INGCHIPS SDK 开发者用户手册

桃芯科技

官网：www.ingchips.com

www.ingchips.cn

邮箱：service@ingchips.com

电话：010-85160285

地址：北京市海淀区知春路 49 号紫金数 3 号楼 8 层 803

深圳市南山区科技园曙光大厦 1009

目录

插图

表格

欢迎

欢迎使用 *INGCHIPS* 918xx/916xx 软件开发工具包（SDK）.

INGCHIPS 918xxx/916xx 是 BLE 5.x 全功能 SoC 解决方案。本手册将带您从软件角度深入了解如何在 918xx/916xx 上进行 BLE 开发。

第一章　简介

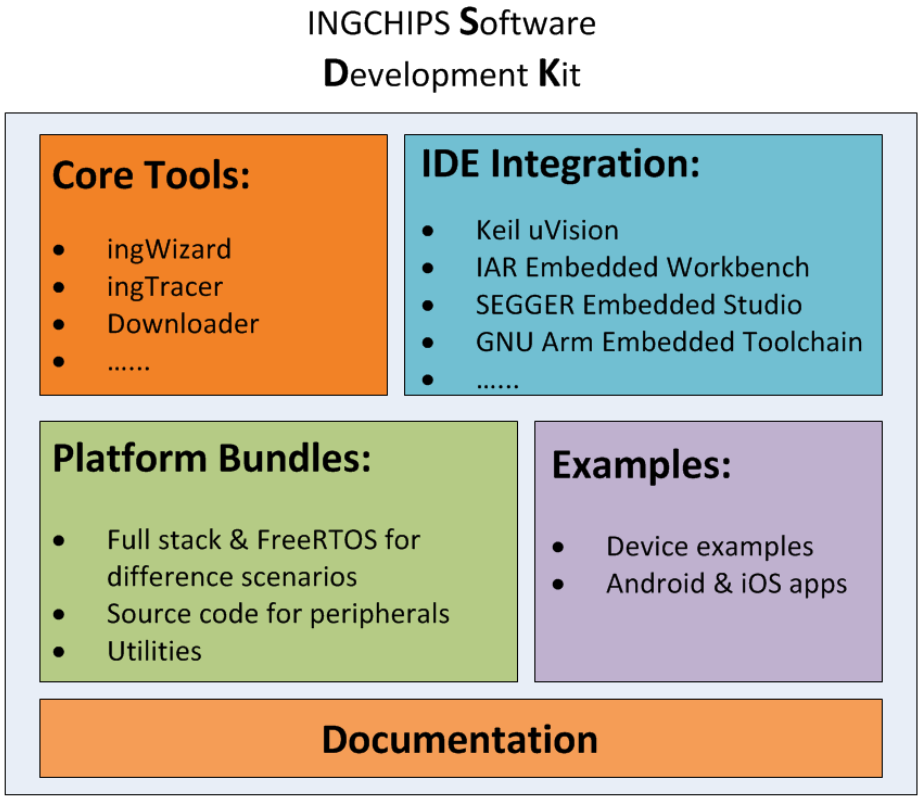

图 1.1: SDK Overview

INGCHIPS 软件开发工具包包含以下主要组件（图 1.1）：

1. 核心工具

   提供项目向导，Flash 下载器及其它功能。这些工具可使 BLE 开发更简单、流畅。

2. 语言和 IDE 集成

支持 Keil µVision[1]，IAR Embedded Workbench[2]，Rowley Crossworks for ARM[3]，SEGGER Embedded Studio for ARM[4]。支持 GNU Arm Embedded Toolchain[5]。核心工具可以自动配置所有这些 IDE、工具链。

3. 平台软件包

   为不同应用场景提供不同的软件包（比如 typical，extension）。每个软件包包含完整的协议栈和（可选的）FreeRTOS 可执行二进制文件及对应的 C 头文件。提供访问芯片外设所需的源代码。

4. 示例

   提供丰富的 BLE 设备示例代码，及对应的 Android、iOS 参考代码。

5. 文档

   提供用户手册（本文档），API 参考，应用指南等。

1.1 范围

本手册介绍平台软件包架构，核心工具，和平台 API。

1.2 架构

平台软件包分为两种类型，一种内置 FreeRTOS（称为 RTOS 软件包），一种不内置 RTOS（称为 “NoOS” 软件包）。

1.2.1 RTOS 软件包

ING918xx/ING916xx 软件架构如图 1.2 所示。Bootloader 存储于 ROM，不可修改，而平台和应用存储于 Flash。BLE 协议栈、FreeRTOS 及部分 SoC 功能编译为平台可执行程序。当系统启动时，平台可执行程序首先完成初始化，然后加载主应用。

[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

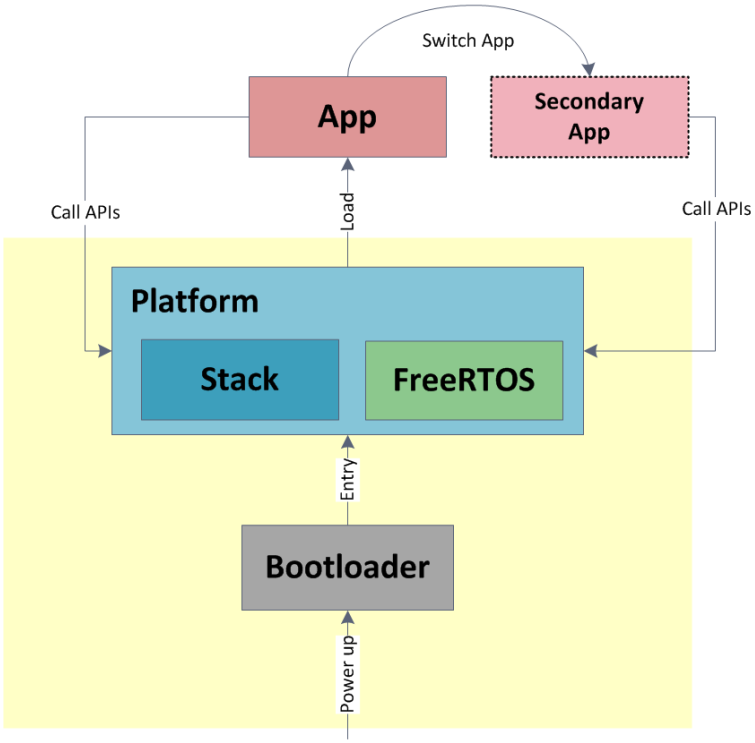

图 1.2: Architecture

辅应用以编程方式告知平台运行。可以下载多个辅应用，编程切换。芯片复位后，平台又会正常加载主应用。主应用的入口地址由 SDK 工具自动管理，辅应用的入口地址由开发者指定。

1.2.1.1 使用 c 语言开发 App

App 的主函数名为 `app_main`，app 在这个函数里进行初始化：

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

`app_main` 应该总是返回 `0`。

平台，BLE 协议栈和 FreeRTOS 的 API 在相应的 c 头文件里声明，包含头文件就可以使用。

1.2.1.2 其它语言

可以使用其它语言开发应用，如：

- Rust[6]
- Nim[7]
- Zig[8]

1.2.2 “NoOS” 软件包

当需要使用另外的 RTOS，或者需要使用那些 RTOS 软件包中未提供的功能时，开发者可以选用 “NoOS” 软件包。

“NoOS” 软件包定义了一个通用 RTOS 接口，开发者需要实现这个接口，并通过 `app_main` 的返回值告知平台。

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

1.3 缩略语和术语

表 1.1: 缩略语

缩略语	解释
ATT	Attribute Protocol （属性协议）
BLE	Bluetooth Low Energy （低功耗蓝牙）
FOTA	Firmware Over-The-Air （固件空中升级）

[6] https://ingchips.github.io/blog/2022-09-24-use-rust/

[7] SDK 示例：*Smart Home Hub*。

[8] SDK 示例：*Central FOTA*。

缩略语	解释
IRQ	Interrupt Request （终端请求）
GAP	Generic Access Profile （通用存取配置）
GATT	Generic Attribute Profile （通用属性配置）
RAM	Random Access Memory （随机存取存储器）
ROM	Read Only Memory （只读存储器）
SDK	Software Development Kit （软件开发工具包）

表 1.2: 术语

Terminology	Notes
Flash Memory	一种电子非易失性计算机存储介质
FreeRTOS	一种实行操作系统内核

1.4 参考资料

1. Host API Reference
2. Bluetooth SIG[9]
3. FreeRTOS[10]
4. Mastering the FreeRTOS™ Real Time Kernel[11]

[9] https://www.bluetooth.com/

[10] https://freertos.org

[11] https://www.freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

第二章　教程

下面的教程将一步一步地讲解 SDK 里的基本概念，核心工具的使用方法。

2.1　世界你好

在本教程里，我们将创建一个设备，发送的广播里带着它的名字：“Hello, 世界”。

从开始菜单里打开 `Wizard`，选择菜单 `Project -> New Project ...`。这个菜单项会打开项目向导。向导的第一页是 `Development Tool` （见图 2.1）。

2.1.1　`Development Tool` 页面

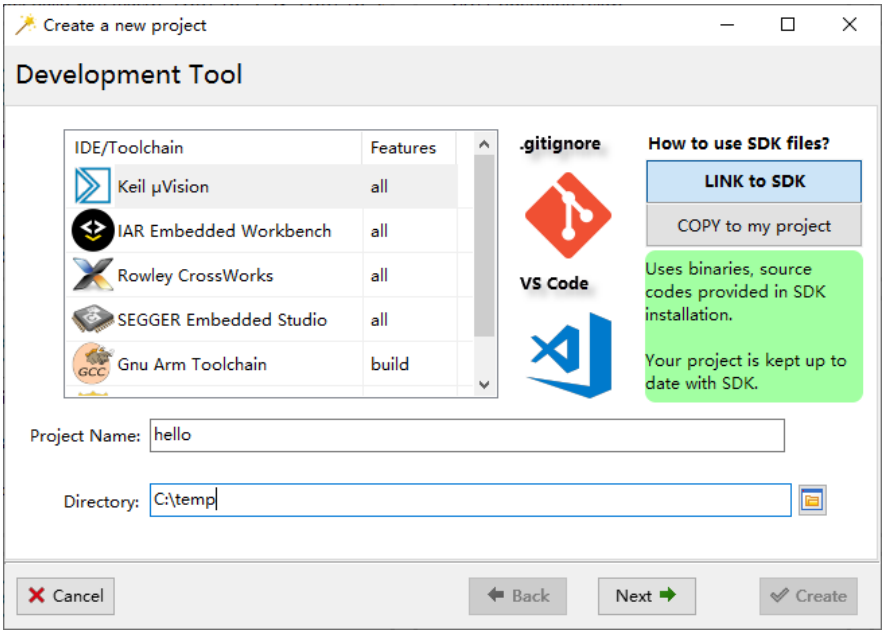

图 2.1: 选择项目类型

在这个页面（图 2.1）：

1. 选择 IDE/工具链
2. 设置项目名称
3. 选择项目的存储位置

Wizard 提供以下便利功能：

- 如果需要用 Git 做软件版本管理，选择 Setup .gitignore；
- 如果准备使用 Visual Studio Code 作为代码编辑器，选择 Setup Visual Studio Code。

然后点击 Next 按钮进入下一页，Choose Chip Series。

2.1.2 Choose Chip Series 页面

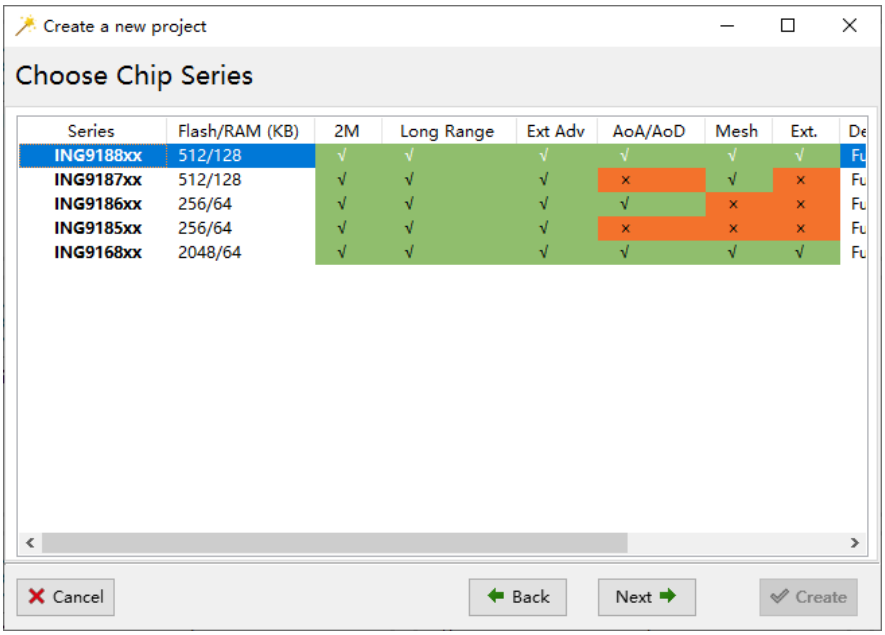

图 2.2: Choose Chip Series

在这个页面（图 2.2）选择项目的目标芯片型号，然后点击 Next 进入下一页，Choose Project Type。

2.1.3 Choose Project Type 页面

在这个页面（图 2.3），选择 Typical。

然后点击 Next 进入下一页，Role of Your Device。

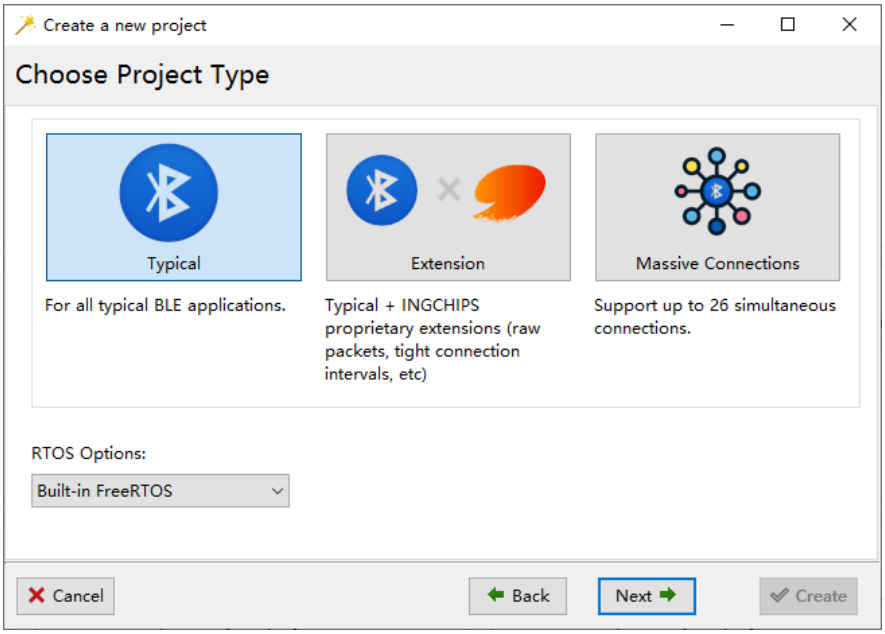

图 **2.3:** Choose Project Type

2.1.4 `Role of Your Device` 页面

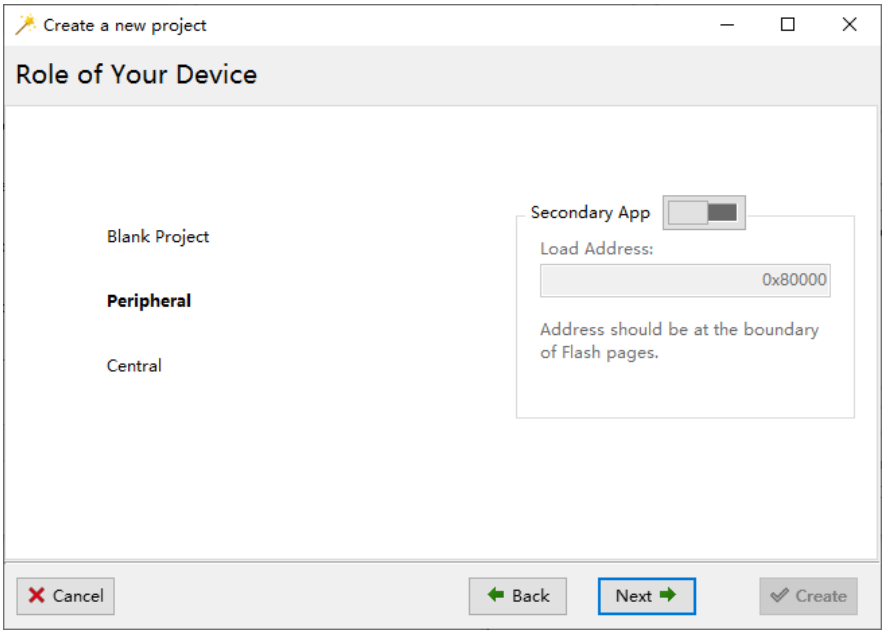

图 **2.4:** Role of Your Device

在这个页面（图 2.4），选择 Peripheral，然后点击 `Next` 进入下一页，`Peripheral Setup`。

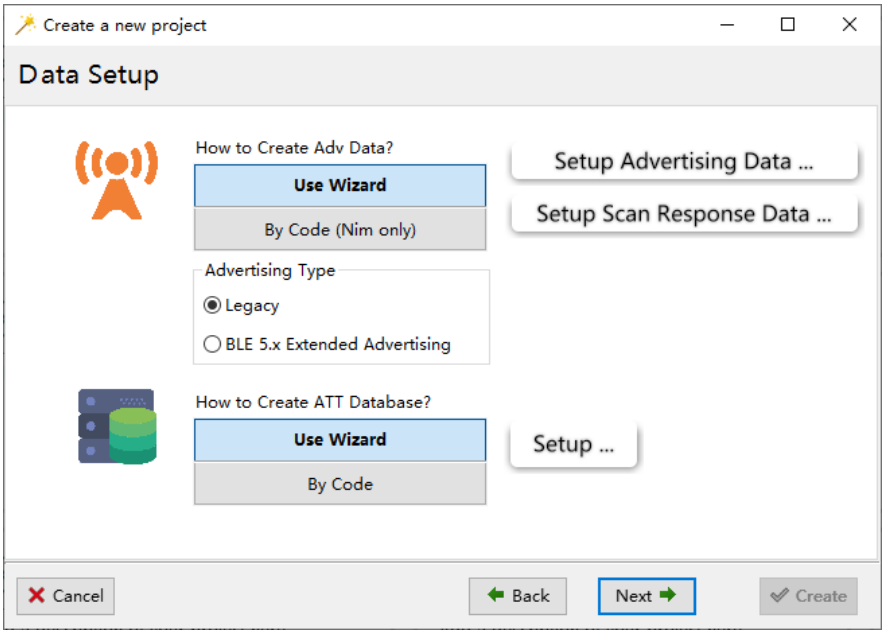

图 2.5: Peripheral Setup

2.1.5 Peripheral Setup 页面

在这个页面（图 2.5），选择 Legacy 广播方式。

支持 BLE 5.x 扩展广播的手机目前依然凤毛麟角，即使声称了“支持” BLE 5.0。为了更好的兼容性，这里我们选择使用 Legacy 广播。此外，待项目创建之后，通过修改一个比特就能将 Legacy 广播切换成 BLE 5.x 扩展广播。

点击 Setup Advertising Data 按钮，打开广播数据编辑器（图 2.6）。在编辑器里，输入 name 可以快速定位到 GAP 广播项 09 - «Complete Local Name»，点击 Add 将这个项添加到设备的广播数据里。

点击刚刚添加的 09 - «Complete Local Name» 数据项，然后将在下面的数据编辑框里输入“Hello, 世界” 并按回车。Data Preview 会更新，整个广播都将已原始字节流的形式显示其中。显而易见，工具对中文字符使用了 UTF-8 编码。

现在点击 OK 回到项目向导，然后点击 Next 进入下一页，Security & Privacy。

2.1.6 Security & Privacy 页面

保持默认值（图 2.7），然后点击 Next 进入下一页，Firmare Over-The-Air。

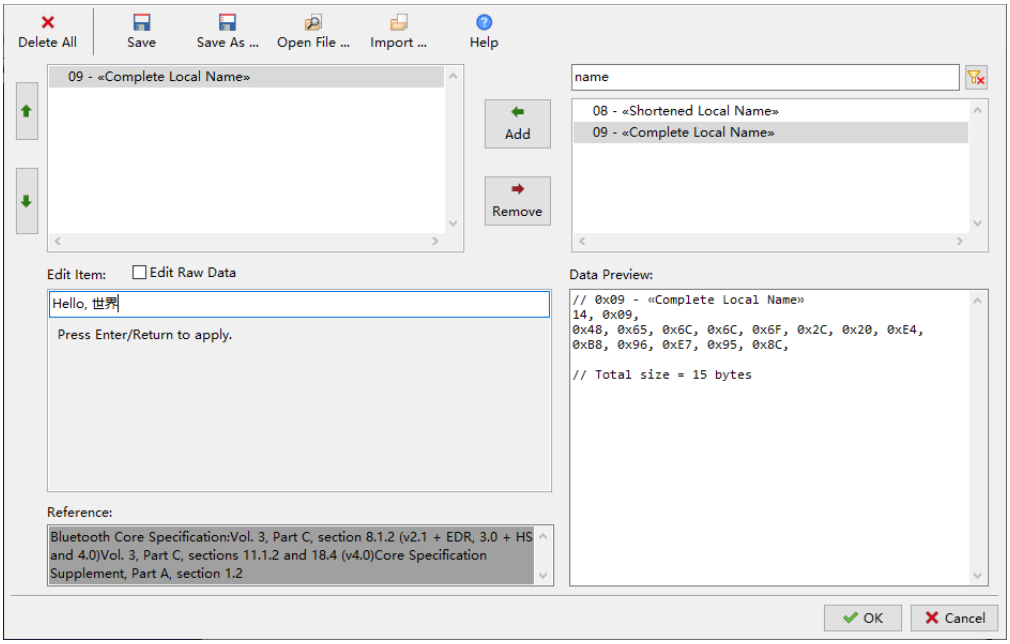

图 **2.6:** Edit Advertising Data

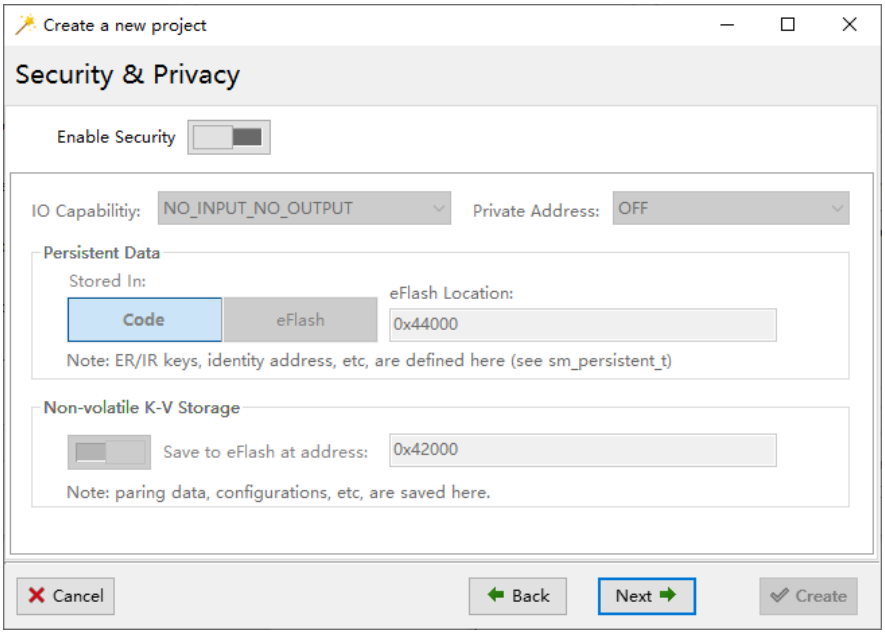

图 **2.7:** Firmare Over-The-Air

2.1.7 Firmare Over-The-Air 页面

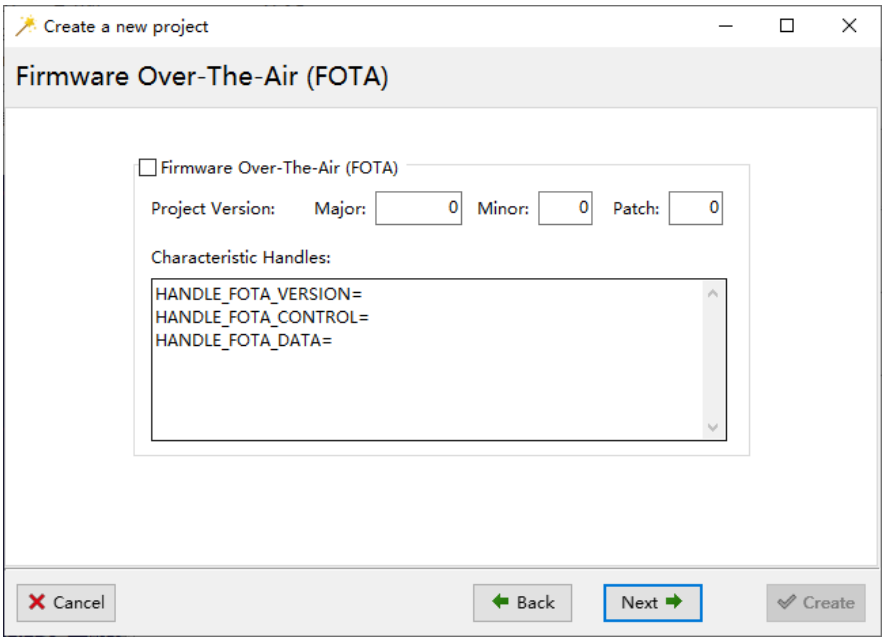

图 2.8: Firmare Over-The-Air

保持默认值（图 2.8），然后点击 Next 进入下一页，Common Functions.

2.1.8 Common Functions 页面

在这一页（图 2.9），我们依然保持默认值不变，点击 Create。现在项目创建好了（图 2.10），可以随时编译、下载。

2.1.9 编译您的项目

回到 Wizard 的主窗口（图 2.10），点击打开您的项目。在 IDE 里编译您的项目。

2.1.10 下载

回到 Wizard（图 2.10），右键点击您的项目，从弹出的快捷菜单中选择 Download to Flash 就可以打开下载工具（图 2.11）。

除了串口号，下载工具的所有设置都已就绪。设置串口口，然后点击 Start。

下载完成之后，用 LightBlue、INGdemo （图 2.12）或者其它 app 检查是否可以找到一个名为 “Hello, 世界” 的设备。注意，这个设备目前可能无法在系统设置的蓝牙菜单里看到。

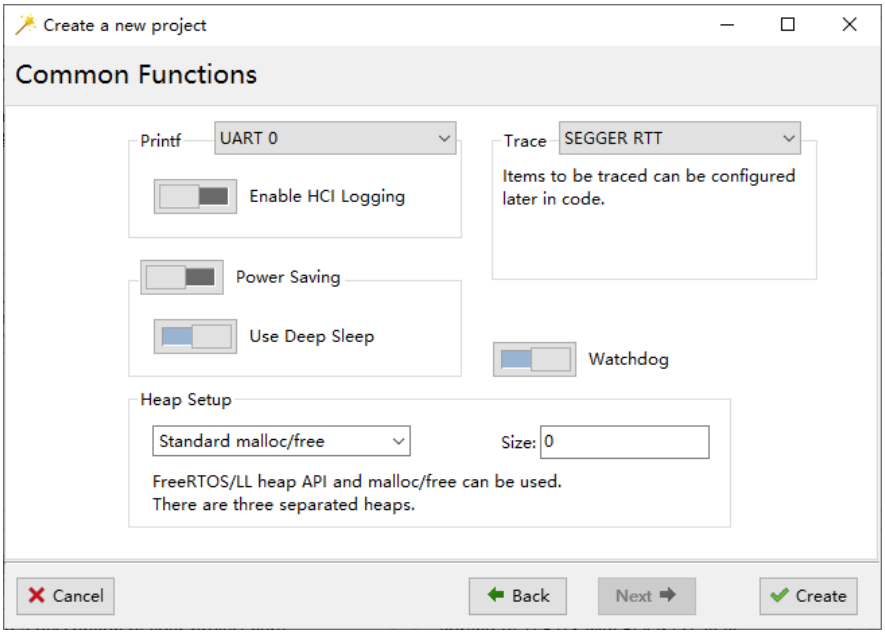

图 **2.9:** Common Functions

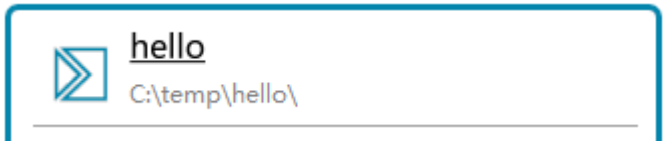

图 **2.10:** ”Hello, 世界” is Ready

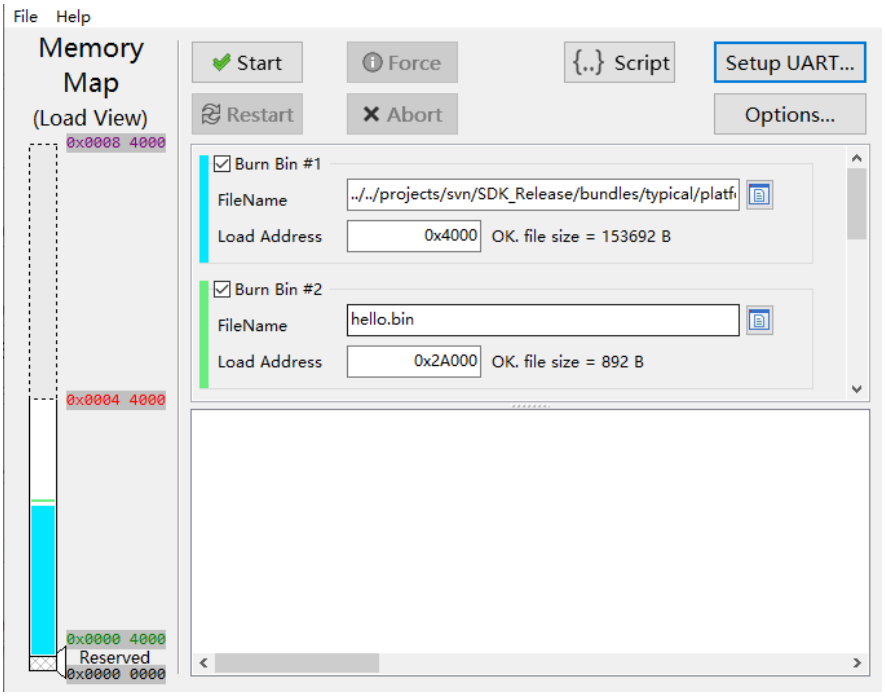

图 **2.11:** Download to Flash

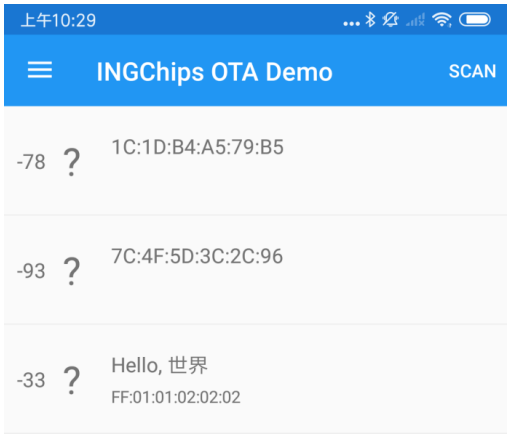

图 **2.12:** Hello, 世界

2.2 iBeacon

在本教程中，让我们创建一个 iBeacon。iBeacon 是由 Apple[1] 开发的协议，并在 2013 年的苹果全球开发者大会上发布。Beacons 是一类低功耗蓝牙 (BLE) 设备，可以将其标识符广播到附近的便携式电子设备。这项技术使智能手机、平板电脑和其他设备在接近 iBeacon 设备时能够执行相应的操作。首先，从 app Store 下载一个 iBeacon 扫描应用程序。在本教程中，我们将使用一个名为 `Locate` 的应用程序。`Locate` 有一个预配置的近距离 UUID 列表，其中包括一个全 0 Null UUID。我们将使用这个 Null UUID[2]。

2.2.1 建立广播数据

iBeacon 广播包中有两个项目。

1. Flags

   值固定为 0x06，即设置了两位，LE General Discoverable Mode & BR/EDR Not Supported。

2. Manufacturer Specific Data

   本项目内容如表 2.1所示

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

[2]Note that UUID is not allowed to be all 0s in final products.

表 2.1: iBeacon 厂商特定数据

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

为了制作一个 iBeacon 设备，我们可以遵循 [Hello World] 示例中的相同步骤，我们只有在个别情况下需要根据规范配置广播包。

在广播数据编辑器中，添加 `0x01 - «Flags»` 和 `0xFF - «Manufacturer Specific Data»`. 单击 `0x01 - «Flags»`，检查 `LE General Discoverable Mode` 和 `BR/EDR Not Supported`。单击 `0xFF - «Manufacturer Specific Data»`，然后点击 `Edit as` 按钮，会有一个菜单弹出并选择 `iBeacon ...` (图2.13) 来打开 iBeacon 厂商特定数据编辑器 (图 2.14).

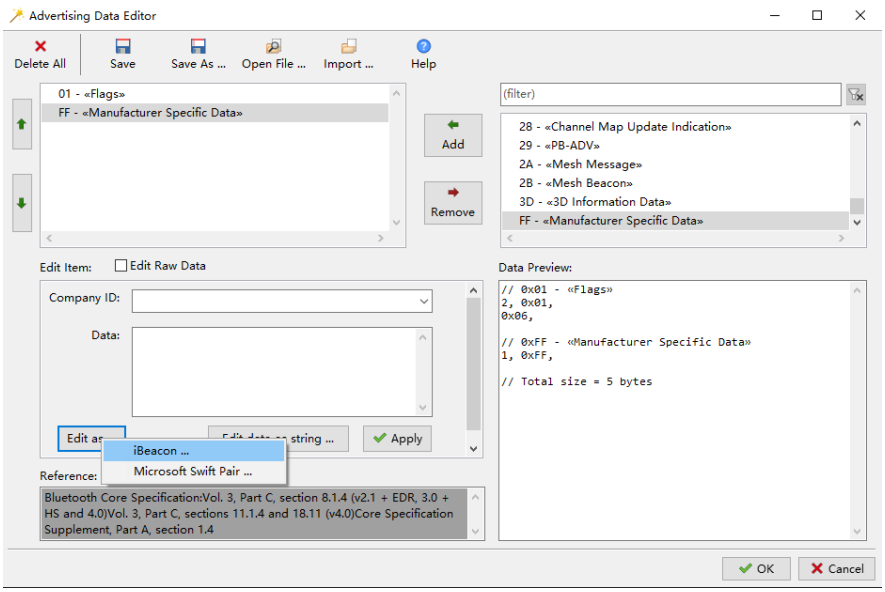

图 2.13: 编辑 iBeacon 广播数据

信号功率可以设置为任何合理值 (如-50dBm)，稍后我们将在 `Locate` 应用程序的帮助下对其进行校准。

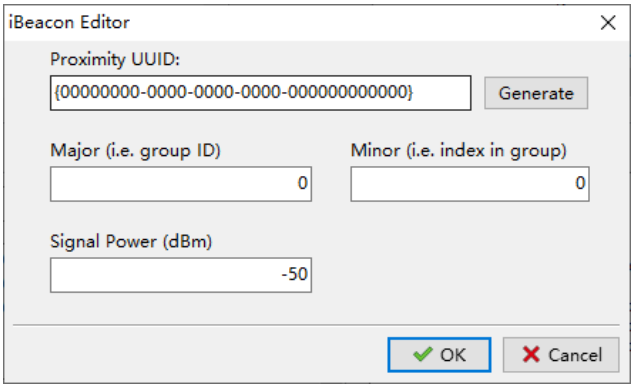

图 2.14: 编辑 iBeacon 厂商特定数据

2.2.2 尝试应用

让我们在 `Choose Project Type` 页面上选择 GNU Arm Embedded Toolchain 作为我们的开发环境，向导会让一切准备就绪 (图 2.15).

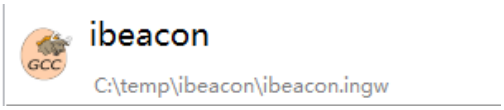

图 2.15: GNU Arm 工具链 iBeacon 已就绪

单击项目以打开控制台，输入 `make`[3]来构建它。回到 `Wizard`，按照相同的步骤将其下载。现在，我们可以在 `Locate` 中找到新创建的 iBeacon 设备。(图 2.16)

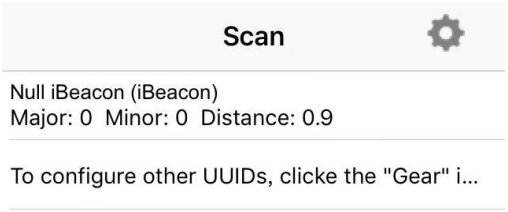

图 2.16: iBeacon 本地 locate APP 界面

点击我们的设备，我们就可以实时校准信号功率或检查距离，如图 2.17所示。

一旦信号功率校准，我们可以在 `Wizard` 中右键单击我们的项目，并选择 `Edit Data -> Advertising` 菜单项，以用我们所熟悉的编辑器来编辑其广播数据。更新广播数据后，重新构建项目，检查距离是否更准确。

[3] `Makefile` follows the syntax of GNU `make`.

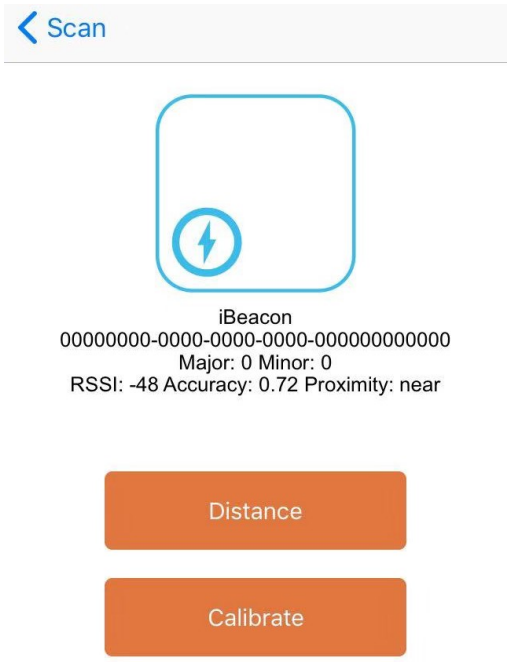

图 2.17: iBeacon 在 locate APP 中的详细信息

根据规范，接近 beacons 必须使用一个不可连接的无定向广播 PDU，使用固定的 100ms 广播间隔。在本教程中，我们不会去修改代码，所以广播参数也不会被修改。为了使这些参数完全符合规范，请参考相应的主机 GAP APIs。

2.3 温度计

在本教程中，我们将制作一个 重要的 BLE 设备，一个温度计。蓝牙 SIG 已经定义了一个称为健康温度计的 GATT 服务[4]。这个 SDK 包含一个名为 `INGdemo` 的参考 APP，它可以安装到 `Android` 或 `iOS` 设备上。使用 `INGdemo`，我们可以查看蓝牙设备的广播数据，如果设备中有健康体温计功能，`INGdemo` 可以连接设备并读取温度。

在本教程中，您将了解如何:

- 广播所支持的服务
- 配置 GATT 配置文件
- 响应 GATT 特性的读请求

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

2.3.1 建立广播数据

同样，我们遵循与 [Hello World] 示例中相同的步骤，并在 `Peripheral Setup` 页面上声明温度计服务并创建 GATT 配置文件。在广播数据中添加以下三项:

1. `Flags`

   值固定为 0x06，即设置了两位，LE General Discoverable Mode & BR/EDR Not Supported。

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

   添加一个如图2.18. 所示的 `0x1809 - Health Thermometer` 服务。

3. `Complete Local Name`

   让我们将设备命名为”AccurateOne”。

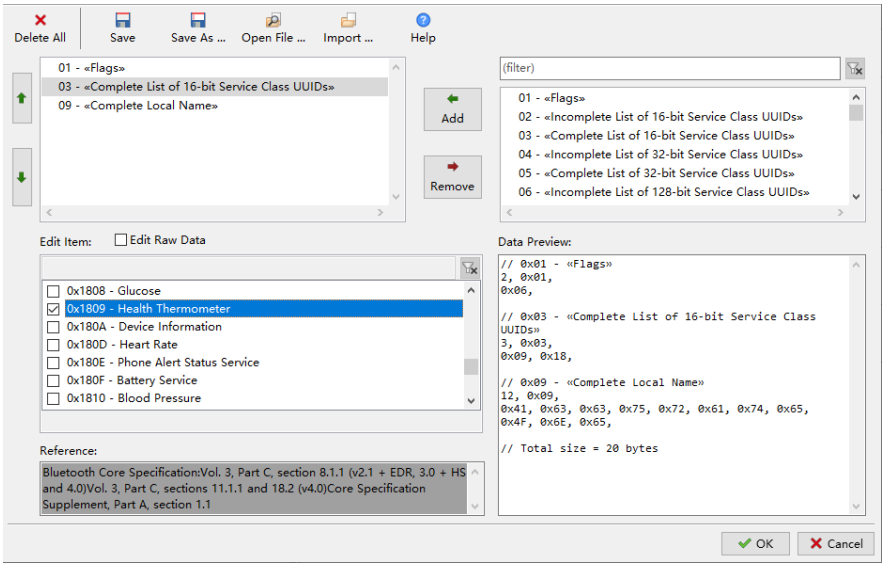

图 2.18: 温度计广播数据

2.3.2 建立 GATT 配置文件

返回 `Peripheral Setup` 页面，单击 `Setup ATT database ...` 打开 GATT 配置编辑器。添加两个服务，General Access (0x1800) 和 Health Thermometer (0x1809)。删除 General Access service 的所有非必选特性。对于 Health Thermometer service 保留两个特性，即温度测量和温度类型，删除其他两个。

接下来，编辑每个特性的值:

1. Device Name of General Access:

   右键单击特征，选择 `Edit String Value ...` 菜单，并设置值为”AccurateOne”。

2. Appearance of General Access:

   右键单击特性，选择 `Help` ，编辑器将在 Bluetooth SIG 网站上打开相应的文档。找到普通温度计的值 (0x0300)，然后单击 `Edit` 按钮，并在数据字段中输入 `0x00, 0x03` 。

3. Temperature Measurement of Health Thermometer

   请查看蓝牙 SIG 网站文档。点击 `Edit` 按钮并在 data 字段中输入 5 个 0(`0, 0, 0, 0, 0`)。这里的第一个字节包含标志，表明之后的度量单位是一个以摄氏度为单位的 `FLOAT` 值。检查 `read` 和 `dynamic` 属性 (图 2.19)。

   `FLOAT` 类型为 IEEE-11073 32 位浮点。最基本的是，它有一个 24 位的尾数和一个 8 位的指数 (最重要的字节)，以 *10* 为基数。

4. Temperature Type of Health Thermometer

   请查看 Bluetooth SIG 网站文档。通过单击 `Edit` 按钮将其设置为任何有效值。

2.3.3 代码编写

项目创建完成后，在 IDE 中打开 `profile.c` ，由 `Wizard` 自动生成温度测量特征处理函数 `att_read_callback` 。

```
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)
    {
    case HANDLE_TEMPERATURE_MEASUREMENT:
        if (buffer)
        {
            // add your code
            return buffer_size;
        }
```

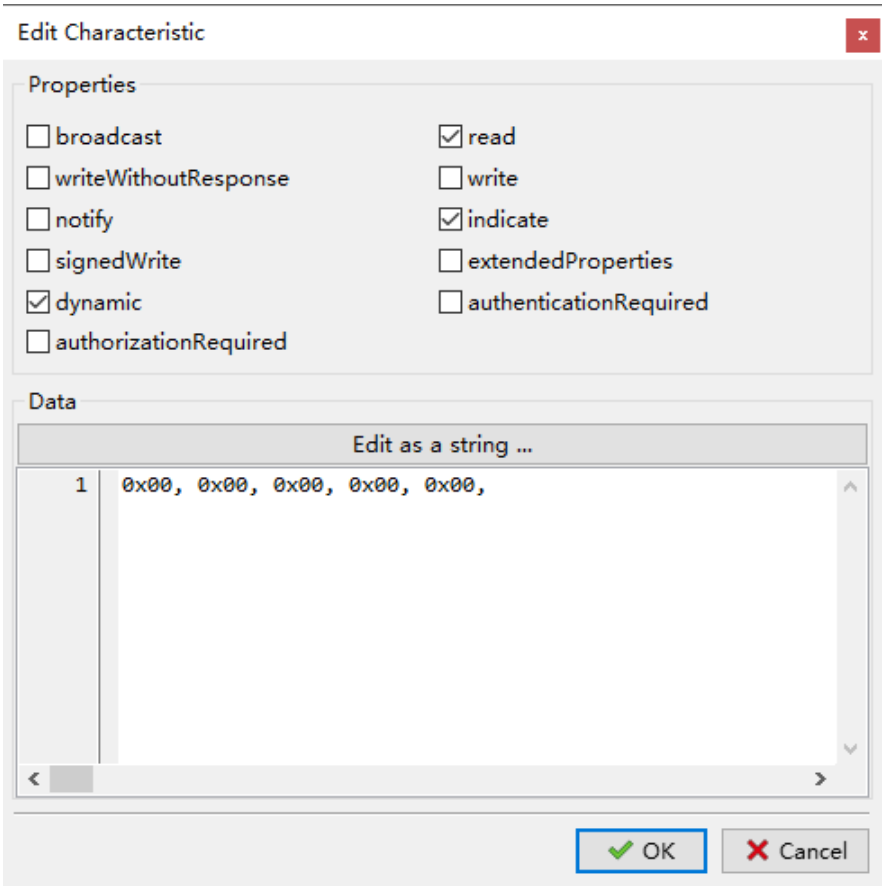

图 2.19: 编辑温度测量

```
        else
            return 1; // TODO: return required buffer size

    default:

        return 0;
    }
}
```

当 APP 读取一个具有 `dynamic` 特点的特性，`att_read_callback` 会被调用两次或更多: 一次用于查询所需的缓冲区大小，一次用于读取数据。如果数据很大，`att_read_callback` 可能会被调用更多次，每次读取由 `offset` 指定的部分数据。

如上所述，定义一种温度测量类型:

```
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};
```

现在，我们可以完善上面的 `case HANDLE_TEMPERATURE_MEASUREMENT` 语句:

```
    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);
```

构建并下载项目，然后在 `INGdemo` app 中连接到”AccurateOne” 设备。检查每次按下 `Refresh` 按钮时温度是否随机变化 (图 2.20).

温度计 (服务器) 可以使用通知或指示过程来通知 (不需要确认) 或指示 (需要确认) 一个特征值，参见 [带通知的温度计]。在本例中，“AccurateOne” 不使用这两个过程，而是被动地发送其测量结果。

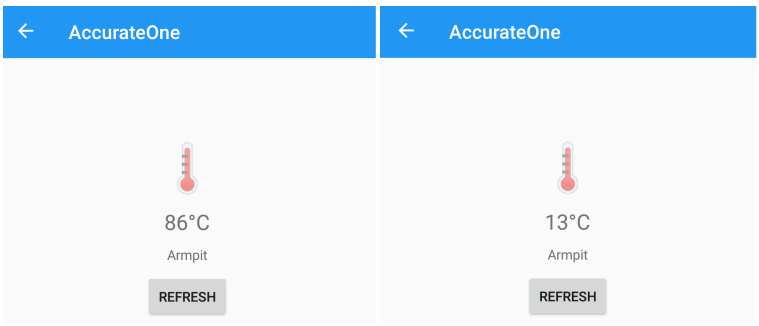

图 2.20: 刷新温度测量

2.3.4 通知

2.4 FOTA 温度计

在本教程中，我们将在我们的温度计中添加无线固件升级（FOTA）功能。此 SDK 提供了一个可开箱即用的 FOTA 设计参考。要使 FOTA 工作，至少需要涉及三个部分，一个设备、一个 APP 和一个 HTTP 服务器。`INGdemo` 应用已经有了，所以在本教程中，我们将重点关注设备和 HTTP 服务器。

2.4.1 FOTA 设备

按照与前面温度计示例相同的步骤创建一个新项目，命名为”ota”。

当编辑广播数据时，我们可以通过单击在编辑器中的 `Open File…`按钮，来导入在前面例子中创建的数据。广播数据存储在 `$(ProjectPath)/data/advertising.adv`. 让我们将设备命名为”clickty Click”。

当编辑 GATT 协议数据库时,我们可以通过单击在编辑器中的打开文件 `Open File…`按钮来导入在前面例子中创建的数据。GATT 协议的数据存储在 `$(ProjectPath)/data/gatt.profile`. 中。在 `Add Service` 按钮的下拉菜单中选择 `INGChips Service`，添加”INGChips FOTA Service”，接下来，编辑该服务的特征值:

1. FOTA Version:

   FOTA Version 确定了我们项目的完整版本号。如 flash downloader 所示，整个项目由两个二进制文件组成，一个来自 SDK 软件包，称为平台二进制文件；另一个来自我们的项目，称为 APP 二进制文件。FOTA 版本包含两个子版本，它们分别对应一个二进制文件。每个子版本包含三个字段:

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

每个软件包都有自己的版本 (同平台二进制文件版本)，使用相同的编号方案，可以在 SDK 页面的 Environment Options 对话框中找到 (使用菜单项 Tools -> Environment Options 打开此对话框)。假设平台版本为 1.0.1，我们想要的 APP 版本为 1.0.0，那么我们将该特征值设为 (图 2.21):

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

2. FOTA Control

这是升级期间的控制点。将其值设置为 0 (即 OTA_STATUS_DISABLED)，这是 FOTA 的初始状态。

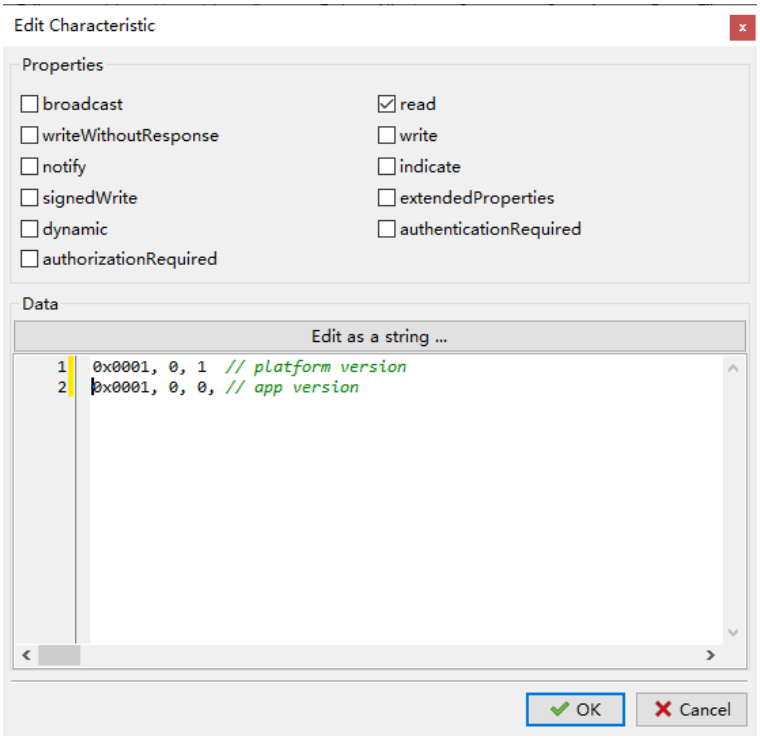

图 2.21: FOTA 版本设置

单击 OK 关闭 GATT 协议编辑器。(注意: 不要点击 Save，除非你想改变已经在编辑器中打开的 $(ProjectPath)/data/gatt.profile 。)

回到项目向导，按下 Next 继续下一页的 Firmare Over-The-Air 。在这页上，我们勾选 FOTA 。注意，与 FOTA 相关的特征句柄是通过检查 GATT 协议自动生成的。然后在项目向导上完成其余步骤。

打开我们新的项目”ota”，从上一个例子复制代码来使我们的温度计在 INGdemo APP 中可以响应 Refresh 。

接下来，让我们制作一个新版本。

2.4.2 创建一个新版本

我们”ota” 的新版本命名为”Barba Trick”，APP 版本号将升级到 2.0.0 。这些数据分别保存在广播数据和协议数据中，右键单击项目并使用编辑器更新它。数据升级以后，使用 Save As ... 将数据保存在相同路径下的另一个文件中。例如，更新广播数据并将其保存到 $(ProjectPath)/data/advertising_2.adv 。并更新 GATT 配置文件到 $(ProjectPath)/data/gatt_2.profile 。

用一个宏 V2 来控制实际用到的广播数据和协议数据：

```
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
};
```

用定义的 V2 宏重新编译项目,将 ota.bin 和 platform.bin (在 SDK_DIR/sdk/bundles/typical 中) 复制到一个空路径下，如 ota_app_v2 。

在 ota_app_v2 创建一个名为 manifest.json 的文件，包含以下数据:

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

这些地址可以在 Environment Options . entry 中找到。地址值固定为 0x4000 ，即 16384 。注意 json 不识别常规的 0xabcd 十六进制文字。INGdemo 可以下载其他 bin 指定的二进制文件到设备。在本例中，我们没有这样的二进制文件，所以这个字段作为空数组保留。

然后为这个更新创建一个 readme 文件，其中包含一些关于本次更新的信息。

现在 FOTA 包已经准备好了。为整个 ota_app_v2 路径制作一个 ota_app_v2.zip 的 ZIP 压缩包。注意，不应该将 ota_app_v2 设置为 ota_app_v2.zip 中的子目录。表 2.2 给出了压缩文件中的文件清单。

表 2.2: FOTA 包文件清单

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

File Name	Notes
ota.bin	App binary

回到 IDE，在没有定义宏 V2 的情况下重新构建项目，然后下载该项目到开发板。

2.4.3 FOTA 服务器

INGdemo APP 需要一个 FOTA 服务器 URL，定义在 class Thermometer.FOTA_SERVERr 。将 ota_app_v2.zip 移动到 HTTP 服务器的文档目录，并创建一个 latest.json 文件，它包含最新版本的信息。内容是:

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

确保这两个文件可以通过 (FOTA_SERVER + latest.json) 和 (FOTA_SERVER + ota_app_v2.zip) 访问.

2.4.4 尝试应用

在 INGdemo 中连接”Clickety Click”，点击 Update (图 2.22). 由于 platform.bin 是最新的，所以只需要升级 app.bin 就可以了，整个升级过程在很短的时间内完成。回到主页，再次扫描并检查我们的新版本是否起作用，有了一个名为”Barba Trick” 的设备。连接到”Barba Trick”，可以看到现在固件是最新的。

本教程给出了一个实现 FOTA 的例子。用户可以自由设计新的 FOTA 解决方案，从版本定义到 FOTA 服务和特性。也可以为 FOTA 开发一个专用的第二 APP。

安全性是必须要考虑的。

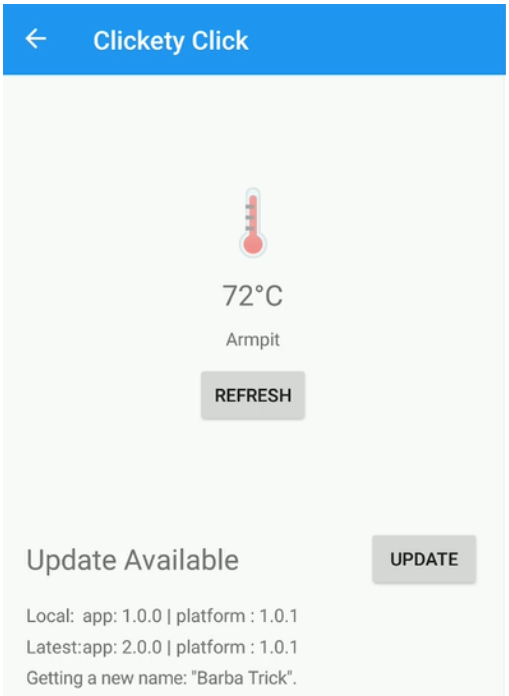

图 2.22: ”Clickety Click” 升级可用

2.5 iBeacon 扫描设备

我们已经知道如何配置 iBeacon 设备。在本教程中，我们将创建一个 iBeacon 扫描器。

扫描器在蓝牙微距网络中起着核心作用。和之前一样，我们在 `Wizard` 中创建一个名为”iscanner” 的新项目 (图 2.23). 在 `Role of Your Device` 页面，选择 *Central*。中心设备会一直扫描其他设备然后这些设备执行其相应的动作，我们的新项目向导自动添加代码开始扫描。

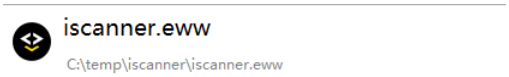

图 2.23: 创建 IAR Embedded Workbench 的“iscanner”

在 IDE 中打开这个新项目，并找到函数 `user_packet_handler` 。我们可以看到有一个名为 `HCI_SUBEVENT_LE_EXTENDED_ADVERTISING_REPORT` 的事件:

```
    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;
```

每次接收到这个事件时，我们可以检查广播报告是否包含 0xFF - «Manufacturer Specific Data»，以及它是否是一个 iBeacon 包。有了制作 iBeacon 设备的知识，就可以直接用 C 语言定义一个 iBeacon 数据包的类型。

```
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 是一个扩展关键字，用于指定数据类型使用第一种数据对齐方式。幸运的是，*ARM* 和 *IAR* 编译器都支持它。或者可以使用 #pragma pack 指令:

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

在继续之前，让我们创建一个将 UUID 转换为字符串的辅助函数。

```
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 估算距离

接收到的信号强度指示 (RSSI) 与广播数据一起播报。一般来说，点源辐射出的电磁波强度与信号源距离的平方成反比。著名的自由空间损失方程为:

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

其中 d 的单位是 km, f 的单位是 MHz, $Loss$ 的单位是 dB。通过比较 RSSI 和 1 米距离下的测量功率 (`ref_power`)，我们可以利用自由空间损失方程大致估计出扫描仪和信标之间的距离:

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

现在，我们用不到 20 行代码就可以实现一个功能完整的 iBeancon 扫描器:

```
    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;
```

使用 Locate 应用程序发送 iBeacon 信号，并查看我们的设备是否能找到它 (图 2.24). 最后，由于 RSSI 值是波动的，可以在 RSSI 上增加一个低通滤波器使估算值更加稳定。

```
[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
```

图 2.24: iBeacon 扫描结果

需要注意该 APP 的二进制文件的大小会急剧增加。这主要是因为 Cortex-M3 没有硬件浮点单元，浮点操作都是由库函数执行的。使用浮点运算前请 仔细考虑一下。

2.5.2 并发广播 & 扫描

作为一个练习，我们可以合并这个 iBeacon 项目，并检查我们的设备是否可以在发送 iBeacon 信号的同时继续扫描其他 iBeacon 设备。

蓝牙无线电采用 TDD (Time Division Duplex) 拓扑结构，该结构要求同一时刻的数据发送在一个方向上进行，数据接收在另一个方向上进行，设备不能接收到自己的 iBeacon 信号。

2.6 通知 & 指示

服务器可以使用通知或指示过程来通知 (不需要确认) 或指示 (需要确认) 一个特征值。现在，让我们将通知和指示功能添加到我们在之前教程中创建的温度计中。

我们分别使用 `att_server_notify` 和 `att_server_indicate` 来通知或指示一个特征值。这些 `API` 必须在蓝牙协议栈 (`Host`) 任务中调用。

主动产生的通知和指示可以由定时器或中断触发，例如蓝牙任务栈之外的源。SDK 提供了基于 RTOS 消息的任务间通信机制以便调用这些蓝牙协议栈 API。

2.6.1 任务间通信

可使用 `btstack_push_user_msg` 发送消息到蓝牙协议栈:

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

这个消息将被传递到你的 `user_packet_handler` 下的事件 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;
    // ......
    }
}
```

在这里，我们将用户消息的处理传递给另一个叫 user_msg_handler 的函数。注意 user_msg_handler 是在蓝牙协议栈任务的上下文中运行的，现在我们可以调用那些蓝牙栈 APIs 了。

事件 BTSTACK_EVENT_USER_MSG 被广播到所有 HCI 事件回调函数。

2.6.2 定时器

现在让我们让温度计"AccurateOne" 每秒钟更新一次它的值。首先，在初始化时创建一个定时器，例如在 app_main 或 setup_profile 中。

```
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);
        // ...
}
```

定时器回调函数可以被定义为：

```
#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);
}
```

当我们在 HCI_EVENT_LE_META 中得到 HCI_SUBEVENT_LE_ENHANCED_CONNECTION_COMPLETE 时，定时器开始计时，并在我们得到 HCI_EVENT_DISCONNECTION_COMPLETE 时定时器停止。

这里的 temperture_notify_enable 和 temperture_indicate_enable 是两个初始化为 0s 的标志，并在 att_write_callback 中设置为 1：

```
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;
    // ...
    }
}
```

在这里，我们将 connection_handle 存储到一个全局变量 handle_send ，并在之后会使用该变量。最后一段代码是在 user_msg_handler 中处理消息 USER_MSG_ID_REQUEST_SEND：

```
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;
    }
}
```

并在 ATT_EVENT_CAN_SEND_NOW 中报告温度值：

```
...
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;
...
```

尝试重新构建并下载项目，并检查 INGdemo 中显示的温度值是否每秒变化一次。

有一个完整功能的温度计示例，即 thermoota，它支持 FOTA，通知和指示。

2.7 吞吐量

BLE 5.0 介绍了一种新的采样率为 2M 的非编码 PHY。

2.7.1 理论峰值吞吐量

数据物理通道 PDU 的最大有效载荷长度为 251 字节。采用 2M PHY，传输时间为 1048 µs。一个空的数据物理通道 PDU 的传输时间为 44 µs。

为了实现一个方向上的最大吞吐量，该方向上所有 pdu 的长度应该为 251 字节，而另一个方向上所有 pdu 的长度应该为空。所以，251 个字节总的传输时间为：

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

因此，链路层提供的理论峰值吞吐量为：

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

对于一个运行在 GATT 之上的应用程序，I2CAP 和 ATT 都有它们自己的开销。通常，GATT 的最大有效负载为 (251 - 7 =)244 字节。因此，GATT 可以提供的理论上的峰值吞吐量为：

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

2.7.2 测试吞吐量

在 SDK 中有一对用于吞吐量测试的示例（图 2.25）.

2.7.2.1 对 INGdemo 进行测试

下载 peripheral_throughput. 使用 INGdemo 连接到 ING Tpt , 打开吞吐量测试页面。在这个页面上，我们可以测试从主到从、从到主或同时在两个方向上的吞吐量。

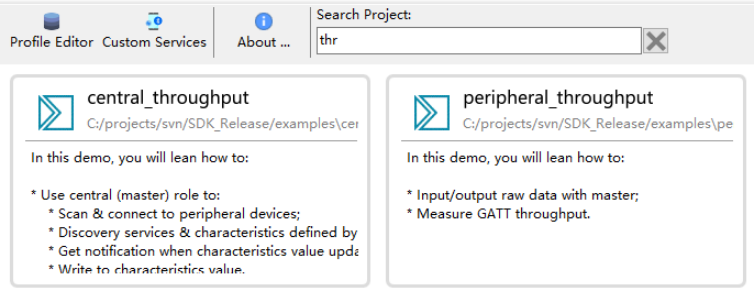

图 2.25: 吞吐量测试示例

图中（图 2.26）显示，使用支持 2M PHY 的普通 Android 手机，我们可以实现 1M+ bps 的空中传输吞吐量。

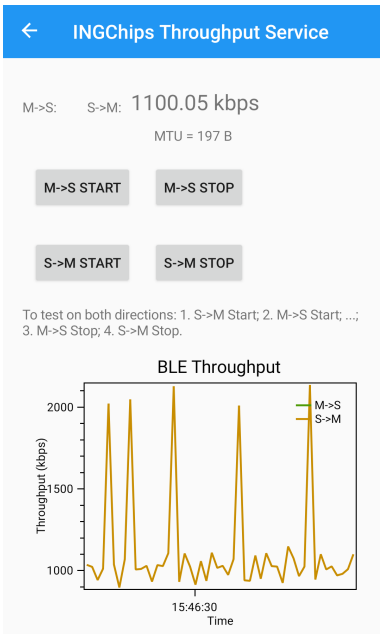

图 2.26: Android 手机上的吞吐量

2.7.2.2 对我们的 APP 进行测试

示例 `central_throughput` 演示了 BLE 中心设备的一般工作过程:

1. 扫描并连接到在其广播中声明了吞吐量服务的设备;
2. 发现吞吐量服务;
3. 发现服务的特性;
4. 发现特性描述。

INGChips Throughput Service 有两个特点。

- 通用输出

  通过这一特性，外围设备向中心设备发送数据。

  这个特性有一个 Client Characteristic Configuration 描述符。

- 通用输入

  通过这个特性，中心设备向外围设备发送数据。

下载 central_throughput 到另一块板。这个应用程序有一个 UART 命令行接口给到主机。连接到主机，输入"?"以查看所支持的命令。这个 APP 自动连接到 peripheral_throughput 。输入命令 start s->m 或 start m->s 开始测试从外设到中心设备的吞吐量，或从中心设备到外设的。

```
?
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
```

图 2.27: 指令接口

图中（图 2.28）显示，使用两块板，我们在空中实现了 1.2M+ bps 的稳定数据传输。

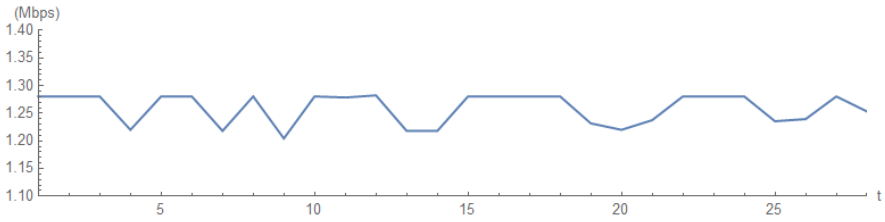

图 2.28: 板间吞吐量

这个吞吐量是在空中测试的，比理论峰值略低，但真实性更高。

2.8 双角色 & BLE 网关

在本教程中，我们将创建一个 BLE 网关，它从几个外设收集数据并将数据报告给一个中心设备。在收集数据时，这个网关是一个中心设备，而报告数据时，它是一个外设，也就是说，我们的 APP 能够扮演双角色。

具体而言，我们的网关只支持从温度计收集数据。我们称之为 `smart_meter` 。

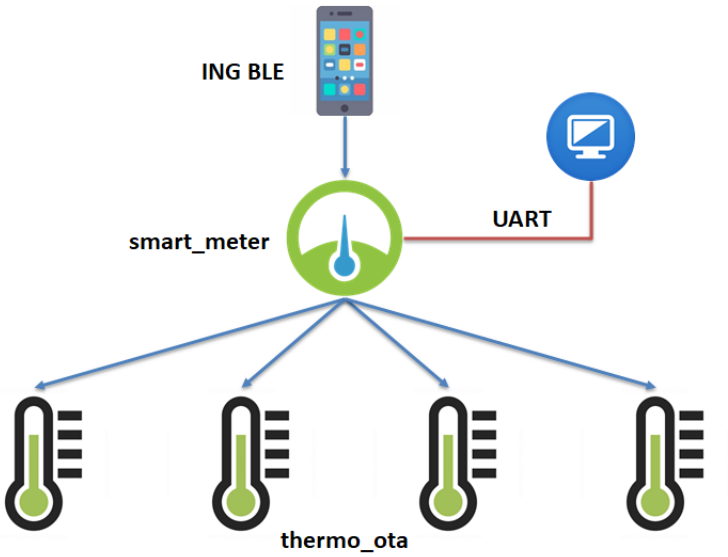

图 2.29: Smart Meter 架构

`smart_meter` 使用一个通用型基于字符串的输出服务将数据报告到中心设备，比如运行在智能手机上的 `INGdemo` 。它还拥有可以连接到主机的 UART 控制接口。

检查示例 `peripheral_console` 以了解如何进行字符串输入和输出。

我们同样提供了完整功能的 `smart_meter` app 作为示例。在创建您自己的示例时，请参考此示例。

现在，让我们创建这个 BLE 网关。

2.8.1 用 `Wizard` 创建一个外设 APP

使用 GUI 编辑器编辑广播数据，命名我们的应用程序为”ING Smart Meter”。

使用 GUI 编辑器编辑 GATT 配置文件。将 `INGChips Console Service` 添加到 GATT 配置文件（图 2.30）.

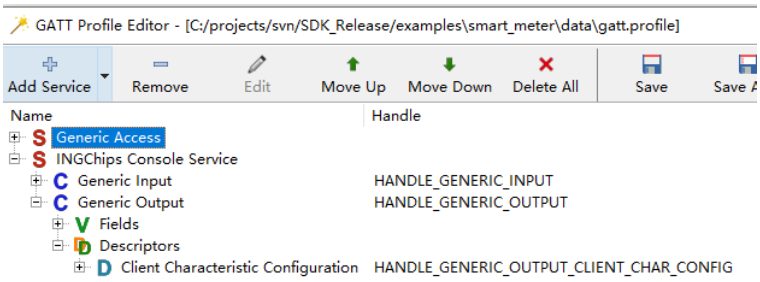

图 **2.30:** Smart Meter GATT 配置文件

2.8.2 定义温度计数据

温度计由它的设备地址和 id 来识别。每个温度计使用由 `conn_handle` 来显示自己的连接状态。

```
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;
```

设定了 4 个温度计。

2.8.3 扫描温度计

调用两个 GAP API 接口开始扫描。一旦找到一个设备，会检查它的设备地址是否是温度计。如果是，停止扫描并调用 `gap_ext_create_connection` 进行连接。连接建立后，如果有温度计未连接，则重新开始扫描。

2.8.4 发现服务

连接建立后，调用 gatt_client API 接口来发现它的服务。这些 API 接口遵循类似 Android 和 iOS 的逻辑。

2.8.5 数据处理

预定温度计的 Temperature Measurement 特性。当接收到一个新的测量值时，将该值转换为一个字符串并将其报告给主机。如果我们的应用程序已经连接到一个中心设备，通过 GATT 特性将该信息转发给它。

2.8.6 鲁棒性

为了让我们的应用更稳健:

- 如果与温度计断开连接，则开始扫描;
- 如果与中心设备断开连接，则开始广播。

2.8.7 准备温度计

我们可以将示例 thermo_ota 用做温度计。但是我们需要为每一个温度计配置不同的地址。

我们可以写一个简单的脚本为下载程序自动生成这些地址:

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

有关下载脚本的更多信息，请参见Scripting & Mass Production.

2.8.8 测试

在主机上输入 `start` 命令来启动我们的应用程序 (开始扫描并广播)。使用 `INGdemo` 连接到名为”ING Smart Meter” 的设备，检查温度测量结果。

关闭和打开一个或多个温度计，我们的应用程序应该能够重新连接到它们。

2.9 从示例开始

`Wizard` 主界面展示了 SDK 附带的所有示例，开发者直接可以直接修改例子，查看效果。当重新安装 SDK 时，这些修改会被覆盖。如果需要基于示例开发项目，开发者可以在快捷菜单选择 “Copy this Example …” 将示例完整复制到其它位置。

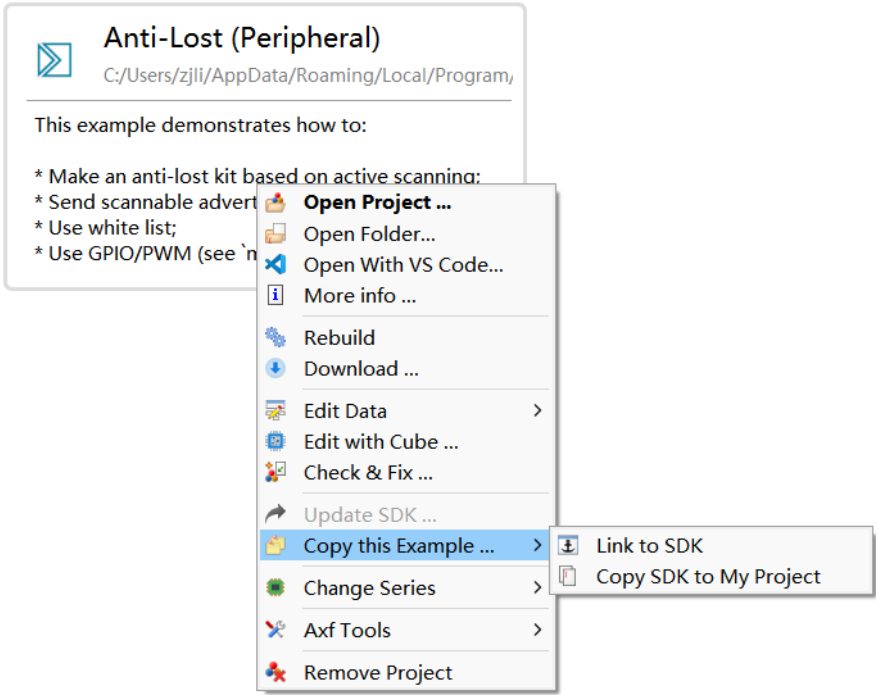

图 2.31: 复制 SDK 示例

第三章　核心工具

SDK 核心工具在整个 BLE 开发过程中扮演重要角色。

3.1　向导

向导（`Wizard`）是整个开发周期中的推荐入口。我们用这个工具创建、打开项目，编辑项目数据，迁移项目版本等。

1. 创建项目

   `Wizard` 的新项目向导辅助我们建立新项目。在向导里，可以选择喜欢的 IDE，蓝牙角色，编辑广播、配置数据，使能 FOTA、日志，等等。

   项目创建后，向导同时会生成以下只供 `Wizard` 使用的文件——不要删除这些文件，否则 `Wizard` 将无法正常工作：

   - `$(ProjectName).ingw`

     这个文件的文件名与与项目相同，但是扩展名为 `.ingw`。它包含关于项目和 SDK 的关键信息。如果没有这些信息，就无法进行版本迁移。

2. 广播数据编辑器（Advertising Data Editor）

   这个编辑器帮助我们生成广播数据。它也可用从主菜单 `Tools -> Advertising Data Editor ...` 打开。

3. GATT 配置编辑器（GATT Profile Editor 或者 GATT/ATT Database Editor)

   这个编辑器帮助我们生成 GATT 配置数据。它也可从主菜单 `Tools -> Profile Database Editor ...` 打开。

   这个编辑器支持三种类型的服务：SIG 定义的服务，*INGChips* 定义的服务，以及用户自定义的服务。用户自定义服务需要事先定义（见后文）然后才能添加。

4. 管理自定义服务

   这个编辑器可以通过主菜单 `Tools -> Manage Custom GATT Services ...` 打开。我们可以添加、删除、编辑自定义服务。

   自定义服务和特征的名字都以安装时指定的公司名称作为前辍。公司名称可以通过 `Environment Options` 修改。

5. 迁移

   当更新了 SDK 版本后，platform 占用的 Flash 和 RAM 可能发生变化，项目的设置也就需要相应更新。这个过程可以通过在项目上点击右键，然后选择 `Check & Fix Settings ...` 自动完成。

在使用迁移功能之前 务必先对项目做备份：将项目提交到版本管理系统或者直接备份所有文件。

3.2 下载器

3.2.1 介绍

下载器可以将最多 6 个映像文件（二进制文件）通过 UART 烧录到芯片。它需要芯片内的引导程序（Bootloader）的配合。Bootloader 可以通过以下途径进入 Flash 下载状态：

- 使能 Boot 引脚[1]（这是最常用的情况），或者
- 将 Flash 里的入口地址设为非法值（仅适用于 ING918xx）。

当芯片上电时，Bootloader 会检查上述各个条件，只要有一条满足，就会进入下载状态，并发送握手信息。当 ING916 进入 Flash 下载状态时，Bootloader 还将检查 GPIO15：如果其电平为高，USB 端口也将被启用。

用户可以下载任意文件，尽管典型情况下，这些文件都是由 IDE 工具生成。程序映像文件的加载地址必须与 Flash 烧写单位对齐：

[1] ING918 具备专门的 Boot 引脚。ING916 复用 GPIO0。

- 对于 ING918xx，烧写单位为页

  映像文件的加载地址必须与页边界对齐。每页 8192 (0x2000) 字节。Flash 起始地址为 0x4000，所以加载地址应该满足 0x4000 + X * 0x2000, 这里 X 为自然数。

- 对于 ING916xx，烧写单位为扇区

  映像文件的加载地址必须与扇区边界对齐。每个扇区 4096 (0x1000) 字节。Flash 起始地址为 0x02000000，所以加载地址应该满足 0x02000000 + X * 0x1000, 这里 X 为自然数。

如果加载地址有误，下载器会报告错误信息。说明：当从 Wizard 里启动下载器时，地址应该都已正确配置。

点击 Setup UART ... 或者 Setup Port ... 配置通信端口参数（图 3.1）。用户需要将 Port 设置成 Windows 设备管理器显示的串口号，例如，如果使用 “COM9” 就将 Port 设置为 COM9，或者直接写 9。对于支持通过 USB 下载的芯片，可以将 Port 设置为 USB 来通过 USB 下载。波特率可以设置成比默认值 115200 更大的数值，比如 460800, 921600 等，来获得更高的下载速率。工具支持的最高波特率为 921600。受限于内部 Flash 的自身特性，将波特率设为比 512000 更高的数值并不会带来下载速率的进一步提升。其它参数保持不变。

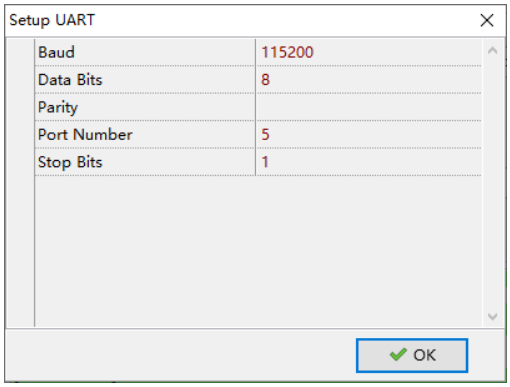

图 3.1: Configurate UART

整个下载过程包含多个步骤：下载、验证、设置启动地址、启动程序。这些步骤可通过点击 Options 进行设置（图 3.2）。对于 ING918，当使用 platform 时，启动地址（入口地址）0x4000 等于 platform 的加载地址。对于 ING916，当启动地址不属于 RAM 范围时，该地址不起作用。

如果启用了 “Verify Download” 功能，数据下载之后会从 Flash 回读并与原始数据对比以验证下载是否正确。由于下载时数据已经过 CRC 验证，一般情况下，“Verify Download” 并不需要启用。如果在特定位置持续出现下载错误，那么我们可用启用验证功能以便检查 Flash 是否出现异常。当验证失败时，下载器会把回读的数据保存到文件里以便进一步分析。

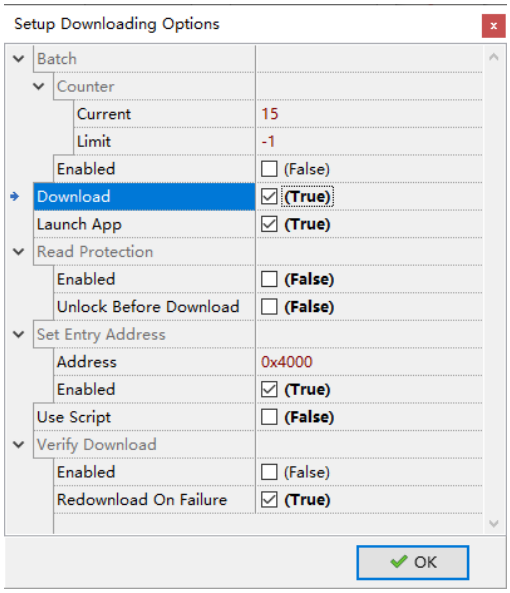

图 3.2: Downloader Options

当启用了批量（“Batch”）模式时，下载器会持续等待 Bootloader 的握手信号，一旦收到握手信号就会开始下载；下载完成后，下载器再次进入等待状态。如果未启用 “Batch” 模式，下载完成后下载器就不会再次等待握手信号。

点击 `Start` 开始下载，更确切地说是开始等待握手信号。Bootloader 只发送一次握手信号，如果芯片已经上电，Bootloader 已经发送过了握手信号，这种情况下，我们可以点击 `Force` 跳过等待过程，直接开始下载。

3.2.2 脚本与量产

下载器支持脚本，可以应用于量产。在下载脚本里可以捕捉到两个事件：

- `OnStartRun`

  这个事件处理函数在每轮下载开始时都会调用；

- `OnStartBin`

  这个事件处理函数在每个二进制文件开始下载时都会调用。在这个函数里，可以即时修改待下载的数据。

当启用了 “Batch” 时，下载器会维护一个计数器，每次下载完成，该计数器加 1。这个计数器即图 3.2 中的 `Counter.Current`。还有一个名为 `Counter.Limit` 的变量：在 “Batch” 模式下，当要开始新的一轮下载时，会检查 `Counter.Current`，如果**超过**了限制，“Batch” 模式自动停止。

例如 Counter.Current 和 Counter.Limit 分别为 10 和 13，那么 “Batch” 模式将一共执行 4 轮下载，每一轮的 Counter.Current 等于 10、11、12 和 13。“Batch” 停止后，Counter.Current 等于 14.

下载器使用的脚本语言为 *RemObjects Pascal Script*[2]，与 C 比较类似，易于开发。下面是一个简单但是实用的例子，把下载计数器写入二进制文件的固定位置：

```
// 我们可以使用常数
const
  BD_ADDR_ADDR = $1;

// BatchCounter 就是 Counter.Current
procedure OnStartRun(const BatchCounter: Integer; var Abort: Boolean);
begin
  // 使用 Print 输出日志、调试信息
  Print('OnStartRun %d', [BatchCounter]);
  // 把 Abort 赋值为 True 可以中止下载
  // Abort := True;
end;

procedure OnStartBin(const BatchCounter, BinIndex: Integer;
                     var Data: TBytes; var Abort: Boolean);
begin
  // 注意 BinIndex 从 1 开始计数（而不是 0），与图形界面一致
  if BinIndex <> 2 then Exit;
  // 我们在数据下载到 Flash 之前修改它
  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;
```

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

3.2.3 Flash 读保护（适用于 ING918xx）

为防止对 Flash 内的数据和程序的非法访问，918xx 设计了一种读保护机制。读保护一旦启用，JTAG/SWD 就下载器都无法访问 Flash。要使芯片重新启用 JTAG/SWD 调试、下载功能，需要一个经过 解锁过程。解锁时，Flash 里的数据被全部擦除。

当 app 已准备就绪，并且认为需要保护 Flash 里的数据和程序不被非法访问，可如图 3.2 所示启用读保护（“Read Protection”）功能。启用 `Unlock Before Download` 选项可以为已经启用了读保护的设备重新下载程序。由于 解锁时，Flash 里的数据被全部擦除，不要忘记重新下载 platform。

所有的配置参数都保存在一个 *ini* 文件内。

3.2.4 Python 版本

SDK 同时提供了一个 Python 版本的下载器（*icsdw.py*）。它是开源的，可以与其它工具集成。

本工具是用 Python 3 开发，使用 `PySerial`[3] 包访问串口。这个包可以通过运行命令 “`pip install pyserial`” 安装。

Python 版下载器与图形界面版（GUI）下载器使用相同的 *ini* 文件，只有一处例外：脚本。GUI 版工具将 *RemObjects Pascal* 保存在 “*options*” 里 “*script*” 域，而 Python 版在这个域里保存的是一个用户模块的路径。这个路径可以是绝对路径，也可以是相对路径（相对于 *ini* 文件）。

在用户模块里，与 GUI 版本类似，可以定义两个事件回调函数，`on_start_run` 和 `on_start_bin`。下面的例子演示了如何将批量计数器写入第 2 号二进制文件的固定地址：

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

# 返回 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)
```

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

```
    addr = batch_counter.to_bytes(4, 'little')
    ba[1:5] = addr
    return False, bytes(ba)
```

3.3 Trace 工具

Trace 工具（Tracer）是调试与跟踪 里提到的 Trace 数据的可视化工具.

为了限制屏幕上的数据，Tracer 将数据划分为若干帧。每一帧的长度为 5 秒。为了使呈现的数据更具有连续性，当选择了一帧时，除了当前帧，其前后各一帧也会显示出来。

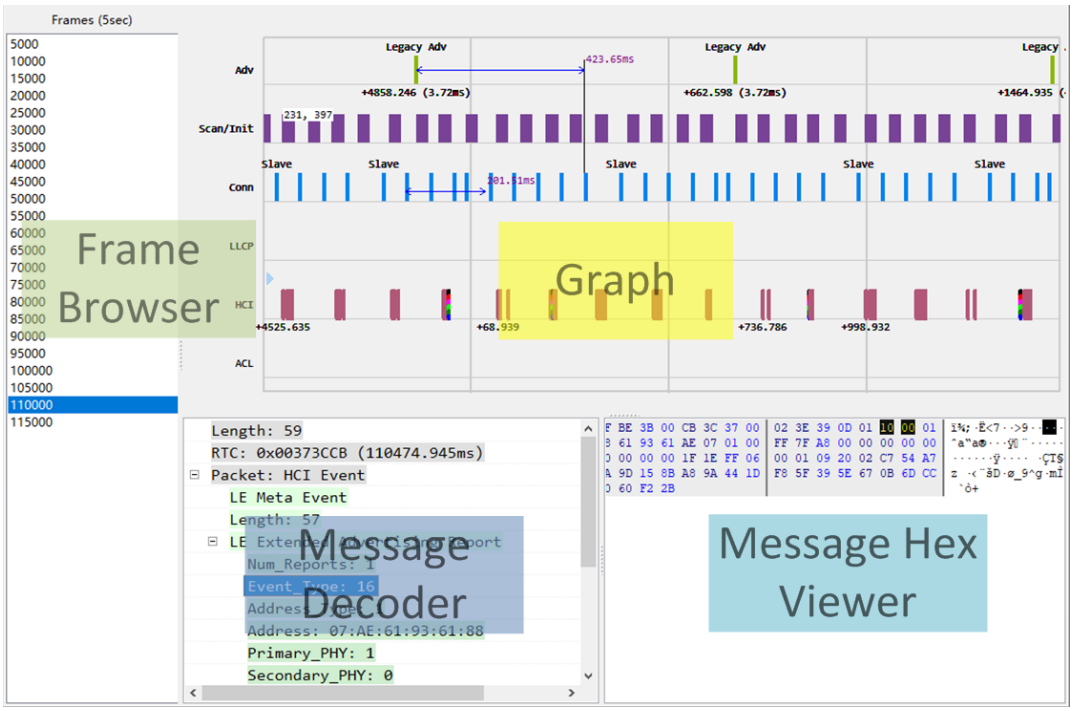

图 3.3: Tracer 主界面

Graph 将所有数据图形化显示。点击 **Graph** 里的一项，会在 **Message Decoder** 和 **Message Hex Viewer** 显示详细的解码信息。**Graph** 支持一些 CAD 操作，如缩放、平移、测量等。选择菜单 Help -> About 查看详细信息。（图 3.3）

为辅助分析 app 及高层协议栈问题，ingTrace 可为每一个连接生成 MSC （消息序列图）。**Graph** 窗口侧重事件之间的定时关系，而 MSC 则侧重流程，更便于分析协议层面的问题。点击 MSC 里的 [+] 可以解码消息（图 3.4）。

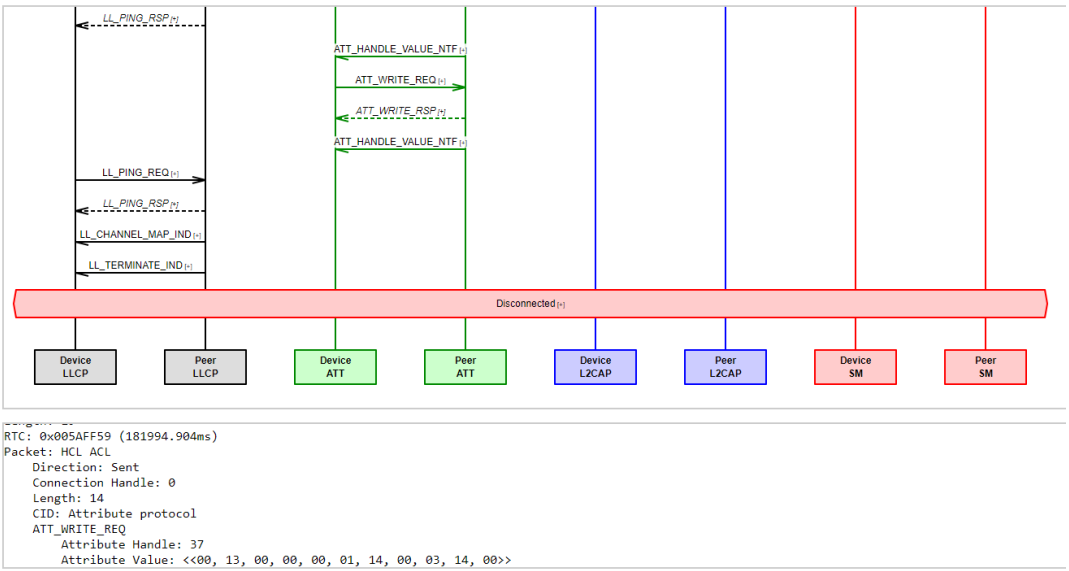

图 3.4: MSC Generated by Tracer

3.4 Axf Tool

`Axf Tool` 是一个用于分析可执行程序和内存转储的命令行工具，可从 `Wizard` 的项目快捷菜单执行。包含多种功能：

- **stack-usage**: 静态分析栈的使用情况，并报告栈空间使用最多前 N 条函数调用链；
- **bt-api-thread-safety**: 分析对蓝牙 API 的调用，检查是否违反了单线程约定；
- **call-stack**: 尝试从内存转储中恢复函数调用栈；
- **history**: 给出关于 BLE 活动简史；
- **check-heap**: 尝试检查堆的状态；
- **check-task**: 在内存转储的帮助下动态检查 FreeRTOS 里的各个任务的情况。

通过 `axf_tool.exe help {function}` 命令获得每种功能的详细信息。

第四章　深入 SDK

本章介绍关于高效使用 SDK 的一些关键问题。

4.1　内存管理

有以下三种主要的内存管理方法：

1. 全局的静态分配
2. 在栈上的动态分配和释放
3. 在堆上的动态分配和释放

RAM 由 platform 和 app 共享使用。`Wizard` 创建的新项目 RAM 的位置、大小会自动得以配置。不建议开发者修改此设置。

4.1.1　全局分配

对于在 app 整个生命周期都存在的变量，这是推荐的分配方式。全局变量地址固定，用调试器可以方便地检查其中的数据。

4.1.2　使用栈

对于仅在有限范围内——比如一个函数内部——存在的变量，我们可以在栈上分配它们。

必须注意：栈的大小是有限的，如果分配的空间过多，栈会溢出。

1. app_main 函数及中断服务程序与 platform 的 main 使用同一个全局栈。

   对于 RTOS 软件包，这个栈由 platform 定义，其大小为 1024 字节。如果大小不合适，可利用 platform_install_isr_stack 做替换。

   对于 “NoOS” 软件包，这个栈由 app 定义。

2. 蓝牙协议栈的各种回调函数与协议栈使用同一个栈，其大小为 1024 字节，一般情况下，进入回调函数时大致还有一半空间空闲。

3. 开发者可以创建新的 RTOS 任务。这时，栈的大小需要仔细核对。

用工具检查函数所需要的最大栈空间（栈的深度）。

4.1.3 使用堆

总体而言，不太建议在嵌入式应用里使用堆管理内存。这种方法至少存在以下不足：

- 空间开销

  对于每块空间，为了存储额外的信息，若干字节被浪费。

- 时间开销

  分配、释放内存块时需要消耗时间。

- 碎片

基于以上考虑，Wizard 创建项目时在默认情况下，堆大小设置为 0，完全禁用了 malloc 和 free。如果确实需要使用堆，可以在创建项目时，把堆大小修改为合适的数值。在使用 malloc 和 free 之前，请查看以下建议：

- 使用全局变量
- 使用内存池[1]

  这可能是一种适用于多数场景的方案。

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

- 使用 RTOS 的堆内存接口，如 `pvPortMalloc` 和 `pvPortFree`

  注意，这个堆由 platform 和 `FreeRTOS` 共用，留给 app 的空间可能不多。标准的 `malloc` & `free` 在 `Wizard` 的堆设置里可配置为由 `pvPortMalloc` & `pvPortFree` 实现，此时，libc 里的堆分配器不会链接到程序里，`malloc` & `free` 完全依赖 `pvPortMalloc` & `pvPortFree` 实现。

4.2 多任务

建议先阅读《Mastering the FreeRTOS™ Real Time Kernel》。几点建议：

1. 不要在中断服务程序里做过多处理，应该尽快延续到任务里，
2. 蓝牙协议栈的回调函数在协议栈任务的上下文里运行，所以也不要做过多处理，
3. 调用蓝牙协议栈 API （除了 `btstack_push_user_msg` 本身）之前先利用消息传递函数 `btstack_push_user_msg` 或其它特殊函数[2] 与协议栈同步（参见任务间通信）

4.3 中断管理

Apps 通过 platform API `platform_set_irq_callback` 创建经典的中断服务程序。

Apps 可以使用下列 API 修改中断配置或者状态：

- `NVIC_SetPriority`

  注意,对于 RTOS 软件包,中断优先级最高为 `configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY + 2`，也就是说，优先级参数必须**大于**或者**等于**该值，以表示 更低或者 相等的优先级。

- `NVIC_EnableIRQ`
- `NVIC_DisableIRQ`
- `NVIC_ClearPendingIRQ`
- 等等……

[2]请参考《低功耗蓝牙开发者手册》“线程安全性”一节。

4.4 功耗管理

在大多数情况下，platform 负责自动管理系统的省电功能，尽量降低功能。只有一个例外，深睡眠。

在深睡眠模式下，芯片内除低功耗管理及 RTC 时钟以外的所有部件都会掉电或者进入其它最省电状态。由于 platform 无法知晓 app 用到了哪些外设、如何配置这些外设，所以，app 需要参与从深睡眠退出时的唤醒流程。Platform 会询问 app 当前是否允许进入深睡眠，如果不允许，则进入其它相对不激进的省电模式。

为了使用深睡眠，需要定义两个回调函数，参见`platform_set_evt_callback`。为了方便开发和调试，提供`platform_config` API 控制是否开启省电功能。

除了上面的自动省电机制，app 还可以主动将整个芯片系统关闭一段时间然后重启。关闭状态下芯片功耗降至最低，在关闭状态下，可以维持一段内存里的数据不丢失，代价是需要增加一点功耗。参见`platform_shutdown`。如果仅有很少量的数据需要保持：

如果只需要保存少量的几个比特,SDK 提供了一对 API 可供使用`platform_write_persistent_reg`和`platform_read_persistent_reg`。

4.5 CMSIS API

SDK 尝试对一些 CMSIS API 做了封装以便开发。当直接使用 CMSIS API 时务必小心，因为可能会影响系统的运行。

以下操作严格禁止：

1. 修改中断向量表相关的寄存器
2. 修改中断的配置

4.6 调试与跟踪

除了在线调试外，SDK 提供以下两种辅助调试的方法：

1. `printf`

   `printf` 是检查程序行为最方便的方法。`Wizard` 能够为 `printf` 生成支撑代码.

2. 跟踪（Trace）

   内部状态和 HCI 接口消息可通过 Trace 记录下来。`Wizard` 也能够为 Trace 生成支撑代码。Trace 包含几种预先定义的数据类型，可以编程选择记录哪些数据类型。使用Tracer 查看 Trace 数据。

表 4.1： printf 和 Trace 的对比

方法	优点	缺点
printf	通用	慢
Trace	二进制数据，速度快	数据类型为预先定义

`printf` 和 trace 都可以配置成从 UART 口或者 SEGGER RTT[3] 输出。表 4.2 对比了两种传输方式。

表 4.2： UART 的 SEGGER RTT 的对比

传输方式	优点	缺点
UART	通用，使用简单	慢，占用更多的 CPU 时间
SEGGER RTT	速度快	需要 J-Link 等工具，难以抓取上电阶段的数据

4.6.1 有关 SEGGER RTT 的使用提示

- 使用 `J-LINK RTT Viewer` 实时查看 `printf` 输出
- 使用 `J-LINK RTT Logger` 将 trace 记录到文件

  这个工具会询问有关 RTT 的设置：设备名称（Device name）为 “CORTEX-M3”；目标接口（Target interface）为 “SWD”；RTT 控制块（RTT Control Block）的地址即名为 `_SEGGER_RTT` 的变量的地址，可从 `.map` 文件中找到；RTT 通道号为 `0`。以下是一个示例：

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

Device name. Default: CORTEX-M3 >
Target interface. > SWD
```

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

```
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 Lite ..........
  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.
```

或者，这个工具可以从命令行启动，通过参数设定 _SEGGER_RTT 的地址范围，工具会自动搜索实际地址。例如：

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

对于 ING916xx，将上面的 CORTEX-M3 替换为 CORTEX-M4。

4.6.2 内存转储

我们致力于提供高质量的 platform 软件包。如果在 platform 二进制文件内发生断言（assertion）错误，建议转储全部内存，记录各寄存器的值。请从开发者手册获得各内存区域的地址范围。使用Axf Tool 分析内存转储。如果问题无法解决，请联系技术支持。

- Keil µVision

  在调试状态下，打开命令窗口用 save 命令保存每个区域。以 ING918xx 为例：

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

- J-Link Commander

  连接到设备后，使用 regs 查看所有寄存器的当前值，使用 savebin 将每个区域保存到文件。以 ING918xx 为例：

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

- IAR Embedded Workbench

  在调试状态下，打开内存窗口，打开快捷菜单，用 “Memory Save …” 保存数据。

- Rowley Crossworks for ARM & SEGGER Embedded Studio for ARM

  在调试状态下，打开内存窗口，对于每个区域：

  1. 填写起始地址和大小；
  2. 打开快捷菜单，用 “Memory Save …” 保存数据。

- GDB (GNU Arm Embedded Toolchain)

  在 GDB 调试模式下，使用 dump 命令保存每个区域。

内存也可以使用一小段专门的代码导出。例如在 PLATFORM_CB_EVT_ASSERTION 事件的回调里，将整个内存数据通过 UART 导出。

第五章　Platform API 参考

本章节介绍 platform API。

5.1　配置与信息

5.1.1　`platform_config`

配置一些 platform 功能。

5.1.1.1　原型

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

5.1.1.2　参数

- `const platform_cfg_item_t item`

  指定要配置的项目。可以是以下值之一：

  - `PLATFORM_CFG_LOG_HCI`：打印主机控制器接口消息。默认：禁用。

    仅在 ING918 上可用。HCI 日志仅用于对 BLE 行为进行快速检查。请考虑使用跟踪（参见调试与跟踪）。

  - `PLATFORM_CFG_POWER_SAVING`：省电模式。默认：禁用。

  - `PLATFORM_CFG_TRACE_MASK`：选定的跟踪项的位图。默认：`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_RT_RC_EN：启用/禁用实时 RC 时钟。默认：启用。
- PLATFORM_CFG_RT_OSC_EN：启用/禁用实时晶体振荡器。默认：启用。
- PLATFORM_CFG_RT_CLK：实时时钟选择。标志为 platform_rt_clk_src_t。默认：PLATFORM_RT_RC

```
typedef enum
{
    PLATFORM_RT_OSC,            // 外部实时晶体振荡器
    PLATFORM_RT_RC              // 内部实时 RC 时钟
} platform_rt_clk_src_t;
```

对于 ING918，修改此配置时，RT_RC 和 RT_OSC 都应**启用**并**运行**：

  * 对于 RT_OSC，等待直到 RT_OSC 的状态为 OK；
  * 对于 RT_RC，启用后等待 100µs。

并在禁用未使用的时钟之前再等待 100µs。

- PLATFORM_CFG_RT_CLK_ACC：配置实时时钟精度（ppm）。
- PLATFORM_CFG_RT_CALI_PERIOD：实时时钟自动校准周期（秒）。默认：3600 * 2（2 小时）。
- PLATFORM_CFG_DEEP_SLEEP_TIME_REDUCTION:睡眠时间缩减（深度睡眠模式）（微秒）。ING918 默认：~550µs。
- PLATFORM_CFG_SLEEP_TIME_REDUCTION：睡眠时间缩减（其他睡眠模式）（微秒）。ING918 默认：~450µs。

– PLATFORM_CFG_LL_DBG_FLAGS：链路层标志。ll_cfg_flag_t 中的位组合。

```
typedef enum
{
    LL_FLAG_DISABLE_CTE_PREPROCESSING = 1,  // 禁用 CTE 预处理
    LL_FLAG_LEGACY_ONLY_INITIATING = 4,     // 仅针对传统 ADV 进行发起
    LL_FLAG_LEGACY_ONLY_SCANNING = 8,       // 仅针对传统 ADV 进行扫描
} ll_cfg_flag_t;
```

– PLATFORM_CFG_LL_LEGACY_ADV_INTERVAL：链路层传统广告间隔，高占空比模式（高 16 位）和正常占空比模式（低 16 位）（微秒）。高占空比模式默认：1250；正常占空比模式默认：1500。

– PLATFORM_CFG_RTOS_ENH_TICK：启用 RTOS 的增强型滴答。默认：禁用。启用后，当外设频繁生成中断请求时，可以保持滴答的准确性。

– PLATFORM_CFG_LL_DELAY_COMPENSATION：链路层的延迟补偿。

当系统以较低频率运行时，链路层需要更多时间（以 µs 为单位）来调度 RF 任务。例如，如果 ING916 以 24MHz 运行，则需要约 2500µs 的补偿。

– PLATFORM_CFG_24M_OSC_TUNE：24M OSC 调谐。不适用于 ING918。

对于 ING916，调谐值可能在 0x16~0x2d 之间。

– PLATFORM_CFG_ALWAYS_CALL_WAKEUP：无论深度睡眠过程是否完成或中止（失败），始终触发 PLATFORM_CB_EVT_ON_DEEP_SLEEP_WAKEUP 事件。ING918 默认：禁用以保持向后兼容性。ING916 默认：启用。

– PLATFORM_CFG_FAST_DEEP_SLEEP_TIME_REDUCTION：快速深度睡眠模式的睡眠时间减少（微秒）。不适用于 ING918。

此配置必须小于或等于 PLATFORM_CFG_DEEP_SLEEP_TIME_REDUCTION。当等于 PLATFORM_CFG_DEEP_SLEEP_TIME_REDUCTION 时，不使用快速深度睡眠模式。

ING916 默认：~2000µs。

– PLATFORM_CFG_AUTO_REDUCE_CLOCK_FREQ：在这些情况下自动降低 CPU 时钟频率：

  * 默认的 IDLE 过程，
  * 进入睡眠模式时。

不适用于 ING918。ING916 默认：启用。

- const uint32_t flag

用于禁用或启用项目。可以是以下值之一：

- PLATFORM_CFG_ENABLE
- PLATFORM_CFG_DISABLE

5.1.1.3 返回值

无。

5.1.1.4 备注

无。

5.1.1.5 示例

```
// 在 ING918 上启用 HCI 日志记录
platform_config(PLATFORM_CFG_LOG_HCI, PLATFORM_CFG_ENABLE);
```

5.1.2 platform_get_version

获取 platform 的版本号。

5.1.2.1 函数原型

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

5.1.2.2 参数

无。

5.1.2.3 返回值

指向 platform_ver_t 的指针。

5.1.2.4 备注

platform 版本号由三部分组成：主版本号、次版本号和修订号：

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

5.1.2.5 示例

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

5.1.3 platform_read_info

读取 platform 信息。

5.1.3.1 函数原型

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

5.1.3.2 参数

- const platform_info_item_t item

  信息项。

  - PLATFORM_INFO_RT_OSC_STATUS: 读取实时晶体振荡器的状态。值为 0：不正常；非 0：正常。

对于 ING916：此时钟在被选为实时时钟源之后才会开始运行。

- PLATFORM_INFO_RT_CLK_CALI_VALUE: 读取当前实时时钟校准结果。
- PLATFOFM_INFO_IRQ_NUMBER: 获取 platform IRQ 的底层 IRQ 编号。

  例如，获取 UART0 的底层 IRQ 编号：

```
platform_read_info(
    PLATFOFM_INFO_IRQ_NUMBER + PLATFORM_CB_IRQ_UART0)
```

5.1.3.3 返回值

信息项的值。

5.1.3.4 备注

无。

5.1.3.5 示例

```
platform_read_info(PLATFORM_INFO_RT_OSC_STATUS);
```

5.1.4 platform_switch_app

切换到次级应用程序。

5.1.4.1 函数原型

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

5.1.4.2 参数

- `const uint32_t app_addr`

  次级应用程序的入口地址。

5.1.4.3 返回值

无。

5.1.4.4 备注

调用此函数后，其后的代码将不会被执行。

5.1.4.5 示例

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

5.2 事件与中断

5.2.1 platform_set_evt_callback_table

为所有 platform 事件注册一个回调函数表。

5.2.1.1 函数原型

```
void platform_set_evt_callback_table(
    const platform_evt_cb_table_t *table);
```

5.2.1.2 参数

- `const platform_evt_cb_table_t *table`

  回调函数表的地址。

5.2.1.3 返回值

无。

5.2.1.4 备注

此函数应仅在 app_main 中调用。如果使用了 platform_set_evt_callback，则不应使用此函数。

与 platform_set_evt_callback 相比，使用此函数可以节省 sizeof(platform_evt_cb_table_t) 字节的 RAM 空间。

5.2.1.5 示例

```
static const platform_evt_cb_table_t evt_cb_table =
{
    .callbacks = {
        [PLATFORM_CB_EVT_HARD_FAULT] = {
            .f = (f_platform_evt_cb)cb_hard_fault,
        },
        [PLATFORM_CB_EVT_PROFILE_INIT] = {
            .f = setup_profile,
        },
        // ...
    }
};

int app_main()
{
    // ...
    platform_set_evt_callback_table(&evt_cb_table);
    // ...
}
```

5.2.2 platform_set_irq_callback_table

为所有 platform 中断请求注册一个回调函数表。

5.2.2.1 函数原型

```
void platform_set_irq_callback_table(
    const platform_irq_cb_table_t *table);
```

5.2.2.2 参数

- `const platform_irq_cb_table_t *table`

  回调函数表的地址。

5.2.2.3 返回值

无。

5.2.2.4 备注

此函数只能在 `app_main` 中调用。如果使用了 `platform_set_irq_callback`，则不应使用此函数。

与 `platform_set_irq_callback` 相比,使用此函数可以节省 `sizeof(platform_irq_cb_table_t)` 字节的 RAM。

5.2.2.5 示例

```
static const platform_irq_cb_table_t irq_cb_table =
{
    .callbacks = {
        [PLATFORM_CB_IRQ_UART0] = {
```

```
            .f = (f_platform_irq_cb)cb_irq_uart,
            .user_data = APB_UART0
        },
        // ...
    }
};

int app_main()
{
    // ...
    platform_set_irq_callback_table(&irq_cb_table);
    // ...
}
```

5.2.3 platform_set_evt_callback

注册回调函数以处理 platform 事件。

5.2.3.1 函数原型

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

5.2.3.2 参数

- platform_evt_callback_type_t type

  指定要注册回调函数的事件类型。可以是以下值之一：

  - PLATFORM_CB_EVT_PUTC：输出 ASCII 字符事件

    当 platform 想要输出 ASCII 字符用于日志记录时，会触发此事件。传递给回调函数的参数 void *data 是从 char * 类型转换而来。

如果在创建新项目时在 Common Function 中勾选了 Print to UART，Wizard 可以自动生成将 platform 日志重定向到 UART 的代码。

- PLATFORM_CB_EVT_PROFILE_INIT：配置文件初始化事件

  当主机初始化时，会触发此事件以请求应用程序初始化 GATT 配置文件。

  在创建新项目时，Wizard 可以自动为此事件生成代码。

- PLATFORM_CB_EVT_ON_DEEP_SLEEP_WAKEUP：从深度睡眠唤醒事件

  当从深度睡眠唤醒时，会触发此事件。在深度睡眠期间，外设接口（如 UART、I2C 等）都已关闭。因此，唤醒时可能需要重新初始化这些接口。

  如果在创建新项目时在 Common Function 中勾选了 Deep Sleep，Wizard 可以自动为此事件生成代码。

  传递给回调函数的参数 void *data 是从 platform_wakeup_call_info_t * 类型转换而来。

  此时 RTOS 尚未恢复，某些 RTOS API 不可用；某些 platform API（如 platform_get_us_time）也可能不可用。

- PLATFORM_CB_EVT_ON_IDLE_TASK_RESUMED：操作系统从节能模式完全恢复。

  如果唤醒原因是 PLATFORM_WAKEUP_REASON_NORMAL,则在 PLATFORM_CB_EVT_ON_DEEP_SLEEP_WAKE
  之后调用此回调。对于 NoOS 变体，回调由 platform_os_idle_resumed_hook() 调用。此事件与 PLATFORM_CB_EVT_ON_DEEP_SLEEP_WAKEUP 不同：

  * 所有操作系统功能都已恢复（对于 NoOS 变体,这取决于 platform_os_idle_resumed_hook(
    的正确使用）
  * 所有 platform API 都可用
  * 回调在空闲任务中调用。

  参数 void *data 始终为 NULL。

- PLATFORM_CB_EVT_QUERY_DEEP_SLEEP_ALLOWED：查询是否允许深度睡眠事件

  当 platform 准备进入深度睡眠模式时，会触发此事件以查询应用程序当前是否允许深度睡眠。回调函数可以通过返回 0 拒绝深度睡眠，或通过返回非 0 值允许深度睡眠。

  如果在创建新项目时在 Common Function 中勾选了 Deep Sleep，Wizard 可以自动为此事件生成代码。

- PLATFORM_CB_EVT_HARD_FAULT：硬件故障发生

  当硬件故障发生时，会触发此事件。传递给回调函数的参数 void *data 是从 hard_fault_info_t * 类型转换而来。如果未定义此回调，当硬件故障发生时，CPU 将进入死循环。

- PLATFORM_CB_EVT_ASSERTION：软件断言失败

  当软件断言失败时，会触发此事件。传递给回调函数的参数 void *data 是从 assertion_info_t * 类型转换而来。如果未定义此回调，当断言发生时，CPU 将进入死循环。

- PLATFORM_CB_EVT_LLE_INIT：链路层引擎初始化。

  当链路层引擎初始化时，会触发此事件。

- PLATFORM_CB_EVT_HEAP_OOM：内存不足。

  当堆分配失败（堆内存不足）时，会触发此事件。如果触发此事件且未定义回调，CPU 将进入死循环。

- PLATFORM_CB_EVT_TRACE：跟踪输出。

  当发出跟踪项时，会触发此事件。应用程序可以为此事件定义回调函数以保存或记录跟踪输出。回调的 param 是从 platform_trace_evt_t * 类型转换而来（参见调试与跟踪）。

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

  跟踪项是 data1 和 data2 的组合。注意：

  1. len1 或 len2 可能为 0，但不能同时为 0；
  2. 如果回调函数发现无法输出大小为 len1 + len2 的数据，则应丢弃 data1 和 data2 以避免跟踪项损坏。

- PLATFORM_CB_EVT_EXCEPTION：硬件异常。

  参数 void *data 是从 platform_exception_id_t * 类型转换而来。

- PLATFORM_CB_EVT_IDLE_PROC：自定义 IDLE 过程。

  参见 “Programmer’s Guide - Power Saving”[1]。

- PLATFORM_CB_EVT_HCI_RECV：接管 HCI 并完全隔离内置主机。

  当定义时：

[1] https://ingchips.github.io/application-notes/pg_power_saving_en/

* HCI 事件和 ACL 数据传递给此回调；
* PLATFORM_CB_EVT_PROFILE_INIT 被忽略。

参数 void *data 是从 const platform_hci_recv_t * 类型转换而来。另请参见platform_get_link_layer_interf。

- f_platform_evt_cb f

  注册到事件 type 的回调函数。f_platform_evt_cb 定义为：

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

- void *user_data

  此参数原封不动地传递给回调函数的 user_data。

5.2.3.3 返回值

无。

5.2.3.4 备注

不需要为每个事件注册回调函数。

如果未为 PLATFORM_CB_EVT_PUTC 事件注册回调函数，所有 platform 日志（包括 platform_printf）将被丢弃。

如果未为 PLATFORM_CB_EVT_PROFILE_INIT 事件注册回调函数，BLE 设备的配置文件将为空。

如果未为 PLATFORM_CB_EVT_ON_DEEP_SLEEP_WAKEUP 事件注册回调函数，应用程序在从深度睡眠唤醒时将不会收到通知。

如果未为 PLATFORM_CB_EVT_QUERY_DEEP_SLEEP_ALLOWED 事件注册回调函数，深度睡眠将被禁用。

5.2.3.5 示例

```
uint32_t cb_putc(char *c, void *dummy)
{
    // TODO: 将字符 c 输出到 UART
    return 0;
}

......

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

5.2.4 platform_set_irq_callback

注册中断请求的回调函数。

开发者无需在应用中定义 IRQ 处理程序，而是使用回调函数代替。

5.2.4.1 函数原型

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

5.2.4.2 参数

- platform_irq_callback_type_t type

  指定要注册回调函数的 IRQ 类型。不同芯片系列的值有所不同。以 ING918 为例：

```
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`

  注册到 IRQ type 的回调函数。`f_platform_irq_cb` 定义为：

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

- `void *user_data`

  该参数将原封不动地传递给回调函数的 `user_data`。

5.2.4.3 返回值

无。

5.2.4.4 备注

当回调函数注册到 IRQ 时，IRQ 会自动启用。另请参见`platform_enable_irq`。

5.2.4.5 示例

```
uint32_t cb_irq_uart0(void *dummy)
{
    // TODO: 添加 UART0 IRQ 处理代码
    return 0;
```

```
}

......

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

5.2.5 platform_enable_irq

启用或禁用指定的 IRQ（中断请求）。

5.2.5.1 函数原型

```
void platform_enable_irq(
    platform_irq_callback_type_t type,
    uint8_t flag);
```

5.2.5.2 参数

- platform_irq_callback_type_t type:

  需要配置的 IRQ。

- uint8_t flag:

  启用（1）或禁用（0）。

5.2.5.3 返回值

无。

5.2.5.4 备注

这里的 “启用” 或 “禁用” 中断是从 CPU 的角度来看的。以 UART 为例，UART 本身需要配置为在接收、发送或超时时生成中断，这不在本函数的范围内。

5.2.5.5 示例

启用来自 UART0 的中断请求：

```
platform_enable_irq(
    PLATFORM_CB_IRQ_UART0,
    1);
```

5.3 时钟

另请参阅 “Programmer’s Guide - Power Saving”[2] 中的 “The Real-time Clock”[3]。

5.3.1 platform_calibrate_rt_clk

校准实时时钟并获取校准值。

5.3.1.1 函数原型

```
uint32_t platform_calibrate_rt_clk(void);
```

5.3.2 platform_rt_rc_auto_tune

自动调整内部实时 RC 时钟，并获取调整值。

对于 ING918，此函数将内部实时 RC 时钟调整至 50kHz[4]。对于其他设备，它将内部实时 RC 时钟调整至 32768Hz。

[2] https://ingchips.github.io/application-notes/pg_power_saving_en/

[3] https://ingchips.github.io/application-notes/pg_power_saving_en/ch-api.html#the-real-time-clock

[4] 从 v8.4.6 版本开始。对于更早的版本，使用的是 32768Hz。

5.3.2.1 函数原型

```
uint16_t platform_rt_rc_auto_tune(void);
```

5.3.2.2 参数

无。

5.3.2.3 返回值

16 位的调整值。

5.3.2.4 备注

如果应用程序启用了省电模式，并且使用实时 RC 时钟作为时钟源，则必须调用此函数。

此操作耗时约 250ms。建议调用一次并存储返回值以供后续使用。

5.3.2.5 示例

```
// 最简单的示例：在`PLATFORM_CB_EVT_PROFILE_INIT`的回调函数中调用此函数，
// 不保存返回值。
uint32_t setup_profile(void *user_data)
{
    platform_rt_rc_auto_tune();
    ...
}
```

5.3.3 platform_rt_rc_auto_tune2

自动调整内部实时 RC 时钟至特定频率，并获取调整值。

5.3.3.1 函数原型

```
uint16_t platform_rt_rc_auto_tune2(
    uint32_t target_frequency);
```

5.3.3.2 参数

- `uint32_t target_frequency`

  目标频率，单位为赫兹（Hz）。

5.3.3.3 返回值

返回 16 位的调整值。

5.3.4 platform_rt_rc_tune

使用调谐值调整内部实时 RC 时钟。

5.3.4.1 函数原型

```
void platform_rt_rc_tune(uint16_t value);
```

5.3.4.2 参数

- `uint16_t value`

  用于调谐时钟的值（由`platform_rt_rc_auto_tune` 或`platform_rt_rc_auto_tune2` 返回）

5.3.4.3 返回值

无。

5.3.4.4 备注

无。

5.3.4.5 示例

```
platform_rt_rc_tune(value);
```

5.4 RF

5.4.1 platform_set_rf_clk_source

选择射频时钟源。此函数供内部使用。

5.4.2 platform_set_rf_init_data

自定义射频初始化数据。此函数供内部使用。

5.4.3 platform_set_rf_power_mapping

功率等级在内部由一个索引表示。存在一个功率映射表，该表列出了每个索引对应的实际发射功率等级。以 ING918 为例，功率索引的范围是 [0..63]，功率映射表是一个包含 64 个条目的数组，每个条目给出了以 0.01 dBm 为单位的发射功率等级。

对于需要更好功率等级控制的应用，可以测量每个索引的实际功率等级。使用此函数更新映射表后，堆栈可以确定请求功率等级的适当索引。

5.4.3.1 函数原型

```
void platform_set_rf_power_mapping(
    const int16_t *rf_power_mapping);
```

5.4.3.2 参数

- const int16_t *rf_power_mapping

  新的功率映射表。

5.4.3.3 返回值

无。

5.4.3.4 备注

无。

5.4.3.5 示例

```
static const int16_t power_mapping[] =
{
    -6337,  // 索引 0: -63.37dBm
    // ...
    603     // 索引 63: 6.03dBm
};


platform_set_rf_power_mapping(
    power_mapping);
```

5.4.4 platform_patch_rf_init_data

修补部分内部射频初始化数据。此函数供内部使用。

5.5 内存与实时操作系统（RTOS）

5.5.1 platform_call_on_stack

在独立的专用堆栈上调用函数。当需要偶尔调用一个使用大量堆栈的函数时，可以使用此接口。

5.5.1.1 函数原型

```
void platform_call_on_stack(
    f_platform_function f,
    void *user_data,
    void *stack_start,
    uint32_t stack_size);
```

5.5.1.2 参数

- `f_platform_function f`

  要调用的函数。

- `void *user_data`

  传递给 `f` 的用户数据。

- `void *stack_start`

  专用堆栈的起始（最低）地址。

- `uint32_t stack_size`

  专用堆栈的大小，以字节为单位。

5.5.1.3 返回值

无。

5.5.1.4 备注

尽管提供了 stack_size，但此函数不会保护堆栈不被 f 覆盖。

5.5.2 platform_get_current_task

获取调用此 API 的当前任务。

5.5.2.1 函数原型

```
platform_task_id_t platform_get_current_task(void);
```

5.5.2.2 参数

无。

5.5.2.3 返回值

```
typedef enum
{
    PLATFORM_TASK_CONTROLLER,  // 控制器任务
    PLATFORM_TASK_HOST,        // 主机任务
    PLATFORM_TASK_RTOS_TIMER,  // RTOS 定时器任务
} platform_task_id_t;
```

5.5.2.4 备注

此 API 仅在带有内置 RTOS 的软件包中可用。

5.5.3 platform_get_gen_os_driver

获取通用操作系统驱动。对于 “NoOS” 变体，返回由应用程序提供的驱动；对于内置 RTOS 的软件包，返回一个模拟出来的驱动接口。

5.5.3.1 函数原型

```
const void *platform_get_gen_os_driver(void);
```

5.5.3.2 参数

无。

5.5.3.3 返回值

返回值被强制转换为 `const gen_os_driver_t *`。开发者可以将返回值强制转换回 `const gen_os_driver_t *` 并使用其中的 API。

5.5.3.4 备注

`gen_os_driver_t` 是 RTOS 上的一个抽象层。使用其中的 API 而不是直接使用 RTOS API 可以使应用程序跨 RTOS（独立于底层 RTOS）。

5.5.4 platform_get_heap_status

获取当前堆状态，例如可用大小等。

5.5.4.1 函数原型

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

5.5.4.2 参数

- `platform_heap_status_t *status`

  堆状态。

5.5.4.3 返回值

无。

5.5.4.4 备注

堆状态定义为：

```
typedef struct
{
    uint32_t bytes_free;                // 总空闲字节数
    uint32_t bytes_minimum_ever_free;   // 自启动以来 bytes_free 的最小值
} platform_heap_status_t;
```

5.5.4.5 示例

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

5.5.5 platform_get_task_handle

获取特定 platform 任务的 RTOS 句柄。

5.5.5.1 函数原型

```
uintptr_t platform_get_task_handle(
    platform_task_id_t id);
```

5.5.5.2 参数

- `platform_task_id_t id`

  platform 任务 ID。

5.5.5.3 返回值

如果 platform 知道该任务的句柄，则返回任务句柄；否则返回 0。例如，在 “NoOS” 变体中，platform 不知道 `PLATFORM_TASK_RTOS_TIMER` 的句柄，因此返回 0。

5.5.6 platform_install_task_stack

为特定 platform 任务安装一个新的 RTOS 堆栈。

当默认堆栈大小不足以满足内部任务需求时，使用此函数来扩大堆栈。例如，用户开发的 RTOS 定时器回调可能需要更大的堆栈空间。

开发者可以查阅 RTOS 文档以了解如何检查堆栈使用情况。例如，在 FreeRTOS 中，`uxTaskGetStackHighWaterMark` 用于查询任务接近溢出分配给它的堆栈空间的程度。

5.5.6.1 函数原型

```
void platform_install_task_stack(
    platform_task_id_t id,
    void *start,
    uint32_t size);
```

5.5.6.2 参数

- `platform_task_id_t id`

  任务标识符。

- `void *start`

  堆栈的起始（最低）地址。地址应针对底层 CPU 进行适当对齐。

- `uint32_t size`

  新堆栈的大小，以字节为单位。

5.5.6.3 返回值

无。

5.5.6.4 备注

此函数应仅在 app_main 中调用。

对于 NoOS 变体，在实现通用操作系统接口时，可以替换 RTOS 堆栈（修改其大小等）。

5.5.7 platform_install_isr_stack

为中断服务例程（ISR）安装一个新的堆栈。

5.5.7.1 函数原型

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

5.5.7.2 参数

- void *top

  新堆栈的顶部，必须针对底层 CPU 进行适当对齐。

5.5.7.3 返回值

无。

5.5.7.4 备注

如果应用程序需要比 ISR 中默认堆栈更大的堆栈，可以安装一个新堆栈来替换默认堆栈。

此函数仅允许在 app_main 中调用。新堆栈将在 app_main 返回后投入使用。

5.5.7.5 示例

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

5.6 时间与定时器

读取当前时间（定时器计数器）的 API：

- `platform_get_timer_counter`
- `platform_get_us_time`

使用 625 微秒分辨率的定时器的 API：

- `platform_set_abs_timer`
- `platform_set_timer`
- `platform_delete_timer`

使用 1 微秒分辨率的定时器的 API：

- `platform_create_us_timer`
- `platform_cancel_us_timer`

这两种类型的定时器都可以在节能模式下使用，即当节能模式启用时，它们会按预期工作。表5.6比较了这两种定时器。

表: 两种 platform 定时器的比较

类型	625 微秒分辨率	1 微秒分辨率
回调	从类似任务的上下文中调用	从中断服务例程（ISR）中调用
标识符	回调函数指针	定时器句柄

5.6.1 `platform_cancel_us_timer`

取消之前由 `platform_create_us_timer` 创建的 platform 定时器。

5.6.1.1 函数原型

```
int platform_cancel_us_timer(
    platform_us_timer_handle_t timer_handle);
```

5.6.1.2 参数

- `platform_us_timer_handle_t timer_handle`

  定时器的句柄。

5.6.1.3 返回值

如果指定的定时器成功取消，该函数返回 0。否则，返回一个非 0 值，这也意味着定时器的回调函数正在执行中。### `platform_create_us_timer`

设置一个具有 1 微秒（μs）分辨率的单次触发 platform 定时器。

5.6.1.4 函数原型

```
platform_us_timer_handle_t platform_create_us_timer(
    uint64_t abs_time,
    f_platform_us_timer_callback callback,
    void *param);
```

5.6.1.5 参数

- `uint64_t abs_time`

  当 `platform_get_us_timer() == abs_time` 时，回调函数将被调用。

- `f_platform_us_timer_callback callback`

  回调函数。其签名为：

```
typedef void * (* f_platform_us_timer_callback)(
    platform_us_timer_handle_t timer_handle,
    uint64_t time_us,
    void *param);
```

其中，timer_handle 是 platform_create_us_timer 的返回值，time_us 是调用回调时 platform_get_us_timer 的当前值，param 是创建此定时器时的用户参数。

- void *param

  用户参数。

5.6.1.6 返回值

该函数返回创建的定时器的句柄。如果定时器成功创建，则返回非 NULL 值。否则，返回 NULL。

5.6.1.7 备注

尽管 abs_time 以微秒（μs）为单位，但**不保证**回调函数会以这样的分辨率被调用。

这种类型的定时器与 platform_set_timer 非常相似，除了：

1. 分辨率更高；
2. 回调函数在中断服务例程（ISR）的上下文中被调用。

不要在 callback 中再次调用 platform_create_us_timer。### platform_delete_timer

删除先前由 platform_set_timer 或 platform_set_abs_timer 创建的 platform 定时器。

5.6.1.8 函数原型

```
void platform_delete_timer(f_platform_timer_callback callback)
```

5.6.1.9 参数

- `f_platform_timer_callback callback`

  回调函数，同时也是定时器的标识符。

5.6.1.10 返回值

无。

5.6.1.11 备注

调用此函数时，回调可能已经排队等待在任务中调用。因此，在调用此函数后，回调仍可能被调用。

5.6.2 platform_get_timer_counter

读取 platform 定时器的计数器，分辨率为 625 微秒。

5.6.2.1 函数原型

```
uint32_t platform_get_timer_counter(void);
```

5.6.2.2 参数

无。

5.6.2.3 返回值

一个完整的 32 位值，表示当前计数器的值，大致等于 `platform_get_us_time() / 625`。### platform_get_us_time

读取自 BLE 初始化以来的内部时间计数。

5.6.2.4 函数原型

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

5.6.2.5 参数

无。

5.6.2.6 返回值

内部时间计数器的值，以 1 微秒为单位计数。该计数器大约每 21.8 年溢出一次。

5.6.2.7 备注

该计数器在关机后重新启动。

5.6.2.8 示例

```
uint64_t now = platform_get_us_time();
```

5.6.3 platform_set_abs_timer

设置一个单次触发的 platform 定时器，在绝对时间触发，分辨率为 625 微秒。

5.6.3.1 函数原型

```
void platform_set_abs_timer(
    f_platform_timer_callback callback,
    uint32_t abs_time);
```

5.6.3.2 参数

- `f_platform_timer_callback callback`

  定时器到期时的回调函数，该函数在类似 RTOS 任务的上下文中调用，而不是在中断服务例程（ISR）中调用。

- `uint32_t abs_time`

  当 `platform_get_timer_counter() == abs_time` 时，`callback` 被调用。如果 `abs_time` 刚刚超过 `platform_get_timer_counter()`，`callback` 会立即被调用，例如，`abs_time` 为 `platform_get_timer_counter() - 1`。

5.6.3.3 返回值

无。

5.6.3.4 备注

此函数总是成功，除非内存不足。

5.6.3.5 示例

使用此函数模拟一个周期性定时器。

```
#define PERIOD  100
static uint32_t last_timer = 0;

void platform_timer_callback(void)
{
    last_timer += PERIOD;
    platform_set_abs_timer(platform_timer_callback, last_timer);

    // 执行周期性任务
    // ...
}
```

```
last_timer = platform_get_timer_counter() + PERIOD;
platform_set_abs_timer(platform_timer_callback, last_timer);
```

5.6.4 platform_set_timer

设置一个单次触发的 platform 定时器，从 “现在” 开始延迟，分辨率为 625 微秒。

5.6.4.1 函数原型

```
void platform_set_timer(
    f_platform_timer_callback callback,
    uint32_t delay);
```

5.6.4.2 参数

- `f_platform_timer_callback callback`

  定时器到期时的回调函数，该函数在类似 RTOS 任务的上下文中调用，而非中断服务例程（ISR）。

- `uint32_t delay`

  定时器到期前的延迟时间（单位：625 微秒）。

  有效范围：0~0x7fffffff。当 `delay` 为 0 时，定时器被清除。

5.6.4.3 返回值

无。

5.6.4.4 备注

此函数总是成功，除非内存耗尽。

`platform_set_timer(f, 100)` 等同于：

```
platform_set_abs_timer(f,
    platform_get_timer_counter() + 100);
```

platform_set_timer(f, 0) 等同于：

```
platform_delete_timer(f);
```

但不等同于：

```
platform_set_abs_timer(f,
    platform_get_timer_counter() + 0);
```

由于 callback 也是定时器的标识符，以下两行代码仅定义了一个在 200 个单位后到期的定时器，而非两个独立的定时器：

```
platform_set_timer(f, 100);
platform_set_timer(f, 200); // 更新定时器，而非创建新的
```

如果 f 再次用于 platform_set_abs_timer，则定时器再次被更新：

```
platform_set_abs_timer(f, ...);
```

5.7 实用工具

5.7.1 platform_hrng

使用硬件随机数生成器生成随机字节。

5.7.1.1 函数原型

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

5.7.1.2 参数

- uint8_t *bytes

  随机数据输出。

- const uint32_t len

  要生成的随机字节数。

5.7.1.3 返回值

无。

5.7.1.4 备注

生成固定长度数据所需的时间是不确定的。

5.7.1.5 示例

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

5.7.2 platform_rand

通过内部伪随机数生成器（PRNG）生成一个伪随机整数。

5.7.2.1 函数原型

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

5.7.2.2 参数

无。

5.7.2.3 返回值

返回一个范围在 0 到 RAND_MAX 之间的伪随机整数。

5.7.2.4 备注

内部 PRNG 的种子在启动时由硬件随机数生成器（HRNG）初始化。此函数可用作标准库中 `rand()` 函数的替代。

5.7.2.5 示例

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

5.7.3 platform_read_persistent_reg

从持久寄存器中读取值。另请参阅`platform_write_persistent_reg`。

5.7.3.1 函数原型

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

5.7.3.2 参数

无。

5.7.3.3 返回值

由platform_write_persistent_reg 写入的值。

5.7.3.4 备注

无。

5.7.3.5 示例

```
platform_read_persistent_reg();
```

5.7.4 platform_reset

重置 platform（SoC）。

5.7.4.1 函数原型

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

5.7.4.2 参数

无。

5.7.4.3 返回值

无。

5.7.4.4 备注

调用此函数后，其后的代码将不会被执行。

5.7.4.5 示例

```
if (内存不足)
  platform_reset();
```

5.7.5 platform_shutdown

将整个系统置于关机状态，并在指定时间后重新启动。可选地，在关机期间可以保留一部分内存，重启后应用程序可以继续使用这部分内存。

请注意，除非关机过程无法启动，否则此函数不会返回。可能导致失败的原因包括：

1. 外部唤醒信号被发出；
2. 输入参数不正确；
3. 内部组件繁忙。

5.7.5.1 函数原型

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

5.7.5.2 参数

- `const uint32_t duration_cycles`

  重新上电（重启）前的等待时间（以实时时钟周期为单位）。最短持续时间为 825 个周期（约 25.18 毫秒）。如果使用 `0`，系统将保持在关机状态，直到外部唤醒信号被发出。

- `const void *p_retention_data`

  指向要保留数据的起始位置的指针。只有 SYSTEM 内存中的数据可以被保留。当 `data_size` 为 `0` 时，此参数可以设置为 `NULL`。

- `data_size`

  要保留的数据的大小。当不需要保留内存时，设置为 `0`。

5.7.5.3 返回值

无。

5.7.5.4 备注

无。

5.7.5.5 示例

```
// 关闭系统并在 1 秒后重启
platform_shutdown(32768, NULL, 0);
```

5.7.6 platform_write_persistent_reg

向持久寄存器写入一个值。该值在省电模式、关机模式或切换到另一个应用程序时仍会保留。

只有少量位被保存，如表 5.7.6 所示。

表: 持久寄存器位大小

芯片系列	寄存器大小 (位)
ING918	4
ING916	5

5.7.6.1 函数原型

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

5.7.6.2 参数

- `const uint8_t value`

  要写入的值。

5.7.6.3 返回值

无。

5.7.6.4 备注

无。

5.7.6.5 示例

```
platform_write_persistent_reg(1);
```

5.8 调试与追踪

5.8.1 platform_printf

存储在 platform 二进制文件中的 printf 函数。

5.8.1.1 原型

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

5.8.1.2 参数

- const char *format

  格式化字符串。

- ...

  格式化字符串的可变参数。

5.8.1.3 返回值

无。

5.8.1.4 备注

使用此函数有利有弊。

优点：

- 此函数位于 platform 二进制文件中，可以节省应用程序二进制文件的大小。

缺点：

- 输出被定向到 PLATFORM_CB_EVT_PUTC 事件，因此必须定义其回调函数。

5.8.1.5 示例

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

5.8.2 platform_raise_assertion

触发软件断言。

5.8.2.1 函数原型

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

5.8.2.2 参数

- const char *file_name

  断言发生的文件名。

- int line_no

  断言发生的行号。

5.8.2.3 返回值

无。

5.8.2.4 备注

无。

5.8.2.5 示例

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

5.8.3 platform_trace_raw

将一块原始数据输出到 TRACE。ID 为 PLATFORM_TRACE_ID_RAW。

5.8.3.1 函数原型

```
void platform_trace_raw(
    const void *buffer,
    const int byte_len);
```

5.8.3.2 参数

- const void *buffer

  缓冲区的指针。

- const int byte_len

  数据缓冲区的长度，以字节为单位。

5.9 其他

5.9.1 platform_get_link_layer_interf

获取链路层驱动 API。

5.9.1.1 函数原型

```
const platform_hci_link_layer_interf_t *
    platform_get_link_layer_interf(void);
```

5.9.1.2 参数

无。

5.9.1.3 返回值

返回驱动接口 `platform_hci_link_layer_interf_t *`。

5.9.1.4 备注

此 API 暴露了控制器 HCI 接口。该驱动接口仅在定义了 `PLATFORM_CB_EVT_HCI_RECV` 时可用，此时内置的主机功能将被禁用。

5.9.2 sysSetPublicDeviceAddr

设置设备的公共地址。

BLE 设备的公共地址是一个 48 位的扩展唯一标识符（EUI-48），按照 IEEE 802-2014 标准创建[5]。

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

INGCHIPS 91x **没有**公共地址。此函数应**仅**用于调试或测试，**切勿**在最终产品中使用。

5.9.2.1 函数原型

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

5.9.2.2 参数

- `const unsigned char *addr`

  新的公共地址。

5.9.2.3 返回值

无。

5.9.2.4 备注

为了避免潜在问题，应在调用任何 GAP 函数之前调用此函数。建议在 `app_main` 或 `PLATFORM_CB_EVT_PROFILE_INIT` 事件回调函数中调用此函数。

5.9.2.5 示例

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

第六章　版本历史

版本	信息	日期
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
1.3.0	Fix some errors	2024-05-28
1.4.0	Update Platform API	2025-01-20