We can use example `thermo_ota` as thermometers. But we need to configure different address for each one.

We can write a simple script for downloader to generate these addresses automatically:

```
procedure OnStartBin(const BatchCounter, BinIndex: Integer;
                     var Data: TBytes; var Abort: Boolean);
begin
  if BinIndex <> 6 then Exit;
  Data[0] := BatchCounter;
end;
```

For further information on downloader scripting, see Scripting & Mass Production.

2.8.8 Test

Input command `start` on host computer to start our app (start scanning & advertising). Use `INGdemo` to connect to a device named "ING Smart Meter" and check temperature measurements.

Turn off and on one or more thermometers, and our app should be able to reconnect to them.

2.9 Hello, Nim

To use `Nim` to develop apps, `nim` and `Gnu Toolchain` are both required. `Nim` compiler translates `Nim` source code into `C` source code, then `Gnu Toolchain` is invoked to compile and link the translated `C` source code together with SDK, as shown in Figure 2.31.

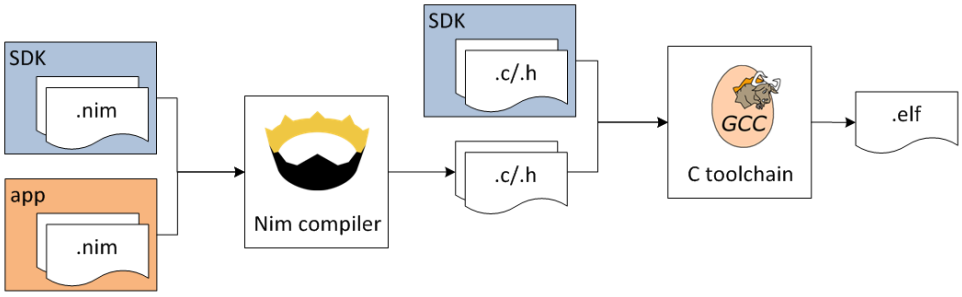

Figure 2.31: Build a Nim App

`Visual Studio Code` is recommended for `Nim` code editing and building. Let's make a simple app using `Nim`.

2.9.1 Create a `Nim` Project

On the `Develpment Tool` page, Select `Nim + Gnu Toolchain`. Select `By Code` for both advertising & ATT database generation (Figure 2.32).

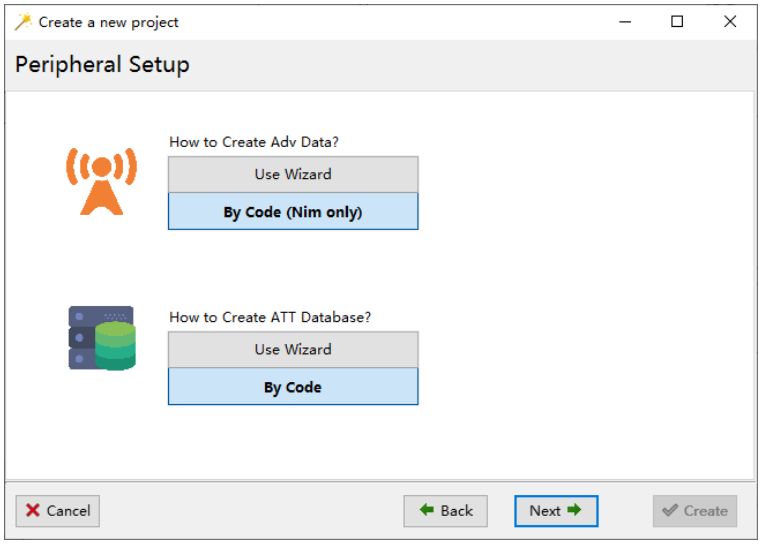

Figure 2.32: Use Code to Generate Data

`ingWizard` also support create these data for `Nim` apps. In this tutorial, we are going to show that it is easy to create these data with meta-programming[6] in `Nim`.

2.9.2 Create Advertising Data

Using `Nim` module `btdatabuilder`, we can create advertising and GATT profile easily.

[6]https://en.wikipedia.org/wiki/Metaprogramming

- Example 1: Create a device named “Hello, Nim”

```
let
  advData = ToArray([Flags({LEGeneralDiscoverableMode, BR_EDR_NotSupported}),
                     LocalName("Hello, Nim")])
```

- Example 2: Create an iBeacon

```
let
  advData = ToArray([Flags({LEGeneralDiscoverableMode, BR_EDR_NotSupported}),
                     iBeacon("{E9052F1E-9D67-4A6E-B2D7-459D132D6A94}",
                             0, 0, -50)])
```

2.9.3 Create Profile Data

```
defineProfile([Service(SIG_UUID_SERVICE_GENERIC_ACCESS),
               Characteristic(SIG_UUID_CHARACT_GAP_DEVICE_NAME, ATT_PROPERTY_READ,
                              "Hello, Nim"),
               Characteristic(SIG_UUID_CHARACT_GAP_APPEARANCE, ATT_PROPERTY_READ,
                              [0u8, 0]),
               Service(SIG_UUID_SERVICE_BATTERY_SERVICE),
               Characteristic(SIG_UUID_CHARACT_BATTERY_LEVEL, ATT_PROPERTY_READ,
                              [20u8], "HANDLE_BATTERY_LEVEL")],
               "profileData")
```

Once above code got compiled, ATT database is stored in `profileData`, handle of battery level characteristic is identified by a const `HANDLE_BATTERY_LEVEL`, and the offset (in byte) of battery level value in ATT base (i.e. `profileData`) is identified by a const `HANDLE_BATTERY_LEVEL_OFFSET`.

We can use these variables and constants generated by macro `defineProfile` just as *normal* ones. For example, let’s create a task that updates battery level pseudo randomly:

```
proc updateBatteryLevel(unused: pointer) {.noconv.} =
  while true:
    vTaskDelay(pdMS_TO_TICKS(1000))
    profileData[HANDLE_BATTERY_LEVEL_OFFSET] = rand_level()
...
discard xTaskCreate(updateBatteryLevel, "b",
                    configMINIMAL_STACK_SIZE, nil,
                    configMAX_PRIORITIES - 1, nil)
```

There are at least three ways to generate pseudo random number in Nim, use PRNG provided by C's stdlib, use PRNG provided by provided by Nim, or create our own PRNG.

- Use C's PRNG

```
# It's easy to import C functions and use them
proc rand(): cint {. importc: "rand", header: "stdlib.h".}

proc rand_level(): uint8 = cast[uint8](rand() mod 101)
```

- Use Nim's PRNG

```
import random
proc rand_level(): uint8 = cast[uint8](rand(0..100))
```

- Create a simple PRNG

```
proc rand_level(): uint8=
  var last {.global.} = 0u16
  last = (last * 173 + 31) and 0x7fffu16
  return cast[uint8](last mod 101)
```

As we see, all three ways are easy in Nim.

platform_hrng can be used to initialized PRNG.

2.9.4 Benefits of Adopting Nim

Nim is as powerful as C because SDK provides bindings of all C APIs for Nim. There are many benefits of adopting Nim, such as it supports meta-programming and it is strongly typed.

- *Meta-programming*

  With metaprogramming, we can create advertising and ATT database at compile time, which has 0 overhead in runtime obviously.

- *Strongly Typed*

  Nim is more strongly typed than C, which can help to make code safer.

Chapter 3

Core Tools

SDK core tools play an important role in the BLE device development.

3.1 ingWizard

`ingWizard` is the recommended entry point in the whole development life cycle. With it, we can create & open project, edit project data, and migrate projects, etc.

1. Create Project

   `ingWizard`'s new project wizard assists the creation of new projects. We can select favourate IDE, peripheral role, edit advertising and profile data, enable FOTA and logging, etc.

   Once a project is created, following files are also created, used by `ingWizard` but not IDE, and they shoule not be deleted, or `ingWizard` will not to function properly:

   - `$(ProjectName).ingw`
     This file shares the same name with the project with an extension `.ingw`. It contains crucial information about the project and SDK. Without this information, it becomes impossible to do migration.

2. Advertising Data Editor

   This editor helps us to generate advertising data. It can also be opened from main menu `Tools -> Advertising Data Editor ...`.

3. GATT Profile Editor (or GATT/ATT Database Editor)

   This editor helps us to build GATT profile data. It can also be opened from main menu `Tools -> Profile Database Editor ...`.

   This editor supports three type of services, SIG defined services, *INGChips* defined services and user defined services. To add an user defined service, it must be create beforehandle (see below).

4. Manage Custom Services

   This editor can be opened from main menu `Tools -> Manage Custom GATT Services ...`. We can add, delete and edit custom services.

   Custom Services and characteristics are all named with a prefix which is deduced from company name initialized when installing SDK, and updatable through `Environment Options`.

5. Migration

   In case a new version of SDK is installed, ROM and RAM used by platform might be changed, so projects settings need to be updated accordingly. This process is automated by right click on a project and select `Check & Fix Settings ...`.

Alway remember to backup your project before perform a migration, either by committing all changes into version control system or making a full backup.

3.2 Downloader

3.2.1 Introduction

This downloader downloads up-to six images (binaries) to flash through UART connection. It co-operate with bootloader. Bootloader can be made into flash downloading mode either by:

- Asserting boot pin (this is used in the vast majority of cases),
- Setting entry point which is stored Flash to an invalid address.

When 918xx is powered on, bootloader checks above conditions. If any conditions are true, bootloader sends the handshaking message.

User can download any files, although typically these files are generated by IDE tools. The load address of image (binary) must be aligned at flash page boundary. Each flash page has `8192` (`0x2000`) bytes. Flash starts from `0x4000`, so the load address should be `0x4000 + X * 0x2000`, where `X` is an integer. Downloader complains if the load address is not correct. Note that when this downloader is started from `ingWizard`, binaries have already been correctly configured.

Click `Setup UART ...` to configure UART (Figure 3.1). Users need to set `Port Number` to the value shown in Windows Device Manager, for example, if “COM9” is used, then set `Port Number` to `COM9`, or simply `9`. Baud rate can be set to a value larger than `115200`, such as `256000`, `512000`, `750000`, `921600`, etc, to achieve a faster download speed. The maximum supported baud rate is `921600`. Due to the limitation of internal flash characters, there isn’t any further significant improvement for baud rate larger than `512000`. Other fields should be left unchanged.

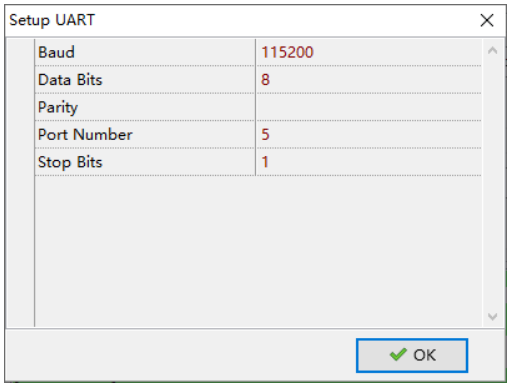

Figure 3.1: Configurate UART

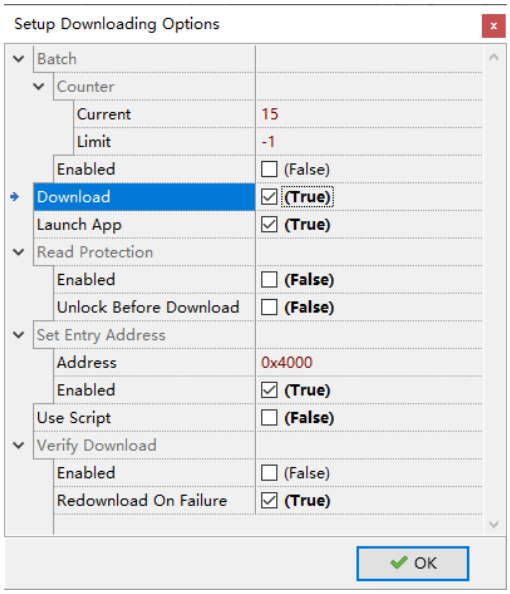

Figure 3.2: Downloader Options

The whole downloading procedure is composed of several steps, such as downloading, verification, set entry address, and launching app. These steps can be configured by clicking `Options` (Figure 3.2).

Entry address `0x4000` equals to the load address of platform binary. If "Verify Download" is enabled, then data will be read back and compare with origin file to ensure data is correctly downloaded. Data blocks are CRC checked, so "Verify Download" can be kept disabled on a regular basis. If downloading keeps failing on specific address, then we can enable it to double check if flash is malfunctional. In this case, when mismatch is found, read-back data will be stored to a file.

When "Batch" mode is enabled, downloader will keep waiting for bootloader handshaking, and once received handshaking, downloading starts; after downloading completes, downloader will start waiting again. When "Batch" mode is disabled, downloader will no longer wait for handshaking after downloading completes.

Click `Start` to start downloading, or rather start waiting for handshaking. Bootloader sends handshaking message only once, and if chips are already powered up, it may be too late to receive

handshaking. In this case, we can click `Force` to skip handshaking and start downloading immediately.

3.2.2 Scripting & Mass Production

This downloader supports powerful scripting, making it suitable for mass production. In the script, two event handlers (functions) are required to be defined.

- `OnStartRun`

  This event handler gets called when each round of downloading starts;

- `OnStartBin`

  This event handler gets called when a binary starts downloading. Here, binary data can be modified on-the-fly before it is written into flash.

When "Batch" mode is enabled, this downloader keeps a counter which is increased by `1` after downloading finishes. This counter is shown as `Counter.Current` shown in Figure 3.2. There is also a variable called `Counter.Limit`. In "Batch" mode, before a new round of downloading starts, `Counter.Current` is checked against this limit, if it is *larger* than limit, "Batch" mode stops automatically. For example, if `Counter.Current` and `Counter.Limit` are set to `10` and `13` respectively, then "Batch" mode will run for `4` rounds in total, with `Counter.Current` equals to `10`, `11`, `12` and `13`. After "Batch" mode stops, `Counter.Current` equals to `14`.

The language used for scripting is *RemObjects Pascal Script*[1], which is quite similar to `C`, and easy to develop. Below is an simple but working example, in which, the batch round number (`BatchCounter`) is written to a fixed location in the binary.

```
// we can use constants
const
  BD_ADDR_ADDR = $1;

// BatchCounter is just Counter.Current
procedure OnStartRun(const BatchCounter: Integer; var Abort: Boolean);
begin
  // Use *Print* for logging and debugging
  Print('OnStartRun %d', [BatchCounter]);
  // we can abort downloading by assigning True to *Abort*
  // Abort := True;
end;

procedure OnStartBin(const BatchCounter, BinIndex: Integer;
                     var Data: TBytes; var Abort: Boolean);
```

[1] https://github.com/remobjects/pascalscript

```
begin
  // Note that BinIndex counts from 1 (not 0), just as shown on GUI
  if BinIndex <> 2 then Exit;
  // We can modify binary data before it is downloaded into flash
  Data[BD_ADDR_ADDR + 0] := BatchCounter and $FF;
  Data[BD_ADDR_ADDR + 1] := (BatchCounter shr 8) and $FF;
  Data[BD_ADDR_ADDR + 2] := (BatchCounter shr 8) and $FF;
end;
```

3.2.3 Flash Read Protection

To protect illegal access of data & program stored in flash, 918xx has a read-protection mechanism. Once read-protection is enabled, JTAG/SW and this downloader can not be able to access flash any more. To re-enable JTAG/SW debugging functionality and downloading, the read-protection must be turned off by a procedure called *unlock*. Flash data is erased in this procedure.

Once the app is ready to ship, and it is decided that data & program must be protected from illegal access, just enable "Read Protection" as shown in Figure 3.2. To download program into a read protected, check `Unlock Before Download` option. As flash data is erased during *unlocking*, do not forget to re-download `platform` binary.

All configurations are stored in an *ini* file.

3.2.4 Python Version

SDK also provides a Python version downloader (*icsdw.py*). It's open source and easy to be integrated with other tools.

This version is written in Python 3. It uses `PySerial`[2] package to access serial port, so run "`pip install pyserial`" to install the package.

Python downloader shares the same *ini* file with only one exception: Scripting. The GUI downloader stores *RemObjects Pascal* source code with key named "*script*" in section "*options*", while the python version stores the path to a user module. The path can be a full path or a relative path (relative to the location of the *ini* file).

In the user module, two methods are required to be defined to handle events as in the GUI downloader, `on_start_run` & `on_start_bin`. Below is an example, in which, the batch round number (`batch_counter`) is written to a fixed location in #2 binary.

```
# return abort_flag
def on_start_run(batch_counter: int):
```

[2]https://pypi.org/project/pyserial/

```
    return False

# return abort_flag, new_data
def on_start_bin(batch_counter: int, bin_index: int, data: bytes):
    if bin_index != 2:
        return False, data
    ba = bytearray(data)
    addr = batch_counter.to_bytes(4, 'little')
    ba[1:5] = addr
    return False, bytes(ba)
```

3.3 ingTracer

`ingTracer` is the visual tool for inspecting recorded Trace data introduced in Debugging & Tracing.

To limit items drawn on screen, `ingTracer` breaks trace data into frames. Each frame has a length of 5sec. When a frame is selected, besides the current frame, the previous and the next one are also shown for continuity.

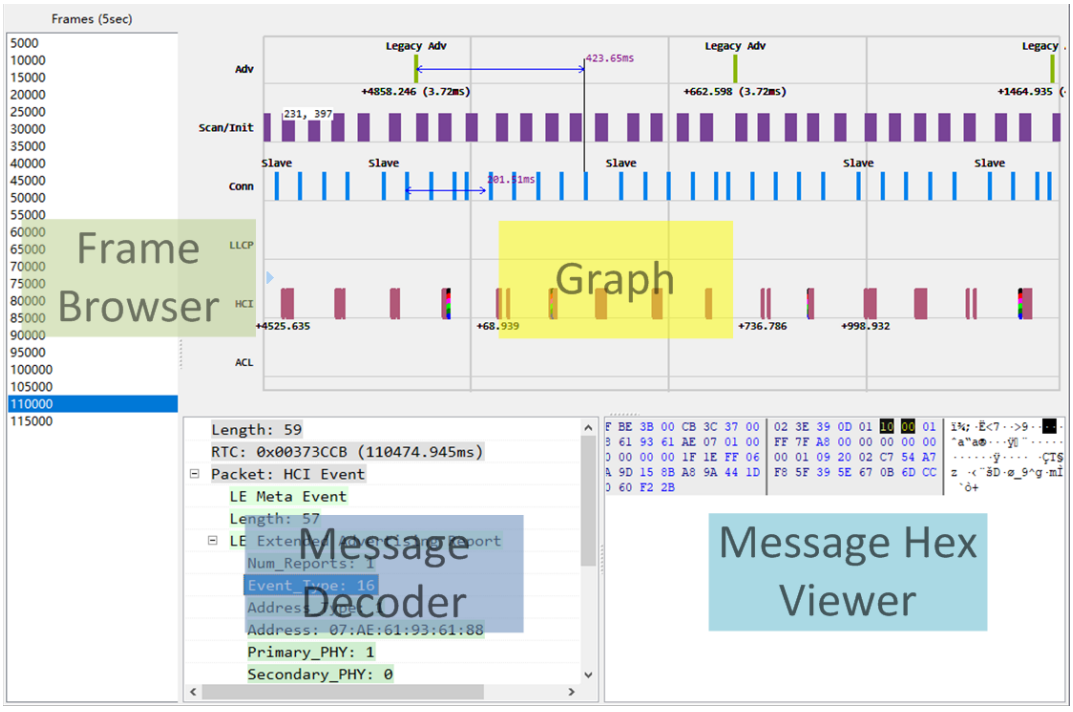

Figure 3.3: ingTracer Main UI

Graph shows all trace data visually. By clicking an item in **Graph**, detailed information is decoded and shown in **Message Decoder** and **Message Hex Viewer**. **Graph** supports some of CAD operations, such as zooming, panning, measuring, etc. Checkout menu `Help` -> `About` for detailed information. (Figure 3.3)

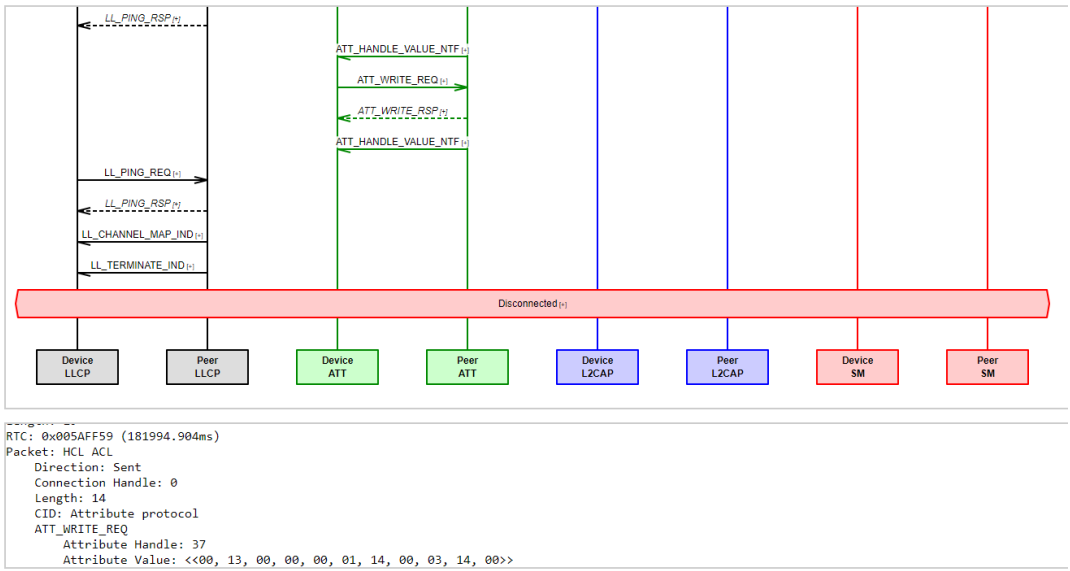

Figure 3.4: MSC Generated by ingTracer

To help analyzing app & high layer issues, `ingTrace` can generate MSC (message sequence chart) for each connection. While **Graph** emphasizes on timing between events, MSC emphasizes on procedure and fits better for protocol analysis. Message can be decoded by clicking on the `[+]` mark (Figure 3.4).

3.4 Axf Tool

`Axf Tool` is a command line tool analyzing executables and memory dump, which can be invoked from popup menu on a project in `ingWizard`. It has several functionalities:

- **stack-usage**: Statically analyze stack usage, and report call chains with top N maximum stack depth.
- **bt-api-thread-safety**: Audit the usage of Bluetooth API, and check thread confinement.
- **call-stack**: Try to recover call stack from memory dump.
- **history**: Give a brief history of BLE activities.
- **check-heap**: Try to check for errors in heaps.
- **check-task**: Runtime check of FreeRTOS tasks with the help of dump.

Use `axf_tool.exe help {function}` to get help on a specific functionality.

Chapter 4

Dive Into SDK

This chapter discusses some important topics that are critical to use SDK efficiently.

4.1 Memory Management

There are mainly three type of memory management methods:

1. Statically allocated global variables
2. Dynamically allocated and freed on stack
3. Manually allocated and freed on heap

RAM is shared between platform and user applications. When a new project is created by `ingWizard`, RAM settings is configured properly. Developers are not suggested to modify these settings.

4.1.1 Global Variables

This is the *recommended* way to define variables that have a full lift span in the app. They are allocated in the fixed location and their content can be checked easily in debugger.

4.1.2 Using Stack

For variables that are only used within a limited scope, such as a function, we can allocated them on stack.

Cares must be taken that size of stack is limited, and it might overflow if too much memory is allocated.

1. The `app_main` function & interrupts serving routines shares the same global stack with platform's `main` function.

   For RTOS bundles, this stack is defined in platform binaries as `1024` bytes, and can be replaced by a user defined one with the help of `platform_install_isr_stack`.

   For "NoOS" bundles, this stack is defined in app binariy as usual.

2. Callback functions registered into Bluetooth stack shares the same task stack with the stack task, whose size is defined as `1024` bytes, and about half is left to be used by app.

3. Developers can create new tasks by calling RTOS APIs. In these cases, stack size should be carefully examined.

Use tools to check required stack maximum depth of functions.

4.1.3 Using Heap

Generally, heap is not a recommended way for memory management in embedded applications. There are several cons included but not limited to:

- Space Overhead

  Some bytes are *wasted* to store extra information and extra program.

- Time Overhead

  It costs cycles to allocate and free memory blocks.

- Fragmentation

Based on these considerations, the heap used by `malloc` & `free` has been totally disabled by setting its size to `0`. If such heap is *TRULY* required, it can be re-enabled by changing its size to a proper value when creating projects. Be sure to check follow alternatives before using `malloc` & `free`:

- Use global variables
- Use memory pool[1]

  This is probably the choice for most cases.

- Use `FreeRTOS`'s heap and memory functions, `pvPortMalloc` & `pvPortFree`

  Note that this heap is used by platform & `FreeRTOS` itself, and it may not have too much free space left for apps. The standard `malloc` & `free` can be configured to be overridden and backed by `pvPortMalloc` & `pvPortFree` when setting up heap in `ingWizard`. Once overridden, the allocator from libc is omitted, and `malloc` & `free` are implemented by `pvPortMalloc` & `pvPortFree` respectively.

[1] https://en.wikipedia.org/wiki/Memory_pool

4.2 Multitasking

It is recommend to have a check on *Mastering the FreeRTOS™ Real Time Kernel*. Some tips:

1. Do not do too much processing in interrupt handlers, but defer it to tasks as soon as possible
2. Callback functions registered into Bluetooth stack are executed in the context of the stack task, so do not do too much processing in these functions either
3. Use message passing function `btstack_push_user_msg` or other special functions[2] to get synchronized with Bluetooth stack (see Inter-task Communication)

4.3 Interrupt Management

To create traditional ISR for interrupts, apps only need to register callback functions through a platform API `platform_set_irq_callback`.

Apps can use following APIs to modify interrupts configuration and states:

- `NVIC_SetPriority`

  Note that the highest allowed priority is `configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY + 2`, i.e. that priority parameter must be *larger* than or *equal* to this value, indicating a *lower* or *equal* priority.
- `NVIC_EnableIRQ`
- `NVIC_DisableIRQ`
- `NVIC_ClearPendingIRQ`
- etc…

4.4 Power Management

In most case, platform manages the power saving feature of ING918xx/ING9186xx SoC automatically and tries the minimize the power consumption in all circumstances, with only one exception, *deep sleep*.

In deep sleep, all components, except those required for power saving control and real-time clocks, are powered down. Some peripherals may be used by apps, and platform does not know how to configure them. So, apps have to get involved in the waking up process after deep sleep. Platform will also check with app if deep sleep is allowed, and fall back to less aggressive power saving modes when deep sleep is not allowed.

[2] `btstack_push_user_runnable`. See *Thread Safety* in *Developer's Guide for Bluetooth LE*.

To use deep sleep, two callback functions are needed, see `platform_set_evt_callback`. To ease development & debug, power saving can be turned on or off by calling `platform_config`.

Besides the above automatic power management schema, apps can also shutdown the whole system and reboot after a specified duration. In shutdown state the whole system has the least power consumption. See `platform_shutdown`. In shutdown state, a portion of data can be kept optionally at the cost of a little more power consumption. In case of only a little piece of data needs to be kept, SDK provides a pair of APIs for this, `platform_write_persistent_reg` and `platform_read_persistent_reg`.

4.5 CMSIS API

SDK tries to encapsulate CMSIS APIs to ease the development. Be careful when calling these APIs in apps as it may affect the platform program.

Following operations are strictly forbidden:

1. Changing the vector table offset register
2. Modify configurations of internal interrupts, i.e. those not listed in Table 5.1.

4.6 Debugging & Tracing

Besides online debugging, SDK provides two methods to assist debugging.

1. `printf`

   `printf` is the most convenient way to check program's behaviour. `ingWizard` can generate necessary code to use `printf`.

2. Trace

   Internal state & HCI messages can be recorded through this trace machenism. `ingWizard` can generate necessary code to use trace, too. There are several types of trace data, which are predefined and can't be changed. Which types of trace data are going to be recorded is programmable. Use

   ingTracer to view the recorded trace data.

Table 4.1: Comparison of printf and Trace

Debug Option	Pros	Cons
printf	Universal	*slow*
Trace	Binary data, fast	Data types are predefined

Both `printf` and trace can be directed to UART ports or SEGGER RTT[3]. Table 4.2 is a comparison of these two transport options.

Table 4.2: Comparison of UART and SEGGER RTT

Transport Option	Pros	Cons
UART	Universal, easy to use	Slower, consume more CPU cycles
SEGGER RTT	Fast	J-Link is required, hard to capture power up log

4.6.1 Tips on SEGGER RTT

- Use `J-LINK RTT Viewer` to view `printf` outputs in real-time.
- Use `J-LINK RTT Logger` to record trace outputs to files.

  This logger will ask for the settings of RTT. Device name is "CORTEX-M3". Target interface is "SWD". RTT Control Block address is the address of a variable named `_SEGGER_RTT`, which can be found in `.map` file. RTT channel index is `0`. Blow is a sample session.

```
------------------------------------------------------------

Device name. Default: CORTEX-M3 >
Target interface. > SWD
Interface speed [kHz]. Default: 4000 kHz >
RTT Control Block address. Default: auto-detection > 0x2000xxxx
RTT Channel name or index. Default: channel 1 > 0
Output file. Default: RTT_<ChannelName>_<Time>.log >

------------------------------------------------------------
Connected to:
  J-Link ...
  S/N: ...

Searching for RTT Control Block...OK. 1 up-channels found.
RTT Channel description:
  Index: 0
  Name:  Terminal
  Size:  500 bytes.

Output file: .....log

Getting RTT data from target. Press any key to quit.
```

[3]https://www.segger.com/products/debug-probes/j-link/technology/about-real-time-transfer/

Alternativaly, this tool can be called from command line. Address of `_SEGGER_RTT` can be specified by a range, and the tool will search for it automatically. For examples,

```
JLinkRTTLogger.exe -If SWD -Device CORTEX-M3 -Speed 4000
-RTTSearchRanges "0x20005000 0x8000"
-RTTChannel 0
file_name
```

4.6.2 Memory Dump

We are committed to delivery high quality platform binary. If an assertion or hard fault had occurred in platform binary, it is suggested to create a full memory dump and save all registers. Check out Developers' Guide for addresses of all memory regions. After a dump is got, use Axf Tool to analyze it. If the problem can not be resolved, contact technical support.

Memory can be dumped through debuggers:

- Keil µVision

  In debug session, open the Command Window, use `save` to save each memory region. Take ING918xx as an example:

  ```
  save sysm.hex  0x20000000,0x2000FFFF
  save share.hex 0x400A0000,0x400AFFFF
  ```

- J-Link Commander

  Once connected, use `regs` to shows all current register values, and `savebin` to save target memory into binary file. Take ING918xx as an example:

  ```
  savebin sysm.bin  0x20000000 0x10000
  savebin share.bin 0x400A0000 0x10000
  ```

- IAR Embedded Workbench

  In debug session, open a Memory window, and select "Memory Save …" from popup menu.

- Rowley Crossworks for ARM & SEGGER Embedded Studio for ARM

  In debug session, open a Memory window, for each memory region:

  1. Fill in the start address and size;
  2. Use "Memory Save …" from popup menu.

- GDB (GNU Arm Embedded Toolchain and Nim)

  In GDB debug session, use `dump` command to save each memory region.

Memory can be also dumped by a piece of specific code. For example, in the event handler of `PLATFORM_CB_EVT_ASSERTION`, dump all memory data to UART.

Chapter 5

Platform API Reference

5.1 `platform_32k_rc_auto_tune`

Automatically tune the internal 32k RC clock, and get the tuning value.

5.1.1 Prototype

```
uint16_t platform_32k_rc_auto_tune(void);
```

5.1.2 Parameters

Void.

5.1.3 Return Value

The 16-bits tuning value.

5.1.4 Remarks

This operation costs ~250ms. It is recommended to call this once and store the returned value for later usage.

5.1.5 Example

```
uint16_t value = platform_32k_rc_auto_tune();
```

5.2 platform_32k_rc_tune

Tune internal the 32k RC clock with the tune value.

5.2.1 Prototype

```
void platform_32k_rc_tune(uint16_t value);
```

5.2.2 Parameters

- `uint16_t value`
  Value used to tune the clock (returned by `platform_32k_rc_auto_tune`)

5.2.3 Return Value

Void.

5.2.4 Remarks

void.

5.2.5 Example

```
platform_32k_rc_tune(value);
```

5.3 platform_config

Configure some platform functionalities.

5.3.1 Prototype

```
void platform_config(const platform_cfg_item_t item,
                     const uint32_t flag);
```

5.3.2 Parameters

- `const platform_cfg_item_t item`

  Specify the item to be configured. It can be one of following values:

  - `PLATFORM_CFG_LOG_HCI`: Host controller interface messages. Default: Disabled.
  - `PLATFORM_CFG_POWER_SAVING`: Power saving. Default: Disabled.
  - `PLATFORM_CFG_TRACE_MASK`: Bit map of selected trace items. Default: `0`.

    ```
    typedef enum
    {
        PLATFORM_TRACE_ID_EVENT                 = 0,
        PLATFORM_TRACE_ID_HCI_CMD               = 1,
        PLATFORM_TRACE_ID_HCI_EVENT             = 2,
        PLATFORM_TRACE_ID_HCI_ACL               = 3,
        PLATFORM_TRACE_ID_LLCP                  = 4
    } platform_trace_item_t;
    ```

  - `PLATFORM_CFG_RC32K_EN`: Enable/Disable RC 32k clock. Default: Enabled.
  - `PLATFORM_CFG_OSC32K_EN`: Enable/Disable 32k crystal oscillator. Default: Enabled.
  - `PLATFORM_CFG_32K_CLK`: 32k clock selection. Flag is `platform_32k_clk_src_t`. Default: `PLATFORM_32K_RC`

    ```
    typedef enum
    {
        PLATFORM_32K_OSC,           // External 32k crystal oscillator
        PLATFORM_32K_RC             // Internal RC 32k clock
    } platform_32k_clk_src_t;
    ```

    When modifying this configuration, both RC32K and OSC32K should be **enabled** and **run**:

    ```
    * For OSC32K, wait until status of OSC32K is OK;
    * For RC32K, wait 100us after enabled.
    ```

Note: Wait another 100us before disabling the unused clock.

- PLATFORM_CFG_32K_CLK_ACC: Configure 32k clock accurary in ppm.
- PLATFORM_CFG_32K_CALI_PERIOD: 32K clock auto-calibartion period in seconds. Default: 3600 * 2 (2 hours).
- PLATFORM_CFG_DEEP_SLEEP_TIME_REDUCTION: Sleep time reduction (deep sleep mode) in micro seconds. Default: ~550us.
- PLATFORM_CFG_SLEEP_TIME_REDUCTION: Sleep time reduction (other sleep mode) in micro seconds. Default: ~450us.
- PLATFORM_CFG_LL_DBG_FLAGS: Link layer flags. Combination of bits in ll_cfg_flag_t.

```
typedef enum
{
    LL_FLAG_DISABLE_CTE_PREPROCESSING = 1,  // disable CTE processing
    LL_FLAG_LEGACY_ONLY_INITIATING = 4,     // initiating only using legacy ADV
    LL_FLAG_LEGACY_ONLY_SCANNING = 8,       // scanning only using legacy ADV
} ll_cfg_flag_t;
```

- PLATFORM_CFG_LL_LEGACY_ADV_INTERVAL: Link layer legacy advertising intervals for high duty cycle (higher 16bits) and normal duty cycle (lower 16bits) in micro seconds. Default for high duty cycle: 1250; default for normal duty cycle: 1500.

- const uint32_t flag

  To disable or enable an item. It can be one of following values:

  - PLATFORM_CFG_ENABLE
  - PLATFORM_CFG_DISABLE

5.3.3 Return Value

Void.

5.3.4 Remarks

Void.

5.3.5 Example

```
// Enable HCI logging
platform_config(PLATFORM_CFG_LOG_HCI, PLATFORM_CFG_ENABLE);
```

5.4 platform_get_heap_status

Get current heap status, such as available size, etc.

5.4.1 Prototype

```
void platform_get_heap_status(platform_heap_status_t *status);
```

5.4.2 Parameters

- `platform_heap_status_t *status`

  Heap status.

5.4.3 Return Value

Void.

5.4.4 Remarks

Heap status is defined as:

```
typedef struct
{
    uint32_t bytes_free;                // total free bytes
    uint32_t bytes_minimum_ever_free;   // mininum of bytes_free from startup
} platform_heap_status_t;
```

5.4.5 Example

```
platform_heap_status_t status;
platform_get_heap_status(&status);
```

5.5 platform_get_us_time

Read the internal timer counting from BLE initialization.

5.5.1 Prototype

```
int64_t platform_get_us_time(void);
```

5.5.2 Parameters

Void.

5.5.3 Return Value

Value of the internal timer counting at 1us.

5.5.4 Remarks

This timer restarts after shutdown, while RTC timer does not.

5.5.5 Example

```
platform_get_us_time();
```

5.6 platform_get_version

Get version number of platform.

5.6.1 Prototype

```
const platform_ver_t *platform_get_version(void);
```

5.6.2 Parameters

Void.

5.6.3 Return Value

Pointer to `platform_ver_t`.

5.6.4 Remarks

Platform version number has three parts, major, minor and patch:

```
typedef struct platform_ver
{
    unsigned short major;
    char  minor;
    char  patch;
} platform_ver_t;
```

5.6.5 Example

```
const platform_ver_t *ver = platform_get_version();
printf("Platform version: %d.%d.%d\n", ver->major, ver->minor, ver->patch);
```

5.7 `platform_hrng`

Generate random bytes by using hardware random-number generator.

5.7.1 Prototype

```
void  platform_hrng(uint8_t *bytes, const uint32_t len);
```

5.7.2 Parameters

- `uint8_t *bytes`

  Random data output.

- `const uint32_t len`

  Number of random bytes to be generated.

5.7.3 Return Value

Void.

5.7.4 Remarks

Time consumption to generate a fix length of data is undetermined.

5.7.5 Example

```
uint32_t strong_random;
platform_hrng(&strong_random, sizeof(strong_random));
```

5.8 platform_install_isr_stack

Install a new stack for ISR.

5.8.1 Prototype

```
void platform_install_isr_stack(void *top);
```

5.8.2 Parameters

- `void *top`

  Top of the new stack, which must be 4-bytes aligned.

5.8.3 Return Value

Void.

5.8.4 Remarks

In case apps need a much larger stack than the default one in ISR, a new stack can be installed to replace the default one.

This function is only allowed to be called in `app_main`. The new stack is put into use after `app_main` returns.

5.8.5 Example

```
uint32_t new_stack[2048];
...
platform_install_isr_stack(new_stack + sizeof(new_stack) / sizeof(new_stack[0]));
```

5.9 platform_printf

The `printf` function stored in platform binary.

5.9.1 Prototype

```
void  platform_printf(const char *format, ...);
```

5.9.2 Parameters

- `const char *format`

  Format string.

- `...`

  Variable arguments for format string.

5.9.3 Return Value

Void.

5.9.4 Remarks

There are pros & cons to use this function.

Pros:

- This function is located in platform binary, app binary size can be saved.

Cons:

- Output is directed `PLATFORM_CB_EVT_PUTC` event, so its callback function must be defined.

5.9.5 Example

```
platform_printf("Hello world");
```

5.10 platform_raise_assertion

Raise a software assertion.

5.10.1 Prototype

```
void  platform_raise_assertion(const char *file_name, int line_no);
```

5.10.2 Parameters

- `const char *file_name`
  File name where the assertion occurred.
- `int line_no`
  Line number where the assertion occurred.

5.10.3 Return Value

Void.

5.10.4 Remarks

Void.

5.10.5 Example

```
if (NULL == ptr)
  platform_raise_assertion(__FILE__, __LINE__);
```

5.11 platform_rand

Generate a pseudo random integer by internal PRNG.

5.11.1 Prototype

```
int platform_rand(void);
```

5.11.2 Parameters

Void.

5.11.3 Return Value

A pseudo random integer in range of 0 to RAND_MAX.

5.11.4 Remarks

Seed of the internal PRNG is initialized by HRNG at startup.

5.11.5 Example

```
printf("rand: %d\n", platform_rand());
```

5.12 platform_read_info

Read platform information

5.12.1 Prototype

```
uint32_t platform_read_info(const platform_info_item_t item);
```

5.12.2 Parameters

- `const platform_info_item_t item`

  Information item.

```
typedef enum
{
    PLATFORM_INFO_OSC32K_STATUS,     // Read status of 32k crystal oscillator.
                                     // Value 0: not OK; Non-0: OK
} platform_info_item_t;
```

5.12.3 Return Value

Value of the information item.

5.12.4 Remarks

Void.

5.12.5 Example

```
platform_read_info(PLATFORM_INFO_OSC32K_STATUS);
```

5.13 platform_read_persistent_reg

Read value from the persistent register. See also `platform_write_persistent_reg`.

5.13.1 Prototype

```
uint32_t platform_read_persistent_reg(void);
```

5.13.2 Parameters

Void.

5.13.3 Return Value

The **four* bits value writen by `platform_write_persistent_reg`.

5.13.4 Remarks

Void.

5.13.5 Example

```
platform_read_persistent_reg();
```

5.14 platform_reset

Reset platform (SoC).

5.14.1 Prototype

```
void platform_reset(void);
```

5.14.2 Parameters

Void.

5.14.3 Return Value

Void.

5.14.4 Remarks

When calling this function, the code after it will not be executed.

5.14.5 Example

```
  if (out-of-memory)
    platform_reset();
```

5.15 platform_set_evt_callback

Registers callback functions to platform events.

5.15.1 Prototype

```
void platform_set_evt_callback(platform_evt_callback_type_t type,
                               f_platform_evt_cb f,
                               void *user_data);
```

5.15.2 Parameters

- `platform_evt_callback_type_t type`

  Specify the event type to which the callback function is registered. It can be one of following values:

  - `PLATFORM_CB_EVT_PUTC`: Ouput ASCII character event

    When platform want to output ASCII characters for logging, this event is fired. Parameter `void *data` passed into the callback function is casted from `char *`.

    `ingWizard` can automatically generate code that redirects platform log to UART if `Print to UART` is checked on `Common Function` when creating a new project.

  - `PLATFORM_CB_EVT_PROFILE_INIT`: Profile initialization event

    When host initializes, this event is fired to request app to initialize GATT profile.

    `ingWizard` can automatically generate code for this event when creating a new project.

  - `PLATFORM_CB_EVT_ON_DEEP_SLEEP_WAKEUP`: Wakeup from deep sleep event

    When waking up from deep sleep, this event is fired. During deep sleep, peripheral interfaces (such as UART, I2C, etc) are all powered off. So, when waking up, these interfaces might need to be re-initialized.

    `ingWizard` can automatically generate code for event if `Deep Sleep` is checked on `Common Function` when creating a new project.

  - `PLATFORM_CB_EVT_QUERY_DEEP_SLEEP_ALLOWED`: Query if deep sleep is allowed event

    When platform prepares to enter deep sleep mode, this event is fired to query app if deep sleep is allow at this moment. Callback function can reject deep sleep by returning `0`, and allow it by returning a non-`0` value.

    `ingWizard` can automatically generate code for event if `Deep Sleep` is checked on `Common Function` when creating a new project.

  - `PLATFORM_CB_EVT_HARD_FAULT`: Hard fault occurs

    When hard fault occurs, this event is fired. Parameter `void *data` passed into the callback function is casted from `hard_fault_info_t *`. If this callback is not defined, CPU enters a dead loop when hard fault occurs.

  - `PLATFORM_CB_EVT_ASSERTION`: Software assertion fails

    When software assertion fails, this event is fired. Parameter `void *data` passed into the callback function is casted from `assertion_info_t *`. If this callback is not defined, CPU enters a dead loop when assertion occurs.

  - `PLATFORM_CB_EVT_LLE_INIT`: Link layer engine initialized.

    When link layer engine initialized, this event is fired.

  - `PLATFORM_CB_EVT_HEAP_OOM`: Out of memory.

    When allocation on heap fails (heap out of memory), this event is fired. If this event is fired and no callback is defined, CPU enters a dead loop.

  - `PLATFORM_CB_EVT_TRACE`: Trace output.

When a trace item is emitted, this event is fired. Apps can define a callback function for this event to save or log trace output. `param` to the callback is casted from `platform_trace_evt_t *` (See Debugging & Tracing).

```
typedef struct
{
    const void *data1;
    const void *data2;
    uint16_t len1;
    uint16_t len2;
} platform_evt_trace_t;
```

A trace item is a combination of `data1` and `data2`. Note:

1. `len1` or `len2` might be 0, but not both;
2. If callback function finds that it can't output data of size `len1` + `len2`, then, both `data1` & `data2` should be discarded to avoid trace item corruption.

- `f_platform_evt_cb f`

  The callback function registered to event `type`. `f_platform_evt_cb` is:

```
typedef uint32_t (*f_platform_evt_cb)(void *data, void *user_data);
```

- `void *user_data`

  This is passed to callback function's `user_data` unchanged.

5.15.3 Return Value

Void.

5.15.4 Remarks

It is not required to register callback functions to each event.

If no callback function is registered to `PLATFORM_CB_EVT_PUTC` event, all platform log is discarded.

If no callback function is registered to `PLATFORM_CB_EVT_PROFILE_INIT` event, BLE device's profile is empty.

If no callback function is registered to `PLATFORM_CB_EVT_ON_DEEP_SLEEP_WAKEUP` event, app will not be notified when waking up from deep sleep.

If no callback function is registered to `PLATFORM_CB_EVT_QUERY_DEEP_SLEEP_ALLOWED` event, deep sleep is *disabled*.

5.15.5 Example

```
uint32_t cb_putc(char *c, void *dummy)
{
    // TODO: output char c to UART
    return 0;
}

......

platform_set_evt_callback(PLATFORM_CB_EVT_PUTC, (f_platform_evt_cb)cb_putc,
                          NULL);
```

5.16 `platform_set_irq_callback`

Registers callback functions to interrupt requests.

Developers do not need to define IRQ handlers in apps, but use callback functions instead. There are 11 such IRQs, summarized in Table 5.1.

Table 5.1: IRQ Summary

Peripheral Type	Number	Notes
RTC	1	Real Time Clock
TIMER	3	Timer
GPIO	1	General Purpose Input/Output
SPI	2	Serial Peripheral Interface
UART	2	Universal Asynchronous Receiver-Transmitter
I2C	2	Inter-Integrated Circuit

5.16.1 Prototype

```
void platform_set_irq_callback(platform_irq_callback_type_t type,
                               f_platform_irq_cb f,
                               void *user_data);
```

5.16.2 Parameters

- `platform_irq_callback_type_t type`

  Specify the IRQ type to which the callback function is registered. It can be one of following values:

```
PLATFORM_CB_IRQ_RTC,
PLATFORM_CB_IRQ_TIMER0,
PLATFORM_CB_IRQ_TIMER1,
PLATFORM_CB_IRQ_TIMER2,
PLATFORM_CB_IRQ_GPIO,
PLATFORM_CB_IRQ_SPI0,
PLATFORM_CB_IRQ_SPI1,
PLATFORM_CB_IRQ_UART0,
PLATFORM_CB_IRQ_UART1,
PLATFORM_CB_IRQ_I2C0,
PLATFORM_CB_IRQ_I2C1
```

- `f_platform_irq_cb f`

  The callback function registered to IRQ `type`. `f_platform_irq_cb` is:

```
typedef uint32_t (*f_platform_irq_cb)(void *user_data);
```

- `void *user_data`

  This is passed to callback function's `user_data` unchanged.

5.16.3 Return Value

Void.

5.16.4 Remarks

When a callback function is registered to an IRQ, the IRQ is enabled automatically.

5.16.5 Example

```
uint32_t cb_irq_uart0(void *dummy)
{
    // TODO: add UART0 IRQ handling code
    return 0;
}

......

platform_set_irq_callback(PLATFORM_CB_IRQ_UART0, cb_irq_uart0,
                          NULL);
```

5.17 platform_shutdown

Bring the whole system into shutdown state, and reboot after a specified duration. Optionally, a portion of memory can be retentioned during shutdown, and apps can continue to use it after reboot.

Note that this function will NOT return except that shutdown procedure fails to initiate. Possible causes for failures include:

1. External wake-up signal is issued;
2. Input parameters are not proper;
3. Internal components are busy.

5.17.1 Prototype

```
void platform_shutdown(const uint32_t duration_cycles,
                       const void *p_retention_data,
                       const uint32_t data_size);
```

5.17.2 Parameters

- `const uint32_t duration_cycles`

  Duration (measured in cycles of 32k clock) before power on again (reboot). The minimum duration is 825 cycles (about 25.18ms). If `0` is used, the system will stay in shutdown state until external wake-up signal is issued.

- `const void *p_retention_data`

  Pointer to the start of data to be retentioned. Only data within SYSTEM memory can be retentioned. This parameter can be set to `NULL` when `data_size` is `0`.

- `data_size`

  Size of the data to be retentioned. Set to `0` when memory retention is not needed.

5.17.3 Return Value

Void.

5.17.4 Remarks

Void.

5.17.5 Example

```
// Shutdown the system and reboot after 1s.
platform_shutdown(32768, NULL, 0);
```

5.18 `platform_switch_app`

Switch to a secondary app.

5.18.1 Prototype

```
void platform_switch_app(const uint32_t app_addr);
```

5.18.2 Parameters

- `const uint32_t app_addr`

  Entry address of the secondary app.

5.18.3 Return Value

Void.

5.18.4 Remarks

When calling this function, the code after it will not be executed.

5.18.5 Example

```
platform_switch_app(0x80000);
```

5.19 platform_write_persistent_reg

Write a value to the persistent register. This value is kept even in power saving, shutdown mode, or when switching to another app.

5.19.1 Prototype

```
void platform_write_persistent_reg(const uint8_t value);
```

5.19.2 Parameters

- `const uint8_t value`

  The value.

5.19.3 Return Value

Void.

5.19.4 Remarks

Only **four** bits are saved.

5.19.5 Example

```
platform_write_persistent_reg(1);
```

5.20 sysSetPublicDeviceAddr

Set the public address of device.

The public address of a BLE device is a 48-bit extended unique identifier (EUI-48) created in accordance with the IEEE 802-2014 standard[1].

INGCHIPS 918xx *DO NOT* have public addresses. This function should *ONLY* be used for debugging or testing, and *NEVER* be used in final products.

5.20.1 Prototype

```
void sysSetPublicDeviceAddr(const unsigned char *addr);
```

5.20.2 Parameters

- `const unsigned char *addr`

  New public address.

5.20.3 Return Value

Void.

5.20.4 Remarks

In order to avoid potential issues, this function should be called before calling any GAP functions. It is recommended to call this function in `app_main` or `PLATFORM_CB_EVT_PROFILE_INIT` event callback function.

[1] http://standards.ieee.org/findstds/standard/802-2014.html

5.20.5 Example

```
const unsigned char pub_addr[] = {1,2,3,4,5,6};
sysSetPublicDeviceAddr(pub_addr);
```

Chapter 6

Revision History

Version	Notes	Date
1.0	Initial release	2020-07-28
1.1	Add Python downloader	2020-10-10
1.2	Update API descriptions	2020-07-05
1.2.1	Update memory dump section	2020-08-02
1.2.2	Fix typos, other minor updates	2020-09-08
1.2.3	Fix order of versions in “Device With FOTA” and typo	2020-09-09
1.2.4	Fix outdated information in tutorials	2020-10-20
1.2.5	Update for “NoOS” bundles	2020-11-15
1.2.6	Add ING9168xx	2022-01-10
1.2.7	Minor fixes	2022-07-30
1.2.8	Add `btstack_push_user_runnable`	2022-10-31
1.2.9	Add information about `Axf Tool`	2023-10-23