PAN107x/PAN101x NDK 开发套件使用手册

发布 0.6.99

Panchip BLE SoC Team

2024 年 08 月 05 日

Table of contents

Chapter 1

快速入门

1.1 NDK 快速入门指南

1.1.1 1 概述

本文是 PAN107x/PAN101x NDK 开发的快速入门指引，旨在帮助使用者快速入门 PAN107x/PAN101x NDK 的相关开发。

1.1.2 2 PAN10xx EVB 介绍

PAN10xx EVB (EValuation Board) 是 Panchip 提供给 PAN107x/PAN101x 芯片用户的一系列开发板的总称，目前包括 2 种 EVB 核心板，1 种 EVB 底板：

开发板名称	SoC 型号	封装	Flash 大小	SRAM 大小
PAN1070UA1A EVB 核心板	PAN1070UA1A	QFN32 (4x4)	512 KB	48 KB
PAN1010S9FA EVB 核心板	PAN1010S9FA	SSOP24	256 KB	16 KB
PAN10xx EVB 底板	-	-	-	-

关于 PAN10xx EVB 开发板硬件的详细介绍，请参考PAN10xx EVB 硬件资源介绍。

1.1.3 3 NDK 开发环境确认

3.1 PC 环境检查

请确认 KEIL（推荐 5.25 版本以上），芯片 SWD 下载与调试依赖的 FLM 文件，Jlink 设备等准备就绪。

> **注**：PAN107x/PAN101x 芯片的 FLM 文件位于：`<PAN10XX-NDK>\03_MCU\mcu_misc` 目录，使用前需要将其拷贝到 Keil 安装目录（例如 `C:\Keil_v5\ARM\Flash`）下

3.2 快速编译运行一个简单的例程

开始硬件接线，若您操作的 EVB 核心板主控为 PAN107x 系列芯片，请将：

1. SWD（P00：SWD_CLK，P01：SWD_DAT，GND：SWD_GND）接口通过 JLink 连接至 PC
2. SoC UART0 接口通过板上的 USB 转串口模块连接至 PC
   - UART0-Tx：`P16`，UART0-Rx：`P17`
3. 打开一个 Sample 工程，例如 `03_MCU\mcu_samples\FMC` 下 Keil 子目录中的工程文件 `FMC.uvprojx`

4. 点击 `Build` 编译按钮，然后点击 `Download` 按钮进行下载（若无法正常下载，请检查 FLM 文件是否正常载入）
5. 下载完成可以通过串口观察 log 输出（**串口波特率：** 921600）

   注：若您操作的 EVB 核心板主控为 SSOP24 封装的 PAN101x 芯片，则 SDK 例程中默认的 UART 引脚为：UART0-Tx：`P11`，UART0-Rx：`P12`

1.2 NDK 开发环境介绍

1.2.1 开发 IDE

KEIL 官方下载连接：

https://www.keil.com/download/product/

图 1: Keil 下载 MDK-Arm 版本

当前 NDK 开发使用的工具为 KEIL, 开发中使用的版本为 5.25, 所以建议使用该版本及以上版本。

Keil 使用版本如下：

1.2.2 Keil Flash 下载程序

为使用 Keil + Jlink 烧录代码，请将此目录下的 FLM 文件，拷贝到 Keil MDK 安装目录下，如：

- C:\Keil_v5\ARM\Flash

相关 FLM 文件默认在 SDK 中路径为 `pan1070-ndk\03_MCU\mcu_misc`，keil 的工程配置根据芯片的类型选择不同的 FLM 文件

- `PAN107x_508KB_FLASH.FLM`
- `PAN101x_252KB_FLASH.FLM`

1.3 NDK 整体框架介绍

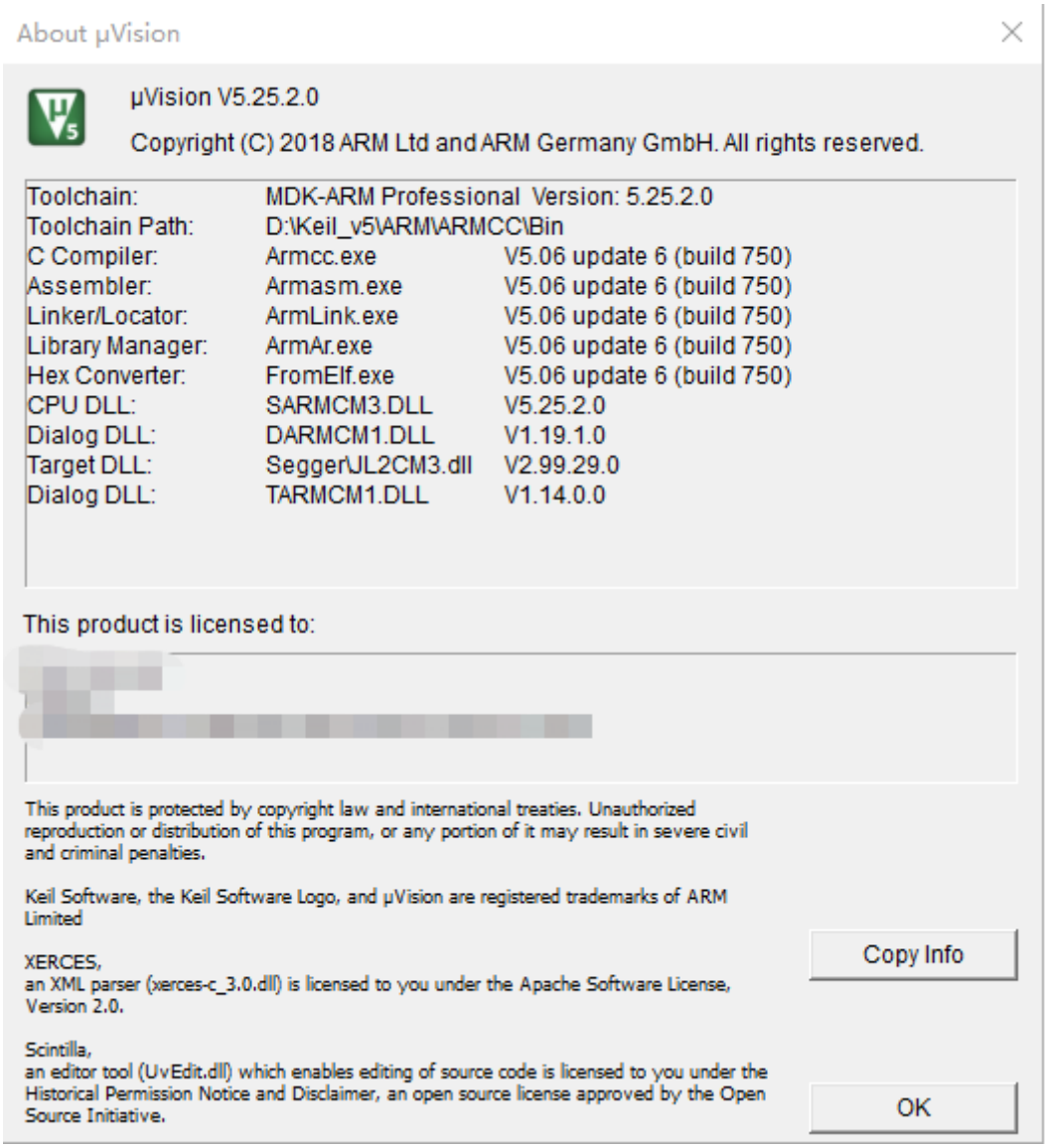

About µVision
µVision V5.25.2.0
Copyright (C) 2018 ARM Ltd and ARM Germany GmbH. All rights reserved.
Toolchain: MDK-ARM Professional Version: 5.25.2.0
Toolchain Path: D:\Keil_v5\ARM\ARMCC\Bin
C Compiler: Armcc.exe V5.06 update 6 (build 750)
Assembler: Armasm.exe V5.06 update 6 (build 750)
Linker/Locator: ArmLink.exe V5.06 update 6 (build 750)
Library Manager: ArmAr.exe V5.06 update 6 (build 750)
Hex Converter: FromElf.exe V5.06 update 6 (build 750)
CPU DLL: SARMCM3.DLL V5.25.2.0
Dialog DLL: DARMCM1.DLL V1.19.1.0
Target DLL: Segger\JL2CM3.dll V2.99.29.0
Dialog DLL: TARMCM1.DLL V1.14.0.0
This product is licensed to:
This product is protected by copyright law and international treaties. Unauthorized reproduction or distribution of this program, or any portion of it may result in severe civil and criminal penalties.
Keil Software, the Keil Software Logo, and µVision are registered trademarks of ARM Limited
XERCES,
an XML parser (xerces-c_3.0.dll) is licensed to you under the Apache Software License, Version 2.0.
Scintilla,
an editor tool (UvEdit.dll) which enables editing of source code is licensed to you under the Historical Permission Notice and Disclaimer, an open source license approved by the Open Source Initiative.
Copy Info
OK

图 2: Keil 版本

1.3.1 1 简介

PAN107x NDK 是基于开源蓝牙协议栈 Nimble(Host) 以及开源系统 FreeRTOS，以及私有 BLE Controller 实现完成。Nimble 和 FreeRTOS 均开发源码，BLE Controller 通过标准化 HCI 接口实现。需要注意的是 NDK 依赖的 IDE 主要是 KEIL。

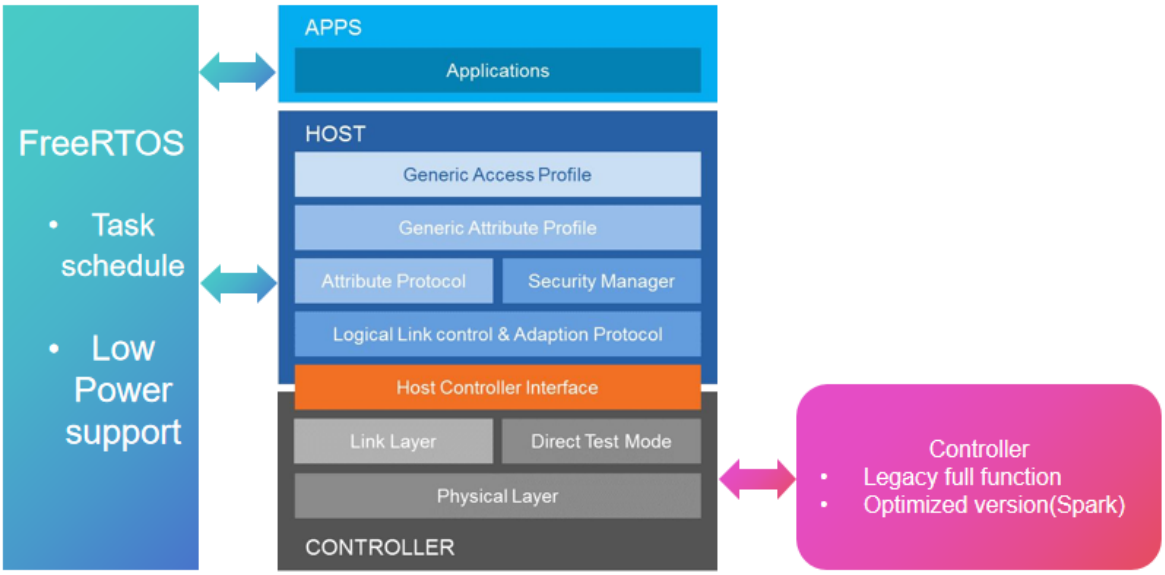


图 3: NDK 整体结构

1.3.2 2 NDK 目录结构

PAN107x NDK 源码树结构如下：

```
<home>/01_SDK
  modules
      hal
  nimble
      README.md
      controller
      host
      lib
      os
      samples
```

- `moudules/hal`：与 ZDK 同根同源，属于外设和 driver 相关的硬件抽象层
- `README.md`：nimble 基本介绍
- `controller`：nimble 工程所需要的 controller 相关头文件
- `host`：nimble host 协议栈所在的主要目录，同时包含 kv_store 组件，该组件主要用于 flash 数据库存储。
- `lib`：该文件包含两个版本 controller 的实现，具体实现以 lib 的形式添加到 keil 工程中。Controller 两个版本分别是 Origin 版本，该版本是全功能版本，但是具体实现优化比较少，执行速度较慢，代码相对较大；另外一个版本是优化版本 (Spark 版本)，该版本为精简和优化版本，速度更快功耗更低，但是目前主要实现 BLE 4.2+ 版本 feature，其他 5.0+feature 功能后续迭代升级。
- `os`:freertos 的相关代码
- `samples`: 基于 nimble 的相关 demo

2.1 modules

modules 在 NDK 中主要为外设以及 USB，2.4G 等依赖的库文件

```
<home>/01_SDK/modules
.
  hal
      panchip
          panplat
              pan1070
                  bsp
                      cmsis
                      device
                      peripheral
                      radio
                      usb
```

2.2 controller

controller 中主要包含了两个版本 controller 依赖的头文件

```
<home>/01_SDK/nimble/controller
.
  dummy.txt
  pan107x
      shrd_utils
  pan107x_spark
       include
```

2.3 host

host 中包含 nimble 的主体实现以及其他必须的组件，目前 nvs 数据库存储使用是的 kv_store 组件，主要用于存储 ble 的配对信息。当然开发者也可以用于自己的项目存储数据。

```
<home>/01_SDK/nimble/host
.
  kv_store
      mtb_init.c
      mtb_kvstore.c
      mtb_kvstore.h
      mtd_kv_port.h
  nimble
       CODING_STANDARDS.md
       LICENSE
       NOTICE
       README.md
       RELEASE_NOTES.md
       apps
       babblesim
       docs
       ext
       nimble
       porting
       repository.yml
       targets
       uncrustify.cfg
       version.yml
```

- `kv_store` 组件：
  - 支持任何可以建模为块设备的存储，包括内部闪存或外部闪存（例如通过 QSPI）。

  - 通过实例化库的多个实例来对存储进行分区。
  - 设计为抗电源故障。
  - 旨在促进存储的均匀磨损。

- `nimble`:

  参考 nimble 专页介绍，nimble 基于 V1.5.0 版本移植实现。

2.4 lib

lib 中主要包含了两个版本 controller 主体实现的库文件:

```
<home>/01_SDK/nimble/lib
.
  pan107x
      ble.lib
  pan107x_spark
      ble_spark.lib
```

Controller 两个版本分别是 `Origin` 版本，该版本是全功能版本，但是具体实现优化比较少，执行速度较慢，代码相对较大，所对应的库为 `pan107x/ble.lib`;

另外一个版本是优化版本，内部代号为 `Spark`, 所对应的库为 `pan107x_spark/ble_spark.lib`。该版本为精简和优化版本，速度更快功耗更低，但是目前主要实现 BLE 4.2+ 版本 feature，其他 5.0+feature 功能后续迭代升级。

2.5 OS

OS 目前只包含 freertos 的主体代码

2.6 samples

nimble 示例工程:

```
<home>/01_SDK/nimble/samples
.
  samples
      bluetooth
          ble_cent_prph
          ...
      solutions
          ble_hid_selfie
          ...
```

1.4 Nimble **简介**

NimBLE 是一个开源的蓝牙 5.1 协议栈（包括主机和控制器），其也是 Apache Mynewt 项目的一部分。PAN107x NDK 中使用的 NimBLE 版本为 v1.5.0。

特点:

- 支持 251 字节数据包长度。
- 支持 4 种角色并发工作：Broadcaster, Observer, Peripheral and Central。
- 支持 32 个连接并发工作。
- 支持 Legacy 和 SC (secure connections) SMP 配对和绑定。

- 支持扩展广播。
- 支持周期广播。
- 支持 Code phy 和 2M phy。
- 支持蓝牙 mesh[NDK 暂未测试对接]。

1.4.1 支持硬件

目前支持 PAN107x PAN101x 等系列芯片，同时 OS 使用 freertos。

1.4.2 概览

如果您在浏览源代码树，并且想要查看一些主要的功能块，这里有一些指引：

- nimble/controller: nrf 的部分 controller 实现，Pan10xx 封装于相关 lib 中。
- nimble/drivers: 射频相关收发 (Nordic nRF51 and nRF52)，Pan10xx 封装于相关 lib 中。
- nimble/host: 包含主机子系统的代码。这包括 L2CAP 和 ATT 等协议，支持 HCI 命令和事件，通用访问配置文件（GAP），通用属性配置文件（GATT）和安全管理器（SM）。
- nimble/host/mesh: 包含蓝牙 Mesh 子系统的代码。
- nimble/transport: 包含支持主机和控制器之间的传输协议的代码。这包括 UART、emSPI 和 RAM（在主机和控制器在同一 CPU 上运行时使用的组合构建）。
- porting: 包含支持的操作系统的 NimBLE 移植层（NPL）的实现。
- ext: 包含 NimBLE 使用的外部库。如果操作系统没有提供这些库，则会使用它们。

1.4.3 应用示例

还有一些示例应用程序，展示了如何使用 Apache Mynewt NimBLE 协议栈。如下:

- `ble_central`: 蓝牙主机示例，主要用于演示主机连接从机 ANS 报警通知服务. 该示例可以直接和 `bleprph_enc` 完成对测。
- `bleprph_hr`: 蓝牙从机示例，主要演示从机心跳服务.
- `bleprph_enc`: 蓝牙从机加密示例，主要演示从机特定特性读写时会进行加密配对服务. 可以和 `ble_central` 完成连接对测。但是 `ble_central` 并没有访问加密特性，所以不会触发加密配对流程。

1.4.4 API 接口

如果想在线了解 API 可以参考官方提供的 API 接口指南 NimBLE Host 去了解相关接口功能。

Chapter 2

硬件资料

文档列表：

2.1 PAN10xx EVB 介绍

2.1.1 1 概述

本文为 PAN10xx Evaluation Board (EVB) 开发板介绍，包括相关板级硬件模块、各模块在 EVB 板上的位置、以及对应电路原理图，旨在帮助开发者快速了解 PAN10xx EVB 开发板。

PAN10xx EVB 开发板由核心板、底板两大部分组成，其中：

- 核心板提供了 PAN10xx SoC 的最小系统，主要包含有 PAN10xx SoC 芯片、32MHz 高速晶振、32768Hz 低速晶振、复位按钮、板载天线以及一些必要的无源器件。
- 底板上提供了诸多 PAN10xx SoC 支持的外设模块，其中包含：
  - 电源管理系统、USB_Type-C 转串口模块、RGB 三色灯、三轴加速度传感器、外部 SPI FLASH、无源蜂鸣器、独立按键、可调电阻、红外模块、独立 LED 灯等等；
- 对外接口有 USB-Type-C 接口、USB_Type-C 转串口接口、鼠标接口，矩阵按键接口、OLED 显示屏接口、全 GPIO 测试接口等，如下图所示

2.1.2 2 开发板硬件资源

2.1 PAN10xx 最小系统

PAN10xx 最小系统由核心板和转接板组成。最小系统以模块形式嵌入开发板底板中，可分离式设计方便单独调试及应用于其它场景，如下图所示：

核心板搭载 PAN10xx 主控芯片、外部 32M 晶振、板载天线等，通过标准间距 2.54mm 双排针引出了所有 GPIO，PAN10xx 核心板原理图如下所示：

2.2 电源模块

PAN10xx EVB 开发板可选择使用以下两种供电方式之一：

- 5V 的 USB 供电；
- 3V 的 CR2032 纽扣电池供电；

EVB 开发板电源模块原理图如下：

EVB 开发板左侧有两个 USB-Type-C 接口 `U5` 与 `U7`，它们的电源引脚均与 EVB 的 5V 电源网络连在一起；除此之外，二者还有以下区别：

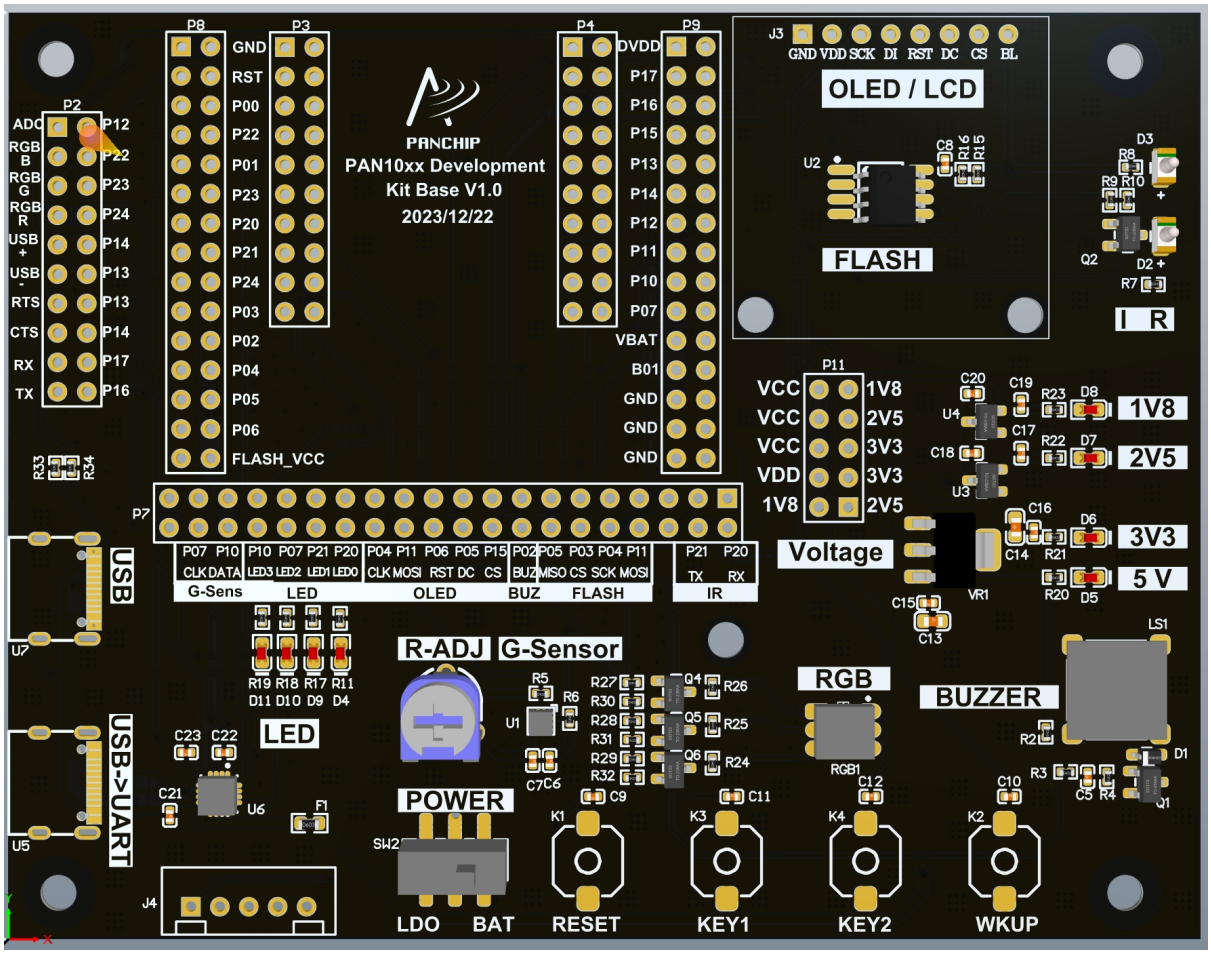


图 1: PAN10xx EVB V1.0 3D 图

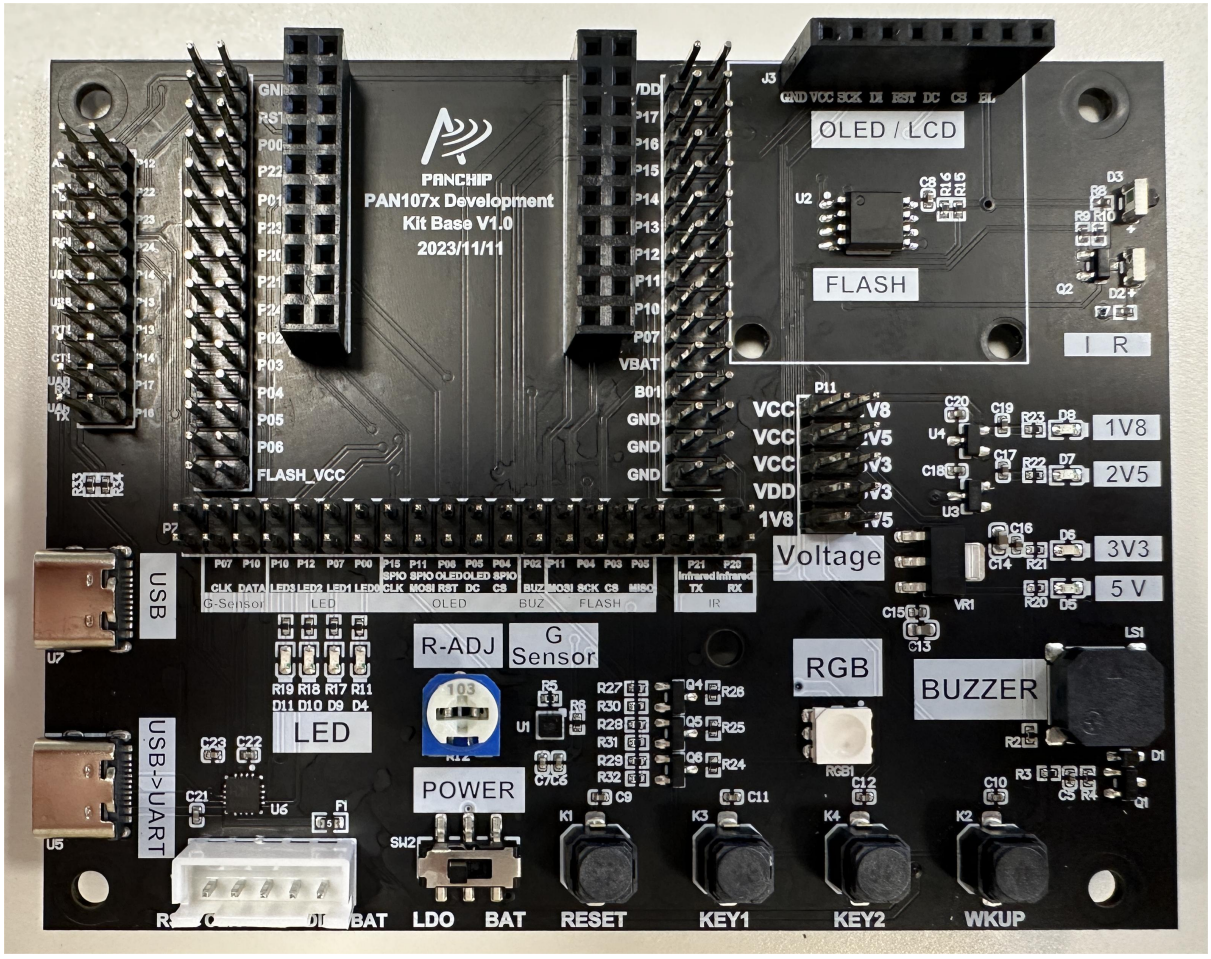


图 2: PAN10xx EVB V1.0 实物图

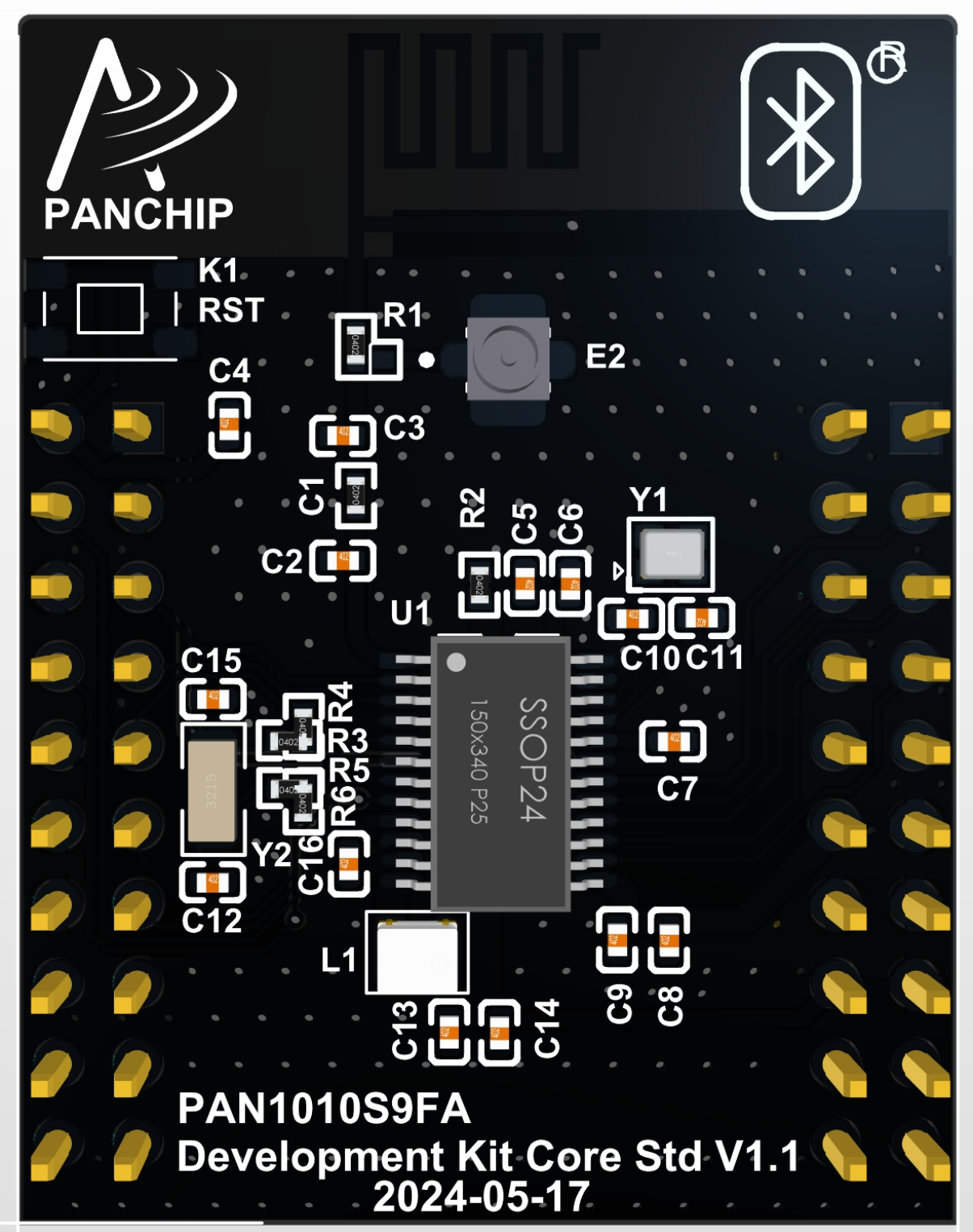


图 3: PAN1010S9FA 最小系统板三维图顶部

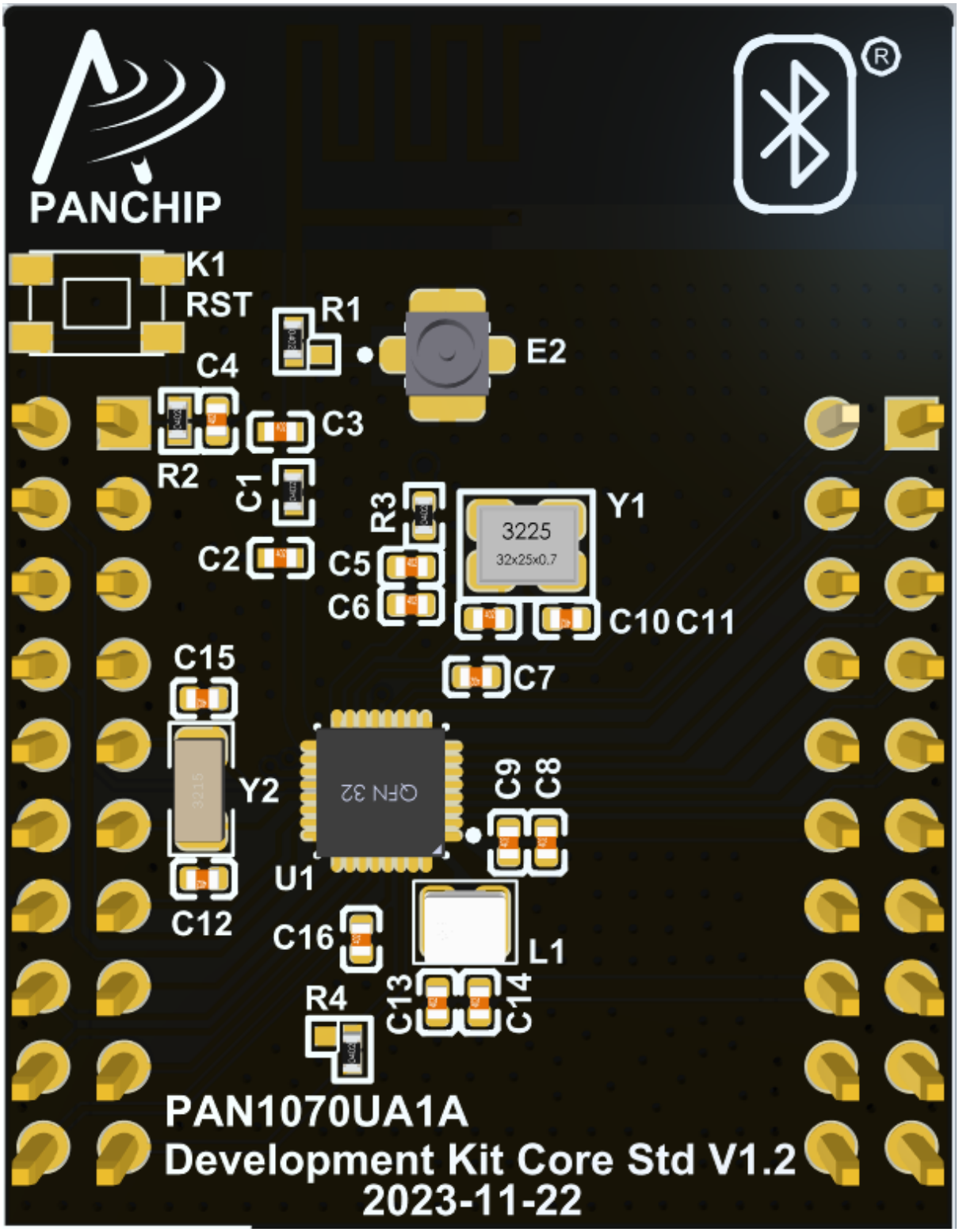


图 4: PAN1070UA1A 最小系统板三维图顶部

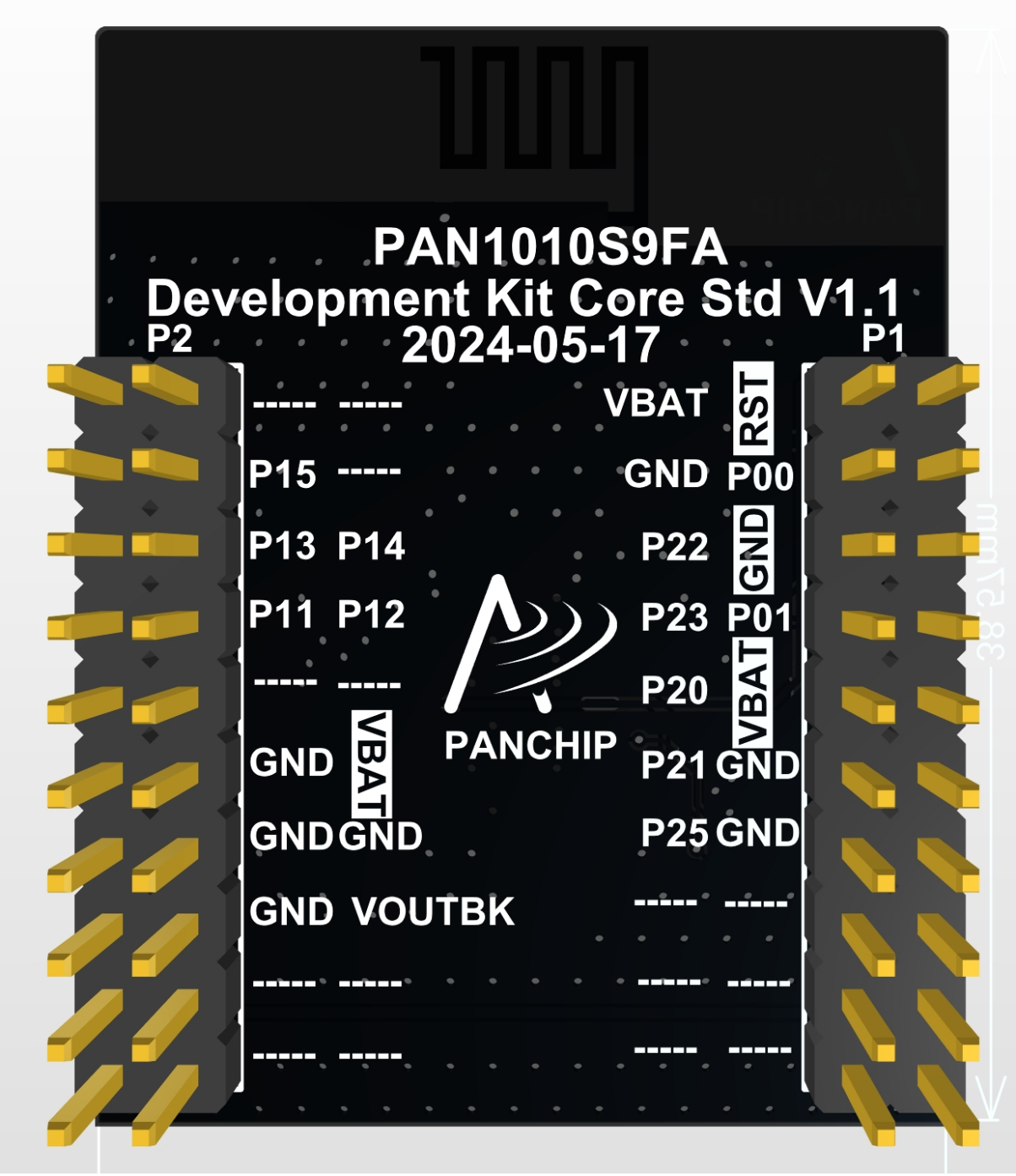


图 5: PAN1010S9FA 最小系统板三维图底部

图 6: PAN1070UA1A 最小系统板三维图底部

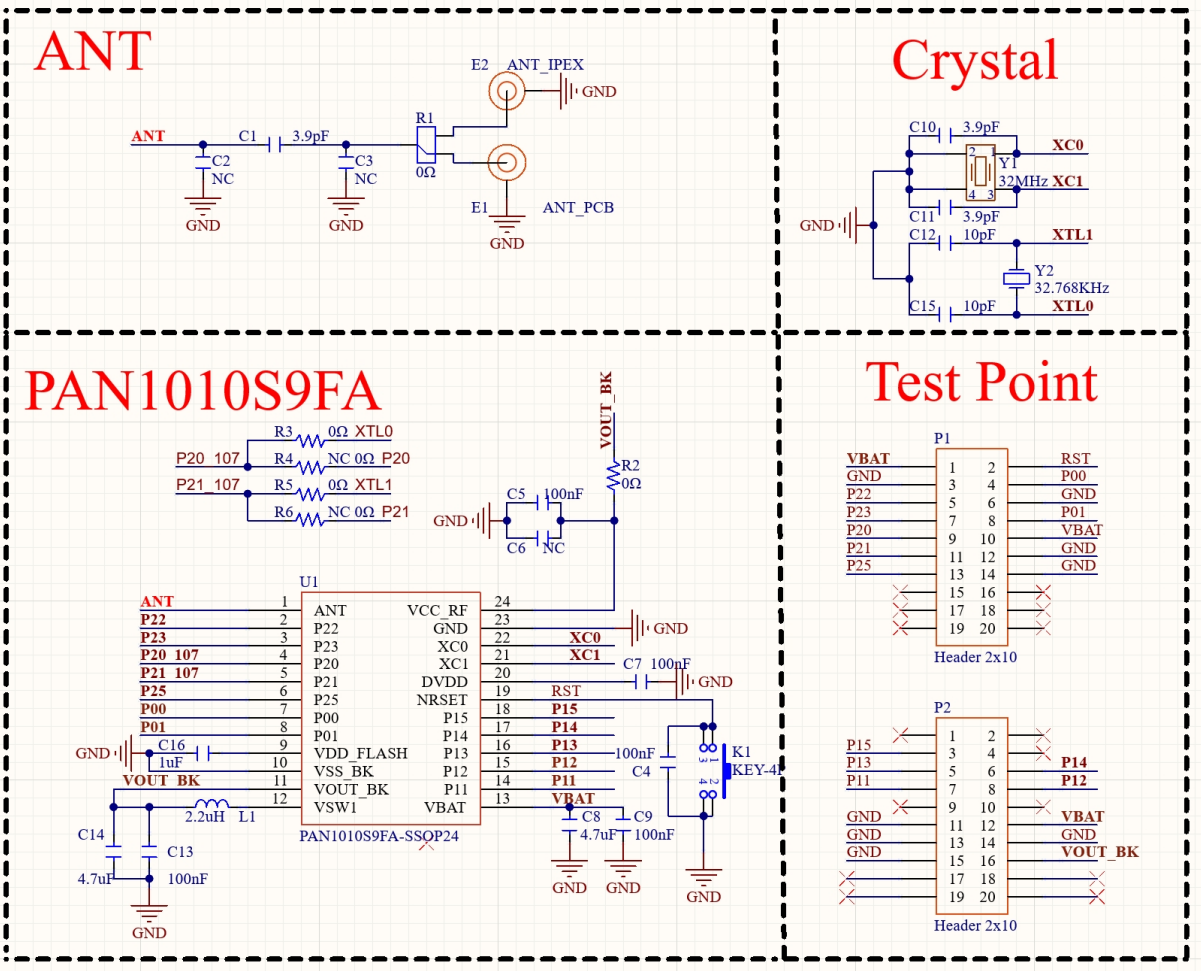


图 7: SSOP24 封装核心板原理图

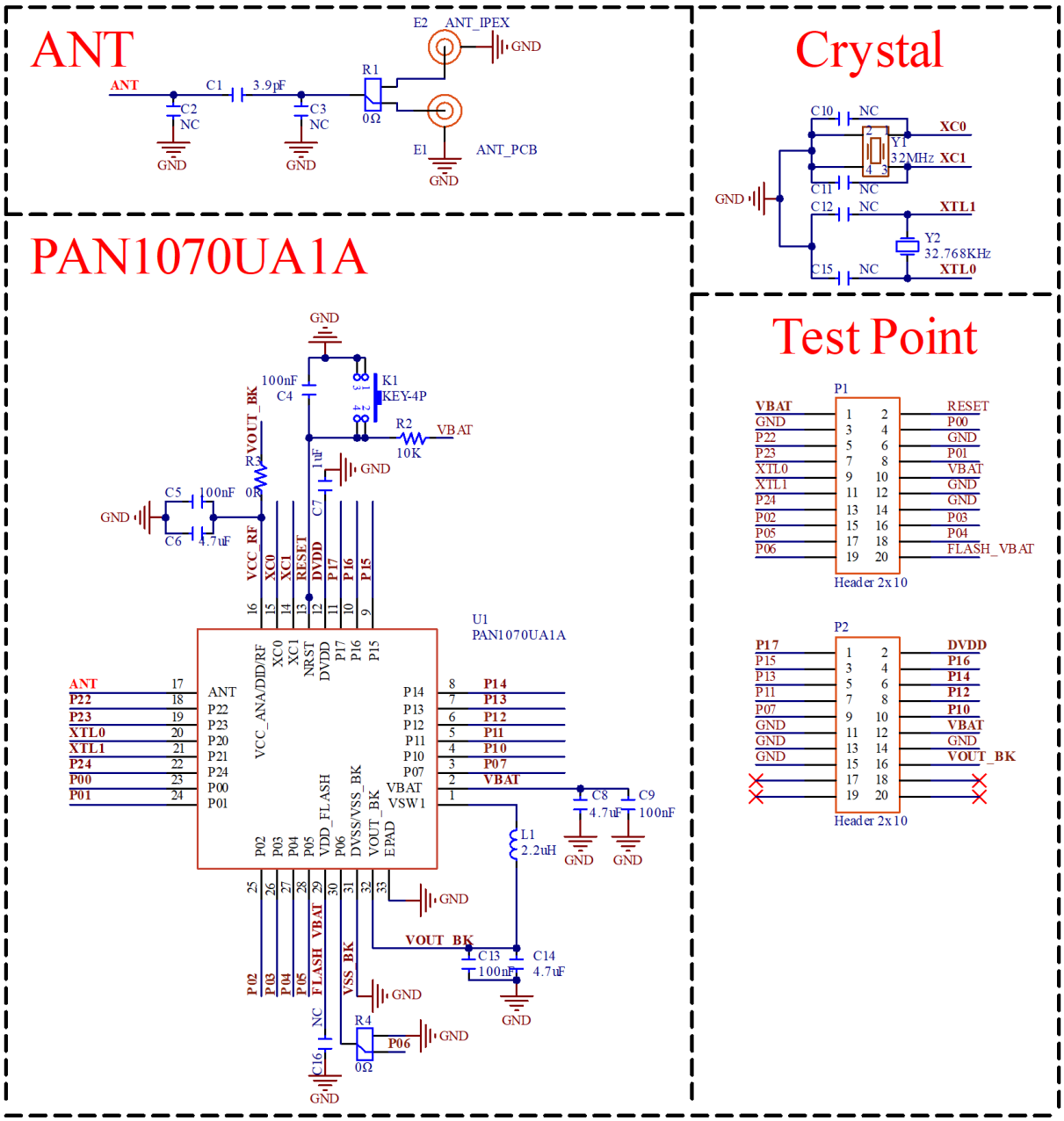


图 8: QFN32 封装核心板原理图

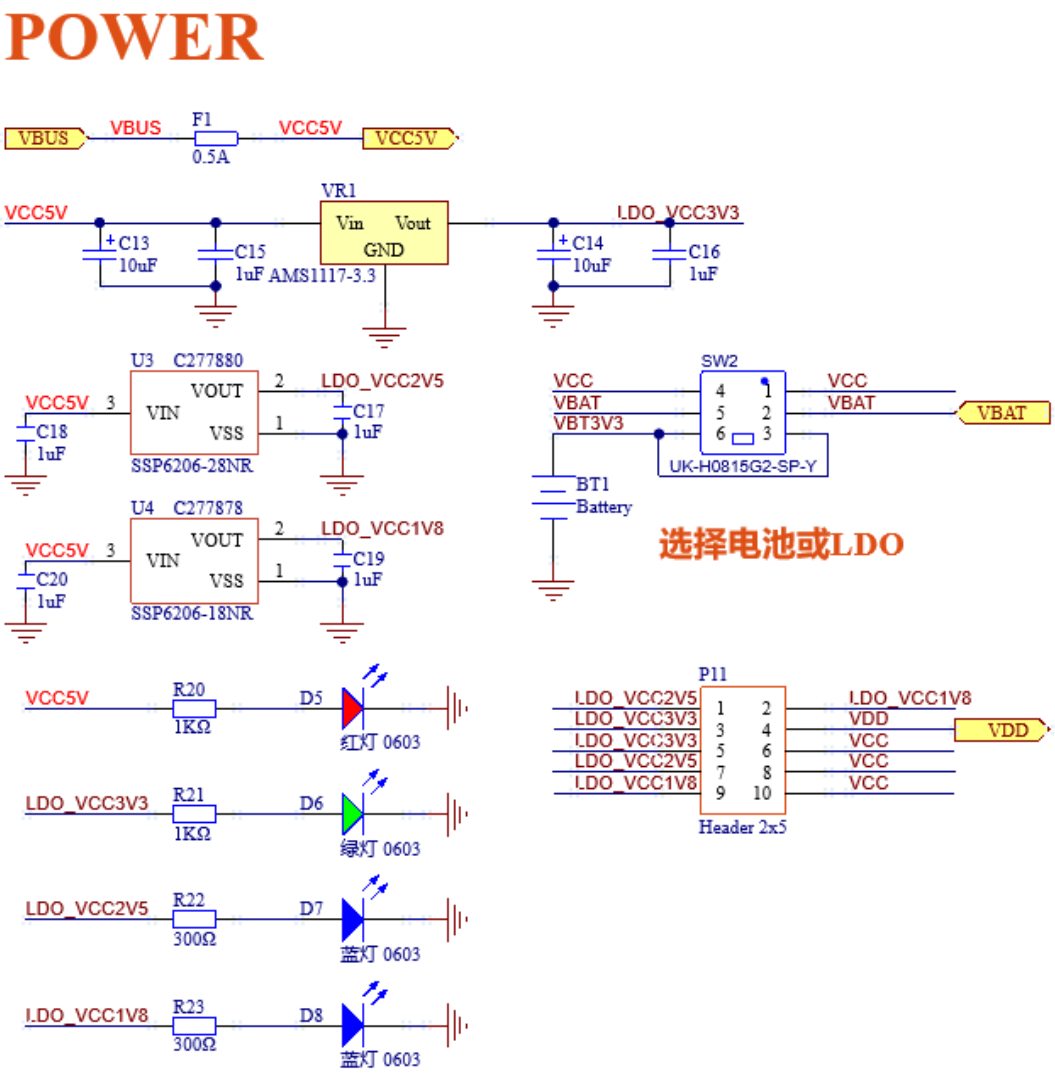

POWER
VBUS
VBUS
F1
0.5A
VCC5V
VCC5V
VR1
Vin
Vout
GND
AMS1117-3.3
LDO_VCC3V3
C13
10uF
C15
1uF
C14
10uF
C16
1uF
U3
C277880
VOUT
VIN
VSS
SSP6206-28NR
LDO_VCC2V5
C17
1uF
C18
1uF
U4
C277878
SSP6206-18NR
LDO_VCC1V8
C19
1uF
C20
1uF
SW2
VCC
VBAT
VBT3V3
UK-H0815G2-SP-Y
BT1
Battery
选择电池或LDO
R20
1KΩ
D5
红灯 0603
R21
1KΩ
D6
绿灯 0603
R22
300Ω
D7
蓝灯 0603
R23
300Ω
D8
蓝灯 0603
P11
Header 2x5
VDD

图 9: 电源模块原理图

- U7 为普通 USB 接口，使用跳线帽将 EVB 底板的 USBDM/USBDP 分别与 SoC 的对应引脚相连，即可使用 PAN10xx SOC 的 USB 模块；
- U5 为 USB 转 UART 模块的 USB 端接口，使用跳线帽将 EVB 底板的 TXD/RXD 分别与 SoC 对应的引脚相连，即可通过 USB 转串口模块，实现 PAN10xx SOC 的 UART0 与 PC 进行通信；

  注：在实际使用过程中，选用任意一个 USB 口供电即可，但需注意，板载电源切换开关应拨动至 LDO 档位。

一种典型的供电方法如下图所示：

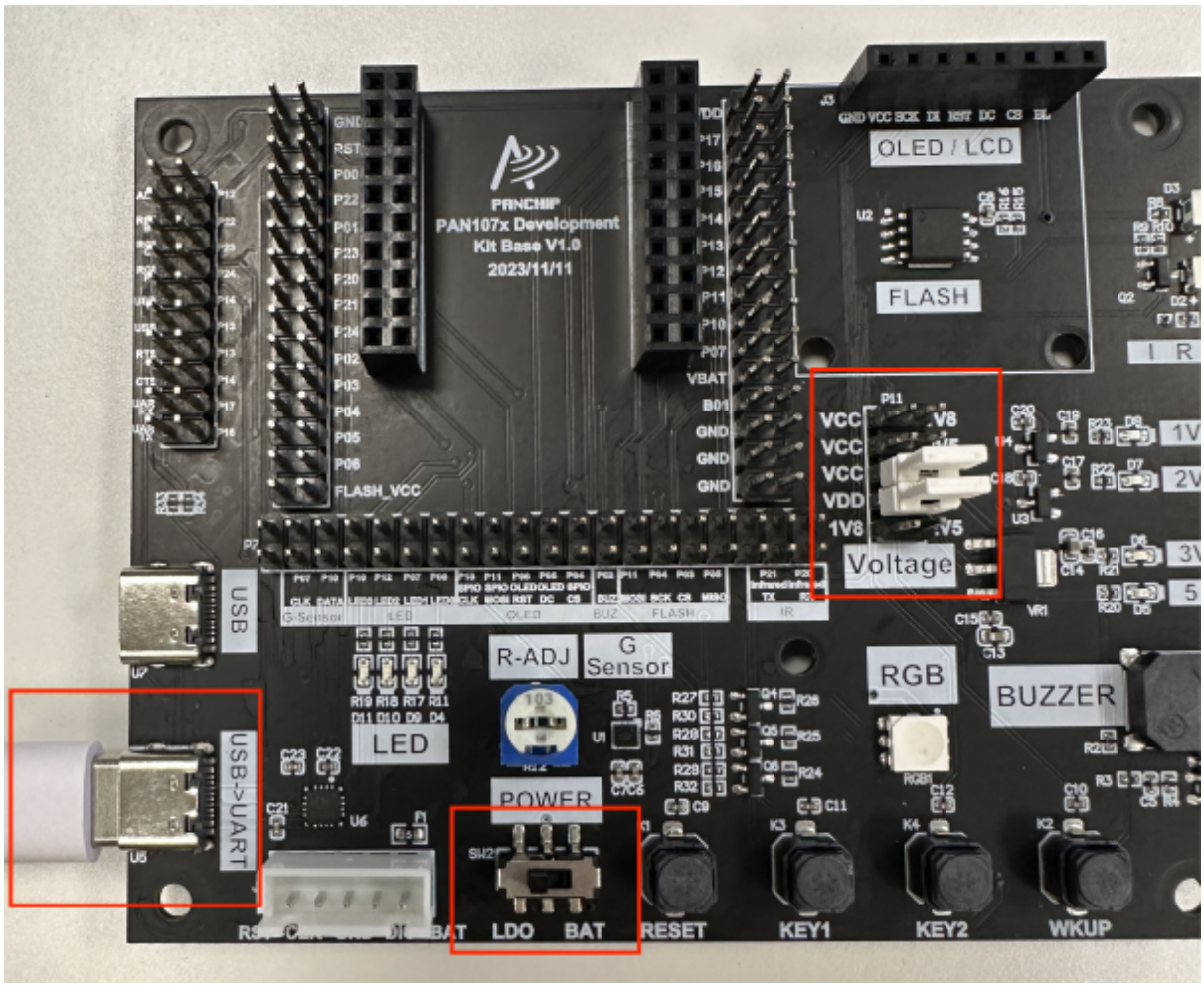


图 10: 供电方式示意图

其中：

1. 拨动开关 SW2 左拨到 LDO 档位；
2. 使用跳线帽短接图右侧排针，作用是将电源分别连接至 VCC（即核心板系统电源）、VDD（即开发板外设供电），电压分别有 3.3V、2.5V、1.8V 可选。

   电源网络的三个 LDO 模块，分别输出 3.3V、2.5V、1.8V，故 VCC（核心板系统电源）、VDD（开发板外设供电）可以各自分别选择所需要的电压。

另外，若希望开发板由左下角纽扣电池供电，则拨动开关 SW2 往右拨到 BAT 档位即可。

2.3 SWD **调试接口**

开发板提供了单排针接口用于连接 J-LINK 实现 SWD 调试和程序下载功能，该排针接口位于整板左上方。

一种典型的使用方法如下图所示：

1. JLINK 下载器插到 SWD 接口 `J4`；
2. 供电方式参考上文。

2.4 USB **转串口模块**

PAN10xx SoC 的 `P16`、`P17` 引脚可通过软件配置成 UART 串口功能，然后通过 CH343 模块转为 USB_Type-C 接口。

USB 转 UART 模块使用 U5 USB_Type-C 接口，并且使用跳线帽短接排针对应的引脚，如下图所示 (**流控还需用跳线帽将** `CTS0`、`RTS0` **连接到** SOC)：

2.5 RGB **灯**

开发板搭载单颗 RGB 灯，可由芯片的三个 IO 通过晶体管控制，实现亮灭或渐变等效果。

为使用此模块功能，需要将跳线帽短接排针上对应的三对引脚，如下图所示：

2.6 USB **模块**

开发板提供一个 USB_Type-C 接口，原理图如下:

为使用此模块功能，需要将跳线帽短接排针上对应的引脚，如下图所示：

2.7 **运动传感器**

开发板搭载了三轴加速度计传感器 SC7A20，提供了 IIC 接口与主控芯片进行通信，IIC 通信地址为: 0X18，原理图如下：

为使用此模块功能，需要将跳线帽短接排针上对应的引脚，如下图所示：

2.8 OLED **显示屏**

开发板搭载了常见的 0.96 寸、七针接口、128*64 分辨率的 OLED 显示屏模块接口，OLED 模块使用 SSD1306 显示驱动芯片进行控制，具备内部升压，对外默认提供三线 SPI 接口与主控芯片进行通信，其原理图如下：

使用此模块功能之前，需要使用跳线帽短接 OLED 功能对应的排针，如下图所示：

> **注**：OLED 显示屏模块与板载 SPI FLASH 芯片共用主控芯片的 SPI 接口。

2.9 **外部** SPI FLASH

开发板搭载了具备 1MB 存储空间的外部 FLASH 芯片 GD25WQ80，该芯片与板载显示屏模块共用主控芯片的 SPI 接口，其原理图如下：

为使用此模块功能，需要将跳线帽短接排针上对应的引脚，如下图所示：

2.10 **蜂鸣器**

开发板搭载了贴片无源蜂鸣器电路用于声音提示、报警等功能，可由 PAN10xx SoC 通过 PWM 输出 2KHz~3KHz 频率的方波控制发声，其原理图如下：

为使用此模块功能，需要将跳线帽短接排针上对应的引脚，如下图所示：

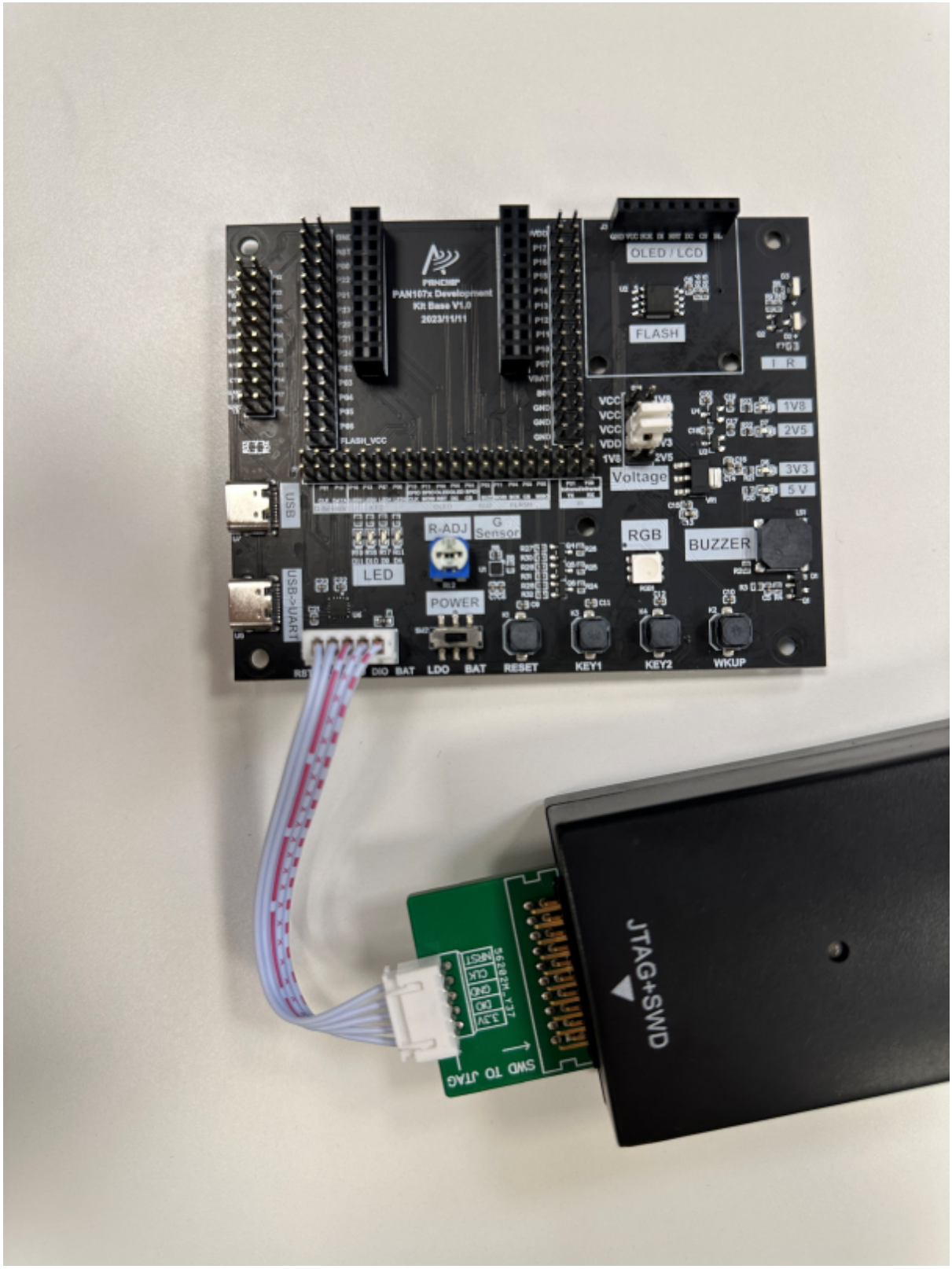

PAN107x Development
Kit Base V1.0
2023/11/11
OLED / LCD
FLASH
I R
VCC
VCC
VCC
VDD
1V8
2V5
Voltage
1V8
2V5
3V3
5 V
USB
USB-UART
R-ADJ
G Sensor
LED
POWER
RGB
BUZZER
RESET
KEY1
KEY2
WKUP
LDO
BAT
JTAG+SWD
SWD TO JTAG

图 11: JLink 连线

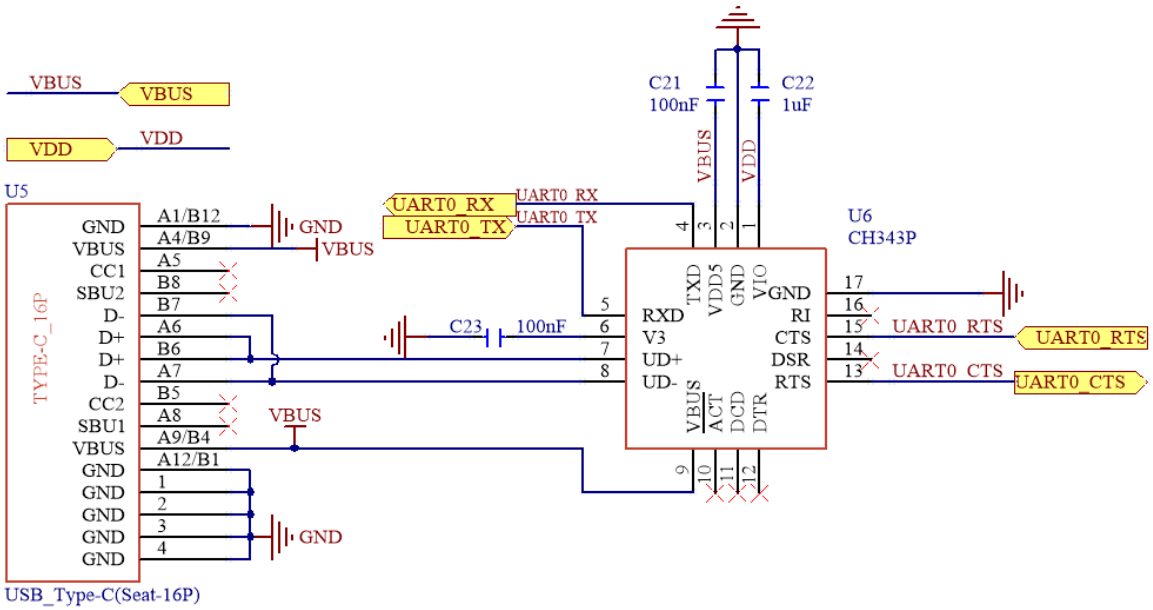


图 12: USB 转 UART 模块原理图

2.11 可调电阻

开发板搭载了一个最大阻值为 10K 的可调电阻与 0 欧姆精密电阻串联接入电源电路，在 LDO 提供 VDD 电源的系统中，可在 ADC 采样点产生 0V~ VDDV 的可调电压，用于测试 PAN10xx SoC 的 ADC（模数转换器）采样功能，其原理图如下：

为使用此模块功能，需要将跳线帽短接排针上对应的引脚，如下图所示：

2.12 轻触按键

开发板底板配备了 4 个按键：2 个普通 GPIO 按键、1 个低功耗唤醒按键和 1 个复位按键。其中：

- 按键 `K1` 可通过跳线帽连接至 PAN10xx SoC 的 `RST` 引脚，用于控制芯片复位；
- 按键 `K2` 可通过跳线帽连接至 PAN10xx SoC 的 `P02` 引脚，在 PAN10xx SoC 处于待机（Standby）模式下时，P56 引脚可被配置为低功耗唤醒引脚。
- 按键 `K3`、`K4` 连接至 PAN10xx SoC 的 GPIO 口 `P06`、`P12`，当做普通按键使用；

按键，原理图如下：

为正常使用所有按键，需要将 GPIO 内部上拉电阻开关打开，PCBA 如下图所示：

2.13 独立 LED 灯

开发板底板配备了 4 独立 LED 灯，原理图如下：

为使用此模块功能，需要将跳线帽短接排针上对应的引脚，如下图所示：

2.14 红外模块

开发板搭载了一个红外收发模块，包括一个发射电路和一个接收电路，其原理图如下：

为使用此模块功能，需要将跳线帽短接排针上对应的引脚，如下图所示（可选择只短接需要使用的 LED 灯，不需要全部短接）：

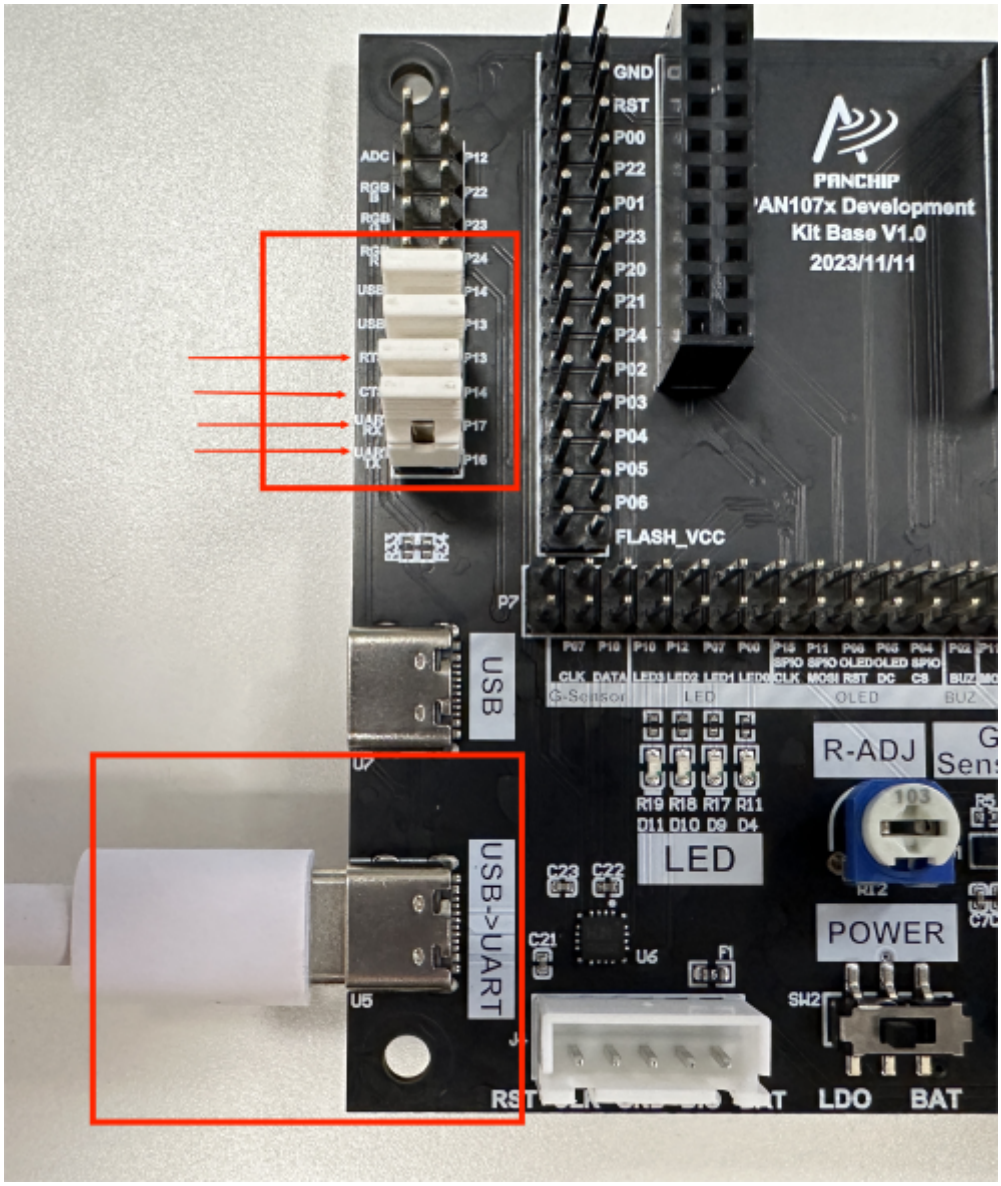


图 13: USB 转 UART 模块实物接线图

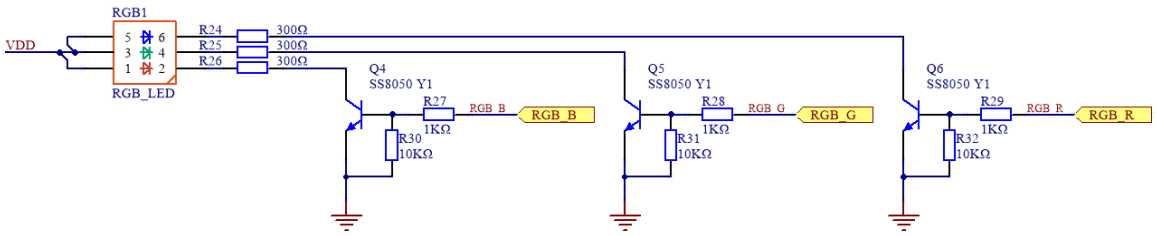


图 14: RGB 灯原理图

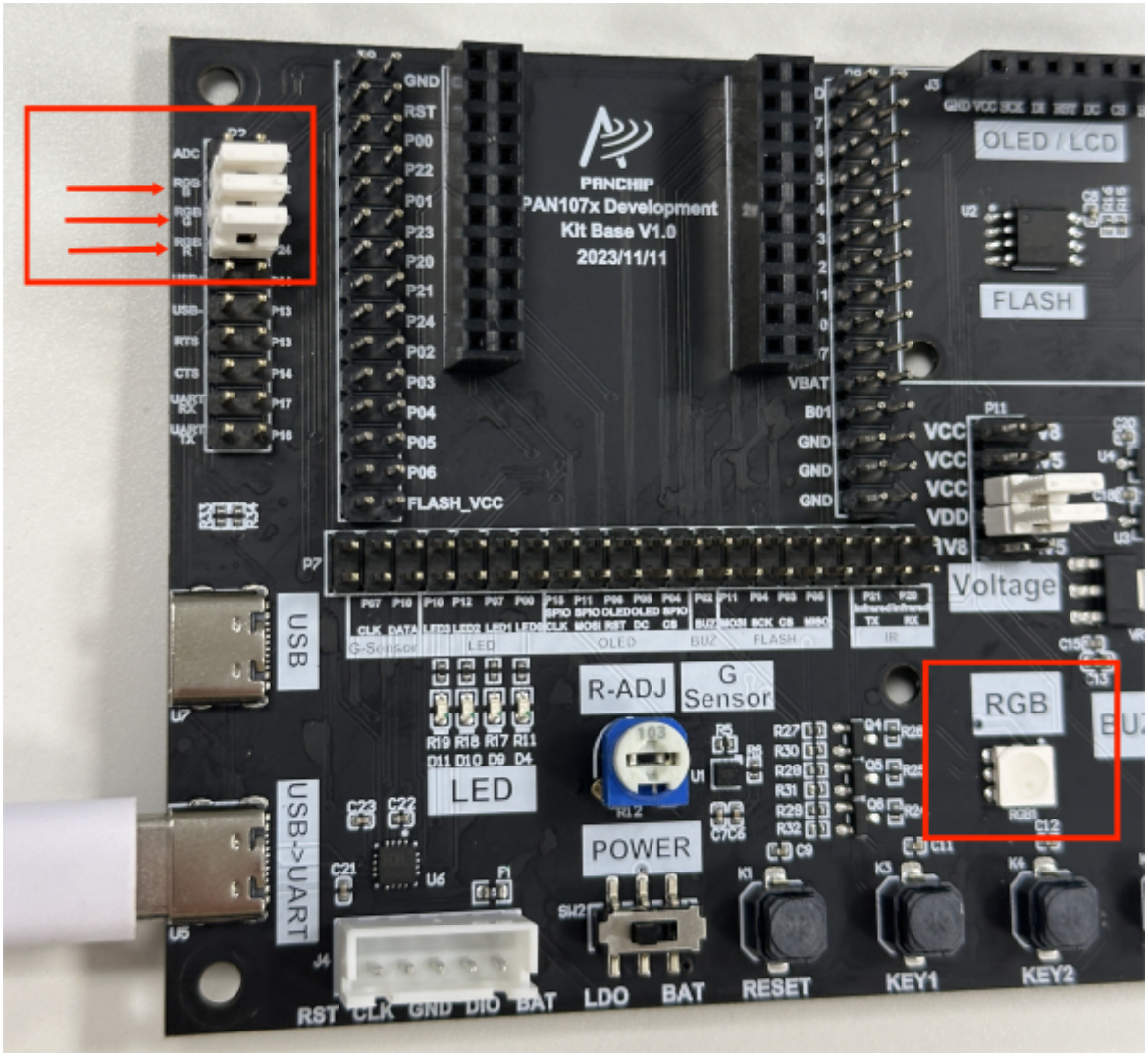


图 15: RGB 灯实物图

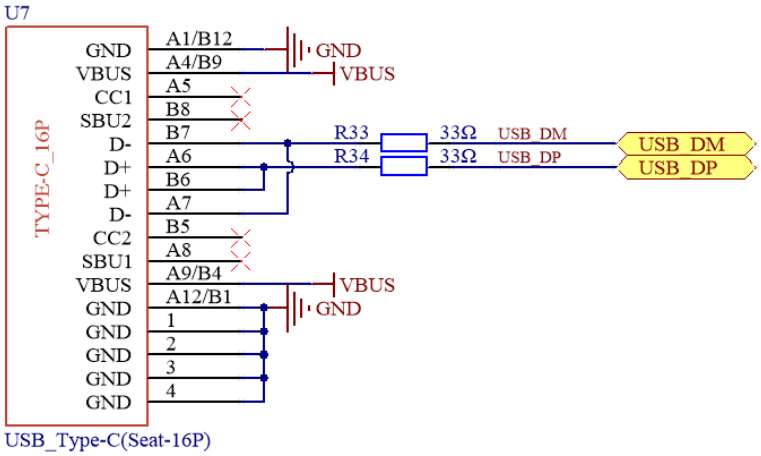


图 16: USB 模块外围电路原理图

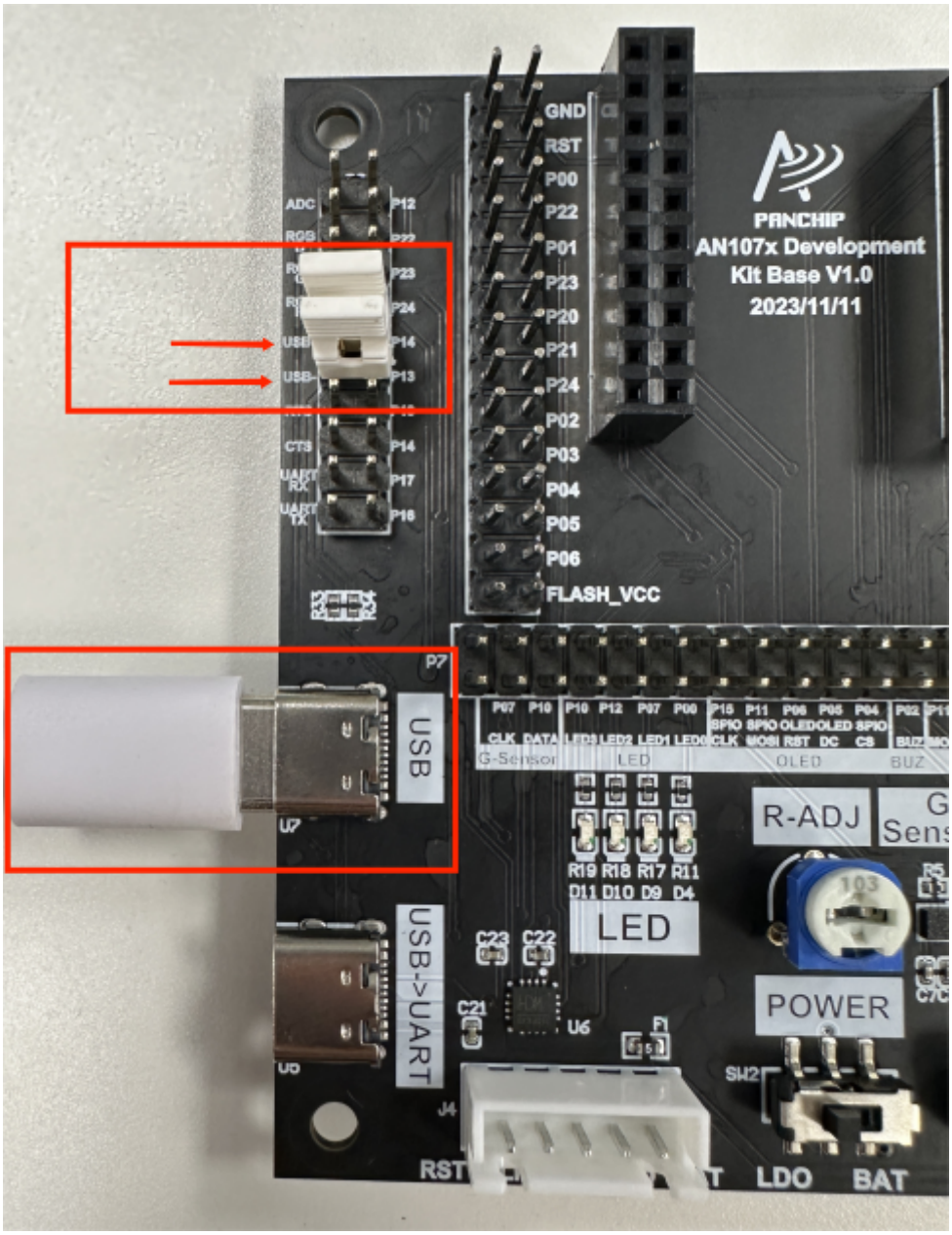


图 17: USB 模块外围电路实物图

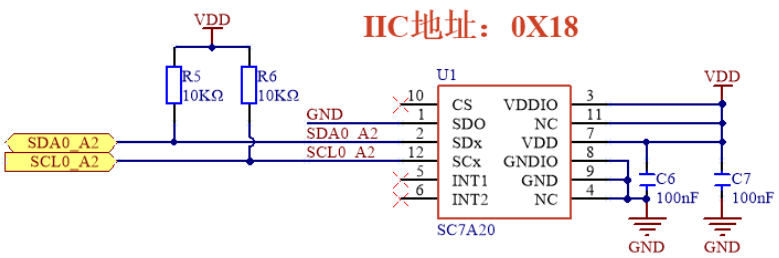


图 18: G-Sensor 模块原理图

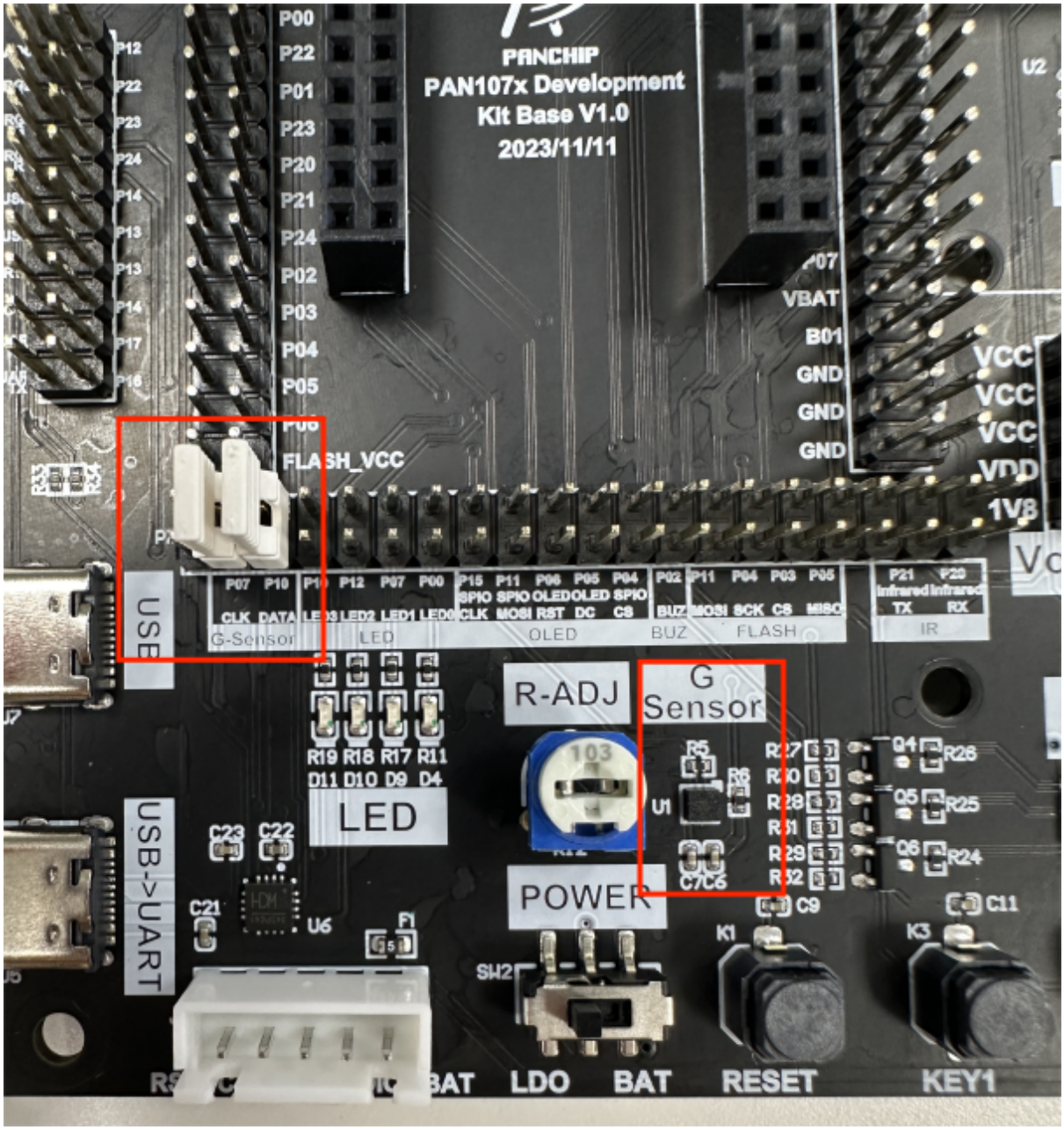


图 19: G-Sensor 模块实物接线图

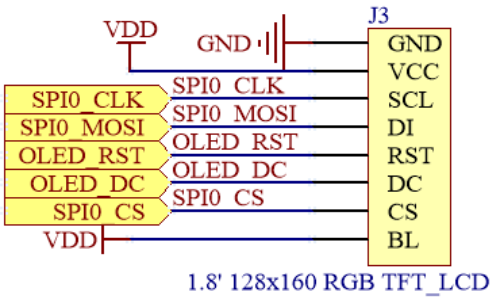


图 20: OLED 模块原理图

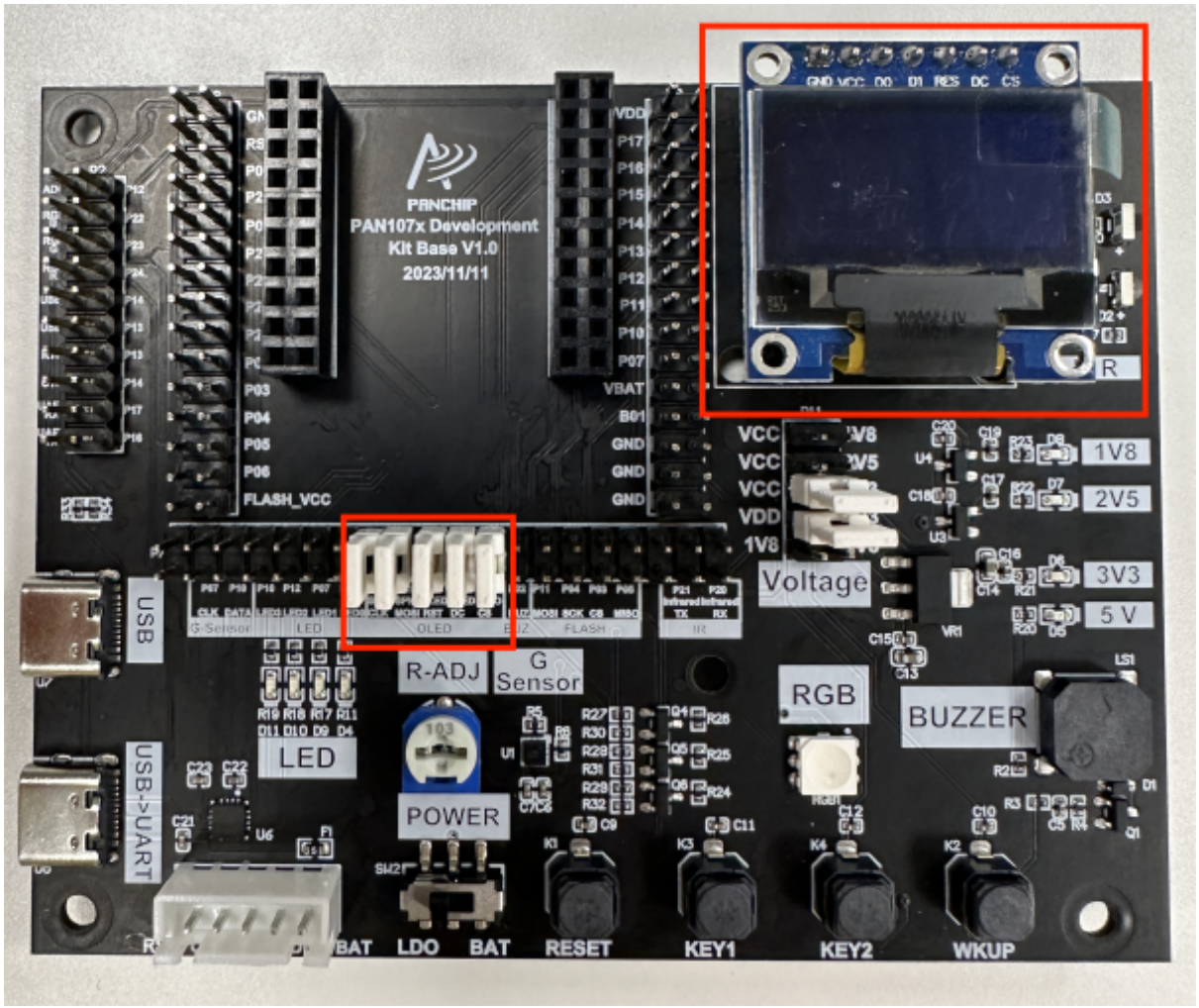


图 21: OLED 模块接线实物图

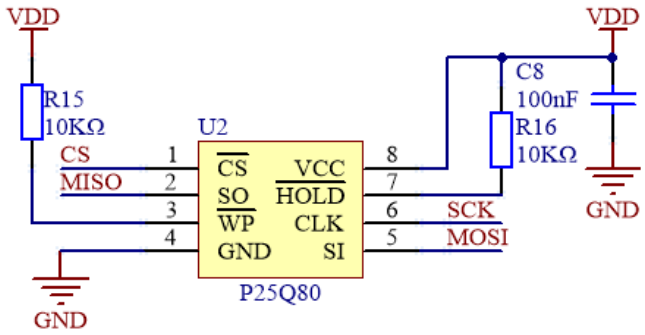


图 22: 外部 SPI Flash 模块原理图

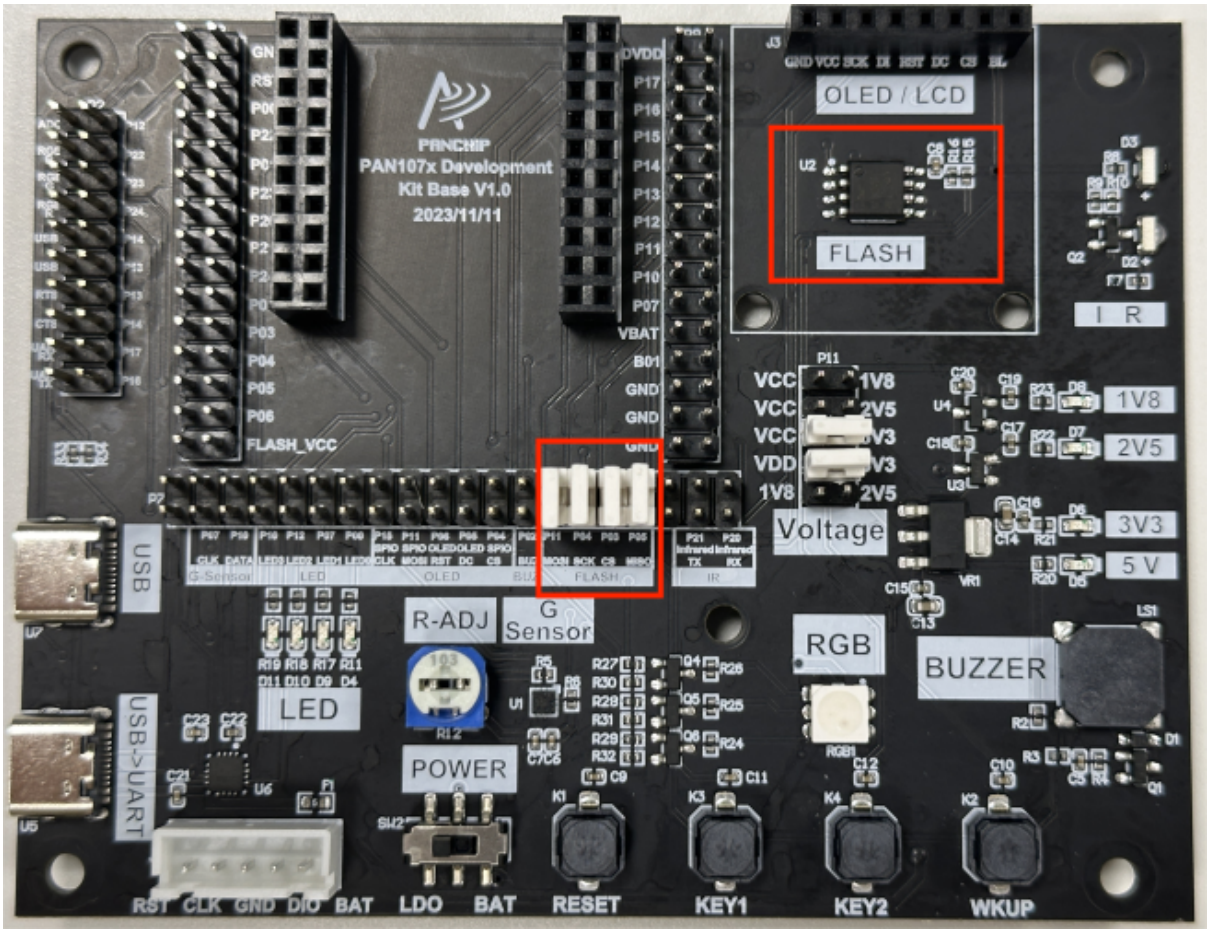


图 23: 外部 SPI Flash 模块实物接线图

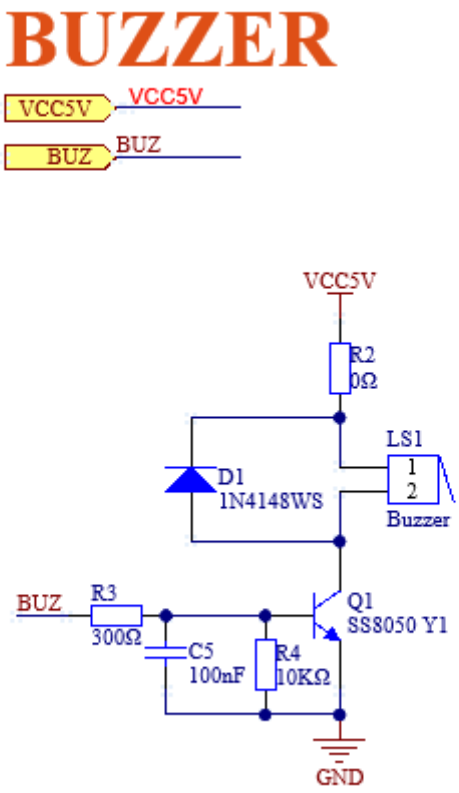


图 24: 蜂鸣器模块原理图

2.2 PAN10xx 硬件参考设计

2.2.1 1 概述

本文档主要介绍 PAN10xx 系列芯片方案的硬件原理图设计、PCB 设计建议、天线设计。本文档提供 PAN1070UA1A 芯片的硬件设计方法。

2.2.2 2 原理图设计建议

2.1 PAN1010S9FA 硬件参考设计原理图

2.2 PAN1070UAEC 硬件参考设计原理图

2.3 PAN1070UA1A 硬件参考设计原理图

如上图电路系统由电源去耦电容、DC-DC 降压、晶振电路、天线匹配网络组成。

2.3 电源

- VBAT 为芯片电源脚，要求供电能力不小于 60mA，供电范围为 1.8V–3.6V 。
- VBAT、VCC_RF、VOUT_BK 电源相关引脚需要至少预留 1 个电容，预留一大一小 2 个电容更佳。电容推荐为 4.7uF 和 100nF。
- 电容靠近芯片引脚摆放，电容焊盘和芯片焊盘之间最大距离不超过 5mm。请遵循指导要求，否则易引起 DC-DC 带不起 RF 以及 EVM 异常。

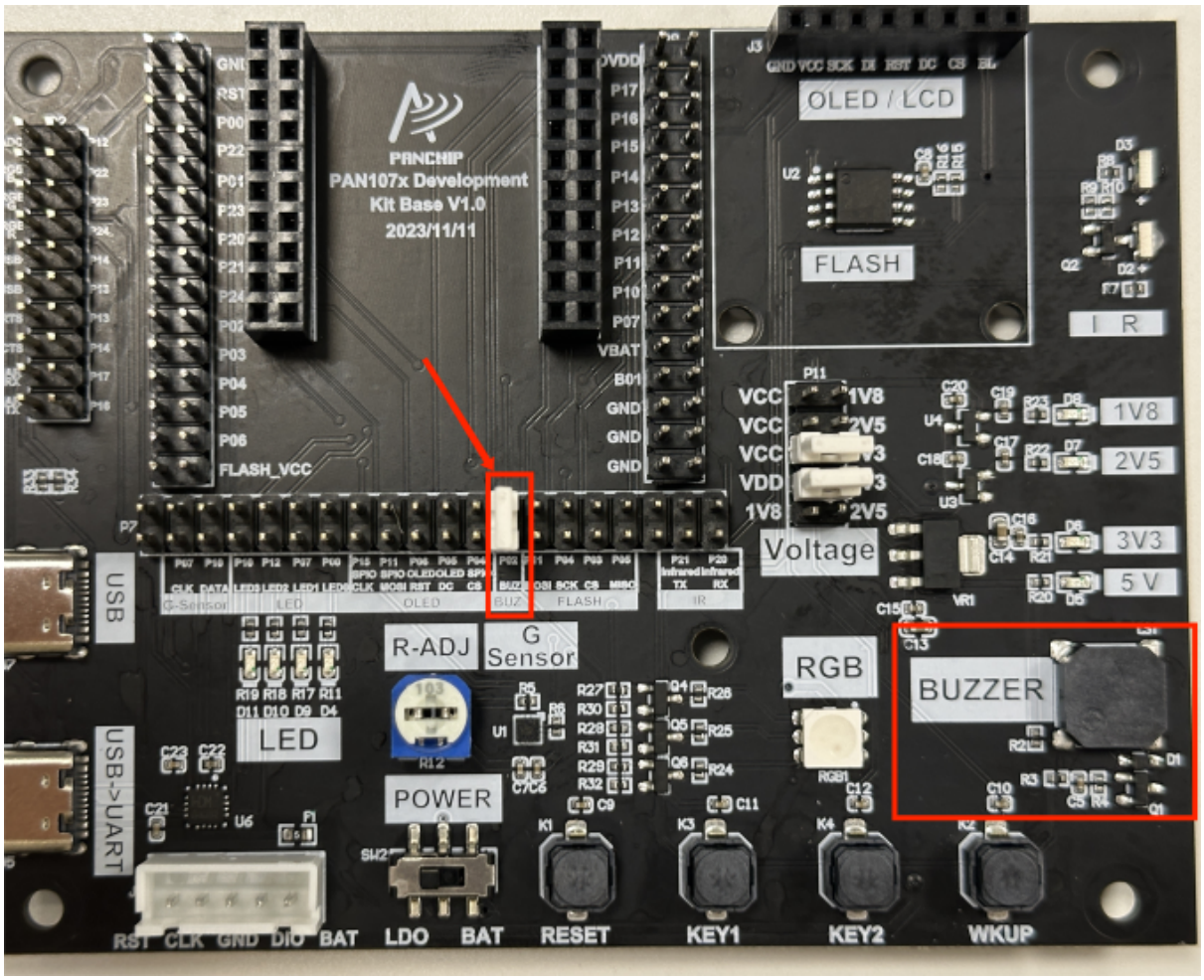


图 25: 蜂鸣器模块实物接线图

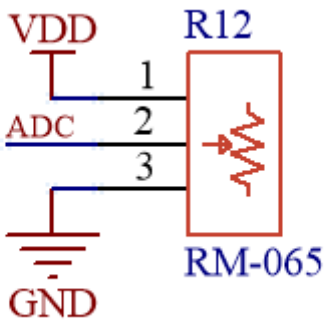


图 26: 可调电阻原理图

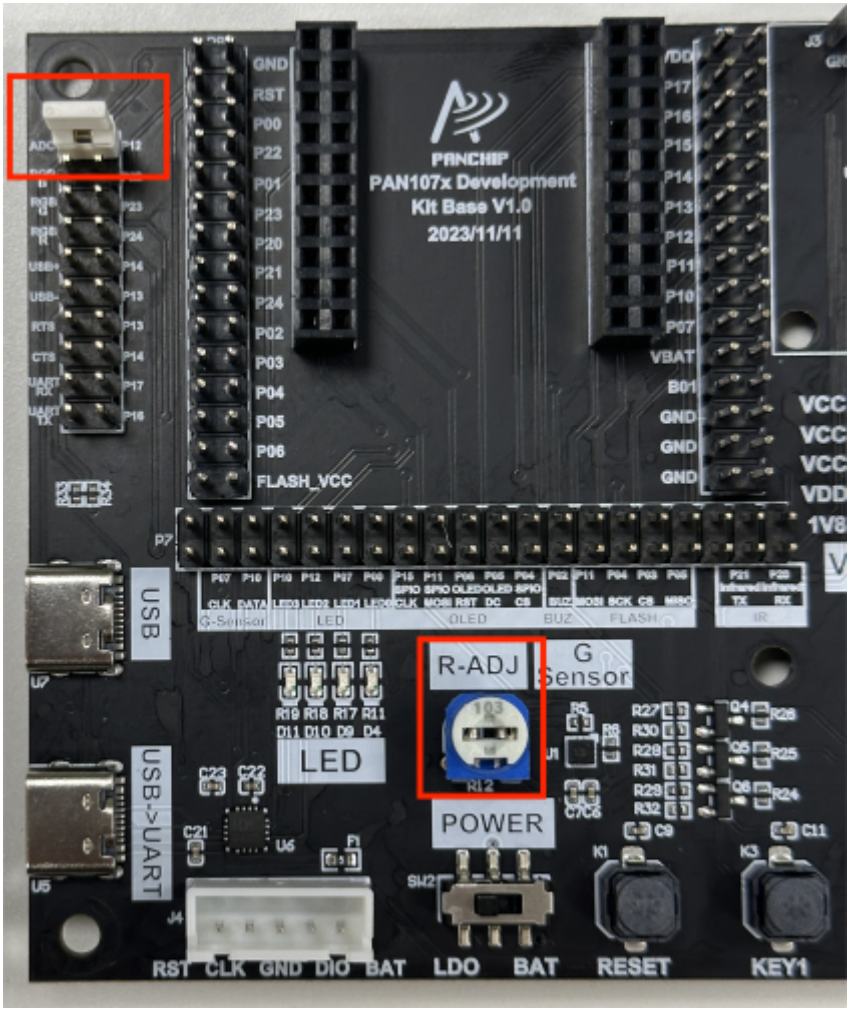


图 27: ADC 模块示例接线图

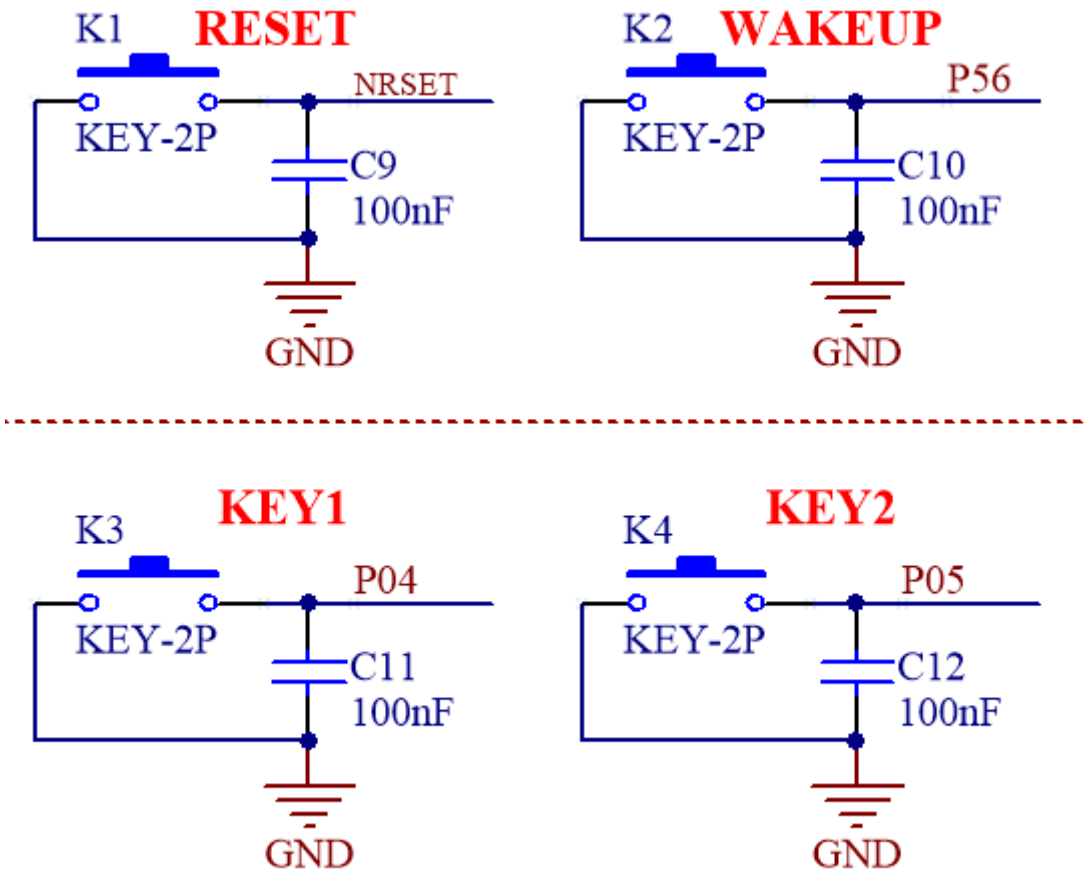


图 28: 按键原理图

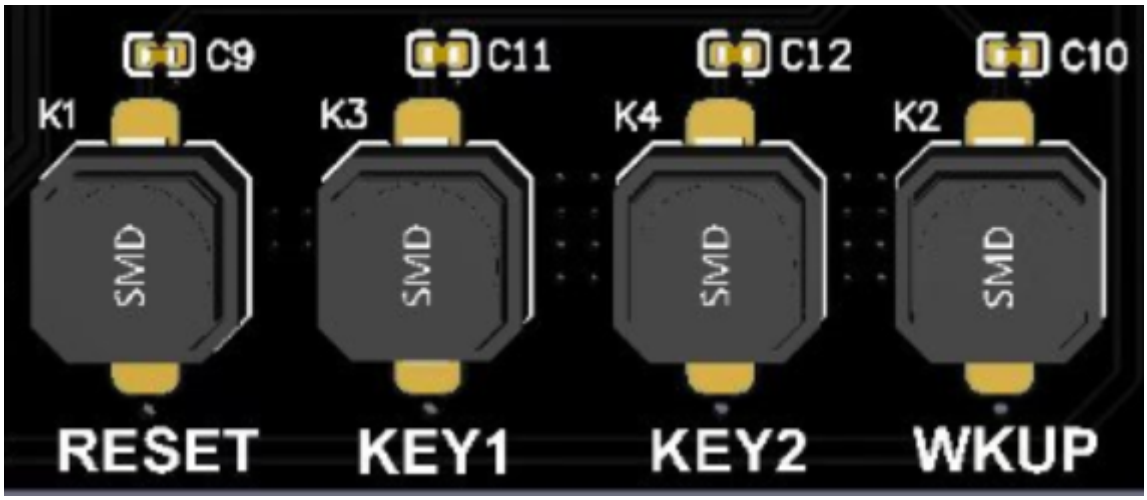


图 29: 按键图

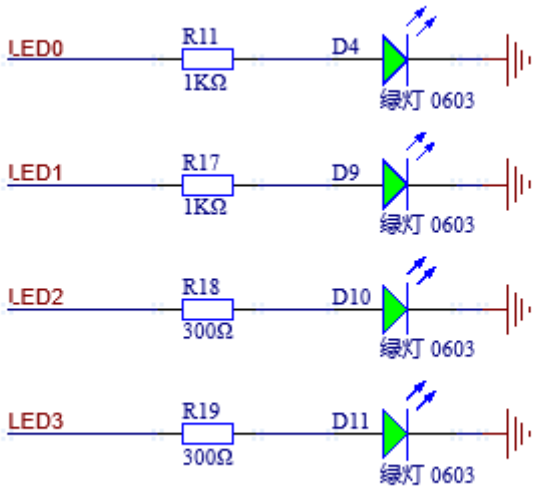


图 30: LED 灯模块原理图

2.3.1 DC-DC　DC-DC 芯片外围电路

1. PAN10x 系列芯片 DC-DC 外围电路组成为：2.2 H 电感、100nF 电容、4.7 F 电容。
2. L1 推荐型号：PIM252010-2R2MTS00。选择功率电感，2.2 H，**最小峰值电流为** 150mA，DCR **不超过** 80mΩ，未满足要求在 DC-DC 模式可能会造成 RF 功能异常。
3. DCR 过大会影响 BUCK 效率，能量会转化成热量损耗掉，DC-DC 输出的驱动电流是有限的，效率越低，能够供给到芯片的有效能量就越少。

- DC-DC 的两种工作模式:

1. 开启 DC-DC 模式可以降低系统功耗。
2. 开启 LDO（Bypass）模式后芯片内部将 VBAT 连接到 VSW1，这时 VSW1 处的 2.2uH 电感作用为一段导体，可以用 0Ω 电阻替换。
3. DC-DC、LDO 两种模式不能同时开启。
4. 在不考虑功耗的前提下，可将 VCC_RF 直接连接到 VBAT，此时应将电源模式设置为 LDO 模式。

- DC-DC 相关引脚说明:

1. VBAT 为 DC-DC 的供电引脚。
2. VSW1 为 DC-DC 的功率开关（P-MOS）漏极输出引脚，功率电感应靠近该引脚放置。
3. VOUT_BK 为 DC-DC 的反馈引脚，电容应靠近该引脚放置。
4. VSS_BK 为 DC-DC 电源的 GND 引脚。

2.3.2 DVDD **电容**　芯片 DVDD 管脚**推荐放置** 100nF **电容**。电容最大不超过 1uF，否则会影响芯片正常启动，影响功耗，电容应靠近该引脚放置。

注意：为避免电路异常，该电容容值请不要随意更改！

2.3.3 VDD_FLASH **电容**　芯片 VDD_FLASH 管脚**推荐放置** 1uF **电容**, 电容应靠近该引脚放置。

注意：为避免电路异常，该电容容值请不要随意更改！

2.3.4 VCC_RF　VCC_RF 外部需要接一级 RC 滤波器并尽量靠近该引脚。R3 一般为 0 Ω、C6 为 4.7uF。请遵循先 R 后 C，电容摆放位置距离芯片引脚不超过 5mm。

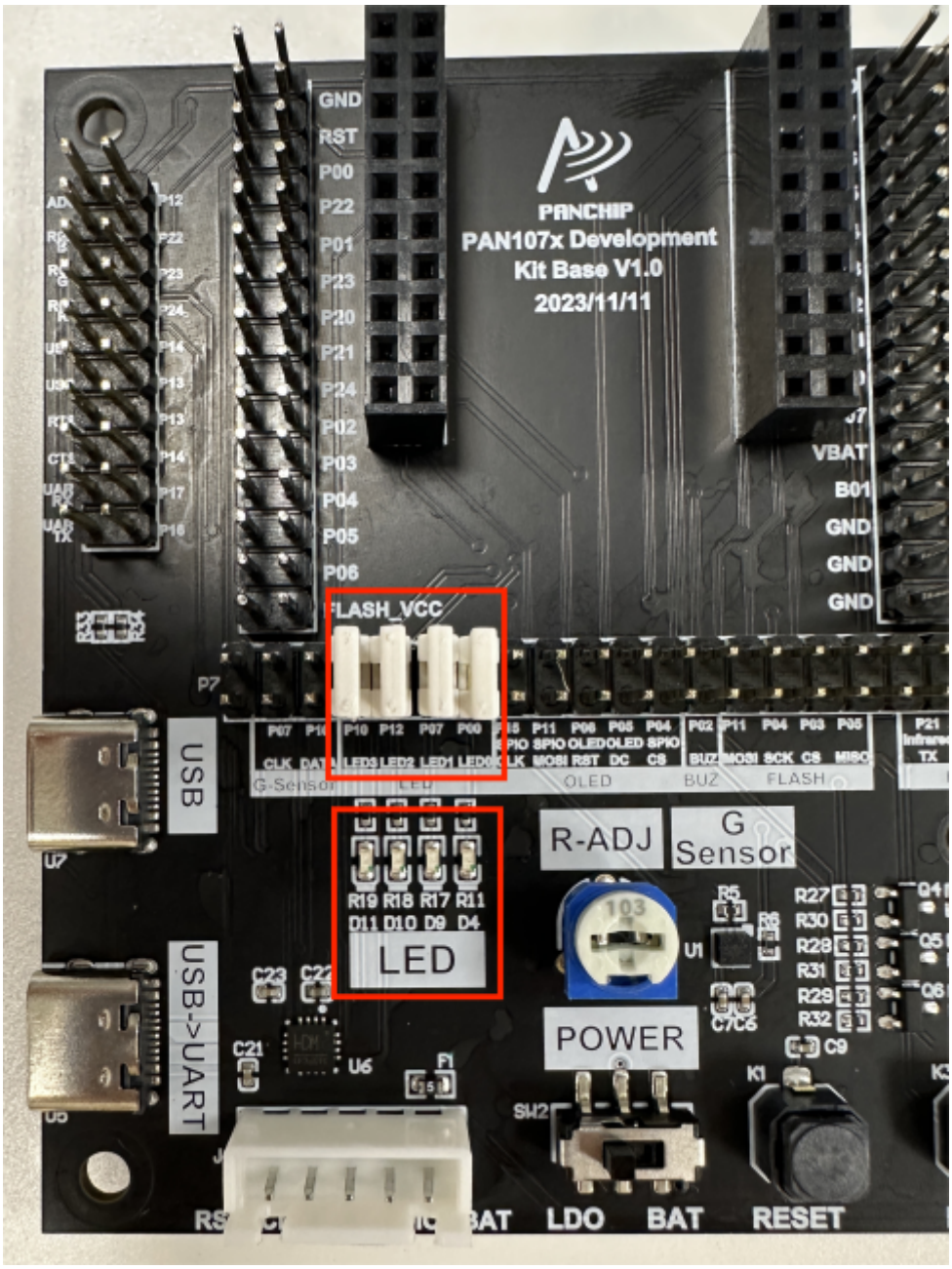


图 31: LED 灯模块实物接线图

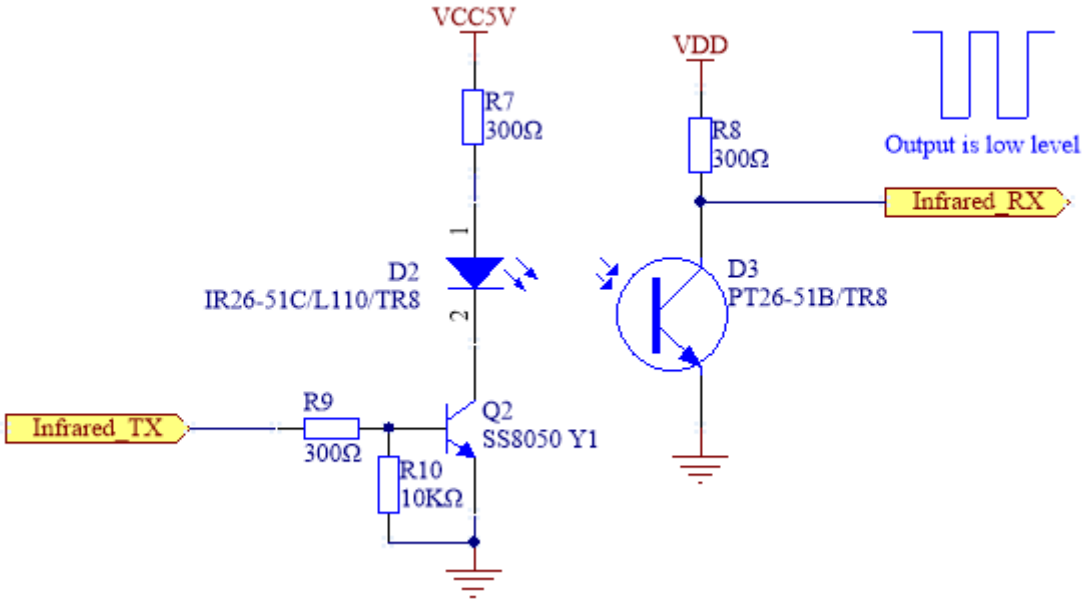


图 32: 红外模块原理图

2.4 晶振

2.4.1 晶振 32Mhz

- 上图振荡器由晶振、反馈电阻 R1、负载电容 CL、放大器构成。可以通过晶体所需负载电容 CL 来调整晶体振荡器频率，负载电容 CL1、CL2 取值为 3.9pF。推荐型号为：E1SB32E000016E 32MHz 9pF ±10ppm 。该晶体温漂较好。

2.4.2 晶振 32.768Khz

- 低速晶振电路支持外部 32.768KHz 无源晶振。低速晶振推荐用户选择 ESR<80KΩ 的晶振，负载电容 C12、C15 取值为 10pF。推荐型号为：Q13FC1350000200 32KHz 12.5pF ±20ppm 。该晶体温漂较好。

2.5 复位电路

复位引脚可以悬空，或增加外部按键。在外部按键应用中必须有电容，参数为 100nF。加电容的作用是在系统受到强干扰时，稳定复位脚的电平状态。

> **注意**：该引脚内部有一个 50KΩ 左右的上拉电阻，低电平会使复位生效，为避免电路异常，该电容容值请不要随意更改！

2.6 静电防护

2.6.1 IO 端静电防护 使用的 IO 要预留串联电阻和 ESD 静电防护元件焊盘位置，便于过认证前的调试整改。串电阻的作用主要是减少 IO 信号的反射、降低外部毛刺信号干扰以及削弱静电对 IO 的影响。**频繁与外部进行数据交换的** IO **例如使用** USB **功能脚等，必须使用** TVS **管进行保护**。建议使用的 TVS 管类型如单向的 ESD5Z3V3 或双向的 CESD923NC5VB，**靠近外设接口位置摆放，**ESD **静电防护元件附近建议保留完整、连续的地，周围打尽量多过孔，有利电荷泄放**。

2.6.2 电源端静电防护 电源输入端 VBAT 建议加上静电防护元件，若有静电进入可快速将电荷泄放到地，尽可能避免损坏芯片。ESD **静电防护元件靠近电源输入端接口摆放**。可选择 ESD5Z3V3。

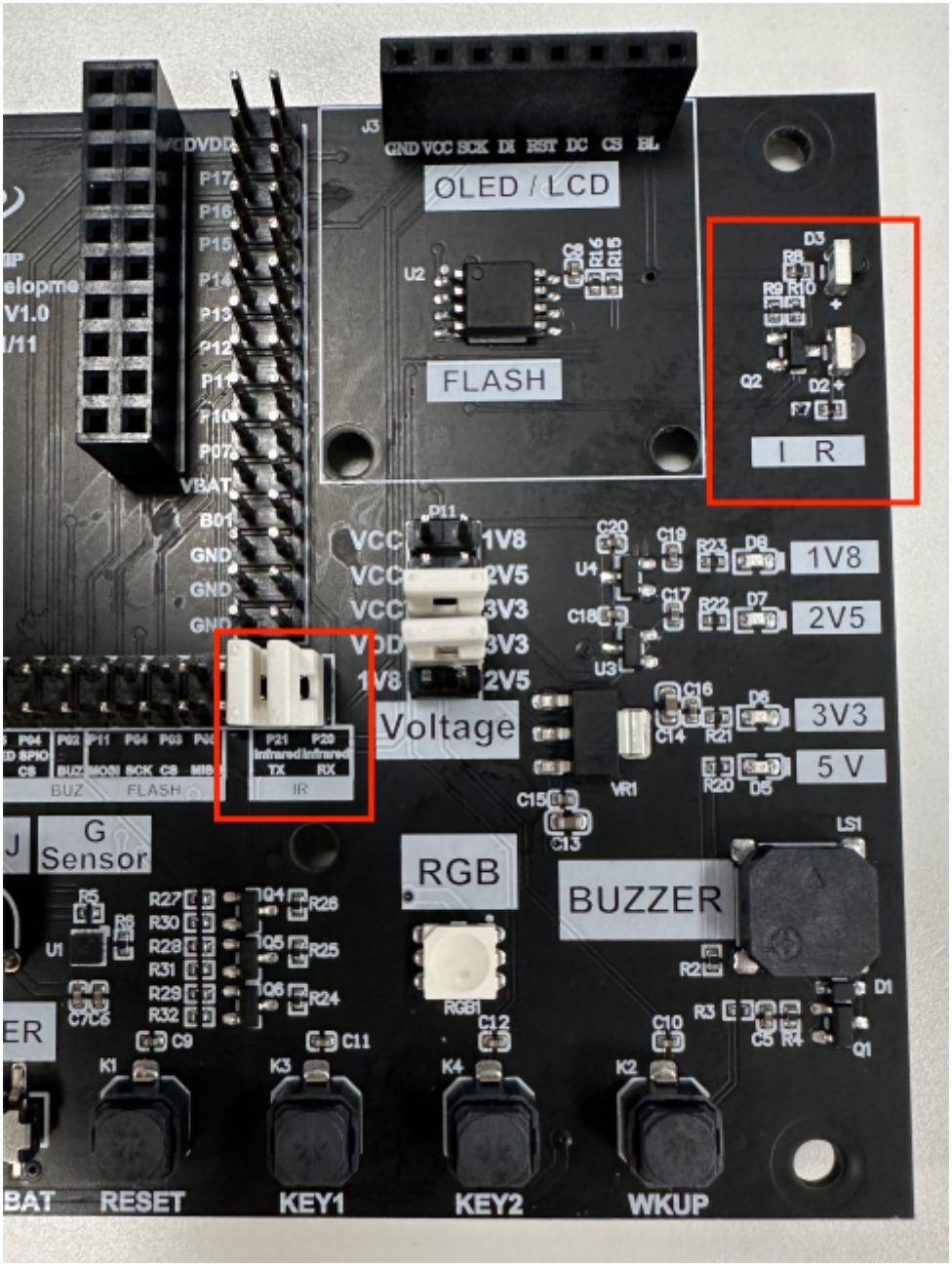


图 33: 红外模块实物接线图

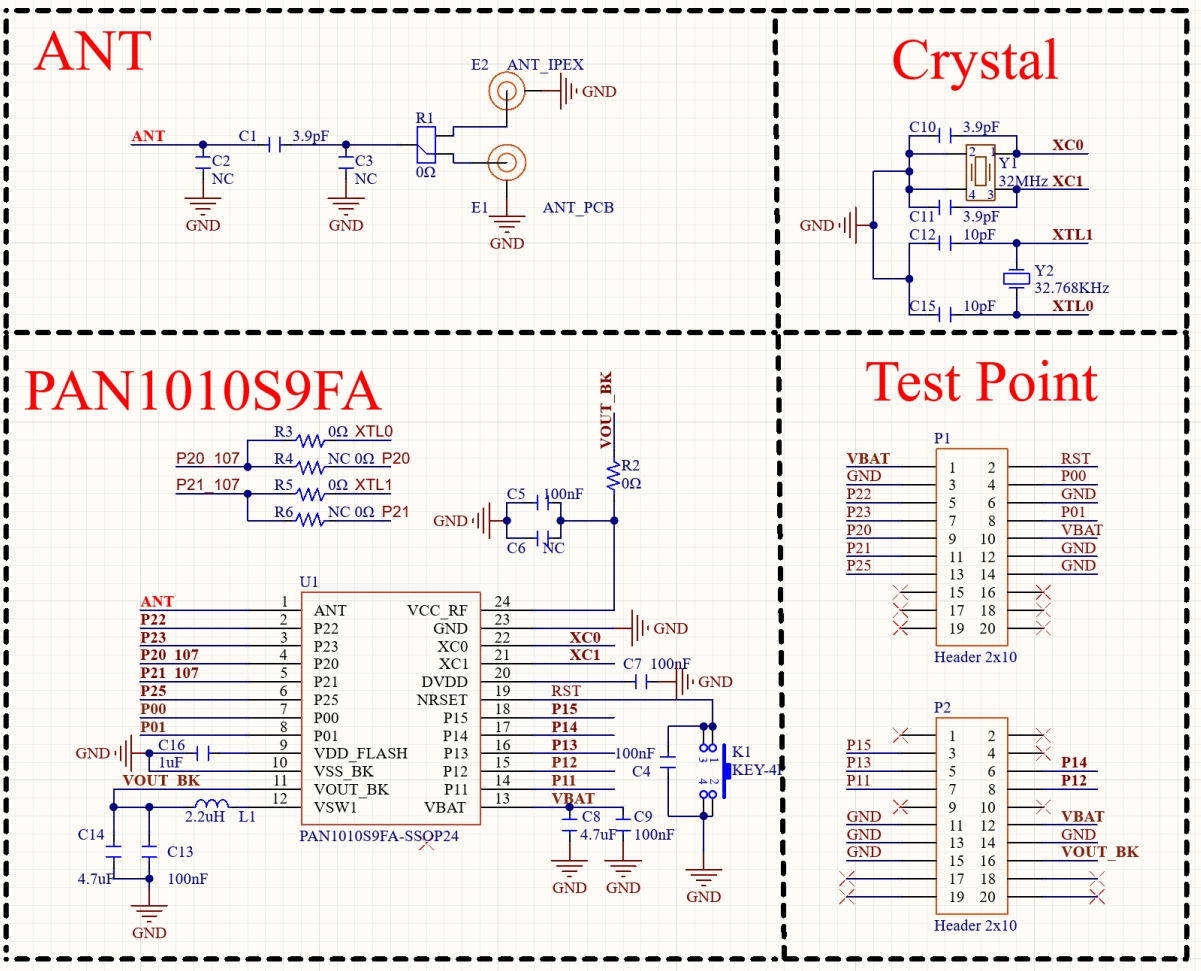


图 34: PAN1010S9FA 最小系统参考设计原理图

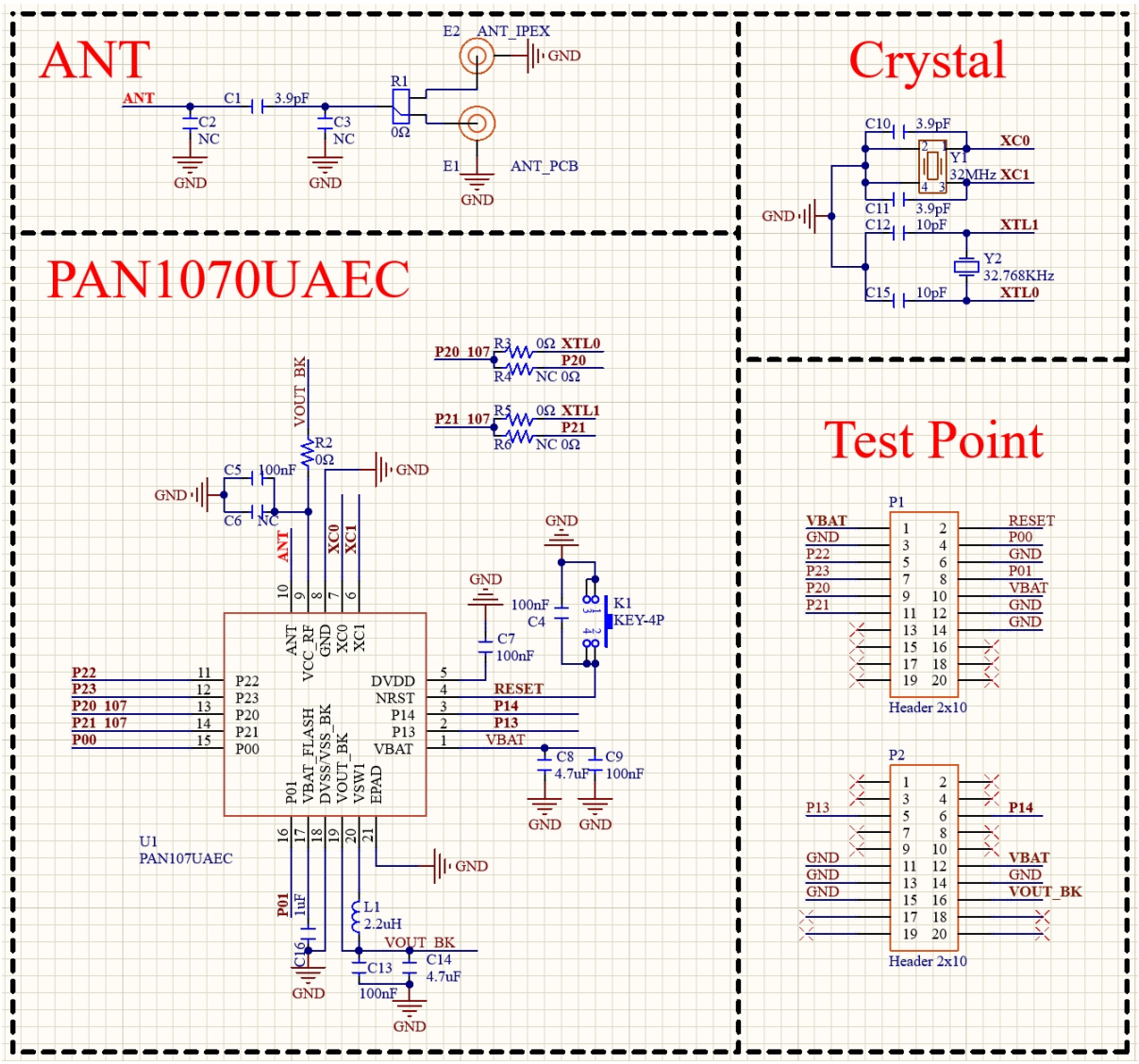


图 35: PAN1070UA1A 最小系统参考设计原理图

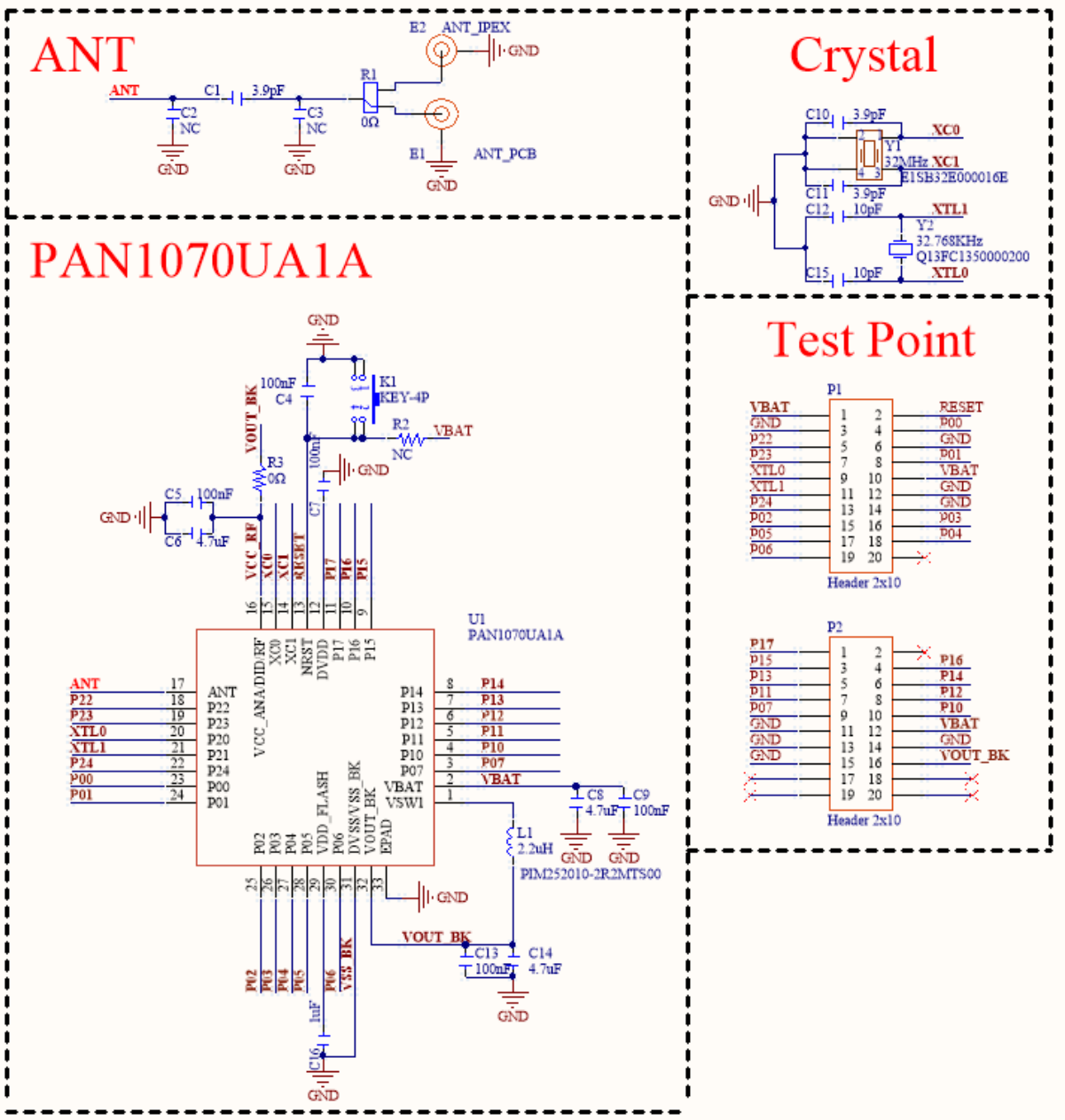


图 36: PAN1070UA1A 最小系统参考设计原理图

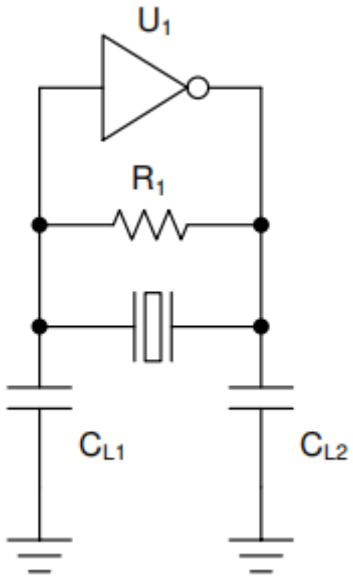


图 37: 32MHz 晶振外围电路示意图

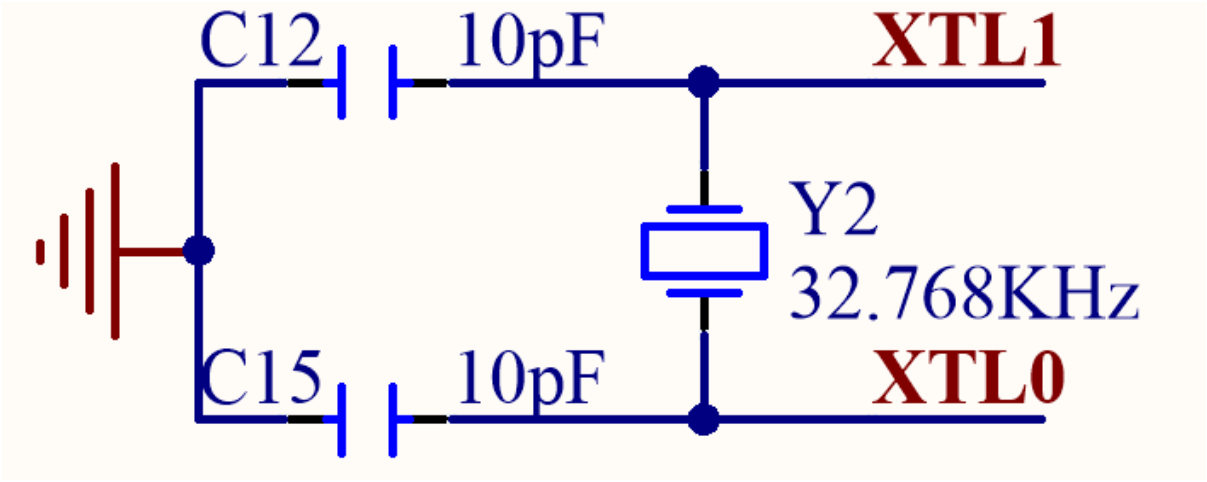


图 38: 32KHz 晶振外围电路原理图

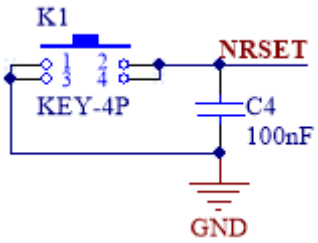


图 39: 复位电路

2.6.3 **天线端静电防护** 无论是板载天线还是其他天线，本质上都是一段长导体，必然有概率吸引到静电电荷，为预防静电打坏芯片 RF，天线端建议加上静电防护元件，**必须使用低容值** (**小于** 0.5pF)、**低钳位电压的** TVS **元器件，尽可能不影响** RF **阻抗，如** JEB03SCDF。ESD **静电防护元件靠近芯片摆放，若** RF **阻抗受到影响还可通过 匹配调整**。在天线的位置放置一个 ESD 管。

推荐使用有馈地点的天线，板载天线推荐 PIFA。

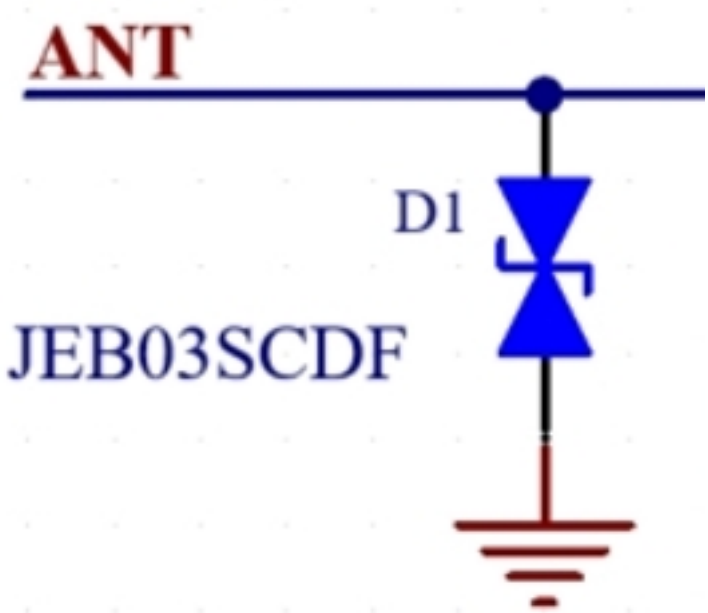


图 40: 天线端静电防护示意图

2.7 IO 功能分配

靠近天线端的 IO（P22、P23）尽量不要频繁的翻转，比如用作 PWM 功能。这会使射频底噪变差，影响灵敏度。

2.2.3 3 PCB 设计建议

3.1 制版工艺

- 本 Guide 主要针对二层板并且单面贴设计，叠层如下图所示。PCB 具体厚度根据实际情况和阻抗要求适当调整。

* 线宽推荐如下：

板材属性	参数
PCB 板材	FR4
PCB 板厚	1.6mm
50 欧姆 RF 线宽	20mil
接地铺铜与 RF 走线间距	5mil

3.2 电源部分注意事项

3.2.1 电源去耦电容布局

- VBAT，VCC_RF, VOUT_BK,DVDD 管脚就近放置电容，走线尽量短粗, 如下图绿色框图部分。

	Layer Name	Type	Material	Thickness (mil)	Dielectric Material	Dielectric Constant
	Top Overlay	Overlay				
	Top Solder	Solder Mask/Coverlay	Surface Material	0.4	Solder Resist	3.5
1	Top Layer	Signal	Copper	1.8		
	Dielectric1	Dielectric	None	59.4	FR-4	4.6
2	Bottom Layer	Signal	Copper	1.8		
	Bottom Solder	Solder Mask/Coverlay	Surface Material	0.4	Solder Resist	3.5
	Bottom Overlay	Overlay				

图 41: 制版工艺说明

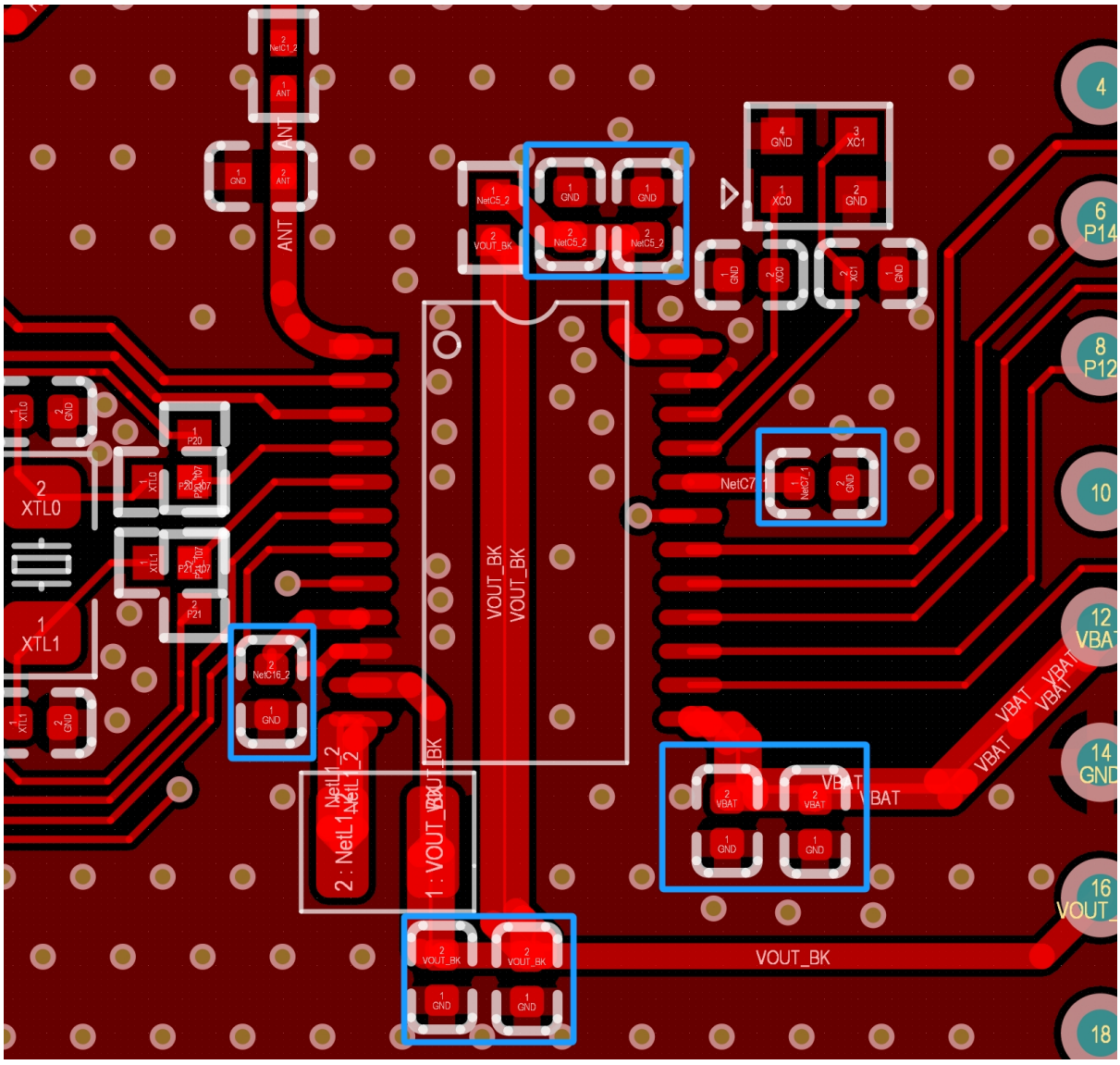


图 42: PAN1010S9FA 电源去耦电容布局

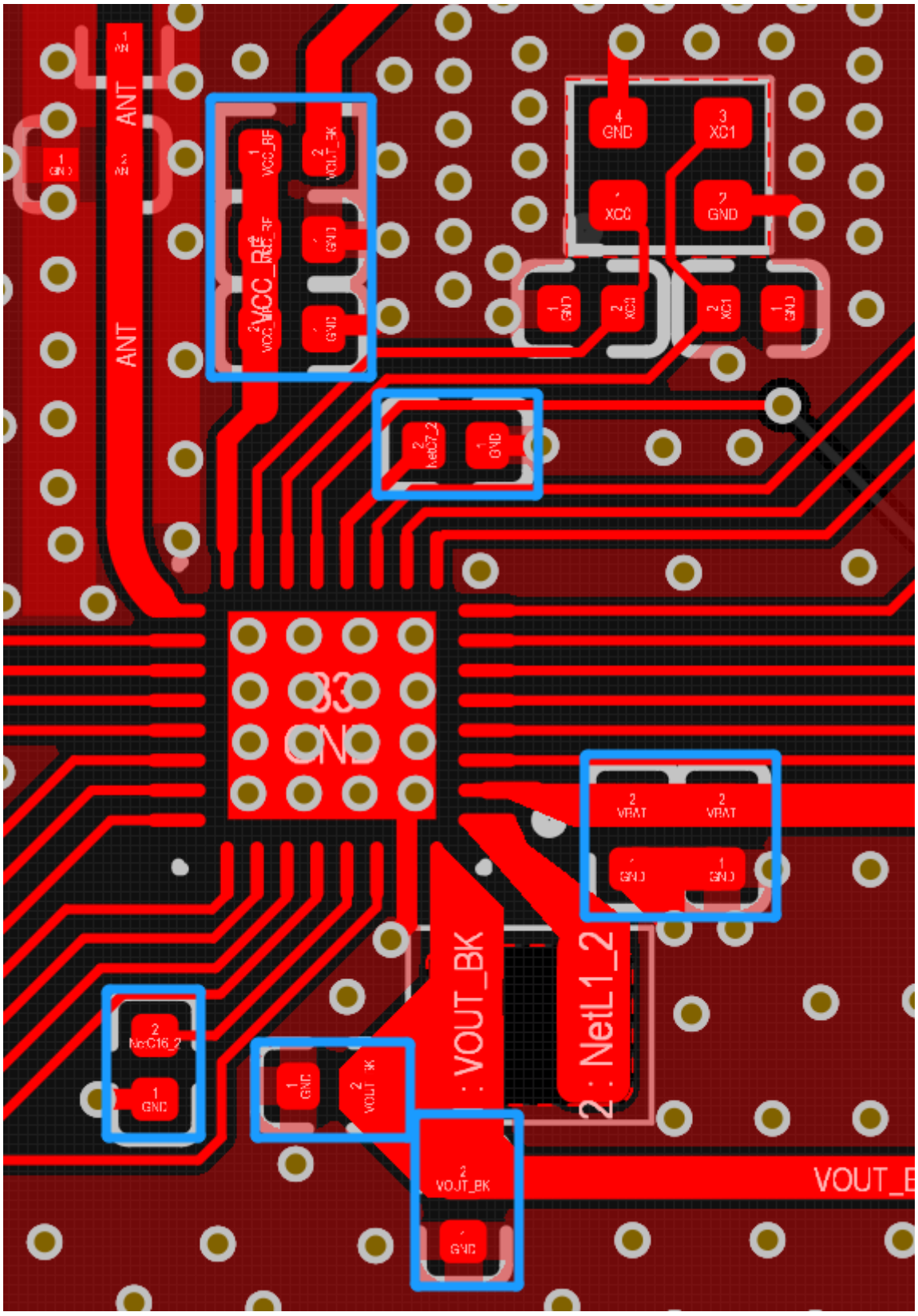


图 43: PAN1070UA1A 电源去耦电容布局

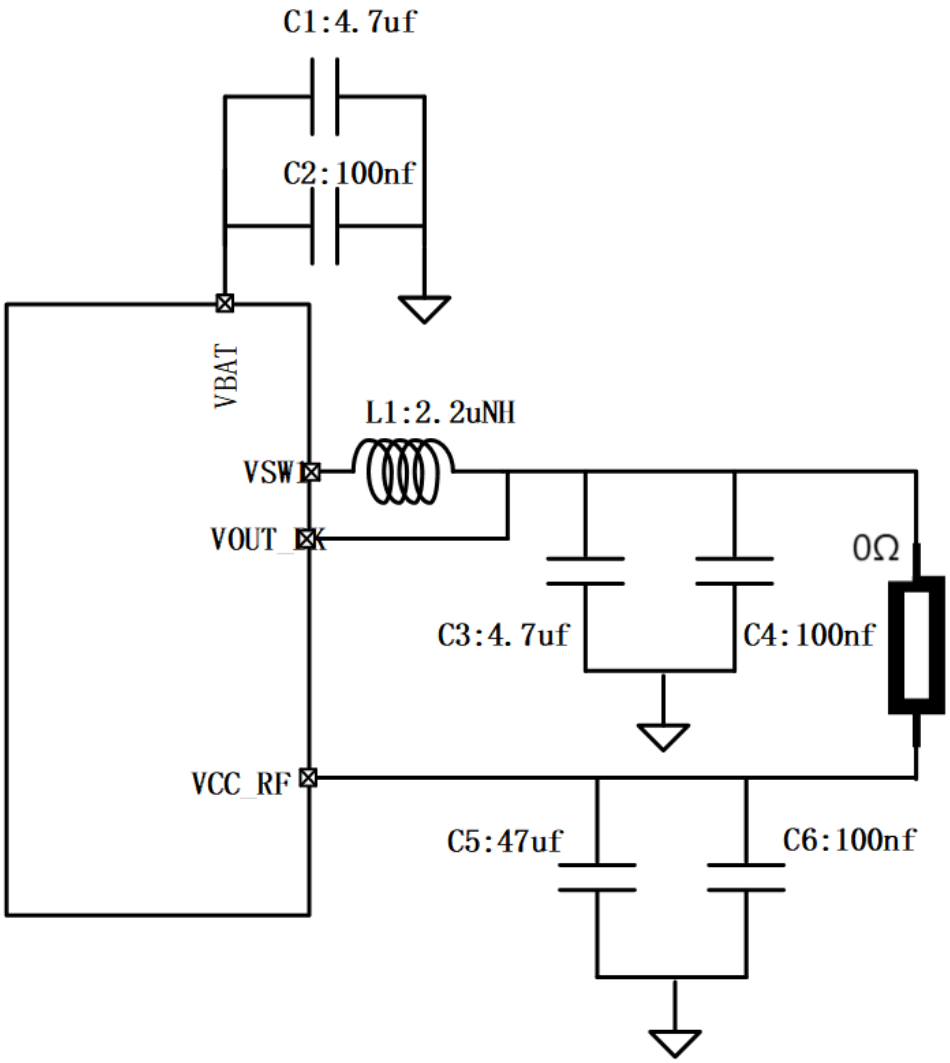


图 44: DC-DC 外围电路

3.2.2 DC-DC PCB **布局参考设计**

- VSW1 管脚与电感 L1 距离尽量的短，且附铜面积尽量大。
- L1 与 C3&C4 之间的走线尽量短，且附铜面积尽量大。
- C1&C2 靠近 VBAT_BK 管脚位置摆放，走线尽量短
- VOUT 与 VCC_RF 之间增加小电阻靠近 C5&C6 摆放
- 电感 L1 尽量远离晶振 XTH
- 电感四周用 GND 隔离
- DCDC 的地不能与 XTH 共用，DCDC 的地通过 0Ω 电阻和其他地连接。DCDC 的地不能直连到 EPAD。

  总体原则为 DCDC **电流回路路径尽可能短**；DCDC **地属于干扰源尽量隔离，避免干扰晶振**。

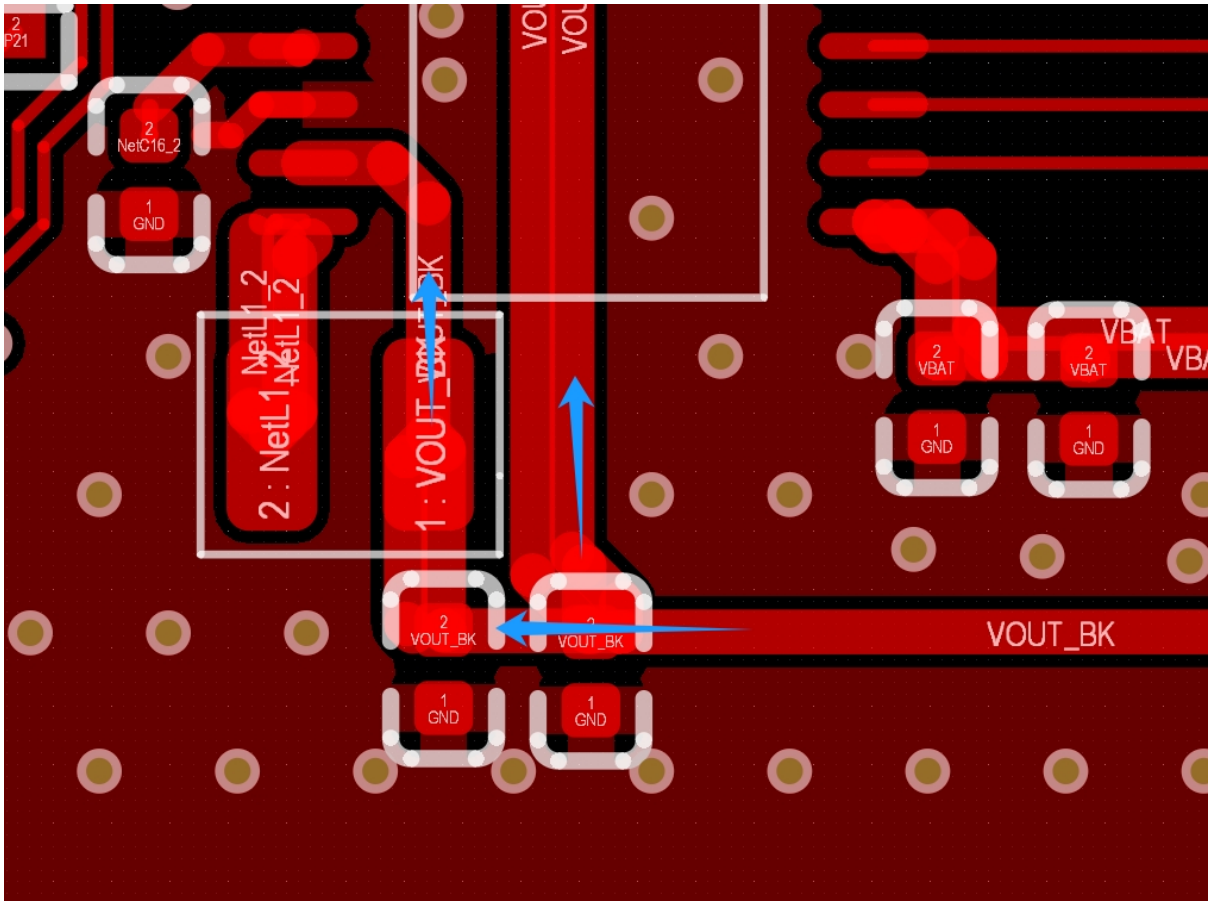


图 45: PAN1010S9FA DC-DC Layout 示意图

3.3 **射频走线注意事项**

- 晶振尽量靠近芯片引脚摆放。
- 射频匹配链路按照 50Ω 走线，可以参考 TOP 和 BOTTOM 层的 GND 平面，RF 走线尽可能短，RF 线与焊盘宽度一致，天线的 型匹配并联元件焊盘和走线重合为佳。
- RF 线有完整的参考地，从 IC 端出来就进行包地处理，两边打 GND 过孔，底层地平面尽量宽，如**标签** 1 所指信号走线。
- IPEX-1 代天线端子信号引脚挖空，周围包地，尽量减小寄生电容导致阻抗突变，如**标签** 3。
- 芯片底部多打过孔，QFN 封装则打在 E-PAD 上，如**标签** 2。

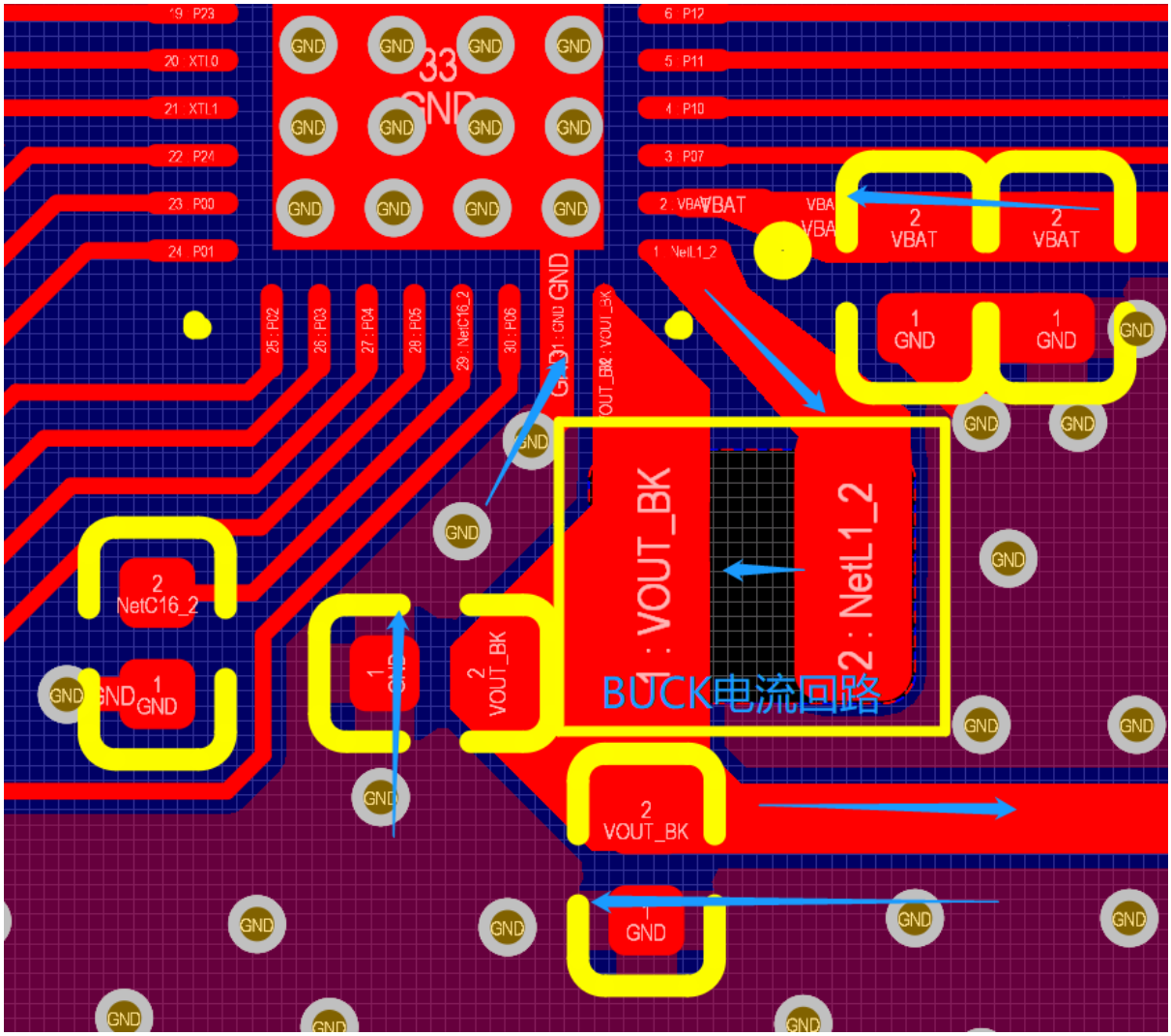


图 46: PAN1070UA1A DC-DC Layout 示意图

- 晶振应远离天线，TOP 层挖空，周围包地，降低对电源和 RF 的干扰，需要挖空的部分如**标签** 3。
- 天线辐射区域不要摆放金属器件，净空区挖空处理。

射频链路走线参考如下：

- 天线匹配链路底层不要走线，保证地回路到芯片最短。天线匹配链路的地和芯片 EPAD 是一块完整连续的地。如**标签 "射频地"**。
- 芯片底层不要走线。

射频地线走线如下：

3.4 板载天线

PCB Layout 参考中 MIFA 天线尺寸如图所示。

天线设计尺寸参考

3.5 RF 网络匹配调整

- 如果客户的样板有足够的空间和成本预算来放置匹配的网络组件以及拥有天线调谐能力，我们还建议使用匹配网络。

1. 要进行 RF 匹配调试，建议在芯片引脚附近、靠近天线辐射端都各加入一个 形匹配，并且在两个 匹配之间串上一个 0Ω 电阻便于调试。如下图 RF **网络匹配原理示意图**。
2. 建议先调远端，先进行天线端匹配调试。断开两个 形匹配之间的串联电阻（如 RF **网络匹配原理示意图**中的**标记** 2），使用如矢量网络分析仪可观测天线阻抗、S11 参数的仪器接入到天线端 形匹配网络（如 RF **网络匹配原理示意图**中的**标记** 3 位置），调试天线端阻抗和 S11 驻波。
3. 之后进行芯片输出功率调试，断开两个 形匹配之间的串联电阻，使用如矢量网络分析仪等可观测天线阻抗、S11 参数的仪器接入到芯片端的 形匹配网络（如 RF **网络匹配原理示意图**中的**标记** 1 位置），调整芯片输出阻抗以及功率。
4. 若没有仪器观测阻抗参数，亦可断开两个 形匹配之间的串联电阻，将频谱仪或其他可以检测 RF 输出功率的仪器接入到芯片端的 形匹配网络，通过检测 2402、2440、2480 三个频点的功率值，来调整芯片的输出功率。
5. 最后还需要测试芯片灵敏度情况，因为发射机和接收机内部匹配不完全相等，所以当发射功率调好以后，最后需要确认接收灵敏度情况，如果发射功率调整好后发生接收机灵敏度下降，联系我司工程师修改内部匹配。

   注意：量产前一定要进行距离测试!

2.2.4 4 BOM

PAN1010S9FA **最小系统** BOM **参考下表**:

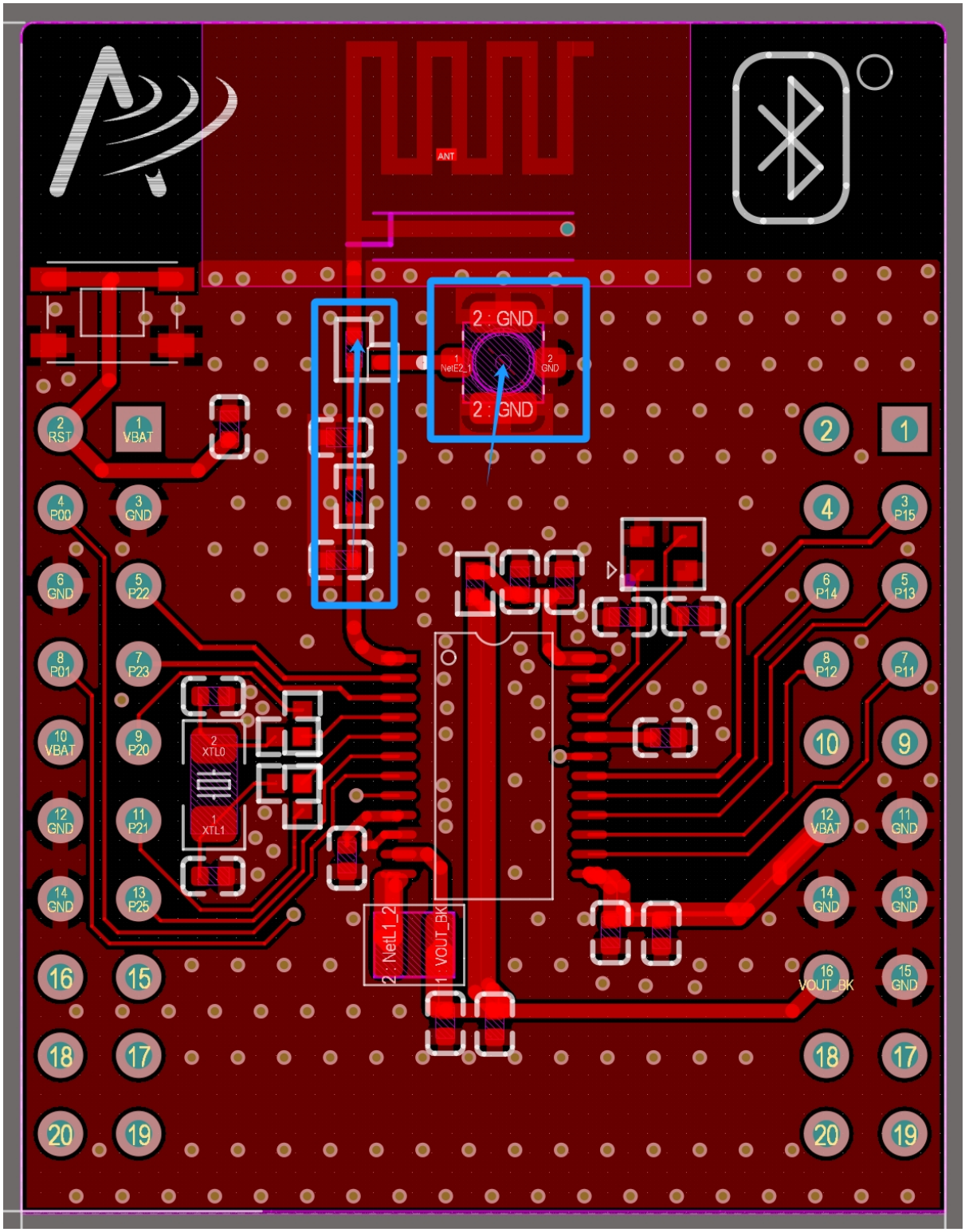


图 47: PAN1010S9FA 射频链路走线示意图

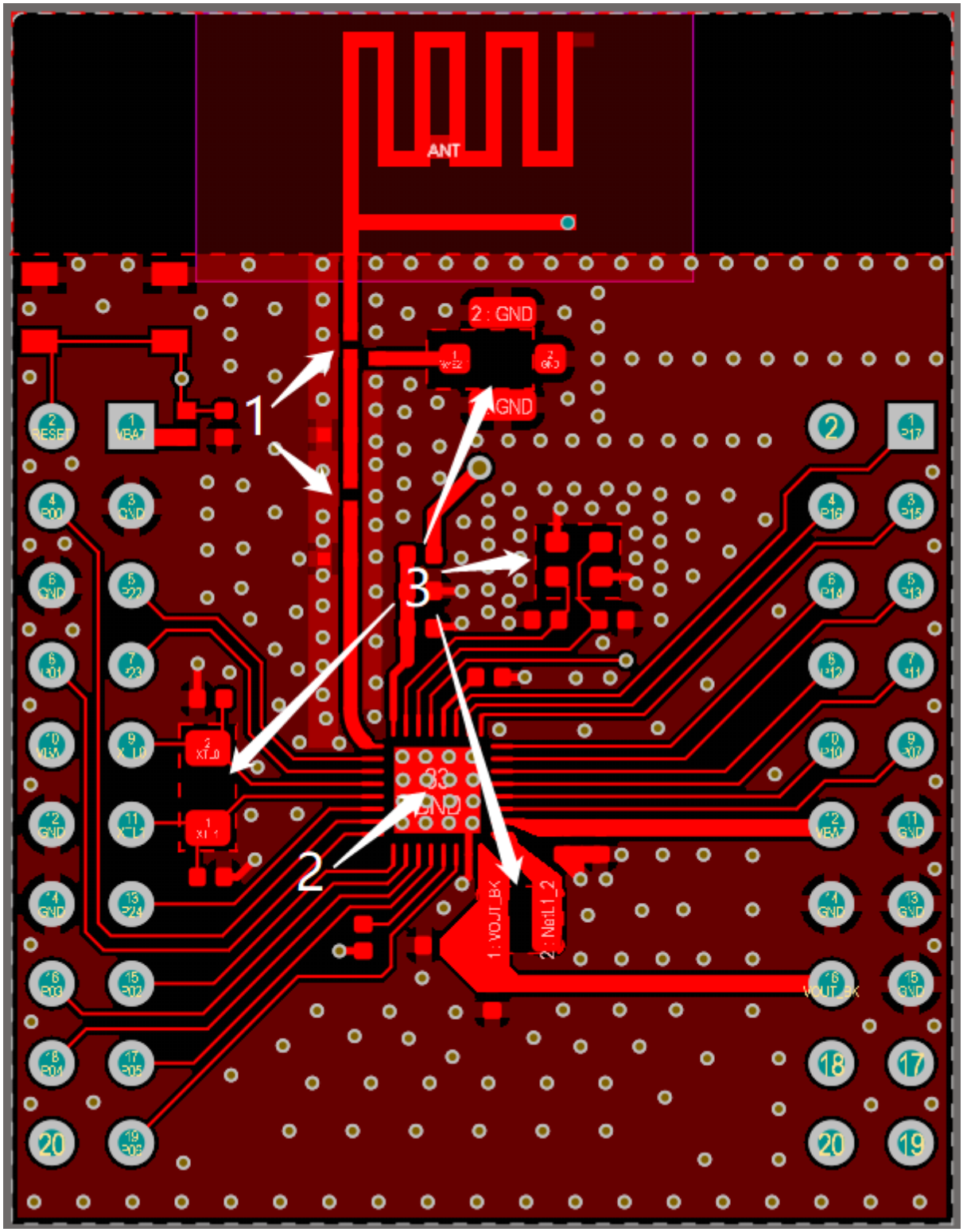


图 48: PAN1070UA1A 射频链路走线示意图

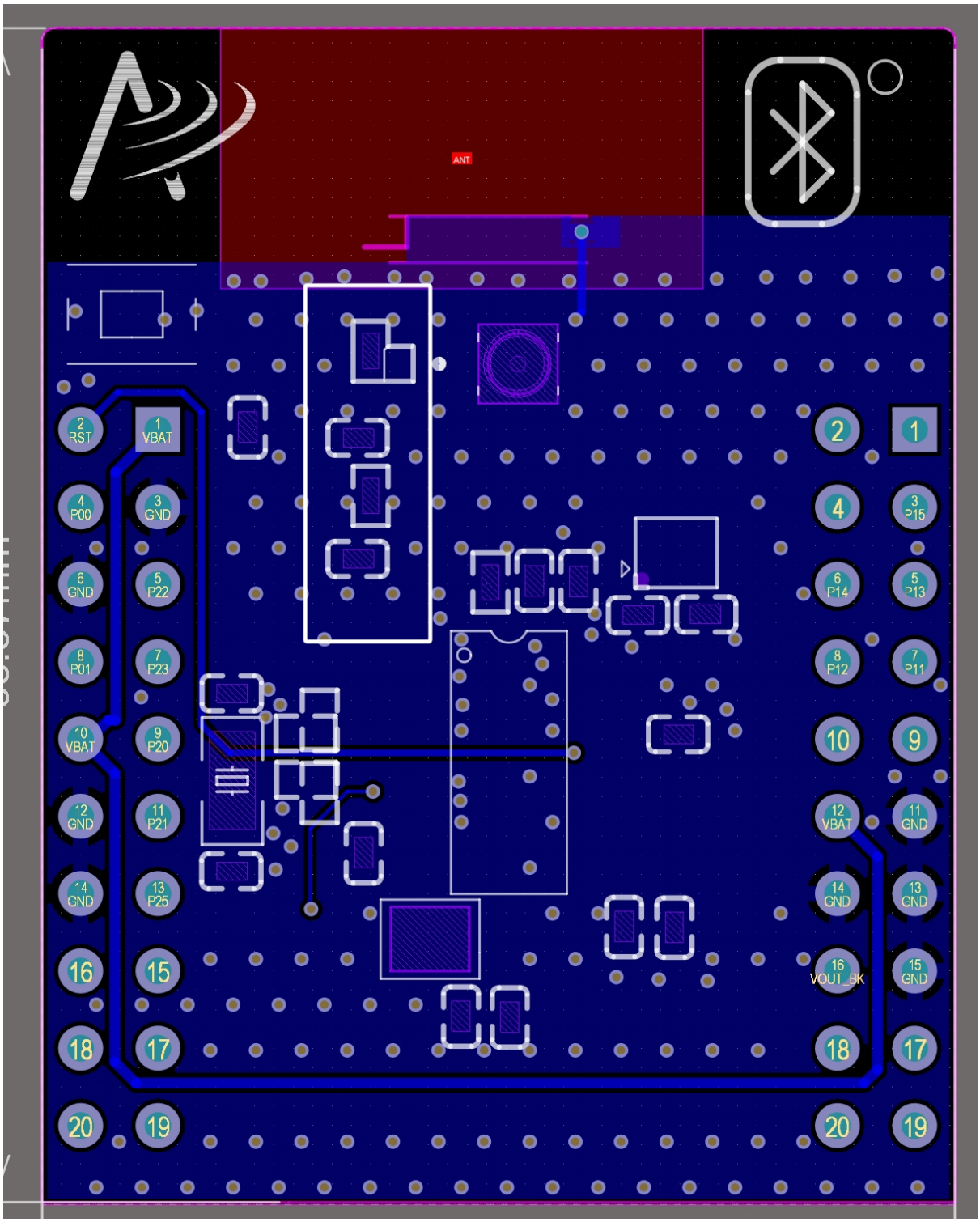


图 49: PAN1010S9FA 射频地线走线示意图

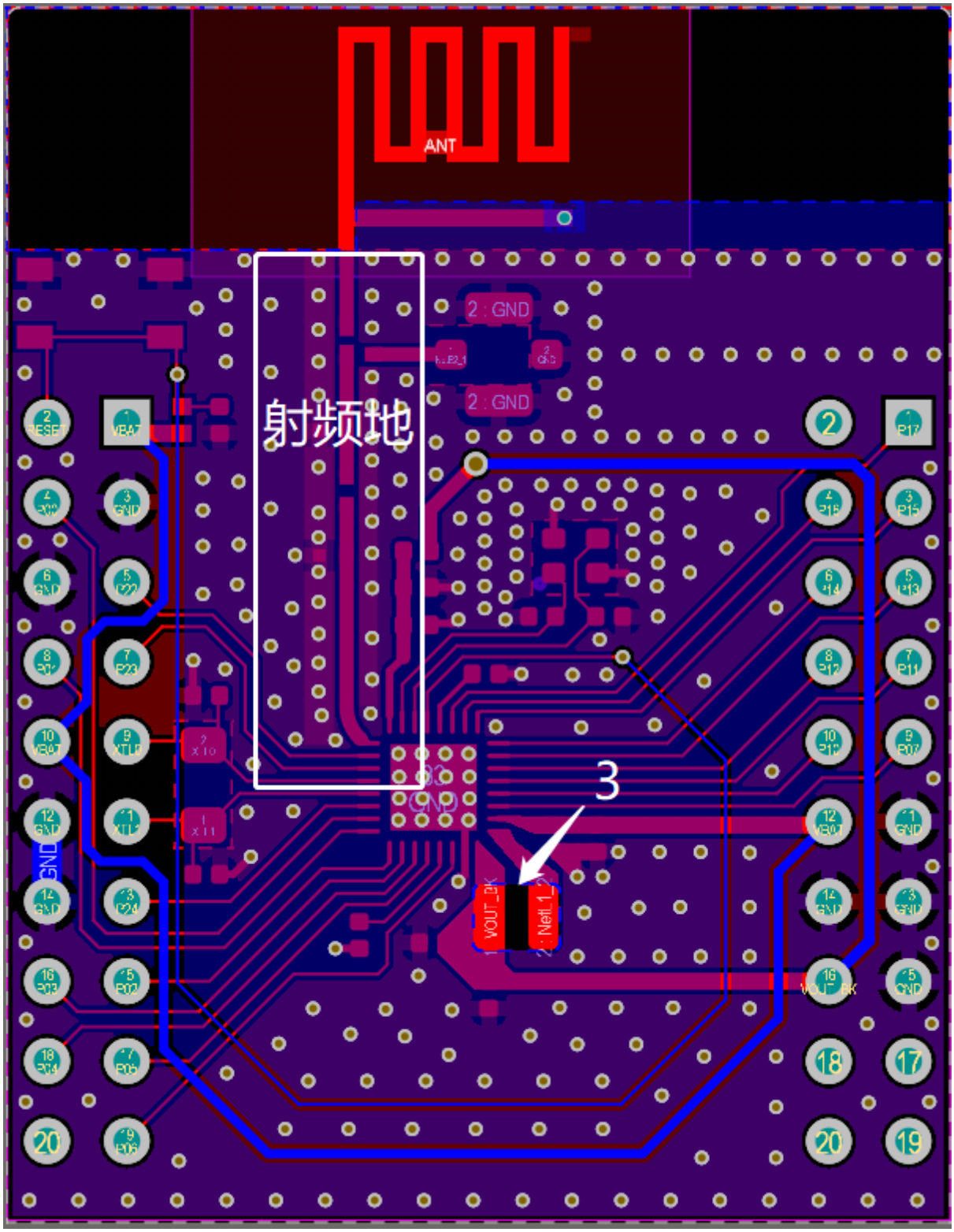


图 50: PAN1070UA1A 射频地线走线示意图

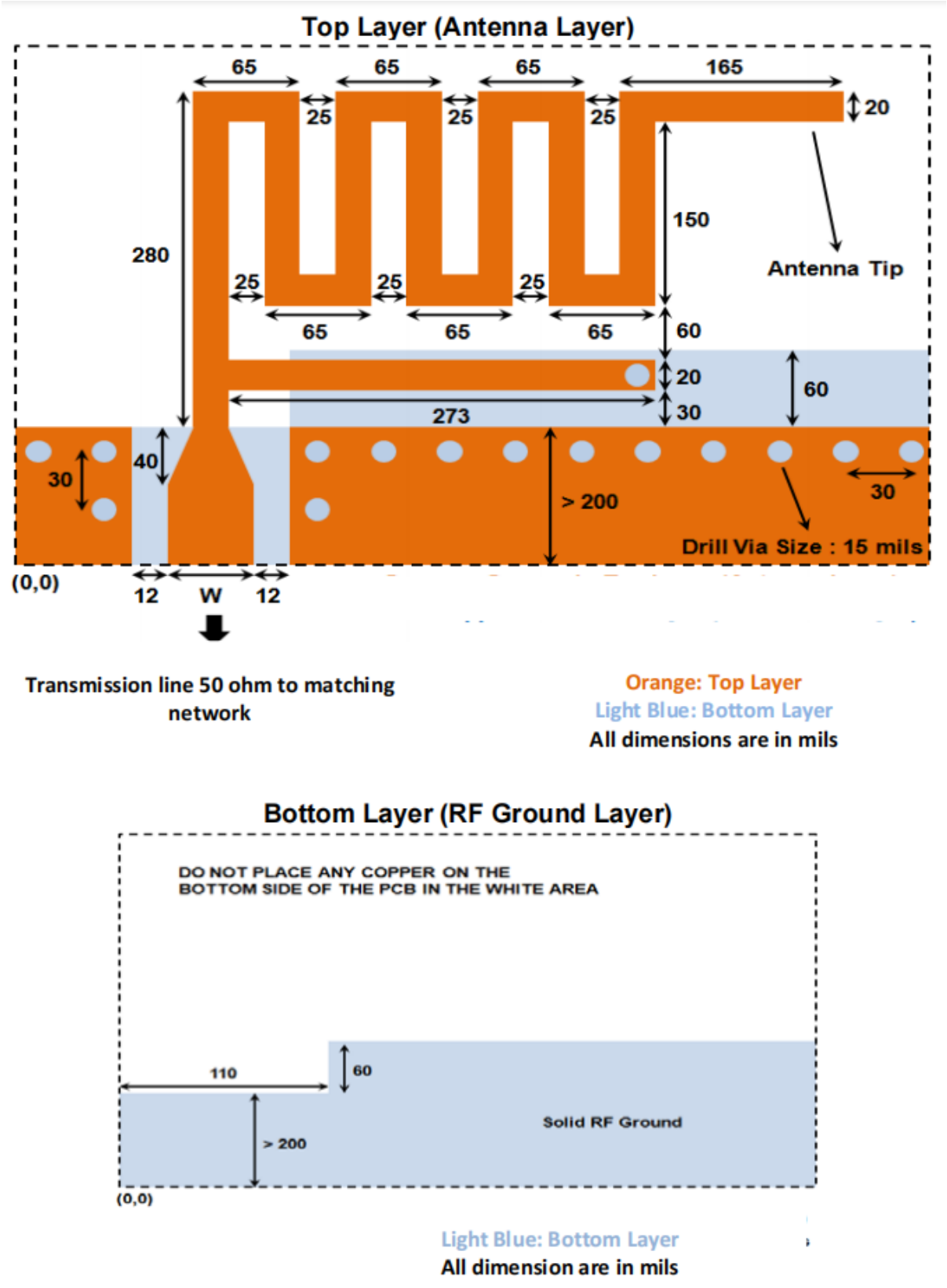
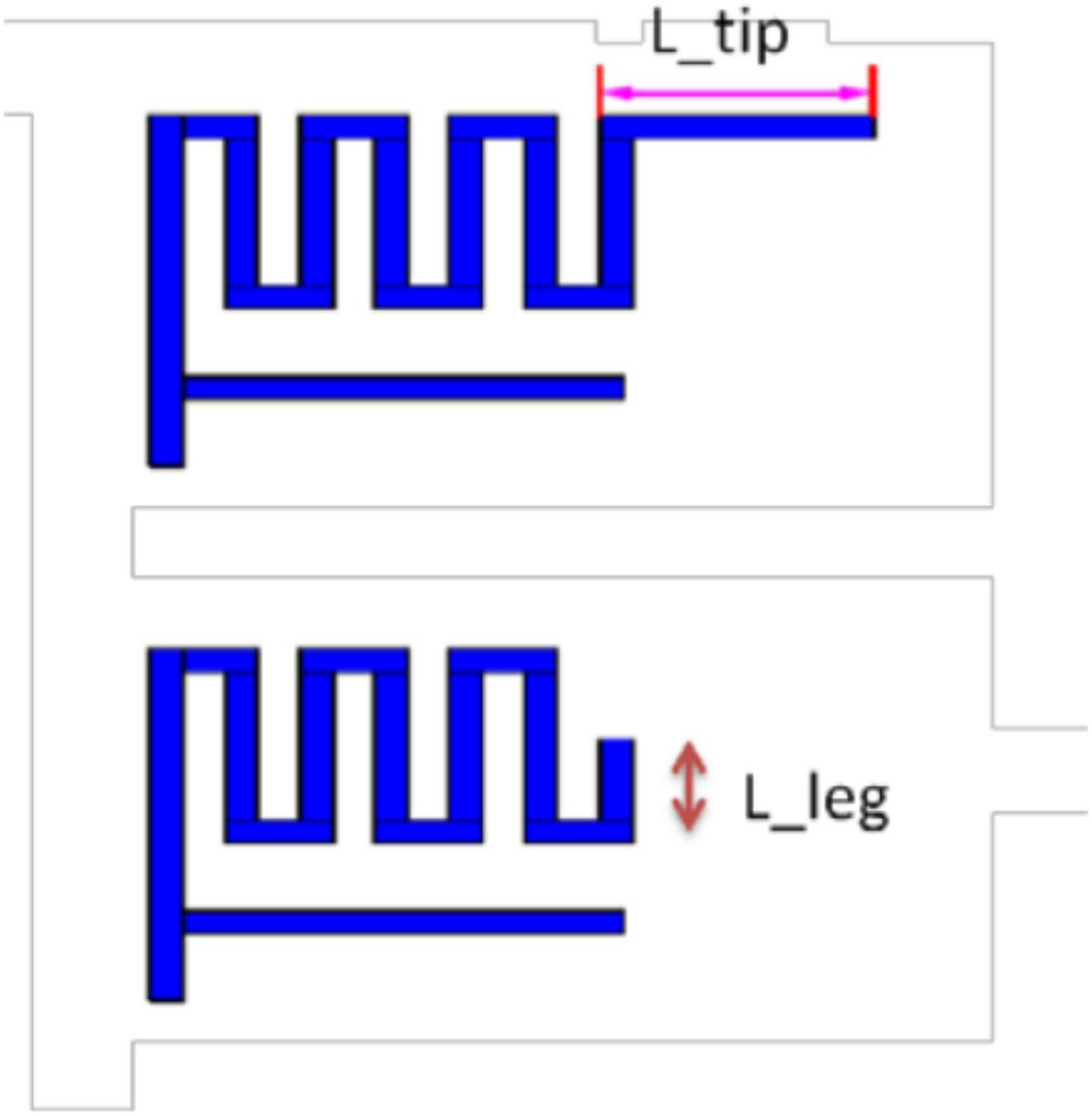


图 51: 天线设计尺寸参考示意图



PCB Thickness	Antenna L_Tip / L_leg
16 mils	L_tip= 353 Mils
31 mils	L_tip= 165 Mils
47 mils	L_tip= 125 Mils
62 mils	L_leg= 115 Mils

图 52: 天线设计匹配参考示意图

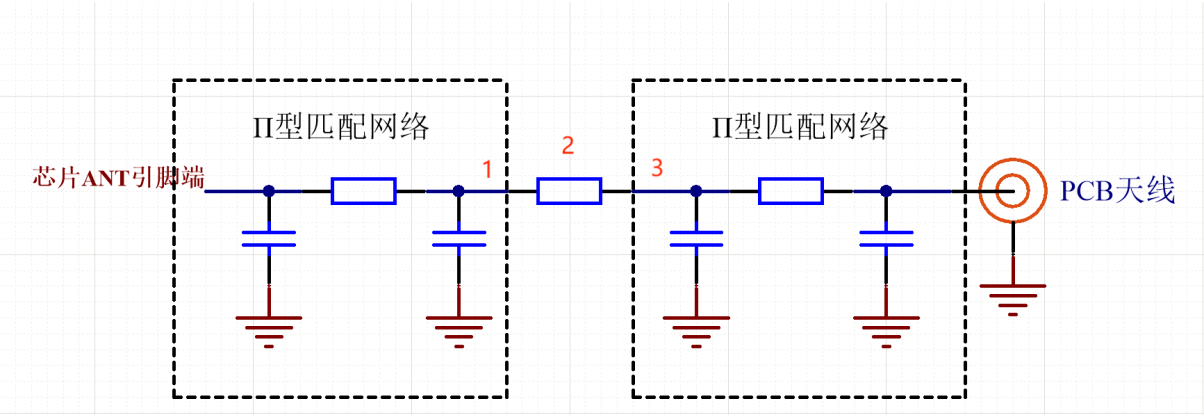


图 53: RF 网络匹配原理示意图

品种	参数	型号	品牌	立创编号	位号	封装	数量
贴片陶瓷电容	4.7uF	0402X475M6R3NT	广东风华高新科技股份有限公司	C168172	C8, C14	0402_C	2
贴片陶瓷电容	1uF	0402X105K6R3NT	广东风华高新科技股份有限公司	C142376	C16	0402_C	1
贴片陶瓷电容	100nF	0402B104K160NT	广东风华高新科技股份有限公司	C41851	C4,C5,C7,C9,C13	0402_C	5
贴片陶瓷电容	10PF	0402CG100J500NT	广东风华高新科技股份有限公司	C1545	C12,C15	0402_C	2
贴片陶瓷电容	3.9PF	0402CG3R9B500NT	广东风华高新科技股份有限公司	C313081	C1,C10,C11	0402_C	3
贴片按键	4P-4.2mm x 3.25mm	K2-1808SN-A4SW-01	韩荣电子有限公司	C92589	K1	SW-SMD(4.2x3.25x2.5)	1
贴片功率电感	2.2uH	PIM252010-2R2MTS00	广东风华高新科技股份有限公司	C2986792	L1	SMD-2520-1.0	1
贴片电阻	0Ω 1%	RC-02000FT	广东风华高新科技股份有限公司	C140225	R1,R2,R3,R5	0402_R	4
贴片 4 脚晶振	32MHz 10ppm 9pF	E1SB32E000016U	鸿星科技	C275757	Y1	SMD-3225_4P	1
贴片 2 脚晶振	32.768KHz 12.5pF	X321532768KGB2SI	汇创扬兴科技有限公司	C620155	Y2	SMD-3215_2P FC - 135	1
贴片 IC	PAN1010S9FA	\	上海磐启微电子有限公司	\	U1		

PAN1070UAEC 最小系统 BOM 参考下表:

品种	参数	型号	品牌	立创编号	位号	封装	数量
贴片陶瓷电容	4.7uF	0402X475M6R3NT	广东风华高新科技股份有限公司	C168172	C8, C14	0402_C	2
贴片陶瓷电容	1uF	0402X105K6R3NT	广东风华高新科技股份有限公司	C142376	C16	0402_C	1
贴片陶瓷电容	100nF	0402B104K160NT	广东风华高新科技股份有限公司	C41851	C4,C5,C7,C9,C13	0402_C	5
贴片陶瓷电容	10PF	0402CG100J500NT	广东风华高新科技股份有限公司	C1545	C12,C15	0402_C	2
贴片陶瓷电容	3.9PF	0402CG3R9B500NT	广东风华高新科技股份有限公司	C313081	C1,C10,C11	0402_C	3
贴片按键	4P-4.2mm x 3.25mm	K2-1808SN-A4SW-01	韩荣电子有限公司	C92589	K1	SW-SMD(4.2x3.25x2.5)	1
贴片功率电感	2.2uH	PIM252010-2R2MTS00	广东风华高新科技股份有限公司	C2986792	L1	SMD-2520-1.0	1
贴片电阻	0Ω 1%	RC-02000FT	广东风华高新科技股份有限公司	C140225	R1,R2,R3,R5	0402_R	4
贴片 4 脚晶振	32MHz 10ppm 9pF	E1SB32E000016P3	鸿星科技	C275757	Y1	SMD-3225_4P	1
贴片 2 脚晶振	32.768KHz 12.5pF	X321532768KGB2SI	扬兴科技有限公司	C620155	Y2	SMD-3215_2P FC - 135	1
贴片 IC	PAN1070UAEC	\	上海磐启微电子有限公司	\	U1	QFN20	1

PAN1070UA1A **最小系统** BOM **参考下表**:

品种	参数	型号	品牌	立创编号	位号	封装	数量
贴片陶瓷电容	4.7uF	0402X475M6R3NT	广东风华高新科技股份有限公司	C168172	C6, C8, C14	0402_C	3
贴片陶瓷电容	1uF	0402X105K6R3NT	广东风华高新科技股份有限公司	C142376	C16	0402_C	1
贴片陶瓷电容	100nF	0402B104K160NT	广东风华高新科技股份有限公司	C41851	C4,C5,C7,C9,C13	0402_C	5
贴片陶瓷电容	10PF	0402CG100J500NT	广东风华高新科技股份有限公司	C1545	C12,C15	0402_C	2
贴片陶瓷电容	3.9PF	0402CG3R9B500NT	广东风华高新科技股份有限公司	C313081	C1,C10,C11	0402_C	3
贴片按键	4P-4.2mm x 3.25mm	K2-1808SN-A4SW-01	韩荣电子有限公司	C92589	K1	SW-SMD(4.2x3.25x2.5)	1
贴片功率电感	2.2uH	PIM252010-2R2MTS00	广东风华高新科技股份有限公司	C2986792	L1	SMD-2520-1.0	1
贴片电阻	0Ω 1%	RC-02000FT	广东风华高新科技股份有限公司	C140225	R1,R3	0402_R	2
贴片 4 脚晶振	32MHz 10ppm 9pF	E1SB32E000016P3	鸿星科技	C275757	Y1	SMD-3225_4P	1
贴片 2 脚晶振	32.768KHz 12.5pF	X321532768KGB2SI	扬兴科技有限公司	C620155	Y2	SMD-3215_2P FC - 135	1
贴片 IC	PAN1070UA1A	\	上海磐启微电子有限公司	\	U1	QFN32	1

HDK **内容**:

硬件开发资料（HDK）位于: `<PAN107X-DK>\02_HDK`，其包含如下内容:

02_HDK **子目录**	**包含内容**
PAN107x Development Kit Base V1.1	PAN107x EVB 底板硬件设计资料（原理图、PCB 文件等）和生产资料（BOM、gerber、坐标等文件）
PAN1070UA1A	PAN107x EVB QFN32 核心板硬件设计资料（原理图、PCB 文件等）和生产资料（BOM、gerber、坐标等文件）
PAN1010S9FA	PAN101x EVB SSOP24 核心板硬件设计资料（原理图、PCB 文件等）和生产资料（BOM、gerber、坐标等文件）

Chapter 3

演示例程

3.1 蓝牙例程

3.1.1 BLE Central and Peripheral

1 功能概述

此项目演示蓝牙主从一体功能，其实就是整合了 ble_central 以及 ble_peripheral_hr 功能。

- 作为主机：可以直接扫描和连接 `ble_peripheral_enc` 示例，可以直接下载 `ble_peripheral_enc` 到另外一块 EVB 板上。
- 作为从机：其实就是一个 `ble_peripheral_hr` 例程，可以使用手机 nrf_connect app 与其相连。

作为主机和从机的功能可以同时使用。

2 环境要求

- board: `pan107x evb`
- uart(option): 用来显示串口 log（波特率 921600，选项 `8n1`）

3 编译和烧录

例程位置：`<home>\nimble\samples\bluetooth\ble_cent_prph\keil_107x`

使用 `keil` 进行打开项目进行编译烧录。

4 演示说明

1. 烧录完成后，如果空中有 `ble_peripheral_enc` 存在，则会主动连接上。同时使用 nrf connect 扫描发发现 `cent_prph` 设备，连接上后会出现 `heartrate service` 服务。
   - 主机 log 如下：

```
[15:32:17.967]Try to load HW calibration data.. DONE.
- Chip Type         : 0x80
- Chip CP Version   : None
- Chip FT Version   : 8
- Chip MAC Address  : D0000C0CBBF5
- Chip Flash UID    : 32313334320EAC834330FFFFFFFFFFFF
- Chip Flash Size   : 1024 KB
```

(下页继续)

(续上页)

```
LL Spark Controller Version:b0e99c4

[15:32:18.011]ble_store_config_num_our_secs:0
ble_store_config_num_peer_secs:0
ble_store_config_num_cccds:0
blehr_advertise

[15:32:19.216]Connection established handle=0 our_ota_addr_type=0 our_ota_
↪addr=06:05:04:03:06:06 our_id_addr_type=0 our_id_addr=06:05:04:03:06:06 peer_ota_addr_
↪type=0 peer_ota_addr=06:05:04:03:02:01 peer_id_addr_type=0 peer_id_
↪addr=06:05:04:03:02:01 conn_itvl=40 conn_latency=0 supervision_timeout=256 encrypted=0␣
↪authenticated=0 bonded=0
/*作为主机连接 slave 设备*/
[15:32:21.923]Service discovery complete; status=0 conn_handle=0

[15:32:22.021]Read complete; status=0 conn_handle=0 attr_handle=12 value=0x00
Write complete; status=270 conn_handle=0 attr_handle=22

[15:32:22.073]Subscribe complete; status=0 conn_handle=0 attr_handle=20

[15:32:27.516]connection established; status=0
/*作为从机被手机主机连接*/
[15:32:30.908]subscribe event; cur_notify=1
 value handle; val_handle=3
```

- 从机 ble_peripheral_enclog 如下：

```
[15:32:15.447] Try to load HW calibration data.. DONE.
- Chip Type         : 0x80
- Chip CP Version   : None
- Chip FT Version   : 7
- Chip MAC Address  : D0000C06FB6E
- Chip Flash UID    : 31373237304A23094330FFFFFFFFFFFF
- Chip Flash Size   : 1024 KB
LL Spark Controller Version:b0e99c4

[15:32:15.491] ble_store_config_num_our_secs:0
ble_store_config_num_peer_secs:0
ble_store_config_num_cccds:0
registered service 0x1800 with handle=1
registering characteristic 0x2a00 with def_handle=2 val_handle=3
registering characteristic 0x2a01 with def_handle=4 val_handle=5
registered service 0x1801 with handle=6
registering characteristic 0x2a05 with def_handle=7 val_handle=8
registered service 0x1811 with handle=10
registering characteristic 0x2a47 with def_handle=11 val_handle=12
registering characteristic 0x2a46 with def_handle=13 val_handle=14
registering characteristic 0x2a48 with def_handle=16 val_handle=17
registering characteristic 0x2a45 with def_handle=18 val_handle=19
registering characteristic 0x2a44 with def_handle=21 val_handle=22
registered service 59462f12-9543-9999-12c8-58b459a2712d with handle=23
registering characteristic 33333333-2222-2222-1111-111100000000 with def_handle=24 val_
↪handle=25
registering descriptor 34343434-2323-2323-1212-121201010101 with handle=27
Device Address: 01 02 03 04 05 06


[15:32:19.216] connection established; status=0 handle=1 our_ota_addr_type=0 our_ota_addr=01␣
↪02 03 04 05 06
```

(下页继续)

(续上页)

```
our_id_addr_type=0 our_id_addr=01 02 03 04 05 06
peer_ota_addr_type=0 peer_ota_addr=06 06 03 04 05 06
peer_id_addr_type=0 peer_id_addr=06 06 03 04 05 06
conn_itvl=40 conn_latency=0 supervision_timeout=256 encrypted=0 authenticated=0 bonded=0


[15:32:22.021] subscribe event; conn_handle=1 attr_handle=19 reason=1 prevn=0 curn=1 previ=0␣
↪curi=0
```

1. 具体 app 功能实现可以参考 BLE Central 和 BLE Peripheral ENC 的文档。
2. 多连接相关：
   - 对于 controller 层,：指示 origin controller 多连接资源的宏 CONFIG_BT_CTLR_MAX_NUM_OF_STATES；指示 spark controller 多连接资源的宏 CONFIG_BT_CTLR_MAX_MST_CONN 和 CONFIG_BT_CTLR_MAX_SLV_CONN.
   - 对于 host 层：MYNEWT_VAL_BLE_MAX_CONNECTIONS 会影响 host 层的连接资源数量。

5 RAM/Flash 资源使用情况

PAN107x:

```
RAM Size:37.84 k
Flash Size: 129.54k
```

3.1.2 BLE Central

1 功能概述

此项目演示蓝牙主机功能，通过扫描其他 BLE 设备，并通过特定的服务 UUID 进行识别，此处是 ANS 服务 0x1811。作为对端，可以直接使用 bleprph_enc 进行编译下载另外一个 EVB 上和主机 sample 完成测试。

2 环境要求

- board: pan107x evb
- uart(option): 用来显示串口 log（波特率 921600，选项 8n1）

3 编译和烧录

例程位置：<home>\nimble\samples\bluetooth\ble_central\keil_107x

使用 keil 进行打开项目进行编译烧录。

4 演示说明

1. 烧录完成后，设备会一直打印收到的广播信息，连接上会显示 Connection established, 服务发现完成后输出 Service discovery complete。

```
[10:48:46.565]LL Controller Version:bd5923c

[10:48:46.603]ble_store_config_num_our_secs:0
ble_store_config_num_peer_secs:0
```

(下页继续)

(续上页)

```
ble_store_config_num_cccds:0

[10:48:46.673]flags=0x06
    uuids16(complete)=0xe0ff
    mfg_data=0x53:0x50:0x04:0x11:0x23:0x00:0x00:0x00:0x00:0x60:0x13
    flags=0x06
    name(complete)=QHM-C109
    mfg_
↪data=0x06:0x00:0x01:0x09:0x20:0x02:0x3f:0x6b:0x8e:0x3d:0x22:0x86:0x72:0xf0:0x67:0x3e:0x52:0x1f:0x94:0xe1
    flags=0x1a
    tx_pwr_lvl=12
    mfg_data=0x4c:0x00:0x10:0x05:0x1c:0x18:0x03:0x6b:0xee
    flags=0x06
    uuids16(complete)=0xe0ff
    mfg_data=0x53:0x50:0x04:0x11:0x23:0x00:0x00:0x00:0x00:0x60:0x13
    flags=0x06
    name(complete)=QHM-C109
    flags=0x06
    uuids16(complete)=0xe0ff
    mfg_data=0x53:0x50:0x04:0x11:0x23:0x00:0x00:0x00:0x00:0x60:0x13
    flags=0x06
    name(complete)=QHM-C109
.....................................

[10:48:48.154]flags=0x06
    name(complete)=QHM-C109
    flags=0x06
    uuids16(complete)=0xe0ff
    mfg_data=0x53:0x50:0x04:0x11:0x23:0x00:0x00:0x00:0x00:0x60:0x13
    flags=0x06
    uuids16(complete)=0x1811
    name(complete)=nimble-bleprph
    tx_pwr_lvl=0

[10:48:48.242]Connection established handle=0 our_ota_addr_type=0 our_ota_
↪addr=06:05:04:03:06:06 our_id_addr_type=0 our_id_addr=06:05:04:03:06:06 peer_ota_addr_
↪type=0 peer_ota_addr=06:05:04:03:02:01 peer_id_addr_type=0 peer_id_
↪addr=06:05:04:03:02:01 conn_itvl=32 conn_latency=0 supervision_timeout=256 encrypted=0␣
↪authenticated=0 bonded=0

[10:48:50.397]Service discovery complete; status=0 conn_handle=0

[10:48:50.515]Read complete; status=0 conn_handle=0 attr_handle=12 value=0x00
Write complete; status=270 conn_handle=0 attr_handle=22
Subscribe complete; status=0 conn_handle=0 attr_handle=20
```

2. 断链时会有如下输出。

```
disconnect; reason=0x08
handle=0 our_ota_addr_type=0 our_ota_addr=06:05:04:03:06:06 our_id_addr_type=0 our_id_
↪addr=06:05:04:03:06:06 peer_ota_addr_type=0 peer_ota_addr=06:05:04:03:02:01 peer_id_
↪addr_type=0 peer_id_addr=06:05:04:03:02:01 conn_itvl=32 conn_latency=0 supervision_
↪timeout=256 encrypted=0 authenticated=0 bonded=0
```

3. **广播显示和连接过滤调整**

如果广播打印太过频繁，影响查看相关 log 可以手动关掉：

```
blecent_gap_event(struct ble_gap_event *event, void *arg)
{
    struct ble_gap_conn_desc desc;
    struct ble_hs_adv_fields fields;
```

(下页继续)

(续上页)

```
    int rc;

    switch (event->type) {
    case BLE_GAP_EVENT_DISC:
        rc = ble_hs_adv_parse_fields(&fields, event->disc.data,
                                     event->disc.length_data);
        if (rc != 0) {
            return 0;
        }

        /* An advertisment report was received during GAP discovery. */
        print_adv_fields(&fields); /* 此函数用于输出广播数据，可以屏蔽盖函数关掉 */

        /* Try to connect to the advertiser if it looks interesting. */
        blecent_connect_if_interesting(&event->disc); /* 连接时的过滤函数 */
        return 0;
```

连接时的过滤条件 blecent_connect_if_interesting:

```
/**
 * Connects to the sender of the specified advertisement of it looks
 * interesting.  A device is "interesting" if it advertises connectability and
 * support for the Alert Notification service.
 */
static void
blecent_connect_if_interesting(const struct ble_gap_disc_desc *disc)
{
    uint8_t own_addr_type;
    int rc;

    /*Filter adv by rssi*/
    if (disc->rssi < -70) /* 此处可以调整 RSSI 进行过滤 */
    {
        return;
    }

    /* Don't do anything if we don't care about this advertiser. */
    if (!blecent_should_connect(disc)) {
        return;
    }

    /* Scanning must be stopped before a connection can be initiated. */
    rc = ble_gap_disc_cancel();
    if (rc != 0) {
        printf("Failed to cancel scan; rc=%d\n", rc);
        return;
    }

    /* Figure out address to use for connect (no privacy for now) */
    rc = ble_hs_id_infer_auto(0, &own_addr_type);
    if (rc != 0) {
        printf("error determining address type; rc=%d\n", rc);
        return;
    }

    /* Try to connect the the advertiser.  Allow 30 seconds (30000 ms) for
     * timeout.
     */
    rc = ble_gap_connect(own_addr_type, &disc->addr, 30000, NULL,
                         blecent_gap_event, NULL);
```

(下页继续)

(续上页)

```
    if (rc != 0) {
        printf("Error: Failed to connect to device; addr_type=%d "
                           "addr=%s\n; rc=%d",
                    disc->addr.type, addr_str(disc->addr.val), rc);
        return;
    }
}
```

5 RAM/Flash 资源使用情况

PAN107x:

```
RAM Size:36.40 k
Flash Size: 126.91k
```

3.1.3 BLE MULTI ROLE

1 功能概述

此项目演示蓝牙多主多从功能以及多个连接设备之间的数据交互。此工程提供一个设备支持 3 主 2 从（也即一个设备作为 Master role 可以同时连接 Peer 的 3 个 Slave Role 的设备，同时作为 Slave Role 可以同时被 Peer 的 2 个 Master Role 的设备连接，也即 6 个设备同时通讯）

工程默认的配置是 3 主 2 从。

2 环境要求

- board: `pan107x evb`

3 工程路径

例程位置：`<home>\nimble\samples\bluetooth\ble_multi_role\keil_107x`

使用 `keil` 进行打开项目进行编译烧录。

4 演示说明

4.1 编译 3 主 2 从固件 初次打开工程默认配置就是 3 主 2 从，用户需要配置如下参数：

3 主 2 从作为 Master Role 时是通过蓝牙地址和特定的 Server 来自动建立连接的，因此，为了测试，用户需要提供 3 个 Peer Slave Role 的蓝牙地址并填在如下表中：

当然，用户也可以不用修改，只需要将表中的地址设置到🄵测的 Slave Role 即可。

4.2 编译对测的 Slave Role 固件 为了与 3 主 2 从设备对测，需要编译 3 个对测的 Slave Role 固件。用户需要做如下修改：

1. 设置 APP_SLV_TEST_EN=1，设置 APP_SLV_DEVICE_ID 分别为 0,1,2，对应 3 个 Slave 设备。
1. 按下图设置 BT_MAX_NUM_OF_PERIPHERAL=1，BT_MAX_NUM_OF_CENTRAL=0
1. 编译 3 个🄵测的 Slave Role 固件，**需要注意的是，编译时需要修改** APP_SLV_DEVICE_ID =0,1,2**（这与蓝牙地址相关）对应 3 个🄵测** Slave Role **固件**

main.c

```
43
44  /* Application-specified header. */
45  #include "blecent.h"
46  #include "blehr_sens.h"
47  #include "gw_svc.h"
48
49
50  #define APP_DATA_TEST_EN     1
51  #define APP_DATA_TOKEN       0xAA
52
53  /* Connection parameter config */
54  #define APP_MST_CONN_INTERVAL  BLE_GAP_CONN_ITVL_MS(50)
55  #define APP_SLV_CONN_INTERVAL  BLE_GAP_CONN_ITVL_MS(30)
56  #define APP_CONN_TIMEOUT       BLE_GAP_SUPERVISION_TIMEOUT_MS(2500)
57
58  /* Test Config */
59  #define APP_SLV_TEST_EN      0 /* Need Set CONFIG_BT_MAX_NUM_OF_PERIPHERAL = 1 && CONFIG_BT_MAX_NUM_OF_CENTRAL = 0. */
60  #define APP_MST_TEST_EN      0 /* Need Set CONFIG_BT_MAX_NUM_OF_PERIPHERAL = 0 && CONFIG_BT_MAX_NUM_OF_CENTRAL = 1. */
61
62  #define APP_SLV_DEVICE_ID    0 /* Slave device can be 0,1,2 */
63
64  #if APP_MST_TEST_EN
65  bdAddr_t addrFilterTable [] = {
66      {0xD0, 0x00, 0x00, 0x00, 0x01, 0x39},
67  };
68  #else
69  bdAddr_t addrFilterTable [] ={
70      {0x66, 0x66, 0x66, 0x66, 0x01, 0xC2},
71      {0x66, 0x66, 0x66, 0x66, 0x03, 0xC2},
72      {0x66, 0x66, 0x66, 0x66, 0x04, 0xC2},
73  };
74  #endif
75
76  /* Need Set CONFIG_BT_MAX_NUM_OF_PERIPHERAL = 1 && CONFIG_BT_MAX_NUM_OF_CENTRAL = 0. */
77  #if APP_SLV_TEST_EN
78      static const char *device_name = "mutil_conn_test_S";
79      #undef APP_DATA_TOKEN
80      #define APP_DATA_TOKEN       0x77
81  /* Need Set CONFIG_BT_MAX_NUM_OF_PERIPHERAL = 0 && CONFIG_BT_MAX_NUM_OF_CENTRAL = 1. */
```

图 1: Peer Slave Role Address Table

main.c app_config_spark.h

```
46  #include "blehr_sens.h"
47  #include "gw_svc.h"
48
49
50  #define APP_DATA_TEST_EN     1
51  #define APP_DATA_TOKEN       0xAA
52
53  /* Connection parameter config */
54  #define APP_MST_CONN_INTERVAL  BLE_GAP_CONN_ITVL_MS(50)
55  #define APP_SLV_CONN_INTERVAL  BLE_GAP_CONN_ITVL_MS(30)
56  #define APP_CONN_TIMEOUT       BLE_GAP_SUPERVISION_TIMEOUT_MS(2500)
57
58  /* Test Config */
59  #define APP_SLV_TEST_EN      1 /* Need Set CONFIG_BT_MAX_NUM_OF_PERIPHERAL = 1 && CONFIG_BT_MAX_NUM_OF_CENTRAL = 0. */
60  #define APP_MST_TEST_EN      0 /* Need Set CONFIG_BT_MAX_NUM_OF_PERIPHERAL = 0 && CONFIG_BT_MAX_NUM_OF_CENTRAL = 1. */
61
62  #define APP_SLV_DEVICE_ID    0 /* Slave device can be 0,1,2 */
63
64  #if APP_MST_TEST_EN
65  bdAddr_t addrFilterTable [] = {
66      {0xD0, 0x00, 0x00, 0x00, 0x01, 0x39},
67  };
68  #else
69  bdAddr_t addrFilterTable [] ={
70      {0x66, 0x66, 0x66, 0x66, 0x01, 0xC2},
71      {0x66, 0x66, 0x66, 0x66, 0x03, 0xC2},
72      {0x66, 0x66, 0x66, 0x66, 0x04, 0xC2},
73  };
```

图 2: 对测 slave role 配置

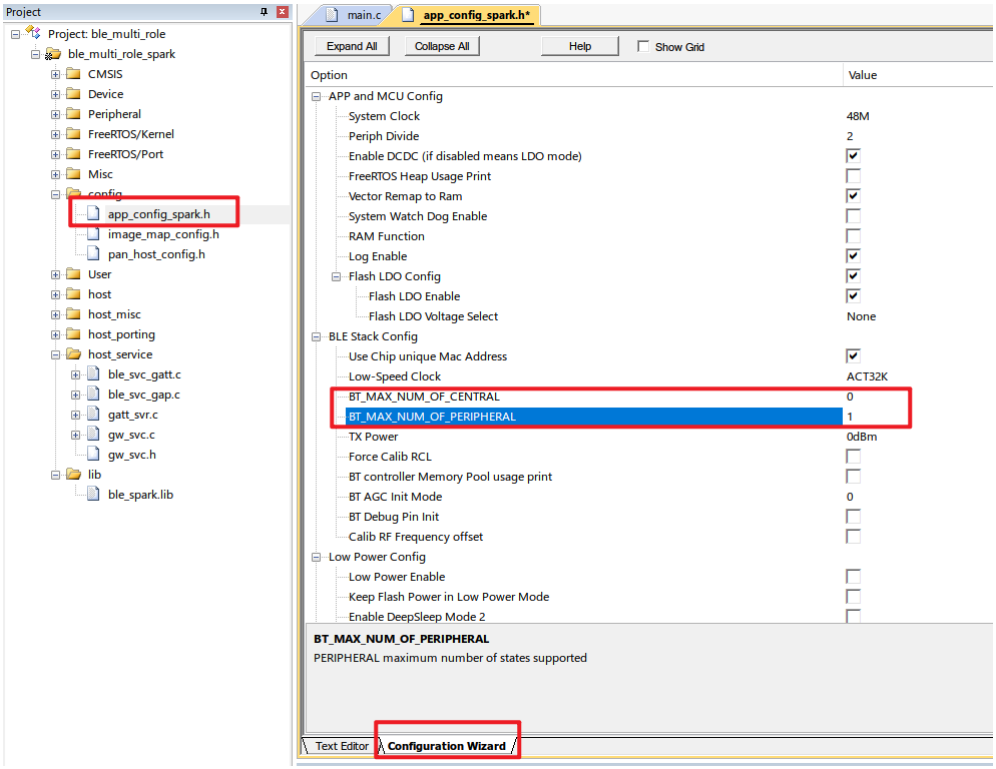


图 3: 对测 slave 设备数量配置

4.3 **编译对测的** Master Role **固件**　为了与 3 主 2 从设备对测，需要编译 2 个对测的 Master Role 固件。用户需要做如下修改：

1. 设置 APP_MST_TEST_EN=1，将 3 主 2 从设备的蓝牙地址填到如图所示的表中
1. 按下图设置 BT_MAX_NUM_OF_PERIPHERAL=0，BT_MAX_NUM_OF_CENTRAL=1
1. 编译工程得到两个对此的 Master Role 固件，**当然，用户也可以使用手机充当** Master **设备**

4.3 **固件烧录**　将编译得到的固件通过 Jlink 分别烧录到对应的 pan107 EVB 板中，上电设备将自动连接，建议先上电 3 主 2 从的设备，然后依次上电其他对测设备。

4.4 **数据交互测试**　设备连接好后（可以通过 log 查看连接请况），用户可以按 pan107 EVB 板上的 KEY1 按键，即可启动数据传输和接收测试。

按键按下以后，3 主 2 从的设备会每间隔 1s 发送数据到对测的 5 个设备，同时会接收对测的 5 个设备发送的数据，并对数据的 counter 以及内容进行检验，如果错误，则会报错，同时通过 log 输出。

其他对测设备按下按键以后，会每隔 1s 发送数据到 3 主 2 从的设备，同时也会接收 3 主 2 从设备发送的数据，并对数据的 counter 以及内容进行检验，如果错误，则会报错，同时通过 log 输出。

5 RAM/Flash **资源使用情况**

PAN107x:

```
RAM Size:30.77 k
Flash Size: 130.88k
```

3.1.4 BLE Distance

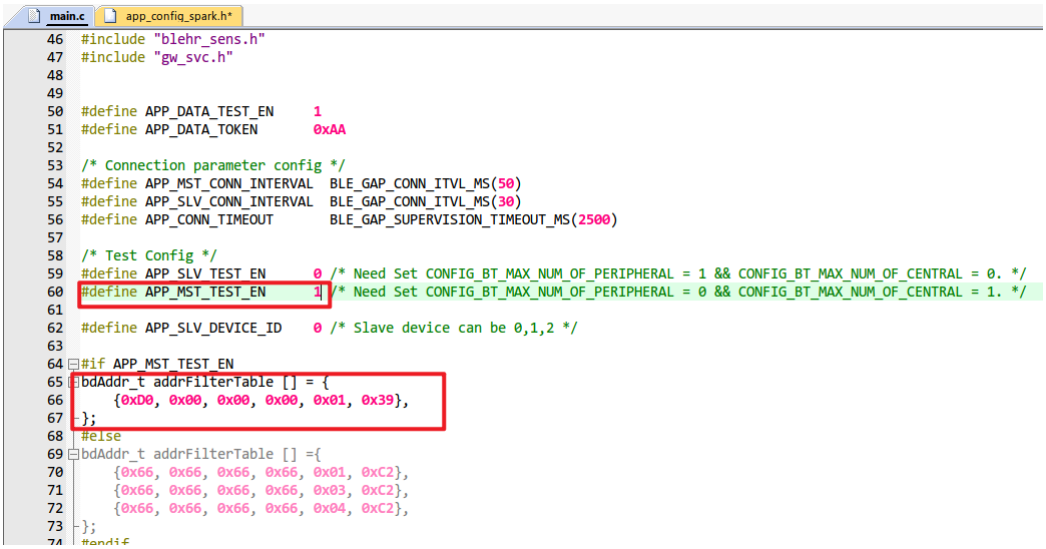


图 4: 对测 Master 设备配置

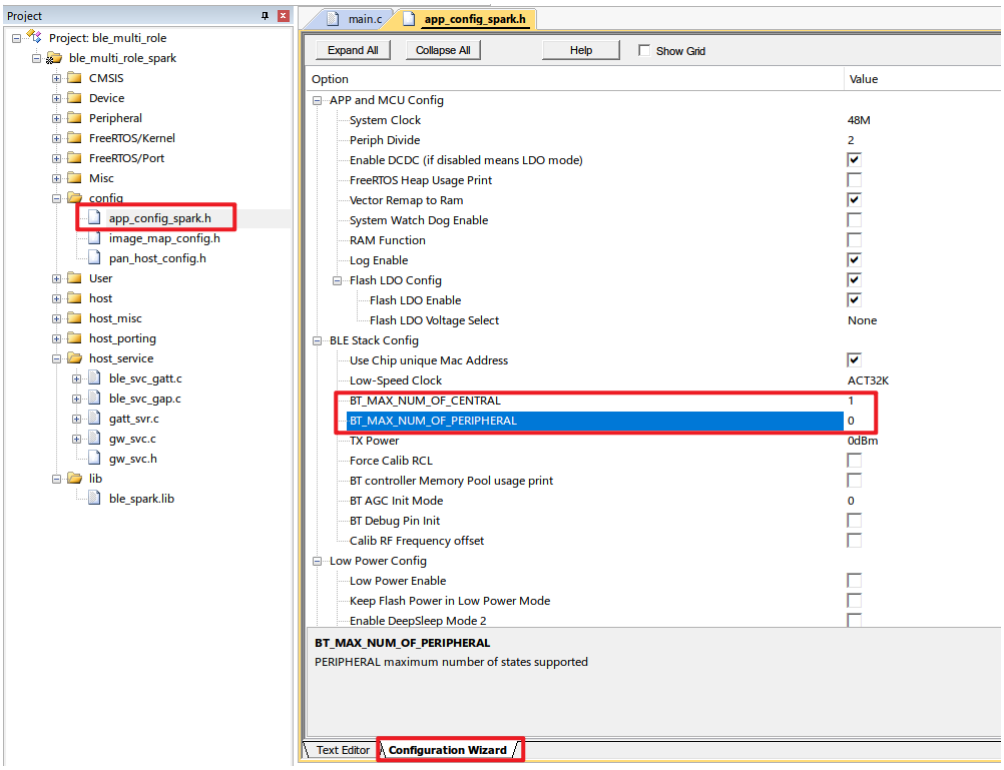


图 5: 对测 Master 设备数量配置

1 功能概述

此项目演示从机 heartrate 服务, 可以配合手机 `nrf connect` 进行距离演示。基本功能和 heartrate 从机功能类似，在其基础上增加了写配合调整 phy 的动作, 主要是 S8 coded。

2 环境要求

- board: `pan107x evb`
- uart(option): 用来显示串口 log（波特率 921600，选项 `8n1`）
- 手机 app `nrf connect`

3 编译和烧录

例程位置：`<home>\nimble\samples\bluetooth\bleprph_distance\keil_107x`

使用 `keil` 进行打开项目进行编译烧录。

4 演示说明

1. 烧录完成后，设备会显示上电 log，连接上会显示 `Connection established`, 主机订阅完成后输出 `subscribe event;` 。

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 4
- Chip MAC Address  : D000000001E5
- Chip UID          : E501017FFD375603B8
- Chip Flash UID    : 4250315632333917017FFD375603B878
- Chip Flash Size   : 512 KB
LL Spark Controller Version:d7c4bfa
app started
APP version: 1.240.65406
connection established; status=0
```

2. 使用手机 `nrf connect` 扫描蓝牙设备名称 `ble_distance` 并且连接
3. 设置相应的 phy 并且连接

需要注意的是，对于 1M，2M, S2 模式直接使用 nrf connect app 操作接口，但是 s8 模式需要 EVB 板进行配合操作下。首先确认 EVB 上的 RGB 的跳线帽是否连接，然后按 3 次 `key1` 键使得 RGB 颜色变成蓝色，然后再次使用 nrf connect 连接，切换到 s8 模式即可。

5 RAM/Flash 资源使用情况

PAN107x:

```
RAM Size:35.14 k
Flash Size: 114.38k
```

3.1.5 BLE Peripheral ENC

1 功能概述

此项目演示从机 ANS 服务以及自定义加密特性演示功能，可以配合主机 sample`ble central` 演示主从连接，同时也可以配合手机 `nrf connect` 演示配对加密功能。

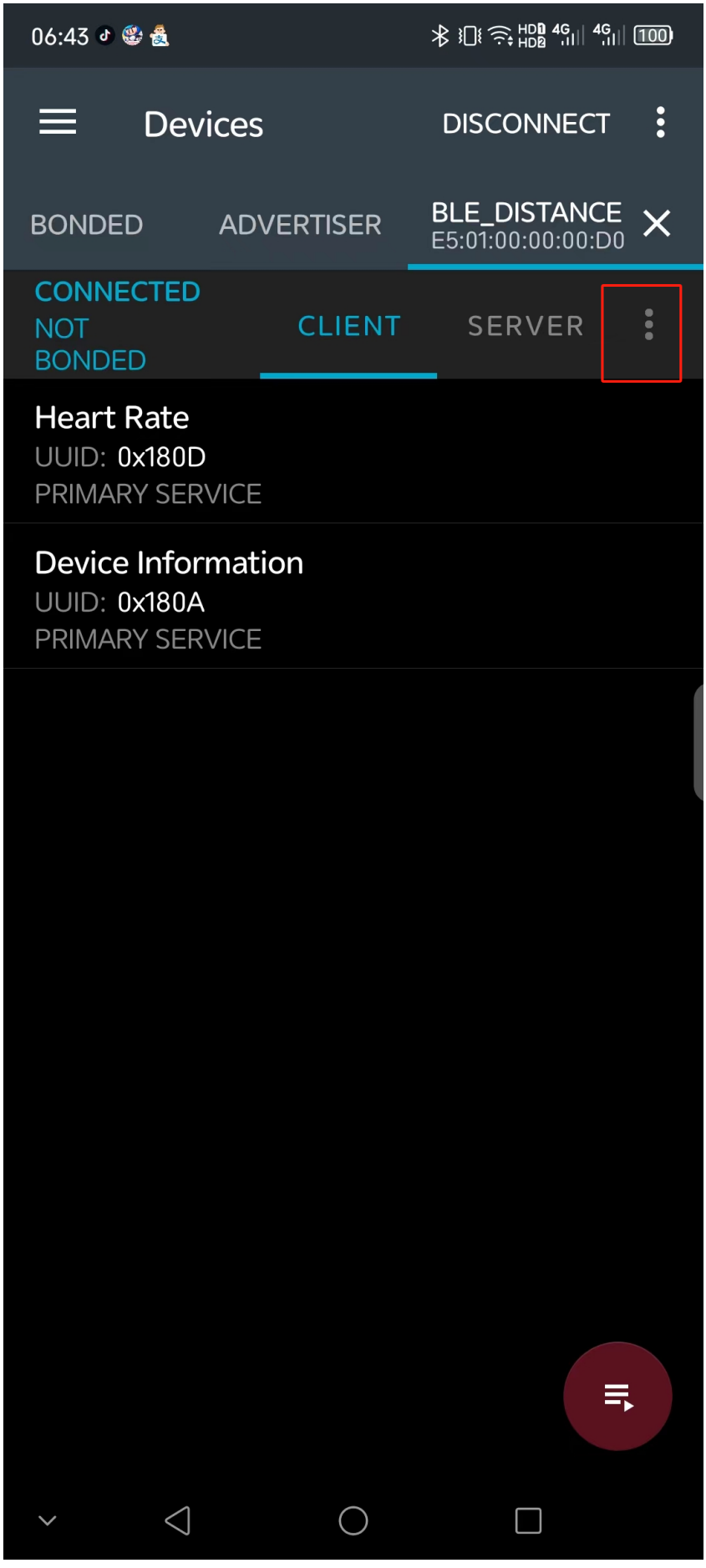


图 6: nrf connect 连接 ble_distance

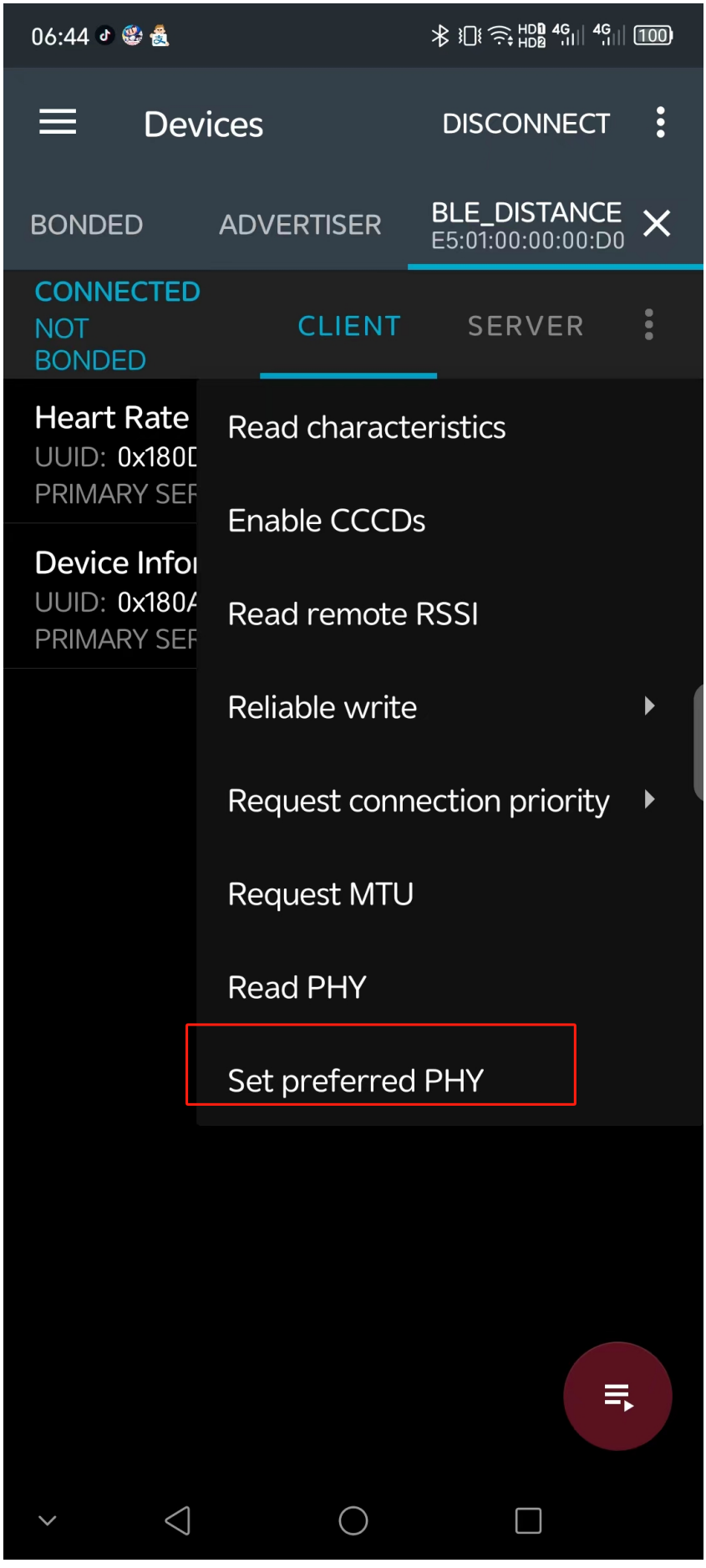


图 7: nrf connect 设置 phy

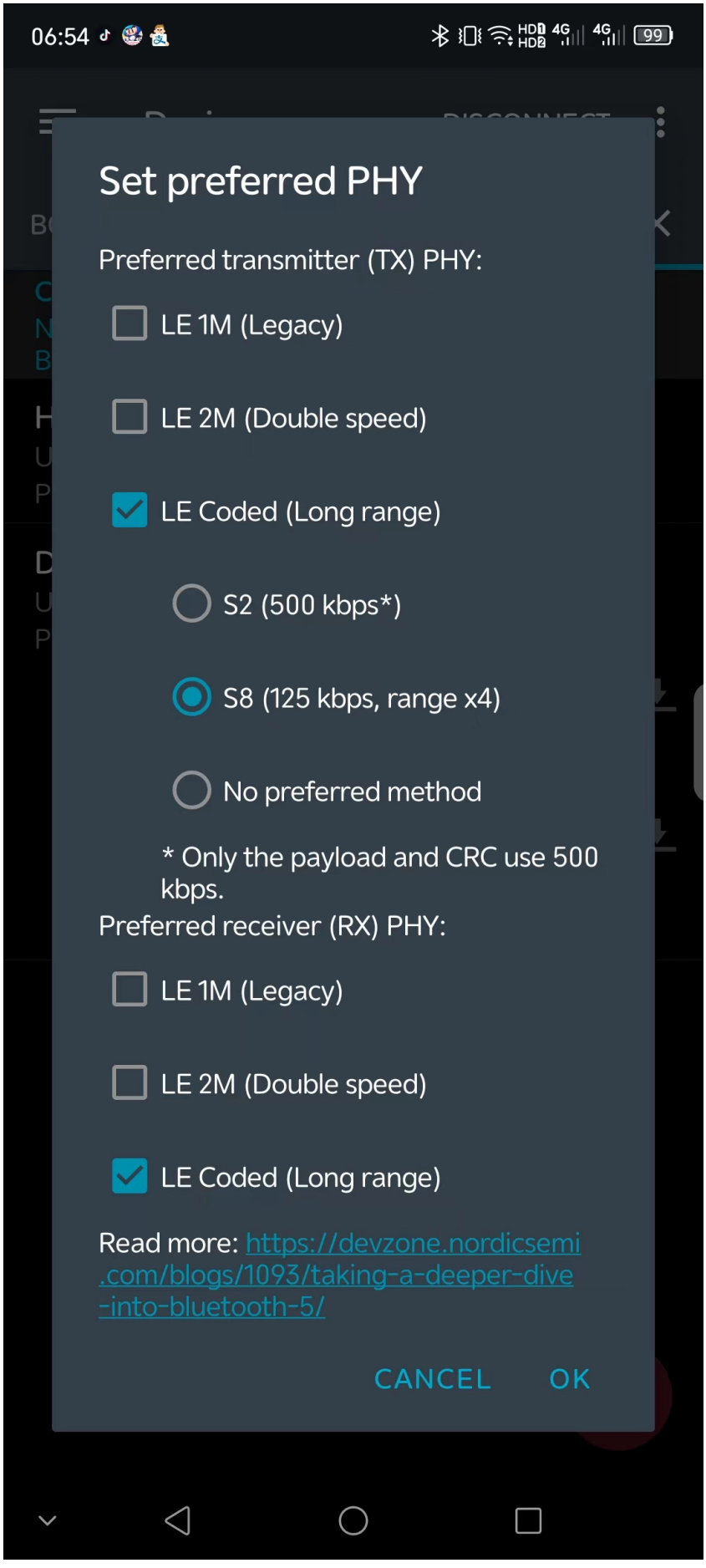


图 8: nrf connect 设置 S8

2 环境要求

- board: pan107x evb
- uart(option): 用来显示串口 log（波特率 921600，选项 8n1）
- 手机 app nrf connect

3 编译和烧录

pan107x 芯片例程位置：<home>\nimble\samples\bluetooth\bleprph_enc\keil_107x

pan101x 芯片例程位置：<home>\nimble\samples\bluetooth\bleprph_enc\keil_101x

使用 keil 进行打开项目进行编译烧录。

4 演示说明

1. 烧录完成后，设备会显示上电 log，连接上会显示 Connection established, 主机订阅完成后输出 subscribe event; 。

```
[11:46:57.699]LL Controller Version:bd5923c

[11:46:57.736]ble_store_config_num_our_secs:0
ble_store_config_num_peer_secs:0
ble_store_config_num_cccds:0
registered service 0x1800 with handle=1
registering characteristic 0x2a00 with def_handle=2 val_handle=3
registering characteristic 0x2a01 with def_handle=4 val_handle=5
registered service 0x1801 with handle=6
registering characteristic 0x2a05 with def_handle=7 val_handle=8
registered service 0x1811 with handle=10
registering characteristic 0x2a47 with def_handle=11 val_handle=12
registering characteristic 0x2a46 with def_handle=13 val_handle=14
registering characteristic 0x2a48 with def_handle=16 val_handle=17
registering characteristic 0x2a45 with def_handle=18 val_handle=19
registering characteristic 0x2a44 with def_handle=21 val_handle=22
registered service 59462f12-9543-9999-12c8-58b459a2712d with handle=23
registering characteristic 33333333-2222-2222-1111-111100000000 with def_handle=24 val_
↪handle=25
registering descriptor 34343434-2323-2323-1212-121201010101 with handle=27

[11:46:57.796]Device Address: 01 02 03 04 05 06

[11:47:00.271]connection established; status=0 handle=0 our_ota_addr_type=0 our_ota_
↪addr=01 02 03 04 05 06
 our_id_addr_type=0 our_id_addr=01 02 03 04 05 06
 peer_ota_addr_type=0 peer_ota_addr=06 06 03 04 05 06
 peer_id_addr_type=0 peer_id_addr=06 06 03 04 05 06
 conn_itvl=32 conn_latency=0 supervision_timeout=256 encrypted=0 authenticated=0 bonded=0

[11:47:02.454]subscribe event; conn_handle=0 attr_handle=19 reason=1 prevn=0 curn=1␣
↪previ=0 curi=0
```

2. 使用手机 nrf connect 扫描蓝牙设备名称 nimble-bleprph 并且连接

5 RAM/Flash 资源使用情况

PAN107x:

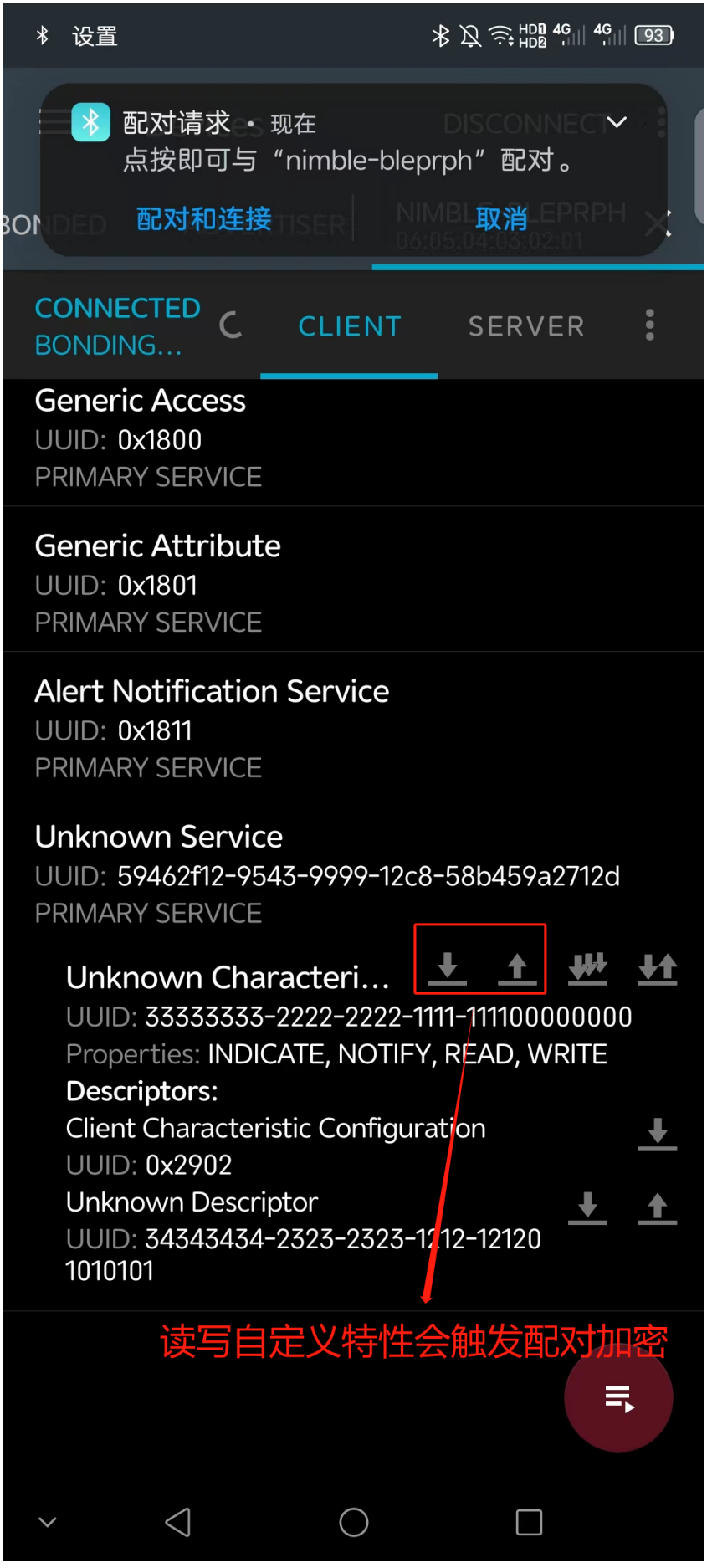


图 9: nrf connect 连接 nimble-bleprph

```
RAM Size:35.17 k
Flash Size: 147.96k
```

PAN101x:

```
RAM Size:14.23 k
Flash Size: 135.34k
```

3.1.6 BLE Peripheral HR

1 功能概述

此项目演示从机 heartrate 服务，可以配合手机 nrf connect 进行演示，此功能支持 pan101x 和 pan107x 芯片，pan101x 芯片在功耗大小和执行速度方面的弱于 pan107x 芯片

2 环境要求

- board: pan107x evb 或 pan101x evb
- uart(option): 用来显示串口 log（波特率 921600，选项 8n1）
- 手机 app nrf connect

3 编译和烧录

pan107x 芯片例程位置：<home>\nimble\samples\bluetooth\bleprph_hr\keil_107x

pan101x 芯片例程位置：<home>\nimble\samples\bluetooth\bleprph_hr\keil_101x

使用 keil 进行打开项目进行编译烧录。

4 演示说明

1. 烧录完成后，设备会显示上电 log，连接上会显示 Connection established, 主机订阅完成后输出 subscribe event; 。

   ```
   [13:17:18.158]LL Controller Version:bd5923c

   [13:17:18.197]app started

   [13:18:20.460]connection established; status=0

   [13:18:26.943]subscribe event; cur_notify=1
    value handle; val_handle=3
   ```

2. 使用手机 nrf connect 扫描蓝牙设备名称 ble_hr 并且连接

5 RAM/Flash 资源使用情况

PAN107x:

```
RAM Size:33.64 k
Flash Size: 116.52k
```

PAN101x:

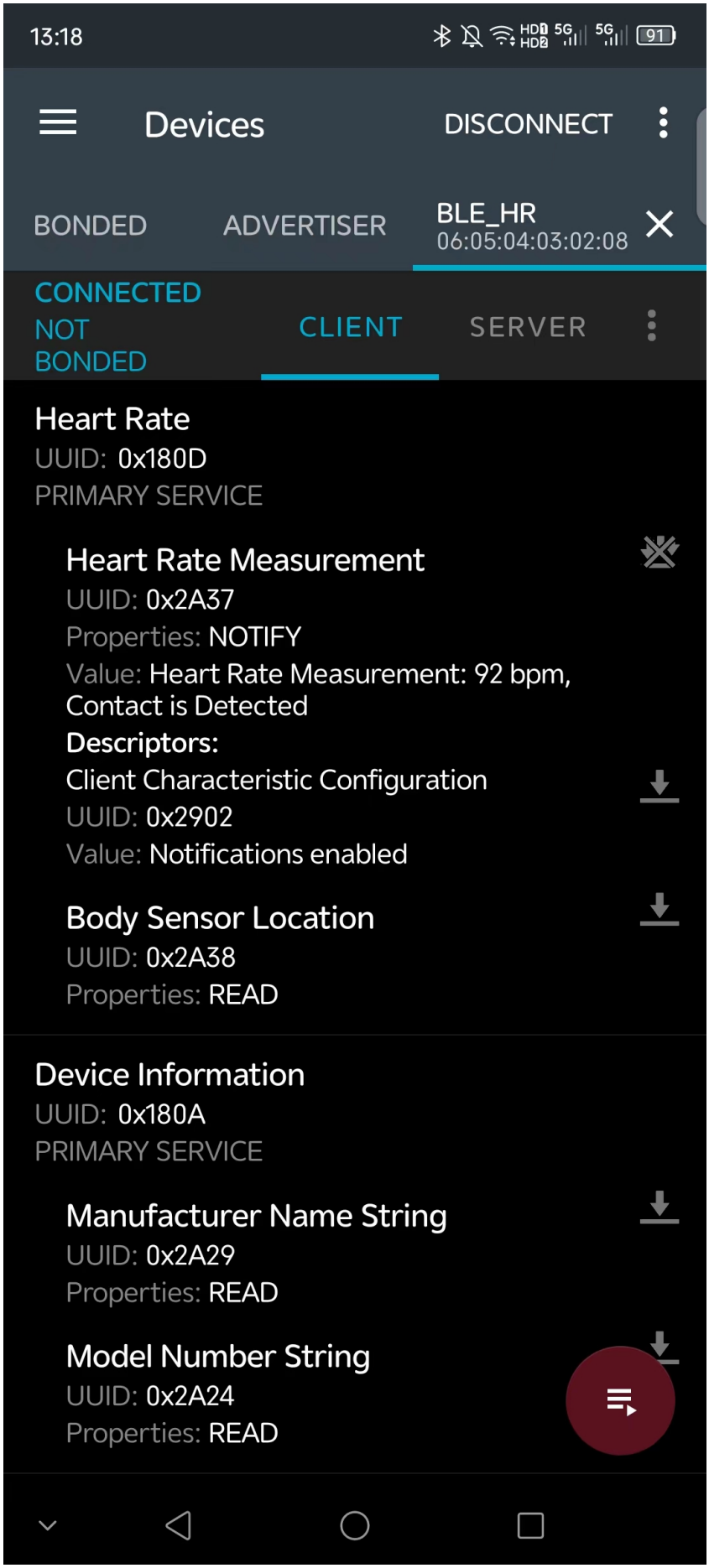


图 10: nrf connect 连接 ble_hr

```
RAM Size:12.91 k
Flash Size: 99.89k
```

3.1.7 BLE Peripheral HR OTA

1 功能概述

此项目演示从机 heartrate 以及用于 BLE 升级的 SMP 服务, 可以配合手机 nrf connect 进行演示该 OTA 功能。

2 环境要求

- board: pan107x evb
- uart(option): 用来显示串口 log（波特率 921600，选项 8n1）
- 手机 app nrf connect

3 编译和烧录

例程位置：<home>\nimble\samples\bluetooth\bleprph_hr_ota\keil_107x

使用 keil 进行打开项目进行编译烧录。

4 演示说明

1. 分别编译和烧录 pan107x_mcu_boot 和 bleprph_hr_ota 工程，上电后 log 如下。

```
Try to load HW calibration data.. DONE.
- Chip Type          : 0x80
- Chip CP Version    : None
- Chip FT Version    : 8
- Chip MAC Address   : D0000C0CBBF5
- Chip Flash UID     : 32313334320EAC834330FFFFFFFFFFFF
- Chip Flash Size    : 512 KB
LL Spark Controller Version:d7c4bfa
app started
```

2. 然后编译用于升级的测试固件，任意工程均可用于升级，但是其他工程暂无 OTA 功能，为了模拟实际应用中的连续更新 OTA 升级，我们将 bleprph_hr_ota 修改启动 log，以便生成的固件不一样。如果升级的固件和运行的固件是一样的，nrf connectapp 检查校验签名一致则不进行升级，**校验签名和** bin **文件内容有关**。

```
void app_main(void)
{
    int rc;

    printf("app started ota success\n");/* 此处为修改处，升级后将打印该 log*/

    #if CONFIG_SMP_OTA
    img_mgmt_module_init();
    #endif
```

3. 生成的 image 在路径 bleprph_hr_ota\keil\Images 路径下，找到 ndk_app.signed.bin 文件，将该文件导入到手机 APP。
4. nrf connect 扫描连接 ble_hr 设备，连接上会显示 SMP Service，同时右上角会 DFU 标示。点击该 DFU 标示选则待升级的 “ndk_app.signed.bin‘文件，点击 Test and Confirm, 启动 OTA 升级流程。

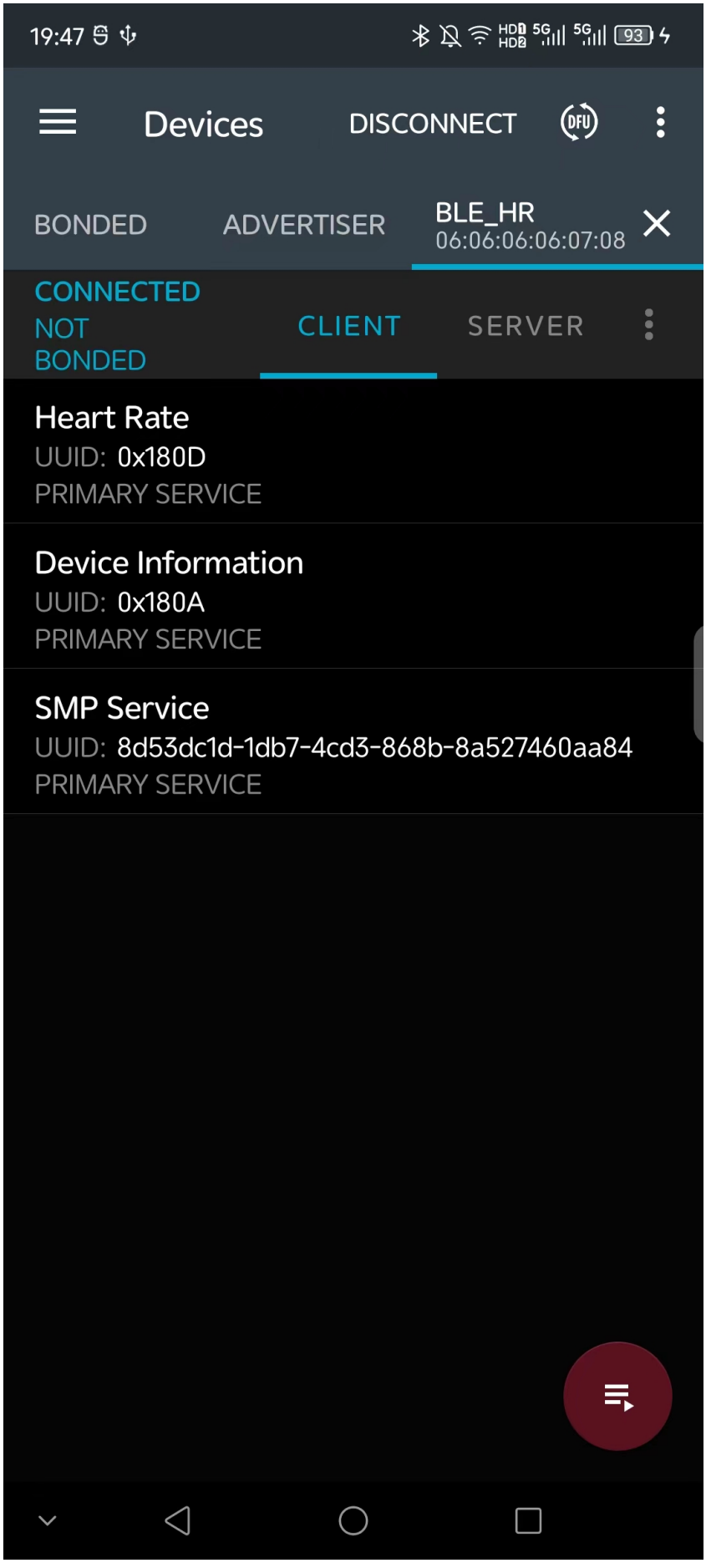


图 11: nrf connect 连接 ble_hr, 显示 SMP 服务

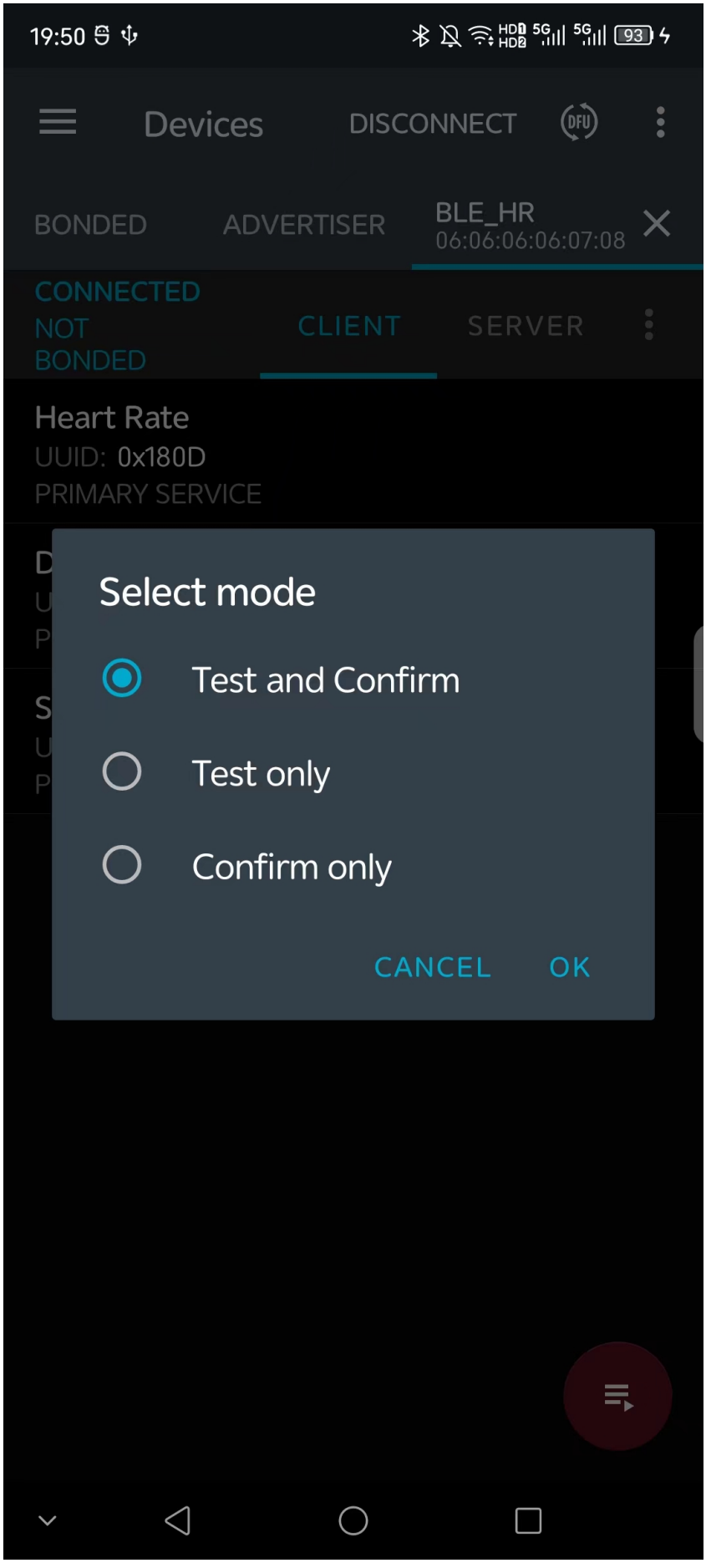


图 12: nrf connect 选则待升级文件

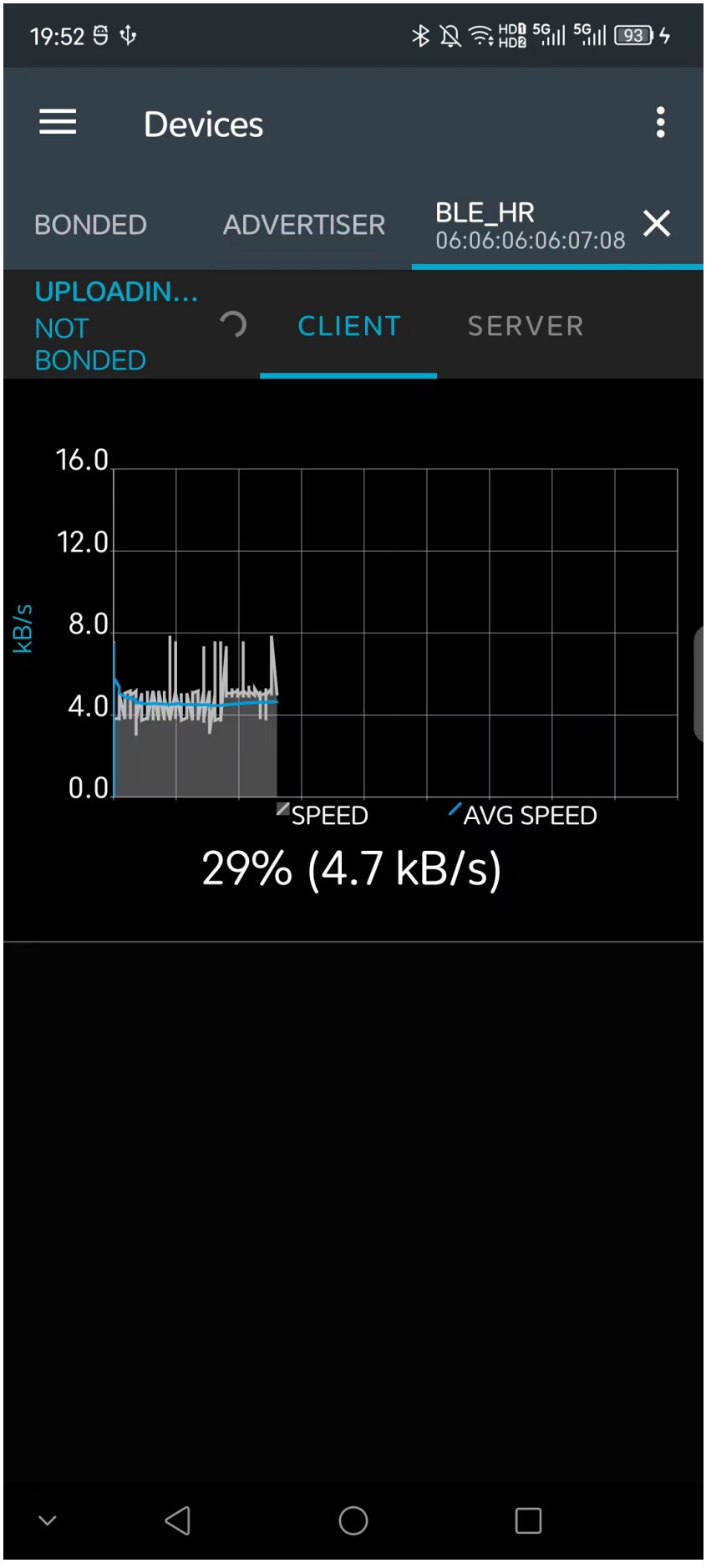


图 13: nrf connect SMP OTA 升级中

5. 升级完成后设备会自动复位，并且打印对应 bin 文件启动 log。

```
Try to load HW calibration data.. DONE.
- Chip Type          : 0x80
- Chip CP Version    : None
- Chip FT Version    : 8
- Chip MAC Address   : D0000C0CBBF5
- Chip Flash UID     : 32313334320EAC834330FFFFFFFFFFF
- Chip Flash Size    : 512 KB
LL Spark Controller Version:d7c4bfa
app started ota success /*此处为我们修改后的启动 log*/
connection established; status=0
```

5 RAM/Flash 资源使用情况

PAN107x:

```
RAM Size:34.11 k
Flash Size: 120.33k
```

3.1.8 BLE Peripheral Throughput Test

1 功能概述

此项目演示从机吞吐率测试, 当前版本软件需要 `nrf connect dongle` 或者手机 `nrf_connection` 进行演示。

2 环境要求

- board: `pan107x evb`
- uart(option): 用来显示串口 log（波特率 921600，选项 `8n1`）
- `nrf connect dongle`

3 编译和烧录

例程位置: `<home>\nimble\samples\bluetooth\bleprph_throughput\keil_107x`

使用 `keil` 进行打开项目进行编译烧录。

4 演示说明

1. 烧录完成后, 设备会显示上电 log, 使用 `pc nrf connect` 连接即可, 当然也可用 `nrf_connection` 软件进行操作，方法基本一样，这儿只介绍 `pc nrf connect` 操作

```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 4
- Chip MAC Address   : D0000000066D
- Chip UID           : 6D06010CF8375603CE
- Chip Flash UID     : 42503156323339170 10CF8375603CE78
- Chip Flash Size    : 512 KB

LL Spark Controller Version:d7c4bfa
BT controller memory pool used: 5624 bytes, remain bytes: 0, total:5624
```

(下页继续)

(续上页)

```
app started
APP version: 104.70.30977

registered service 0x1800 with handle=1
registering characteristic 0x2a00 with def_handle=2 val_handle=3
registering characteristic 0x2a01 with def_handle=4 val_handle=5
registered service 0x1801 with handle=6
registering characteristic 0x2a05 with def_handle=7 val_handle=8
registered service 00000001-8c26-476f-89a7-a108033a69c7 with handle=10
registering characteristic 00000006-8c26-476f-89a7-a108033a69c7 with def_handle=11 val_
↪handle=12
registering characteristic 0000000a-8c26-476f-89a7-a108033a69c7 with def_handle=13 val_
↪handle=14
registering characteristic 0000000b-8c26-476f-89a7-a108033a69c7 with def_handle=16 val_
↪handle=17
gatts_on_sync
Device Address: 06:09:09:09:09:08adv start
```

2. 使用软件 nrf connect for Desktop Bluetooth Low Energy 扫描蓝牙设备名称 ble_throughput 设备并且连接

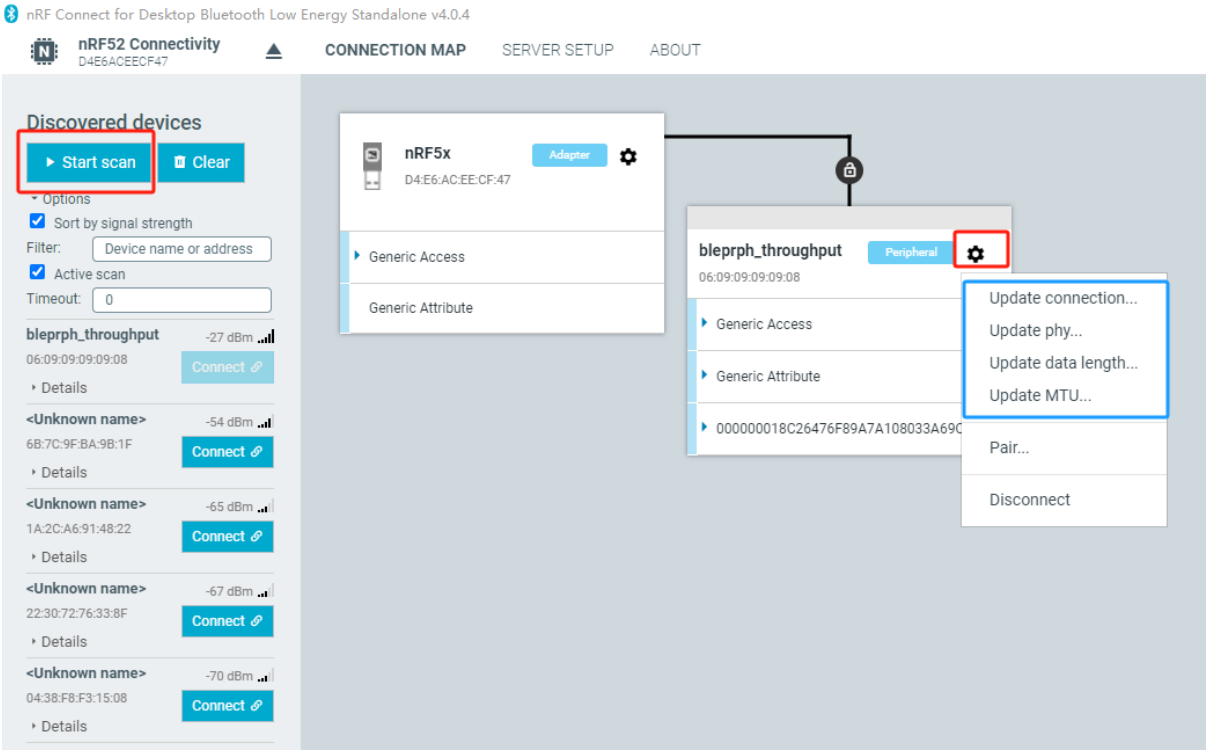


图 14: nrf connect 连接 throughput 设备

3. 点击齿轮设置图标，

依次需要设置

1. Update Connection, 设置连接间隔为 15ms （手机可以忽略此操作）。
2. Update phy, 根据测试需要设置为 1M 或者 2M。
3. Update data length, 设置为 251bytes，**若不能更新 DLE（例如用手机进行操作），MTU 请设置** 210
4. Update MTU, 设置为 247bytes。
5. 订阅 notify，进行测试，串口打印测试结果。

6. 重新测试请关闭订阅 notify，然后再重新订阅。

PHY 模式	参考速率
1M	630113 bps
2M	371850 bps,

5 RAM/Flash **资源使用情况**

PAN107x:

```
RAM Size:40.98 k
Flash Size: 112.18k
```

3.2 低功耗例程

3.2.1 DeepSleep GPIO Key Wakeup

1 功能概述

本例程演示如何使 SoC 进入 DeepSleep 状态，然后通过 GPIO 按键将其唤醒。

2 环境准备

- 硬件设备与线材：
  - PAN107X EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - **电流计**（本文使用电流可视化测量设备 PPK2 [Nordic Power Profiler Kit II] 进行演示）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 为确保能够准确地测量 SoC 本身的功耗，排除底板外围电路的影响，请确认 EVB 底板上的：
    - Voltage 排针组中的 VCC 和 VDD 均接至 3V3
    - POWER 开关从 LDO 档位拨至 BAT 档位（并确认底板背部的电池座内**没有**纽扣电池）
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将 JLink 仿真器的：
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
  - 将 PPK2 硬件的：
    - USB DATA/POWER 接口连接至 PC USB 接口
    - VOUT 连接至 EVB 底板的 VBAT 排针
    - GND 连接至 EVB 底板的 GND 排针

- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）
  - nRF Connect Desktop（用于配合 PPK2 测量 SoC 电流）

3 编译和烧录

例 程 位 置: `<PAN10XX-NDK>\01_SDK\nimble\samples\low_power\deepsleep_gpio_key_wakeup\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录，烧录成功后断开 JLink 连线以避免漏电。

4 例程演示说明

1. PC 上打开 PPK2 Power Profiler 软件，供电电压选择 3300 mV，然后打开供电开关
2. 从串口工具中看到如下的打印信息:

```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 6
- Chip MAC Address   : E11000000FF8
- Chip UID           : 6D00014654544553 54
- Chip Flash UID     : 4250315A3538380B005B7B4356037D78
- Chip Flash Size    : 512 KB
APP version: 255.255.65535
Wait for Task Notifications..
```

3. 此时观察芯片电流波形，发现稳定在 4uA 左右（说明芯片成功进入了 DeepSleep 模式）:

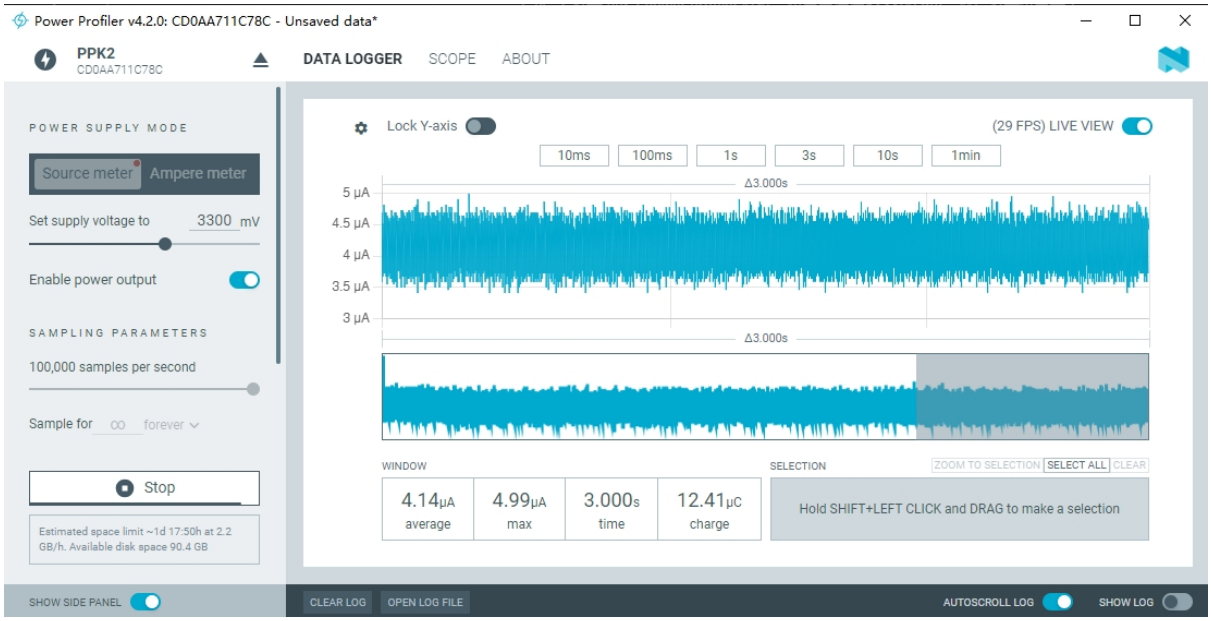


图 15: 系统初始化后进入 DeepSleep 模式

芯片低功耗状态下的底电流（漏电流）与环境温度相关，温度越高，漏电流越大。

4. 分别尝试按下 EVB 底板上的 3 个按键：KEY1、KEY2 和 WKUP，由串口打印信息可知 3 个按键事件均触发了芯片唤醒:

```
P0_6 INT occurred.
A notification received, value: 1.

Wait for Task Notifications..
P1_2 INT occurred.
A notification received, value: 1.

Wait for Task Notifications..
P0_2 INT occurred.
A notification received, value: 1.

Wait for Task Notifications..
```

5. 此时再观察芯片电流波形，可以看到芯片触发了 3 次唤醒，最后又进入 DeepSleep 状态等待下次按键唤醒：

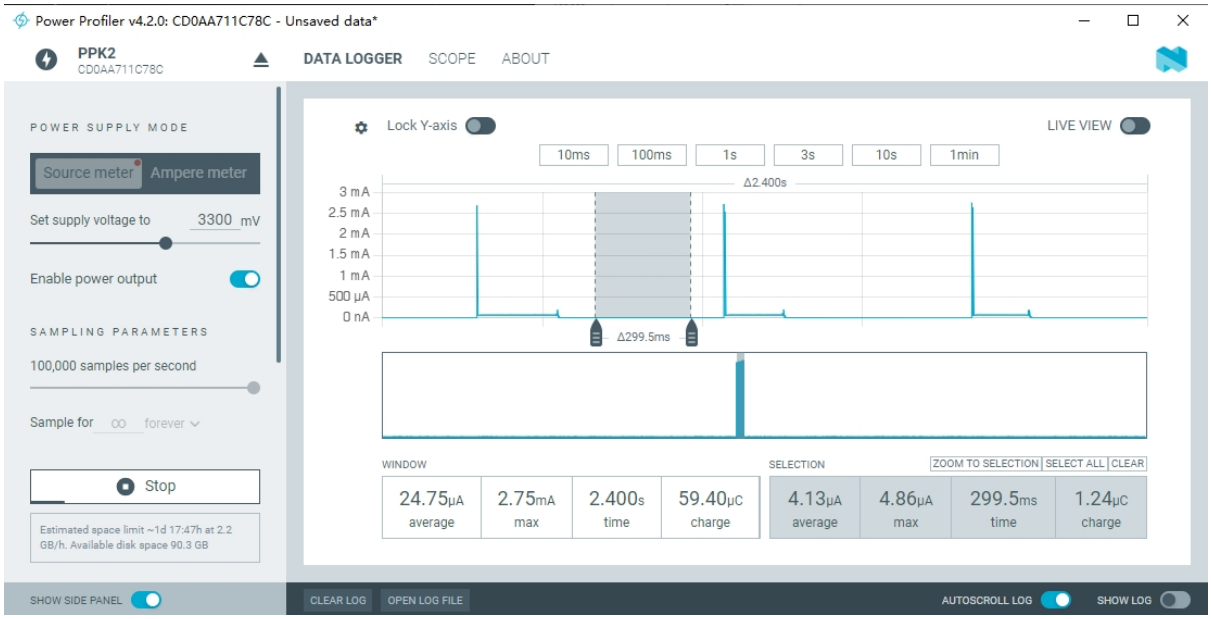


图 16: 分别使用 3 个按键唤醒芯片

由电流波形可知芯片每次唤醒后均重新进入了 DeepSleep 模式，此模式下芯片电流保持在 4uA 左右。

5 开发者说明

5.1 App Config 配置

本例程的 App Config（对应 app_config_spark.h 文件）配置如下：

其中，与本例程相关的配置有：

- Enable DCDC (`CONFIG_SOC_DCDC_PAN1070 = 1`)：使能芯片 DCDC 供电模式，以降低芯片动态功耗
- Log Enable (`CONFIG_LOG_ENABLE = 1`)：使能串口 Log 输出
- Low Power Enable (`CONFIG_PM = 1`)：使能系统低功耗流程

5.2 程序代码

5.2.1 主程序

主程序 app_main() 函数内容如下：

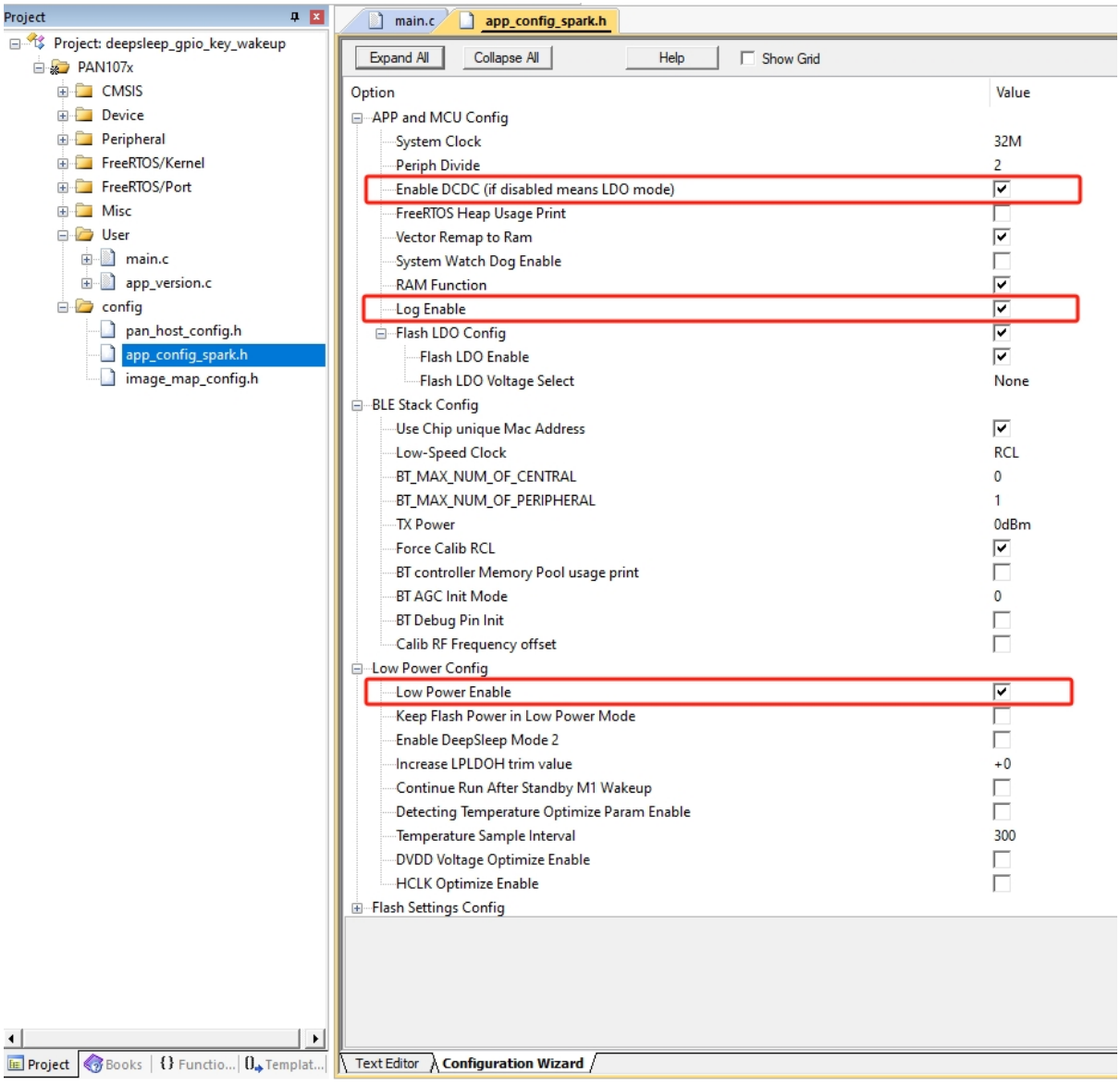


图 17: App Config File

```
void app_main(void)
{
    BaseType_t r;

    print_version_info();

    /* Create an App Task */
    r = xTaskCreate(app_task,                 // Task Function
                    "App Task",               // Task Name
                    APP_TASK_STACK_SIZE,      // Task Stack Size
                    NULL,                     // Task Parameter
                    APP_TASK_PRIORITY,        // Task Priority
                    NULL                      // Task Handle
    );

    /* Check if task has been successfully created */
    if (r != pdPASS) {
        printf("Error, App Task created failed!\n");
        while (1);
    }
}
```

1. 打印 App 版本信息
2. 创建 App 主任务 “App Task”，对应任务函数 app_task
3. 确认线程创建成功，否则打印出错信息

5.2.2 App 主任务

App 主任务 app_task() 函数内容如下：

```
void app_task(void *arg)
{
    uint32_t ulNotificationValue;

    /* Store the handle of current task. */
    xTaskToNotify = xTaskGetCurrentTaskHandle();
    if(xTaskToNotify == NULL) {
        printf("Error, get current task handle failed!\n");
        while (1);
    }

    wakeup_gpio_keys_init();

    while (1) {
        printf("Wait for Task Notifications..\n");

        /*
         * Wait to be notified that gpio key is pressed (gpio irq occured). Note the first␣
↪parameter
         * is pdTRUE, which has the effect of clearing the task's notification value back to 0,
↪ making
         * the notification value act like a binary (rather than a counting) semaphore.
         */
        ulNotificationValue = ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

        printf("A notification received, value: %d.\n\n", ulNotificationValue);
    }
}
```

1. 获取当前任务的 Task Handle，用于后续中断中给次任务发送通知使用
2. 在 wakeup_gpio_keys_init() 函数中初始化 GPIO 配置

3. 在 while (1) 主循环中尝试获取任务通知（Task Task Notify），并打印相关的状态信息

5.2.3 GPIO 初始化程序

GPIO 初始化程序 wakeup_gpio_keys_init() 函数内容如下：

```
static void wakeup_gpio_keys_init(void)
{
    /* Configure GPIO P06 (KEY1) / P12 (KEY2) / P02 (WKUP) as Falling Edge Interrupt/Wakeup */

    /* Set pinmux func as GPIO */
    SYS_SET_MFP(P0, 6, GPIO);
    SYS_SET_MFP(P1, 2, GPIO);
    SYS_SET_MFP(P0, 2, GPIO);

    /* Configure debounce clock */
    GPIO_SetDebounceTime(GPIO_DBCTL_DBCLKSRC_RCL, GPIO_DBCTL_DBCLKSEL_4);
    /* Enable input debounce function of specified GPIOs */
    GPIO_EnableDebounce(P0, BIT6);
    GPIO_EnableDebounce(P1, BIT2);
    GPIO_EnableDebounce(P0, BIT2);

    /* Set GPIOs to input mode */
    GPIO_SetMode(P0, BIT6, GPIO_MODE_INPUT);
    GPIO_SetMode(P1, BIT2, GPIO_MODE_INPUT);
    GPIO_SetMode(P0, BIT2, GPIO_MODE_INPUT);
    CLK_Wait3vSyncReady(); /* Necessary for P02 to do manual aon-reg sync */

    /* Enable internal pull-up resistor path */
    GPIO_EnablePullupPath(P0, BIT6);
    GPIO_EnablePullupPath(P1, BIT2);
    GPIO_EnablePullupPath(P0, BIT2);
    CLK_Wait3vSyncReady(); /* Necessary for P02 to do manual aon-reg sync */

    /* Wait for a while to ensure the internal pullup is stable before entering low power mode␣
↪*/
    SYS_delay_10nop(0x10000);

    /* Enable GPIO interrupts and set trigger type to Falling Edge */
    GPIO_EnableInt(P0, 6, GPIO_INT_FALLING);
    GPIO_EnableInt(P1, 2, GPIO_INT_FALLING);
    GPIO_EnableInt(P0, 2, GPIO_INT_FALLING);

    /* Enable GPIO IRQs in NVIC */
    NVIC_EnableIRQ(GPIO0_IRQn);
    NVIC_EnableIRQ(GPIO1_IRQn);
}
```

1. 此函数使用 Panchip Low-Level GPIO Driver 对 GPIO 进行配置

   实际上也可使用更上层的 Panchip HAL GPIO Driver 进行配置，具体可参考GPIO Digital Input Interrupt例程中的相关介绍

2. 由于 EVB 底板上有 3 个按键，分别对应核心板 P06/P12/P02 等 3 个引脚，因此这里仅配置这 3 个 GPIO 引脚：
   - 配置引脚 Pinmux 至 GPIO 功能
   - 使能去抖功能（并配置去抖时间）
   - 使能 GPIO 数字输入模式
   - 使能内部上拉电阻（按键没有外部上拉电阻）
   - 使能 GPIO 中断，将其配置为下降沿触发中断（即下降沿唤醒），并使能相关 NVIC IRQ

5.2.4 GPIO 中断服务程序

GPIO P0 和 P1 的中断服务程序分别如下：

```
void GPIO0_IRQHandler(void)
{
    BaseType_t xHigherPriorityTaskWoken = pdTRUE;

    for (size_t i = 0; i < 8; i++) {
        if (GPIO_GetIntFlag(P0, BIT(i))) {
            GPIO_ClrIntFlag(P0, BIT(i));
            printf("P0_%d INT occurred.\r\n", i);
            /* Notify the task that gpio key is pressed. */
            vTaskNotifyGiveFromISR(xTaskToNotify, &xHigherPriorityTaskWoken);
        }
    }
}

void GPIO1_IRQHandler(void)
{
    BaseType_t xHigherPriorityTaskWoken = pdTRUE;

    for (size_t i = 0; i < 8; i++) {
        if (GPIO_GetIntFlag(P1, BIT(i))) {
            GPIO_ClrIntFlag(P1, BIT(i));
            printf("P1_%d INT occurred.\r\n", i);
            /* Notify the task that gpio key is pressed. */
            vTaskNotifyGiveFromISR(xTaskToNotify, &xHigherPriorityTaskWoken);
        }
    }
}
```

1. 每个 GPIO Port 均需编写自己的中断服务函数，其内部可通过 `GPIO_GetIntFlag()` 接口判断触发中断的是当前 port 的哪根 pin
2. 在中断服务函数中需注意调用 `GPIO_ClrIntFlag()` 接口清除中断标志位
3. 使用 FreeRTOS `vTaskNotifyGiveFromISR()` 接口向 App Task 发送通知，表示有 GPIO 中断产生（对应按键按下），此接口中 `xHigherPriorityTaskWoken` 变量被配置为 `pdTRUE`，表示当中断返回后将会触发线程调度，而对于此例程来说则是重新调度至 App Task 中的 `ulTaskNotifyTake()` 处继续执行

5.2.5 与低功耗相关的 Hook 函数

本例程还用到了 2 个与低功耗密切相关的 Hook 函数：

```
CONFIG_RAM_CODE void vSocDeepSleepEnterHook(void)
{
#if CONFIG_LOG_ENABLE
    reset_uart_io();
#endif
}

CONFIG_RAM_CODE void vSocDeepSleepExitHook(void)
{
#if CONFIG_LOG_ENABLE
    set_uart_io();
#endif
}
```

1. FreeRTOS 有一个优先级最低的 Idle Task，当系统调度到此任务后会对当前状态进行检查，以判断是否允许进入芯片 DeepSleep 低功耗流程
2. 若程序执行到 Idle Task 的 DeepSleep 子流程中，会在 SoC 进入 DeepSleep 模式之前执行 `vSocDeepSleepEnterHook()` 函数，在 SoC 从 DeepSleep 模式下唤醒后执行 `vSocDeepSleepExitHook()` 函数

- 本例程在 `vSocDeepSleepEnterHook()` 函数中，为防止 UART IO 漏电，编写了相关代码（`reset_uart_io()`，具体实现见例程源码）以确保在进入 DeepSleep 模式前：
  - 串口 Log 数据都打印完毕（即 UART0 Tx FIFO 应为空）
  - P16 引脚 Pinmux 功能由 UART0 Tx 切换回 GPIO
  - P17 引脚 Pinmux 功能由 UART0 Rx 切换回 GPIO，并将其数字输入功能关闭
- 本例程在 `vSocDeepSleepExitHook()` 函数中，编写了相关代码（`set_uart_io()`，具体实现见例程源码）以恢复串口 Log 打印功能：
  - P16 引脚 Pinmux 功能由 GPIO 重新切换成 UART0 Tx
  - P17 引脚 Pinmux 功能由 GPIO 重新切换成 UART0 Rx，并将其数字输入功能重新打开

3.2.2 DeepSleep GPIO PWM Wakeup

1 功能概述

本例程演示如何使 SoC 进入 DeepSleep 状态，然后使用外部 PWM 波形通过 GPIO 将其唤醒。

2 环境准备

- 硬件设备与线材：
  - PAN107X EVB **核心板**与**底板**各**两**块
  - JLink 仿真器（用于烧录例程程序）
  - 逻辑分析仪（Logic Analyzer, LA，用于观察 PWM 波形信息和 GPIO 中断信息）
  - **电流计**（本文使用电流可视化测量设备 PPK2 [Nordic Power Profiler Kit II] 进行演示）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线（本例程）：
  - 将其中一块 EVB 核心板插到其中一块底板上
  - 为确保能够准确地测量 SoC 本身的功耗，排除底板外围电路的影响，请确认 EVB 底板上的：
    - Voltage 排针组中的 VCC 和 VDD 均接至 3V3
    - POWER 开关从 LDO 档位拨至 BAT 档位（并确认底板背部的电池座内**没有**纽扣电池）
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将 JLink 仿真器的：
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
  - 将逻辑分析仪硬件的：
    - USB 接口连接至 PC USB 接口
    - 三个通道的信号线分别连接至 EVB 底板的 P13 / P14 / P15 排针
    - GND 连接至 EVB 底板的 GND 排针
  - 将 PPK2 硬件的：
    - USB DATA/POWER 接口连接至 PC USB 接口

    * VOUT 连接至 EVB 底板的 VBAT 排针
    * GND 连接至 EVB 底板的 GND 排针

- 硬件接线（PWM Waveform Generator 例程）:
  - 参考DeepSleep PWM Waveform Generator 例程中的介绍搭建 PWM 波形输出环境
  - 将 PWM 波形输出例程的 P03 引脚与 GPIO 唤醒例程的 P13 引脚相连
  - 将 PWM 波形输出例程的 P04 引脚与 GPIO 唤醒例程的 P14 引脚相连
- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）
  - Logic（用于配合逻辑分析仪抓取 IO 波形）
  - nRF Connect Desktop（用于配合 PPK2 测量 SoC 电流）

3 编译和烧录

例程位置: `<PAN10XX-NDK>\01_SDK\nimble\samples\low_power\deepsleep_gpio_pwm_wakeup\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录，烧录成功后断开 JLink 连线以避免漏电。

4 例程演示说明

1. PC 上打开 PPK2 Power Profiler 软件，供电电压选择 3300 mV，然后打开供电开关
2. 从串口工具中看到如下的打印信息:

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 6
- Chip MAC Address  : E11000000FF8
- Chip UID          : 6D0001465454455354
- Chip Flash UID    : 4250315A3538380B005B7B4356037D78
- Chip Flash Size   : 512 KB
APP version: 255.255.65535
Wait for Task Notifications..
```

3. 将 PWM 波形输出板卡上电，使能 PWM 输出，此时观察 GPIO 唤醒例程的 Log，可以看到触发了多次的唤醒过程:

```
[Uptime: 90 ms] GPIO IRQ triggered: P14.
A notification received, value: 1.

Wait for Task Notifications..
[Uptime: 100 ms] GPIO IRQ triggered: P14.
A notification received, value: 1.

Wait for Task Notifications..
[Uptime: 109 ms] GPIO IRQ triggered: P14.
A notification received, value: 1.

Wait for Task Notifications..
[Uptime: 116 ms] GPIO IRQ triggered: P13.
A notification received, value: 1.

Wait for Task Notifications..
```

（下页继续）

(续上页)

```
[Uptime: 118 ms] GPIO IRQ triggered: P14.
A notification received, value: 1.
...
```

4. 此时观察芯片电流波形，可以更清晰地看出芯片多次睡眠唤醒的切换过程：

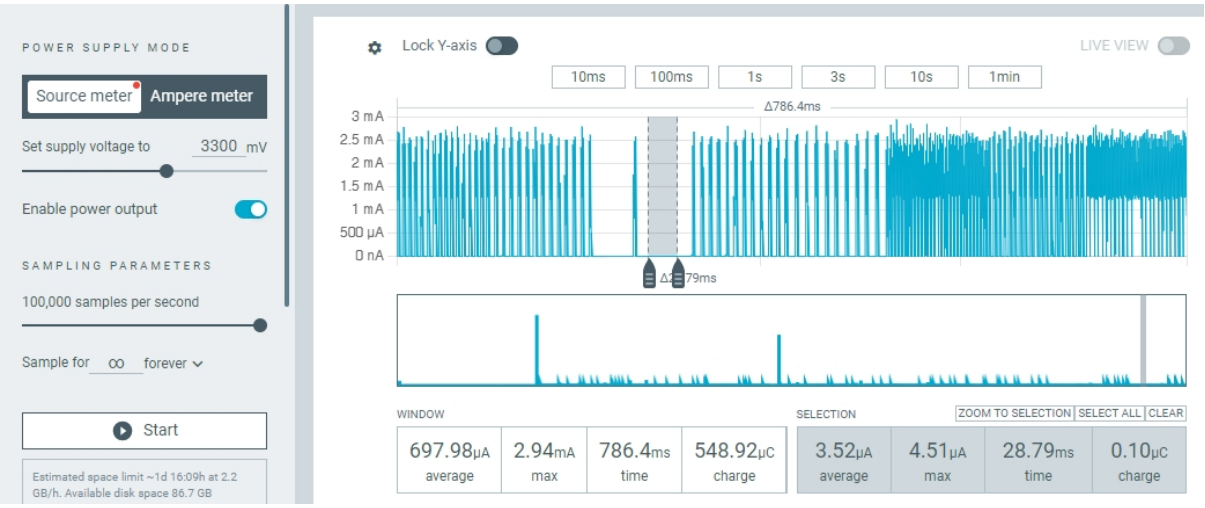


图 18: 系统被外部 PWM 波唤醒电流波形

5. 再观察逻辑分析仪波形，可以看出 GPIO P13/P14 分别被各自输入的 PWM 波形的上升沿唤醒并触发中断：

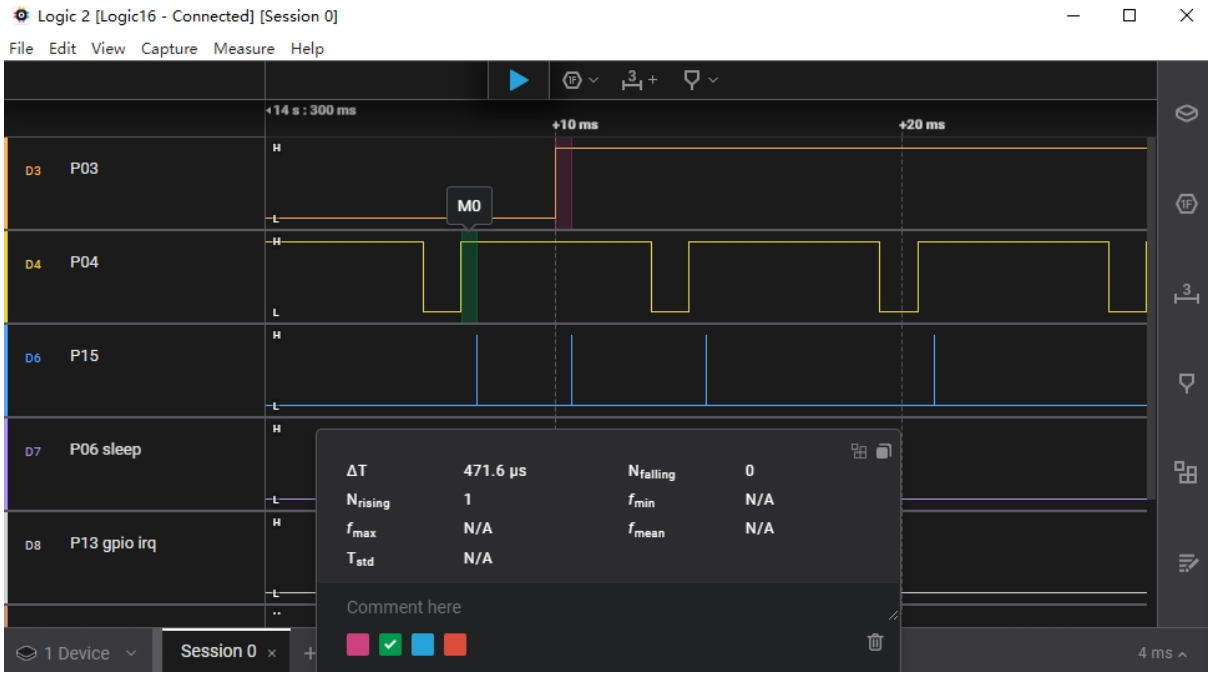


图 19: 系统被外部 PWM 波唤醒 LA 波形

程序中的两个 GPIO 均被配置为上升沿唤醒，从 LA 波形中可以看出，两个 IO 中的任意一个检测到 GPIO 上升沿，均在经过 470us 左右的时间后触发芯片唤醒并执行 GPIO 中断（由 P15 引脚指示）

5 开发者说明

5.1 App Config 配置 本例程的 App Config（对应 app_config_spark.h 文件）配置与DeepSleep GPIO Key Wakeup 例程完全相同：

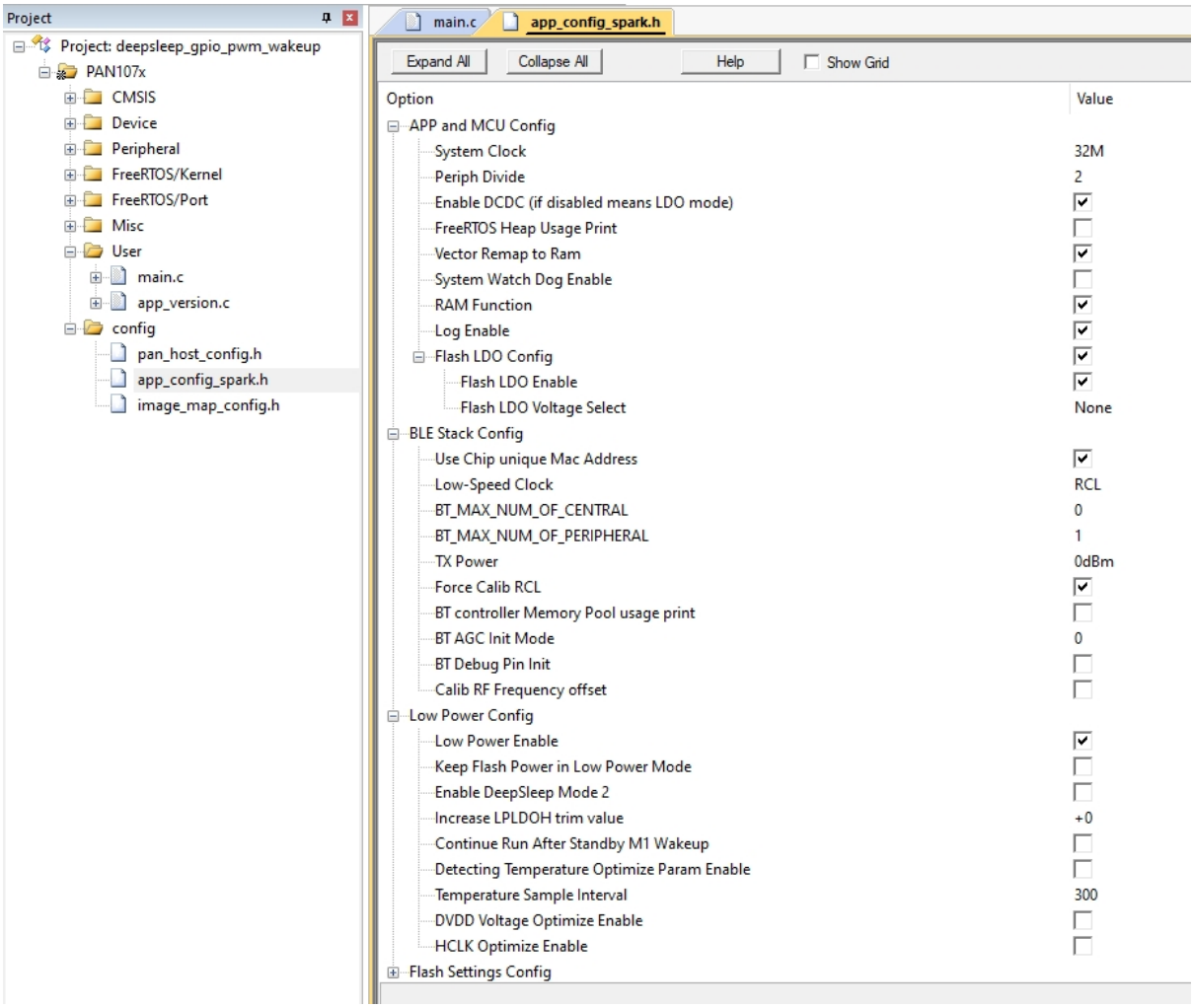


图 20: App Config File

5.2 程序代码

5.2.1 主程序 主程序 app_main() 函数内容如下：

```
void app_main(void)
{
    BaseType_t r;

    print_version_info();

    /* Create an App Task */
    r = xTaskCreate(app_task,               // Task Function
                    "App Task",             // Task Name
                    APP_TASK_STACK_SIZE,    // Task Stack Size
                    NULL,                   // Task Parameter
                    APP_TASK_PRIORITY,      // Task Priority
                    NULL                    // Task Handle
```

（下页继续）

(续上页)

```
    );

    /* Check if task has been successfully created */
    if (r != pdPASS) {
        printf("Error, App Task created failed!\n");
        while (1);
    }
}
```

1. 打印 App 版本信息
2. 创建 App 主任务 “App Task”，对应任务函数 app_task
3. 确认线程创建成功，否则打印出错信息

5.2.2 App 主任务

App 主任务 app_task() 函数内容如下：

```
void app_task(void *arg)
{
    uint32_t ulNotificationValue;

    /* Store the handle of current task. */
    xTaskToNotify = xTaskGetCurrentTaskHandle();
    if(xTaskToNotify == NULL) {
        printf("Error, get current task handle failed!\n");
        while (1);
    }

    /*
     * Init specific GPIOs:
     * - to input mode for wake up use
     * - to output mode for ISR indication use
     */
    gpio_init();

    while (1) {
        printf("Wait for Task Notifications..\n");

        /*
         * Wait to be notified that gpio gpio irq occured. Note the first parameter is pdTRUE,
         * which has the effect of clearing the task's notification value back to 0, making
         * the notification value act like a binary (rather than a counting) semaphore.
         */
        ulNotificationValue = ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

        printf("A notification received, value: %d.\n\n", ulNotificationValue);
    }
}
```

1. 获取当前任务的 Task Handle，用于后续中断中给次任务发送通知使用
2. 在 gpio_init() 函数中初始化 GPIO 配置
3. 在 while (1) 主循环中尝试获取任务通知（Task Task Notify），并打印相关的状态信息

5.2.3 GPIO 初始化程序

GPIO 初始化程序 gpio_init() 函数内容如下：

```
static void gpio_init(void)
{
    /* Configure GPIO P13/P14 as Rising Edge Interrupt/Wakeup */
```

(下页继续)

(续上页)

```
    /* Set pinmux func as GPIO */
    SYS_SET_MFP(P1, 3, GPIO);
    SYS_SET_MFP(P1, 4, GPIO);

    /* Set GPIOs to input mode */
    GPIO_SetMode(P1, BIT3 | BIT4, GPIO_MODE_INPUT);

    /* Enable GPIO interrupts and set trigger type to Falling Edge */
    GPIO_EnableInt(P1, 3, GPIO_INT_RISING);
    GPIO_EnableInt(P1, 4, GPIO_INT_RISING);

    /* Configure debounce clock to 32K clock so that GPIO edge could wake up SoC */
    GPIO_SetDebounceTime(GPIO_DBCTL_DBCLKSRC_RCL, GPIO_DBCTL_DBCLKSEL_4);

    /* Enable GPIO IRQs in NVIC */
    NVIC_EnableIRQ(GPIO1_IRQn);

    /* Configure GPIO P15 to push-pull output mode (with init low level) */
    P15 = 0;
    GPIO_SetMode(P1, BIT5, GPIO_MODE_OUTPUT);
}
```

1. 此函数使用 Panchip Low-Level GPIO Driver 对 GPIO 进行配置

   实际上也可使用更上层的 Panchip HAL GPIO Driver 进行配置，具体可参考GPIO Digital Input Interrupt例程中的相关介绍

2. 配置流程如下：

   - 配置 P13 和 P14 引脚 Pinmux 至 GPIO 功能（此步骤可省略，原因是这两个引脚默认就是 GPIO 功能）
   - 使能 P13 和 P14 的数字输入模式
   - 使能 P13 和 P14 的中断，将其配置为上升沿触发中断（即上升沿唤醒）
   - 配置 GPIO 去抖时钟为 32K Clock，以支持边沿唤醒（注意此处并不使能各个 IO 的去抖功能）
   - 使能 GPIO1 NVIC IRQ
   - 将 P15 配置为推挽输出功能

5.2.4 GPIO 中断服务程序

GPIO P1 的中断服务程序如下：

```
void GPIO1_IRQHandler(void)
{
    BaseType_t xHigherPriorityTaskWoken = pdTRUE;

    for (size_t i = 0; i < 8; i++) {
        if (GPIO_GetIntFlag(P1, BIT(i))) {
            GPIO_ClrIntFlag(P1, BIT(i));
            P15 = 1;
            P15 = 0;
            printf("[Uptime: %d ms] GPIO IRQ triggered: P1%d.\n", soc_lptmr_uptime_get_ms(),␣
↪i);
        }
    }

    /* Notify the task that gpio IRQ occured. */
    vTaskNotifyGiveFromISR(xTaskToNotify, &xHigherPriorityTaskWoken);
}
```

1. GPIO 中断服务函数内部可通过 `GPIO_GetIntFlag()` 接口判断触发中断的是哪根引脚

2. 在中断服务函数中需注意调用 `GPIO_ClrIntFlag()` 接口清除中断标志位
3. 使用 `soc_lptmr_uptime_get_ms()` 接口获取系统的上电时间戳并打印到串口
4. 使用 FreeRTOS `vTaskNotifyGiveFromISR()` 接口向 App Task 发送通知，表示有 GPIO 中断产生（对应按键按下），此接口中 `xHigherPriorityTaskWoken` 变量被配置为 `pdTRUE`，表示当中断返回后将会触发线程调度，而对于此例程来说则是重新调度至 App Task 中的 `ulTaskNotifyTake()` 处继续执行

5.2.5 **与低功耗相关的** Hook **函数**

本例程还用到了 2 个与低功耗密切相关的 Hook 函数：

```
CONFIG_RAM_CODE void vSocDeepSleepEnterHook(void)
{
#if CONFIG_LOG_ENABLE
    reset_uart_io();
#endif
}

CONFIG_RAM_CODE void vSocDeepSleepExitHook(void)
{
#if CONFIG_LOG_ENABLE
    set_uart_io();
#endif
}
```

1. 上述两个 Hook 函数用于在进入 DeepSleep 前和从 DeepSleep 唤醒后做一些额外操作，如关闭和重新配置 IO 为串口功能，以防止 DeepSleep 状态下 IO 漏电
2. 详细解释请参考DeepSleep GPIO Key Wakeup 例程中的相关介绍

3.2.3 DeepSleep SleepTimer Wakeup

1 功能概述

本例程演示如何使 SoC 进入 DeepSleep 状态，然后通过 SleepTimer 定时器将其唤醒。

2 环境准备

- 硬件设备与线材：
  - PAN107X EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - 逻辑分析仪（Logic Analyzer, LA，用于观察各个 Timer 定时器超时中断的触发时间）
  - **电流计**（本文使用电流可视化测量设备 PPK2 [Nordic Power Profiler Kit II] 进行演示）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 为确保能够准确地测量 SoC 本身的功耗，排除底板外围电路的影响，请确认 EVB 底板上的：
    - Voltage 排针组中的 VCC 和 VDD 均接至 3V3
    - POWER 开关从 LDO 档位拨至 BAT 档位（并确认底板背部的电池座内**没有**纽扣电池）
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17

  - 使用杜邦线将 JLink 仿真器的：
    * SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    * SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    * SWD_GND 引脚与 EVB 底板的 GND 排针相连
  - 将逻辑分析仪硬件的：
    * USB 接口连接至 PC USB 接口
    * 三个通道的信号线分别连接至 EVB 底板的 P13 / P14 / P15 排针
    * GND 连接至 EVB 底板的 GND 排针
  - 将 PPK2 硬件的：
    * USB DATA/POWER 接口连接至 PC USB 接口
    * VOUT 连接至 EVB 底板的 VBAT 排针
    * GND 连接至 EVB 底板的 GND 排针
- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）
  - Logic（用于配合逻辑分析仪抓取 IO 波形）
  - nRF Connect Desktop（用于配合 PPK2 测量 SoC 电流）

3 编译和烧录

例 程 位 置：`<PAN10XX-NDK>\01_SDK\nimble\samples\low_power\deepsleep_slptmr_wakeup\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录，烧录成功后断开 JLink 连线。

4 例程演示说明

1. PC 上打开 PPK2 Power Profiler 软件，供电电压选择 3300 mV，然后打开供电开关
2. 从串口工具中看到如下的打印信息：

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 6
- Chip MAC Address  : E11000000FF8
- Chip UID          : 6D0001465454455354
- Chip Flash UID    : 4250315A3538380B005B7B4356037D78
- Chip Flash Size   : 512 KB
APP version: 255.255.65535
[Uptime: 83 ms] Main Thread sleep..
[Uptime: 583 ms] SleepTimer 1 IRQ triggered.
[Uptime: 1083 ms] SleepTimer 1 IRQ triggered.
[Uptime: 1084 ms] SleepTimer 2 IRQ triggered.
[Uptime: 1583 ms] SleepTimer 1 IRQ triggered.
[Uptime: 2083 ms] SleepTimer 1 IRQ triggered.
[Uptime: 2084 ms] SleepTimer 2 IRQ triggered.
[Uptime: 2583 ms] SleepTimer 1 IRQ triggered.
[Uptime: 3083 ms] SleepTimer 1 IRQ triggered.
[Uptime: 3084 ms] SleepTimer 2 IRQ triggered.
[Uptime: 3583 ms] SleepTimer 1 IRQ triggered.
```

(下页继续)

(续上页)

```
[Uptime: 4083 ms] SleepTimer 1 IRQ triggered.
[Uptime: 4084 ms] SleepTimer 2 IRQ triggered.
[Uptime: 4583 ms] SleepTimer 1 IRQ triggered.
[Uptime: 5082 ms] Main Thread waked up.
[Uptime: 5083 ms] SleepTimer 2 IRQ triggered.
[Uptime: 5084 ms] SleepTimer 1 IRQ triggered.
[Uptime: 5085 ms] Main Thread sleep..
[Uptime: 5584 ms] SleepTimer 1 IRQ triggered.
[Uptime: 6083 ms] SleepTimer 1 IRQ triggered.
[Uptime: 6084 ms] SleepTimer 2 IRQ triggered.
[Uptime: 6584 ms] SleepTimer 1 IRQ triggered.
[Uptime: 7083 ms] SleepTimer 1 IRQ triggered.
[Uptime: 7084 ms] SleepTimer 2 IRQ triggered.
[Uptime: 7584 ms] SleepTimer 1 IRQ triggered.
[Uptime: 8083 ms] SleepTimer 1 IRQ triggered.
[Uptime: 8084 ms] SleepTimer 2 IRQ triggered.
[Uptime: 8584 ms] SleepTimer 1 IRQ triggered.
[Uptime: 9083 ms] SleepTimer 1 IRQ triggered.
[Uptime: 9084 ms] SleepTimer 2 IRQ triggered.
[Uptime: 9584 ms] SleepTimer 1 IRQ triggered.
[Uptime: 10082 ms] Main Thread waked up.
[Uptime: 10083 ms] SleepTimer 2 IRQ triggered.
[Uptime: 10084 ms] SleepTimer 1 IRQ triggered.
[Uptime: 10083 ms] Main Thread sleep..
[Uptime: 10584 ms] SleepTimer 1 IRQ triggered.
[Uptime: 11084 ms] SleepTimer 2 IRQ triggered.
[Uptime: 11084 ms] SleepTimer 1 IRQ triggered.
[Uptime: 11584 ms] SleepTimer 1 IRQ triggered.
...
```

由 Log 中的时间戳可以看出，每隔 500ms 触发 SleepTimer 1 中断，每隔 1s 触发 SleepTimer 2 中断，每隔 5s 触发 Main Task 调度

3. 观察逻辑分析仪波形，发现芯片 P13 引脚每隔 5s 左右拉低一次；P14 引脚每隔 500ms 左右拉低一次；P15 引脚每隔 1s 左右拉低一次：

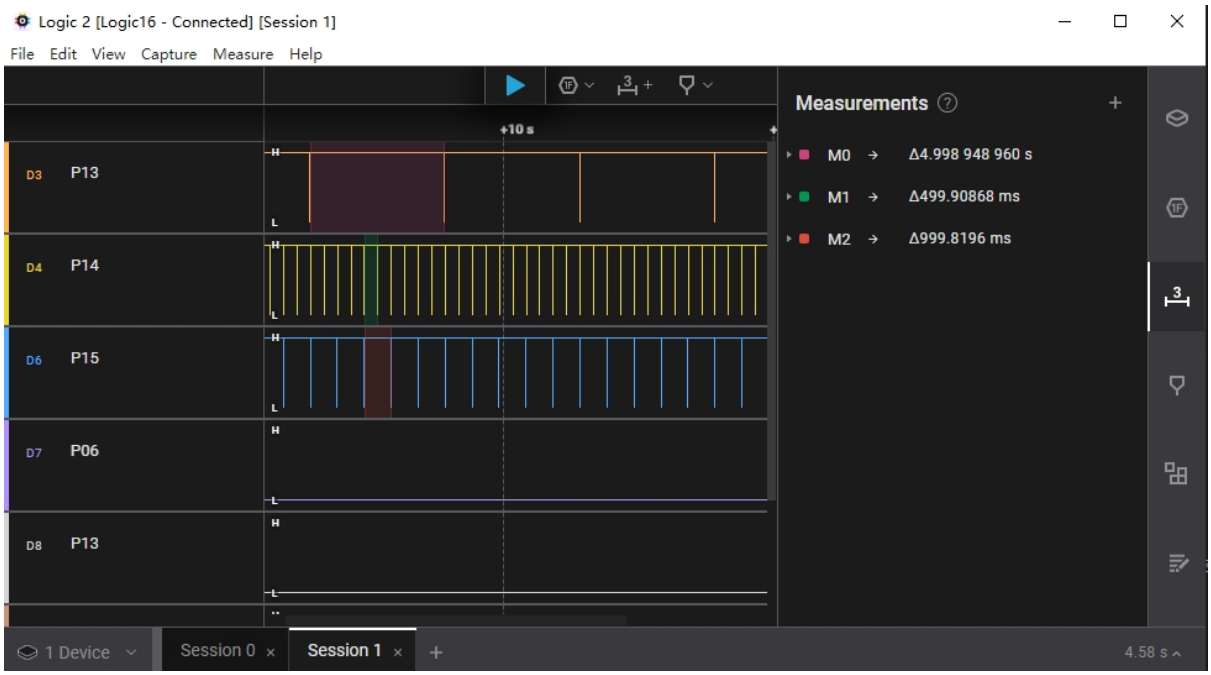


图 21: 使用逻辑分析仪抓取 GPIO 波形

P13 引脚指示 Main Task 执行时刻，P14 引脚只是 SleeTimer 1 中断触发时刻，P15 引脚只是 SleeTimer 2 中断触发时刻，可见它们与 Log 中的时间戳正好对应

4. 将逻辑分析仪从芯片上断开（避免影响电流观测），再观察芯片电流波形：

图 22: 使用电流计抓取电流波形

可见芯片低功耗底电流为 4uA 左右，并且每隔一段时间芯片被某个 SleepTimer 定时器从 DeepSleep 状态下唤醒。

5 开发者说明

5.1 App Config 配置 本例程的 App Config（对应 app_config_spark.h 文件）配置与DeepSleep GPIO Key Wakeup 例程完全相同：

5.2 程序代码

5.2.1 主程序 主程序 app_main() 函数内容如下：

```
void app_main(void)
{
    BaseType_t r;

    print_version_info();

    /* Create an App Task */
    r = xTaskCreate(app_task,                 // Task Function
                    "App Task",               // Task Name
                    APP_TASK_STACK_SIZE,      // Task Stack Size
                    NULL,                     // Task Parameter
                    APP_TASK_PRIORITY,        // Task Priority
                    NULL                      // Task Handle
    );

    /* Check if task has been successfully created */
    if (r != pdPASS) {
        printf("Error, App Task created failed!\n");
        while (1);
    }
}
```

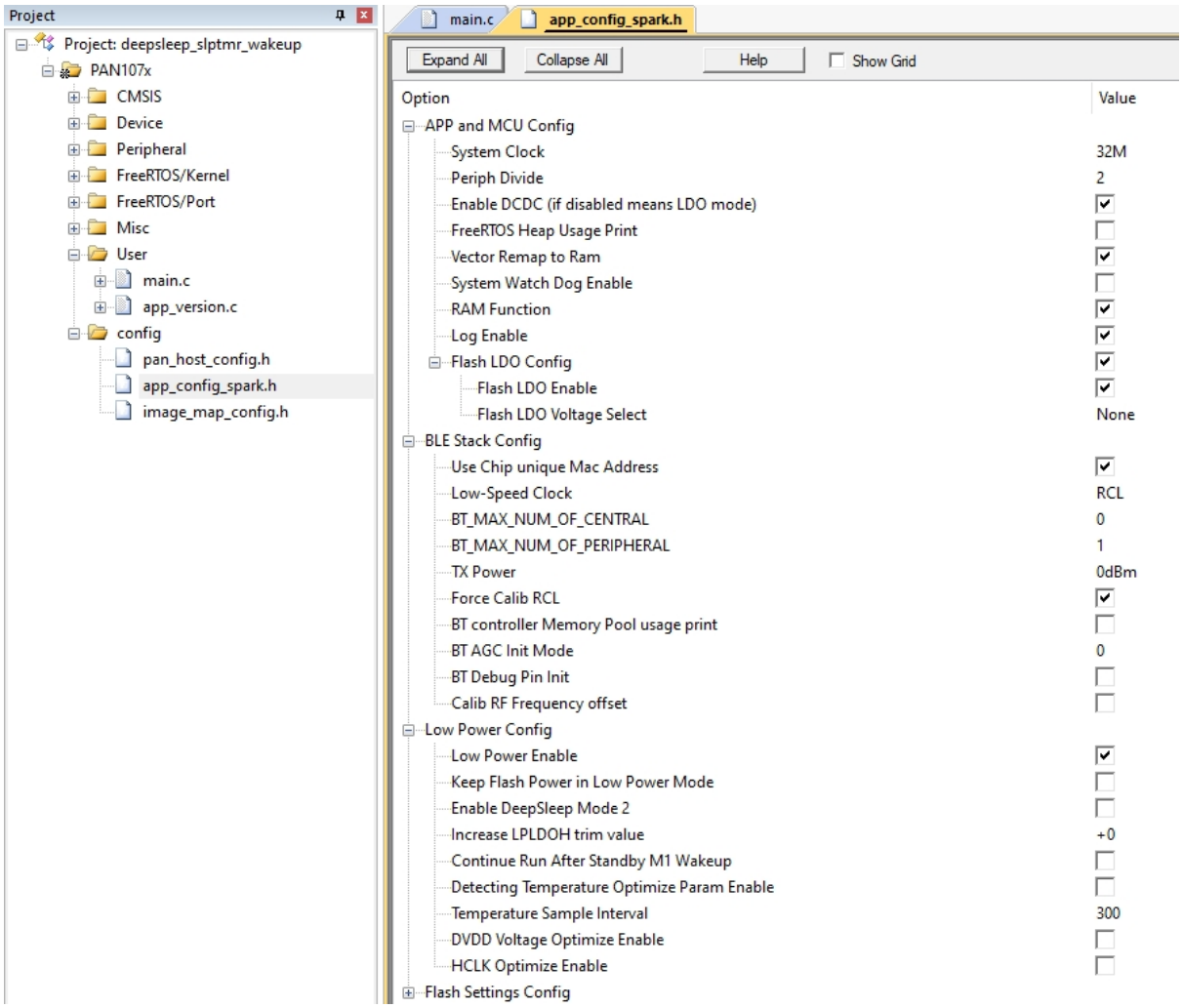


图 23: App Config File

1. 打印 App 版本信息
2. 创建 App 主任务 “App Task”，对应任务函数 `app_task`
3. 确认线程创建成功，否则打印出错信息

5.2.2 App 主任务

App 主任务 app_task() 函数内容如下：

```
void app_task(void *arg)
{
    /* Init specific GPIO as output for ISR indication use */
    indication_gpio_init();
    /* Init Sleep Timers for wake up use */
    slptmr_init();

    while (1) {
        printf("[Uptime: %d ms] Main Thread sleep..\n", soc_lptmr_uptime_get_ms());
        vTaskDelay(pdMS_TO_TICKS(5000));
        P13 = 0;
        P13 = 1;
        printf("[Uptime: %d ms] Main Thread waked up.\n", soc_lptmr_uptime_get_ms());
    }
}
```

1. 在 indication_gpio_init() 函数中初始化 GPIO，用于指示各个 SleepTimer 唤醒时刻
2. 在 slptmr_init() 函数中初始化 SleepTimer 1/2 的配置
3. 在 while (1) 主循环打印当前 Task 的睡眠唤醒 Log，并通过 `vTaskDelay()` 接口触发系统进出 DeepSleep 状态（注意通过此方式进入低功耗，实际上 OS 内部使用的是 SleepTimer 0 进行定时）

5.2.3 Indication GPIO 初始化程序

Indication GPIO 初始化程序 indication_gpio_init() 函数内容如下：

```
static void indication_gpio_init(void)
{
    /* Configure GPIO P13/P14/P15 to push-pull output mode */
    GPIO_SetMode(P1, BIT3 | BIT4 | BIT5, GPIO_MODE_OUTPUT);
}
```

1. 此函数使用 Panchip Low-Level GPIO Driver 对 GPIO 进行配置

   实际上也可使用更上层的 Panchip HAL GPIO Driver 进行配置，具体可参考GPIO Push-Pull Output例程中的相关介绍

2. 这里我们直接使用 `GPIO_SetMode()` 接口将 P13/P14/P15 配置为推挽输出模式（初始输出高电平）

5.2.4 SleepTimer 初始化程序

SleepTimer 1/2 初始化程序 slptmr_init() 函数内容如下：

```
static void slptmr_init(void)
{
    /* Configure timeout of SleepTimer 1&2 with interrupt and wakeup enabled */

    /*
     * Configure timeout:
     * timeout1 = SLPTMR1_TIMEOUT_CNT / PAN_RCC_CLOCK_FREQUENCY_SLOW
     *          = (32000 / 2) / 32000 (s)
     *          = 0.5 s = 500 ms
     * timeout2 = SLPTMR2_TIMEOUT_CNT / PAN_RCC_CLOCK_FREQUENCY_SLOW
     *          = 32000 / 32000 (s)
     *          = 1 s
```

(下页继续)

(续上页)

```
     */
    const uint32_t SLPTMR1_TIMEOUT_CNT = soc_32k_clock_freq_get() / 2;
    const uint32_t SLPTMR2_TIMEOUT_CNT = soc_32k_clock_freq_get();

    /* Set timeout of SleepTimer 1 */
    LP_SetSleepTime(ANA, SLPTMR1_TIMEOUT_CNT, 1);
    /* Set timeout of SleepTimer 2 */
    LP_SetSleepTime(ANA, SLPTMR2_TIMEOUT_CNT, 2);
}
```

1. 此函数功能是：
   - 将 SleepTimer 1 超时时间配置为 500ms
   - 将 SleepTimer 2 超时时间配置为 1s
2. 配置时间的具体公式计算请参考代码注释

5.2.5 SleepTimer 1/2 中断服务回调函数

SleepTimer 1/2 的中断服务回调函数分别如下：

```
/* This function overrides the reserved weak function with same name in os_lp.c */
void sleep_timer1_handler(void)
{
    P14 = 0;
    P14 = 1;
    printf("[Uptime: %d ms] SleepTimer 1 IRQ triggered.\n", soc_lptmr_uptime_get_ms());
}

/* This function overrides the reserved weak function with same name in os_lp.c */
void sleep_timer2_handler(void)
{
    P15 = 0;
    P15 = 1;
    printf("[Uptime: %d ms] SleepTimer 2 IRQ triggered.\n", soc_lptmr_uptime_get_ms());
}
```

1. 芯片共有 3 个 SleepTimer，分别为 SleepTimer 0、SleepTimer 1 和 SleepTimer 2，它们共用一个中断服务函数 `SLPTMR_IRQHandler()`，此函数是在 os_lp.c 中实现的；其中 SleepTimer0 被用作 OS Tick，因此在 App 层只可通过回调函数使用 SleepTimer 1 和 SleepTimer 2
2. 在 App 中实现 `sleep_timer1_handler()` 和 `sleep_timer1_handler()` 两个中断回调函数，在其中拉 Indication IO 并打印 Log，这样当对应 SleepTimer 超时后，即可唤醒系统并执行回调函数中的代码

5.2.6 与低功耗相关的 Hook 函数

本例程还用到了 2 个与低功耗密切相关的 Hook 函数：

```
CONFIG_RAM_CODE void vSocDeepSleepEnterHook(void)
{
#if CONFIG_LOG_ENABLE
    reset_uart_io();
#endif
}

CONFIG_RAM_CODE void vSocDeepSleepExitHook(void)
{
#if CONFIG_LOG_ENABLE
    set_uart_io();
#endif
}
```

1. 上述两个 Hook 函数用于在进入 DeepSleep 前和从 DeepSleep 唤醒后做一些额外操作，如关闭和重新配置 IO 为串口功能，以防止 DeepSleep 状态下 IO 漏电
2. 详细解释请参考DeepSleep GPIO Key Wakeup 例程中的相关介绍

3.2.4 DeepSleep PWM Waveform Generator

1 功能概述

本例程演示如何使 SoC 在 DeepSleep 状态下输出 PWM 波形，并使用 APB HW Timer0 定时唤醒并修改 PWM 波形周期和占空比。

2 环境准备

- 硬件设备与线材：
  - PAN107X EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - 逻辑分析仪（Logic Analyzer, LA，用于观察 PWM 波形信息）
  - **电流计**（本文使用电流可视化测量设备 PPK2 [Nordic Power Profiler Kit II] 进行演示）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 为确保能够准确地测量 SoC 本身的功耗，排除底板外围电路的影响，请确认 EVB 底板上的：
    - Voltage 排针组中的 VCC 和 VDD 均接至 3V3
    - POWER 开关从 LDO 档位拨至 BAT 档位（并确认底板背部的电池座内**没有**纽扣电池）
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将 JLink 仿真器的：
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
  - 将逻辑分析仪硬件的：
    - USB 接口连接至 PC USB 接口
    - 三个通道的信号线分别连接至 EVB 底板的 P03 / P04 / P15 排针
    - GND 连接至 EVB 底板的 GND 排针
  - 将 PPK2 硬件的：
    - USB DATA/POWER 接口连接至 PC USB 接口
    - VOUT 连接至 EVB 底板的 VBAT 排针
    - GND 连接至 EVB 底板的 GND 排针
- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）

- Logic（用于配合逻辑分析仪抓取 IO 波形）
- nRF Connect Desktop（用于配合 PPK2 测量 SoC 电流）

3 编译和烧录

例程位置：`<PAN10XX-NDK>\01_SDK\nimble\samples\low_power\deepsleep_pwm_waveform_generator\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录，烧录成功后断开 JLink 连线。

4 例程演示说明

1. PC 上打开 PPK2 Power Profiler 软件，供电电压选择 3300 mV，然后打开供电开关
2. 从串口工具中看到如下的打印信息：

```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 6
- Chip MAC Address   : E1100000101D
- Chip UID           : 0D0001465454455354
- Chip Flash UID     : 4250315A3538380B004B554356034578
- Chip Flash Size    : 512 KB
APP version: 130.99.13608
Wait for Task Notifications..
[Uptime: 183 ms] Timer0 ISR in, pwm_period = 0, pwm_pulse = 0.
[Uptime: 185 ms] Timer0 ISR in, pwm_period = 0, pwm_pulse = 0.
[Uptime: 284 ms] Timer0 ISR in, pwm_period = 0, pwm_pulse = 0.
[Uptime: 384 ms] Timer0 ISR in, pwm_period = 0, pwm_pulse = 0.
[Uptime: 484 ms] Timer0 ISR in, pwm_period = 0, pwm_pulse = 0.
[Uptime: 584 ms] Timer0 ISR in, pwm_period = 0, pwm_pulse = 0.
[Uptime: 684 ms] Timer0 ISR in, pwm_period = 0, pwm_pulse = 0.
[Uptime: 784 ms] Timer0 ISR in, pwm_period = 0, pwm_pulse = 0.
[Uptime: 884 ms] Timer0 ISR in, pwm_period = 0, pwm_pulse = 0.
[Uptime: 1083 ms] Timer0 ISR in, pwm_period = 172, pwm_pulse = 91.
[Uptime: 1183 ms] Timer0 ISR in, pwm_period = 158, pwm_pulse = 30.
[Uptime: 1283 ms] Timer0 ISR in, pwm_period = 43, pwm_pulse = 28.
[Uptime: 1383 ms] Timer0 ISR in, pwm_period = 276, pwm_pulse = 84.
[Uptime: 1483 ms] Timer0 ISR in, pwm_period = 14, pwm_pulse = 9.
...
```

3. 观察逻辑分析仪波形，发现芯片 P03 引脚输出周期为 100ms，占空比为 50% 的波形；P04 引脚输出周期为 0~10ms 可变，占空比可变的波形；P15 引脚以 100ms 左右的间隔翻转：

   可见 PWM 波形输出是连续不间断的，说明低功耗期间 PWM 仍可以正常工作，期间 P15 引脚翻转说明 Timer0 定时时间到了触发芯片唤醒，并改变 P04 引脚的 PWM 波形参数。

4. 将逻辑分析仪从芯片上断开（避免影响电流观测），再观察芯片电流波形：

   可见芯片低功耗底电流为 6uA 左右（比 GPIO 唤醒等其他模式的 4uA 底电流稍大一些，这是因为当前 DeepSleep 状态下内部 PWM 模块仍在工作），并且每隔 100ms 左右芯片从 DeepSleep 状态下唤醒。

5 开发者说明

5.1 App Config 配置

本例程的 App Config（对应 app_config_spark.h 文件）配置如下：

其中，与本例程相关的配置有：

图 24: 使用逻辑分析仪抓取 PWM 输出波形

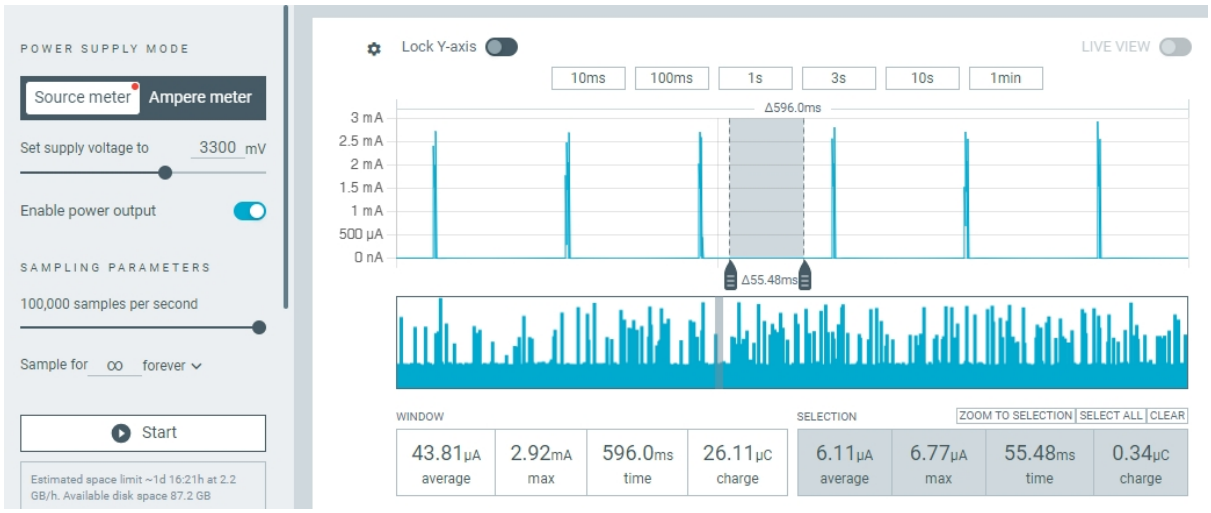


图 25: 使用电流计抓取电流波形

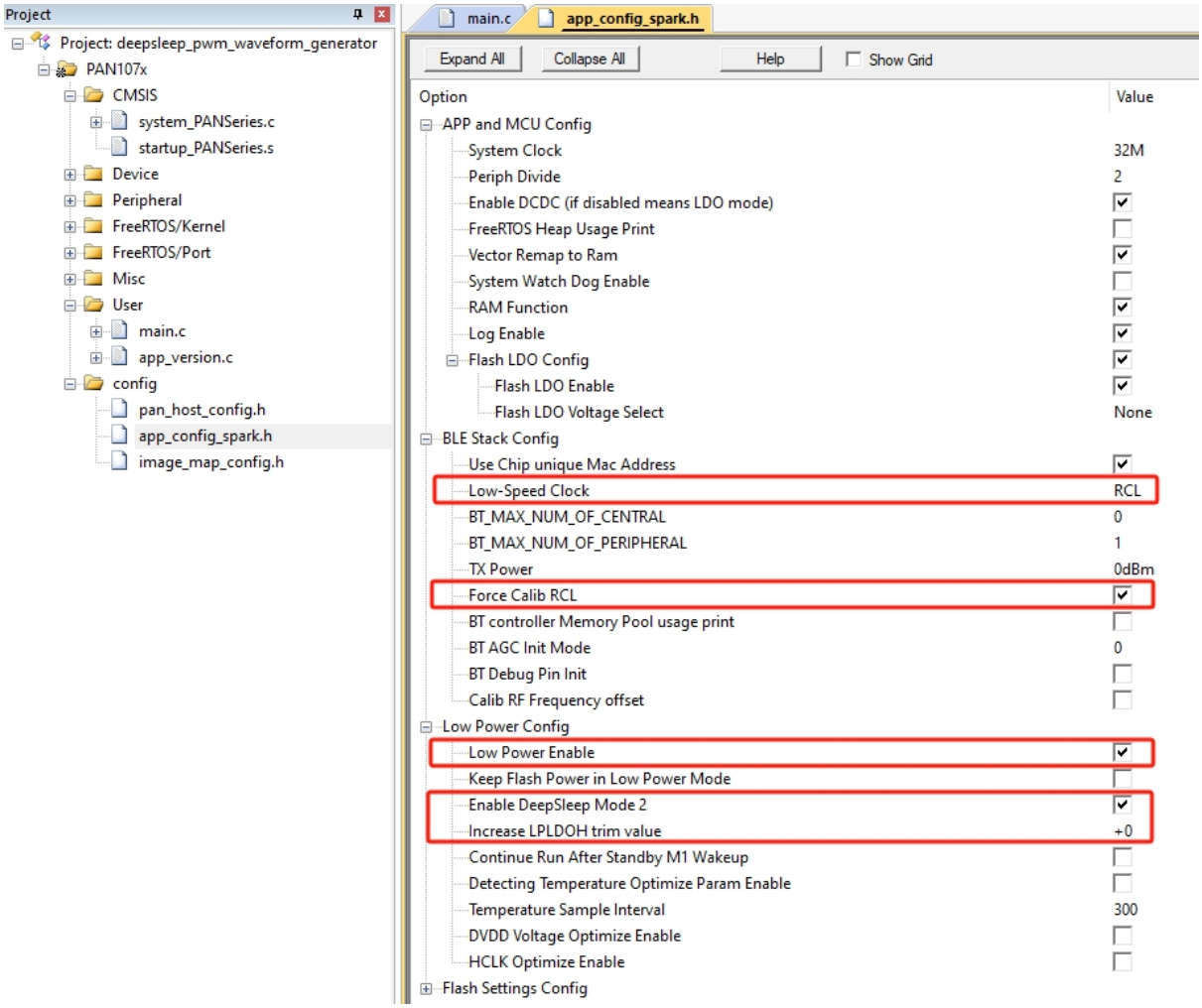


图 26: App Config File

- Low-Speed Clock (`CONFIG_LOW_SPEED_CLOCK_SRC = 0 (RCL)`)：系统低速时钟使用内部 32K 低速 RC 时钟（RCL）
- Force Calib RCL (`CONFIG_FORCE_CALIB_RCL_CLK = 1`)：在系统初始化阶段强制校准一次 RCL 时钟，此操作会增加 50ms 左右的启动时间
- Low Power Enable (`CONFIG_PM = 1`)：使能系统低功耗流程
- Enable DeepSleep Mode2 (`CONFIG_DEEPSLEEP_MODE_2 = 1`)：使能系统低功耗 DeepSleep 模式 2，此模式下使用电压较高的 LP-LDO-H（0.7v 以上）给芯片所有外设模块供电，以确保低功耗下 PWM 输出和 Timer 唤醒均正常
- Increase LPLDOH trim value (`CONFIG_SOC_INCREASE_LPLDOH_CALIB_CODE = 0`)：提高 LP-LDO-H 的电压，默认配置为 0，表示使用当前芯片默认的校准电压，但有时候此电压可能无法确保 PWM 输出和 Timer 唤醒正常，因此可以配置此选项以提高 LP-LDO-H 的电压值

5.2 程序代码

5.2.1 主程序

主程序 app_main() 函数内容如下：

```
void app_main(void)
{
    BaseType_t r;

    print_version_info();

    /* Create an App Task */
    r = xTaskCreate(app_task,                 // Task Function
                    "App Task",               // Task Name
                    APP_TASK_STACK_SIZE,      // Task Stack Size
                    NULL,                     // Task Parameter
                    APP_TASK_PRIORITY,        // Task Priority
                    NULL                      // Task Handle
    );

    /* Check if task has been successfully created */
    if (r != pdPASS) {
        printf("Error, App Task created failed!\n");
        while (1);
    }
}
```

1. 打印 App 版本信息
2. 创建 App 主任务 “App Task”，对应任务函数 `app_task`
3. 确认线程创建成功，否则打印出错信息

5.2.2 App 主任务

App 主任务 app_task() 函数内容如下：

```
void app_task(void *arg)
{
    uint32_t ulNotificationValue;

    /* Store the handle of current task. */
    xTaskToNotify = xTaskGetCurrentTaskHandle();
    if(xTaskToNotify == NULL) {
        printf("Error, get current task handle failed!\n");
        while (1);
    }
```

(下页继续)

(续上页)

```
    /* Initialize random seed for later pwm random duty use */
    srand(2024);

    /* Init GPIO P15 as indication IO */
    GPIO_SetMode(P1, BIT5, GPIO_MODE_OUTPUT);
    /* Init specific Timers for wake up use */
    wakeup_timer_init();
    /* Init specific PWM channels */
    pwm_init();

    while (1) {
        printf("Wait for Task Notifications..\n");

        /*
         * Here we try to take the task notify to let OS fall into idle task and
         * enter SoC DeepSleep mode. We'll never wakeup this task by giving notify.
         */
        ulNotificationValue = ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

        printf("A notification received, value: %d.\n\n", ulNotificationValue);
    }
}
```

1. 获取当前任务的 Task Handle，用于后续中断中给次任务发送通知使用
2. 初始化随机数种子，以供后续 Timer0 中断的修改 PWM 周期和占空比流程中使用
3. 将 P15 初始化为推挽输出模式
4. 在 wakeup_timer_init() 函数中初始化 HW APB Timer0
5. 在 pwm_init() 函数中初始化 PWM Channel 2 和 Channel 3
6. 在 while (1) 主循环中尝试获取任务通知（Task Task Notify），并打印相关的状态信息

5.2.3 Timer0 初始化程序

Timer0 初始化程序 wakeup_timer_init() 函数内容如下：

```
static void wakeup_timer_init(void)
{
    /* Configure HW APB Timer0 with interrupt and wakeup enabled */

    /*
     * Configure timeout
     * timeout = TIMER_CMP_VAL * (TIMER_PRESCALE + 1) / SYS_CLOCK_32K
     *         = 32000 / 10 * (0 + 1) / 32000 (s)
     *         = 0.1 s = 100 ms
     */
    const uint32_t TIMER_PRESCALE = 0;
    const uint32_t TIMER_CMP_VAL = soc_32k_clock_freq_get() / 10;

    /* Enable HW Timer0 Module clock */
    CLK_APB1PeriphClockCmd(CLK_APB1Periph_TMR0, ENABLE);
    /* Select Timer counting clock source to low-speed 32K clock */
    CLK_SetTmrClkSrc(TIMER0, CLK_APB1_TMR0SEL_RCL32K);
    /* Set Timer to periodic mode */
    TIMER_SetCountingMode(TIMER0, TIMER_PERIODIC_MODE);
    /* Enable Timer interrupt */
    TIMER_EnableInt(TIMER0);
    /* Enable Timer wakeup */
    TIMER_EnableWakeup(TIMER0);
    /* Enable Timer0 interrupt flag0 and wakeup flag0 signal */
    TIMER0->CTL |= BIT8 | BIT11;
```

(下页继续)

(续上页)

```
    /* Enable NVIC IRQ for Timer */
    NVIC_EnableIRQ(TMR0_IRQn);
    /* Set timeout value */
    TIMER_SetPrescaleValue(TIMER0, TIMER_PRESCALE);
    TIMER_SetCmpValue(TIMER0, TMR0_COMPARATOR_SEL_CMP, TIMER_CMP_VAL + TIMER0_CMPDAT_
↪DEVIATION);
    /* Start Timer */
    TIMER_Start(TIMER0);
}
```

1. 此函数目的是将 Timer0 模块配置为每隔 100ms 唤醒系统并产生中断
2. 由公式 `timeout = TIMER_CMP_VAL * (TIMER_PRESCALE + 1) / SYS_CLOCK_32K`，可知一个比较简单的配置方法是将 TIMER_CMP_VAL 配置为 3200，TIMER_PRESCALE 配置为 1
3. 配置并使能 Timer0 的流程包括：
   - 使能 APB1 上的 Timer0 时钟
   - 配置 Timer 计数时钟为 32K Clock
   - 将 Timer0 配置为周期模式
   - 使能 Timer0 的中断功能
   - 使能 Timer0 的唤醒功能
   - 使能 Timer0 的计数器 0 的中断和唤醒控制位
   - 使能 Timer0 的 NVIC IRQ
   - 配置 Timer0 的 Prescaler 预分频
   - 配置 Timer0 的 Compare Value
   - 启动 Timer0 计数

5.2.4 Timer0 中断服务程序

Timer0 的中断服务程序如下：

```
CONFIG_RAM_CODE void TMR0_IRQHandler(void)
{
    static uint32_t cnt;
    uint32_t pwm_pulse = 0;
    uint32_t pwm_period = 0;

    /* Handle timer interrupt event */
    if (TIMER_GetIntFlag(TIMER0)) {
        /* Clear timer int flags */
        TIMER_ClearTFFlag(TIMER0, TIMER_GetTFFlag(TIMER0));
        TIMER_ClearIntFlag(TIMER0);
        /* Randomize PWM CH3 duty cycle after 10 times this handler execute */
        if (++cnt > 9) {
            pwm_period = rand() % 319 + 1;
            pwm_pulse = rand() % pwm_period;
            /*
             * PWM Channel 3:
             * - Period = (<0~319> + 1) cycles = <1~320> * 1 / 32000 s = (0~10) ms
             * - Pulse = (<0~319> + 1) cycles = (0~10) ms
             */
            PWM_SetPeriodAndDuty(PWM, PWM_CH3, pwm_period, pwm_pulse);
        }
        /* Toggle GPIO to indicate timer is timeout */
        GPIO_Toggle(P1, BIT5);
        /* Print system uptime to UART */
```

(下页继续)

(续上页)

```
        printf("[Uptime: %d ms] Timer0 ISR in, pwm_period = %d, pwm_pulse = %d.\n",
            soc_lptmr_uptime_get_ms(), pwm_period, pwm_pulse);
    }

    /* Clear wakeup flag if there is. */
    TIMER_ClearWakeupFlag(TIMER0, TIMER_GetWakeupFlag(TIMER0));
}
```

1. 在函数名称前面加上 CONFIG_RAM_CODE 以将此函数编译成 RAM Code（需确保 app_config_spark.h 中的 CONFIG_RAM_FUNCTION 为使能状态）
2. 使用 TIMER_GetIntFlag() 接口检查 Timer0 中断 Flag 是否置位，若是则：
   - 清除 Timeout Flag
   - 清除中断 Flag
   - 满足一定条件后重新随机配置 PWM Channel 3 波形的周期和占空比
   - 翻转 GPIO P15，并向串口打印当前时间戳和 PWM 参数信息
3. 使用 TIMER_ClearWakeupFlag() 接口清除 Timer0 唤醒 Flag

5.2.5 PWM 初始化程序

PWM 初始化程序 pwm_init() 函数内容如下：

```
static void pwm_init(void)
{
    /* Configure HW PWM module which can output waveform even in SoC DeepSleep mode */

    /*
     * Configure PWM waveform period and duty:
     * PWM CH2 (PWM clock source is divided by PWM CLKDIV and CLKPSC):
     * - pwm_cnt_freq = PAN_RCC_CLOCK_FREQUENCY_SLOW / (PWM_CH23_CLKPSC + 1) / PWM_CH2_CLKDIV
     *                = 32000 / (3 + 1) / 2 (Hz)
     *                = 4000 Hz
     * - period = 1 / pwm_cnt_freq * (PWM_CH2_PERIOD_CNT + 1)
     *          = 1 / 4000 * (399 + 1) (s)
     *          = 100 ms
     * - pulse = 1 / pwm_cnt_freq * (PWM_CH2_PULSE_CNT + 1)
     *         = 1 / 4000 * (199 + 1) (s)
     *         = 50 ms
     * PWM CH3 (PWM clock source is configured as directly from 32K Clock):
     * - pwm_cnt_freq = PAN_RCC_CLOCK_FREQUENCY_SLOW
     *                = 32000 Hz
     * - period = 1 / pwm_cnt_freq * (PWM_CH3_PERIOD_CNT + 1)
     *          = 1 / 32000 * (15999 + 1) (s)
     *          = 500 ms
     * - pulse = 1 / pwm_cnt_freq * (PWM_CH3_PULSE_CNT + 1)
     *         = 1 / 32000 * (3199 + 1) (s)
     *         = 100 ms
     */
    const uint32_t PWM_CH23_CLKPSC = 3;

    const PWM_ClkDivDef PWM_CH2_CLKDIV = PWM_CLK_DIV_2;
    const uint32_t PWM_CH2_PERIOD_CNT = 399;
    const uint32_t PWM_CH2_PULSE_CNT = 199;

    const PWM_ClkDivDef PWM_CH3_CLKDIV = PWM_CLK_DIRECT;
    const uint32_t PWM_CH3_PERIOD_CNT = 15999;
    const uint32_t PWM_CH3_PULSE_CNT = 3199;

    /* Set PWM channel 2/3 counting clock source to 32K clock */
```

(下页继续)

(续上页)

```
    CLK_SetPwmClkSrc(PWM_CH2, CLK_APB1_PWM_CH23_SEL_CLK32K);

    /* Enable clock of PWM and related channel */
    CLK_APB1PeriphClockCmd(CLK_APB1Periph_PWM0_EN | CLK_APB1Periph_PWM0_CH23, ENABLE);

    /* Enable Pinmux of target PWM channel */
    SYS_SET_MFP(P0, 3, PWM_CH2);
    SYS_SET_MFP(P0, 4, PWM_CH3);

    /*
     * Set clock prescaler (clk_psc) of PWM Channel 2/3.
     * NOTE that the channel 2 and 3 share the same prescaler.
     */
    PWM_SetPrescaler(PWM, PWM_CH2, PWM_CH23_CLKPSC);

    /* Set clock divider (clk_div) of PWM channel 2/3 */
    PWM_SetDivider(PWM, PWM_CH2, PWM_CH2_CLKDIV);
    PWM_SetDivider(PWM, PWM_CH3, PWM_CH3_CLKDIV);

    /* Init PWM channel 2/3 with different period and pulse cycles */
    PWM_SetPeriodAndDuty(PWM, PWM_CH2, PWM_CH2_PERIOD_CNT, PWM_CH2_PULSE_CNT);
    PWM_SetPeriodAndDuty(PWM, PWM_CH3, PWM_CH3_PERIOD_CNT, PWM_CH3_PULSE_CNT);

    /* Invert output level of PWM channel 3 */
    PWM_EnableOutputInverter(PWM, BIT(PWM_CH3));

    /* Enable Output of PWM channel 2/3 */
    PWM_EnableOutput(PWM, BIT(PWM_CH2));
    PWM_EnableOutput(PWM, BIT(PWM_CH3));

    /* Start PWM internal counter */
    PWM_StartChannel(PWM, PWM_CH2);
    PWM_StartChannel(PWM, PWM_CH3);
}
```

1. 此函数功能是：
   - 将 PWM Channel 2 输出周期 100ms，占空比 50% 的波形
   - 将 PWM Channel 3 初始输出周期 500ms，占空比 20% 的波形（后续会在 Timer0 中断中修改）
2. PWM 配置也有 Prescaler 和 Clock Divisor 的概念，它们共同决定了 1 个 PWM Count 的时间，具体公式计算请参考代码注释
3. 配置并使能 PWM 的流程包括：
   - 配置 PWM 计数时钟为 32K Clock
   - 使能 APB1 上的 PWM 时钟和 PWM Channel 2 / Channel 3 时钟
   - 配置 Pinmux 引脚，将 P03 配置为 PWM Channel 2 功能，P04 配置为 PWM Channel 3 功能
   - 配置 PWM Channel 2 和 Channel 3 的 Prescaler 预分频参数（两个通道共用一个参数）
   - 配置 PWM Channel 2 和 Channel 3 的 Clock Dividor 分频系数（各个通道可单独配置）
   - 配置 PWM Channel 2 和 Channel 3 的周期和占空比
   - 反相 PWM Channel 3 输出波形
   - 使能 PWM Channel 2 和 Channel 3 的输出功能
   - 启动 PWM Channel 2 和 Channel 3

5.2.6 **与低功耗相关的** Hook **函数**

本例程还用到了 2 个与低功耗密切相关的 Hook 函数：

```
CONFIG_RAM_CODE void vSocDeepSleepEnterHook(void)
{
#if CONFIG_LOG_ENABLE
    reset_uart_io();
#endif
}

CONFIG_RAM_CODE void vSocDeepSleepExitHook(void)
{
#if CONFIG_LOG_ENABLE
    set_uart_io();
#endif
}
```

1. 上述两个 Hook 函数用于在进入 DeepSleep 前和从 DeepSleep 唤醒后做一些额外操作，如关闭和重新配置 IO 为串口功能，以防止 DeepSleep 状态下 IO 漏电
2. 详细解释请参考DeepSleep GPIO Key Wakeup 例程中的相关介绍

3.2.5 Standby Mode1 GPIO Key Wakeup

1 功能概述

本例程演示如何使 SoC 进入 Standby Mode 1 状态，然后通过 GPIO 按键将其唤醒。

2 环境准备

- 硬件设备与线材：
  - PAN107X EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - **电流计**（本文使用电流可视化测量设备 PPK2 [Nordic Power Profiler Kit II] 进行演示）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 为确保能够准确地测量 SoC 本身的功耗，排除底板外围电路的影响，请确认 EVB 底板上的：
    - Voltage 排针组中的 VCC 和 VDD 均接至 3V3
    - POWER 开关从 LDO 档位拨至 BAT 档位（并确认底板背部的电池座内**没有**纽扣电池）
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将 JLink 仿真器的：
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
  - 将 PPK2 硬件的：
    - USB DATA/POWER 接口连接至 PC USB 接口
    - VOUT 连接至 EVB 底板的 VBAT 排针

    * GND 连接至 EVB 底板的 GND 排针

- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）
  - nRF Connect Desktop（用于配合 PPK2 测量 SoC 电流）

3 编译和烧录

例 程 位 置: `<PAN10XX-NDK>\01_SDK\nimble\samples\low_power\standby_m1_gpio_key_wakeup\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录，烧录成功后断开 JLink 连线以避免漏电。

4 例程演示说明

1. PC 上打开 PPK2 Power Profiler 软件，供电电压选择 3300 mV，然后打开供电开关
2. 从串口工具中看到如下的打印信息:

```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 6
- Chip MAC Address   : E11000000FF8
- Chip UID           : 6D0001465454455354
- Chip Flash UID     : 4250315A3538380B005B7B4356037D78
- Chip Flash Size    : 512 KB
APP version: 255.255.65535

Reset Reason: nRESET Pin Reset.

Try to enter SoC sleep/deepsleep mode and wait for 3-times key pressing..
```

3. 此时观察芯片电流波形，发现稳定在 4uA 左右（说明芯片成功进入了 DeepSleep 模式）:

图 27: 系统初始化后进入 DeepSleep 模式

芯片低功耗状态下的底电流（漏电流）与环境温度相关，温度越高，漏电流越大。

4. 尝试按下 EVB 底板上的 3 个按键中的任意一个或几个：KEY1、KEY2 和 WKUP，由串口打印信息可知触发 3 次 DeepSleep 唤醒后，芯片会延时 500ms，然后进入 Standby Mode 1 状态:

```
P0_6 INT occurred.
First key pressed, notify value = 1.
P1_2 INT occurred.
Second key pressed, notify value = 1.
P0_2 INT occurred.
Third key pressed, notify value = 1.

Busy wait 500ms to keep SoC in active mode..

Now try to enter SoC standby mode 1 (wakeup reset)..
```

5. 此时再观察芯片电流波形，可以看到芯片触发了 3 次 DeepSleep 唤醒，最后一次唤醒持续了约 500ms 时间，随后进入 Standby Mode 1 状态等待下次按键唤醒：

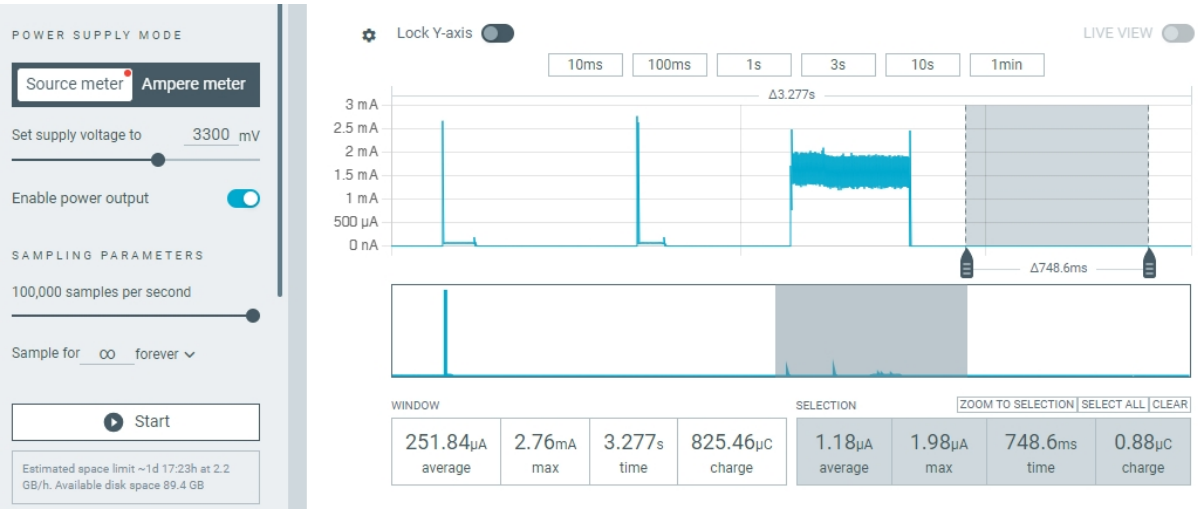


图 28: 使用按键将芯片从 DeepSleep 状态下唤醒 3 次后，芯片进入 Stnadby Mode 1 状态

由电流波形可知芯片前两次唤醒后均重新进入了 DeepSleep 模式，此模式下芯片电流保持在 4uA 左右；第三次唤醒后，芯片最终进入了 Standby Mode 1 模式，此模式下芯片电流保持在 1.2uA 左右。

6. 再次尝试按下 EVB 底板上的 3 个按键中的任意一个（如 KEY1），由串口打印信息可知芯片从 Standby Mode 1 状态下唤醒并复位，复位的原因为 GPIO P06 唤醒（对应按键 KEY1），随后芯片重新进入 DeepSleep 状态继续等待按键唤醒：

```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 6
- Chip MAC Address   : E11000000FF8
- Chip UID           : 6D0001465454455354
- Chip Flash UID     : 4250315A3538380B005B7B4356037D78
- Chip Flash Size    : 512 KB
APP version: 255.255.65535

Reset Reason: Standby Mode 1 GPIO Wakeup (cnt = 1).
gpio wakeup src flag = 0x00000040
SoC is waked up by GPIO P0_6.
P0_6 INT occurred.

Try to enter SoC sleep/deepsleep mode and wait for 3-times key pressing..
First key pressed, notify value = 1.
```

5 开发者说明

5.1 App Config 配置

本例程的 App Config（对应 app_config_spark.h 文件）配置如下：

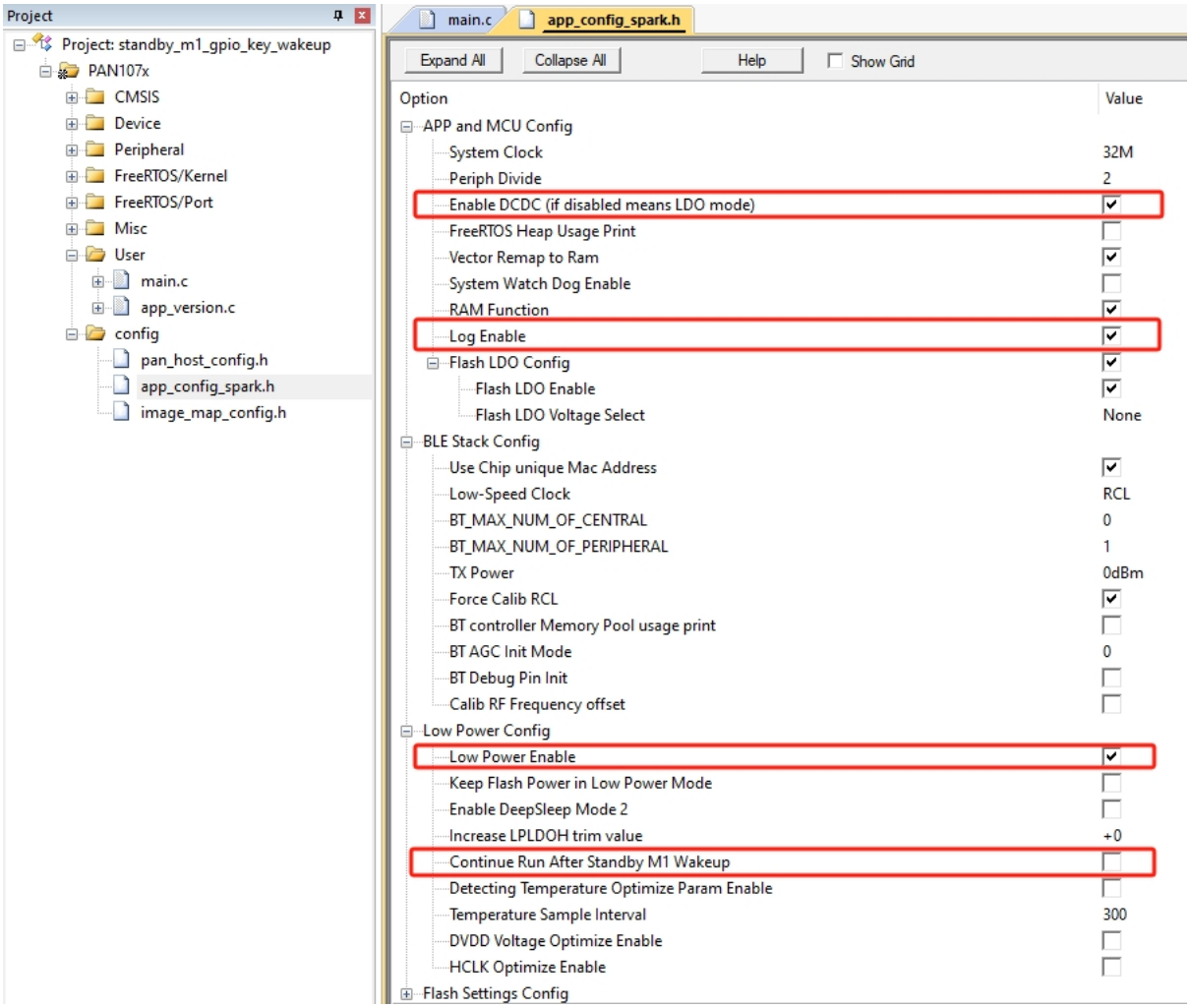


图 29: App Config File

其中，与本例程相关的配置有：

- Enable DCDC (`CONFIG_SOC_DCDC_PAN1070 = 1`)：使能芯片 DCDC 供电模式，以降低芯片动态功耗
- Log Enable (`CONFIG_LOG_ENABLE = 1`)：使能串口 Log 输出
- Low Power Enable (`CONFIG_PM = 1`)：使能系统低功耗流程
- Continue Run After Standby M1 Wakeup (`CONFIG_PM_STANDBY_M1_WAKEUP_WITHOUT_RESET = 0`)：配置 Standby Mode 1 唤醒后芯片行为是复位（而不是继续执行 WFI 后面的指令）

5.2 程序代码

5.2.1 主程序

主程序 app_main() 函数内容如下：

```
void app_main(void)
{
    BaseType_t r;
```

（下页继续）

(续上页)

```
    print_version_info();

    /* Create an App Task */
    r = xTaskCreate(app_task,                // Task Function
                    "App Task",              // Task Name
                    APP_TASK_STACK_SIZE,     // Task Stack Size
                    NULL,                    // Task Parameter
                    APP_TASK_PRIORITY,       // Task Priority
                    NULL                     // Task Handle
    );

    /* Check if task has been successfully created */
    if (r != pdPASS) {
        printf("Error, App Task created failed!\n");
        while (1);
    }
}
```

1. 打印 App 版本信息
2. 创建 App 主任务 “App Task”，对应任务函数 app_task
3. 确认线程创建成功，否则打印出错信息

5.2.2 App 主任务

App 主任务 app_task() 函数内容如下：

```
void app_task(void *arg)
{
    uint32_t ulNotificationValue;
    uint8_t rst_reason;

    /* Store the handle of current task. */
    xTaskToNotify = xTaskGetCurrentTaskHandle();
    if(xTaskToNotify == NULL) {
        printf("Error, get current task handle failed!\n");
        while (1);
    }

    /* Get the last reset reason */
    printf("\nReset Reason: ");
    rst_reason = soc_reset_reason_get();
    switch (rst_reason) {
    case SOC_RST_REASON_POR_RESET:
        printf("Power On Reset.\n");
        break;
    case SOC_RST_REASON_PIN_RESET:
        printf("nRESET Pin Reset.\n");
        break;
    case SOC_RST_REASON_SYS_RESET:
        printf("NVIC System Reset.\n");
        break;
    case SOC_RST_REASON_STBM1_GPIO_WAKEUP:
        printf("Standby Mode 1 GPIO Wakeup (cnt = %d).\n", ++wkup_cnt);
        parse_stbm1_gpio_wakeup_source();
        break;
    default:
        printf("Unhandled Reset Reason, refer to more reason define in os_lp.h!\n");
    }

    /* Enable 3 GPIO keys low-level wakeup for standby mode 1 */
    wakeup_gpio_keys_init();
```

(下页继续)

(续上页)

```
    printf("\nTry to enter SoC sleep/deepsleep mode and wait for 3-times key pressing..\n");
    /*
     * Wait to be notified that gpio key is pressed (gpio irq occured). Note the first␣
↪parameter
     * is pdFALSE, which has the effect of decrease the task's notification value by 1, making
     * the notification value act like a counting semaphore.
     */
    ulNotificationValue = ulTaskNotifyTake(pdFALSE, portMAX_DELAY);
    printf("First key pressed, notify value = %d.\n", ulNotificationValue);
    ulNotificationValue = ulTaskNotifyTake(pdFALSE, portMAX_DELAY);
    printf("Second key pressed, notify value = %d.\n", ulNotificationValue);
    ulNotificationValue = ulTaskNotifyTake(pdFALSE, portMAX_DELAY);
    printf("Third key pressed, notify value = %d.\n", ulNotificationValue);

    printf("\nBusy wait 500ms to keep SoC in active mode..\n");
    soc_busy_wait(500000);

#if !CONFIG_PM_STANDBY_M1_WAKEUP_WITHOUT_RESET
    printf("\nNow try to enter SoC standby mode 1 (wakeup reset)..\n\n");
#if CONFIG_LOG_ENABLE
    /* Waiting for UART Tx done and re-set UART IO before entering standby mode 1 to avoid␣
↪current leakage */
    reset_uart_io();
#endif
    /* Enter standby mode1 with all sram power off */
    soc_enter_standby_mode_1(STBM1_WAKEUP_SRC_GPIO, STBM1_RETENTION_SRAM_NONE);

    printf("WARNING: Failed to enter SoC standby mode 1 due to unexpected interrupt detected.\n
↪");
    printf("         Please check if there is an unhandled interrupt during the standby mode 1\
↪n");
    printf("         entering flow.\n");

    while (1) {
        /* Busy wait */
    }
#else
    while (1) {
        printf("\nNow try to enter SoC standby mode 1 (wakeup continuous run)..\n\n");
#if CONFIG_LOG_ENABLE
        /* Waiting for UART Tx done and re-set UART IO before entering standby mode 1 to avoid␣
↪current leakage */
        reset_uart_io();
#endif
        /* Enter standby mode1 with all common 48KB sram retention */
        soc_enter_standby_mode_1(STBM1_WAKEUP_SRC_GPIO, STBM1_RETENTION_SRAM_BLOCK0 | STBM1_
↪RETENTION_SRAM_BLOCK1);
        /* Restore hardware peripherals manually */
        restore_hw_peripherals();
        printf("Waked up from SoC standby mode 1 and continue run (cnt = %d)..\n", ++wkup_cnt);
    }
#endif /* CONFIG_PM_STANDBY_M1_WAKEUP_WITHOUT_RESET */
}
```

1. 获取当前任务的 Task Handle，用于后续中断中给次任务发送通知使用
2. 获取并打印本次芯片启动的复位原因，若复位原因为 Standby Mode 1 GPIO 唤醒，则额外解析并打印唤醒 IO 是哪个引脚
3. 在 wakeup_gpio_keys_init() 函数中初始化按键 GPIO 配置
4. 尝试获取任务通知（Task Task Notify）3 次，使系统 3 次进入 DeepSleep 状态并等待 GPIO 按键

唤醒

5. 第三次从 DeepSleep 状态下唤醒后，调用 `soc_busy_wait()` 接口使芯片在 Active 状态下全速运行 500ms
6. 根据 `CONFIG_PM_STANDBY_M1_WAKEUP_WITHOUT_RESET` 配置，决定进入 Standby Mode 1 的流程，这里由于我们将此配置设置为 0，因此会执行**唤醒后复位**的 Standby Mode 1 流程
7. 进入 Standby Mode 1 前，先将 Log UART IO 配置还原为 GPIO 模拟输入状态，以避免 Standby 模式下 IO 漏电
8. 调用 `soc_enter_standby_mode_1()` 接口，使系统进入 Standby Mode 1 低功耗状态，此接口接受两个参数：第一个参数用于配置唤醒源，第二个参数用于配置低功耗下保电的 SRAM，这里我们将唤醒源配置为 GPIO，且低功耗状态下所有 SRAM 均掉电以节约功耗（由于当前低功耗配置为唤醒后复位，因此所有 SRAM 均没有保电的必要）

5.2.3 GPIO **初始化程序**

GPIO 初始化程序 wakeup_gpio_keys_init() 函数内容如下：

```
static void wakeup_gpio_keys_init(void)
{
    /* Configure GPIO P06 (KEY1) / P12 (KEY2) / P02 (WKUP) as Falling Edge Interrupt/Wakeup */

    /* Set pinmux func as GPIO */
    SYS_SET_MFP(P0, 6, GPIO);
    SYS_SET_MFP(P1, 2, GPIO);
    SYS_SET_MFP(P0, 2, GPIO);

    /* Configure debounce clock */
    GPIO_SetDebounceTime(GPIO_DBCTL_DBCLKSRC_RCL, GPIO_DBCTL_DBCLKSEL_4);
    /* Enable input debounce function of specified GPIOs */
    GPIO_EnableDebounce(P0, BIT6);
    GPIO_EnableDebounce(P1, BIT2);
    GPIO_EnableDebounce(P0, BIT2);

    /* Set GPIOs to input mode */
    GPIO_SetMode(P0, BIT6, GPIO_MODE_INPUT);
    GPIO_SetMode(P1, BIT2, GPIO_MODE_INPUT);
    GPIO_SetMode(P0, BIT2, GPIO_MODE_INPUT);
    CLK_Wait3vSyncReady(); /* Necessary for P02 to do manual aon-reg sync */

    /* Enable internal pull-up resistor path */
    GPIO_EnablePullupPath(P0, BIT6);
    GPIO_EnablePullupPath(P1, BIT2);
    GPIO_EnablePullupPath(P0, BIT2);
    CLK_Wait3vSyncReady(); /* Necessary for P02 to do manual aon-reg sync */

    /* Wait for a while to ensure the internal pullup is stable before entering low power mode␣
↪*/
    soc_busy_wait(10000);

    /* Enable GPIO interrupts and set trigger type to Falling Edge */
    GPIO_EnableInt(P0, 6, GPIO_INT_FALLING);
    GPIO_EnableInt(P1, 2, GPIO_INT_FALLING);
    GPIO_EnableInt(P0, 2, GPIO_INT_FALLING);

    /* Enable GPIO IRQs in NVIC */
    NVIC_EnableIRQ(GPIO0_IRQn);
    NVIC_EnableIRQ(GPIO1_IRQn);
}
```

1. 此函数使用 Panchip Low-Level GPIO Driver 对 GPIO 进行配置

   实际上也可使用更上层的 Panchip HAL GPIO Driver 进行配置，具体可参考GPIO Digital Input Interrupt例程中的相关介绍

2. 由于 EVB 底板上有 3 个按键，分别对应核心板 P06/P12/P02 等 3 个引脚，因此这里仅配置这 3 个 GPIO 引脚：
   - 配置引脚 Pinmux 至 GPIO 功能
   - 使能去抖功能（并配置去抖时间）
   - 使能 GPIO 数字输入模式
   - 使能内部上拉电阻（按键没有外部上拉电阻）
   - 使能 GPIO 中断，将其配置为下降沿触发中断（即下降沿唤醒），并使能相关 NVIC IRQ

5.2.4 GPIO 中断服务程序

GPIO P0 和 P1 的中断服务程序分别如下：

```
void GPIO0_IRQHandler(void)
{
    BaseType_t xHigherPriorityTaskWoken = pdTRUE;

    for (size_t i = 0; i < 8; i++) {
        if (GPIO_GetIntFlag(P0, BIT(i))) {
            GPIO_ClrIntFlag(P0, BIT(i));
            printf("P0_%d INT occurred.\r\n", i);
            /* Notify the task that gpio key is pressed. */
            vTaskNotifyGiveFromISR(xTaskToNotify, &xHigherPriorityTaskWoken);
        }
    }
}

void GPIO1_IRQHandler(void)
{
    BaseType_t xHigherPriorityTaskWoken = pdTRUE;

    for (size_t i = 0; i < 8; i++) {
        if (GPIO_GetIntFlag(P1, BIT(i))) {
            GPIO_ClrIntFlag(P1, BIT(i));
            printf("P1_%d INT occurred.\r\n", i);
            /* Notify the task that gpio key is pressed. */
            vTaskNotifyGiveFromISR(xTaskToNotify, &xHigherPriorityTaskWoken);
        }
    }
}
```

1. 每个 GPIO Port 均需编写自己的中断服务函数，其内部可通过 `GPIO_GetIntFlag()` 接口判断触发中断的是当前 port 的哪根 pin
2. 在中断服务函数中需注意调用 `GPIO_ClrIntFlag()` 接口清除中断标志位
3. 使用 FreeRTOS `vTaskNotifyGiveFromISR()` 接口向 App Task 发送通知，表示有 GPIO 中断产生（对应按键按下），此接口中 `xHigherPriorityTaskWoken` 变量被配置为 `pdTRUE`，表示当中断返回后将会触发线程调度，而对于此例程来说则是重新调度至 App Task 中的 `ulTaskNotifyTake()` 处继续执行

5.2.5 与低功耗相关的 Hook 函数

本例程还用到了 2 个与低功耗密切相关的 Hook 函数：

```
CONFIG_RAM_CODE void vSocDeepSleepEnterHook(void)
{
#if CONFIG_LOG_ENABLE
    reset_uart_io();
#endif
}

CONFIG_RAM_CODE void vSocDeepSleepExitHook(void)
```

(下页继续)

(续上页)

```
{
#if CONFIG_LOG_ENABLE
    set_uart_io();
#endif
}
```

1. FreeRTOS 有一个优先级最低的 Idle Task，当系统调度到此任务后会对当前状态进行检查，以判断是否允许进入芯片 DeepSleep 低功耗流程
2. 若程序执行到 Idle Task 的 DeepSleep 子流程中，会在 SoC 进入 DeepSleep 模式之前执行 `vSocDeepSleepEnterHook()` 函数，在 SoC 从 DeepSleep 模式下唤醒后执行 `vSocDeepSleepExitHook()` 函数
   - 本例程在 `vSocDeepSleepEnterHook()` 函数中，为防止 UART IO 漏电，编写了相关代码（`reset_uart_io()`，具体实现见例程源码）以确保在进入 DeepSleep 模式前：
     - 串口 Log 数据都打印完毕（即 UART0 Tx FIFO 应为空）
     - P16 引脚 Pinmux 功能由 UART0 Tx 切换回 GPIO
     - P17 引脚 Pinmux 功能由 UART0 Rx 切换回 GPIO，并将其数字输入功能关闭
   - 本例程在 `vSocDeepSleepExitHook()` 函数中，编写了相关代码（`set_uart_io()`，具体实现见例程源码）以恢复串口 Log 打印功能：
     - P16 引脚 Pinmux 功能由 GPIO 重新切换成 UART0 Tx
     - P17 引脚 Pinmux 功能由 GPIO 重新切换成 UART0 Rx，并将其数字输入功能重新打开

5.2.6 其他功能函数

本例程还编写了 2 个比较常用的功能函数：

```
static void parse_stbm1_gpio_wakeup_source(void)
{
    uint32_t src;

    src = soc_stbm1_gpio_wakeup_src_get();

    printf("gpio wakeup src flag = 0x%08x\n", src);

    for (size_t i = 0; i < 32; i++) {
        if (src & (1u << i)) {
            printf("SoC is waked up by GPIO P%d_%d.\n", i / 8, i % 8);
        }
    }
}

#ifdef CONFIG_PM_STANDBY_M1_WAKEUP_WITHOUT_RESET
void restore_hw_peripherals(void)
{
#if CONFIG_LOG_ENABLE
        UART_InitTypeDef Init_Struct = {
                .UART_BaudRate = 921600,
                .UART_LineCtrl = Uart_Line_8n1,
        };
        /* Re-init UART */
        UART_Init(UART0, &Init_Struct);
        UART_EnableFifo(UART0);
        /* Set IO pinmux to UART again */
        set_uart_io();
#endif
        /* Re-enable NVIC GPIO IRQs */
        NVIC_EnableIRQ(GPIO0_IRQn);
        NVIC_EnableIRQ(GPIO1_IRQn);
```

(下页继续)

(续上页)

```
        /* Re-init GPIO keys */
        wakeup_gpio_keys_init();
}
#endif /* CONFIG_PM_STANDBY_M1_WAKEUP_WITHOUT_RESET */
```

1. `parse_stbm1_gpio_wakeup_source()` 用于获取当前 GPIO 中断引脚，本例程中在初始化阶段检测到当前为 Standby Mode 1 GPIO 唤醒后，调用此接口即可获取并打印唤醒 IO 是哪个引脚
2. 当 `CONFIG_PM_STANDBY_M1_WAKEUP_WITHOUT_RESET` 配置被设置为 1 时，芯片从 Standby Mode 1 下唤醒后，将不会触发复位，而是从睡眠之前的代码处接着执行，但由于 Standby Mode 1 状态下芯片内部大部分模块均已掉电，因此醒来后需通过函数 `restore_hw_peripherals()` 重新初始化一些用到的硬件模块

3.2.6 Standby Mode1 SleepTimer Wakeup

1 功能概述

本例程演示如何使 SoC 进入 Standby Mode 1 状态，然后通过 SleepTimer 定时器将其唤醒。

2 环境准备

- 硬件设备与线材：
  - PAN107X EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - **电流计**（本文使用电流可视化测量设备 PPK2 [Nordic Power Profiler Kit II] 进行演示）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 为确保能够准确地测量 SoC 本身的功耗，排除底板外围电路的影响，请确认 EVB 底板上的：
    - Voltage 排针组中的 VCC 和 VDD 均接至 3V3
    - POWER 开关从 LDO 档位拨至 BAT 档位（并确认底板背部的电池座内**没有**纽扣电池）
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将 JLink 仿真器的：
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
  - 将 PPK2 硬件的：
    - USB DATA/POWER 接口连接至 PC USB 接口
    - VOUT 连接至 EVB 底板的 VBAT 排针
    - GND 连接至 EVB 底板的 GND 排针
- PC 软件：
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）

– nRF Connect Desktop（用于配合 PPK2 测量 SoC 电流）

3 编译和烧录

例 程 位 置: `<PAN10XX-NDK>\01_SDK\nimble\samples\low_power\standby_m1_slptmr_wakeup\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录，烧录成功后断开 JLink 连线。

4 例程演示说明

1. PC 上打开 PPK2 Power Profiler 软件，供电电压选择 3300 mV，然后打开供电开关
2. 从串口工具中看到如下的打印信息:

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 6
- Chip MAC Address  : E11000000FF8
- Chip UID          : 6D0001465454455354
- Chip Flash UID    : 4250315A3538380B005B7B4356037D78
- Chip Flash Size   : 512 KB
APP version: 255.255.65535

Reset Reason: nRESET Pin Reset.

Busy wait 5s to keep SoC in active mode..
[Uptime: 1083 ms] SleepTimer 1 IRQ triggered.
[Uptime: 2083 ms] SleepTimer 1 IRQ triggered.
[Uptime: 3083 ms] SleepTimer 1 IRQ triggered.
[Uptime: 4083 ms] SleepTimer 1 IRQ triggered.
[Uptime: 5083 ms] SleepTimer 1 IRQ triggered.

Now try to enter SoC standby mode 1 (wakeup reset)..

Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 6
- Chip MAC Address  : E11000000FF8
- Chip UID          : 6D0001465454455354
- Chip Flash UID    : 4250315A3538380B005B7B4356037D78
- Chip Flash Size   : 512 KB
APP version: 255.255.65535

Reset Reason: Standby Mode 1 SleepTimer 1 Wakeup (cnt = 1).

Busy wait 5s to keep SoC in active mode..
[Uptime: 7167 ms] SleepTimer 1 IRQ triggered.
[Uptime: 8167 ms] SleepTimer 1 IRQ triggered.
...
```

由 Log 中的时间戳可以看出，每隔 1s 触发 SleepTimer 1 中断，触发 5 次中断后，芯片进入 Standby Mode 1，接着被唤醒（复位方式），如此循环

3. 再观察芯片电流波形:

可见芯片基本上每过 5.5s 左右的时间进入低功耗，底电流为 1.2uA 左右，并在一段时间以后被唤醒，如此循环。

图 30: 使用电流计抓取电流波形

5 开发者说明

5.1 App Config 配置　本例程的 App Config（对应 app_config_spark.h 文件）配置与Standby Mode1 GPIO Key Wakeup 例程完全相同：

5.2 程序代码

5.2.1 主程序　主程序 app_main() 函数内容如下：

```
void app_main(void)
{
    BaseType_t r;

    print_version_info();

    /* Create an App Task */
    r = xTaskCreate(app_task,                // Task Function
                    "App Task",              // Task Name
                    APP_TASK_STACK_SIZE,     // Task Stack Size
                    NULL,                    // Task Parameter
                    APP_TASK_PRIORITY,       // Task Priority
                    NULL                     // Task Handle
    );

    /* Check if task has been successfully created */
    if (r != pdPASS) {
        printf("Error, App Task created failed!\n");
        while (1);
    }
}
```

1. 打印 App 版本信息
2. 创建 App 主任务 “App Task”，对应任务函数 app_task
3. 确认线程创建成功，否则打印出错信息

5.2.2 App 主任务　App 主任务 app_task() 函数内容如下：

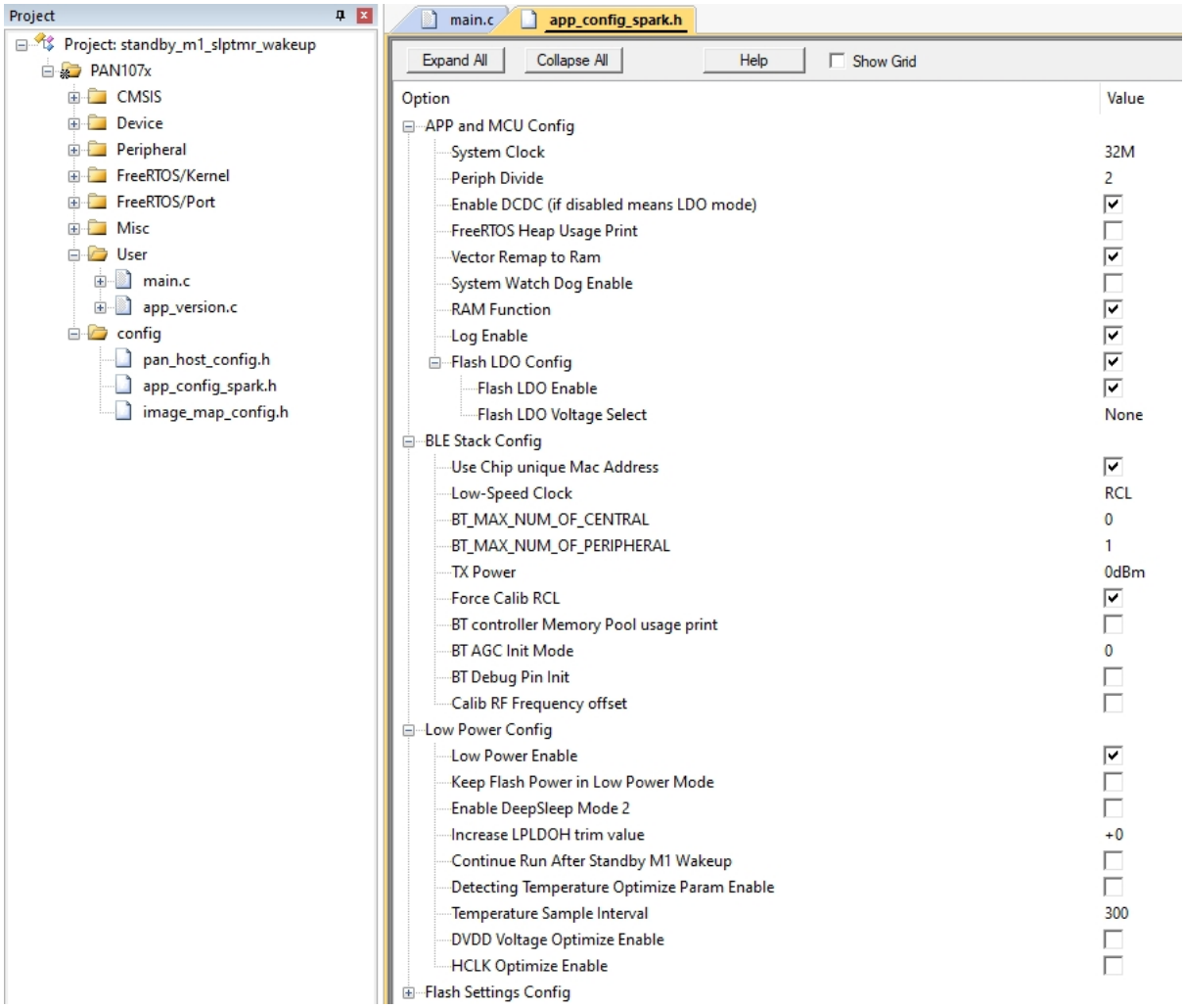


图 31: App Config File

```
void app_task(void *arg)
{
    uint8_t rst_reason;

    /* Get the last reset reason */
    printf("\nReset Reason: ");
    rst_reason = soc_reset_reason_get();
    switch (rst_reason) {
    case SOC_RST_REASON_POR_RESET:
        printf("Power On Reset.\n");
        break;
    case SOC_RST_REASON_PIN_RESET:
        printf("nRESET Pin Reset.\n");
        break;
    case SOC_RST_REASON_SYS_RESET:
        printf("NVIC System Reset.\n");
        break;
    case SOC_RST_REASON_STBM1_SLPTMR1_WAKEUP:
        printf("Standby Mode 1 SleepTimer 1 Wakeup (cnt = %d).\n", ++wkup_cnt);
        break;
    default:
        printf("Unhandled Reset Reason (%d), refer to more reason define in soc.h!\n", rst_
↪reason);
    }

    /* Init Sleep Timer for wake up use */
    slptmr_init();

    printf("\nBusy wait 5s to keep SoC in active mode..\n");
    soc_busy_wait(5500000);

#if !CONFIG_PM_STANDBY_M1_WAKEUP_WITHOUT_RESET
    printf("\nNow try to enter SoC standby mode 1 (wakeup reset)..\n\n");
#if CONFIG_LOG_ENABLE
    /* Waiting for UART Tx done and re-set UART IO before entering standby mode 1 to avoid␣
↪current leakage */
    reset_uart_io();
#endif
    /* Enter standby mode1 with all sram power off */
    soc_enter_standby_mode_1(STBM1_WAKEUP_SRC_SLPTMR, STBM1_RETENTION_SRAM_NONE);

    printf("WARNING: Failed to enter SoC standby mode 1 due to unexpected interrupt detected.\n
↪");
    printf("         Please check if there is an unhandled interrupt during the standby mode 1\
↪n");
    printf("         entering flow.\n");

    while (1) {
        /* Busy wait */
    }
#else
    while (1) {
        printf("\nNow try to enter SoC standby mode 1 (wakeup continuous run)..\n\n");
#if CONFIG_LOG_ENABLE
        /* Waiting for UART Tx done and re-set UART IO before entering standby mode 1 to avoid␣
↪current leakage */
        reset_uart_io();
#endif
        /* Enter standby mode1 with all common 48KB sram retention */
        soc_enter_standby_mode_1(STBM1_WAKEUP_SRC_SLPTMR, STBM1_RETENTION_SRAM_BLOCK0 | STBM1_
↪RETENTION_SRAM_BLOCK1);
        /* Restore hardware peripherals manually */
```

(下页继续)

(续上页)

```
        restore_hw_peripherals();
        printf("Waked up from SoC standby mode 1 and continue run (cnt = %d)..\n", ++wkup_cnt);
        printf("\nBusy wait 5s to keep SoC in active mode..\n");
        soc_busy_wait(5000000);
    }
#endif /* CONFIG_PM_STANDBY_M1_WAKEUP_WITHOUT_RESET */
}
```

1. 获取并打印本次芯片启动的复位原因
2. 在 slptmr_init() 函数中初始化 SleepTimer 1 配置
3. 调用 `soc_busy_wait()` 接口使芯片在 Active 状态下全速运行 5.5s
4. 根据 `CONFIG_PM_STANDBY_M1_WAKEUP_WITHOUT_RESET` 配置，决定进入 Standby Mode 1 的流程，这里由于我们将此配置设置为 0，因此会执行**唤醒后复位**的 Standby Mode 1 流程
5. 进入 Standby Mode 1 前，先将 Log UART IO 配置还原为 GPIO 模拟输入状态，以避免 Standby 模式下 IO 漏电
6. 调用 `soc_enter_standby_mode_1()` 接口，使系统进入 Standby Mode 1 低功耗状态，此接口接受两个参数：第一个参数用于配置唤醒源，第二个参数用于配置低功耗下保电的 SRAM，这里我们将唤醒源配置为 SleepTimer，且低功耗状态下所有 SRAM 均掉电以节约功耗（由于当前低功耗配置为唤醒后复位，因此所有 SRAM 均没有保电的必要）

5.2.3 SleepTimer **初始化程序**

SleepTimer 1 初始化程序 slptmr_init() 函数内容如下：

```
static void slptmr_init(void)
{
    /* Configure timeout of SleepTimer 1 with interrupt and wakeup enabled */

    /*
     * Configure timeout:
     * timeout1 = SLPTMR1_TIMEOUT_CNT / FREQ_32K_HZ
     *          = (32000 / 1) / 32000 (s)
     *          = 1 s
     */
    const uint32_t SLPTMR1_TIMEOUT_CNT = soc_32k_clock_freq_get() / 1;

    /* Set timeout of SleepTimer 1 */
    LP_SetSleepTime(ANA, SLPTMR1_TIMEOUT_CNT, 1);
}
```

1. 此函数功能是将 SleepTimer 1 超时时间配置为 1s
2. 配置时间的具体公式计算请参考代码注释

5.2.4 SleepTimer 1 **中断服务回调函数**

SleepTimer 1 的中断服务回调函数如下：

```
/* This function overrides the reserved weak function with same name in os_lp.c */
void sleep_timer1_handler(void)
{
    printf("[Uptime: %d ms] SleepTimer 1 IRQ triggered.\n", soc_lptmr_uptime_get_ms());
}
```

1. 芯片共有 3 个 SleepTimer，分别为 SleepTimer 0、SleepTimer 1 和 SleepTimer 2，它们共用一个中断服务函数 `SLPTMR_IRQHandler()`，此函数是在 os_lp.c 中实现的；其中 SleepTimer0 被用作 OS Tick，因此在 App 层只可通过回调函数使用 SleepTimer 1 和 SleepTimer 2
2. 本例程我们在 App 中实现了 `sleep_timer1_handler()` 中断回调函数，在其中打印芯片上电时间戳 Log

5.2.5 **与低功耗相关的 Hook 函数** 本例程还用到了 2 个与低功耗密切相关的 Hook 函数:

```
CONFIG_RAM_CODE void vSocDeepSleepEnterHook(void)
{
#if CONFIG_LOG_ENABLE
    reset_uart_io();
#endif
}

CONFIG_RAM_CODE void vSocDeepSleepExitHook(void)
{
#if CONFIG_LOG_ENABLE
    set_uart_io();
#endif
}
```

1. 上述两个 Hook 函数用于在进入 DeepSleep 前和从 DeepSleep 唤醒后做一些额外操作，如关闭和重新配置 IO 为串口功能，以防止 DeepSleep 状态下 IO 漏电
2. 详细解释请参考Standby Mode1 GPIO Key Wakeup 例程中的相关介绍

5.2.6 **其他功能函数** 本例程还编写了 1 个用于 Standby Mode 1 非 Reset 方式下唤醒后的外设恢复函数:

```
#if CONFIG_PM_STANDBY_M1_WAKEUP_WITHOUT_RESET
void restore_hw_peripherals(void)
{
#if CONFIG_LOG_ENABLE
    UART_InitTypeDef Init_Struct = {
        .UART_BaudRate = 921600,
        .UART_LineCtrl = Uart_Line_8n1,
    };
    /* Re-init UART */
    UART_Init(UART0, &Init_Struct);
    UART_EnableFifo(UART0);
    /* Set IO pinmux to UART again */
    set_uart_io();
#endif
    /* Re-enable Sleep Timer interrupt */
    NVIC_EnableIRQ(SLPTMR_IRQn);
}
#endif /* CONFIG_PM_STANDBY_M1_WAKEUP_WITHOUT_RESET */
```

本例程中，我们可以通过手动将 `CONFIG_PM_STANDBY_M1_WAKEUP_WITHOUT_RESET` 配置设置为 1，效果是芯片从 Standby Mode 1 下唤醒后，将不会触发复位，而是从睡眠之前的代码处接着执行，但由于 Standby Mode 1 状态下芯片内部大部分模块均已掉电，因此醒来后需通过函数 `restore_hw_peripherals()` 重新初始化一些用到的硬件模块。

3.2.7 Standby Mode0 P02 Key Wakeup

1 功能概述

本例程演示如何使 SoC 进入 Standby Mode 0 状态，然后通过 WKUP (P02) 按键将其唤醒。

Standby Mode 0 是芯片的最低功耗模式，只支持 P00/P01/P02 这三个 IO 将芯片唤醒（唤醒电平可配且共用）。由于 P00/P01 两个引脚默认为 SWD 功能，为方便演示本例程仅以 P02 唤醒为例进行说明。

2 环境准备

- 硬件设备与线材：
  - PAN107X EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - **电流计**（本文使用电流可视化测量设备 PPK2 [Nordic Power Profiler Kit II] 进行演示）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 为确保能够准确地测量 SoC 本身的功耗，排除底板外围电路的影响，请确认 EVB 底板上的：
    - Voltage 排针组中的 VCC 和 VDD 均接至 3V3
    - POWER 开关从 LDO 档位拨至 BAT 档位（并确认底板背部的电池座内**没有**纽扣电池）
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将 JLink 仿真器的：
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
  - 将 PPK2 硬件的：
    - USB DATA/POWER 接口连接至 PC USB 接口
    - VOUT 连接至 EVB 底板的 VBAT 排针
    - GND 连接至 EVB 底板的 GND 排针
- PC 软件：
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）
  - nRF Connect Desktop（用于配合 PPK2 测量 SoC 电流）

3 编译和烧录

例程位置：`<PAN10XX-NDK>\01_SDK\nimble\samples\low_power\standby_m0_p02_key_wakeup\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录，烧录成功后断开 JLink 连线以避免漏电。

4 例程演示说明

1. PC 上打开 PPK2 Power Profiler 软件，供电电压选择 3300 mV，然后打开供电开关
2. 从串口工具中看到如下的打印信息：

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 6
- Chip MAC Address  : E11000000FF8
```

（下页继续）

(续上页)

```
- Chip UID          : 6D0001465454455354
- Chip Flash UID    : 4250315A3538380B005B7B4356037D78
- Chip Flash Size   : 512 KB
APP version: 255.255.65535

Reset Reason: nRESET Pin Reset.

Busy wait 100ms to keep SoC in active mode..
Try to enter SoC sleep/deepsleep mode for 1000ms..
Waked up from SoC sleep/deepsleep mode.
Try to enter SoC standby mode 0..
```

3. 此时观察芯片电流波形，发现稳定在 220nA 左右（说明芯片成功进入了 Standby Mode 0 状态）：

图 32: 系统初始化后进入 Standby Mode 0 状态

芯片低功耗状态下的底电流（漏电流）与环境温度相关，温度越高，漏电流越大。

4. 尝试按下 EVB 底板上的 WKUP 按键，由串口打印信息可知触发了唤醒并复位，复位的原因为 Standby Mode 0 EXTIO Wakeup（即 WKUP 按键唤醒），随后芯片依次进入 100ms 的 Busy 状态，1s 的 DeepSleep 状态，接着唤醒后进入 Standby Mode 0 状态继续等待 WKUP 按键唤醒：

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 6
- Chip MAC Address  : E11000000FF8
- Chip UID          : 6D0001465454455354
- Chip Flash UID    : 4250315A3538380B005B7B4356037D78
- Chip Flash Size   : 512 KB
APP version: 255.255.65535

Reset Reason: Standby Mode 0 EXTIO Wakeup (cnt = 1).

Busy wait 100ms to keep SoC in active mode..
Try to enter SoC sleep/deepsleep mode for 1000ms..
Waked up from SoC sleep/deepsleep mode.
Try to enter SoC standby mode 0..
```

5 开发者说明

5.1 App Config 配置

本例程的 App Config（对应 app_config_spark.h 文件）配置如下：

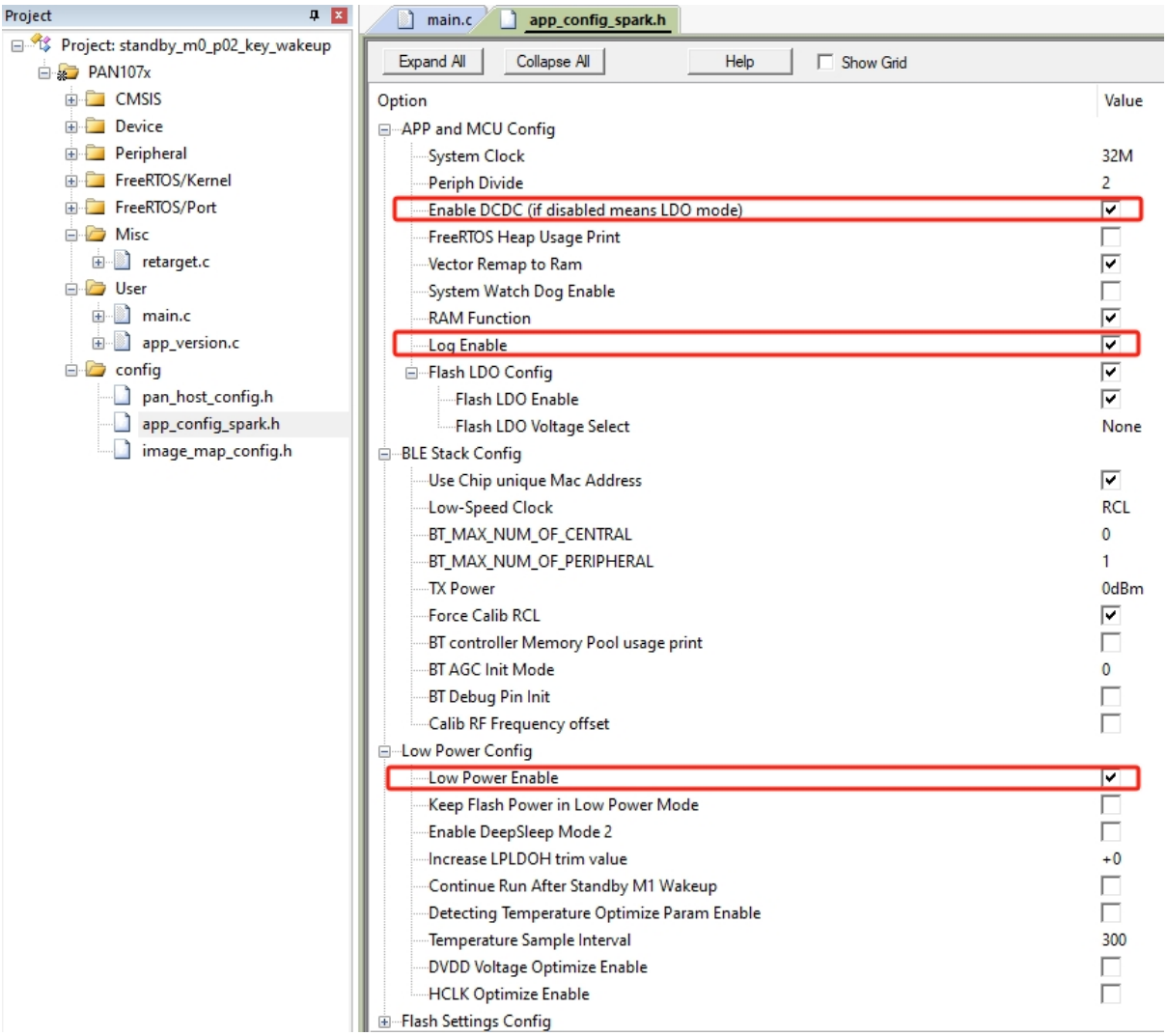


图 33: App Config File

其中，与本例程相关的配置有：

- Enable DCDC (`CONFIG_SOC_DCDC_PAN1070 = 1`)：使能芯片 DCDC 供电模式，以降低芯片动态功耗
- Log Enable (`CONFIG_LOG_ENABLE = 1`)：使能串口 Log 输出
- Low Power Enable (`CONFIG_PM = 1`)：使能系统低功耗流程

5.2 程序代码

5.2.1 主程序

主程序 app_main() 函数内容如下：

```
void app_main(void)
{
    BaseType_t r;

    print_version_info();

    /* Create an App Task */
    r = xTaskCreate(app_task,                // Task Function
                    "App Task",              // Task Name
                    APP_TASK_STACK_SIZE,     // Task Stack Size
                    NULL,                    // Task Parameter
                    APP_TASK_PRIORITY,       // Task Priority
                    NULL                     // Task Handle
    );

    /* Check if task has been successfully created */
    if (r != pdPASS) {
        printf("Error, App Task created failed!\n");
        while (1);
    }
}
```

1. 打印 App 版本信息
2. 创建 App 主任务 “App Task”，对应任务函数 `app_task`
3. 确认线程创建成功，否则打印出错信息

5.2.2 App 主任务

App 主任务 app_task() 函数内容如下：

```
void app_task(void *arg)
{
    uint8_t rst_reason;

    /* Get the last reset reason */
    printf("\nReset Reason: ");
    rst_reason = soc_reset_reason_get();
    switch (rst_reason) {
    case SOC_RST_REASON_POR_RESET:
        printf("Power On Reset.\n");
        break;
    case SOC_RST_REASON_PIN_RESET:
        printf("nRESET Pin Reset.\n");
        break;
    case SOC_RST_REASON_SYS_RESET:
        printf("NVIC System Reset.\n");
        break;
    case SOC_RST_REASON_STBMO_EXTIO_WAKEUP:
        printf("Standby Mode 0 EXTIO Wakeup (cnt = %d).\n", ++wkup_cnt);
```

(下页继续)

(续上页)

```
        break;
    default:
        printf("Unhandled Reset Reason, refer to more reason define in os_lp.h!\n");
    }

    /* Enable P02 low level wakeup for standby mode 0 */
    wakeup_p02_key_init(0);

    printf("\nBusy wait 100ms to keep SoC in active mode..\n");
    soc_busy_wait(100000);

    printf("Try to enter SoC sleep/deepsleep mode for 1000ms..\n");
    vTaskDelay(pdMS_TO_TICKS(1000));
    printf("Waked up from SoC sleep/deepsleep mode.\n");

    printf("Try to enter SoC standby mode 0..\n\n");
    soc_busy_wait(1000); /* Wait for all log print done */
    soc_enter_standby_mode_0();

    printf("WARNING: Failed to enter SoC standby mode 0 due to unexpected interrupt detected.\n
↪");
    printf("         Please check if there is an unhandled interrupt during the standby mode 0\
↪n");
    printf("         entering flow.\n");

    while (1) {
        /* Busy wait */
    }
}
```

1. 获取并打印本次芯片启动的复位原因
2. 在 wakeup_p02_key_init() 函数中初始化 EXTIO (P02) 配置
3. 调用 `soc_busy_wait()` 接口使芯片在 Active 状态下全速运行 100ms
4. 调用 `vTaskDelay()` 接口使系统切换至 Idle Task 执行 1s，实际上会使芯片进入持续 1s 的 DeepSleep 状态
5. 调用 `soc_busy_wait()` 接口延时 1ms，目的是确保所有串口打印消息都发送完成
6. 调用 `soc_enter_standby_mode_0()` 接口，使系统进入 Standby Mode 0 低功耗状态

5.2.3 WKUP Key（P02）初始化程序

P02 初始化程序 wakeup_p02_key_init() 函数内容如下：

```
static void wakeup_p02_key_init(uint8_t wakeup_edge)
{
    /* Configure GPIO P02 (WKUP Key) as Lowlevel Wakeup */

    /* Configure P02 wakeup level due to wakeup_edge */
    LP_SetExternalWake(ANA, wakeup_edge);

    /* Set pinmux func of P56 as GPIO */
    SYS_SET_MFP(P0, 2, GPIO);
    /* Set P02 to input mode */
    GPIO_SetMode(P0, BIT2, GPIO_MODE_INPUT);

    if (wakeup_edge == 0) {
        /* Enable internal pull-up resistor if P02 is low level wakeup */
        GPIO_EnablePullupPath(P0, BIT2);
    } else {
        /* Enable internal pull-down resistor if P02 is high level wakeup */
```

(下页继续)

(续上页)

```
        GPIO_EnablePulldownPath(P0, BIT2);
    }

    /* Necessary for P02 to do manual 3v aon sync */
    CLK_Wait3vSyncReady();

    /* Wait for a while to ensure the internal pullup is stable before entering low power mode␣
↪*/
    soc_busy_wait(10000);
}
```

1. 配置 EXTIO 的唤醒电平
2. 配置 P02 引脚 Pinmux 至 GPIO 功能
3. 配置 P02 为数字输入模式
4. 根据唤醒电平使能内部上拉或下拉电阻
5. 执行一次 3v always-on 寄存器同步动作，以确保 P02 引脚的上下拉电阻配置生效
6. 等待一段时间，确保上下拉电阻稳定

5.2.4 与低功耗相关的 Hook 函数

本例程还用到了 2 个与低功耗密切相关的 Hook 函数：

```
CONFIG_RAM_CODE void vSocDeepSleepEnterHook(void)
{
#if CONFIG_LOG_ENABLE
    reset_uart_io();
#endif
}

CONFIG_RAM_CODE void vSocDeepSleepExitHook(void)
{
#if CONFIG_LOG_ENABLE
    set_uart_io();
#endif
}
```

1. 上述两个 Hook 函数用于在进入 DeepSleep 前和从 DeepSleep 唤醒后做一些额外操作，如关闭和重新配置 IO 为串口功能，以防止 DeepSleep 状态下 IO 漏电
2. 详细解释请参考Standby Mode1 GPIO Key Wakeup 例程中的相关介绍

3.2.8 Multiple Wakeup Source

1 功能概述

本例程是一个稍微复杂一些的应用，用于演示多种唤醒源、多种低功耗模式的切换，具体包括以下功能：

- DeepSleep 状态下，通过 UART1 Rx 引脚唤醒芯片，并完成 UART1 数据接收（对应 SoC GPIO 唤醒）
- DeepSleep 状态下，通过 SleepTimer 定时唤醒
- Standby Mode 1 状态下，通过按键唤醒芯片（对应 SoC GPIO 唤醒）

2 环境准备

- 硬件设备与线材：
  - PAN107X EVB 核心板与底板各一块

  - JLink 仿真器（用于烧录例程程序）
  - **电流计**（本文使用电流可视化测量设备 PPK2 [Nordic Power Profiler Kit II] 进行演示）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - USB 转串口模块一个（用于与芯片 UART1 通信）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 为确保能够准确地测量 SoC 本身的功耗，排除底板外围电路的影响，请确认 EVB 底板上的：
    * Voltage 排针组中的 VCC 和 VDD 均接至 3V3
    * POWER 开关从 LDO 档位拨至 BAT 档位（并确认底板背部的电池座内**没有**纽扣电池）
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17（用于打印串口 Log）
  - 使用杜邦线将 USB 转串口模块的 RX 引脚与核心板上的 P10 引脚相连，TX 引脚与核心板上的 P07 引脚相连（用于通过 UART1 接收 PC 发送的数据）
  - 使用杜邦线将 JLink 仿真器的：
    * SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    * SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    * SWD_GND 引脚与 EVB 底板的 GND 排针相连
  - 将 PPK2 硬件的：
    * USB DATA/POWER 接口连接至 PC USB 接口
    * VOUT 连接至 EVB 底板的 VBAT 排针
    * GND 连接至 EVB 底板的 GND 排针
- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（接至芯片 UART0，用于接收串口打印 Log）
  - 串口调试助手（UartAssist），波特率 9600（接至芯片 UART1，用于接受 PC 发送过来的数据）
  - nRF Connect Desktop（用于配合 PPK2 测量 SoC 电流）

3 编译和烧录

例程位置：`<PAN10XX-NDK>\01_SDK\nimble\samples\low_power\multiple_wakeup_source\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录，烧录成功后断开 JLink 连线以避免漏电。

4 例程演示说明

1. PC 上打开 PPK2 Power Profiler 软件，供电电压选择 3300 mV，然后打开供电开关
2. 从串口 Log (UART0) 中看到如下的打印信息：

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 6
- Chip MAC Address  : E1100000101D
- Chip UID          : 0D0001465454455354
- Chip Flash UID    : 4250315A3538380B004B554356034578
- Chip Flash Size   : 512 KB
APP version: 130.99.13608

Reset Reason: nRESET Pin Reset.
Busy wait a while..
Wait for Task Notifications..
SleepTimer1 started, timeout=30s
```

3. 此时观察芯片电流波形，发现稳定在 4uA 左右（说明芯片成功进入了 DeepSleep 模式）：

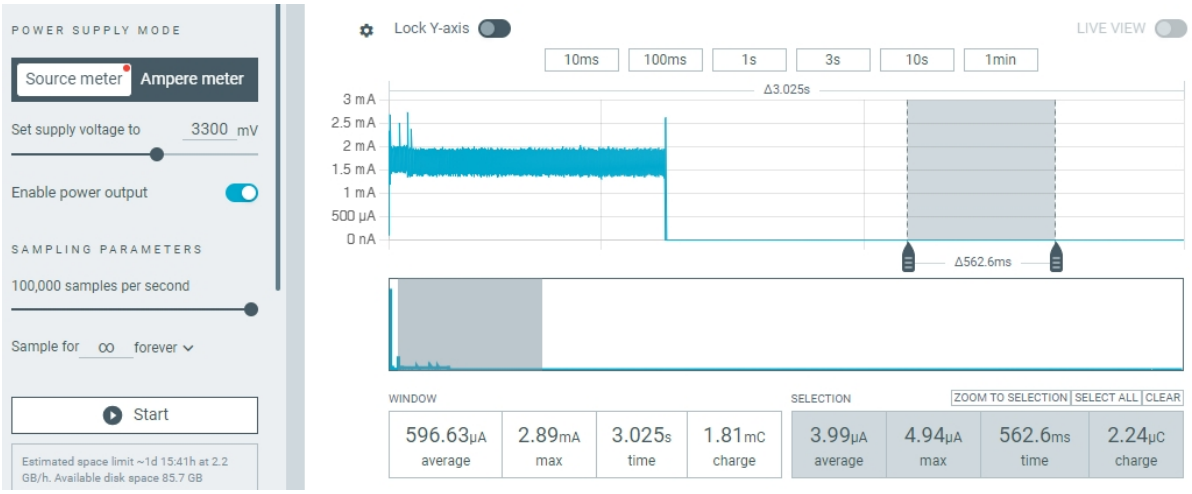


图 34: 系统初始化后进入 DeepSleep 模式

4. 在 30s 内，使用串口调试助手通过 UART1（波特率 9600）向芯片发送起始字节为 0x00 的二进制字节串：

5. 此时再观察串口 Log，可以看到成功检测到 GPIO P07 中断，随后 app task 向下执行，SleepTimer1 被停止，随后 UART1 中断服务程序触发，收到并打印除起始字节 0x00 外的剩下的所有数据，随后 app task 重新执行到等待 Notification 的位置，并触发系统调度到 Idle 线程，并触发重新启动 SleepTimer1 的流程：

```
UART1 Rx (P07) GPIO wakeup triggered..
A notification received, value: 1.

SleepTimer1 stopped.
Busy wait a while..
Data received from UART: 0x01 0x02 0x03 0x04
Wait for Task Notifications..
SleepTimer1 started, timeout=30s
```

6. 再观察芯片电流波形，可以看到芯片触发了 DeepSleep 唤醒，一次唤醒持续了约 1s 左右时间，随后重新进入 DeepSleep 状态等待下次唤醒：

7. 这次在 DeepSleep 状态下等待 30s 以上，可以看到会触发 SleepTimer1 超时唤醒，并随即进入 Standby Mode 1 状态：

```
Soc has stayed in deepsleep mode more than 30s, now try to enter standby mode 1..
```

图 35: 通过串口将芯片从 DeepSleep 状态下唤醒

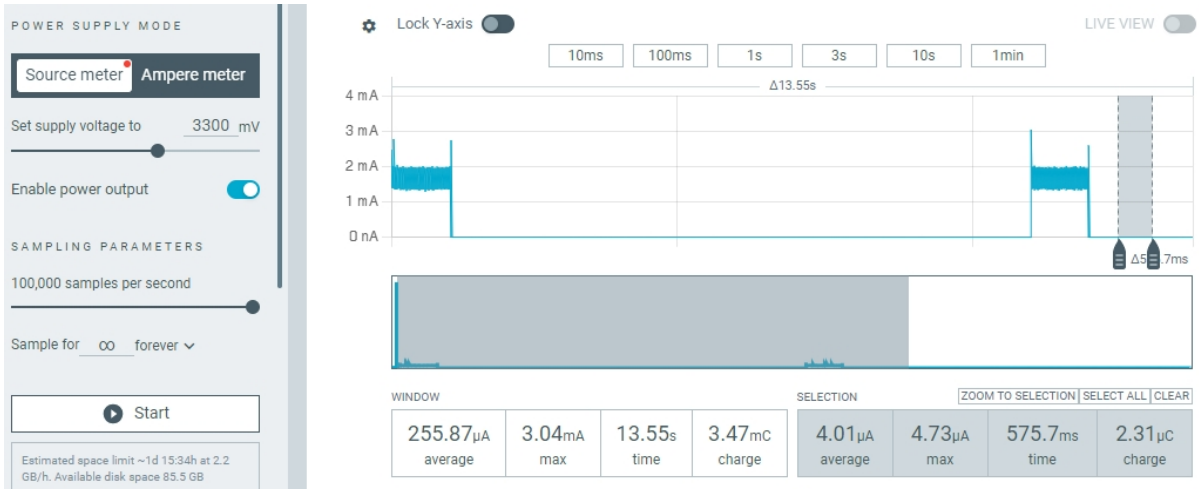


图 36: 通过 UART1 Rx IO（P07）的 GPIO 方式将系统唤醒，之后重新进入 DeepSleep 状态

8. 此时再观察芯片电流波形，可以看到芯片触发了 DeepSleep 唤醒，随后很快进入了 Standby Mode 1 状态（从底电流变为 1uA 左右可看出）：

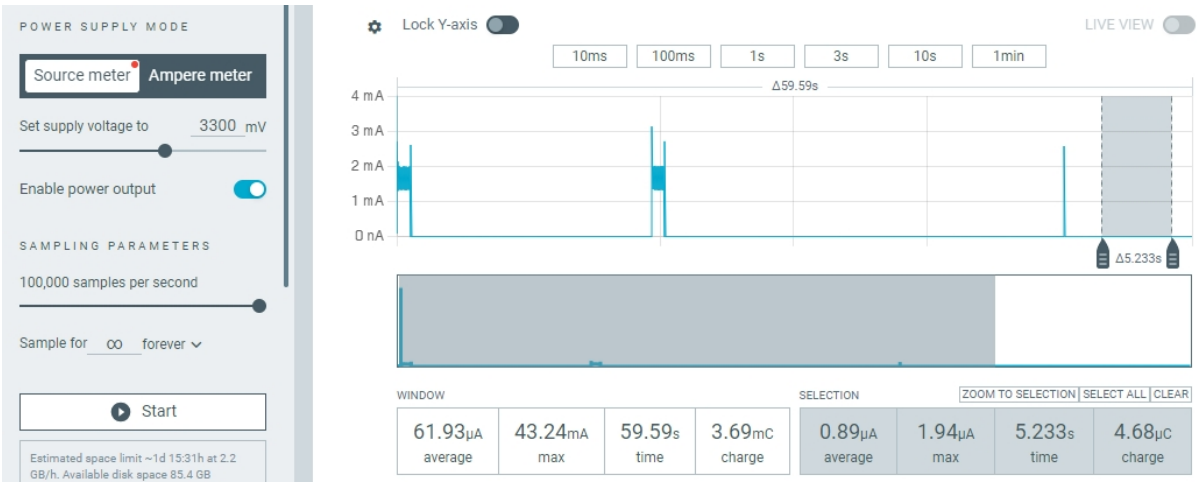


图 37: 通过 SleepTimer1 将芯片从 DeepSleep 状态下唤醒，随即进入 Standby Mode 1 状态

9. 本例程中 Stanby Mode 1 状态下至支持 WKUP 按键（P02）唤醒，此时我们按一下底板的 WKUP 按键，可以看到芯片成功被唤醒，随后重新进入 DeepSleep 状态：

```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 6
- Chip MAC Address   : E1100000101D
- Chip UID           : 0D0001465454455354
- Chip Flash UID     : 4250315A3538380B004B554356034578
- Chip Flash Size    : 512 KB
APP version: 130.99.13608

Reset Reason: Standby Mode 1 GPIO Wakeup.
gpio wakeup src flag = 0x00000004
SoC is waked up by GPIO P0_2.

WKUP key (P02) pressed..
Busy wait a while..
Wait for Task Notifications..
SleepTimer1 started, timeout=30s
A notification received, value: 1.

SleepTimer1 stopped.
Busy wait a while..
Wait for Task Notifications..
SleepTimer1 started, timeout=30s
```

5 开发者说明

5.1 Config 配置

1. 本例程的 App Config（对应 app_config_spark.h 文件）配置与Standby Mode1 GPIO Key Wakeup 例程完全相同：

2. 除 App Config 外，本例程还使用到了 FreeRTOS 的 Applicaiton Idle Hook 机制，因此还需要在 FreeRTOSConfig.h 中使能此机制：

5.2 程序代码

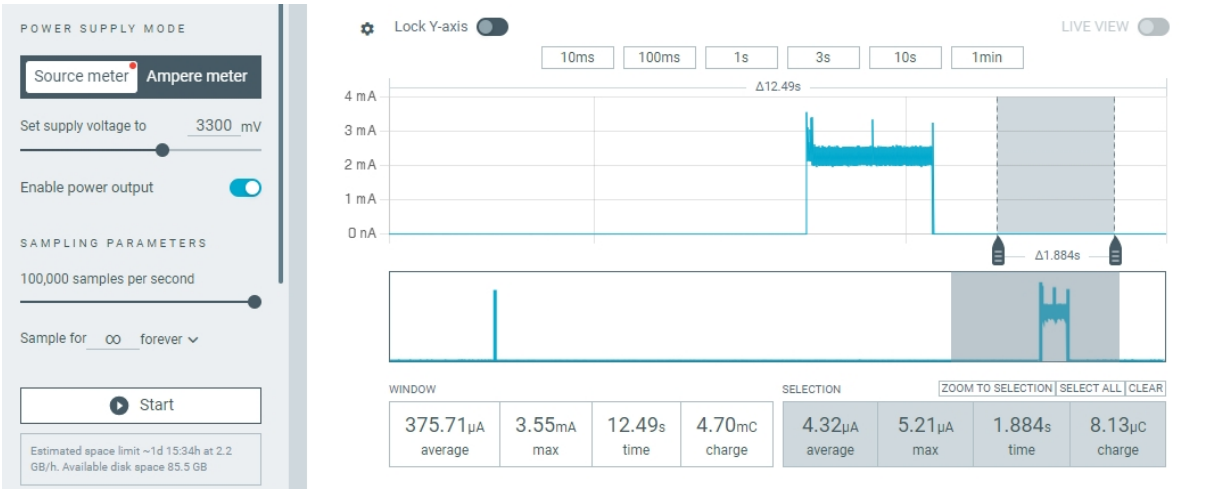


图 38: 通过 WKUP 按键将芯片从 Standby Mode 1 状态下唤醒，随后重新进入 DeepSleep 状态

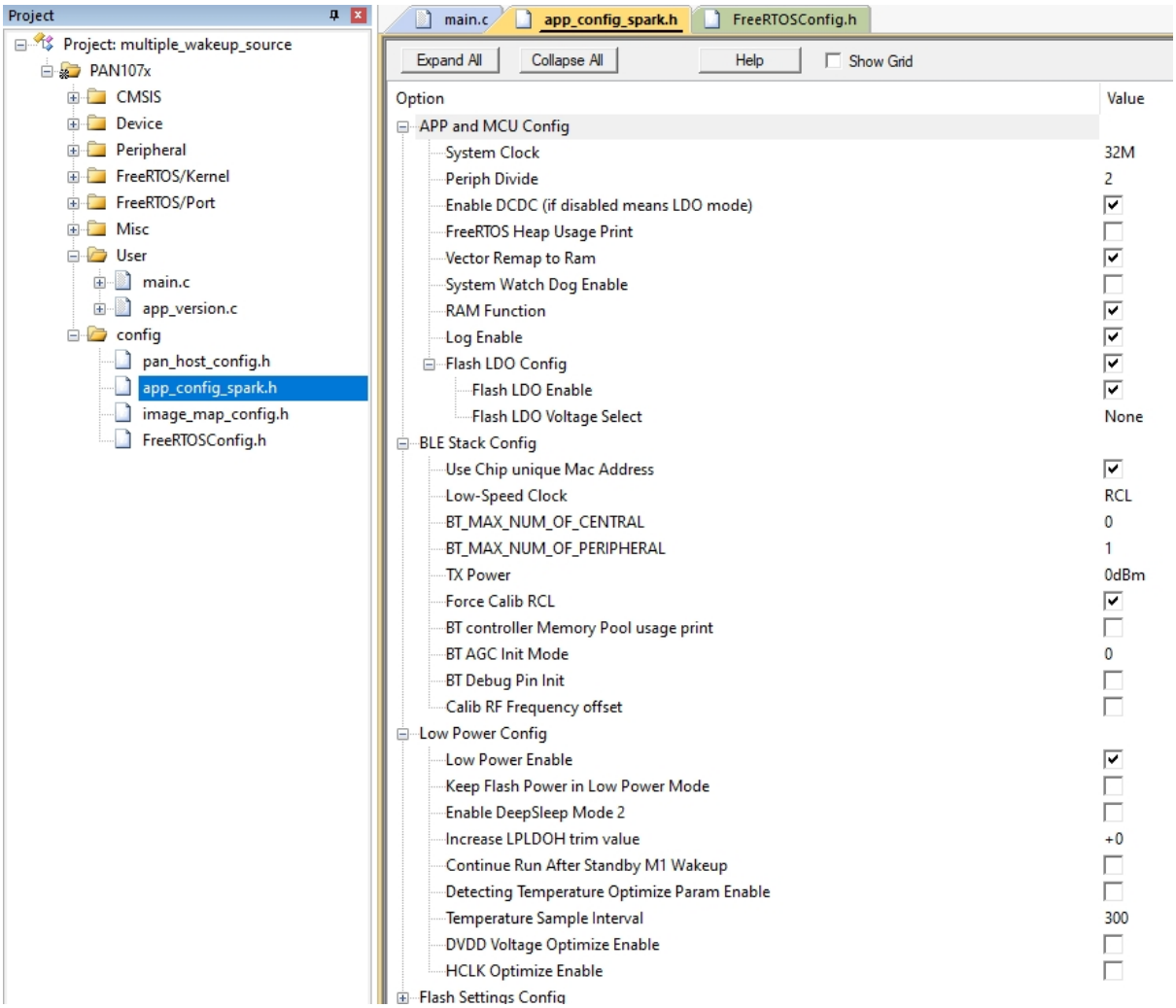


图 39: App Config File

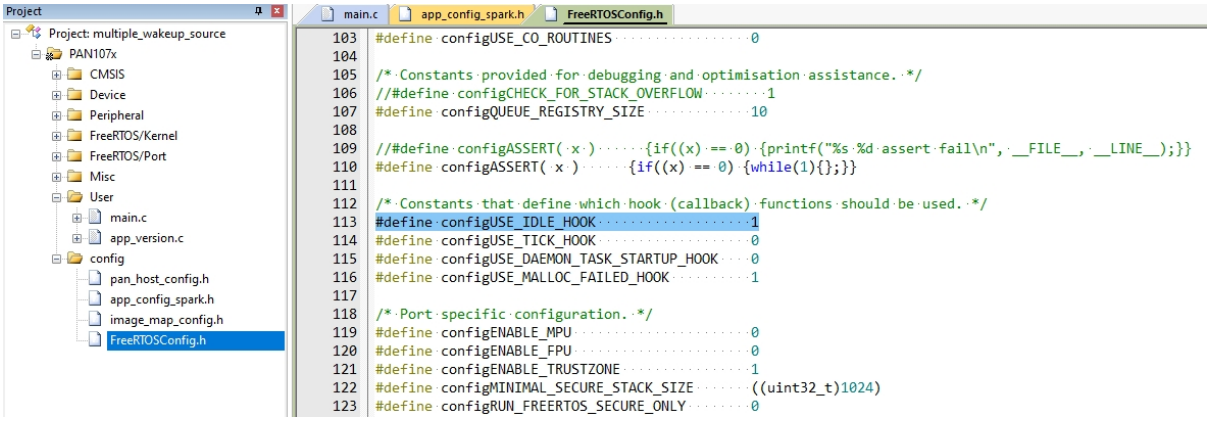


图 40: FreeRTOS Config File

5.2.1 **主程序** 主程序 app_main() 函数内容如下：

```
void app_main(void)
{
    BaseType_t r;

    print_version_info();

    /* Create an App Task */
    r = xTaskCreate(app_task,               // Task Function
                    "App Task",             // Task Name
                    APP_TASK_STACK_SIZE,    // Task Stack Size
                    NULL,                   // Task Parameter
                    APP_TASK_PRIORITY,      // Task Priority
                    NULL                    // Task Handle
    );

    /* Check if task has been successfully created */
    if (r != pdPASS) {
        printf("Error, App Task created failed!\n");
        while (1);
    }
}
```

1. 打印 App 版本信息
2. 创建 App 主任务 “App Task”，对应任务函数 app_task
3. 确认线程创建成功，否则打印出错信息

5.2.2 App **主任务** App 主任务 app_task() 函数内容如下：

```
void app_task(void *arg)
{
    uint32_t ulNotificationValue;
    uint8_t rst_reason;

    /* Store the handle of current task. */
    xTaskToNotify = xTaskGetCurrentTaskHandle();
    if (xTaskToNotify == NULL) {
        printf("Error, get current task handle failed!\n");
        while (1);
    }
```

(下页继续)

(续上页)

```
    /* Get the last reset reason */
    printf("\nReset Reason: ");
    rst_reason = soc_reset_reason_get();
    switch (rst_reason) {
    case SOC_RST_REASON_POR_RESET:
        printf("Power On Reset.\n");
        break;
    case SOC_RST_REASON_PIN_RESET:
        printf("nRESET Pin Reset.\n");
        break;
    case SOC_RST_REASON_SYS_RESET:
        printf("NVIC System Reset.\n");
        break;
    case SOC_RST_REASON_STBM1_GPIO_WAKEUP:
        printf("Standby Mode 1 GPIO Wakeup.\n");
        parse_stbm1_gpio_wakeup_source();
        break;
    default:
        printf("Unhandled Reset Reason (%d), refer to more reason define in os_lp.h!\n", rst_
↪reason);
    }

    /* Enable and init UART1 for data transmission */
    uart1_comm_init();

    /* Enable specific gpios wakeup for wakeup use */
    wakeup_gpio_init();

    while (1) {
        /* Busy wait a while to simulate cpu busy status */
        printf("Busy wait a while..\n");
        soc_busy_wait(1000000);

        /* Try to take wakeup_sem */
        printf("Wait for Task Notifications..\n");
        /*
         * Wait to be notified that gpio key is pressed (gpio irq occured). Note the first␣
↪parameter
         * is pdTRUE, which has the effect of clearing the task's notification value back to 0,
↪ making
         * the notification value act like a binary (rather than a counting) semaphore.
         */
        ulNotificationValue = ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
        printf("A notification received, value: %d.\n\n", ulNotificationValue);

        /* Stop the slptmr1 timer (by resetting the counter) */
        LP_SetSleepTime(ANA, 0, 1);
        printf("SleepTimer1 stopped.\n");
    }
}
```

1. 获取当前任务的 Task Handle，用于后续中断中给次任务发送通知使用
2. 获取并打印本次芯片启动的复位原因，若复位原因为 Standby Mode 1 GPIO 唤醒，则额外解析并打印唤醒 IO 是哪个引脚
3. 在 uart1_comm_init() 函数中初始化 UART1 通信配置
4. 在 wakeup_gpio_init() 函数中初始化按键 GPIO P02 配置
5. 在随后的 `while(1)` 循环中：
    1. 调用 `soc_busy_wait()` 接口使芯片在 Active 状态下全速运行 1s

2. 尝试获取任务通知（Task Task Notify），并打印相关的状态信息
3. 若成功获取到任务通知，则停止 SleepTimer 1 定时器

5.2.3 UART1 初始化程序

UART1 初始化程序 uart1_comm_init() 函数内容如下：

```
static void uart1_comm_init(void)
{
    /* Enable UART1 Clock */
    CLK_APB2PeriphClockCmd(CLK_APB2Periph_UART1, ENABLE);

    /* Configure UART Pinmux */
    SYS_SET_MFP(P1, 0, UART1_TX);
    SYS_SET_MFP(P0, 7, UART1_RX);
    /* Enable digital input path of UART Rx Pin */
    GPIO_EnableDigitalPath(P0, BIT7);

    /* Init UART1 */
    UART_InitTypeDef Init_Struct = {
        .UART_BaudRate = 9600,
        .UART_LineCtrl = Uart_Line_8n1,
    };
    UART_Init(UART1, &Init_Struct);
    UART_EnableFifo(UART1);

    /* Configure interrupt of UART1 */
    UART_SetRxTrigger(UART1, UART_RX_FIFO_HALF_FULL);
    UART_EnableIrq(UART1, UART_IRQ_RECV_DATA_AVL);      // Enable RDA Interrupt
    NVIC_EnableIRQ(UART1_IRQn);                         // Enable target UART INT in NVIC
}
```

1. 此函数使用 Panchip Low-Level UART Driver 对 UART 进行配置

   实际上也可使用更上层的 Panchip HAL UART Driver 进行配置，具体可参考UART FIFO 例程中的相关介绍

2. 配置 UART1 的流程为：
   - 开启 APB1 上 UART1 时钟
   - 分别将 P10 和 P07 的 PINMUX 配置为 UART1 Tx 和 RX 功能
   - 使能 UART1 Rx 引脚的数字输入功能
   - 初始化 UART1，参数配置为波特率 9600，数据宽度 8-bit，无奇偶校验，1 位停止位
   - 使能 UART1 的 FIFO 功能
   - 配置 UART1 Rx FIFO 的 Trigger Level 为 Half-Full（即 8 字节）
   - 使能 UART1 Rx Data Receive 中断
   - 使能 NVIC 层的 UART1 IRQ

5.2.4 UART1 中断服务程序

UART1 的中断服务程序如下：

```
static void UART_HandleReceivedData(UART_T* UARTx)
{
    static uint8_t rx_index = 0;

    printf("Data received from UART: ");
    while (!UART_IsRxFifoEmpty(UARTx)) {
        uart_data_buf[rx_index] = UART_ReceiveData(UARTx);
        printf("0x%02x ", uart_data_buf[rx_index]);
        rx_index++;
```

（下页继续）

(续上页)

```
        // Handle Rx buffer full
        if (rx_index >= UART_DATA_BUFF_SIZE) {
            printf("\nWARNING: Too much data received, UART Rx buffer full!\n");
            rx_index = 0;   // Reset Rx buf index
            break;
        }
    }
    printf("\n");
}

static void UART_HandleProc(UART_T *UARTx)
{
    UART_EventDef event = UART_GetActiveEvent(UARTx);

    switch (event) {
    case UART_EVENT_DATA:
    case UART_EVENT_TIMEOUT:
        /* Handle received data */
        UART_HandleReceivedData(UARTx);
        break;
    case UART_EVENT_NONE:
        /* Just ignore this event. */
        break;
    default:
        SYS_TEST("WARNING: Unhandled event, IID: 0x%x\n", event);
        break;
    }
}

void UART1_IRQHandler(void)
{
    UART_HandleProc(UART1);
}
```

1. 上述 3 个函数中，UART1_IRQHandler() 是中断服务程序入口，UART_HandleProc() 是通用的 UART Rx 处理流程模板，UART_HandleReceivedData() 是本例程的逻辑处理
2. UART_HandleReceivedData() 函数的主要逻辑是，当发现 UART Rx FIFO 非空时，将其读空，并将数据存储在一个全局的数组中，同时向串口打印数据信息

5.2.5 GPIO 初始化程序

GPIO 初始化程序 wakeup_gpio_init() 函数内容如下：

```
static void wakeup_gpio_key_init(void)
{
    /* Set pinmux func as GPIO */
    SYS_SET_MFP(P0, 2, GPIO);

    /* Configure debounce clock */
    GPIO_SetDebounceTime(GPIO_DBCTL_DBCLKSRC_RCL, GPIO_DBCTL_DBCLKSEL_4);
    /* Enable input debounce function of specified GPIO */
    GPIO_EnableDebounce(P0, BIT2);

    /* Set GPIO to input mode */
    GPIO_SetMode(P0, BIT2, GPIO_MODE_INPUT);
    CLK_Wait3vSyncReady(); /* Necessary for P02 to do manual aon-reg sync */

    /* Enable internal pull-up resistor path */
    GPIO_EnablePullupPath(P0, BIT2);
    CLK_Wait3vSyncReady(); /* Necessary for P02 to do manual aon-reg sync */
```

(下页继续)

(续上页)

```
    /* Wait for a while to ensure the internal pullup is stable before entering low power mode␣
↪*/
    soc_busy_wait(10000);

    /* Disable P02 interrupt at this time */
    GPIO_DisableInt(P0, 2);
}

static void wakeup_gpio_uart1_rx_pin_init(void)
{
    /* Set GPIO to input mode */
    GPIO_SetMode(P0, BIT7, GPIO_MODE_INPUT);

    /* Enable internal pull-up resistor path */
    GPIO_EnablePullupPath(P0, BIT7);

    /* Disable P07 interrupt before entering deepsleep */
    GPIO_DisableInt(P0, 7);
}

static void wakeup_gpio_init(void)
{
    /* Configure gpio wakeup key (P02) */
    wakeup_gpio_key_init();

    /* Configure gpio wakeup pin for uart rx */
    wakeup_gpio_uart1_rx_pin_init();

    /* Enable GPIO IRQs in NVIC */
    NVIC_EnableIRQ(GPIO0_IRQn);
}
```

本例程中的 GPIO 配置分三部分：

1. GPIO P02 (WKUP 按键) 初始化配置函数 `wakeup_gpio_key_init()`
   - 配置 P02 引脚的 Pinmux 为 GPIO 功能（芯片上电默认功能，因此可以省略）
   - 使能 P02 去抖功能（并配置去抖时间）
   - 配置 P02 为数字输入模式
   - 使能 P02 内部上拉电阻（按键没有外部上拉电阻）
   - 关闭 P02 的中断使能
2. UART Rx（P07）引脚的 GPIO 初始化配置函数 `wakeup_gpio_uart1_rx_pin_init()`
   - 配置 P07 引脚为数字输入功能
   - 使能 P07 引脚的内部上拉电阻
   - 关闭 P07 引脚的中断使能
3. 在 `wakeup_gpio_init()` 函数尾部使能 NVIC 层的 GPIO 中断 IRQ

5.2.6 GPIO 中断服务程序

GPIO P0 的中断服务程序如下：

```
CONFIG_RAM_CODE void GPIO0_IRQHandler(void)
{
    /* Check if WKUP (P02) button pressed */
    if (GPIO_GetIntFlag(P0, BIT2)) {
        GPIO_ClrIntFlag(P0, BIT2);
        printf("\nWKUP key (P02) pressed..\n");
```

(下页继续)

(续上页)

```
    }

    /* Check if UART1 Rx (P07) GPIO wakeup triggered */
    if (GPIO_GetIntFlag(P0, BIT7)) {
        /* Clear gpio pin irq flag */
        GPIO_ClrIntFlag(P0, BIT7);
        while (P07 == 0) {
            /* Busy wait until P07 pulled high, which means the 1st wakup
             * trigger character (0x00) send done.
             */
        }
        /* Resume UART1 PIN configs after P07 pulled high */
        SYS_SET_MFP(P1, 0, UART1_TX);
        SYS_SET_MFP(P0, 7, UART1_RX);
        GPIO_DisableInt(P0, 7);
        printf("\nUART1 Rx (P07) GPIO wakeup triggered..\n");
    }

    /* Try to do context switch right after current isr return */
    taskYIELD();
}
```

1. 每个 GPIO Port 均需编写自己的中断服务函数，其内部可通过 `GPIO_GetIntFlag()` 接口判断触发中断的是当前 port 的哪根 pin
2. 在中断服务函数中需注意调用 `GPIO_ClrIntFlag()` 接口清除中断标志位
3. 若检测到 GPIO P0 中断触发原因为 P02，则直接打印 Log
4. 若检测到 GPIO P0 中断触发原因为 P07，则说明当前是通过 UART Rx 引脚的方式触发的中断，此时先一直等待直到确定 P07 拉高，然后重新将 P10/P07 引脚的 PINMUX 功能配置为 UART1，并关闭 P07 中断
5. 在中断服务函数结尾通过调用 FreeRTOS `taskYIELD()` 接口告知系统在退出当前中断服务程序后立刻触发尝试任务切换的流程

5.2.7 SleepTimer 1 中断服务回调函数

SleepTimer 1 的中断服务回调函数如下：

```
/* This function overrides the reserved weak function with same name in os_lp.c */
void sleep_timer1_handler(void)
{
    printf("\nSoc has stayed in deepsleep mode more than %ds, now try to enter standby mode 1..
↪\n\n", INTERVAL_FOR_SWITCH_LOWPOWER_MODE);

    /* Wait until all UART0 data sending done before entering standby mode */
    while (!(UART_GetLineStatus(UART0) & UART_LINE_TXSR_EMPTY)) {
        /* Busy wait */
    }

    /* Enable P02 interrupt before entering standby mode for wakeup */
    GPIO_EnableInt(P0, 2, GPIO_INT_FALLING);

    /* Enter standby mode1 with all sram power off */
    soc_enter_standby_mode_1(STBM1_WAKEUP_SRC_GPIO, STBM1_RETENTION_SRAM_NONE);

    printf("WARNING: Failed to enter SoC standby mode 1 due to unexpected interrupt detected.\n
↪");
    printf("         Please check if there is an unhandled interrupt during the standby mode 1\
↪n");
    printf("         entering flow.\n");
```

(下页继续)

(续上页)

```
    while (1) {
        /* Busy wait */
    }
}
```

1. 芯片共有 3 个 SleepTimer，分别为 SleepTimer 0、SleepTimer 1 和 SleepTimer 2，它们共用一个中断服务函数 `SLPTMR_IRQHandler()`，此函数是在 os_lp.c 中实现的；其中 SleepTimer0 被用作 OS Tick，因此在 App 层只可通过回调函数使用 SleepTimer 1 和 SleepTimer 2
2. 本例程我们在 App 中实现了 `sleep_timer1_handler()` 中断回调函数，其行为是：
   - 向串口打印将要进入 Standby Mode 1 状态的 Log
   - 确保 UART0 Tx FIFO 中没有数据（即确保所有串口 Log 均已成功发出）
   - 使能 P02 引脚中断，并配置为下降沿触发中断（唤醒）
   - 调用 `soc_enter_standby_mode_1()` 接口使芯片进入 Standby Mode 1 的 GPIO 唤醒模式，且在此模式下所有 SRAM 均掉电以节省功耗

5.2.8 与低功耗相关的 Hook 函数

本例程用到了 3 个与低功耗密切相关的 Hook 函数：

```
/*
 * This is the application hook function running in OS IDLE task. Before use this, the
 * macro configUSE_IDLE_HOOK should be enabled in FreeRTOSConfig.h.
 *
 * Note that the function definition is a bit different with FreeRTOS original one, that
 * is, here we add a return value for flexibility.
 *
 * Return Value:
 * - false:   Continue run the following code after vApplicationIdleHook() in IDLE task,
 *            which is equivalent to the original FreeRTOS implementation.
 * - true:    Avoid run following code after vApplicationIdleHook() in IDLE task, and this
 *            could prevent SoC entering DeepSleep flow when CONFIG_PM enabled.
 */
bool vApplicationIdleHook(void)
{
    /*
     * Enter sleep mode (instead of deepsleep) if UART Rx FIFO is not empty
     * (Which means there is already data receiving from remote)
     */
    if (!UART_IsRxFifoEmpty(UART1)) {
        __disable_irq();
        __WFI();
        __enable_irq();
        /* Avoid entering DeepSleep flow here */
        return true;
    }

    /* Allow entering DeepSleep flow */
    return false;
}

/*
 * This is hook function runs right after entering SoC DeepSleep Flow.
 * Note that either the os scheduler or systick are stopped before running this
 * function, so do not run any os related objects (such as OS Timer) here.
 */
CONFIG_RAM_CODE void vSocDeepSleepEnterHook(void)
{
    /* Enable and Set timeout of SleepTimer1 */
    uint32_t timeout = soc_32k_clock_freq_get() * INTERVAL_FOR_SWITCH_LOWPOWER_MODE;
```

(下页继续)

(续上页)

```
    LP_SetSleepTime(ANA, timeout, 1);
    printf("SleepTimer1 started, timeout=%ds\n", INTERVAL_FOR_SWITCH_LOWPOWER_MODE);

    /* Handle UART0 (Log UART) */

    /* Wait until all UART0 data sending done before entering deepsleep mode */
    while (!(UART_GetLineStatus(UART0) & UART_LINE_TXSR_EMPTY)) {
        /* Busy wait */
    }
    /* Reset UART0 PINs to GPIO function and disable digital input path of UART0 Rx PIN to␣
↪avoid possible current leakage. */
    SYS_SET_MFP(P1, 6, GPIO);
    SYS_SET_MFP(P1, 7, GPIO);
    GPIO_DisableDigitalPath(P1, BIT7);

    /* Handle UART1 (Data UART) */

    /* Wait until all UART1 data sending done before entering deepsleep mode */
    while (!(UART_GetLineStatus(UART1) & UART_LINE_TXSR_EMPTY)) {
        /* Busy wait */
    }
    /* Reset UART1 PINs to GPIO function and enable gpio interrupt for uart rx pin. */
    SYS_SET_MFP(P1, 0, GPIO);
    SYS_SET_MFP(P0, 7, GPIO);
    GPIO_EnableInt(P0, 7, GPIO_INT_FALLING);

    /* Enable P02 interrupt before entering deepsleep mode for wakeup */
    GPIO_EnableInt(P0, 2, GPIO_INT_FALLING);
}

CONFIG_RAM_CODE void vSocDeepSleepExitHook(void)
{
    /* Resume UART0 PIN Configurations to reenable UART0 function */
    SYS_SET_MFP(P1, 6, UART0_TX);
    SYS_SET_MFP(P1, 7, UART0_RX);
    GPIO_EnableDigitalPath(P1, BIT7);

    /* Resume UART1 PIN Configurations here to reenable UART1 function if is not waked up by␣
↪P07 */
    if (!GPIO_GetIntFlag(P0, BIT7)) {
        SYS_SET_MFP(P1, 0, UART1_TX);
        SYS_SET_MFP(P0, 7, UART1_RX);
        GPIO_DisableInt(P0, 7);
    }

    /* Re-disable P02 interrupt after deepsleep wakeup */
    GPIO_DisableInt(P0, 2);

    /* Trigger App Task reschedule after wakeup from deepsleep mode */
    xTaskNotifyGive(xTaskToNotify);
}
```

1. FreeRTOS 有一个优先级最低的 Idle Task，当系统调度到此任务后会对当前状态进行检查，以判断是否允许进入芯片 DeepSleep 低功耗流程
2. 若程序执行到 Idle Task，则会立刻触发 FreeRTOS 的 Idle Hook 机制，调用名为 `vApplicationIdleHook()` 的函数，我们在此函数中判断 UART1 Rx FIFO 是否为空：
   - 若是则函数直接返回 false，表示允许系统稍后进入 DeepSleep 流程
   - 若非则使系统立刻进入 Sleep（WFI）流程
3. 若程序执行到 Idle Task 的 DeepSleep 子流程中，会在 SoC 进入 DeepSleep 模式

之前执行 `vSocDeepSleepEnterHook()` 函数，在 SoC 从 DeepSleep 模式下唤醒后执行 `vSocDeepSleepExitHook()` 函数：

- 本例程在 `vSocDeepSleepEnterHook()` 函数中，实现了如下逻辑：
  - 使能 SleepTimer1，并设置 30s 超时时间
  - 等待 UART0 串口所有 Log 数据都打印完毕（即 UART0 Tx FIFO 为空）
  - 将 P16、P17 两个引脚切换回 GPIO 功能，并关闭 P17 数字输入使能
  - 等待 UART1 串口所有数据（若有）都发送完毕（即 UART1 Tx FIFO 为空）
  - 将 P10、P07 两个引脚切换回 GPIO 功能，并使能 P07 引脚的中断，将其配置为下降沿触发（即使能 UART Rx 引脚接收数据唤醒芯片的功能）
  - 使能 P02 引脚的中断，并配置为下降沿触发（即使能 WKUP 按键唤醒功能）
- 本例程在 `vSocDeepSleepExitHook()` 函数中，实现了如下逻辑：
  - 将 P16、P17 两个引脚重新切换成 UART0 功能，并重新打开 P17 数字输入使能
  - 检测当前唤醒原因是否为 P07 中断（即 UART1 Rx 引脚接收到了足够时间的低电平），若不是则立刻将 P10、P07 两个芯片重新切换成 UART1 功能，并关闭 P07 GPIO 中断

    注：若是 P07 中断，则此处先不将 P10、P07 两个引脚切换为 UART1 功能，而是随后在 GPIO 中断服务程序中，先确认 P07 引脚检测到高电平之后，才将 PINMUX 切换为 UART1，这样可以确保 UART1 Rx 引脚接收到的第一个唤醒字节 0x00 完全接收完成，再切 PINMUX 收取 UART1 Rx 线上发送过来的实际数据

3.3 外设驱动例程

3.3.1 GPIO Push-Pull Output

1 功能概述

本例程演示如何使用 ADC HAL Driver 实现读取芯片的电压，芯片的温度，以及一个外部通道功能。

2 环境准备

- 硬件设备与线材：
  - PAN107X EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将 EVB 底板上的 LED3 排针与 P10 排针项链
  - 使用杜邦线将 JLink 仿真器的：
    * SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    * SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    * SWD_GND 引脚与 EVB 底板的 GND 排针相连

- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）

3 软件代码介绍

由于 ADC 模块硬件一份，通道却是多个，软件操作多通道的时候不可避免的出现竞争，为了避免冲突，pan_hal_adc 引入了信号量来解决这个问题。

```
HAL_ADC_GetValue(ADC_HandleTypeDef *pADC, void *value);
```

HAL_ADC_GetValue 函数**只能在线程执行**，不能在中断函数执行

4 编译和烧录

例 程 位 置: `<PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\adc_read_multiple_channels\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录。

5 例程演示说明

1. 烧录完成后，芯片会通过串口打印初始化 Log:

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 6
- Chip MAC Address  : E11000000FF8
- Chip UID          : 6D0001465454455354
- Chip Flash UID    : 4250315A3538380B005B7B4356037D78
- Chip Flash Size   : 512 KB
```

2. 观察 log，周期打印温度，电压，以及一个外部通道 1，如下图温度是 28.656721 ，电压是 3.295V，外部通道 CH1 是浮空状态

```
[15:50:00.934] 收 ←CH1 get

[15:50:00.977] 收 ←CH1 value->>371.938324

[15:50:01.934] 收 ←temperature get
battery start get

[15:50:02.036] 收 ←battery value->>3295 mv
temperature value->>28.656721
```

3.3.2 GPIO Input With Interrupt

1 功能概述

本例程演示如何使用 GPIO HAL Driver 实现中断方式的 GPIO 输入检测功能。

2 环境准备

- 硬件设备与线材：
  - PAN107X EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将 JLink 仿真器的：
    * SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    * SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    * SWD_GND 引脚与 EVB 底板的 GND 排针相连
- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）

3 编译和烧录

例程位置：`<PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\gpio_digital_input_interrupt\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录。

4 例程演示说明

1. 烧录完成后，芯片会通过串口打印初始化 Log：

```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 6
- Chip MAC Address   : E11000000FF8
- Chip UID           : 6D0001465454455354
- Chip Flash UID     : 4250315A3538380B005B7B4356037D78
- Chip Flash Size    : 512 KB
Wait for Task Notifications..
```

2. 按一下 EVB 底板的 KEY1 按键（对应 GPIO P06 引脚），芯片成功检测到 GPIO 下降沿中断后会打印按键按下的 Log：

```
==== KEY1 Pressed! ====
A notification received, value: 1.

Wait for Task Notifications..
```

3.3.3 GPIO Input Polling

1 功能概述

本例程演示如何使用 GPIO HAL Driver 实现查询方式的 GPIO 输入检测功能。

2 环境准备

- 硬件设备与线材:
  - PAN107X EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线:
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将 JLink 仿真器的:
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）

3 编译和烧录

例程位置: `<PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\gpio_digital_input_polling\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录。

4 例程演示说明

1. 烧录完成后，芯片会通过串口打印初始化 Log:

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 6
- Chip MAC Address  : E11000000FF8
- Chip UID          : 6D0001465454455354
- Chip Flash UID    : 4250315A3538380B005B7B4356037D78
- Chip Flash Size   : 512 KB
```

2. 按一下 EVB 底板的 KEY1 按键（对应 GPIO P06 引脚），芯片成功检测到 GPIO 电平改变后，会根据当前按键状态打印按键按下和释放的 Log:

```
KEY1 Pressed!
KEY1 Released!
```

3.3.4 GPIO Open-Drain Output

1 功能概述

本例程演示如何使用 GPIO HAL Driver 实现 GPIO 开漏（Open-Drain）输出功能。

2 环境准备

- 硬件设备与线材：
  - PAN107X EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将 EVB 底板上的 LED3 排针与 P10 排针项链
  - 使用杜邦线将 JLink 仿真器的：
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）

3 编译和烧录

例程位置：`<PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\gpio_output_open_drain\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录。

4 例程演示说明

1. 烧录完成后，芯片会通过串口打印初始化 Log：

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 6
- Chip MAC Address  : E11000000FF8
- Chip UID          : 6D0001465454455354
- Chip Flash UID    : 4250315A3538380B005B7B4356037D78
- Chip Flash Size   : 512 KB
```

2. 观察 EVB 底板上的发光二极管 LED3，可以看到其以 1Hz 的频率闪烁

注：此模式下灯光亮度较弱，这是因为 EVB 底板上的 LED3 电路被设计成高电平驱动亮灯，而由于 GPIO 开漏输出模式下配置了 SoC 内部上拉电阻（50KΩ 左右），因此驱动电流较小，从而导致灯光亮度较弱。

3.3.5 GPIO Push-Pull Output

1 功能概述

本例程演示如何使用 GPIO HAL Driver 实现 GPIO 推挽（Push-Pull）输出功能。

2 环境准备

- 硬件设备与线材：
  - PAN107X EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将 EVB 底板上的 LED3 排针与 P10 排针项链
  - 使用杜邦线将 JLink 仿真器的：
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
- PC 软件：
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）

3 编译和烧录

例程位置：`<PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\gpio_output_push_pull\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录。

4 例程演示说明

1. 烧录完成后，芯片会通过串口打印初始化 Log：

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 6
- Chip MAC Address  : E11000000FF8
- Chip UID          : 6D0001465454455354
- Chip Flash UID    : 4250315A3538380B005B7B4356037D78
- Chip Flash Size   : 512 KB
```

2. 观察 EVB 底板上的发光二极管 LED3，可以看到其以 1Hz 的频率闪烁

3.3.6 GPIO Simple Convenient APIs

1 功能概述

本例程演示 GPIO 底层 Driver 中提供的几个简单好用的接口，程序具体行为是通过 EVB 底板上的按键 KEY1，切换 LED3 的输出频率，并通过串口打印 Log 信息。

2 环境准备

- 硬件设备与线材：
  - PAN107X EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将 JLink 仿真器的：
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
- PC 软件：
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）

3 编译和烧录

例程位置：`<PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\gpio_simple_convenient_apis\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录。

4 例程演示说明

1. 烧录完成后，芯片会通过串口打印初始化 Log，观察 LED3 可见其以 1Hz 的频率闪烁：

```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 6
- Chip MAC Address   : E11000000FF8
- Chip UID           : 6D0001465454455354
- Chip Flash UID     : 4250315A3538380B005B7B4356037D78
- Chip Flash Size    : 512 KB
Toggle LED, dly_cnt=500
Toggle LED, dly_cnt=1000
```

(下页继续)

(续上页)

```
Toggle LED, dly_cnt=1500
Toggle LED, dly_cnt=2000
```

2. 按一下 EVB 底板的 KEY1 按键（对应 GPIO P06 引脚），芯片成功检测到按键按下后，会将 LED3 闪烁频率切换至 2Hz，同时打印如下 Log：

```
Toggle LED, dly_cnt=5000
Toggle LED, dly_cnt=5250
Toggle LED, dly_cnt=5500
Toggle LED, dly_cnt=5750
Toggle LED, dly_cnt=6000
Toggle LED, dly_cnt=6250
Toggle LED, dly_cnt=6500
Toggle LED, dly_cnt=6750
Toggle LED, dly_cnt=7000
```

3. 再按一下 EVB 底板的 KEY1 按键，芯片成功检测到按键按下后，会将 LED3 闪烁频率切换至 10Hz，同时打印如下 Log：

```
Toggle LED, dly_cnt=10500
Toggle LED, dly_cnt=10550
Toggle LED, dly_cnt=10600
Toggle LED, dly_cnt=10650
Toggle LED, dly_cnt=10700
Toggle LED, dly_cnt=10750
Toggle LED, dly_cnt=10800
Toggle LED, dly_cnt=10850
Toggle LED, dly_cnt=10900
Toggle LED, dly_cnt=10950
Toggle LED, dly_cnt=11000
```

3.3.7 I2C Receive Send Dma

1 功能概述

本例程演示如何使用 I2C HAL Driver 和 DMA HAL Driver 实现 dma 方式的 I2C 收发功能。

2 环境准备

- 硬件设备与线材：
  - PAN107X EVB **核心板**与**底板**各两块
  - JLink 仿真器（用于烧录例程程序）
  - USB-TypeC 线两条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB0 底板上的 P13(SDA)、P14(CLK) 分别接入 EVB1 对应相同 PAD 上
  - 使用杜邦线将 JLink 仿真器的：
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连

    * SWD_GND 引脚与 EVB 底板的 GND 排针相连

- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）

3 编译和烧录

例 程 位 置: `<PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\i2c_master_dma_receive\keil_107x`

`PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\i2c_slave_dma_send\keil_107x`

双击 NDK>\01_SDK\nimble\samples\peripheral\i2c_master_dma_receive\keil_107x 目录下 Keil Project 文件打开工程进行编译并烧录至 EVB0 板。

双击 NDK>\01_SDK\nimble\samples\peripheral\i2c_slave_dma_send\keil_107x 目录下 Keil Project 文件打开工程进行编译并烧录至 EVB1 板。

4 例程演示说明

1. 先烧录 i2c_slave_dma_send hex 至 EVB1，芯片会通过串口打印初始化 Log，i2c slave 进入发送模式，等待 i2c master 发送读命令并发送数据，发送完成后在中断回调函数中打印 slave DMA callback 及发送的数据:

```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 4
- Chip MAC Address   : D0000000037D
- Chip UID           : 7D0300DDF8375603E8
- Chip Flash UID     : 4250315632333917 00DDF8375603E878
- Chip Flash Size    : 512 KB

APP version: 255.255.65535
slave send start
slave send stop
slave DMA callback
Flag is I2C_CB_FLAG_DMA
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d␣
↪1e 1f
```

2. 烧录 i2c_master_dma_receive hex 至 EVB0，芯片会通过串口打印初始化 Log，i2c master 进入接收模式，发送读命令并接收 slave 发送的数据，在中断回调函数中打印接收到的数据:

```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 4
- Chip MAC Address   : D000000000D1
- Chip UID           : D10000F5F737560347
- Chip Flash UID     : 4250315632333917 00F5F73756034778
- Chip Flash Size    : 512 KB

APP version: 255.255.65535
master receive start
master receive stop
master DMA callback
Flag is I2C_CB_FLAG_DMA
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d␣
↪1e 1f
```

3.3.8 I2C Receive Send Interrupt

1 功能概述

本例程演示如何使用 I2C HAL Driver 实现中断方式的 I2C 收发功能。

2 环境准备

- 硬件设备与线材:
  - PAN107X EVB **核心板**与**底板**各两块
  - JLink 仿真器（用于烧录例程程序）
  - USB-TypeC 线两条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线:
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB0 底板上的 P13(SDA)、P14(CLK) 分别接入 EVB1 对应相同 PAD 上
  - 使用杜邦线将 JLink 仿真器的:
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）

3 编译和烧录

例程位置: `<PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\i2c_master_int_send\keil_107x`

`PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\i2c_slave_int_receive\keil_107x`

双击 NDK>\01_SDK\nimble\samples\peripheral\i2c_master_int_send\keil_107x 目录下 Keil Project 文件打开工程进行编译并烧录至 EVB0 板。

双击 NDK>\01_SDK\nimble\samples\peripheral\i2c_slave_int_receive\keil_107x 目录下 Keil Project 文件打开工程进行编译并烧录至 EVB1 板。

4 例程演示说明

1. 先烧录 i2c_slave_int_receive hex 至 EVB1，芯片会通过串口打印初始化 Log，i2c slave 进入接收模式，等待接收 32Byte 数据:

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 4
- Chip MAC Address  : D0000000037D
- Chip UID          : 7D0300DDF8375603E8
- Chip Flash UID    : 425031563233391700DDF8375603E878
- Chip Flash Size   : 512 KB
```

(下页继续)

(续上页)

```
APP version: 255.255.65535
slave recv start
slave recv stop
```

2. 烧录 i2c_master_int_send hex 至 EVB0，芯片会通过串口打印初始化 Log，i2c master 进入发送模式，发送 8bit 数据 0x00~0x1f，发送完成在中断回调中打印发送的数据：

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 4
- Chip MAC Address  : D000000000D1
- Chip UID          : D10000F5F737560347
- Chip Flash UID    : 4250315632333917 00F5F73756034778
- Chip Flash Size   : 512 KB

APP version: 255.255.65535
master send start
master send stop
tx
Send:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D␣
↪1E 1F
```

3. i2c slave 接收 32 Byte 数据并在中断回调中打印接收的数据，同时 i2c slave 转为发送模式，等待 master 发送读命令并发送数据，发送完成在中断回调中打印发送的数据：

```
rx
Receive
00 01 02 03 04 05 00 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D␣
↪1E 1F
slave send start
slave send stop:0

tx
Send:
00 01 02 03 04 05 00 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D␣
↪1E 1F
```

4. i2c master 发送 32 次读命令，同时接收 32 Byte 数据并在中断回调中打印接收的数据：

```
master receive start
master receive stop
rx
Receive
00 01 02 03 04 05 00 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D␣
↪1E 1F
```

3.3.9 I2C Receive Send Polling

1 功能概述

本例程演示如何使用 I2C HAL Driver 实现查询方式的 I2C 收发功能。

2 环境准备

- 硬件设备与线材：

  - PAN107X EVB **核心板**与**底板**各两块
  - JLink 仿真器（用于烧录例程程序）
  - USB-TypeC 线两条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB0 底板上的 P13(SDA)、P14(CLK) 分别接入 EVB1 对应相同 PAD 上
  - 使用杜邦线将 JLink 仿真器的：
    * SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    * SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    * SWD_GND 引脚与 EVB 底板的 GND 排针相连
- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）

3 编译和烧录

例程位置：`<PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\i2c_master_poll_send\keil_107x`

`PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\i2c_slave_poll_receive\keil_107x`

双击 NDK>\01_SDK\nimble\samples\peripheral\i2c_master_poll_send\keil_107x 目录下 Keil Project 文件打开工程进行编译并烧录至 EVB0 板。

双击 NDK>\01_SDK\nimble\samples\peripheral\i2c_slave_poll_receive\keil_107x 目录下 Keil Project 文件打开工程进行编译并烧录至 EVB1 板。

4 例程演示说明

1. 先烧录 i2c_slave_poll_receive hex 至 EVB1，芯片会通过串口打印初始化 Log，i2c slave 进入接收模式，等待接收 32Byte 数据：

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 4
- Chip MAC Address  : D0000000037D
- Chip UID          : 7D0300DDF8375603E8
- Chip Flash UID    : 425031563233391700DDF8375603E878
- Chip Flash Size   : 512 KB

APP version: 255.255.65535
start i2c slave poll start receive
```

2. 烧录 i2c_master_poll_send hex 至 EVB0，芯片会通过串口打印初始化 Log，i2c master 进入发送模式，发送 8bit 数据 0x00~0x1f，发送前打印 Log start i2c master poll start send，发送完成打印 Log start i2c master poll send over：

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 4
```

（下页继续）

(续上页)

```
- Chip MAC Address  : D000000000D1
- Chip UID          : D10000F5F737560347
- Chip Flash UID    : 425031563233391700F5F73756034778
- Chip Flash Size   : 512 KB

APP version: 255.255.65535
start i2c master poll start send
start i2c master poll send over
```

3. i2c slave 接收 32 Byte 数据并打印接收的数据：

```
start i2c slave poll received: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12␣
↪13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
```

3.3.10 PWM

1 功能概述

本例程演示如何使用 PWM HAL Driver 实现 PWM 输出的功能。

2 环境准备

- 硬件设备与线材：
  - PAN107X EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将 EVB 底板上的 LED3 排针与 P10 排针项链
  - 使用杜邦线将 JLink 仿真器的：
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
- PC 软件：
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）

3 编译和烧录

例程位置：`<PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\pwm\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录。

4 例程演示说明

1. 烧录完成后，芯片会通过串口打印初始化 Log：

```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 6
- Chip MAC Address   : E11000000FF8
- Chip UID           : 6D0001465454455354
- Chip Flash UID     : 4250315A3538380B005B7B4356037D78
- Chip Flash Size    : 512 KB

app started
APP version: 48.120.17921
```

2. 通过逻辑分析仪或者示波器观察 P03 引脚可以观察到一个 1KHZ，占空比 30% 的方波
3. 也可以把 P03 引脚通过杜邦线连接到 RGB 灯的 GPIO，观察一个颜色

3.3.11 SPI Receive Send Dma

1 功能概述

本例程演示如何使用 SPI HAL Driver 和 DMA HAL Driver 实现 dma 方式的 SPI 收发功能。

2 环境准备

- 硬件设备与线材：
  - PAN107X EVB **核心板**与**底板**各两块
  - JLink 仿真器（用于烧录例程程序）
  - USB-TypeC 线两条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB0 底板上的 P11(MOSI)、P03(CS)、P04(CLK)、P05(MISO) 分别接入 EVB1 对应相同 PAD 上
  - 使用杜邦线将 JLink 仿真器的：
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）

3 编译和烧录

例程位置: <PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\spi_master_dma_send_receive\keil_107x

NDK>\01_SDK\nimble\samples\peripheral\spi_slave_dma_receive_send\keil_107x

双击 NDK>\01_SDK\nimble\samples\peripheral\spi_master_dma_send_receive\keil_107x 目录下 Keil Project 文件打开工程进行编译并烧录至 EVB0 板。

双击 NDK>\01_SDK\nimble\samples\peripheral\spi_slave_dma_receive_send\keil_107x 目录下 Keil Project 文件打开工程进行编译并烧录至 EVB1 板。

4 例程演示说明

1. 先烧录 spi_slave_dma_receive_send hex 至 EVB1，芯片会通过串口打印初始化 Log，spi slave 进入接收模式，等待接收 160 个 16bit 数据:

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 4
- Chip MAC Address  : D0000000037D
- Chip UID          : 7D0300DDF8375603E8
- Chip Flash UID    : 425031563233391700DDF8375603E878
- Chip Flash Size   : 512 KB

APP version: 255.255.65535
slave start receive
```

2. 烧录 spi_master_dma_send_receive hex 至 EVB0，芯片会通过串口打印初始化 Log，spi master 进入发送模式，发送 16bit 数据 0x0000~0x009f，Log 打印发送的数据，发送完成后进入 spi master 接收模式，Log 打印 master turn to receive mode:

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 4
- Chip MAC Address  : D000000000D1
- Chip UID          : D10000F5F737560347
- Chip Flash UID    : 425031563233391700F5F73756034778
- Chip Flash Size   : 512 KB

APP version: 255.255.65535
master send data:
0x0000,0x0001,0x0002,0x0003,0x0004,0x0005,0x0006,0x0007,0x0008,0x0009,0x000a,0x000b,
↪0x000c,0x000d,0x000e,0x000f,
0x0010,0x0011,0x0012,0x0013,0x0014,0x0015,0x0016,0x0017,0x0018,0x0019,0x001a,0x001b,
↪0x001c,0x001d,0x001e,0x001f,
0x0020,0x0021,0x0022,0x0023,0x0024,0x0025,0x0026,0x0027,0x0028,0x0029,0x002a,0x002b,
↪0x002c,0x002d,0x002e,0x002f,
0x0030,0x0031,0x0032,0x0033,0x0034,0x0035,0x0036,0x0037,0x0038,0x0039,0x003a,0x003b,
↪0x003c,0x003d,0x003e,0x003f,
0x0040,0x0041,0x0042,0x0043,0x0044,0x0045,0x0046,0x0047,0x0048,0x0049,0x004a,0x004b,
↪0x004c,0x004d,0x004e,0x004f,
0x0050,0x0051,0x0052,0x0053,0x0054,0x0055,0x0056,0x0057,0x0058,0x0059,0x005a,0x005b,
↪0x005c,0x005d,0x005e,0x005f,
0x0060,0x0061,0x0062,0x0063,0x0064,0x0065,0x0066,0x0067,0x0068,0x0069,0x006a,0x006b,
↪0x006c,0x006d,0x006e,0x006f,
0x0070,0x0071,0x0072,0x0073,0x0074,0x0075,0x0076,0x0077,0x0078,0x0079,0x007a,0x007b,
↪0x007c,0x007d,0x007e,0x007f,
0x0080,0x0081,0x0082,0x0083,0x0084,0x0085,0x0086,0x0087,0x0088,0x0089,0x008a,0x008b,
↪0x008c,0x008d,0x008e,0x008f,
```

(下页继续)

(续上页)

```
0x0090,0x0091,0x0092,0x0093,0x0094,0x0095,0x0096,0x0097,0x0098,0x0099,0x009a,0x009b,
↪0x009c,0x009d,0x009e,0x009f,
master turn to receive mode
```

3. spi slave 接收 160 个 16bit 数据并打印接收的数据，同时 spi slave 进入发送模式，spi slave 发送 16bit 数据 0x0000~0x9f9f 并打印发送的数据：

```
slave receive data:
0x0000,0x0001,0x0002,0x0003,0x0004,0x0005,0x0006,0x0007,0x0008,0x0009,0x000a,0x000b,
↪0x000c,0x000d,0x000e,0x000f,
0x0010,0x0011,0x0012,0x0013,0x0014,0x0015,0x0016,0x0017,0x0018,0x0019,0x001a,0x001b,
↪0x001c,0x001d,0x001e,0x001f,
0x0020,0x0021,0x0022,0x0023,0x0024,0x0025,0x0026,0x0027,0x0028,0x0029,0x002a,0x002b,
↪0x002c,0x002d,0x002e,0x002f,
0x0030,0x0031,0x0032,0x0033,0x0034,0x0035,0x0036,0x0037,0x0038,0x0039,0x003a,0x003b,
↪0x003c,0x003d,0x003e,0x003f,
0x0040,0x0041,0x0042,0x0043,0x0044,0x0045,0x0046,0x0047,0x0048,0x0049,0x004a,0x004b,
↪0x004c,0x004d,0x004e,0x004f,
0x0050,0x0051,0x0052,0x0053,0x0054,0x0055,0x0056,0x0057,0x0058,0x0059,0x005a,0x005b,
↪0x005c,0x005d,0x005e,0x005f,
0x0060,0x0061,0x0062,0x0063,0x0064,0x0065,0x0066,0x0067,0x0068,0x0069,0x006a,0x006b,
↪0x006c,0x006d,0x006e,0x006f,
0x0070,0x0071,0x0072,0x0073,0x0074,0x0075,0x0076,0x0077,0x0078,0x0079,0x007a,0x007b,
↪0x007c,0x007d,0x007e,0x007f,
0x0080,0x0081,0x0082,0x0083,0x0084,0x0085,0x0086,0x0087,0x0088,0x0089,0x008a,0x008b,
↪0x008c,0x008d,0x008e,0x008f,
0x0090,0x0091,0x0092,0x0093,0x0094,0x0095,0x0096,0x0097,0x0098,0x0099,0x009a,0x009b,
↪0x009c,0x009d,0x009e,0x009f,

slave send data:
0x0000,0x0101,0x0202,0x0303,0x0404,0x0505,0x0606,0x0707,0x0808,0x0909,0x0a0a,0x0b0b,
↪0x0c0c,0x0d0d,0x0e0e,0x0f0f,
0x1010,0x1111,0x1212,0x1313,0x1414,0x1515,0x1616,0x1717,0x1818,0x1919,0x1a1a,0x1b1b,
↪0x1c1c,0x1d1d,0x1e1e,0x1f1f,
0x2020,0x2121,0x2222,0x2323,0x2424,0x2525,0x2626,0x2727,0x2828,0x2929,0x2a2a,0x2b2b,
↪0x2c2c,0x2d2d,0x2e2e,0x2f2f,
0x3030,0x3131,0x3232,0x3333,0x3434,0x3535,0x3636,0x3737,0x3838,0x3939,0x3a3a,0x3b3b,
↪0x3c3c,0x3d3d,0x3e3e,0x3f3f,
0x4040,0x4141,0x4242,0x4343,0x4444,0x4545,0x4646,0x4747,0x4848,0x4949,0x4a4a,0x4b4b,
↪0x4c4c,0x4d4d,0x4e4e,0x4f4f,
0x5050,0x5151,0x5252,0x5353,0x5454,0x5555,0x5656,0x5757,0x5858,0x5959,0x5a5a,0x5b5b,
↪0x5c5c,0x5d5d,0x5e5e,0x5f5f,
0x6060,0x6161,0x6262,0x6363,0x6464,0x6565,0x6666,0x6767,0x6868,0x6969,0x6a6a,0x6b6b,
↪0x6c6c,0x6d6d,0x6e6e,0x6f6f,
0x7070,0x7171,0x7272,0x7373,0x7474,0x7575,0x7676,0x7777,0x7878,0x7979,0x7a7a,0x7b7b,
↪0x7c7c,0x7d7d,0x7e7e,0x7f7f,
0x8080,0x8181,0x8282,0x8383,0x8484,0x8585,0x8686,0x8787,0x8888,0x8989,0x8a8a,0x8b8b,
↪0x8c8c,0x8d8d,0x8e8e,0x8f8f,
0x9090,0x9191,0x9292,0x9393,0x9494,0x9595,0x9696,0x9797,0x9898,0x9999,0x9a9a,0x9b9b,
↪0x9c9c,0x9d9d,0x9e9e,0x9f9f,
```

4. spi master 接收 160 个 16bit 数据并打印接收的数据：

```
master turn to receive mode
receive data:
0x0000,0x0101,0x0202,0x0303,0x0404,0x0505,0x0606,0x0707,0x0808,0x0909,0x0a0a,0x0b0b,
↪0x0c0c,0x0d0d,0x0e0e,0x0f0f,
0x1010,0x1111,0x1212,0x1313,0x1414,0x1515,0x1616,0x1717,0x1818,0x1919,0x1a1a,0x1b1b,
↪0x1c1c,0x1d1d,0x1e1e,0x1f1f,
0x2020,0x2121,0x2222,0x2323,0x2424,0x2525,0x2626,0x2727,0x2828,0x2929,0x2a2a,0x2b2b,
↪0x2c2c,0x2d2d,0x2e2e,0x2f2f,
```

(下页继续)

(续上页)

```
0x3030,0x3131,0x3232,0x3333,0x3434,0x3535,0x3636,0x3737,0x3838,0x3939,0x3a3a,0x3b3b,
↪0x3c3c,0x3d3d,0x3e3e,0x3f3f,
0x4040,0x4141,0x4242,0x4343,0x4444,0x4545,0x4646,0x4747,0x4848,0x4949,0x4a4a,0x4b4b,
↪0x4c4c,0x4d4d,0x4e4e,0x4f4f,
0x5050,0x5151,0x5252,0x5353,0x5454,0x5555,0x5656,0x5757,0x5858,0x5959,0x5a5a,0x5b5b,
↪0x5c5c,0x5d5d,0x5e5e,0x5f5f,
0x6060,0x6161,0x6262,0x6363,0x6464,0x6565,0x6666,0x6767,0x6868,0x6969,0x6a6a,0x6b6b,
↪0x6c6c,0x6d6d,0x6e6e,0x6f6f,
0x7070,0x7171,0x7272,0x7373,0x7474,0x7575,0x7676,0x7777,0x7878,0x7979,0x7a7a,0x7b7b,
↪0x7c7c,0x7d7d,0x7e7e,0x7f7f,
0x8080,0x8181,0x8282,0x8383,0x8484,0x8585,0x8686,0x8787,0x8888,0x8989,0x8a8a,0x8b8b,
↪0x8c8c,0x8d8d,0x8e8e,0x8f8f,
0x9090,0x9191,0x9292,0x9393,0x9494,0x9595,0x9696,0x9797,0x9898,0x9999,0x9a9a,0x9b9b,
↪0x9c9c,0x9d9d,0x9e9e,0x9f9f,
```

3.3.12 SPI Receive Send Interrupt

1 功能概述

本例程演示如何使用 SPI HAL Driver 实现中断方式的 SPI 收发功能。

2 环境准备

- 硬件设备与线材:
  - PAN107X EVB **核心板**与**底板**各两块
  - JLink 仿真器（用于烧录例程程序）
  - USB-TypeC 线两条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线:
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB0 底板上的 P11(MOSI)、P03(CS)、P04(CLK)、P05(MISO) 分别接入 EVB1 对应相同 PAD 上
  - 使用杜邦线将 JLink 仿真器的:
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）

3 编译和烧录

例程位置: `<PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\spi_master_int_send_receive\keil_107x`

`NDK>\01_SDK\nimble\samples\peripheral\spi_slave_int_receive_send\keil_107x`

双击 NDK>\01_SDK\nimble\samples\peripheral\spi_master_int_send_receive\keil_107x 目录下 Keil Project 文件打开工程进行编译并烧录至 EVB0 板。

双击 NDK>\01_SDK\nimble\samples\peripheral\spi_slave_int_receive_send\keil_107x 目录下 Keil Project 文件打开工程进行编译并烧录至 EVB1 板。

4 例程演示说明

1. 先烧录 spi_slave_int_receive_send hex 至 EVB1，芯片会通过串口打印初始化 Log，spi slave 进入接收模式，等待接收 160 个 16bit 数据：

```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 4
- Chip MAC Address   : D0000000037D
- Chip UID           : 7D0300DDF8375603E8
- Chip Flash UID     : 425031563233391700DDF8375603E878
- Chip Flash Size    : 512 KB

APP version: 255.255.65535
slave start receive
```

2. 烧录 spi_master_int_send_receive hex 至 EVB0，芯片会通过串口打印初始化 Log，spi master 进入发送模式，发送 16bit 数据 0x0000~0x009f，Log 打印发送的数据，发送完成后并进入 spi master 接收模式，Log 打印 master turn to receive mode：

```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 4
- Chip MAC Address   : D000000000D1
- Chip UID           : D10000F5F737560347
- Chip Flash UID     : 425031563233391700F5F73756034778
- Chip Flash Size    : 512 KB

APP version: 255.255.65535
master send data:
0x0000,0x0001,0x0002,0x0003,0x0004,0x0005,0x0006,0x0007,0x0008,0x0009,0x000a,0x000b,
↪0x000c,0x000d,0x000e,0x000f,
0x0010,0x0011,0x0012,0x0013,0x0014,0x0015,0x0016,0x0017,0x0018,0x0019,0x001a,0x001b,
↪0x001c,0x001d,0x001e,0x001f,
0x0020,0x0021,0x0022,0x0023,0x0024,0x0025,0x0026,0x0027,0x0028,0x0029,0x002a,0x002b,
↪0x002c,0x002d,0x002e,0x002f,
0x0030,0x0031,0x0032,0x0033,0x0034,0x0035,0x0036,0x0037,0x0038,0x0039,0x003a,0x003b,
↪0x003c,0x003d,0x003e,0x003f,
0x0040,0x0041,0x0042,0x0043,0x0044,0x0045,0x0046,0x0047,0x0048,0x0049,0x004a,0x004b,
↪0x004c,0x004d,0x004e,0x004f,
0x0050,0x0051,0x0052,0x0053,0x0054,0x0055,0x0056,0x0057,0x0058,0x0059,0x005a,0x005b,
↪0x005c,0x005d,0x005e,0x005f,
0x0060,0x0061,0x0062,0x0063,0x0064,0x0065,0x0066,0x0067,0x0068,0x0069,0x006a,0x006b,
↪0x006c,0x006d,0x006e,0x006f,
0x0070,0x0071,0x0072,0x0073,0x0074,0x0075,0x0076,0x0077,0x0078,0x0079,0x007a,0x007b,
↪0x007c,0x007d,0x007e,0x007f,
0x0080,0x0081,0x0082,0x0083,0x0084,0x0085,0x0086,0x0087,0x0088,0x0089,0x008a,0x008b,
↪0x008c,0x008d,0x008e,0x008f,
0x0090,0x0091,0x0092,0x0093,0x0094,0x0095,0x0096,0x0097,0x0098,0x0099,0x009a,0x009b,
↪0x009c,0x009d,0x009e,0x009f,
master turn to receive mode
```

3. spi slave 接收 160 个 16bit 数据并在中断 callback 函数中打印 slave receive done! receive 160 data as interrupt mode，同时 spi slave 进入发送模式，spi slave 发送 16bit 数据 0x0000~0x9f9f，发送

完成后打印接收的数据和发送的数据：

```
slave receive data:
0x0000,0x0001,0x0002,0x0003,0x0004,0x0005,0x0006,0x0007,0x0008,0x0009,0x000a,0x000b,
↪0x000c,0x000d,0x000e,0x000f,
0x0010,0x0011,0x0012,0x0013,0x0014,0x0015,0x0016,0x0017,0x0018,0x0019,0x001a,0x001b,
↪0x001c,0x001d,0x001e,0x001f,
0x0020,0x0021,0x0022,0x0023,0x0024,0x0025,0x0026,0x0027,0x0028,0x0029,0x002a,0x002b,
↪0x002c,0x002d,0x002e,0x002f,
0x0030,0x0031,0x0032,0x0033,0x0034,0x0035,0x0036,0x0037,0x0038,0x0039,0x003a,0x003b,
↪0x003c,0x003d,0x003e,0x003f,
0x0040,0x0041,0x0042,0x0043,0x0044,0x0045,0x0046,0x0047,0x0048,0x0049,0x004a,0x004b,
↪0x004c,0x004d,0x004e,0x004f,
0x0050,0x0051,0x0052,0x0053,0x0054,0x0055,0x0056,0x0057,0x0058,0x0059,0x005a,0x005b,
↪0x005c,0x005d,0x005e,0x005f,
0x0060,0x0061,0x0062,0x0063,0x0064,0x0065,0x0066,0x0067,0x0068,0x0069,0x006a,0x006b,
↪0x006c,0x006d,0x006e,0x006f,
0x0070,0x0071,0x0072,0x0073,0x0074,0x0075,0x0076,0x0077,0x0078,0x0079,0x007a,0x007b,
↪0x007c,0x007d,0x007e,0x007f,
0x0080,0x0081,0x0082,0x0083,0x0084,0x0085,0x0086,0x0087,0x0088,0x0089,0x008a,0x008b,
↪0x008c,0x008d,0x008e,0x008f,
0x0090,0x0091,0x0092,0x0093,0x0094,0x0095,0x0096,0x0097,0x0098,0x0099,0x009a,0x009b,
↪0x009c,0x009d,0x009e,0x009f,

slave send data:
0x0000,0x0101,0x0202,0x0303,0x0404,0x0505,0x0606,0x0707,0x0808,0x0909,0x0a0a,0x0b0b,
↪0x0c0c,0x0d0d,0x0e0e,0x0f0f,
0x1010,0x1111,0x1212,0x1313,0x1414,0x1515,0x1616,0x1717,0x1818,0x1919,0x1a1a,0x1b1b,
↪0x1c1c,0x1d1d,0x1e1e,0x1f1f,
0x2020,0x2121,0x2222,0x2323,0x2424,0x2525,0x2626,0x2727,0x2828,0x2929,0x2a2a,0x2b2b,
↪0x2c2c,0x2d2d,0x2e2e,0x2f2f,
0x3030,0x3131,0x3232,0x3333,0x3434,0x3535,0x3636,0x3737,0x3838,0x3939,0x3a3a,0x3b3b,
↪0x3c3c,0x3d3d,0x3e3e,0x3f3f,
0x4040,0x4141,0x4242,0x4343,0x4444,0x4545,0x4646,0x4747,0x4848,0x4949,0x4a4a,0x4b4b,
↪0x4c4c,0x4d4d,0x4e4e,0x4f4f,
0x5050,0x5151,0x5252,0x5353,0x5454,0x5555,0x5656,0x5757,0x5858,0x5959,0x5a5a,0x5b5b,
↪0x5c5c,0x5d5d,0x5e5e,0x5f5f,
0x6060,0x6161,0x6262,0x6363,0x6464,0x6565,0x6666,0x6767,0x6868,0x6969,0x6a6a,0x6b6b,
↪0x6c6c,0x6d6d,0x6e6e,0x6f6f,
0x7070,0x7171,0x7272,0x7373,0x7474,0x7575,0x7676,0x7777,0x7878,0x7979,0x7a7a,0x7b7b,
↪0x7c7c,0x7d7d,0x7e7e,0x7f7f,
0x8080,0x8181,0x8282,0x8383,0x8484,0x8585,0x8686,0x8787,0x8888,0x8989,0x8a8a,0x8b8b,
↪0x8c8c,0x8d8d,0x8e8e,0x8f8f,
0x9090,0x9191,0x9292,0x9393,0x9494,0x9595,0x9696,0x9797,0x9898,0x9999,0x9a9a,0x9b9b,
↪0x9c9c,0x9d9d,0x9e9e,0x9f9f,
```

4. spi master 接收 160 个 16bit 数据，在中断 callback 函数中打印 master receive done! receive 160 data as interrupt mode，同时打印接收的数据：

```
master receive done! receive 160 data as interrupt mode
receive data:
0x0000,0x0101,0x0202,0x0303,0x0404,0x0505,0x0606,0x0707,0x0808,0x0909,0x0a0a,0x0b0b,
↪0x0c0c,0x0d0d,0x0e0e,0x0f0f,
0x1010,0x1111,0x1212,0x1313,0x1414,0x1515,0x1616,0x1717,0x1818,0x1919,0x1a1a,0x1b1b,
↪0x1c1c,0x1d1d,0x1e1e,0x1f1f,
0x2020,0x2121,0x2222,0x2323,0x2424,0x2525,0x2626,0x2727,0x2828,0x2929,0x2a2a,0x2b2b,
↪0x2c2c,0x2d2d,0x2e2e,0x2f2f,
0x3030,0x3131,0x3232,0x3333,0x3434,0x3535,0x3636,0x3737,0x3838,0x3939,0x3a3a,0x3b3b,
↪0x3c3c,0x3d3d,0x3e3e,0x3f3f,
0x4040,0x4141,0x4242,0x4343,0x4444,0x4545,0x4646,0x4747,0x4848,0x4949,0x4a4a,0x4b4b,
↪0x4c4c,0x4d4d,0x4e4e,0x4f4f,
0x5050,0x5151,0x5252,0x5353,0x5454,0x5555,0x5656,0x5757,0x5858,0x5959,0x5a5a,0x5b5b,
↪0x5c5c,0x5d5d,0x5e5e,0x5f5f,
```

(下页继续)

(续上页)

```
0x6060,0x6161,0x6262,0x6363,0x6464,0x6565,0x6666,0x6767,0x6868,0x6969,0x6a6a,0x6b6b,
↪0x6c6c,0x6d6d,0x6e6e,0x6f6f,
0x7070,0x7171,0x7272,0x7373,0x7474,0x7575,0x7676,0x7777,0x7878,0x7979,0x7a7a,0x7b7b,
↪0x7c7c,0x7d7d,0x7e7e,0x7f7f,
0x8080,0x8181,0x8282,0x8383,0x8484,0x8585,0x8686,0x8787,0x8888,0x8989,0x8a8a,0x8b8b,
↪0x8c8c,0x8d8d,0x8e8e,0x8f8f,
0x9090,0x9191,0x9292,0x9393,0x9494,0x9595,0x9696,0x9797,0x9898,0x9999,0x9a9a,0x9b9b,
↪0x9c9c,0x9d9d,0x9e9e,0x9f9f,
```

3.3.13 SPI Receive Send Polling

1 功能概述

本例程演示如何使用 SPI HAL Driver 实现查询方式的 SPI 收发功能。

2 环境准备

- 硬件设备与线材:
  - PAN107X EVB **核心板**与**底板**各两块
  - JLink 仿真器（用于烧录例程程序）
  - USB-TypeC 线两条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线:
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB0 底板上的 P11(MOSI)、P03(CS)、P04(CLK)、P05(MISO) 分别接入 EVB1 对应相同 PAD 上
  - 使用杜邦线将 JLink 仿真器的:
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）

3 编译和烧录

例程位置: `<PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\spi_master_poll_send_receive\keil_107x`

`NDK>\01_SDK\nimble\samples\peripheral\spi_slave_poll_receive_send\keil_107x`

双击 NDK>\01_SDK\nimble\samples\peripheral\spi_master_poll_send_receive\keil_107x 目录下 Keil Project 文件打开工程进行编译并烧录至 EVB0 板。

双击 NDK>\01_SDK\nimble\samples\peripheral\spi_slave_poll_receive_send\keil_107x 目录下 Keil Project 文件打开工程进行编译并烧录至 EVB1 板。

4 例程演示说明

1. 先烧录 spi_slave_poll_receive_send hex 至 EVB1，芯片会通过串口打印初始化 Log，spi slave 进入接收模式，等待接收 160 个 16bit 数据：

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 4
- Chip MAC Address  : D0000000037D
- Chip UID          : 7D0300DDF8375603E8
- Chip Flash UID    : 425031563233391700DDF8375603E878
- Chip Flash Size   : 512 KB

APP version: 255.255.65535
slave start receive
```

2. 烧录 spi_master_poll_send_receive hex 至 EVB0，芯片会通过串口打印初始化 Log，spi master 进入发送模式，发送 16bit 数据 0x0000~0x009f，Log 打印发送的数据，发送完成后 Log 打印 master send done! 并进入 spi master 接收模式：

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 4
- Chip MAC Address  : D000000000D1
- Chip UID          : D10000F5F737560347
- Chip Flash UID    : 425031563233391700F5F73756034778
- Chip Flash Size   : 512 KB

APP version: 255.255.65535
master send data:
0x0000,0x0001,0x0002,0x0003,0x0004,0x0005,0x0006,0x0007,0x0008,0x0009,0x000a,0x000b,
↪0x000c,0x000d,0x000e,0x000f,
0x0010,0x0011,0x0012,0x0013,0x0014,0x0015,0x0016,0x0017,0x0018,0x0019,0x001a,0x001b,
↪0x001c,0x001d,0x001e,0x001f,
0x0020,0x0021,0x0022,0x0023,0x0024,0x0025,0x0026,0x0027,0x0028,0x0029,0x002a,0x002b,
↪0x002c,0x002d,0x002e,0x002f,
0x0030,0x0031,0x0032,0x0033,0x0034,0x0035,0x0036,0x0037,0x0038,0x0039,0x003a,0x003b,
↪0x003c,0x003d,0x003e,0x003f,
0x0040,0x0041,0x0042,0x0043,0x0044,0x0045,0x0046,0x0047,0x0048,0x0049,0x004a,0x004b,
↪0x004c,0x004d,0x004e,0x004f,
0x0050,0x0051,0x0052,0x0053,0x0054,0x0055,0x0056,0x0057,0x0058,0x0059,0x005a,0x005b,
↪0x005c,0x005d,0x005e,0x005f,
0x0060,0x0061,0x0062,0x0063,0x0064,0x0065,0x0066,0x0067,0x0068,0x0069,0x006a,0x006b,
↪0x006c,0x006d,0x006e,0x006f,
0x0070,0x0071,0x0072,0x0073,0x0074,0x0075,0x0076,0x0077,0x0078,0x0079,0x007a,0x007b,
↪0x007c,0x007d,0x007e,0x007f,
0x0080,0x0081,0x0082,0x0083,0x0084,0x0085,0x0086,0x0087,0x0088,0x0089,0x008a,0x008b,
↪0x008c,0x008d,0x008e,0x008f,
0x0090,0x0091,0x0092,0x0093,0x0094,0x0095,0x0096,0x0097,0x0098,0x0099,0x009a,0x009b,
↪0x009c,0x009d,0x009e,0x009f,
master send done!
master turn to receive mode
```

3. spi slave 接收 160 个 16bit 数据并打印接收的数据，同时 spi slave 进入发送模式，spi slave 发送 16bit 数据 0x0000~0x9f9f 并打印发送的数据：

```
slave receive data:
0x0000,0x0001,0x0002,0x0003,0x0004,0x0005,0x0006,0x0007,0x0008,0x0009,0x000a,0x000b,
↪0x000c,0x000d,0x000e,0x000f,
0x0010,0x0011,0x0012,0x0013,0x0014,0x0015,0x0016,0x0017,0x0018,0x0019,0x001a,0x001b,
↪0x001c,0x001d,0x001e,0x001f,
```

(下页继续)

(续上页)

```
0x0020,0x0021,0x0022,0x0023,0x0024,0x0025,0x0026,0x0027,0x0028,0x0029,0x002a,0x002b,
↪0x002c,0x002d,0x002e,0x002f,
0x0030,0x0031,0x0032,0x0033,0x0034,0x0035,0x0036,0x0037,0x0038,0x0039,0x003a,0x003b,
↪0x003c,0x003d,0x003e,0x003f,
0x0040,0x0041,0x0042,0x0043,0x0044,0x0045,0x0046,0x0047,0x0048,0x0049,0x004a,0x004b,
↪0x004c,0x004d,0x004e,0x004f,
0x0050,0x0051,0x0052,0x0053,0x0054,0x0055,0x0056,0x0057,0x0058,0x0059,0x005a,0x005b,
↪0x005c,0x005d,0x005e,0x005f,
0x0060,0x0061,0x0062,0x0063,0x0064,0x0065,0x0066,0x0067,0x0068,0x0069,0x006a,0x006b,
↪0x006c,0x006d,0x006e,0x006f,
0x0070,0x0071,0x0072,0x0073,0x0074,0x0075,0x0076,0x0077,0x0078,0x0079,0x007a,0x007b,
↪0x007c,0x007d,0x007e,0x007f,
0x0080,0x0081,0x0082,0x0083,0x0084,0x0085,0x0086,0x0087,0x0088,0x0089,0x008a,0x008b,
↪0x008c,0x008d,0x008e,0x008f,
0x0090,0x0091,0x0092,0x0093,0x0094,0x0095,0x0096,0x0097,0x0098,0x0099,0x009a,0x009b,
↪0x009c,0x009d,0x009e,0x009f,

slave send data:
0x0000,0x0101,0x0202,0x0303,0x0404,0x0505,0x0606,0x0707,0x0808,0x0909,0x0a0a,0x0b0b,
↪0x0c0c,0x0d0d,0x0e0e,0x0f0f,
0x1010,0x1111,0x1212,0x1313,0x1414,0x1515,0x1616,0x1717,0x1818,0x1919,0x1a1a,0x1b1b,
↪0x1c1c,0x1d1d,0x1e1e,0x1f1f,
0x2020,0x2121,0x2222,0x2323,0x2424,0x2525,0x2626,0x2727,0x2828,0x2929,0x2a2a,0x2b2b,
↪0x2c2c,0x2d2d,0x2e2e,0x2f2f,
0x3030,0x3131,0x3232,0x3333,0x3434,0x3535,0x3636,0x3737,0x3838,0x3939,0x3a3a,0x3b3b,
↪0x3c3c,0x3d3d,0x3e3e,0x3f3f,
0x4040,0x4141,0x4242,0x4343,0x4444,0x4545,0x4646,0x4747,0x4848,0x4949,0x4a4a,0x4b4b,
↪0x4c4c,0x4d4d,0x4e4e,0x4f4f,
0x5050,0x5151,0x5252,0x5353,0x5454,0x5555,0x5656,0x5757,0x5858,0x5959,0x5a5a,0x5b5b,
↪0x5c5c,0x5d5d,0x5e5e,0x5f5f,
0x6060,0x6161,0x6262,0x6363,0x6464,0x6565,0x6666,0x6767,0x6868,0x6969,0x6a6a,0x6b6b,
↪0x6c6c,0x6d6d,0x6e6e,0x6f6f,
0x7070,0x7171,0x7272,0x7373,0x7474,0x7575,0x7676,0x7777,0x7878,0x7979,0x7a7a,0x7b7b,
↪0x7c7c,0x7d7d,0x7e7e,0x7f7f,
0x8080,0x8181,0x8282,0x8383,0x8484,0x8585,0x8686,0x8787,0x8888,0x8989,0x8a8a,0x8b8b,
↪0x8c8c,0x8d8d,0x8e8e,0x8f8f,
0x9090,0x9191,0x9292,0x9393,0x9494,0x9595,0x9696,0x9797,0x9898,0x9999,0x9a9a,0x9b9b,
↪0x9c9c,0x9d9d,0x9e9e,0x9f9f,
```

4. spi master 接收 160 个 16bit 数据并打印接收的数据：

```
master turn to receive mode
receive data:
0x0000,0x0101,0x0202,0x0303,0x0404,0x0505,0x0606,0x0707,0x0808,0x0909,0x0a0a,0x0b0b,
↪0x0c0c,0x0d0d,0x0e0e,0x0f0f,
0x1010,0x1111,0x1212,0x1313,0x1414,0x1515,0x1616,0x1717,0x1818,0x1919,0x1a1a,0x1b1b,
↪0x1c1c,0x1d1d,0x1e1e,0x1f1f,
0x2020,0x2121,0x2222,0x2323,0x2424,0x2525,0x2626,0x2727,0x2828,0x2929,0x2a2a,0x2b2b,
↪0x2c2c,0x2d2d,0x2e2e,0x2f2f,
0x3030,0x3131,0x3232,0x3333,0x3434,0x3535,0x3636,0x3737,0x3838,0x3939,0x3a3a,0x3b3b,
↪0x3c3c,0x3d3d,0x3e3e,0x3f3f,
0x4040,0x4141,0x4242,0x4343,0x4444,0x4545,0x4646,0x4747,0x4848,0x4949,0x4a4a,0x4b4b,
↪0x4c4c,0x4d4d,0x4e4e,0x4f4f,
0x5050,0x5151,0x5252,0x5353,0x5454,0x5555,0x5656,0x5757,0x5858,0x5959,0x5a5a,0x5b5b,
↪0x5c5c,0x5d5d,0x5e5e,0x5f5f,
0x6060,0x6161,0x6262,0x6363,0x6464,0x6565,0x6666,0x6767,0x6868,0x6969,0x6a6a,0x6b6b,
↪0x6c6c,0x6d6d,0x6e6e,0x6f6f,
0x7070,0x7171,0x7272,0x7373,0x7474,0x7575,0x7676,0x7777,0x7878,0x7979,0x7a7a,0x7b7b,
↪0x7c7c,0x7d7d,0x7e7e,0x7f7f,
0x8080,0x8181,0x8282,0x8383,0x8484,0x8585,0x8686,0x8787,0x8888,0x8989,0x8a8a,0x8b8b,
↪0x8c8c,0x8d8d,0x8e8e,0x8f8f,
```

(下页继续)

(续上页)

```
0x9090,0x9191,0x9292,0x9393,0x9494,0x9595,0x9696,0x9797,0x9898,0x9999,0x9a9a,0x9b9b,
↪0x9c9c,0x9d9d,0x9e9e,0x9f9f,
```

3.3.14 TIMER BASIC

1 功能概述

本例程演示如何使用 TIMER HAL Driver 实现中 timer 的定时功能，

2 环境准备

- 硬件设备与线材：
  - PAN107 EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将 JLink 仿真器的：
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
- PC 软件：
  - 串口调试助手（Panchip Serial Assistant ），波特率 921600（用于接收串口打印 Log）

3 编译和烧录

例程位置：`<PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\timer_basic\keil_107x`
双击 Keil Project 文件打开工程进行编译烧录。

4 例程演示说明

1. 烧录完成后，芯片会通过串口打印初始化 Log：

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 4
- Chip MAC Address  : D0000000059D
- Chip UID          : 9D0500C2F737560338
- Chip Flash UID    : 425031563233391700C2F73756033878
- Chip Flash Size   : 512 KB
app started
APP version: 129.96.18288
```

2. timer0 工作状态下 1s 打印一次，timer1 每隔 5s 控制 timer0 开启和关闭：

```
[18:43:17.053] 收 ←tmr0 cnt0 1

[18:43:18.053] 收 ←tmr0 cnt0 2

[18:43:19.053] 收 ←tmr0 cnt0 3

[18:43:20.053] 收 ←tmr0 cnt0 4

[18:43:21.053] 收 ←tmr0 cnt0 5
tmr0 stop

[18:43:26.053] 收 ←tmr0 start

[18:43:27.053] 收 ←tmr0 cnt0 6

[18:43:28.053] 收 ←tmr0 cnt0 7

[18:43:29.054] 收 ←tmr0 cnt0 8

[18:43:30.053] 收 ←tmr0 cnt0 9

[18:43:31.053] 收 ←tmr0 cnt0 10
tmr0 stop
```

3.3.15 TIMER CAPTURE

1 功能概述

本例程演示如何使用 TIMER HAL Driver 实现中 timer 的定时捕获功能，外部输入 PWM 波形，timer 可以捕获 PWM 的高电平和低电平的时间。

2 环境准备

- 硬件设备与线材：
  - PAN107 EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将 EVB 底板上的 P15 接至 P22
  - 使用杜邦线将 JLink 仿真器的：
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连
- PC 软件:
  - 串口调试助手（Panchip Serial Assistant ），波特率 921600（用于接收串口打印 Log）

3 编译和烧录

例程位置：<PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\timer_basic\keil_107x

双击 Keil Project 文件打开工程进行编译烧录。

4 例程演示说明

1. 烧录完成后，芯片会通过串口打印初始化 Log：

```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 4
- Chip MAC Address   : D0000000059D
- Chip UID           : 9D0500C2F737560338
- Chip Flash UID     : 425031563233391700C2F73756033878
- Chip Flash Size    : 512 KB
app started
APP version: 129.96.18288
```

2. P22 输出 PWM 波形，频率时 20hz，占空比时 30%，代码如下所示：

```
HAL_PWM_PinConfiguration(P2, 2, PWM_CH0);

PWM_InitOpt pwm_init_obj  = {
    .frequency = 20,
    .dutyCycle = 30,
    .operateType = OPERATION_EDGE_ALIGNED,
    .lowPowerEn = DISABLE,
};
```

3. timer0 捕获到的低电平时间和高电平时间打印如下所示：

```
low level 34761us
high level 14998us
low level 34999us
high level 14998us
low level 34999us
high level 14998us
low level 34999us
high level 14998us
low level 34999us
high level 14998us
low level 34999us
high level 14998us
low level 34999us
high level 14998us
```

高电平时间时是 34999us，低电平时间是 14998us，这个和 PWM 输出的波形基本一样。

3.3.16 UART DMA

1 功能概述

本例程演示如何使用 UART HAL Driver 实现 DMA 方式的 UART 收发功能。

2 环境准备

- 硬件设备与线材：

  - PAN107 EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - 串口模块一块
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将串口模块上的 TX 引脚接至底板上的 P24，RX 引脚接至底板上的 P10
  - 使用杜邦线将 JLink 仿真器的：
    * SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    * SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    * SWD_GND 引脚与 EVB 底板的 GND 排针相连
- PC 软件:
  - 串口调试助手（Panchip Serial Assistant ），串口打印 Log 波特率 921600，串口通信波特率 115200。

3 编译和烧录

例程位置：`<PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\uart_dma\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录。

4 例程演示说明

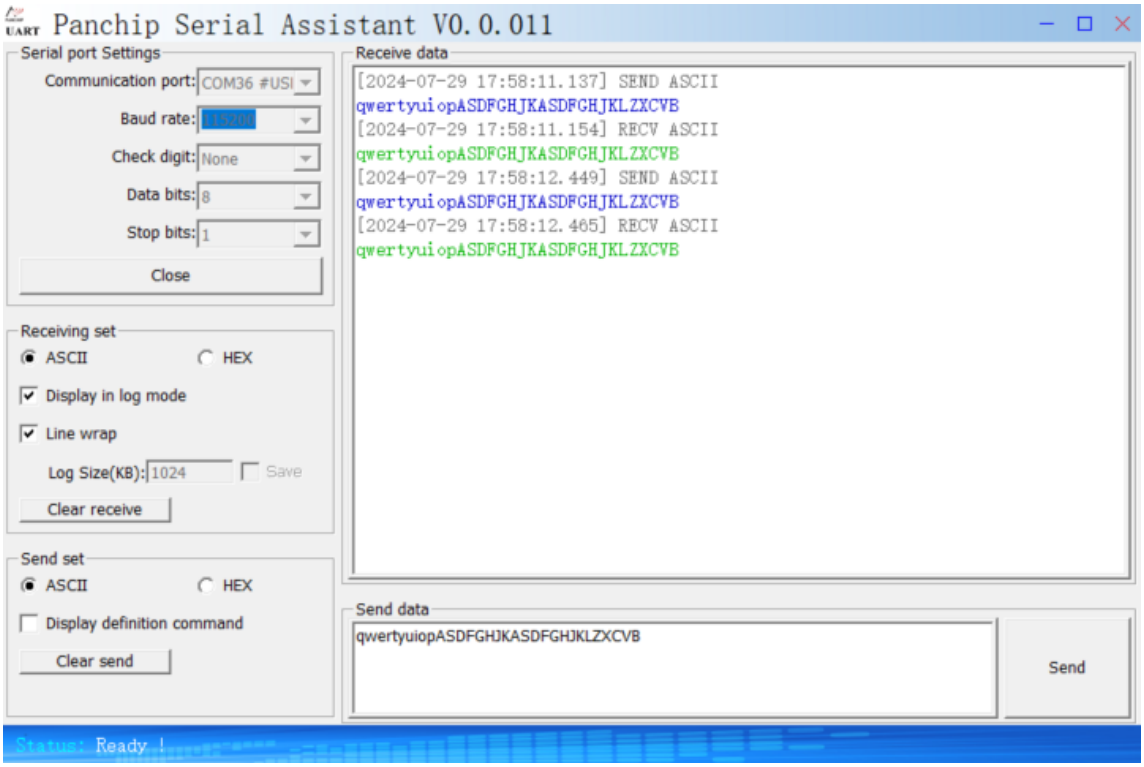
1. 烧录完成后，芯片会通过串口打印初始化 Log：

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 4
- Chip MAC Address  : D0000000059D
- Chip UID          : 9D0500C2F737560338
- Chip Flash UID    : 4250315632333917 00C2F73756033878
- Chip Flash Size   : 512 KB
app started
APP version: 129.96.18288
Recv BlockSize 128,Dma Ch0
Send BlockSize 128,Dma Ch1
rx done
tx complete
rx done
tx complete
rx done
tx complete
```

2. 打开另外一个串口调试助手（Panchip Serial Assistant ）：

   工程中测试一次 dma 传输的大小为 32 字节，当接收到 32 字节时，dma 中断才会触发。

   在 send 区域发送数据，观察 receive 区域收数据的情况，如下图所示



在 receive 区域可以看到接收和发送数据是一样的，芯片把收到的数据原封不动的发给上位机。

3.3.17 UART FIFO

1 功能概述

本例程演示如何使用 UART HAL Driver 实现中断方式的 UART 收发功能。

2 环境准备

- 硬件设备与线材：
  - PAN107 EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程程序）
  - 串口模块一块
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线：
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 使用杜邦线将串口模块上的 TX 引脚接至底板上的 P24，RX 引脚接至底板上的 P10
  - 使用杜邦线将 JLink 仿真器的：
    - SWD_CLK 引脚与 EVB 底板的 P00 排针相连
    - SWD_DAT 引脚与 EVB 底板的 P01 排针相连
    - SWD_GND 引脚与 EVB 底板的 GND 排针相连

- PC 软件:
  - 串口调试助手（Panchip Serial Assistant ），串口打印 Log 波特率 921600，串口通信波特率 115200。

3 编译和烧录

例程位置：`<PAN10XX-NDK>\01_SDK\nimble\samples\peripheral\uart_fifo\keil_107x`

双击 Keil Project 文件打开工程进行编译烧录。

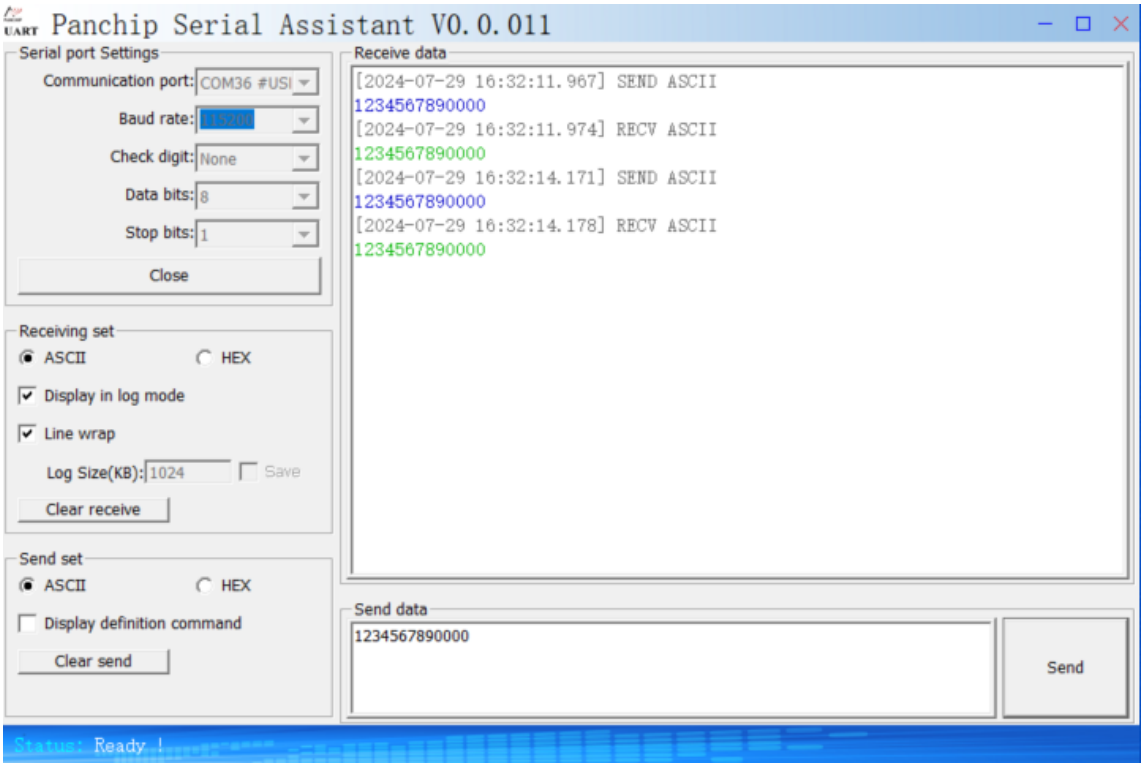
4 例程演示说明

1. 烧录完成后，芯片会通过串口打印初始化 Log:

```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 4
- Chip MAC Address   : D0000000059D
- Chip UID           : 9D0500C2F737560338
- Chip Flash UID     : 425031563233391700C2F73756033878
- Chip Flash Size    : 512 KB
app started
APP version: 129.96.18288
tx complete
```

2. 打开另外一个串口调试助手（Panchip Serial Assistant ）:

   在 send 区域发送数据，观察 receive 区域收数据的情况，如下图所示



   在 receive 区域可以看到接收和发送数据是一样的，芯片把收到的数据原封不动的发给上位机。

3.4 固件保护例程

3.4.1 Firmware Encryption

1 功能概述

本例程演示芯片通过固件加密、硬件解密的机制保护 Flash 关键代码的方法。

重要: 本功能需要用到 PanLink 量产烧录工具，为正常演示本例程，请确保您的 PanLink 上位机工具版本不低于 `v0.0.005`。

警告: 本功能需要通过 PanLink 工具烧录芯片 eFuse 的特定地址，eFuse 的物理特性是同一地址只可烧录一次，无法还原，因此**当某颗芯片使能加密功能后，其将只能正常运行加密后的固件，无法再正常运行普通的明文固件**!

2 环境准备

- 硬件设备与线材:
  - PAN107X EVB **核心板**与**底板**各一块
  - JLink 仿真器（用于烧录例程**明文程序**）
  - PanLink 量产烧录工具（用于烧录固件加密信息至芯片 eFuse，以及烧录例程**加密程序**至芯片 Flash）
  - USB-TypeC 线一条（用于底板供电和查看串口打印 Log）
  - 杜邦线数根或跳线帽数个（用于连接各个硬件设备）
- 硬件接线:
  - 将 EVB 核心板插到底板上
  - 使用 USB-TypeC 线，将 PC USB 插口与 EVB 底板 USB->UART 插口相连
  - 使用杜邦线将 EVB 底板上的 TX 引脚接至核心板 P16，RX 引脚接至核心板 P17
  - 根据情况将 PanLink 或 Jlink 仿真器连接至芯片
- PC 软件:
  - 串口调试助手（UartAssist）或终端工具（SecureCRT），波特率 921600（用于接收串口打印 Log）
  - PanLink 上位机工具

3 编译和烧录

例程位置: `<PAN10XX-NDK>\01_SDK\nimble\samples\security\fw_encryption\keil_107x`

双击 Keil Project 文件打开工程，编译成功后，使用 Keil - Flash Download 按钮，**向未使能加密功能的芯片中烧录明文程序**。

> **注**: 本例程编译过程中会调用 `nimble\scripts\encrypt_tool.py` 脚本，此脚本中使用了名为 Crypto 的库，因此需要提前安装此库，否则可能会编译不过。Windows 下安装此库的方法是:
>
> 1. 直接在 CMD 命令行中执行 `pip install pycryptodome` 安装 Crypto 库
> 2. 重新编译例程，检查是否能编译通过

3. 若仍无法正常编译，请进入 python 安装目录的第三方库子目录 `<python-folder>/Lib/site-packages`，查看此目录下是否有名为 `Crypto` 的子目录，并确认其首字母 `C` 为大写，如果不是，则手动将其改为大写
4. 再次编译例程即可

4 例程演示说明

1. 通过 Keil 将明文程序烧录至芯片后，可以看到芯片通过串口打印如下 Log:

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 6
- Chip MAC Address  : E11000001012
- Chip UID          : 250001465454455354
- Chip Flash UID    : 4250315A3538380B005B474356033C78
- Chip Flash Size   : 512 KB

Hello World!

Secret calcucation result: 0x4604b510
```

2. 断开 JLink 与芯片的连接，然后将 PanLink 连接至芯片（可能需要将芯片从 EVB 底板取下），打开 PanLink 上位机工具，使能“加密信息配置”，并同时选择载入以下两个文件:
   1. 本例程生成的加密配置信息文件（`Images\encrypt_info_enc.bin`）
   2. 本例程生成的**未加密**固件（`Images\ndk_app.bin` 或 `Images\ndk_app.hex`）

   具体操作步骤请参考 PanLink 上位机工具文档。
3. 点击“下载”按钮，即可将本工程的加密配置信息（包括加密使能开关、防注入保护使能开关、加密 Flash 区域配置、加密秘钥等）和明文固件同时烧录到芯片 eFuse 中:
4. eFuse 烧录成功后，复位芯片，可以看到刚才可以正常执行的程序，此时出现了 Hardfault

```
Try to load HW calibration data.. DONE.
- Chip Info         : 0x1
- Chip CP Version   : 255
- Chip FT Version   : 6
- Chip MAC Address  : E11000001012
- Chip UID          : 250001465454455354
- Chip Flash UID    : 4250315A3538380B005B474356033C78
- Chip Flash Size   : 512 KB

Hello World!

In Hard Fault Handler
r0  = 0x20001cf0
r1  = 0x40003000
r2  = 0x1
r3  = 0x1941
r12 = 0x0
lr  = 0x1947
pc  = 0x10a
psr = 0x61000000
```

   这是因为芯片此时已经开启了加密使能，当程序执行到 eFuse 中配置的 Flash 加密区域后，会使用秘钥先将程序解密后运行，而此时由于 Flash 中的程序是明文程序，因此经过解密操作后反而会出错。

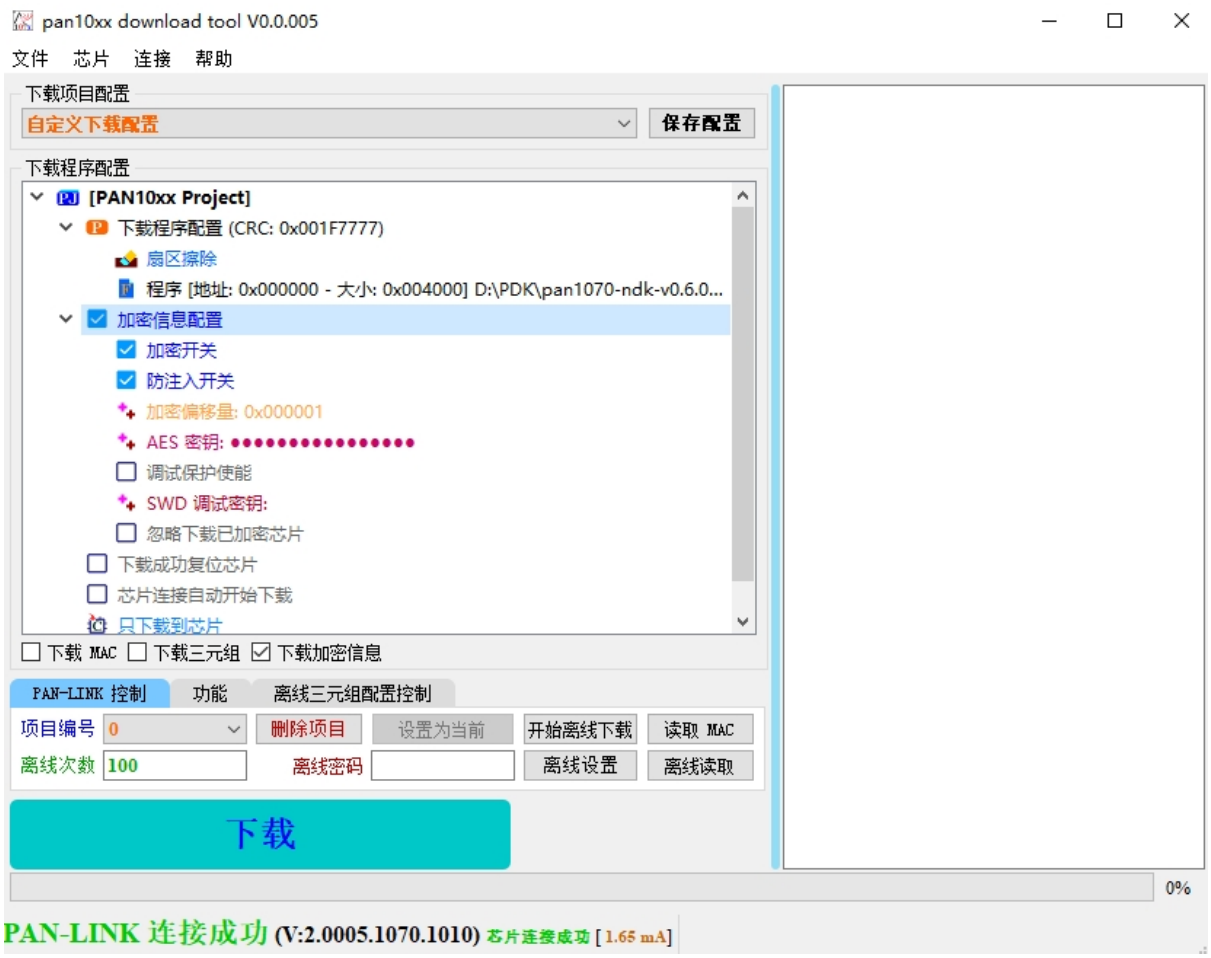


图 41: PanLink 载入加密配置信息文件和未加密固件完成

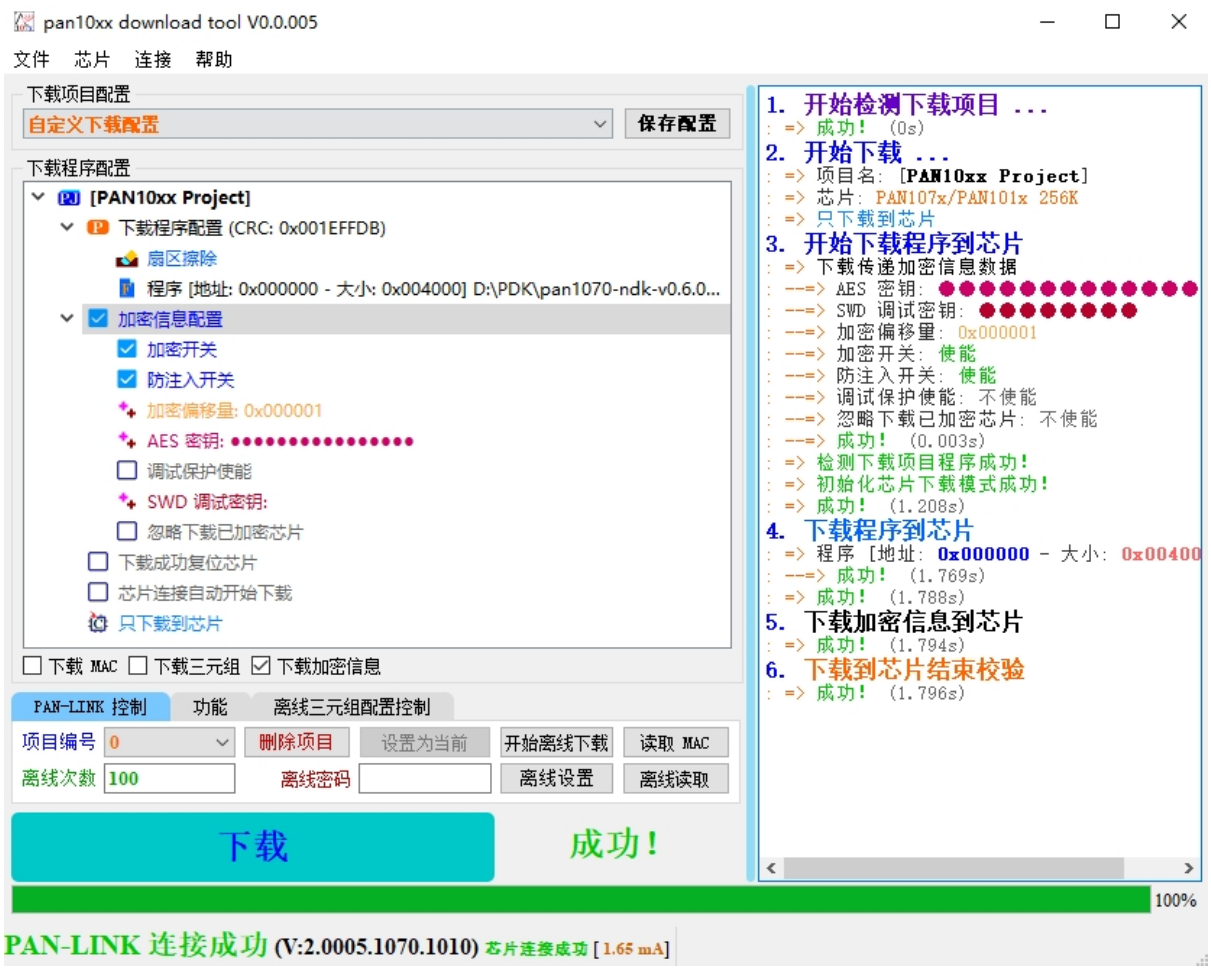


图 42: PanLink 烧录加密配置信息文件和未加密固件成功

5. 重新配置 Panlink，此时我们**取消勾选**“加密信息配置”，并且在“下载程序配置”中，指定烧录的 Image 为 `Images\ndk_app_enc.bin` 或 `Images\ndk_app_enc.hex` 文件，然后点击“下载”按钮，将加密后的 Image 烧录至芯片：

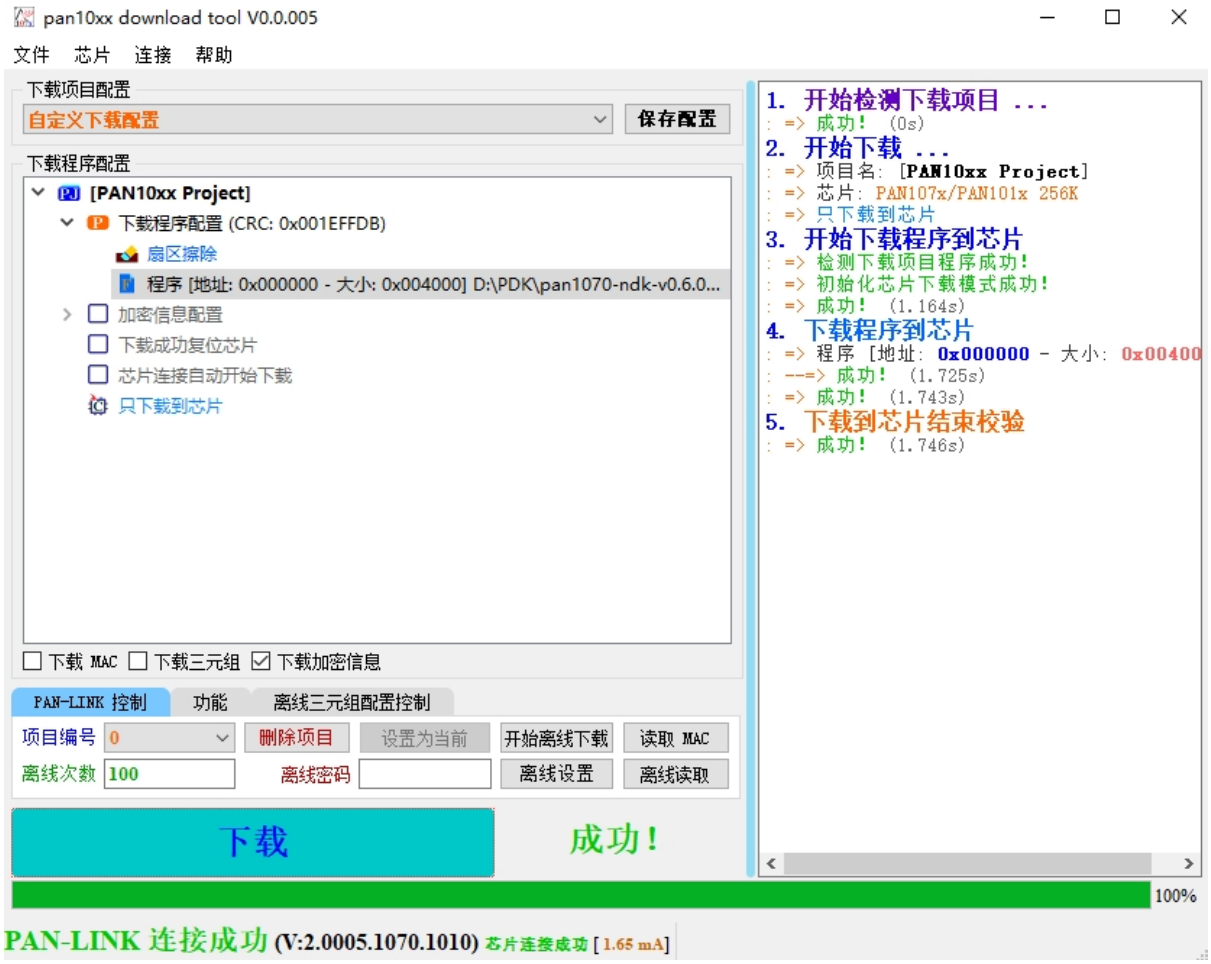


图 43: PanLink 烧录加密固件

6. 再次复位芯片，可以看到此时程序又可以正常执行：

```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 6
- Chip MAC Address   : E11000001012
- Chip UID           : 250001465454455354
- Chip Flash UID     : 4250315A3538380B005B474356033C78
- Chip Flash Size    : 512 KB

Hello World!

Secret calcucation result: 0x4604b510
```

5 开发者说明

5.1 工程配置 本工程相比与其他的普通工程（如蓝牙例程），在配置上有如下改动：

1. 修改了 Memory Config 配置（`configuration\image_map_config.h`）：
2. 修改了工程的分散加载文件（`project.sct`）：

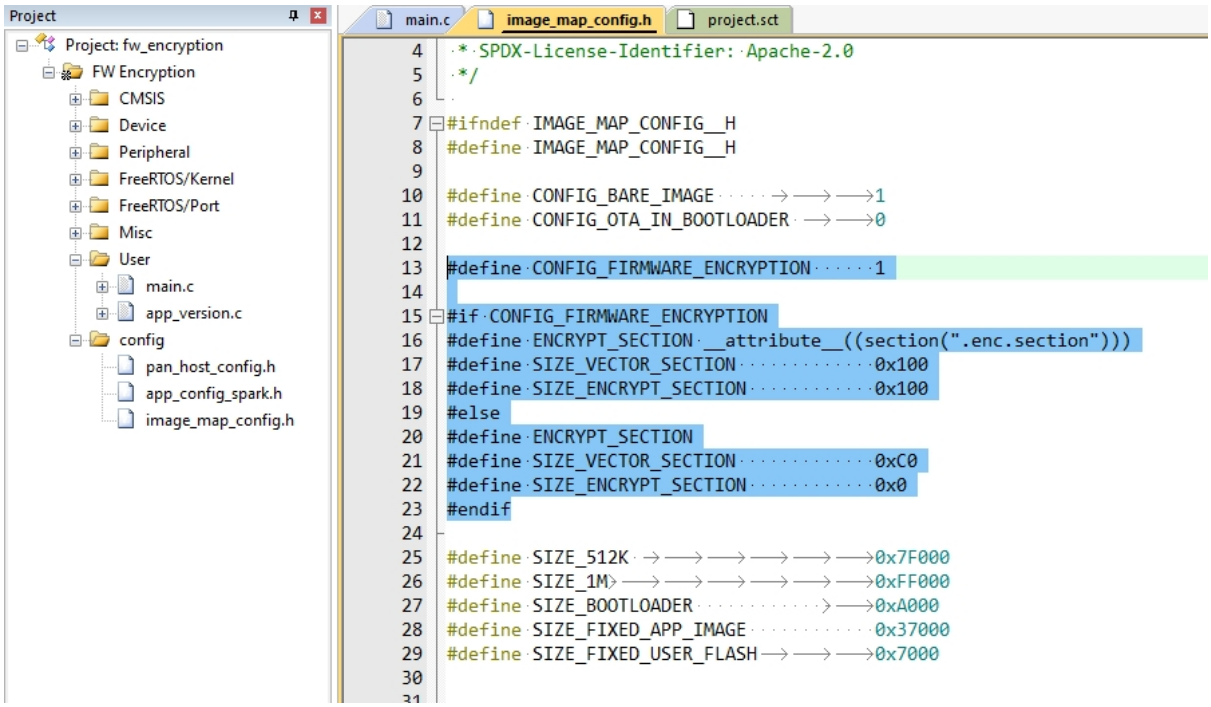


图 44: Memory Config File

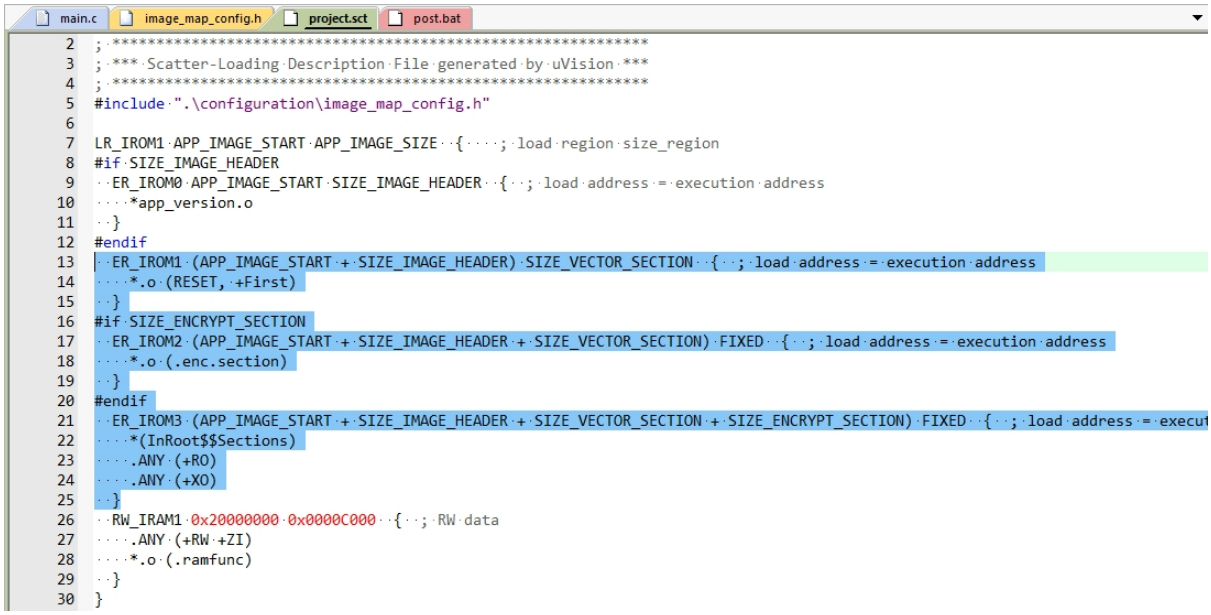


图 45: Project Scatter File

3. 修改了工程的 post.bat 脚本：

main.c | image_map_config.h | project.sct | post.bat

```
%5 --bin --output %1\ndk_app.bin %3
%5 --text -a -c --output %1\ndk_app.asm %3

copy %4 %1\ndk_app.hex
IF %errorlevel% NEQ 0 (
  @ECHO Error occurred, errno is %errorlevel%, exit.
  exit /b %errorlevel%
)

python ..\..\..\..\scripts\encode_yaml.py -i .\ -o %1
IF %errorlevel% NEQ 0 (
  @ECHO Error occurred in encode_yaml.py, errno is %errorlevel%, exit.
  exit /b %errorlevel%
)

python ..\..\..\..\scripts\encrypt_tool.py %1\ndk_app.hex .\encrypt_info.yaml
IF %errorlevel% NEQ 0 (
  @ECHO Error occurred in encrypt_tool.py, errno is %errorlevel%, exit.
  exit /b %errorlevel%
)

python ..\..\..\..\scripts\signed_image.py %1\ndk_app_enc.bin %1\ndk_app_enc.signed.bin
IF %errorlevel% NEQ 0 (
  @ECHO Error occurred in signed_image.py, errno is %errorlevel%, exit.
  exit /b %errorlevel%
)

echo Post Build Complete!
```

图 46: Post Build Script File

4. 修改了工程的 After Build 命令：
5. 新增了加密信息配置文件（encrypt_info.yaml）：

   其中各个 item 解释如下：

   - `secure_enable: true`：使能固件加密功能
   - `anti_injection_enable: true`：使能防注入保护功能，与固件加密功能结合使用，作用是防止通过 SWD Debug 的方式获取到加密 Flash 区域的明文信息
   - `encrypt_flash_offset: 0x1`：配置加密 Flash 的第几个 Page（大小 256 字节），如 0x1 表示加密 Flash 的第 1 个 Page，对应 Flash 绝对地址为 0x100
   - `encrypt_key: '4c68384139f574d836bcf34e9dfb01bf'`：配置加密秘钥（AES-128）
   - `debug_protect_enable: false`：不使能 SWD Debug Protect 功能
   - `debug_key: ''`：不配置 SWD Debug Protect 秘钥
   - `expected_start_addr: 0x0`：配置当前 Image 固件的起始地址为 0x0，用于 encrypt tool 交叉验证输入 Image 的地址是否有效

警告:

1. 本文件中包含明文的加密秘钥（encrypt_key），一定要妥善保管，不可随意外传，以防密钥泄露！
2. 在编译过程中，Keil 的 After Build 流程会基于本文件生成一个二次加密后的 `encrypt_info_enc.bin` 文件，此文件中会对秘钥添加扰码，因此在 PanLink 载入加密信息的过程中建议使用此文件，而不是明文的 yaml 文件。

5.2 程序代码

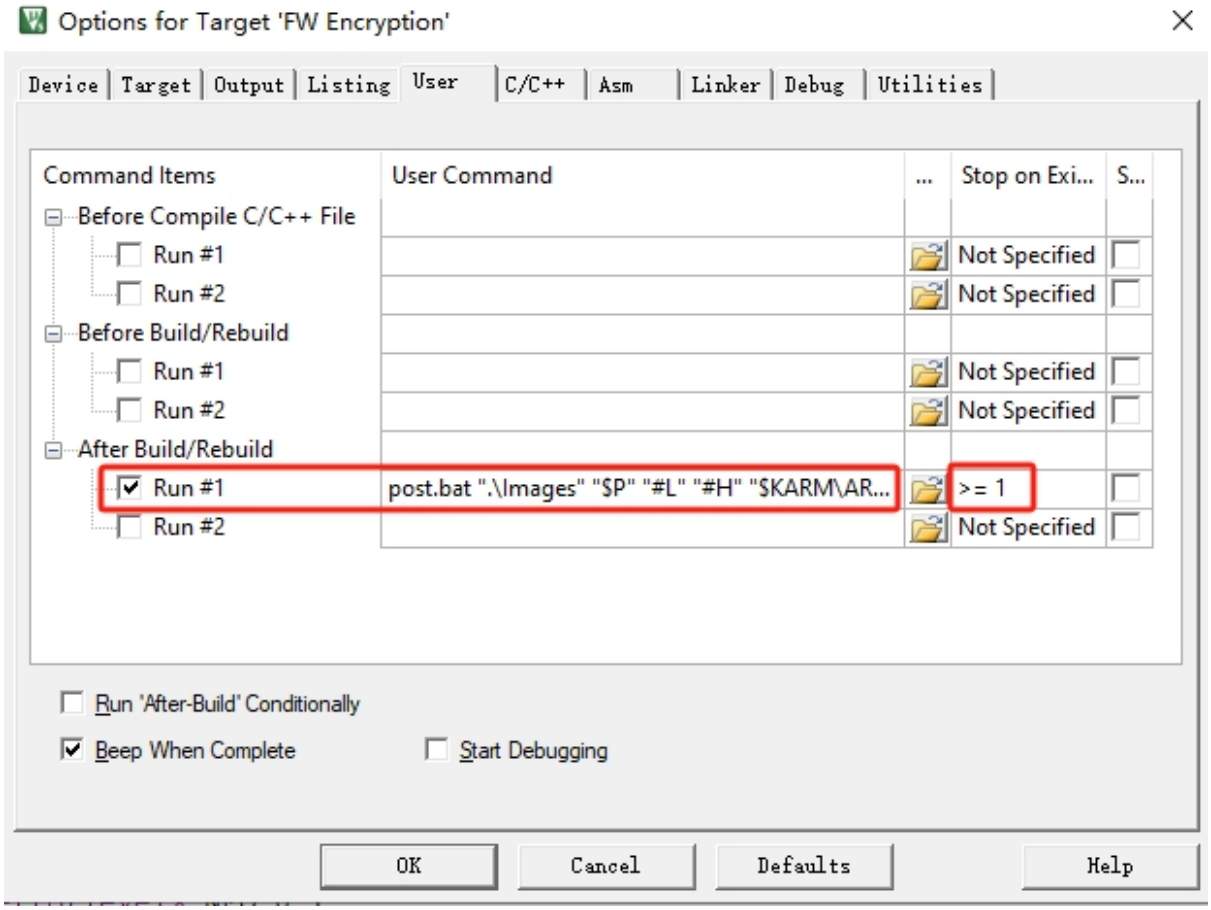


图 47: Keil After Build Command

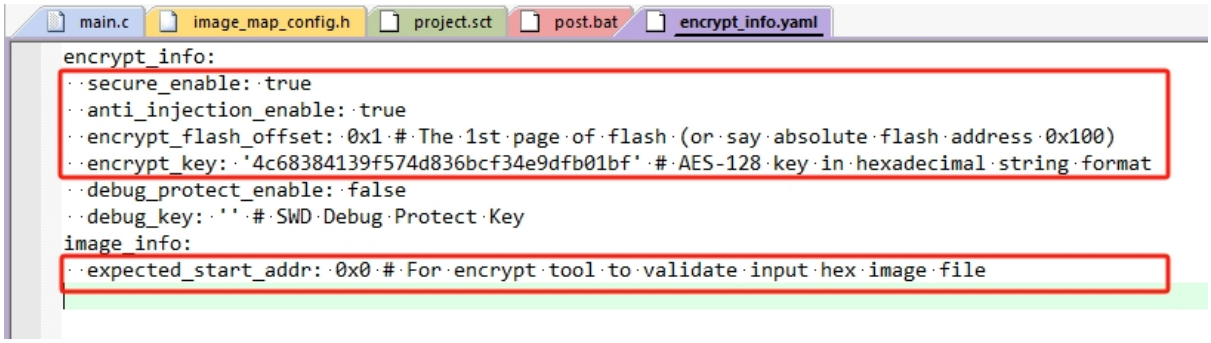


图 48: Encrypt Info File

5.2.1 **主程序** 主程序 app_main() 函数内容如下：

```
void app_main(void)
{
#if (!CONFIG_BARE_IMAGE)
    print_version_info();
#endif

    uint32_t secret_calc;

    printf("\nHello World!\n\n");
    encrypted_test_function(&secret_calc);
    printf("Secret calcucation result: 0x%x\n", secret_calc);
}
```

1. 若当前工程配置为非裸机程序（即支持 Bootloader 的程序），则打印 App 版本信息
2. 向串口打印 “Hello World” 字符串，用于指示主程序执行成功
3. 执行 Flash 加密区域的函数 `encrypted_test_function()`
4. 向串口打印上述加密函数返回的 `secret_calc` 变量值

5.2.2 **加密程序** 编译到 Flash 加密区域的函数内容如下：

```
ENCRYPT_SECTION void encrypted_test_function(uint32_t *calculation)
{
    /* Do some secret operations/calculations here */
    *calculation = *(uint32_t *)((uint32_t)(&encrypted_test_function) & 0xFFFFFFFE);
    printf("Hello from %s!\n\n", __func__);
}
```

1. 使用前缀 `ENCRYPT_SECTION` 以将函数编译到 Flash 加密区域
2. 获取当前函数的首地址，并从首地址所在位置取出一个 word 数据，将其通过 `calculation` 指针传到上层
3. 向串口打印字符串

   芯片支持加密的 Flash 大小为一个 Page（256 字节），因此在实际项目中我们应当选取一个重要且简短的函数作为加密函数段。

3.5 解决方案

3.5.1 Solution: BLE Accelerometer

1 功能概述

本文主要介绍 PAN10xx BLE Accelerometer 和手机 APP 进行连接，通过 APP 上报 Accelerometer 实时坐标并修改上报时间间隔, 此功能支持 `pan107x` 芯片

2 环境要求

- board: `pan107x evb`
- uart (option): 显示串口 log
- NRF Connect/BLE 调试助手 APP

3 编译和烧录

pan107x 芯片例程位置：<home>\nimble\samples\solutions\ble_accelerometer\keil_107x

使用 keil 进行打开项目进行编译烧录。

4 演示说明

1. PAN107 EVB 板 GPIO P07、P10 与 g-sensor 电路用跳线帽连接。
2. EVB 板上电灯的颜色默认是蓝色，BLE 广播设备的名字是” b+acc sensor”。
3. 打开安卓手机” NRF Connect “app，在 app 上启动搜索设备。
4. 搜索到后点击连接，连接成功后就可以控制 g-sensor 的定时上报及显示实时坐标。

5 设备连接和控制

5.1 广播数据

Adv Data Type	Description	Length	Detail
0xff	Device id	10byte	0xD1, 0x07, 0xc9, 0x7a, 0xbb, 0x8f, 0xdd, 0x4b, 0x00, 0x11
0x07	128-bit UUID	16byte	0x9e, 0xca, 0xdc, 0x24, 0x0e, 0xe5, 0xa9, 0xe0,0x93, 0xf3, 0xa3, 0xb5, 0x01, 0x20, 0x40, 0x6e
0x09	Device name	n byte	“b+acc sensor”

5.2 GATT 服务

Function	Service Attribute	UUID(128bit)
Useless	Primary service	0xE9, 0x5D, 0x07, 0x53, 0x25, 0x1D, 0x47, 0x0A,0xA0, 0x62, 0xFA, 0x19, 0x22, 0xDF, 0xA9, 0xA8
Notify g-sensor 的坐标	Notify characteristic declaration	0xE9, 0x5D, 0xCA, 0x4B, 0x25, 0x1D, 0x47, 0x0A,0xA0, 0x62, 0xFA, 0x19, 0x22, 0xDF, 0xA9, 0xA8
控制 g-sensor 的周期	Write characteristic declaration	0xE9, 0x5D, 0xFB, 0x24, 0x25, 0x1D, 0x47, 0x0A,0xA0, 0x62, 0xFA, 0x19, 0x22, 0xDF, 0xA9, 0xA8

5.3 通信协议

5.3.1 Period Control UUID = {0xE9, 0x5D, 0xFB, 0x24, 0x25, 0x1D, 0x47, 0x0A,0xA0, 0x62, 0xFA, 0x19, 0x22, 0xDF, 0xA9, 0xA8}

Period(ms)	Length	Detail
500	1byte	0x01
1000	1byte	0x02
1500	1byte	0x03
2000	1byte	0x04

控制 Accelerometer 上报周期时间。

5.3.2 Notify Accelerometer Coordinate UUID = {0xE9, 0x5D, 0xCA, 0x4B, 0x25, 0x1D, 0x47, 0x0A,0xA0, 0x62, 0xFA, 0x19, 0x22, 0xDF, 0xA9, 0xA8}

每次收到控制命令后将 Accelerometer 的实时坐标通知给手机 app。

6 RAM/Flash 资源使用情况

PAN107x:

```
Flash Size: 155.32k
RAM Size:  34.29 k
```

3.5.2 BLE APP UART

1 功能概述

此项目演示蓝牙从机串口透传功能，从机设备和手机或主机设备连接后可以和串口模块进行数据透传。

2 环境要求

- board: `pan107x evb` 或 `pan101x evb`
- uart log: 打印工程的 log
- uart ble: 和蓝牙透传数据
- 手机 app `nrf connect`

3 编译和烧录

例程位置:

- pan107x: `<home>\nimble\samples\solutions\ble_app_uart\keil_107x`
- pan101x: `<home>\nimble\samples\solutions\ble_app_uart\keil_101x`

使用 `keil` 进行打开项目进行编译烧录。

4 环境准备

uart 接线说明

- pan107x:

UART LOG（波特率 921600）	PIN
TX	P16
RX	P17

UART BLE（波特率 115200）	PIN
TX	P10
RX	P24

- pan101x:

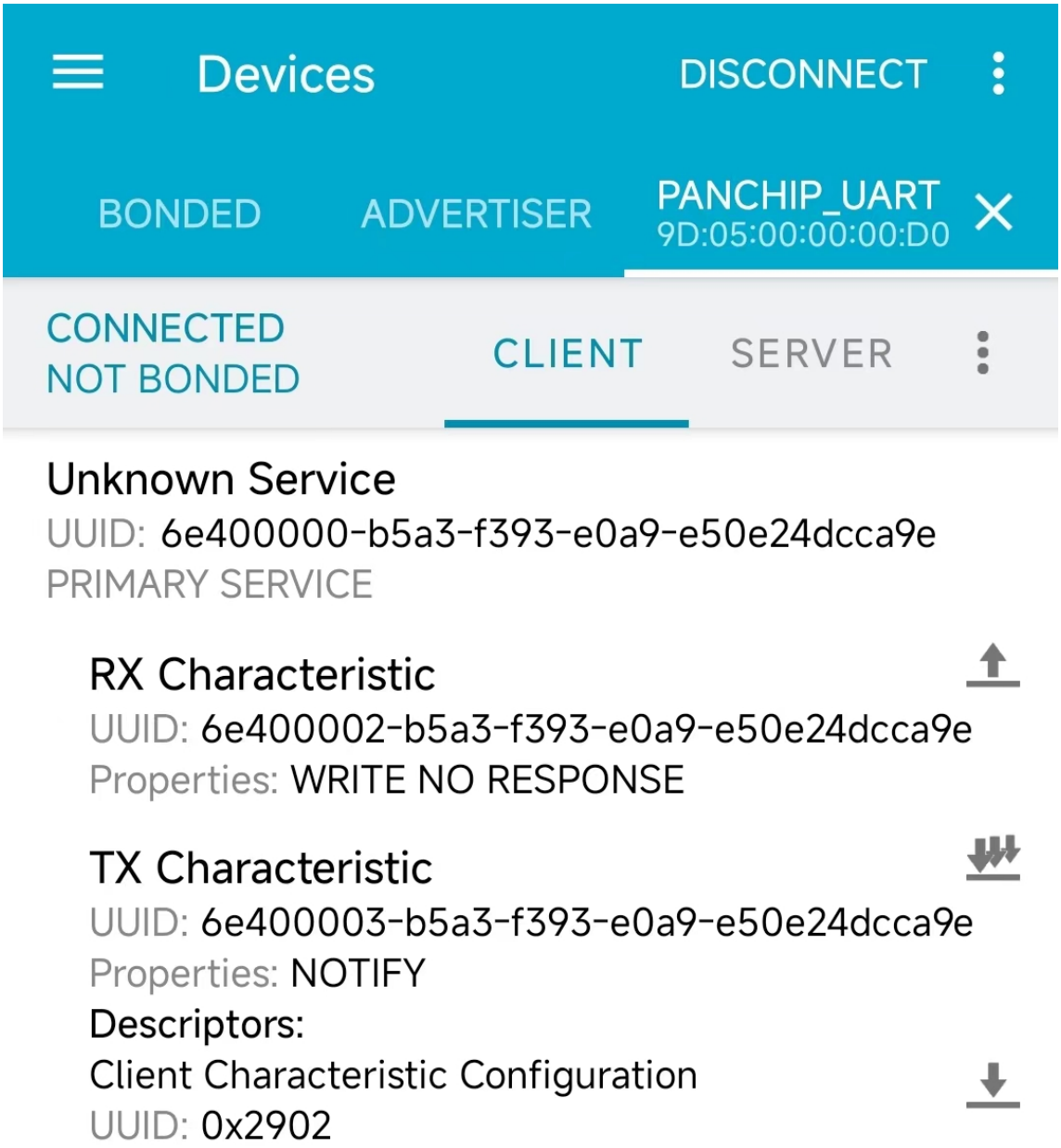
UART LOG（波特率 921600）	PIN
TX	P11
RX	P12

UART BLE（波特率 115200）	PIN
TX	P00
RX	P01

5 演示说明

1. 打开 nrf connect App，扫描广播名称为“panchip_uart”设备，然后连接。
2. 连上后的 log 及 app 界面如下：

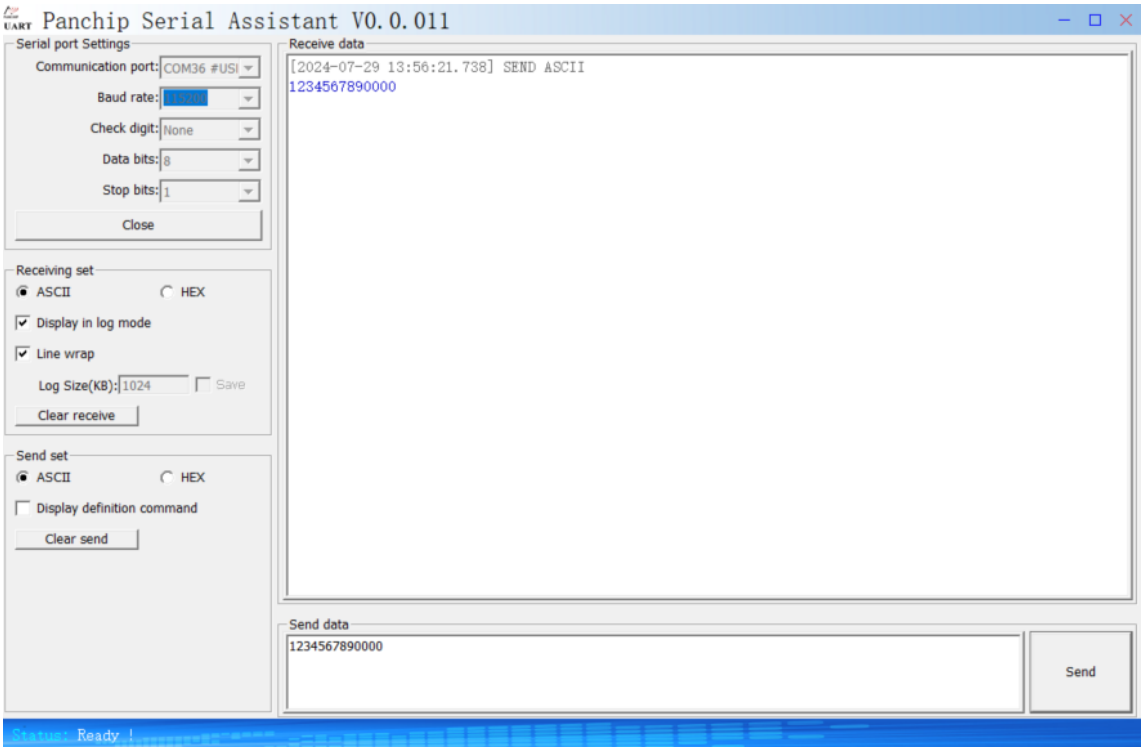
```
Try to load HW calibration data.. DONE.
- Chip Info          : 0x1
- Chip CP Version    : 255
- Chip FT Version    : 4
- Chip MAC Address   : D0000000059D
- Chip UID           : 9D0500C2F737560338
- Chip Flash UID     : 4250315632333917 00C2F73756033878
- Chip Flash Size    : 512 KB
LL Spark Controller Version:d7c4bfa
app started
APP version: 129.96.18288
ble_store_config_num_our_secs:1,1
ble_store_config_num_peer_secs:1, 1
ble_store_config_num_cccds:3, 1
tx complete
Device Address: d0 00 00 00 05 9d

connection established; status=0
```



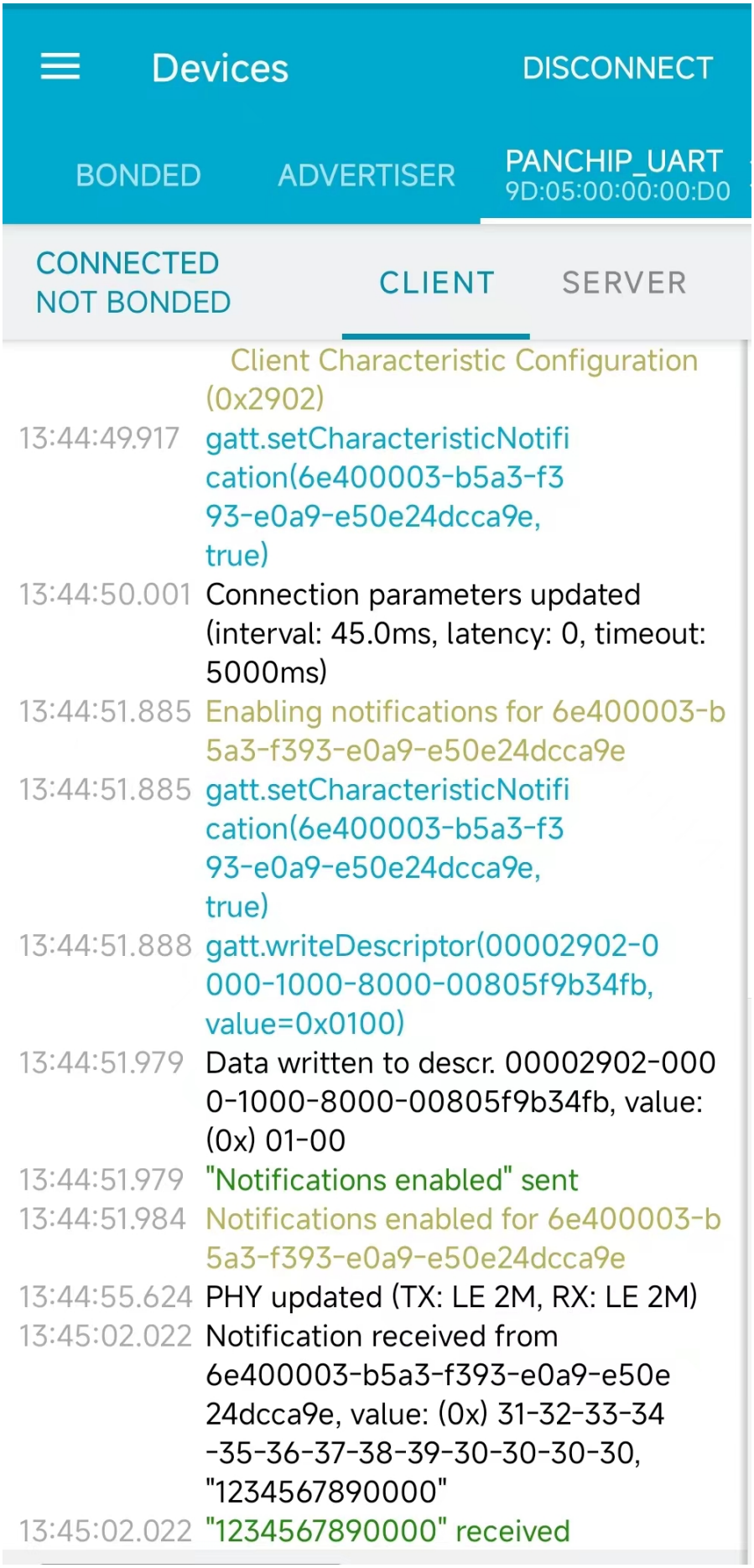
RX 服务用于向串口发送数据，TX 服务用于接收串口的数据，需要使能 notify。

1. 用串口工具发数据，app 收数据的情况如下：

   串口工具：

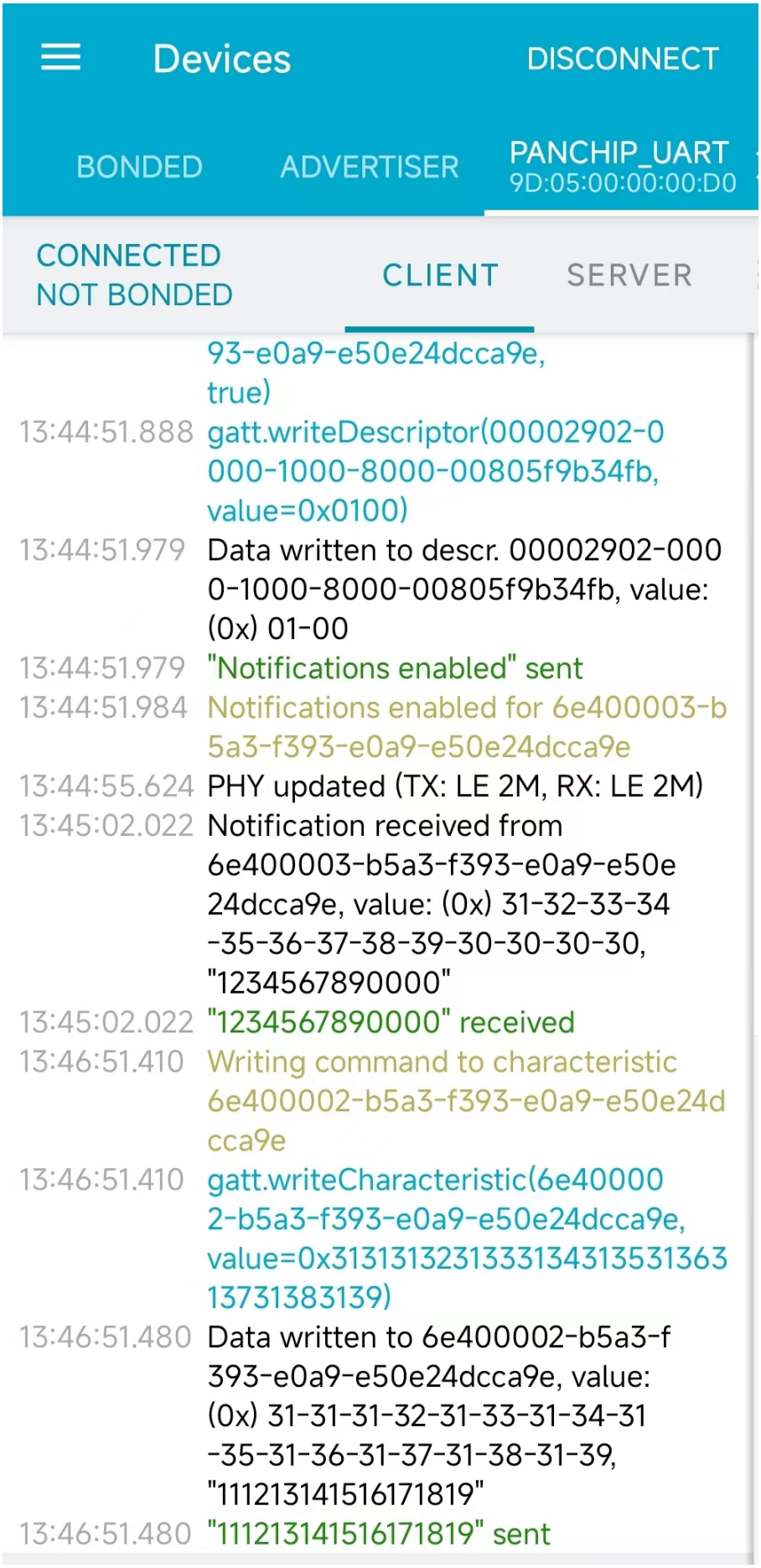

Panchip Serial Assistant V0.0.011
Serial port Settings
Communication port:
Baud rate:
Check digit:
None
Data bits:
8
Stop bits:
1
Close
Receiving set
ASCII
HEX
Display in log mode
Line wrap
Log Size(KB):
1024
Save
Clear receive
Send set
ASCII
HEX
Display definition command
Clear send
Receive data
[2024-07-29 13:56:21.738] SEND ASCII
1234567890000
Send data
1234567890000
Send
Status: Ready !

app：

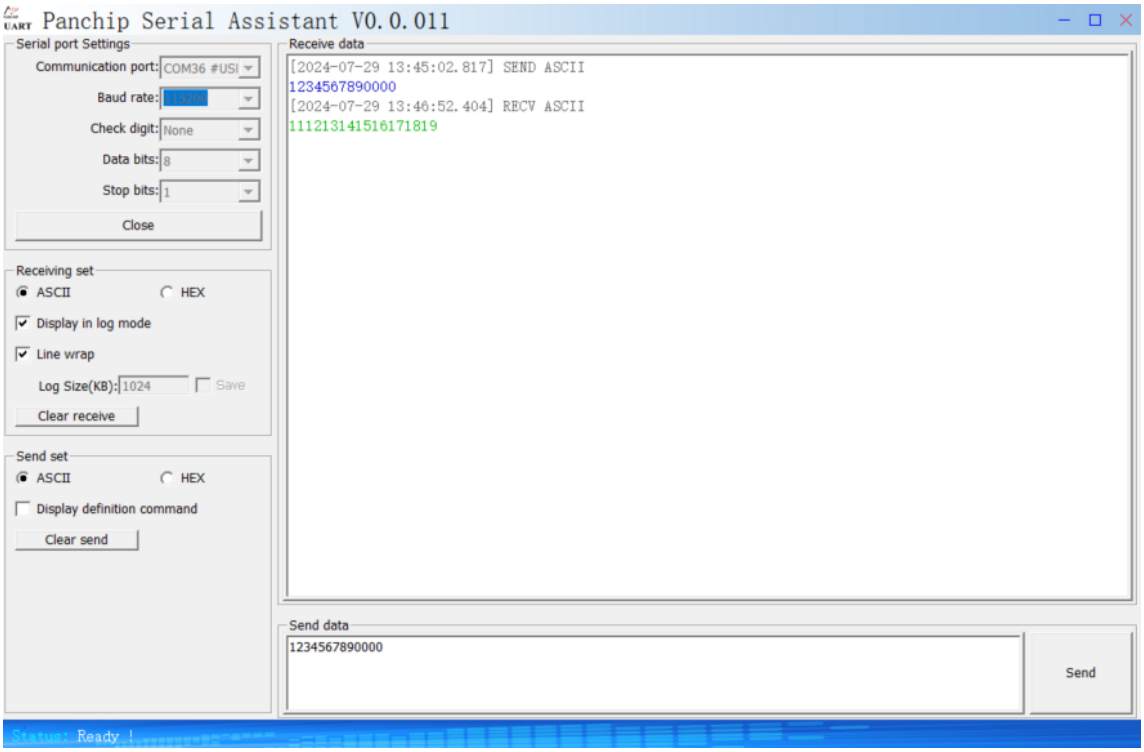

Devices
DISCONNECT
BONDED
ADVERTISER
PANCHIP_UART
9D:05:00:00:00:D0
CONNECTED
NOT BONDED
CLIENT
SERVER
Client Characteristic Configuration (0x2902)
13:44:49.917 gatt.setCharacteristicNotification(6e400003-b5a3-f393-e0a9-e50e24dcca9e, true)
13:44:50.001 Connection parameters updated (interval: 45.0ms, latency: 0, timeout: 5000ms)
13:44:51.885 Enabling notifications for 6e400003-b5a3-f393-e0a9-e50e24dcca9e
13:44:51.885 gatt.setCharacteristicNotification(6e400003-b5a3-f393-e0a9-e50e24dcca9e, true)
13:44:51.888 gatt.writeDescriptor(00002902-0000-1000-8000-00805f9b34fb, value=0x0100)
13:44:51.979 Data written to descr. 00002902-0000-1000-8000-00805f9b34fb, value: (0x) 01-00
13:44:51.979 "Notifications enabled" sent
13:44:51.984 Notifications enabled for 6e400003-b5a3-f393-e0a9-e50e24dcca9e
13:44:55.624 PHY updated (TX: LE 2M, RX: LE 2M)
13:45:02.022 Notification received from 6e400003-b5a3-f393-e0a9-e50e24dcca9e, value: (0x) 31-32-33-34-35-36-37-38-39-30-30-30-30, "1234567890000"
13:45:02.022 "1234567890000" received

2. 用 app 发数据，串口工具收数据的情况如下：

   app:


Devices
DISCONNECT
BONDED
ADVERTISER
PANCHIP_UART
9D:05:00:00:00:D0
CONNECTED
NOT BONDED
CLIENT
SERVER
93-e0a9-e50e24dcca9e,
true)
13:44:51.888 gatt.writeDescriptor(00002902-0
000-1000-8000-00805f9b34fb,
value=0x0100)
13:44:51.979 Data written to descr. 00002902-000
0-1000-8000-00805f9b34fb, value:
(0x) 01-00
13:44:51.979 "Notifications enabled" sent
13:44:51.984 Notifications enabled for 6e400003-b
5a3-f393-e0a9-e50e24dcca9e
13:44:55.624 PHY updated (TX: LE 2M, RX: LE 2M)
13:45:02.022 Notification received from
6e400003-b5a3-f393-e0a9-e50e
24dcca9e, value: (0x) 31-32-33-34
-35-36-37-38-39-30-30-30-30,
"1234567890000"
13:45:02.022 "1234567890000" received
13:46:51.410 Writing command to characteristic
6e400002-b5a3-f393-e0a9-e50e24d
cca9e
13:46:51.410 gatt.writeCharacteristic(6e40000
2-b5a3-f393-e0a9-e50e24dcca9e,
value=0x313131323133313431353136313731383139)
13:46:51.480 Data written to 6e400002-b5a3-f
393-e0a9-e50e24dcca9e, value:
(0x) 31-31-31-32-31-33-31-34-31
-35-31-36-31-37-31-38-31-39,
"1112131415161718 19"
13:46:51.480 "1112131415161718 19" sent

串口工具：



5 RAM/Flash 资源使用情况

PAN107x:

```
Flash Size: 151.77k
RAM Size:  38.89 k
```

PAN101x:

```
Flash Size: 138.38k
RAM Size:  14.76 k
```

3.5.3 BLE HID Selfie

1 功能概述

此项目演示基于 HID 服务的自拍服务，用 K1 和 K2 模拟的音量键，我们可以在相机模式下可以实现拍照功能，pan101x 芯片在功耗大小和执行速度方面的弱于 `pan107x` 芯片。`pan101x` 芯片暂时不支持 K1 和 K2 拟的音量键

2 环境要求

- board: `pan107x evb` 或 `pan101x evb`
- uart(option): 用来显示串口 log（波特率 921600，选项 `8n1`）
- 手机 app `nrf connect`

3 编译和烧录

pan107x 芯片例程位置：<home>\nimble\samples\bluetooth\bleprph_hr\keil_107x

pan101x 芯片例程位置：<home>\nimble\samples\bluetooth\bleprph_hr\keil_101x

使用 keil 进行打开项目进行编译烧录。

4 演示说明

1. 在设置蓝牙界面连接 nimble_hid 进行配对，通过按键 KEY1 和 KEY2 模拟音量增大和减小，同时利用音量增大键和减小键实现拍照快捷键功能。

```
[19:23:23.691]  Try to load HW calibration data.. DONE.
- Chip Type         : 0x80
- Chip CP Version   : None
- Chip FT Version   : 8
- Chip MAC Address  : D0000C0CBBF5
- Chip Flash UID    : 32313334320EAC834330FFFFFFFFFFF
- Chip Flash Size   : 1024 KB
LL Spark Controller Version:b0e99c4

[19:23:23.735]  ble_store_config_num_our_secs:0
ble_store_config_num_peer_secs:0
ble_store_config_num_cccds:0
registered service 0x1812 with handle=1
registering characteristic 0x2a4a with def_handle=2 val_handle=3
registering characteristic 0x2a4b with def_handle=4 val_handle=5
registering characteristic 0x2a4d with def_handle=6 val_handle=7
registering descriptor 0x2908 with handle=9
registering characteristic 0x2a4d with def_handle=10 val_handle=11
registering descriptor 0x2908 with handle=13
registering characteristic 0x2a4c with def_handle=14 val_handle=15
Device Address: 01 02 03 04 05 06


[19:23:42.214]  connection established; status=0 handle=1 our_ota_addr_type=0 our_ota_
↪addr=01 02 03 04 05 06
 our_id_addr_type=0 our_id_addr=01 02 03 04 05 06
 peer_ota_addr_type=1 peer_ota_addr=1e ee c1 b6 69 57
 peer_id_addr_type=1 peer_id_addr=1e ee c1 b6 69 57
 conn_itvl=24 conn_latency=0 supervision_timeout=500 encrypted=0 authenticated=0 bonded=0


[19:23:45.043]  encryption change event; status=0 handle=1 our_ota_addr_type=0 our_ota_
↪addr=01 02 03 04 05 06
 our_id_addr_type=0 our_id_addr=01 02 03 04 05 06
 peer_ota_addr_type=1 peer_ota_addr=1e ee c1 b6 69 57
 peer_id_addr_type=1 peer_id_addr=1e ee c1 b6 69 57
 conn_itvl=24 conn_latency=0 supervision_timeout=500 encrypted=1 authenticated=0 bonded=1

ediv=0 rand=0 authenticated=0 ltk= cddfa0325abc4490ff77622c3075800a irk=␣
↪00000000000000000000000000000000
ediv=0 rand=0 authenticated=0 ltk= cddfa0325abc4490ff77622c3075800a irk=␣
↪4a3711a0c1b7cd2c92d64048e813fe34

[19:23:45.432]  connection updated; status=0 handle=1 our_ota_addr_type=0 our_ota_addr=01␣
↪02 03 04 05 06
 our_id_addr_type=0 our_id_addr=01 02 03 04 05 06
 peer_ota_addr_type=1 peer_ota_addr=1e ee c1 b6 69 57
 peer_id_addr_type=0 peer_id_addr=07 49 34 4d af 50
```

(下页继续)

(续上页)

```
 conn_itvl=6 conn_latency=0 supervision_timeout=500 encrypted=1 authenticated=0 bonded=1


[19:23:45.610]  connection updated; status=0 handle=1 our_ota_addr_type=0 our_ota_addr=01␣
↪02 03 04 05 06
 our_id_addr_type=0 our_id_addr=01 02 03 04 05 06
 peer_ota_addr_type=1 peer_ota_addr=1e ee c1 b6 69 57
 peer_id_addr_type=0 peer_id_addr=07 49 34 4d af 50
 conn_itvl=24 conn_latency=0 supervision_timeout=500 encrypted=1 authenticated=0 bonded=1


[19:23:46.352]  subscribe event; conn_handle=1 attr_handle=7 reason=1 prevn=0 curn=1␣
↪previ=0 curi=0

[19:23:46.412]  subscribe event; conn_handle=1 attr_handle=11 reason=1 prevn=0 curn=1␣
↪previ=0 curi=0

[19:23:48.997]  GPIO ISR in..
P04 occurred  (KEY1 音量增大)
notify_tx event; conn_handle=1 attr_handle=11 status=0 is_indication=0notify_tx event;␣
↪conn_handle=1 attr_handle=11 status=0 is_indication=0key_vol_up pressed

[19:23:49.544]  GPIO ISR in..
P04 occurred
notify_tx event; conn_handle=1 attr_handle=11 status=0 is_indication=0notify_tx event;␣
↪conn_handle=1 attr_handle=11 status=0 is_indication=0key_vol_up pressed

[19:23:50.843]  GPIO ISR in..
P05 occurred  (KEY2 音量减小)
notify_tx event; conn_handle=1 attr_handle=11 status=0 is_indication=0key_vol_down pressed
notify_tx event; conn_handle=1 attr_handle=11 status=0 is_indication=0
[19:23:51.443]  GPIO ISR in..
P05 occurred
notify_tx event; conn_handle=1 attr_handle=11 status=0 is_indication=0key_vol_down pressed
notify_tx event; conn_handle=1 attr_handle=11 status=0 is_indication=0
```

1. hog 相关 GATT Service 的初始化在 gatt_svr.c 中:

```
static const struct ble_gatt_svc_def gatt_svr_svcs[] = {
    {
        /* Service: Heart-rate */
        .type = BLE_GATT_SVC_TYPE_PRIMARY,
        .uuid = BLE_UUID16_DECLARE(BT_UUID_HIDS),
        .characteristics = (struct ble_gatt_chr_def[]) { {
            /* Characteristic: hids information */
            .uuid = BLE_UUID16_DECLARE(BT_UUID_HIDS_INFO),
            .access_cb = gatt_svr_chr_access_hid,
            .flags = BLE_GATT_CHR_F_READ,
        }, {
            /* Characteristic: hids report map */
            .uuid = BLE_UUID16_DECLARE(BT_UUID_HIDS_REPORT_MAP),
            .access_cb = gatt_svr_chr_access_hid,
            .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_READ_ENC,
        }, {
            /* Characteristic: hids inout report */
            .uuid = BLE_UUID16_DECLARE(BT_UUID_HIDS_REPORT),
            .access_cb = gatt_svr_chr_access_hid_input_report,
            .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_NOTIFY,
            .val_handle = &hid_input_handle,
            .descriptors = (struct ble_gatt_dsc_def[]) { {
```

(下页继续)

(续上页)

```
                .uuid = BLE_UUID16_DECLARE(BT_UUID_HIDS_REPORT_REF),
                .access_cb = gatt_svr_chr_access_hid_input_report,
                .att_flags = BLE_ATT_F_READ,
            }, {
                0
            }, }
        }, {
             /* Characteristic: hids consumer report */
            .uuid = BLE_UUID16_DECLARE(BT_UUID_HIDS_REPORT),
            .access_cb = gatt_svr_chr_access_hid_consumer_report,
            .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_NOTIFY,
            .val_handle = &hid_consumer_input_handle,
            .descriptors = (struct ble_gatt_dsc_def[]) { {
                .uuid = BLE_UUID16_DECLARE(BT_UUID_HIDS_REPORT_REF),
                .access_cb = gatt_svr_chr_access_hid_consumer_report,
                .att_flags = BLE_ATT_F_READ,
            }, {
                0
            }, }
        }, {
            /* Characteristic: Body sensor location */
            .uuid = BLE_UUID16_DECLARE(BT_UUID_HIDS_CTRL_POINT),
            .access_cb = gatt_svr_chr_access_hid,
            .flags = BLE_GATT_CHR_F_WRITE_NO_RSP,
        }, {
            0, /* No more characteristics in this service */
        }, }

    }, {
         0, /* No more services. */
    },

};
```

1. 相关 hog 硬件初始化和处理以及发送消息的函数在 `hog.c` 中

5 RAM/Flash 资源使用情况

PAN107x:

```
Flash Size: 151.77k
RAM Size:  38.89 k
```

PAN101x:

```
Flash Size: 138.38k
RAM Size:  14.76 k
```

3.5.4 Solution: BLE HID Uart Mult Roles

1 功能概述

此 sample 为 `pan107` 上演示蓝牙 HID 串口设备的透传功能，支持 1 主 1 从

2 环境要求

- board: `pan107`（芯片型号）开发板 * 3

- uart0: overlay 中设置 P16，P17 作为默认的 LOG 输出端口
- uart1: overlay 中默认 P24 作为 Uart Tx 端-连接开发板 TX0，P10 作为 Uart Rx 端-连接开发板 RX0
- 蓝牙主机设备如手机

3 编译和烧录

例程位置：`<home>\nimble\samples\solutions\ble_hid_uart_mult_roles\keil_107x`

使用 `keil` 进行打开项目进行编译烧录。

4 演示说明

4.1 AT 指令说明

1. 所有 AT 指令必须以`\r\n` 字符结尾。广播状态为 AT 指令模式。连接状态为数据透传。AT 指令模式以字符串格式发送。数据透传串口以 hex 格式发送。
2. 存储参数：

```
typedef struct {
        uint32_t baudrate;
        uint8_t own_mac[6];
        uint8_t device_name[28];
        uint8_t name_length;
        uint8_t bond_mac[6];
        uint32_t passkey;
        uint32_t rst_flag;
} fmc_data;
```

全擦除后上电第一次打印默认初始化参数，用户可以后续通过 AT 命令进行修改

```
default_data_init
Baudrate : 115200
Own_mac : 11 22 33 44 55 66
Bond_mac : 11 22 33 44 66 88
Name_length : 9
Device_name : mult_uart
Passkey : 123456
```

3. AT 指令表

AT 指令序号	AT 指令	回复	说明
1	AT	AT+OK	测试串口通讯是否正常
2	AT+RESET	OK+RESET	复位芯片指令
3	AT+DEFAULT	OK+DEFAULT	恢复出厂设置
4	AT+BAUD?	BAUD+ 波特率	10 进制值（1200-115200）
5	AT+BAUD+115200 示例: AT+BAUD+115200	OK+BAUDNO CHANGE BAUDOVER 115200 REJECT	设置波特率 1200-115200 任意值超出配置限制会被拒绝，相等配置同样不会进行设置切换波特率后需要更换波特率通信
6	AT+MAC?	MAC+ 地址	查询 MAC 地址
7	AT+SETMAC+ 地址 示例: AT+SETMAC+112233445566	OK+SETMAC	设置 MAC 地址成功
8	AT+NAME?	NAME+ 广播名字	查询蓝牙广播名字
9	AT+SETNAME+ 名字 示例: AT+SETNAME+HELLO_PAN	OK+SETNAME	设置广播名字成功，最长 28 字节，超过将会截断
10	AT+BONDMAC?	BONDMAC+ 地址	查询扫描过滤条件的绑定地址
11	AT+BONDMAC+ 地址 示例: AT+BONDMAC+112233446688	OK+BONDMAC	设置扫描过滤条件的绑定地址成功
12	AT+PIN?	PIN+ 设置配对密码	作为从机配对时需要在手机端输入密码
13	AT+PIN+ 配对密码 示例: AT+PIN+234567	OK+PIN	设置密码成功
14	AT+ADV START	OK+ADV START	默认开启了广播
15	AT+ADV STOP	OK+ADV STOP	在广播开启条件下可以停止广播
16	AT+SCAN START	OK+SCAN STARTSCAN DONE	依次输出 2 条第一条代表消息通信成功第二条代表扫描到设备后停止扫描
17	AT+SCAN STOP	OK+SCAN STOP	在扫描开启条件下可以停止扫描
18	AT+CONN 00	OK+CONNCONN DONEDISCOVERY DONECCC DONECONN WHOLE DONE	依次输出 5 条第一条代表消息通信成功后四条代表连接后消息交互
19	AT+DISCONN 00	OK+DISCONNDISCONN DONE	依次输出 2 条第一条代表消息通信成功第二条代表成功断连
20	AT+DEV SHOW	OK+DEV SHOW	显示连接列表目前默认在 log 端口显示列表有需要可以移植到透传端口显示
21	其他	AT+ERROR	未定义

注：当前 NDK 烧录代码后，默认开启广播，未开启扫描，命令 14-20 中仅 16 可用，并且扫描到指定 UUID 进行自动连接

1. 连接状态下，发送透传消息以 hex 格式发送，消息格式如下

Pattern(1B)	Send Connect Index(1B)	data(0~200B)
0x5A	发送给 index 的参数为置位 index bit 位 bit0 为发送给从机，bit1 为发送给主机	

4.2 演示流程

4.2.1AT 配置测试

命令 1-13 可以灵活配置和读取默认配置参数，先进行测试后，建议发送 `AT+DEFAULT` 恢复默认配置

注意：

1. 目前指令 5 设置 921600 会返回信息 OVER 115200 REJECT，设置相等值会返回 NO CHANGE BAUD，设置 115200 以下的串口波特率会返回 OK+BAUD RESET 并 reset 芯片，切换波特率可以继续通信测试
2. 部分设置命令会答应 `RESETING...` 进行芯片重启

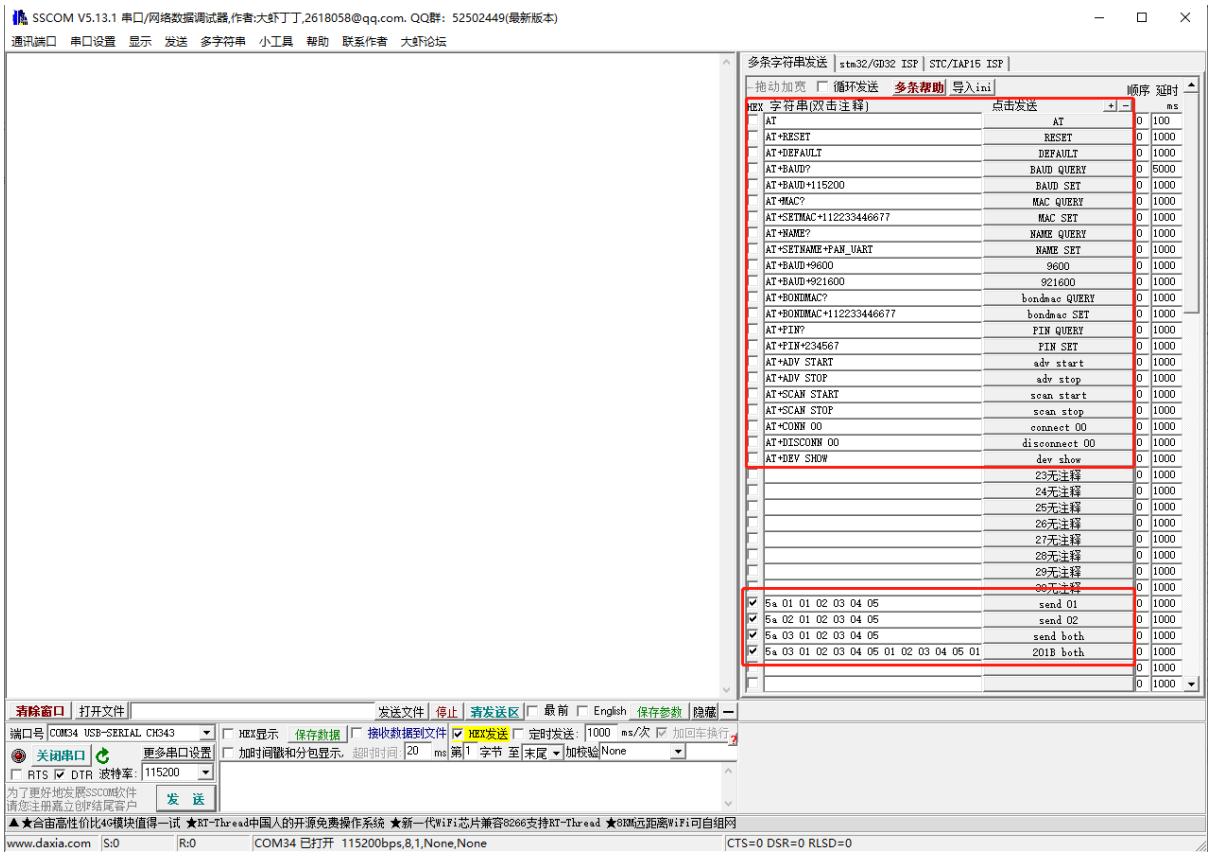


图 49: AT 指令测试窗口

4.2.2 蓝牙状态测试

主机和从机可以同时支持，默认上电开启了广播，最多支持 1 主 1 从

准备 2 块板子 A 和 B，分别烧录程序后，A 板子只作为从机，可以通过 AT 指令修改设备 mac 地址和名字，防止手机扫描到两个设备信息完全一样

然后对 B 板子以以下流程进行测试

4.2.2.1 从机模式

作为从机默认上电开启了蓝牙广播，测试流程可以按照以下顺序

1. 手机端蓝牙连接设备
2. 手机端对属性进行 `notify enable`
3. 芯片端发送消息上报
4. 手机端发送消息下发

4.2.2.2 主机模式

作为主机，需要主动开始扫描，扫描到设备后，主动进行连接已经开启的另一块从机设备，连接从机时默认会进行服务发现、`notify enable`，之后可以进行消息透传测试

1. AT+SCAN START，成功连接后返回 SCAN DONE
2. 芯片主机端进行数据传输测试
3. 可以进行第二块从机的准备和完成扫描连接流程
4. 可以对主机从机同时进行传输

5 开发说明

5.1 **功耗说明** 功耗测试分为广播态测试，连接态测试，并在两种状态下可以通过电流观察空闲 WFI 状态休眠电流

功耗测试结果：

工作模式	平均电流（mA）	WFI 电流（uA）
广播（35ms 广播间隔）	1.18	885
连接（50ms 连接间隔）	0.953	885

6 RAM/Flash 资源使用情况

PAN107x:

```
Flash Size: 156.91k
RAM Size:  45.84 k
```

3.5.5 Solution: BLE Mouse

1 功能概述

此 sample 为 pan107 上演示蓝牙鼠标自动画圈功能（EVB 验证）

2 环境要求

- board: pan107（芯片型号）开发板
- uart0: overlay 中设置 P16，P17 作为默认的 LOG 输出端口
- 蓝牙主机设备如 PC 或者手机

3 编译和烧录

例程位置：<home>\nimble\samples\solutions\ble_mouse\keil_107x

使用 keil 进行打开项目进行编译烧录。

4 演示说明

目前为基础 demo，支持 evb 上配对并识别为鼠标设备，连接后可以运行自动画圈功能

4.1 操作流程说明

1. 编译后全部擦除下载
2. PC 搜索名为 pan_mouse 的鼠标设备并连接
3. 连接后多插拔几次 P04 启动自动画圈功能进行验证

5 RAM/Flash 资源使用情况

PAN107x:

```
Flash Size: 148.24k
RAM Size:  36.01 k
```

3.5.6 Solution: BLE Panchip-CTE Beacon

1 功能概述

此项目演示磐启蓝牙定位标签的功能，通过发送特定的广播数据，实现蓝牙定位功能。
这是磐启蓝牙定位方案中的一部分，有关定位方案的更多信息请参考 **[待补充]**。

2 环境要求

- board: `pan107`
- uart (option): 显示串口 log

3 编译和烧录

例程位置：`<home>\nimble\samples\solutions\ble_panchip_cte_beacon\keil_107x`
使用 `keil` 进行打开项目进行编译烧录。

4 演示说明

烧录完成后，设备自动启动蓝牙广播，可以在手机 `nRF Connect` 或抓包工具上获取如下信息：

- Advertising Type: ADV_SCAN_IND
- Advertising Interval Time: 250ms
- Company ID: (Shanghai Panchip Microelectronics Co., Ltd (0x07D1)
- Device Name: PANCHIP-CTE Beacon

下图是 `nRF Connect`(Android) 扫描到设备后显示的信息。

5 广播数据

广播数据包含两个 AD Element，如下表。

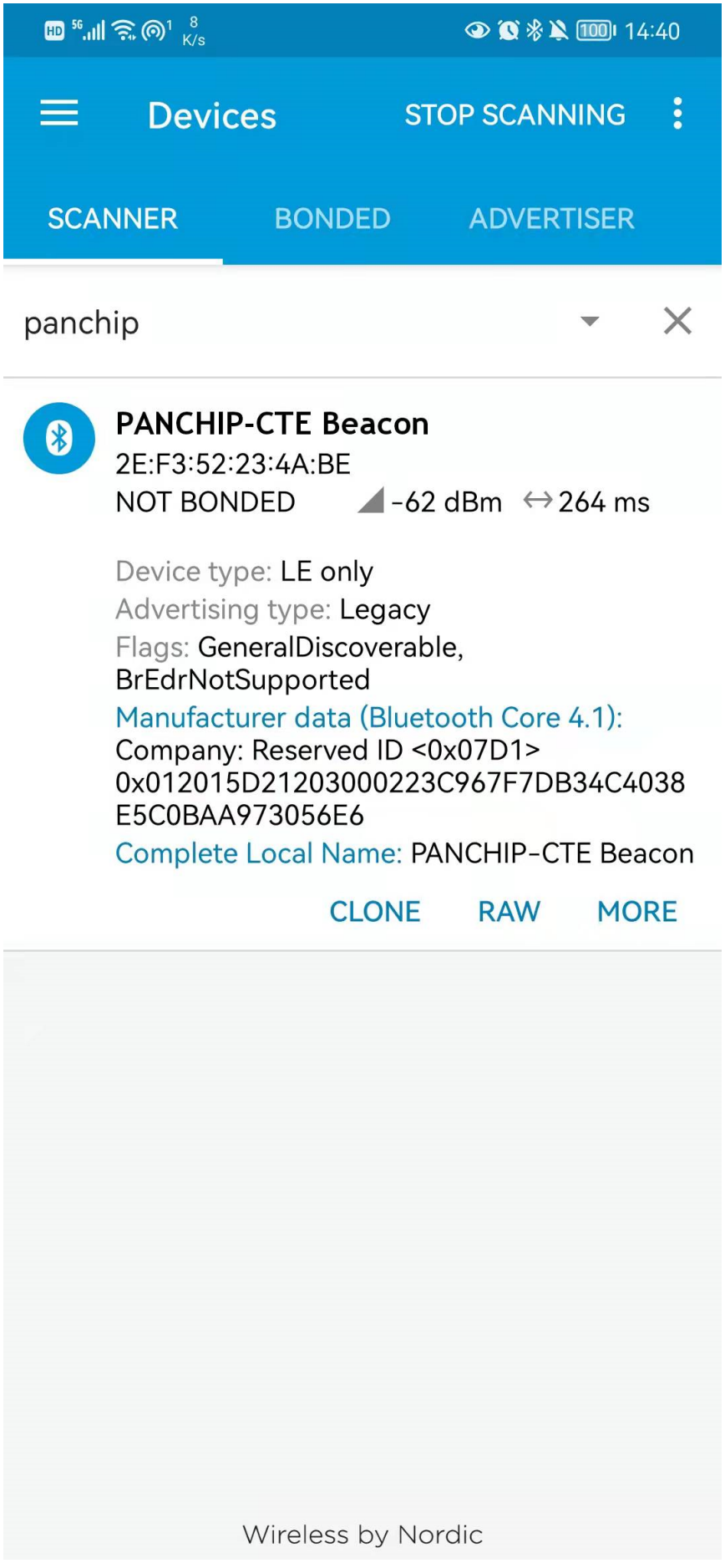


图 50: PANCHIP-CTE Beacon 手机端信息显示界面

Index (Byte)	Data	Name	Description
0	0x02	Length	Length of this AD Element 1
1	0x01	AD Type	Flags
2	0x06	Data	BT_LE_AD_GENERAL, BT_LE_AD_GENERAL
3	0x1B	Length	Length of this AD Element 2
4	0xFF	AD Type	Manufacturer Specific Data
5:6	0x07D1	Company ID	Shanghai Panchip Microelectronics Co., Ltd 厂商 ID 可由用户自定义用于区分设备厂家，标签和基站需要保持一致。
7	0x01	Packet ID	定位包 ID，用于区分同一厂家的不同设备，如标签、手环、IOS 微信小程序和安卓微信小程序等，标签使用 0x01。该部分可由用户自定义，标签和基站需保持一致。
8	0x20	Device Type	设备类型
9	0x15	Header	Carries information of Tag’ s TX rate, TX power and ID type
10:16	0xD2, 0x12, 0x03, 0x00, 0x02, 0x23,	Tag ID	用于区分不同的标签
17	0xC9	Checksum	CRC-8 [Device Type, Header, Tag ID]
18:31	0x67, 0xF7, 0xDB, 0x34, 0xC4, 0x03, 0x8E, 0x5C, 0x0B, 0xAA, 0x97, 0x30, 0x56, 0xE6	DF Field	该字段为辅助定位使用的固定字节。该段内容需保证空中抓取到的是固定频率的电磁波。根据 2402MHz 广播通道的白化算法规则和蓝牙先发送低字节的低比特的特性。修改信道时，需要对此进行调整。

6 RAM/Flash 资源使用情况

PAN107x:

```
Flash Size: 137.78k
RAM Size:  33.01 k
```

3.5.7 BLE PRF SAMPLE

1 功能概述

此项目演示 BLE 从机和 2.4g 同时工作双模例程，BLE 从机例程介绍参考文档bleprph_hr.md。此例程是在 bleprph_hr 例程基础上增加了 prf 2.4g 相关功能。

2 环境要求

- board: `pan107x evb`
- uart(option): 用来显示串口 log（波特率 921600，选项 `8n1`）
- 手机 app `nrf connect`

3 编译和烧录

例程位置：`<home>\nimble\samples\solutions\ble_prf_sample\keil_107x`

使用 keil 进行打开项目进行编译烧录。

4 演示说明

1. 烧录完成后，设备复位会显示上电 log，上电后的 log 如下：

```
[15:57:38.486] 收 ←Try to load HW calibration data..
WARNING: Cannot find valid calib data in current chip!
- Chip Flash UID    : 42503156323339171155OD3756039C78
- Chip Flash Size   : 512 KB

[15:57:38.519] 收 ←rcl calib:30284

[15:57:38.840] 收 ←LL Spark Controller Version:d7c4bfa

[15:57:38.910] 收 ←app started

[15:57:39.421] 收 ←tx done

[15:57:39.921] 收 ←tx done

[15:57:40.421] 收 ←tx done
```

“tx done” 是 2.4g 发送完一包后的打印，例程默认每隔 500ms 发送一次。

2. 使用手机 nrf connect 扫描蓝牙设备名称 ble_hr 并且连接

广播和连接的同时也能发送 2.4g 包。

5 2.4g 初始化说明

2.4g 初始化必须在 BLE 开始广播前，2.4g 初始化代码如下：

```
pan_ant_init();

extern uint32_t BB_UsToTick(uint32_t us);
extern uint32_t RTC_GetCurrentTick(void);

uint32_t tick = RTC_GetCurrentTick();
prf_tx.interval      = BB_UsToTick(500000);                    //prf interval 500ms
prf_tx.slot_duration = 6;                                     //prf work duration 6 * 1.25ms
prf_tx.before_point  = 0;
prf_tx.anchor_point  = tick;
prf_tx.AntStartCback = prf_event_handler;                     //prf event cb
prf_tx.AntStopCback  = NULL;
prf_tx.priority      = 0;                                     //priority lowest 0,1,2; 2␣
↪highest
pan_ant_create(&prf_tx);

panchip_prf_init(&tx_config);
panchip_prf_set_chn(tx_config.rf_channel);

/*adr match bit */
PRI_RF_SetAddrMatchBit(PRI_RF, 0);
panchip_prf_set_data(&tx_payload);

/* Begin advertising */
blehr_advertise();
```

1. 需要定义一个结构体” ab_event_node_t”
2. 首先调用” pan_ant_init” 接口，然后注册结构体” pan_ant_create(&prf_tx)”

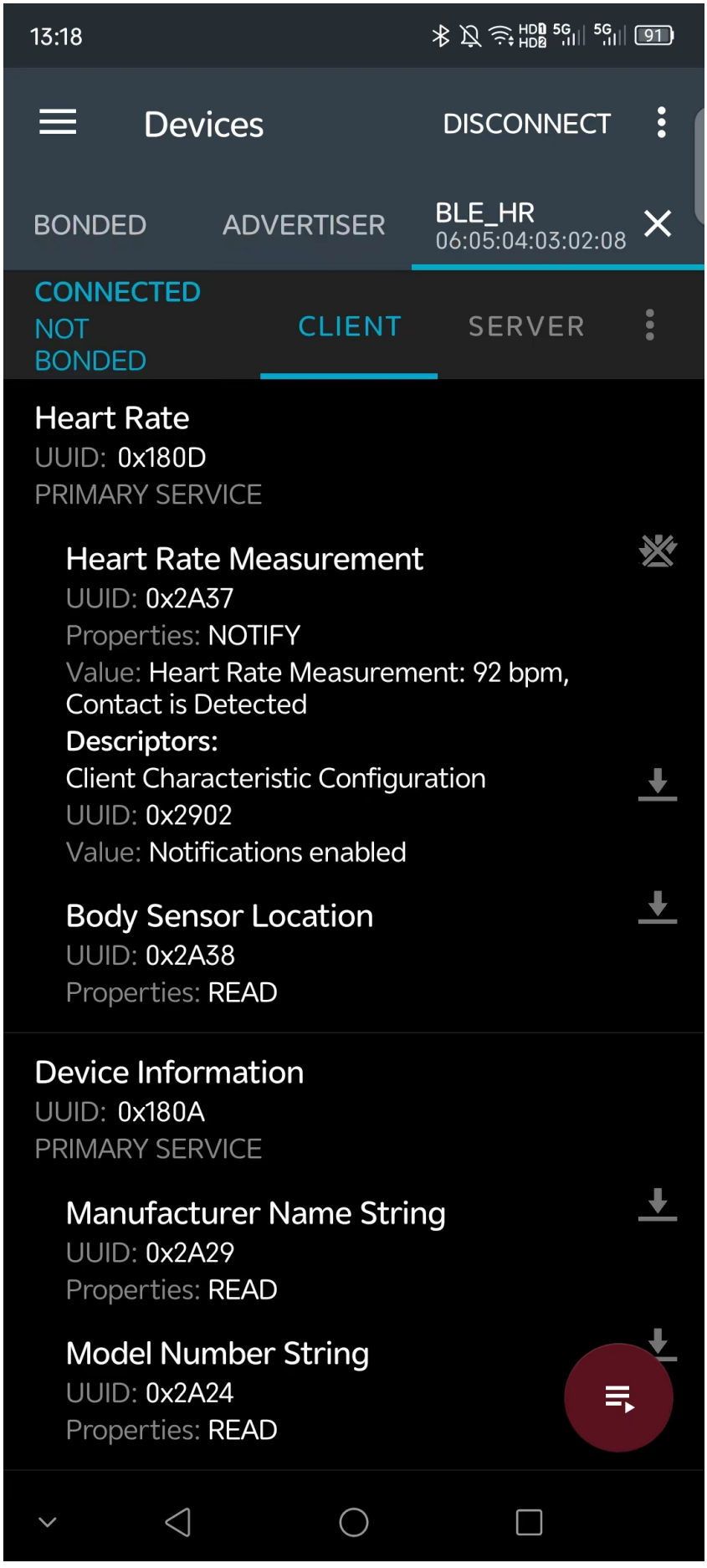


图 51: nrf connect 连接 ble_hr

3. 结构体中需要填写几个关键参数：interval、slot_duration、anchor_point、AntStartCback、priority。
4. 注册完 2.4g 事件后就可以启动 2.4g 收发了，例程在 500ms 定时回调中启动 2.4g 发射
5. 发射完成后会有 tx 的中断

6 RAM/Flash 资源使用情况

PAN107x:

```
Flash Size: 118.58k
RAM Size:  36.12 k
```

3.5.8 Solution: BLE RGB Light

1 功能概述

本文主要介绍 PAN10xx BLE RGB 灯和手机 APP 进行连接，通过 APP 控制 RGB 灯的亮度与颜色, 此功能支持 `pan101x` 和 `pan107x` 芯片

2 环境要求

- board: `pan107x evb` 或 `pan101x evb`
- uart (option): 显示串口 log
- 安卓亿觅精灵灯 app V1.5.5，或微信小程序 (**待补充**)

3 编译和烧录

`pan107x` 芯片例程位置：`<home>\nimble\samples\solutions\ble_rgb_light\keil_107x`

`pan101x` 芯片例程位置：`<home>\nimble\samples\solutions\ble_rgb_light\keil_101x`

使用 `keil` 进行打开项目进行编译烧录。

4 演示说明

1. PAN107 EVB 板 GPIO P11、P12、P14 与 RGB 电路用跳线帽连接。
2. EVB 板上电灯的颜色默认是蓝色，BLE 广播设备的名字是”b+EMIE Elfy”。
3. 打开安卓手机”亿觅精灵灯“app，在 app 上启动搜索设备。
4. 搜索到后点击连接，连接成功后就可以控制灯的开关和颜色了。

5 设备连接和控制

5.1 广播数据

Adv Data Type	Description	Length	Detail
0xff	Device id	10byte	0xD1, 0x07, 0xc9, 0x7a, 0xbb, 0x8f, 0xdd, 0x4b, 0x00, 0x11
0x07	128-bit UUID	16byte	0x9e, 0xca, 0xdc, 0x24, 0x0e, 0xe5, 0xa9, 0xe0,0x93, 0xf3, 0xa3, 0xb5, 0x01, 0x20, 0x40, 0x6e
0x09	Device name	n byte	“b+EMIE Elfy”

5.2 GATT 服务

Function	Service Attribute	UUID(128bit)
Useless	Primary service	0x9e, 0xca, 0xdc, 0x24, 0x0e, 0xe5, 0xa9, 0xe0,0x93, 0xf3, 0xa3, 0xb5, 0x01, 0x20, 0x40, 0x6e
控制灯的状态	Write characteristic declaration	0x9e, 0xca, 0xdc, 0x24, 0x0e, 0xe5, 0xa9, 0xe0, 0x93, 0xf3, 0xa3, 0xb5, 0x02, 0x20, 0x40, 0x6e
Notify 灯的状态	notify characteristic declaration	0x9e, 0xca, 0xdc, 0x24, 0x0e, 0xe5, 0xa9, 0xe0,0x93, 0xf3, 0xa3, 0xb5, 0x03, 0x20, 0x40, 0x6e

5.3 通信协议

5.3.1 Light Control UUID = {0x9e, 0xca, 0xdc, 0x24, 0x0e, 0xe5, 0xa9, 0xe0, 0x93, 0xf3, 0xa3, 0xb5, 0x03, 0x20, 0x40, 0x6e}

Function	Length	Detail
off	2byte	off：0xaa, 0x03
Color	5byte	0xaa,0x16,red：0~255,green：0~255,blue：0~255

控制灯的开关、颜色。

5.3.2 Notify Light Status UUID = {0x9e, 0xca, 0xdc, 0x24, 0x0e, 0xe5, 0xa9, 0xe0, 0x93, 0xf3, 0xa3, 0xb5, 0x02, 0x20, 0x40, 0x6e}

每次收到控制命令后将灯的状态通知给手机 app。

6 RAM/Flash 资源使用情况

PAN107x:

```
Flash Size: 139.41k
RAM Size:  33.76 k
```

PAN101x:

```
Flash Size: 126.77k
RAM Size:  12.89 k
```

3.5.9 Solution: BLE Spi Tft Lcd

1 功能概述

本文主要介绍 PAN10xx BLE TFT LCD 和手机 APP 进行连接，通过 APP 发送英文字符并在 TFT LCD 上显示出来，此功能支持 `pan101x` 和 `pan107x` 芯片

2 环境要求

- board: `pan107x evb` 或 `pan101x evb`
- uart (option): 显示串口 log
- NRF Connect/BLE 调试助手 APP

3 编译和烧录

pan107x 芯片例程位置：<home>\nimble\samples\solutions\ble_spi_tft_lcd\keil_107x

pan101x 芯片例程位置：<home>\nimble\samples\solutions\ble_spi_tft_lcd\keil_101x

使用 keil 进行打开项目进行编译烧录。

4 演示说明

1. PAN107 EVB 板 GPIO P04(CLK)、P05(DC)、P06(RST)、P11(MOSI)、P15(CS) 与 EVB 板 OLED 电路用跳线帽连接。
2. PAN101 EVB 板 GPIO P14(CLK)、P22(DC)、P23(RST)、P11(MOSI)、P15(CS) 与 EVB 板 OLED 电路用跳线帽连接。
3. EVB 板上 LCD 默认是雪花，BLE 广播设备的名字是” b+spi lcd”。
4. 打开安卓手机” NRF Connect “app，在 app 上启动搜索设备。
5. 搜索到后点击连接，连接成功后输入字符发送，观测 TFT LCD 显示是否与输入一致。

5 设备连接和控制

5.1 广播数据

Adv Data Type	Description	Length	Detail
0xff	Device id	10byte	0xD1, 0x07, 0xc9, 0x7a, 0xbb, 0x8f, 0xdd, 0x4b, 0x00, 0x11
0x07	128-bit UUID	16byte	0x9e, 0xca, 0xdc, 0x24, 0x0e, 0xe5, 0xa9, 0xe0,0x93, 0xf3, 0xa3, 0xb5, 0x01, 0x20, 0x40, 0x6e
0x09	Device name	n byte	“b+spi lcd”

5.2 GATT 服务

Function	Service Attribute	UUID(128bit)
Useless	Primary service	0x9e, 0xca, 0xdc, 0x24, 0x0e, 0xe5, 0xa9, 0xe0,0x93, 0xf3, 0xa3, 0xb5, 0x01, 0x20, 0x40, 0x6e
控制 LCD 字符显示	Write characteristic declaration	0x9e, 0xca, 0xdc, 0x24, 0x0e, 0xe5, 0xa9, 0xe0,0x93, 0xf3, 0xa3, 0xb5, 0x01, 0x20, 0x40, 0x6e

6 RAM/Flash 资源使用情况

PAN107x:

```
Flash Size: 142.41k
RAM Size:  33.76 k
```

PAN101x:

```
Flash Size: 100.80k
RAM Size:  12.84 k
```

3.5.10 BLE Vehicles Key

1 功能概述

此项目演示基于 HID 服务的自动连接服务，通过 RSSI 值的大小模拟实现电动二轮车的利用距离自动开关功能。

2 环境要求

- board: `pan107x evb`
- uart(option): 用来显示串口 log（波特率 921600，选项 `8n1`）s_key

3 编译和烧录

例程位置：`<home>\nimble\samples\solutions\ble_vehicles_key\keil_107x`

使用 `keil` 进行打开项目进行编译烧录。

4 演示说明

1. 在设置蓝牙界面连接 `vehicles key` 进行配对后会自动跟踪 rssi 值的大小，模拟电动二轮车钥匙实现近距离自动开关。

```
[19:57:21.854]  Try to load HW calibration data.. DONE.
- Chip Type         : 0x80
- Chip CP Version   : None
- Chip FT Version   : 8
- Chip MAC Address  : D0000C0CBBF5
- Chip Flash UID    : 32313334320EAC834330FFFFFFFFFFF
- Chip Flash Size   : 1024 KB
LL Spark Controller Version:b0e99c4

[19:57:21.919]  ble_store_config_num_our_secs:0
ble_store_config_num_peer_secs:0
ble_store_config_num_cccds:0
registered service 0x1812 with handle=1
registering characteristic 0x2a4a with def_handle=2 val_handle=3
registering characteristic 0x2a4b with def_handle=4 val_handle=5
registering characteristic 0x2a4d with def_handle=6 val_handle=7
registering descriptor 0x2908 with handle=9
registering characteristic 0x2a4d with def_handle=10 val_handle=11
registering descriptor 0x2908 with handle=13
registering characteristic 0x2a4c with def_handle=14 val_handle=15
Device Address: 01 02 03 04 05 06


[19:57:28.164]  connection established; status=0 handle=1 our_ota_addr_type=0 our_ota_
↪addr=01 02 03 04 05 06
 our_id_addr_type=0 our_id_addr=01 02 03 04 05 06
 peer_ota_addr_type=1 peer_ota_addr=f4 be 2e 4e 35 50
 peer_id_addr_type=1 peer_id_addr=f4 be 2e 4e 35 50
 conn_itvl=24 conn_latency=0 supervision_timeout=500 encrypted=0 authenticated=0 bonded=0


[19:57:30.923]  encryption change event; status=0 handle=1 our_ota_addr_type=0 our_ota_
↪addr=01 02 03 04 05 06
 our_id_addr_type=0 our_id_addr=01 02 03 04 05 06
 peer_ota_addr_type=1 peer_ota_addr=f4 be 2e 4e 35 50
```

(下页继续)

(续上页)

```
 peer_id_addr_type=1 peer_id_addr=f4 be 2e 4e 35 50
 conn_itvl=24 conn_latency=0 supervision_timeout=500 encrypted=1 authenticated=0 bonded=1

ediv=0 rand=0 authenticated=0 ltk= f5f99f23fb223b4ff0f5f7f21c0146d1 irk=␣
↪00000000000000000000000000000000
ediv=0 rand=0 authenticated=0 ltk= f5f99f23fb223b4ff0f5f7f21c0146d1 irk=␣
↪4a3711a0c1b7cd2c92d64048e813fe34

[19:57:31.223]  connection updated; status=0 handle=1 our_ota_addr_type=0 our_ota_addr=01␣
↪02 03 04 05 06
 our_id_addr_type=0 our_id_addr=01 02 03 04 05 06
 peer_ota_addr_type=1 peer_ota_addr=f4 be 2e 4e 35 50
 peer_id_addr_type=0 peer_id_addr=07 49 34 4d af 50
 conn_itvl=6 conn_latency=0 supervision_timeout=500 encrypted=1 authenticated=0 bonded=1

connection updated; status=0 handle=1 our_ota_addr_type=0 our_ota_addr=01 02 03 04 05 06
 our_id_addr_type=0 our_id_addr=01 02 03 04 05 06
 peer_ota_addr_type=1 peer_ota_addr=f4 be 2e 4e 35 50
 peer_id_addr_type=0 peer_id_addr=07 49 34 4d af 50
 conn_itvl=24 conn_latency=0 supervision_timeout=500 encrypted=1 authenticated=0 bonded=1

[19:57:32.135]
subscribe event; conn_handle=1 attr_handle=7 reason=1 prevn=0 curn=1 previ=0 curi=0

[19:57:32.189]
subscribe event; conn_handle=1 attr_handle=11 reason=1 prevn=0 curn=1 previ=0 curi=0

[19:57:32.238]  open led
```

2. 如果使用 EVB 板的话，请将 RGB-G 的跳线帽连接，如果 RSSI 距离达到接近阈值时会触发开的动作，足够远时会触发关的距离动作。
3. 关于 rssi 的滤波算法使用的是去除最大和最小值各 RSSI_REMOVE_BORDER_NUM 个，默认 RSSI_REMOVE_BORDER_NUM 为 2 个，排序使用冒泡排序。
4. RSSI 的整体处理在 hog.c 中。

5 RAM/Flash 资源使用情况

PAN107x:

```
Flash Size: 147.89k
RAM Size:  36.22 k
```

3.5.11 Solution: Electronic Shelf Label

1 功能概述

此 sample 为 pan107x(40pin 芯片) 在电子价签板下的应用。

具体支持的 feature 如下：

1. 外挂 spi flash：外挂 flash 存储价签图案数据，每隔 45s 通过 dma 方式从 flash 读取一个图案
2. epd 墨水屏：外挂 flash 读取的数据通过 3 线 spi 传输给墨水屏，启动墨水屏刷屏
3. 低功耗模式：刷图完成后芯片及 epd 均进入休眠模式，模块进入超低功耗模式（standby），定时 15s 唤醒
4. RF 发送：芯片唤醒进入 rf 发送流程
5. 每 3 次发送完成后，启动刷屏流程，1~4 步骤重复

2 环境要求

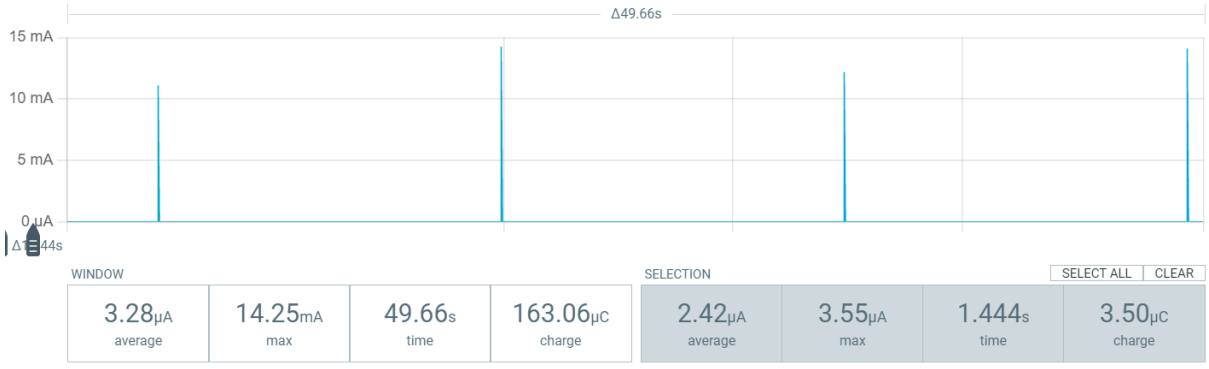
- board: ‘pan107x 40pin esl 价签板
- 外挂 flash、墨水屏
- 电流监测工具 nrf ppk

3 编译和烧录

例程位置：`nimble\pan107x_samples\solutions\esl\keil_107x`
使用 keil 工具可以对其进行编译、烧录、调试等操作。

4 演示说明

1. 准备 esl 价签板，107/FM/EPD 跳线帽短接
2. 插入 epd2266 墨水屏（SE2266JS0C5）
3. 打开 PPK 并使用其供电 3.3v
4. 观测 PPK 电流变化及墨水屏刷屏过程（45s 刷屏一次），电流每 15s 进入低功耗



5 主要数据结构说明

配置的结构体 “pan_prf_config_t”，各成员介绍如下：

Type	name	Description
prf_mode_t	work_mode	工作模式配置，包括普通型和增强型
prf_chip_mode_sel_t	chip_mode	xn297 通信协议和 nordic 通信协议配置
prf_trx_mode_t	trx_mode	收发模式配置
prf_phy_t	phy	通信速率配置，可配置为 1M 和 2M
prf_crc_sel_t	crc	数据包 CRC 配置，可配置为 crc 16bit，crc 8bit，crc 24bit ，no crc
prf_scramble_sel_t	src	数据包扰码的配置，可配置为使用扰码和不使用扰码
uint16_t	rx_timeout	接收超时时间配置，最大 50000us
uint16_t	rf_channel	2.4g 频点配置，任意频点可设 (2402Mhz~2480Mhz)
uint8_t	tx_no_ack	配置增强型模式下 tx 是否需要 ack
prf_trf_t	trf_type	nordic 的长包模式配置，最大 payload 的长度为 255
uint8_t	rx_length	rx 接收数据包长度配置，增强型模式下可不配置
uint8_t	sync_length	接入地址长度配置，可配置为 3、4、5 字节
uint8_t	sync[5]	接入地址的内容 (xn297 模式下可白化地址，防止出现长 0 和长 1 的地址)
prf_dev_sel_t	dev	设置 deviation，可以选择 BLE 模式（1M 250k；2M 500k），NRF 模式（1M 160K；2M 320）
int8_t	tx_power	设置发射功率，范围 (-45dbm~7dbm)
uint8_t	pid_manual_flag	pid 手动配置的标志，使能后可以自定义 pid
uint8_t	crc_include_sync	crc 计算包含地址
uint8_t	src_include_sync	白化包含地址
uint16_t	tx_trans_time	发送传输时间设置
uint16_t	rx_trans_time	接收传输时间设置
prf_pipe_t	pipe	管道配置，可配置为 0~7

prf_mode_t:

Type	Value	Description
PRF_MODE_NORMAL	0	普通型
PRF_MODE_ENHANCE	1	增强型
PRF_MODE_NORMAL_M1	2	普通型 M1 模式

prf_chip_mode_sel_t:

Type	Value	Description
PRF_CHIP_MODE_SEL_BLE	1	蓝牙模式
PRF_CHIP_MODE_SEL_XN297	2	XN297 模式
PRF_CHIP_MODE_SEL_NORDIC	3	NORCDIC 模式

prf_trx_mode_t:

Type	Value	Description
PRF_TX_MODE	0	2.4G 发射
PRF_RX_MODE	1	2.4G 接收

prf_phy_t:

Type	Value	Description
PRF_PHY_1M	1	1M 通信速率
PRF_PHY_2M	2	2M 通信速率
PRF_PHY_CODED_S8	3	S8 模式
PRF_PHY_CODED	4	S2 模式
PRF_PHY_250K	5	250K 模式

prf_crc_sel_t:

Type	Value	Description
PRF_CRC_SEL_NOCRC	0	no crc
PRF_CRC_SEL_CRC8	1	crc 8bit
PRF_CRC_SEL_CRC16	2	crc 16bit
PRF_CRC_SEL_CRC24	3	crc 24bit

prf_scramble_sel_t:

Type	Value	Description
PRF_SRC_SEL_NOSRC	0	不使能扰码
PRF_SRC_SEL_EN	1	使能扰码

prf_dev_sel_t:

Type	Value	Description
PRF_DEV_NRF	1	NRF 模式 deviation 配置，1M 170k，2M 340K
PRF_DEV_BLE	2	NRF 模式 deviation 配置，1M 250k，2M 500K

prf_addr_length_sel_t:

Type	Value	Description
PRF_ADDR_LENGTH_SEL_3	3	3 BYTE 地址长度
PRF_ADDR_LENGTH_SEL_4	4	4 BYTE 地址长度
PRF_ADDR_LENGTH_SEL_5	5	5 BYTE 地址长度

prf_pipe_t:

Type	Value	Description
PRF_PIPE0	1«0	管道 0
PRF_PIPE1	1«1	管道 1
PRF_PIPE2	1«2	管道 2
PRF_PIPE3	1«3	管道 3
PRF_PIPE4	1«4	管道 4
PRF_PIPE5	1«5	管道 5
PRF_PIPE6	1«6	管道 6
PRF_PIPE7	1«7	管道 7

prf_trf_t:

Type	Value	Description
PRF_TRF_NORMAL	0	普通模式传输
PRF_TRF_NRF52	1	NRF 模式传输
PRF_TRF_B250K	2	B250K 模式传输

6 补充说明

补充说明当前功耗测试情况，支持中遇到的问题（供参考）及已知仍可能存在的问题

6.1 功耗说明 略

7 RAM/Flash 资源使用情况

PAN107x:

```
Flash Size:  34.34k
RAM Size:  3.53 k
```

3.5.12 Solution: Multimode Mouse

1 功能概述

此 sample 为 pan107 上演示 2.4G 鼠标跳频发送至 Dongle 自动画圈的功能，后续基于此扩展多模实体鼠标功能

2 环境要求

- board: pan107（芯片型号）开发板 * 2
- uart0: overlay 中设置 P16，P17 作为默认的 LOG 输出端口

3 编译和烧录

例程位置：<home>\nimble\samples\solutions\multimode_mouse_107x

使用 keil 进行打开项目进行编译烧录。

4 演示说明

准备好 multimode_mouse_dongle 烧录 dongle 程序,dongle USB 端插入电脑

烧录 multimode_mouse 工程，重启即可调频自动画圈

5 RAM/Flash 资源使用情况

PAN107x:

```
Flash Size:  22.87k
RAM Size:  7.23 k
```

3.5.13 Solution: Multimode Mouse Dongle

1 功能概述

此 sample 为 pan107 上演示 Dongle 配合主机端自动画圈的功能

2 环境要求

- board: pan107（芯片型号）开发板
- uart0: overlay 中设置 P16，P17 作为默认的 LOG 输出端口

3 编译和烧录

例程位置：`<home>\nimble\samples\solutions\multimode_mouse_dongle\keil_107x`

使用 `keil` 进行打开项目进行编译烧录。

4 演示说明

准备好 `multimode_mouse_dongle` 烧录 dongle 程序，dongle USB 端插入电脑

烧录 `multimode_mouse` 工程，重启即可调频自动画圈

5 RAM/Flash 资源使用情况

PAN107x:

```
Flash Size:  27.59k
RAM Size:  8.35 k
```

蓝牙例程

源码路径: `<PAN1070-NDK>\01_SDK\nimble\samples\bluetooth`

例程	说明
Bluetooth: Central and Peripheral	演示蓝牙主从一体功能
Bluetooth: Central	演示蓝牙主机功能，发现设备并与设备建立连接和断连
Bluetooth: BLE Peripheral ENC	演示外设以及加密配对功能，可以和主机示例进行对测
Bluetooth: Peripheral HR	演示蓝牙从机功能，包含 GATT 服务：HR (Heart Rate)，连接订阅服务后，会上报虚拟的心率值，低功耗演示 demo
Bluetooth: Peripheral HR_OTA	演示蓝牙丛机 OTA 功能，包含完整的蓝牙通用的 SMP 服务，配合手机 nrf connect 进行 OTA 升级
Bluetooth: Peripheral DISTANCE	演示蓝牙从机 s2 s8 编码长距离传输的功能
Bluetooth: BLE Multi Roles	演示蓝牙多主多从功能

低功耗例程

源码路径: `<PAN1070-NDK>\01_SDK\nimble\samples\low_power`

例程	说明
LowPower: DeepSleep GPIO Key Wakeup	演示 SoC 进入 DeepSleep 状态，并通过 GPIO 按键将其唤醒
LowPower: DeepSleep GPIO PWM Wakeup	演示 SoC 进入 DeepSleep 状态，使用外部 PWM 波形通过 GPIO 将其唤醒
LowPower: DeepSleep PWM Waveform Generator	演示 SoC 在 DeepSleep 状态下输出 PWM 波形，并使用 APB HW Timer0 定时唤醒并修改 PWM 波形周期和占空比
LowPower: DeepSleep SleepTimer Wakeup	演示 SoC 进入 DeepSleep 状态，并通过 SleepTimer 定时器将其唤醒
LowPower: Standby Mode1 GPIO Key Wakeup	演示 SoC 进入 Standby Mode 1 状态，并通过 GPIO 按键将其唤醒
LowPower: Standby Mode1 SleepTimer Wakeup	演示 SoC 进入 Standby Mode 1 状态，并通过 SleepTimer 定时器将其唤醒
LowPower: Standby Mode0 P02 Key Wakeup	演示 SoC 进入 Standby Mode 0 状态，并通过 WKUP (P02) 按键将其唤醒
LowPower: Multiple Wakeup Source	演示 SoC 多种唤醒源、多种低功耗模式之间的切换

外设驱动例程

源码路径: `<PAN1070-NDK>\01_SDK\nimble\samples\peripheral`

例程	说明
Peripheral: GPIO Input With Interrupt	演示使用 GPIO HAL Driver 实现中断方式的 GPIO 输入检测功能
Peripheral: GPIO Input Polling	演示使用 GPIO HAL Driver 实现查询方式的 GPIO 输入检测功能
Peripheral: GPIO Open-Drain Output	演示使用 GPIO HAL Driver 实现 GPIO 开漏（Open-Drain）输出功能
Peripheral: GPIO Push-Pull Output	演示使用 GPIO HAL Driver 实现 GPIO 推挽（Push-Pull）输出功能
Peripheral: GPIO Simple Convenient APIs	演示 GPIO 底层 Driver 中提供的几个简单好用的接口

固件保护例程

源码路径: `<PAN1070-NDK>\01_SDK\nimble\samples\security`

例程	说明
Security: Firmware Encryption	演示芯片通过固件加密、硬件解密的机制保护 Flash 关键代码的方法

解决方案

源码路径: `<PAN1070-NDK>\01_SDK\nimble\samples\solutions`

例程	说明
Solution: BLE HID Selfie	自拍解决方案，通过蓝牙 HID 控制手机拍照
Solution: BLE HID Uart Mult Roles	蓝牙串口透传解决方案，演示蓝牙 hid 串口透传功能，支持 1 主 1 从
Solution: BLE Panchip-CTE Beacon	Panchip 蓝牙定位标签方案，通过发送特定的广播数据，实现蓝牙定位功能
Solution: BLE PRF SAMPLE	BLE 和私有 2.4G 协议双模例程，BLE 和 2.4G 可同时工作
Solution: BLE mouse	模拟蓝牙鼠标功能，连接电脑后进行模拟画圈演示
Solution: BLE RGB Light	蓝牙 RGB 灯控方案，演示 BLE RGB 灯与手机 APP 进行连接，通过 APP 控制 RGB 灯的亮度与颜色
Solution: BLE Vehicles Key	蓝牙车钥匙解决方案，演示基于 HID 服务的自动连接服务
Solution: Electronic Shelf Label	电子货架标签方案演示例程，支持外部 SPI Flash 存储、EPD 墨水屏、低功耗模式、RF 通信等功能
Solution: Multimode Mouse	多模鼠标 sample，目前仅支持 2.4G 模式自动画圈
Solution: Multimode Mouse Dongle	多模鼠标接收器，配合 2.4G 鼠标端自动画圈，测试收包数和距离使用

3.6 MCU Keil 例程

例程源码路径: `<PAN1070-NDK>\03_MCU\mcu_samples`

MCU 底层驱动（Low Level Driver）Keil 例程:

例程	说明
MCU Low Level ADC Driver Sample	MCU 底层 ADC 驱动例程演示说明
MCU Low Level CLKTRIM Driver Sample	MCU 底层 Clock Trim 驱动例程演示说明
MCU Low Level CLK Driver Sample	MCU 底层 CLK 驱动例程演示说明
MCU Low Level DMA Driver Sample	MCU 底层 DMA 驱动例程演示说明
MCU Low Level eFuse Driver Sample	MCU 底层 eFuse 驱动例程演示说明
MCU Low Level FMC Driver Sample	MCU 底层 FMC 驱动例程演示说明
MCU Low Level GPIO Driver Sample	MCU 底层 GPIO 驱动例程演示说明
MCU Low Level I2C Driver Sample	MCU 底层 I2C 驱动例程演示说明
MCU Low Level PWM Sample	MCU 底层 PWM 驱动例程演示说明
MCU Low Level SPI Sample	MCU 底层 SPI 驱动例程演示说明
MCU Low Level TIMER Sample	MCU 底层 TIMER 驱动例程演示说明
MCU Low Level UART Sample	MCU 底层 UART 驱动例程演示说明
MCU Low Level WDT Sample	MCU 底层 WDT 驱动例程演示说明
MCU Low Level WWDT Sample	MCU 底层 WWDT 驱动例程演示说明
MCU DebugProtect Sample	MCU Debug Protect 调试接口保护例程演示说明
MCU PRF TRX Sample	MCU 私有 2.4G 通信开发指南
MCU PRF UI Distance Test Sample	MCU 私有 2.4G 距离测试例程演示说明

Chapter 4

开发指南

4.1 NDK Configuration 开发指南

4.1.1 1. 背景介绍

ndk 添加一套配置系统一方面方便用户进行开发，另一方面是为了方便管理不同芯片平台，本章主要介绍配置的含义，以及 `pan107x` 和 `pan101x` 配置的区别。

4.1.2 2. 配置概述

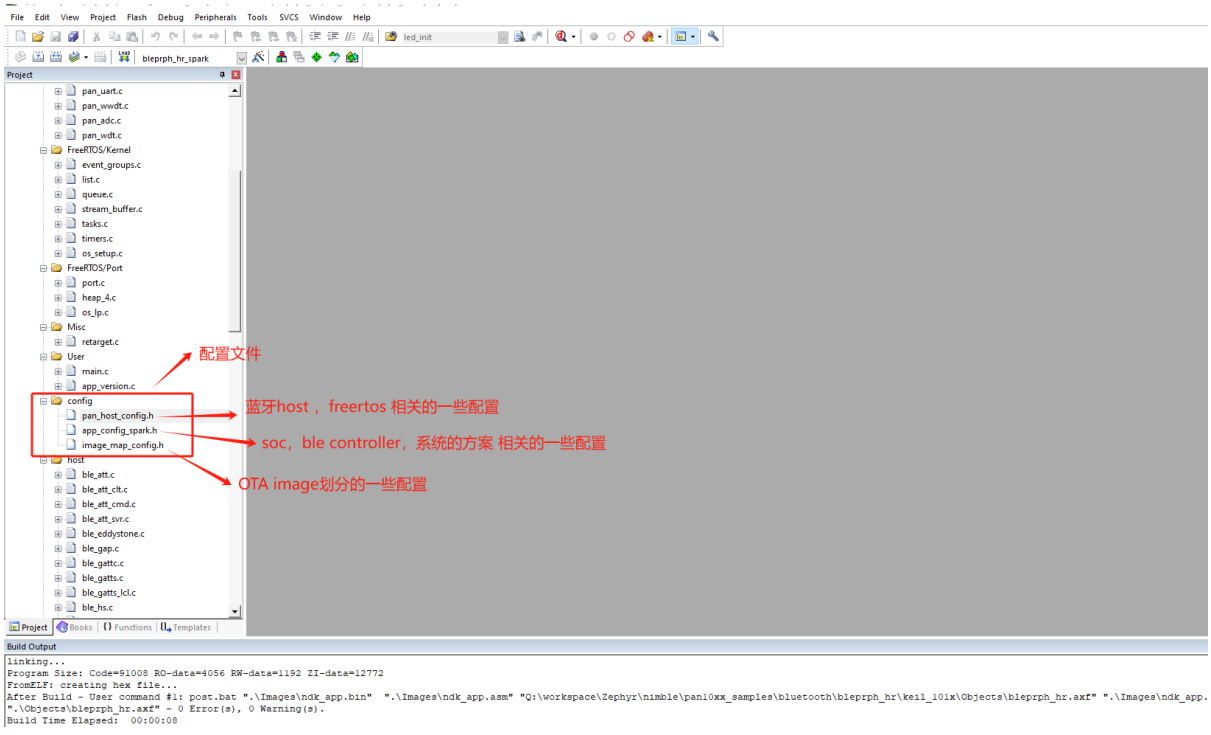


图 1: configuration overview

`pan_host_config` 主要是 freertos 线程栈大小，以及 nimble host 传输的 buf 的配置，单独引用出来，客户可以根据自己的需求优化 ram 的使用量 `app_config_spark.h` 主要是系统方案，soc，ble controller 相关的配置 `image_map_config.h` 主要是 OTA 时候，flash 区域的划分，详情参考 [mcuboot]](./ndk_mcu_boot.md)

4.1.3 3. pan107x 和 pan101x 工程配置以及区别

pan101x 是一个只用 16k ram，pan107x 有 48K ram，所以 pan101 只能运用在一些简单的外设功能，下面介绍 101x 和 107x 配置不同，方便用户移植 107x 的工程到 101x 芯片上，本次 sdk release `BLE Peripheral HR` 和 `BLE RGB Light` 例程中，分别演示了 pan107x 和 pan101x 的工程配置。用户可以参考进行修改，下面解释一些关键的差异点。

app_config_spark.h 配置

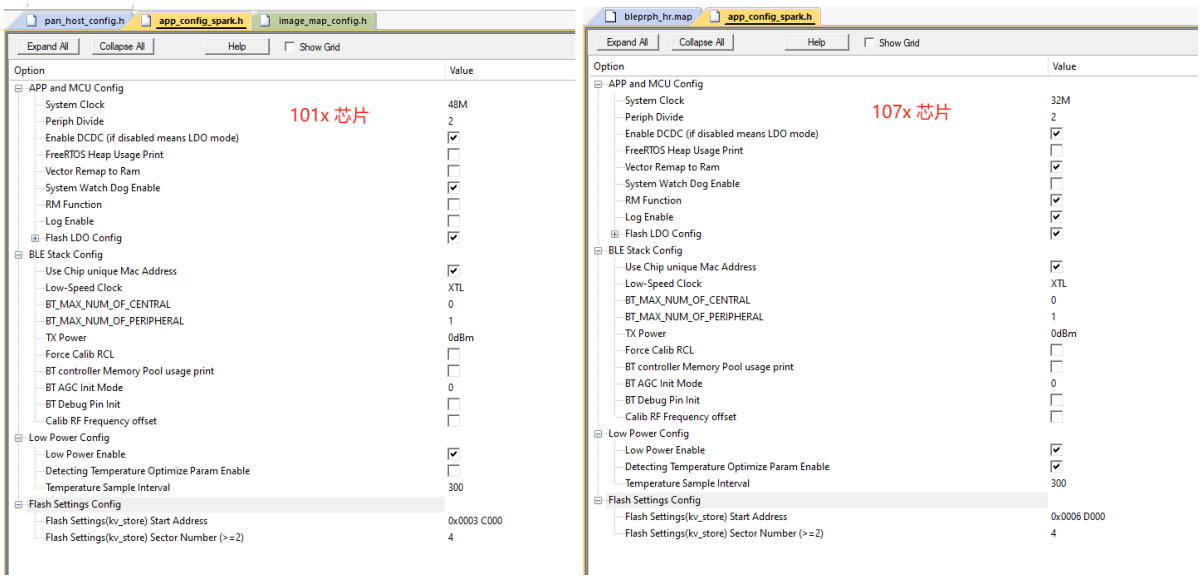


图 2: app configuration

1. pan107x 可以选择 48M 或者 32M，但是 pan101x **暂时只能选择** 48M
2. ram function 功能，107x 可以选择，但是 pan101x **暂时不能选择**，原因是 101x 的 ram 受限导致的
3. flash settings 建议按照上图配置，pan101x 选择 0x3c000，pan107x 选择 0x6d000

其他选项 pan107x 或者 pan101x 都可以选择，只要确保 pan101x 可以编译通过即可。

pan_host_config 配置

自行参考各个工程的配置

ble_spark.lib

pan107x 选择 `lib\pan107x_spark\ble_spark.lib` pan101x 选择 `lib\pan101x_spark\ble_spark.lib`

芯片宏添加

4.2 NDK App 开发指南

本文主要通过一些示例，介绍**蓝牙应用开发**过程中常用的方法以及可能遇到的问题。

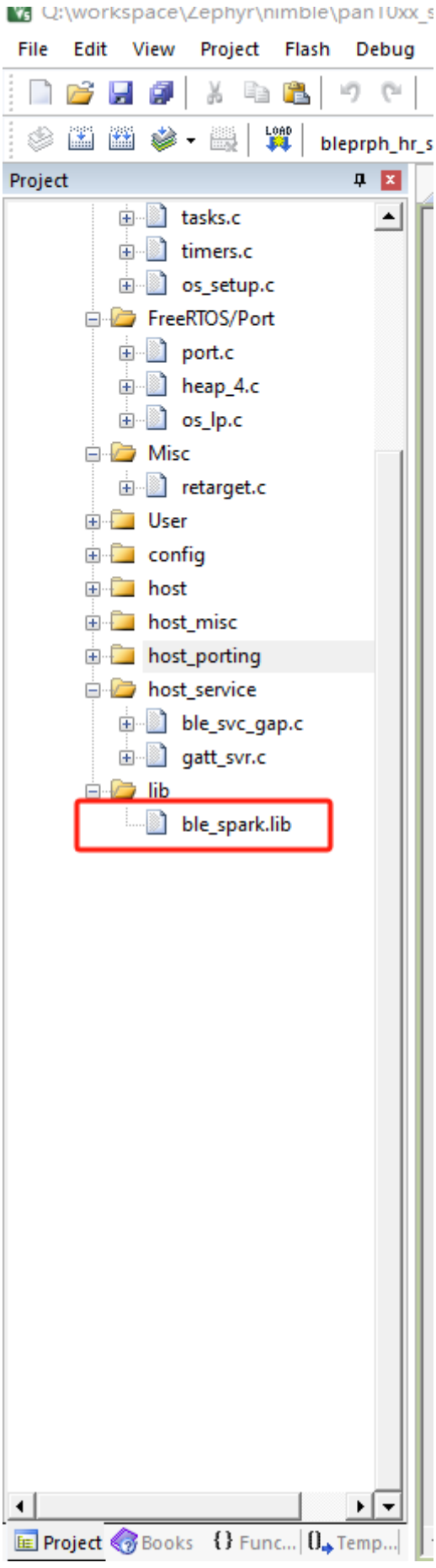


图 3: lib configuration

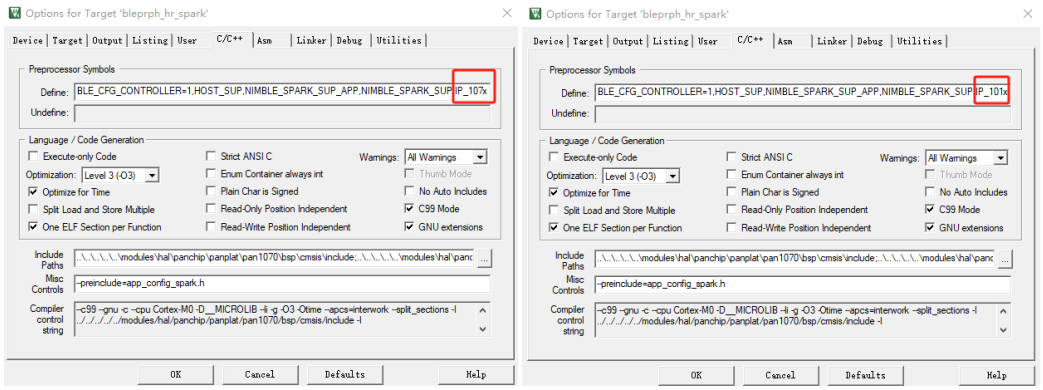


图 4: chip macro

4.2.1 1 基础指标

1.1 功耗

蓝牙在不同的工作模式下功耗如下表所示：

测试条件：

- 基于例程 `nimble\samples\bleprph_hr`
- 发射功率：0 dBm，广播数据：11 Bytes，PAN1070 和 PAN1010 发射功率和广播数据一致；

测试配置:`CONFIG_SOC_DCDC_PAN1070,CONFIG_PM_ENABLE,CONFIG_LOW_SPEED_CLOCK_SRC` 测试选项：`LOW_POWER_TESET_CI_100MS` 和 `LOW_POWER_TESET_CI_1000MS`

工作模式	tx power (dbm)	模式	休眠时钟	峰值电流 (mA)	休眠电流(uA)	平均电流 (uA, 100ms)	平均电流 (uA, 1000ms)
蓝牙广播	0	DCDC	XTL	6.41	4.36	/	12.2
			RCL	5.77	3.84	/	12.4
		LDO	XTL	12.74	4.2	/	20.9
			RCL	11.4	3.78	/	21
蓝牙连接		DCDC	XTL	5.68	4.32	/	8.8
			RCL	5.72	3.83	/	14
		LDO	XTL	11.12	4.19	/	13.1
			RCL	11.38	3.77	/	22
蓝牙广播	7	DCDC	XTL	12.65	4.28	/	19
			RCL	12.75	3.78	/	19
		LDO	XTL	26.42	4.29	/	30.5
			RCL	26.32	3.87	/	30
蓝牙连接		DCDC	XTL	12.55	4.28	/	10.4
			RCL	12.46	3.78	/	15.3
		LDO	XTL	25.52	4.3	/	15
			RCL	25.97	3.87	/	24.4

图 5: PAN1070UA1A EVB 核心板功耗测试数据

4.2.2 2 开发流程

工作模式	tx power (dbm)	模式	休眠时钟	峰值电流 (mA)	休眠电流(uA)	平均电流 (uA, 100ms)	平均电流 (uA, 1000ms)
蓝牙广播	0	DCDC	XTL	6.4	4.26	/	12.7
			RCL	6.41	3.84	/	12.5
		LDO	XTL	12.75	4.29	/	21
			RCL	11.4	3.87	/	21.3
蓝牙连接		DCDC	XTL	5.74	4.28	/	9.9
			RCL	5.77	3.85	/	15.4
		LDO	XTL	11.33	4.3	/	16
			RCL	11.28	3.87	/	25

图 6: PAN1070UA1A EVB 核心板配置 latency 功耗测试数据

工作模式	tx power (dbm)	模式	休眠时钟	峰值电流 (mA)	休眠电流(uA)	平均电流 (uA, 100ms)	平均电流 (uA, 1000ms)
蓝牙广播	0	DCDC	XTL	8.56	3.74	/	15
			RCL	8.7	3.59	/	15.7
		LDO	XTL	15.68	3.76	/	24.5
			RCL	15.85	3.48	/	26.1
蓝牙连接		DCDC	XTL	6.85	3.75	/	13
			RCL	6.86	3.55	/	19.6
		LDO	XTL	13.45	3.76	/	18.4
			RCL	13.5	3.48	/	29.7
蓝牙广播	7	DCDC	XTL	16.3	4.28	/	21.7
			RCL	16.05	3.78	/	22.6
		LDO	XTL	34.49	4.29	/	40
			RCL	34.48	3.87	/	41.3
蓝牙连接		DCDC	XTL	16.06	4.28	/	13.4
			RCL	15.9	3.78	/	19.9
		LDO	XTL	33.82	4.3	/	19
			RCL	34.12	3.87	/	30.6

图 7: PAN1010S9FA EVB 核心板功耗测试数据

2.1 确认开发环境

参考 NDK 快速入门指南，确认软硬件开发环境，可以正常的编译、下载和调试 SDK 提供的基础例程。

建议连接板载的 micro USB，通过串口工具监测 Log。

2.2 参考相关例程

蓝牙开发需要了解一些蓝牙协议相关的知识，可以参考蓝牙协议规范，网上也有很多协议的介绍，此处不作为重点。

当前 SDK 中提供了一些蓝牙相关的例程，涵盖了 central、peripheral 等。

在进行蓝牙开发之前，建议先看一下相关的文档，磨刀不误砍柴工，相信这些例程会对你的开发有所帮助。

2.3 了解蓝牙 app 代码的基本框架

2.3.1 app 和 host 初始化

我们以 `bleprph_hr` 为例，app 和 host 的初始化默认都是在 `app_main` 函数中：

```
void app_main(void)
{
    int rc;

    printf("app started\n");

    /** set public address*/
    uint8_t pub_mac[6]={8,2,3,4,5,6};
    db_set_bd_address(pub_mac);

    /* Initialize the NimBLE host configuration */
    ble_hs_cfg.sync_cb = blehr_on_sync;

    ble_npl_callout_init(&blehr_tx_timer, (struct ble_npl_eventq *)nimble_port_get_dflt_
↪eventq(),
                         blehr_tx_hrate, NULL);

    rc = gatt_svr_init();
    assert(rc == 0);

    /* Set the default device name */
    rc = ble_svc_gap_device_name_set(device_name);
    assert(rc == 0);

        hs_thread_init();
}
```

从这个初始化我们可以将真正需要的初始化切割出来：

- `ble_hs_cfg.sync_cb = blehr_on_sync;` 这个是 host 初始化完成后的回调函数，一般是将需要的**自定义广播函数**的回调添加此处。
- `gatt_svr_init()` GATT **服务**初始化。
- `hs_thread_init` host **协议栈**的初始化。
- `blehr_gap_event` 蓝牙状态事件的处理，比如广播，连接，断连等，可以关注下 `blehr_gap_event` 是在广播启动函数中 `ble_gap_adv_start` 注册的。注意：对于**主机**是在扫描启动函数中 `ble_gap_disc` 注册的。

其实一般来说，一个蓝牙工程有广播，GATT 服务，蓝牙事件处理，基本就搭起一个蓝牙应用的框架了，我们再由此进行展开。

同时 ble_hs_cfg 是一个全局变量，很多关键回调函数和状态依赖它：

```
/** @brief Bluetooth Host main configuration structure
 *
 * Those can be used by application to configure stack.
 *
 * The only reason Security Manager (sm_ members) is configurable at runtime is
 * to simplify security testing. Defaults for those are configured by selecting
 * proper options in application's syscfg.
 */
struct ble_hs_cfg {
    /**
     * An optional callback that gets executed upon registration of each GATT
     * resource (service, characteristic, or descriptor).
     */
    ble_gatt_register_fn *gatts_register_cb;

    /**
     * An optional argument that gets passed to the GATT registration
     * callback.
     */
    void *gatts_register_arg;

    /** Security Manager Local Input Output Capabilities */
    uint8_t sm_io_cap;

    /** @brief Security Manager OOB flag
     *
     * If set proper flag in Pairing Request/Response will be set.
     */
    unsigned sm_oob_data_flag:1;

    /** @brief Security Manager Bond flag
     *
     * If set proper flag in Pairing Request/Response will be set. This results
     * in storing keys distributed during bonding.
     */
    unsigned sm_bonding:1;

    /** @brief Security Manager MITM flag
     *
     * If set proper flag in Pairing Request/Response will be set. This results
     * in requiring Man-In-The-Middle protection when pairing.
     */
    unsigned sm_mitm:1;

    /** @brief Security Manager Secure Connections flag
     *
     * If set proper flag in Pairing Request/Response will be set. This results
     * in using LE Secure Connections for pairing if also supported by remote
     * device. Fallback to legacy pairing if not supported by remote.
     */
    unsigned sm_sc:1;

    /** @brief Security Manager Key Press Notification flag
     *
     * Currently unsupported and should not be set.
     */
    unsigned sm_keypress:1;

    /** @brief Security Manager Local Key Distribution Mask */
    uint8_t sm_our_key_dist;
```

(下页继续)

(续上页)

```
    /** @brief Security Manager Remote Key Distribution Mask */
    uint8_t sm_their_key_dist;

    /** @brief Stack reset callback
     *
     * This callback is executed when the host resets itself and the controller
     * due to fatal error.
     */
    ble_hs_reset_fn *reset_cb;

    /** @brief Stack sync callback
     *
     * This callback is executed when the host and controller become synced.
     * This happens at startup and after a reset.
     */
    ble_hs_sync_fn *sync_cb;

    /* XXX: These need to go away. Instead, the nimble host package should
     * require the host-store API (not yet implemented)..
     */
    /** Storage Read callback handles read of security material */
    ble_store_read_fn *store_read_cb;

    /** Storage Write callback handles write of security material */
    ble_store_write_fn *store_write_cb;

    /** Storage Delete callback handles deletion of security material */
    ble_store_delete_fn *store_delete_cb;

    /** @brief Storage Status callback.
     *
     * This callback gets executed when a persistence operation cannot be
     * performed or a persistence failure is imminent. For example, if is
     * insufficient storage capacity for a record to be persisted, this
     * function gets called to give the application the opportunity to make
     * room.
     */
    ble_store_status_fn *store_status_cb;

    /** An optional argument that gets passed to the storage status callback. */
    void *store_status_arg;
};
```

2.3.2 蓝牙广播或者扫描　广播函数:

```
static void
blehr_advertise(void)
{
    struct ble_gap_adv_params adv_params;
    struct ble_hs_adv_fields fields;
    int rc;

    /*
     *  Set the advertisement data included in our advertisements:
     *     o Flags (indicates advertisement type and other general info)
     *     o Advertising tx power
     *     o Device name
     */
    memset(&fields, 0, sizeof(fields));
```

(下页继续)

(续上页)

```
    /*
     * Advertise two flags:
     *      o Discoverability in forthcoming advertisement (general)
     *      o BLE-only (BR/EDR unsupported)
     */
    fields.flags = BLE_HS_ADV_F_DISC_GEN |
                   BLE_HS_ADV_F_BREDR_UNSUP;


    fields.name = (uint8_t *)device_name;
    fields.name_len = strlen(device_name);
    fields.name_is_complete = 1;

    rc = ble_gap_adv_set_fields(&fields);
    if (rc != 0) {

        printf("error setting advertisement data; rc=%d\n", rc);
        return;
    }

    /* Begin advertising */
    memset(&adv_params, 0, sizeof(adv_params));
    adv_params.conn_mode = BLE_GAP_CONN_MODE_UND;
    adv_params.disc_mode = BLE_GAP_DISC_MODE_GEN;

    #if LOW_POWER_TESET_CI_100MS || LOW_POWER_TESET_LATENCY_100MS
    adv_params.itvl_min = BLE_GAP_ADV_ITVL_MS(100);
    adv_params.itvl_max = BLE_GAP_ADV_ITVL_MS(100);
    #endif

    #if LOW_POWER_TESET_CI_1000MS || LOW_POWER_TESET_LATENCY_1000MS
    adv_params.itvl_min = BLE_GAP_ADV_ITVL_MS(1000);
    adv_params.itvl_max = BLE_GAP_ADV_ITVL_MS(1000);
    #endif

    rc = ble_gap_adv_start(blehr_addr_type, NULL, BLE_HS_FOREVER,
                           &adv_params, blehr_gap_event, NULL);
    if (rc != 0) {
        printf("error enabling advertisement; rc=%d\n", rc);
        return;
    }
}
```

扫描函数:

```
/**
 * Initiates the GAP general discovery procedure.
 */
static void
blecent_scan(void)
{
    uint8_t own_addr_type;
    struct ble_gap_disc_params disc_params;
    int rc;

    /* Figure out address to use while advertising (no privacy for now) */
    rc = ble_hs_id_infer_auto(0, &own_addr_type);
    if (rc != 0) {
```

(下页继续)

(续上页)

```
        printf("error determining address type; rc=%d\n", rc);
        return;
    }

    /* Tell the controller to filter duplicates; we don't want to process
     * repeated advertisements from the same device.
     */
    disc_params.filter_duplicates = 0;

    /**
     * Perform a passive scan.  I.e., don't send follow-up scan requests to
     * each advertiser.
     */
    disc_params.passive = 1;

    /* Use defaults for the rest of the parameters. */
    disc_params.itvl = 60;
    disc_params.window = 50;
    disc_params.filter_policy = 0;
    disc_params.limited = 0;

    rc = ble_gap_disc(own_addr_type, BLE_HS_FOREVER, &disc_params,
                      blecent_gap_event, NULL);
    if (rc != 0) {
        printf("Error initiating GAP discovery procedure; rc=%d\n",
                    rc);
    }
}
```

在初始化状态这两个函数最终会注册到 `ble_hs_cfg.sync_cb`。

2.3.3 GATT 服务初始化

对于 GATT 服务初始化，我们看以下一段代码：

```
int gatt_svr_init(void)
{
    int rc;

    rc = ble_gatts_count_cfg(gatt_svr_svcs);
    if (rc != 0) {
        return rc;
    }

    rc = ble_gatts_add_svcs(gatt_svr_svcs);
    if (rc != 0) {
        return rc;
    }

    return 0;
}
```

`ble_gatts_count_cfg` 获取 GATT config 描述个数和 `ble_gatts_add_svcs``注册 GATT 服务这两个函数都会涉及到 gatt_svr_svcs 这个变量，跳转过去我们发现 gatt_svr_svcs 真正的定义 GATT 服务的数据库，我们可以在这个变量中自定义实现 GATT 服务。

```
static const struct ble_gatt_svc_def gatt_svr_svcs[] = {
    {
        /* Service: Heart-rate */
        .type = BLE_GATT_SVC_TYPE_PRIMARY,
        .uuid = BLE_UUID16_DECLARE(GATT_HRS_UUID),
        .characteristics = (struct ble_gatt_chr_def[]) { {
```

(下页继续)

(续上页)

```
            /* Characteristic: Heart-rate measurement */
            .uuid = BLE_UUID16_DECLARE(GATT_HRS_MEASUREMENT_UUID),/* 声明蓝牙 GATT 服务 */
            .access_cb = gatt_svr_chr_access_heart_rate,
            .val_handle = &hrs_hrm_handle,
            .flags = BLE_GATT_CHR_F_NOTIFY,
        }, {
            /* Characteristic: Body sensor location */
            .uuid = BLE_UUID16_DECLARE(GATT_HRS_BODY_SENSOR_LOC_UUID),
            .access_cb = gatt_svr_chr_access_heart_rate,
            .flags = BLE_GATT_CHR_F_READ,
        }, {
            0, /* No more characteristics in this service */
        }, }
    },

    {
        /* Service: Device Information */
        .type = BLE_GATT_SVC_TYPE_PRIMARY,
        .uuid = BLE_UUID16_DECLARE(GATT_DEVICE_INFO_UUID),
        .characteristics = (struct ble_gatt_chr_def[]) { {
            /* Characteristic: * Manufacturer name */
            .uuid = BLE_UUID16_DECLARE(GATT_MANUFACTURER_NAME_UUID),
            .access_cb = gatt_svr_chr_access_device_info,
            .flags = BLE_GATT_CHR_F_READ,
        }, {
            /* Characteristic: Model number string */
            .uuid = BLE_UUID16_DECLARE(GATT_MODEL_NUMBER_UUID),
            .access_cb = gatt_svr_chr_access_device_info,
            .flags = BLE_GATT_CHR_F_READ,
        }, {
            0, /* No more characteristics in this service */
        }, }
    },

        {
            0, /* No more services */
        },
};
```

对于心跳服务访问 gatt_svr_chr_access_heart_rate 这个函数，实现相对简单。我们可以参考 bleprph_enc 例程中的 gatt_svc_access 函数,，它对**特性的读，写，读描述符，写描述符**等等做了不同的分类和实现：

```
static int
gatt_svc_access(uint16_t conn_handle, uint16_t attr_handle,
                struct ble_gatt_access_ctxt *ctxt, void *arg)
{
    const ble_uuid_t *uuid;
    int rc;

    switch (ctxt->op) {
    case BLE_GATT_ACCESS_OP_READ_CHR:
        if (conn_handle != BLE_HS_CONN_HANDLE_NONE) {
            MODLOG_DFLT("Characteristic read; conn_handle=%d attr_handle=%d\n",
                        conn_handle, attr_handle);
        } else {
            MODLOG_DFLT("Characteristic read by NimBLE stack; attr_handle=%d\n",
                        attr_handle);
        }
        uuid = ctxt->chr->uuid;
        if (attr_handle == gatt_svr_chr_val_handle) {
```

(下页继续)

(续上页)

```
            rc = os_mbuf_append(ctxt->om,
                                &gatt_svr_chr_val,
                                sizeof(gatt_svr_chr_val));
            return rc == 0 ? 0 : BLE_ATT_ERR_INSUFFICIENT_RES;
        }
        goto unknown;

    case BLE_GATT_ACCESS_OP_WRITE_CHR:
        if (conn_handle != BLE_HS_CONN_HANDLE_NONE) {
            MODLOG_DFLT("Characteristic write; conn_handle=%d attr_handle=%d",
                        conn_handle, attr_handle);
        } else {
            MODLOG_DFLT("Characteristic write by NimBLE stack; attr_handle=%d",
                        attr_handle);
        }
        uuid = ctxt->chr->uuid;
        if (attr_handle == gatt_svr_chr_val_handle) {
            rc = gatt_svr_write(ctxt->om,
                                sizeof(gatt_svr_chr_val),
                                sizeof(gatt_svr_chr_val),
                                &gatt_svr_chr_val, NULL);
            ble_gatts_chr_updated(attr_handle);
            MODLOG_DFLT("Notification/Indication scheduled for "
                        "all subscribed peers.\n");
            return rc;
        }
        goto unknown;

    case BLE_GATT_ACCESS_OP_READ_DSC:
        if (conn_handle != BLE_HS_CONN_HANDLE_NONE) {
            MODLOG_DFLT("Descriptor read; conn_handle=%d attr_handle=%d\n",
                        conn_handle, attr_handle);
        } else {
            MODLOG_DFLT("Descriptor read by NimBLE stack; attr_handle=%d\n",
                        attr_handle);
        }
        uuid = ctxt->dsc->uuid;
        if (ble_uuid_cmp(uuid, &gatt_svr_dsc_uuid.u) == 0) {
            rc = os_mbuf_append(ctxt->om,
                                &gatt_svr_dsc_val,
                                sizeof(gatt_svr_chr_val));
            return rc == 0 ? 0 : BLE_ATT_ERR_INSUFFICIENT_RES;
        }
        goto unknown;

    case BLE_GATT_ACCESS_OP_WRITE_DSC:
        goto unknown;

    default:
        goto unknown;
    }

unknown:
    /* Unknown characteristic/descriptor;
     * The NimBLE host should not have called this function;
     */
    assert(0);
    return BLE_ATT_ERR_UNLIKELY;
}
```

我们可以参考这个函数对特性的读写做一些通用的实现。

2.3.4 GAP 事件处理　从上文我们可以知道，GAP 事件函数我们可以注册到广播或者扫描函数中，我们也可以对不同的事件进行分类处理，例如连接，断连，配对事件，订阅事件，扫描收到的广播包等等：

支持的事件如下：

```
#define BLE_GAP_EVENT_CONNECT               0
#define BLE_GAP_EVENT_DISCONNECT            1
/* Reserved                                 2 */
#define BLE_GAP_EVENT_CONN_UPDATE           3
#define BLE_GAP_EVENT_CONN_UPDATE_REQ       4
#define BLE_GAP_EVENT_L2CAP_UPDATE_REQ      5
#define BLE_GAP_EVENT_TERM_FAILURE          6
#define BLE_GAP_EVENT_DISC                  7
#define BLE_GAP_EVENT_DISC_COMPLETE         8
#define BLE_GAP_EVENT_ADV_COMPLETE          9
#define BLE_GAP_EVENT_ENC_CHANGE            10
#define BLE_GAP_EVENT_PASSKEY_ACTION        11
#define BLE_GAP_EVENT_NOTIFY_RX             12
#define BLE_GAP_EVENT_NOTIFY_TX             13
#define BLE_GAP_EVENT_SUBSCRIBE             14
#define BLE_GAP_EVENT_MTU                   15
#define BLE_GAP_EVENT_IDENTITY_RESOLVED     16
#define BLE_GAP_EVENT_REPEAT_PAIRING        17
#define BLE_GAP_EVENT_PHY_UPDATE_COMPLETE   18
#define BLE_GAP_EVENT_EXT_DISC              19
#define BLE_GAP_EVENT_PERIODIC_SYNC         20
#define BLE_GAP_EVENT_PERIODIC_REPORT       21
#define BLE_GAP_EVENT_PERIODIC_SYNC_LOST    22
#define BLE_GAP_EVENT_SCAN_REQ_RCVD         23
#define BLE_GAP_EVENT_PERIODIC_TRANSFER     24
#define BLE_GAP_EVENT_PATHLOSS_THRESHOLD    25
#define BLE_GAP_EVENT_TRANSMIT_POWER        26
```

```
static int blehr_gap_event(struct ble_gap_event *event, void *arg)
{
    switch (event->type) {
    case BLE_GAP_EVENT_CONNECT:
        /* A new connection was established or a connection attempt failed */
        printf("connection %s; status=%d\n",
                    event->connect.status == 0 ? "established" : "failed",
                    event->connect.status);

        if (event->connect.status != 0) {
            /* Connection failed; resume advertising */
            blehr_advertise();
            conn_handle = 0;
        }
        else {
          conn_handle = event->connect.conn_handle;
          #if LOW_POWER_TESET_CI_100MS || LOW_POWER_TESET_CI_1000MS || LOW_POWER_TESET_LATENCY_
↪100MS || LOW_POWER_TESET_LATENCY_1000MS
            LowPower_Test_Timer();
          #endif
        }

        break;

    case BLE_GAP_EVENT_DISCONNECT:
        printf("disconnect; reason=0x%02x\n",  (uint8_t)event->disconnect.reason);
        conn_handle = BLE_HS_CONN_HANDLE_NONE; /* reset conn_handle */
```

(下页继续)

(续上页)

```
        /* Connection terminated; resume advertising */
        blehr_advertise();
        break;

    case BLE_GAP_EVENT_ADV_COMPLETE:
        printf("adv complete\n");
        blehr_advertise();
        break;

    case BLE_GAP_EVENT_SUBSCRIBE:
        printf("subscribe event; cur_notify=%d\n value handle; "
                       "val_handle=%d\n",
                    event->subscribe.cur_notify, hrs_hrm_handle);
        if (event->subscribe.attr_handle == hrs_hrm_handle) {
            notify_state = event->subscribe.cur_notify;
            blehr_tx_hrate_reset();
        } else if (event->subscribe.attr_handle != hrs_hrm_handle) {
            notify_state = event->subscribe.cur_notify;
            blehr_tx_hrate_stop();
        }
        break;

    case BLE_GAP_EVENT_MTU:
        printf("mtu update event; conn_handle=%d mtu=%d\n",
                    event->mtu.conn_handle,
                    event->mtu.value);

        break;

    }

    return 0;
}
```

以下为主机中的 gap 事件处理：

```
static int blecent_gap_event(struct ble_gap_event *event, void *arg)
{
    struct ble_gap_conn_desc desc;
    struct ble_hs_adv_fields fields;
    int rc;

    switch (event->type) {
    case BLE_GAP_EVENT_DISC:
        rc = ble_hs_adv_parse_fields(&fields, event->disc.data,
                                     event->disc.length_data);
        if (rc != 0) {
            return 0;
        }

        /* An advertisment report was received during GAP discovery. */
        print_adv_fields(&fields);

        /* Try to connect to the advertiser if it looks interesting. */
        blecent_connect_if_interesting(&event->disc);
        return 0;

    case BLE_GAP_EVENT_CONNECT:
        /* A new connection was established or a connection attempt failed. */
        if (event->connect.status == 0) {
            /* Connection successfully established. */
```

(下页继续)

(续上页)

```
            printf("Connection established ");

            rc = ble_gap_conn_find(event->connect.conn_handle, &desc);
            assert(rc == 0);
            print_conn_desc(&desc);
            printf("\n");

            /* Remember peer. */
            rc = peer_add(event->connect.conn_handle);
            if (rc != 0) {
                printf("Failed to add peer; rc=%d\n", rc);
                return 0;
            }

            /* Perform service discovery. */
            rc = peer_disc_all(event->connect.conn_handle,
                               blecent_on_disc_complete, NULL);
            if (rc != 0) {
                printf("Failed to discover services; rc=%d\n", rc);
                return 0;
            }
        } else {
            /* Connection attempt failed; resume scanning. */
            printf("Error: Connection failed; status=%d\n",
                        event->connect.status);
            blecent_scan();
        }

        return 0;

    case BLE_GAP_EVENT_DISCONNECT:
        /* Connection terminated. */
        printf("disconnect; reason=0x%02x\n",  (uint8_t)event->disconnect.reason);
        print_conn_desc(&event->disconnect.conn);
        printf("\n");

        /* Forget about peer. */
        peer_delete(event->disconnect.conn.conn_handle);

        /* Resume scanning. */
        blecent_scan();
        return 0;

    case BLE_GAP_EVENT_DISC_COMPLETE:
        printf("discovery complete; reason=%d\n",
                    event->disc_complete.reason);
        return 0;

    case BLE_GAP_EVENT_ENC_CHANGE:
        /* Encryption has been enabled or disabled for this connection. */
        printf("encryption change event; status=%d ",
                    event->enc_change.status);
        rc = ble_gap_conn_find(event->enc_change.conn_handle, &desc);
        assert(rc == 0);
        print_conn_desc(&desc);
        return 0;

    case BLE_GAP_EVENT_NOTIFY_RX:
        /* Peer sent us a notification or indication. */
        printf("received %s; conn_handle=%d attr_handle=%d "
                        "attr_len=%d\n",
```

(下页继续)

(续上页)

```
                    event->notify_rx.indication ?
                        "indication" :
                        "notification",
                    event->notify_rx.conn_handle,
                    event->notify_rx.attr_handle,
                    OS_MBUF_PKTLEN(event->notify_rx.om));

        /* Attribute data is contained in event->notify_rx.attr_data. */
        return 0;

    case BLE_GAP_EVENT_MTU:
        printf("mtu update event; conn_handle=%d cid=%d mtu=%d\n",
                    event->mtu.conn_handle,
                    event->mtu.channel_id,
                    event->mtu.value);
        return 0;

    case BLE_GAP_EVENT_REPEAT_PAIRING:
        /* We already have a bond with the peer, but it is attempting to
         * establish a new secure link.  This app sacrifices security for
         * convenience: just throw away the old bond and accept the new link.
         */

        /* Delete the old bond. */
        rc = ble_gap_conn_find(event->repeat_pairing.conn_handle, &desc);
        assert(rc == 0);
        ble_store_util_delete_peer(&desc.peer_id_addr);

        /* Return BLE_GAP_REPEAT_PAIRING_RETRY to indicate that the host should
         * continue with the pairing operation.
         */
        return BLE_GAP_REPEAT_PAIRING_RETRY;

    default:
        return 0;
    }
}
```

4.3 NDK 低功耗开发指南

本文主要通过一些示例，介绍**低功耗**各个模式、使用的方法以及可能遇到的问题。

4.3.1 1 低功耗模式

低功耗模式介绍如下表所示：

模式名称	进入	唤醒	1.2V 区时钟	时钟	1.2V 供电
STANDBY_M0	Sleep_mode = 3, Buck_en_ctrl=0, Buck_bp_ctrl=0, Flashldo_bp_en_ctrl =0, Flashldo_lp_en_en_ctrl =0, WFI	P00, P01, P02, BOD/LVR (可选, 需保证慢时钟开启), PIN RESET	全部关闭	全部关闭	断电
STANDBY_M1	Sleep_mode = 2, ldo_pwr_ctrl = 0, ldol_ pwr ctrl = 0/1, cpu pwr ctrl = 0/1, sram0/1 pwr ctrl = 0/1, ll_ram pwr ctrl=0/1, phy_ram pwr _ctrl=0/1, Buck_en_ctrl=0, Buck_bp_ctrl=0, Flashldo_bp_en_ctrl =0, Flashldo_lp_en_en_ctrl =0, WFI	所有 GPIO (边沿去抖), SLPTMR, WDT, BOD/LVR (可选), PIN RESET	CLK32K, 其他全部关闭	慢时钟	LPLDOL/H : LL_RAM (可选), PHY_RAM/PHY_REGS (可选), SRAM0/1 (可选), decrypt_ram(可选, cpu 模块不保电, 没办法做到只保少部分寄存器) LPLDOL/H: asnactrl_l (rcc 的寄存器) GPIOWDTBOD, LVR
DEEPSLEEP	Sleep_mode = 1, ldol_power_ctrl = 0/1, Buck_en_ctrl=0, Buck_bp_ctrl=0, Flashldo_bp_en_ctrl =0/1, Flashldo_lp_en_en_ctrl =1/0, WFI	所有 GPIO, SLPTMR, WDT, TIMER0/1/2, BOD/LVR (可选), PIN RESET	CLK32K, 其他全部关闭	慢时钟	LPLDOL/H: 其他数字模块, LPLDOL/H: WakeupGPIO, WDT, Timer0/1/2, (需要测试, 确认如何安全使用)
SLEEP	Sleep_mode = 0, WFI	所有外设中断, BOD/LVR (可选), PIN RESET	CLK32K, CPU_CLK 关闭, RCH、XTH、DPLL 根据软件配置选择打开	慢时钟快时钟	HP_LDO 供电

4.3.2 2 开发流程

2.1 低功耗使用流程

2.1.1 Sleep 模式

- **进入流程:**

1. 配置 sleep_mode 为 sleep 模式
2. 唤醒源配置
3. 设置 flash dp_en 为 0
4. 设置 CPU SLEEPDEEP 寄存器为 0; 地址: 0xE000ED10
5. 考虑安全, 建议进行一次手动 3V 同步操作
6. ___WFI();

- **退出流程：**

1. 唤醒源产生中断；
2. 处理中断，清除中断源；

- **备注：**

1. 支持 m0 调试模式，不支持 riscv 调试模式

2.1.2 Deepsleep 模式　进入流程：

1. 配置 sleep_mode 为 deepsleep 模式，配置各电压域的 power switch，配置 rcl_en_ctrl，xtl_pwr_ctrl，Buck 和 flashldo 的配置建议使用 driver 默认

2.

Power Switch	模式 1（推荐）	模式 2	模式 3
lpldoh_en	1	1	0
lpldol_en	1	0	1
Ldo_pwr_ctrl	1	1	1
Ldol_pwr_ctrl	0	1	1
Peri_pwr_ctrl(cpu/ll_sram/phy_sram/sram0/sram1)	1（ram 可配）	1（ram 可配）	1（ram 可配）
Lpldoh_iso_en	1（配置 0 待测试）	0	0

3. 唤醒源配置：所有 GPIO，SLPTMR，WDT，TIMER0/1/2，BOD/LVR（可选），PIN RESET。对于模式 1，如果 Lpldoh_iso_en 配置为 1，则不支持 TIMER0/1/2 唤醒；如果 Lpldoh_iso_en 配置为 0，则支持 TIMER0/1/2 唤醒和 PWM 输出（需要测试是否有漏电）。对于模式 2 或者模式 3，上述唤醒源都可以唤醒系统。BOD，LVR 唤醒需要开启 32K 时钟。
4. 对于模式 1，如果 Lpldoh_iso_en 配置为 1，不支持 PWM 输出；如果 Lpldoh_iso_en 配置为 0，则支持 PWM 输出（需要测试是否有漏电）。对于模式 2 或者模式 3，PWM 可以输出
5. Flash dp 设置为 1，并退出 enhance 模式。配置合适的 dp，和 rdp 时间
6. 建议 cpu 地址重映射功能开启，映射地址为 ram 保电区域，可加快唤醒后，程序执行速度
7. Dly_time2 需要根据测试结果重新配置，默认值比较大
8. 设置 CPU SLEEPDEEP 寄存器为 1；地址：0xE000ED10
9. Flash 控制器退出增强型模式；
10. 考虑安全，建议进行一次手动 3V 同步操作
11. _WFI()；

退出流程：

1. 唤醒源唤醒，产生 lp 中断或者唤醒源相对应的中断
2. 清除相应 flag

备注：

1. 支持 m0 调试模式（低功耗期间会丢失），不支持 riscv 调试模式
2. gpio 唤醒电平，至少需要保持一个完整的 32K 时钟周期。如果需要读取的 gpio 的中断，需要 7 个 32K 时钟周期（需要 dly2 的延时决定）；如果 32K 时钟关闭，则唤醒需要的时间更久，和 32K 时钟的启动时间相关

2.1.3 Standby_m1 模式 进入流程：

1. 配置 sleep_mode 为 standby_m1 模式，配置各电压域的 power switch，配置 rcl_en_ctrl，xtl_pwr_ctrl，Buck 和 flashldo 的配置建议使用 driver 默认

2.

	模式 1（推荐，需测试）	模式 2	模式 3
lpldoh_en	1	1	0
lpldol_en	1	0	1
Ldo_pwr_ctrl	0	0	0
Ldol_pwr_ctrl	0	1	1
Peri_pwr_ctrl(cpu/ll_sram/phy_sram/sram0/sram1)	1(可配)	1(可配)	1(可配)
Lpldoh_iso_en	1	1	1

3. 唤醒源配置：所有 GPIO，SLPTMR，WDT，BOD/LVR（可选），PIN RESET。BOD，LVR 唤醒需要开启 32K 时钟。
4. flash 如果使用 4 线模式，建议开启 dp 模式，flash 两线切换四线的时间特别长，一般会有 8ms；如果 flash 使用 2 线模式，不建议开启 dp 模式，flash 直接掉电处理。
5. 建议 cpu 地址重映射功能开启，映射地址为 ram 保电区域，可加快唤醒后，程序执行速度
6. 根据需求，决定是否开启 cpu 保电功能，寄存器 LP_FL_CTRL[4]。可硬件恢复现场，代码实现有特定需求，参见说明部分
7. Dly_time2 需要根据测试结果重新配置，默认值比较大
8. 设置 CPU SLEEPDEEP 寄存器为 1；地址：0xE000ED10
9. 考虑安全，建议进行一次手动 3V 同步操作
10. __WFI();

退出流程：

1. 唤醒源唤醒，产生 lp 中断以及唤醒源相对应的中断
2. 清除相应 flag

备注：

1. 不支持 m0 和 riscv 调试模式
2. 支持 m0 的 cpu retention 功能（现场恢复），不支持 riscv 的 cpu retention 功能
3. gpio 唤醒电平，至少需要保持一个完整的 32K 时钟周期。如果需要读取的 gpio 的中断，需要 7 个 32K 时钟周期（需要 dly2 的延时决定）；如果 32K 时钟关闭，则唤醒需要的时间更久，和 32K 时钟的启动时间相关

2.1.4 Standby_m0 模式 进入流程：

1. 配置 sleep_mode 为 standby_m0 模式，配置 rcl_en_ctrl，xtl_pwr_ctrl
2. 唤醒源配置：所有 P00，P01，P02，BOD/LVR（可选），PIN RESET。
3. Flash dp 设置为 0
4. Dly_time1 根据需要决定是否配置，如果不在意 m0 的唤醒时间不建议去修改。此处的时间测试遍历会比较多，设置的值不好控制
5. 设置 CPU SLEEPDEEP 寄存器为 1；地址：0xE000ED10
6. 考虑安全，建议进行一次手动 3V 同步操作
7. __WFI();

退出流程：

1. 唤醒源唤醒，产生 lp 中断以及唤醒源相对应的中断
2. 清除相应 flag

2.2 参考相关例程

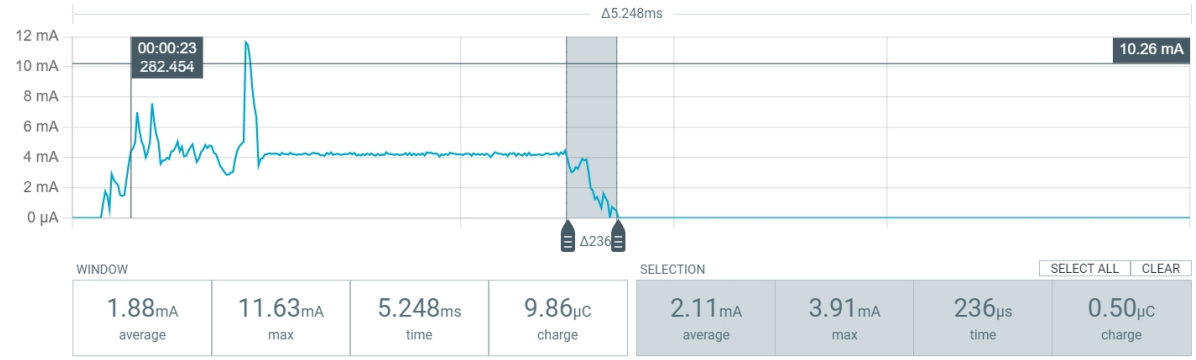
当前 SDK 中提供了一些低功耗相关的例程，涵盖了章节 1 所述的所有低功耗模式等。
例程位置：03_MCU\mcu_samples\LP。

2.3 Standby_m1 休眠唤醒

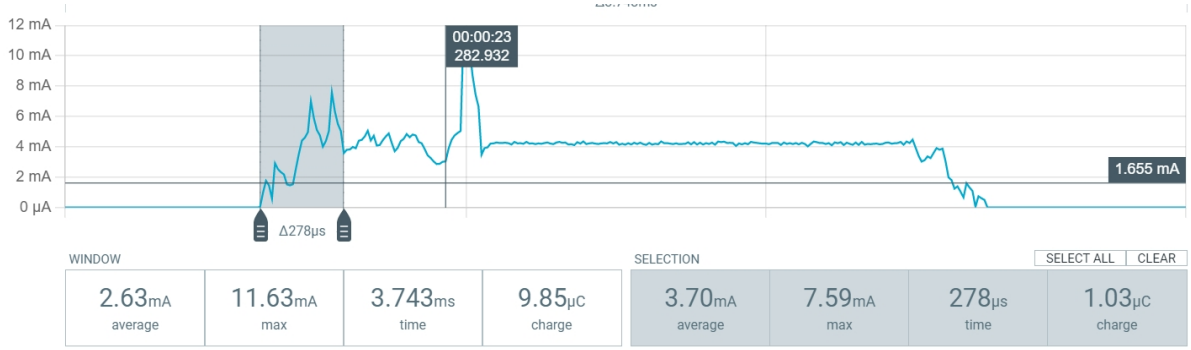
以 standby m1 模式，cpu retention and cpu continue run 模式为基础介绍各个时间阶段 mcu 的动作。

阶段	说明	时 间 (us)	备注
进入休眠	从软件发送休眠指令至硬件完全休眠的时间	236	
硬件唤醒启动	唤醒源触发后硬件完全启动的时间	278	
Standby M1 retention 模式	continue run，唤醒至 rx ready 时间	465	此模式下**软件初始化和** RF **初始化步骤**可以省略
TX/RX(max 59B)	收发最大 payload 的时间，根据传输速率与字节数计算得出	1888	此处按最大数据计算
总计	休眠唤醒至 tx 完成/rx 完成的时间	2867	

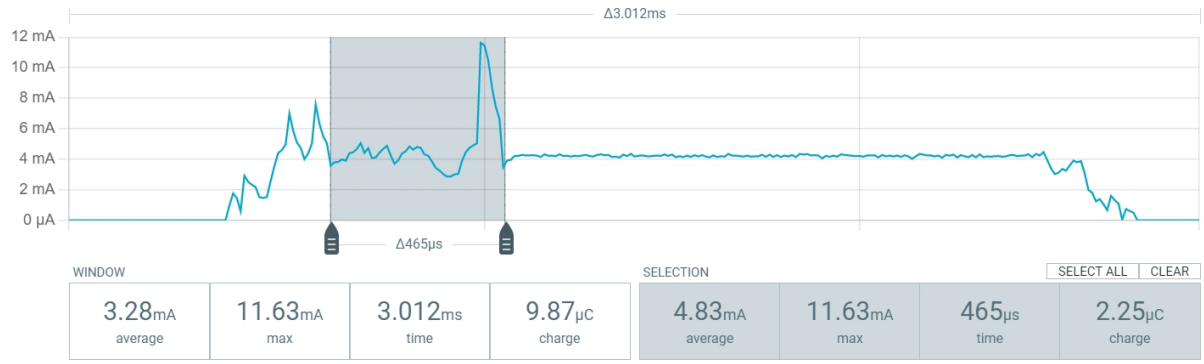
休眠时间



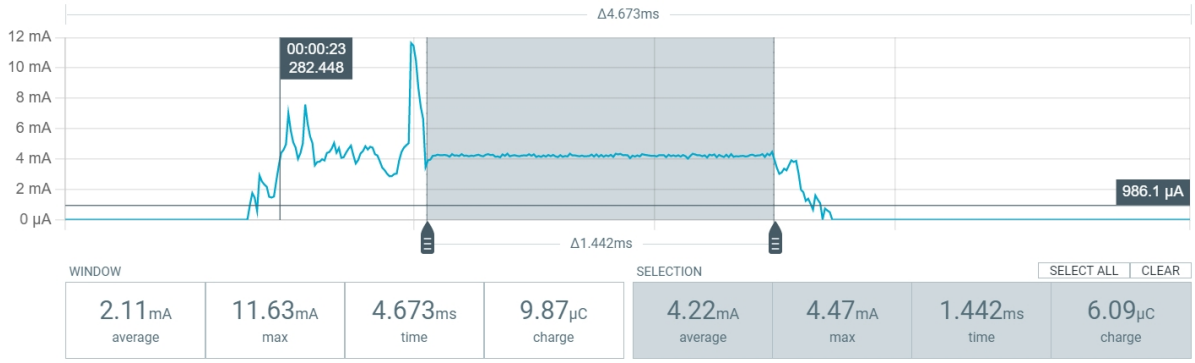
唤醒时间



软件运行至 RF ready 时间



RF 接收 32Byte 时间



4.3.3 3 低功耗注意事项

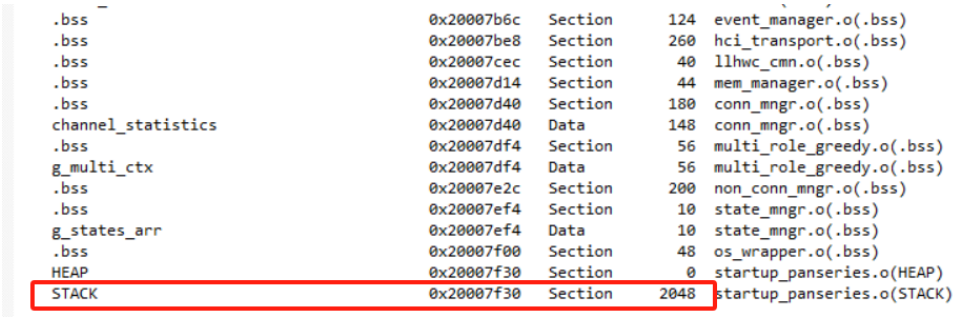
1. 如果 flash 运行在 4 线 enhance 模式，进入功耗前需要退出 enhance 模式。
2. 低功耗模式下供电分两部分 LPLDOL 和 LPLDOH，其中 LPLDOL 给 sram 供电，电压范围从 0.4~0.9v（未校准芯片有差异），LPLDOH 给 always on 区域部分供电，供电范围 0.5~1.2v（未校准芯片有差异），通常在进入低功耗在保证唤醒正常情况下尽量降低两个电压，常温下 LPLDOL/H 可设置未 0/1。
3. 为防止 LPLDOH 在电源抖动时出现不能唤醒的情况，增加了一个 LPLDOH_VREF_TRIM_AON（LP_LPLDO[23:21]）控制位，设置值的作用是弥补电源抖动，稳定电压，同时设置完成此值（例如设置为 2，LPLDOH 电压 0.7v）后再想拉低 LPLDOH 至 0.6v 是不能完成的，此属于正常现象。
4. DeepSleep 模式下外设 timer0/1/2 作为唤醒以及 PWM 在低功耗下输出需要将 deepsleep 低功耗模式设置为模式 2，即 dp_mode= LP_DEEPSLEEP_MODE2.
5. Standby M1 cpu retention 模式唤醒后外设部分及 RF MAC 层寄存器需要重新初始化，PHY、保电的 sram、时钟等不需要重新初始化

4.4 NDK RAM 使用情况分析以及优化指南

4.4.1 1 如何查看 KEIL 的 RAM 和 Flash 使用情况

因为当前工程中很多和 BLE 时间处理相关的函数为了提升处理速度使用了 RAM CODE，所以导致部分 CODE 会占用 RAM 空间，所以查看工程的实际占用空间要从 RAM 和 Flash 空间进行查看。

首先双击 keil 项目打开 map 文件



```
.bss                        0x20007b6c   Section      124  event_manager.o(.bss)
.bss                        0x20007be8   Section      260  hci_transport.o(.bss)
.bss                        0x20007cec   Section       40  llhwc_cmn.o(.bss)
.bss                        0x20007d14   Section       44  mem_manager.o(.bss)
.bss                        0x20007d40   Section      180  conn_mngr.o(.bss)
channel_statistics          0x20007d40   Data         148  conn_mngr.o(.bss)
.bss                        0x20007df4   Section       56  multi_role_greedy.o(.bss)
g_multi_ctx                 0x20007df4   Data          56  multi_role_greedy.o(.bss)
.bss                        0x20007e2c   Section      200  non_conn_mngr.o(.bss)
.bss                        0x20007ef4   Section       10  state_mngr.o(.bss)
g_states_arr                0x20007ef4   Data          10  state_mngr.o(.bss)
.bss                        0x20007f00   Section       48  os_wrapper.o(.bss)
HEAP                        0x20007f30   Section        0  startup_panseries.o(HEAP)
STACK                       0x20007f30   Section     2048  startup_panseries.o(STACK)
```

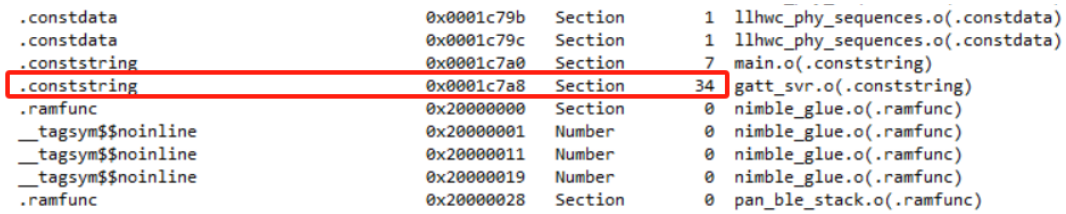
图 8: 通过 STACK 查看 RAM 占用

1.1 如何查看 RAM 空间:

因为 STACK 占用 RAM 边界位置，我们可以通过 STACK 占用可以知道 RAM 最多占用 `0x20007f30+0x800(2048) = 0x20008730` 的 RAM，因为 RAM 地址默认是 `0x20000000` 起始的，所以我们知道 RAM 占用了 `0x8730（34608）`字节的 RAM。

1.2 如何查看 Flash 空间

从 RAM 空间往上查找 flash 边界位置，0x20000000 前面的即是 flash 最大地址



```
.constdata                  0x0001c79b   Section        1  llhwc_phy_sequences.o(.constdata)
.constdata                  0x0001c79c   Section        1  llhwc_phy_sequences.o(.constdata)
.conststring                0x0001c7a0   Section        7  main.o(.conststring)
.conststring                0x0001c7a8   Section       34  gatt_svr.o(.conststring)
.ramfunc                    0x20000000   Section        0  nimble_glue.o(.ramfunc)
__tagsym$$noinline          0x20000001   Number         0  nimble_glue.o(.ramfunc)
__tagsym$$noinline          0x20000011   Number         0  nimble_glue.o(.ramfunc)
__tagsym$$noinline          0x20000019   Number         0  nimble_glue.o(.ramfunc)
.ramfunc                    0x20000028   Section        0  pan_ble_stack.o(.ramfunc)
```

图 9: 查看 flash 边界

通过 map 文件找到边界地址 0x1c7a8+0x22(34) = 0x1c7ca(116682) bytes。

4.4.2 2 关于堆空间的使用说明

2.1 蓝牙 controller 的堆

蓝牙 controller 的堆默认是用于 controller 的广播，连接等各种数据包动态分配使用的。而且是从 host 定义来进行分配的，但是不同的参数可能导致堆的需求空间不一致。

我们可以通过打开 `app_config.h` 或者 `app_config_spark.h` 中的 `BT controller Memory Pool usage print` 选项（对应的宏 `CONFIG_CNTRL_MEM_POOL_PRINT`）显示的输出底层 controller 所需要的内存。

正常分配时如下:

```
[19:32:53.628]收 ←LL Controller Version:bd5923c

[19:32:53.665]收 ←BT controller memory pool used: 400 bytes, remain bytes: 8496, total:8896
BT controller memory pool used: 764 bytes, remain bytes: 8132, total:8896
BT controller memory pool used: 3784 bytes, remain bytes: 5112, total:8896
BT controller memory pool used: 4840 bytes, remain bytes: 4056, total:8896
BT controller memory pool used: 5164 bytes, remain bytes: 3732, total:8896
BT controller memory pool used: 6124 bytes, remain bytes: 2772, total:8896
BT controller memory pool used: 8896 bytes, remain bytes: 0, total:8896
app started
```

上面的 log 显示正常分配的对内存为 8896bytes，所以我们可以打开 nimble_glue.c 或者 nimble_glue_spark.c 找到堆分配的宏 PAN_BLE_CTLR_BUFFER_ALLOC 将其的值修改为 8896。

```
#define PAN_BLE_CTLR_BUFFER_ALLOC        (8896)
#define PAN_BLE_CTLR_BUFFER_SIZE         (((PAN_BLE_CTLR_BUFFER_ALLOC) + 3)& (~((uint32_
↪t)0x03))) /*4 bytes aligned*/

static uint32_t mem_buffer[PAN_BLE_CTLR_BUFFER_SIZE/4];
static uint32_t mem_pos;
```

我们也可以将堆修改为异常很小的值，比如此处设置为 4000，看下实际输出：

```
[19:38:01.115] 收 ← LL Controller Version:bd5923c

[19:38:01.151] 收 ← BT controller memory pool used: 400 bytes, remain bytes: 3600, total:4000
BT controller memory pool used: 764 bytes, remain bytes: 3236, total:4000
BT controller memory pool used: 3784 bytes, remain bytes: 216, total:4000
BT controller allocating 1056 bytes failed
```

此时分配失败后会触发断言在初始化的地方卡住。

我们可以一开始设置比较大的堆值，然后再调整为合适的值即可。

2.2 App 以及 host 使用的堆（应用层堆全局使用 freertos 的堆）

为了方便资源管理，我们 app 和 host 全局使用 freertos 的堆，相应堆的分配函数 pvPortMalloc。

当前 SDK 哪些资源默认使用了 freertos 的堆呢?

- app 中显式使用 pvPortMalloc 的地方
- freertos task 的栈
- 创建 freertos 定时器时的栈（定时器也是 task）、
- freertos 创建信号量等

2.3 如何查看 freertos heap 的使用

我们可以通过打开 app_config.h 或者 app_config_spark.h 中的 FreeRTOS Heap Usage Print 选项（对应的宏 CONFIG_FREERTOS_HEAP_PRINT）显示的输出底层 controller 所需要的内存。

freertos heap 的宏是 FreeRTOSConfig.h 中的 configTOTAL_HEAP_SIZE，我们可以通过修改 configTOTAL_HEAP_SIZE 的值来改变全局堆的大小。

启动时会输出如下：

```
[19:47:57.875] 收 ← total allocated bytes:216,remain:5920
total allocated bytes:304,remain:5832
total allocated bytes:392,remain:5744
total allocated bytes:480,remain:5656
total allocated bytes:536,remain:5600
total allocated bytes:704,remain:5432
total allocated bytes:792,remain:5344
LL Controller Version:bd5923c

[19:47:57.918] 收 ← app started
total allocated bytes:848,remain:5288
total allocated bytes:2864,remain:3272
total allocated bytes:2960,remain:3176
total allocated bytes:3232,remain:2904
```

(下页继续)

(续上页)

```
total allocated bytes:3328,remain:2808
total allocated bytes:4368,remain:1768
total allocated bytes:4464,remain:1672
total allocated bytes:4520,remain:1616
total allocated bytes:4752,remain:1384
total allocated bytes:4776,remain:1360
total allocated bytes:4808,remain:1328
```

我们故意将 configTOTAL_HEAP_SIZE 设置为一个很小的值，分配失败是会有如下显示：

```
[19:50:28.736] 收 ← total allocated bytes:216,remain:2848
total allocated bytes:304,remain:2760
total allocated bytes:392,remain:2672
total allocated bytes:480,remain:2584
total allocated bytes:536,remain:2528
total allocated bytes:704,remain:2360
total allocated bytes:792,remain:2272
LL Controller Version:bd5923c

[19:50:28.778] 收 ← app started
total allocated bytes:848,remain:2216
total allocated bytes:2864,remain:200
total allocated bytes:2960,remain:104
pvPortMalloc failed
allocate 272 bytes failed,remain:104
```

另外我们也要注意一点，堆定义的空间可以适当多分配一点，有些堆的分配是在运行时才会去调用。

4.5 NDK Mcu Boot

4.5.1 1. 背景介绍

BootLoader 是一个硬件系统的引导代码，可以引导系统软件的升级，由于实际产品中 BootLoader 是不可以或者很难更新的，所有确保 BootLoader 的稳定性和鲁棒性是对一个系统最基本的保证。原则上需要保证 BootLoader 的功能尽量简单可靠，本文主要介绍 ndk mcu 的开发指南，将会从 4 个方面进行阐述，分别是 flash 区域的划分，BootLoader 模式，升级的流程和策略

4.5.2 2. flash 区域的划分

Area	size and range
User flash	28K 0x78000->0x7F000
Backup	220K 0x41000->0x78000
Image	220K 0xA000->0x41000
BootLoader	40K 0x00000->0xA000

Image 为应用程序代码，目前 hr_ota 工程代码大小是 113K，那么用户逻辑代码可以使用 100K。

Backup 为升级代码的备份区，确保升级固件的完整性和安全性过后，再搬运到 Image 区域。

User Flash 区域，为用户存储数据的区域。

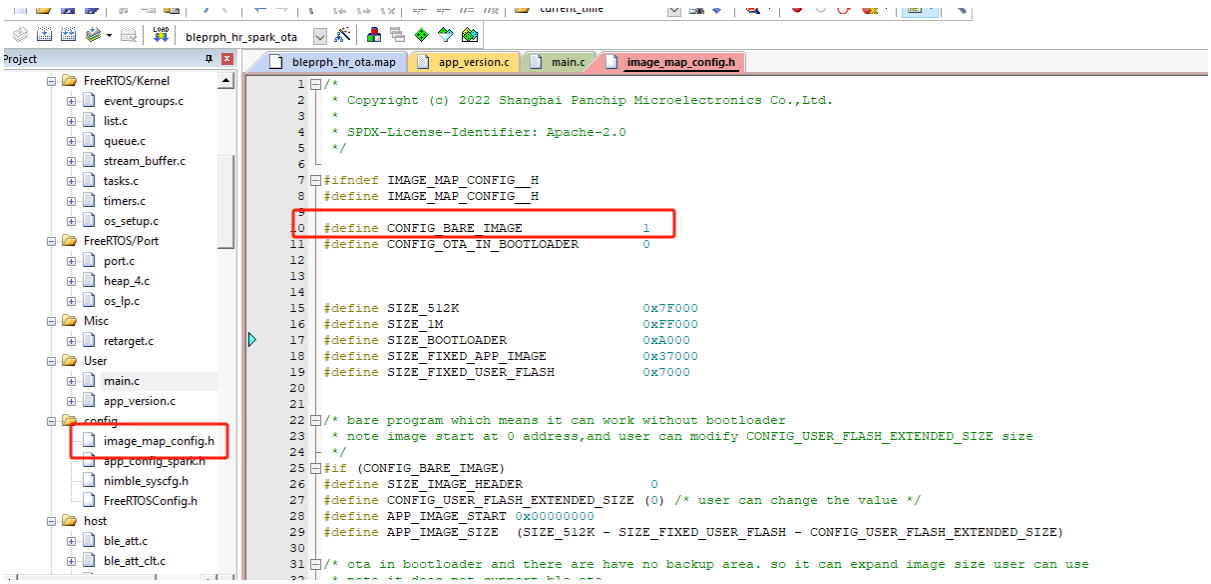
注意：User Flash 的大小是可以有限制的**增大的**，这个特性和 BootLoader 的模式也有很大的关系，详情参考下面 BootLoader 的模式

4.5.3 2.1 BootLoader mode

为了更好适配不同的程序和方案，ndk 的 BootLoader 和应用层配合实现了 3 种 ota 的模式，依次是 bare mode，ota in BootLoader，ota in app

2.1.1 Bare mode

bare mode 以为应用程序是裸机程序，适用于开发阶段的调试，或者特使需求的应用，如下图使能 `CONFIG_BARE_IMAGE` 即可完成工程的选择。

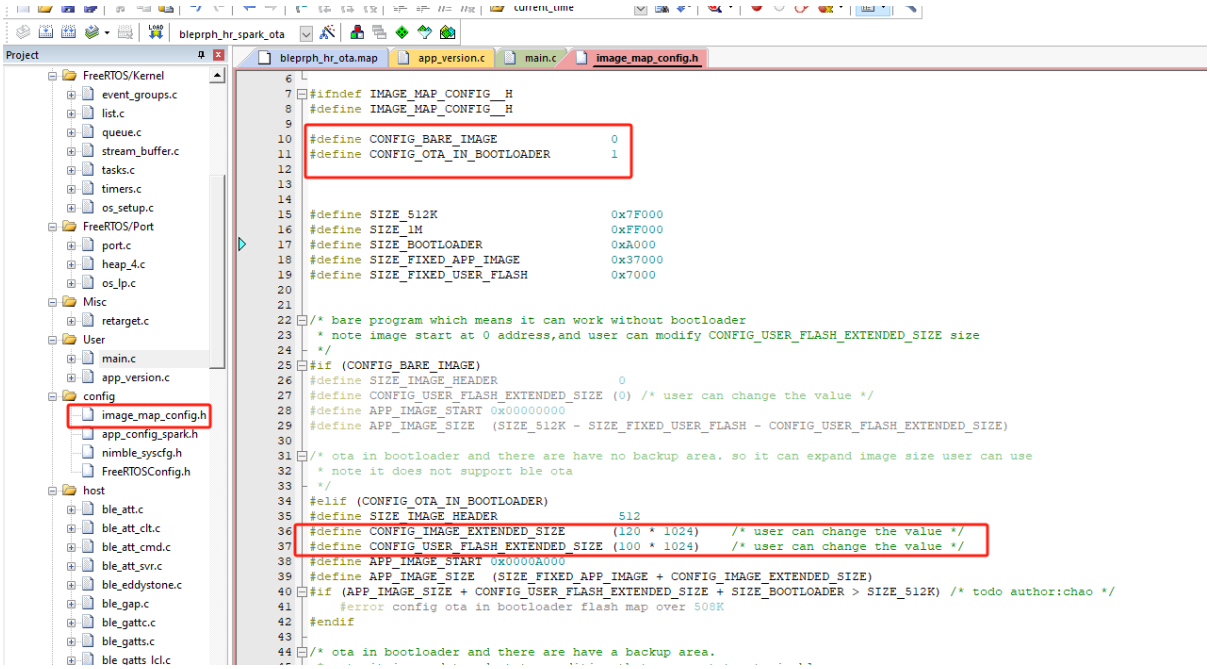


2.1.2 ota in BootLoader

该模式表示 ota 的流程和策略完全在 BootLoader 的程序执行的，这意味着 flash backup 区域可以重新分配给 flash image **和** user flash 区域。

使用该功能的步骤

1. 编译下载 BootLoader 的程序，程序位置为 `ndk\pan107x_mcu_boot` 或者 `ndk\pan108x_mcu_boot`，选择那个 BootLoader 取决于你的芯片版本。
2. 修改应用程序的配置（1）使能 `CONFIG_OTA_IN_BOOTLOADER`，如下图所示（2）修改 image flash 区域的大小或者 user flash 区域的大小（optional），如下图所示



注意：

CONFIG_IMAGE_EXTENDED_SIZE 和 CONFIG_USER_FLASH_EXTENDED_SIZE 扩展 Image 和 User flash 区域的大小，上图配置可以得到下图的公式

```
image_size = SIZE_FIXED_APP_IMAGE + CONFIG_IMAGE_EXTENDED_SIZE
image_size = 220 + 120 = 340 octets

user_flash_size = SIZE_FIXED_USER_FLASH + CONFIG_USER_FLASH_EXTENDED_SIZE
user_flash_size = 28 + 100 = 128 octets
```

2.1.2.1 支持的升级的方法

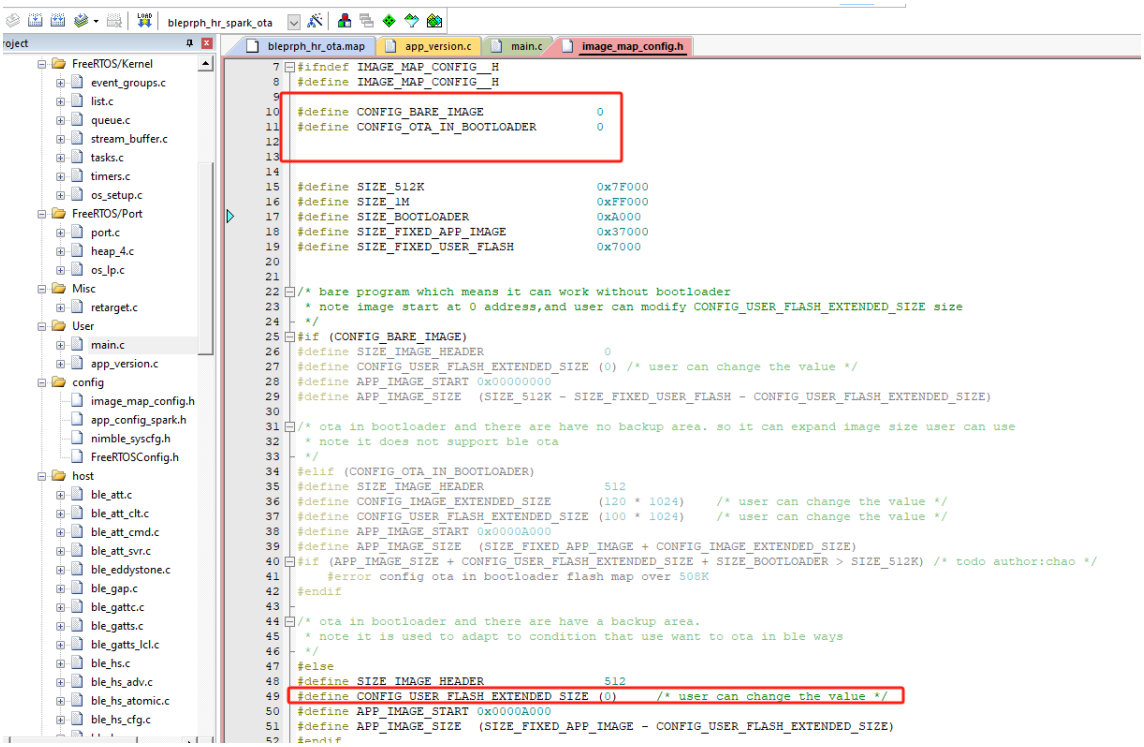
1. USB dfu **模式：107 芯片支持，108 芯片待支持**
2. UART dfu **模式：107 和 108 芯片均支持**
3. PRF OTA **模式：待支持**

2.1.3 ota in APP

默认 OTA 的升级流程是在 APP 完成的，相比于 ota in BootLoader 他主要兼容 smp 的蓝牙升级，同时意味着付出了 image flash size <= 220K 的代价。

使用该功能的步骤

1. 编译下载 BootLoader 的程序，程序位置为 ndk\pan107x_mcu_boot 或者 ndk\pan108x_mcu_boot，选择那个 BootLoader 取决于你的芯片版本。
2. 修改应用程序的配置（1）禁止 CONFIG_OTA_IN_BOOTLOADER 和 CONFIG_BARE_IMAGE，如下图所示（2）修改 user flash 区域的大小（optional），如下图所示



注意：此时依然可可以通过 CONFIG_USER_FLASH_EXTENDED_SIZE 扩张 user flash 区域，如果这样做意味着 Image 区域也变相的减少了。

2.1.2.1 支持的升级的方法

蓝牙 smp 升级，需要应用层支持，详情参考 “‘bleprph_hr_ota‘ 的 demo

4.5.4 3 BootLoader 升级流程和策略

BootLoader 启动的时候，会等待很多个信号，然后依次 trigger 信号的操作。这儿我们抽象成 QT 编程中描述的信号和槽的概念，代码工程 ndk\pan107x_mcu_boot 或者 ndk\pan108x_mcu_boot。

```
signal:
bool sig_key_push_down(void);
bool sig_special_ram_value_detected(void);
bool sig_ota_start_received(void);
bool sig_back_up_is_completed_image(void);

slots:
void on_usb_dfu_enter(void);
void on_prf_ota_enter(void);
void on_uart_dfu_enter(void);
void on_image_load_enter(void);
```

连接信号和槽

```
typedef void (slot_handler_t)(void);
typedef void (signal_handler_t)(void);

void connect(uint8_t priority, signal_handler_t signal, slot_handler_t slot);
```

事件检测流程

```
/* when checking backup image is valid, the on_image_load_enter function will be handled */
ss_connect(0, sig_back_up_is_completed_image, on_image_load_enter);
```

(下页继续)

(续上页)

```
#if BOOT_FROM_UART
/* when detecting key1 down, the on_uart_dfu_enter function will be handled */
ss_connect(1, sig_key1_push_down, on_uart_dfu_enter);
#endif

#if BOOT_FROM_USB
/* when dectecting key2 down, the on_usb_dfu_enter function will be handled */
ss_connect(2, sig_key2_push_down, on_usb_dfu_enter);
#endif

#if BOOT_FROM_PRF_OTA
/* when receive a ota start packet, the on_prf_ota_enter function will be handled */
ss_connect(3, sig_ota_start_received, on_prf_ota_enter);
#endif

/* handle all of events related signal fuction*/
ss_events_handle();

/* recovery gpio status that you used to trigger signal */
sig_hardware_recovery();
```

事件检测是分为优先级的，这儿巧妙通过数组索引的流程实现了这个功能。之所以要分优先级，是因为流程的需要，例如备份区已经有了完整的代码，可能需要提前处理一下，处理完成过后也许就不需要升级了。

3.1 USB dfu 模式

```
void on_usb_dfu_enter(void);
```

进入 dfu 过后，107 写寄存器进入 ROM 模式,108 需要自己实现

3.2 UART dfu 模式

```
void on_uart_dfu_enter(void);
```

进入 dfu 过后，会采用 xmodem 协议进行 OTA 升级

3.3 PRF OTA 模式

```
void on_prf_ota_enter(void);
```

待 ota 设备复位进入 ota 状态，firmware 通过 2.4g dongle 发送给待 ota 设备

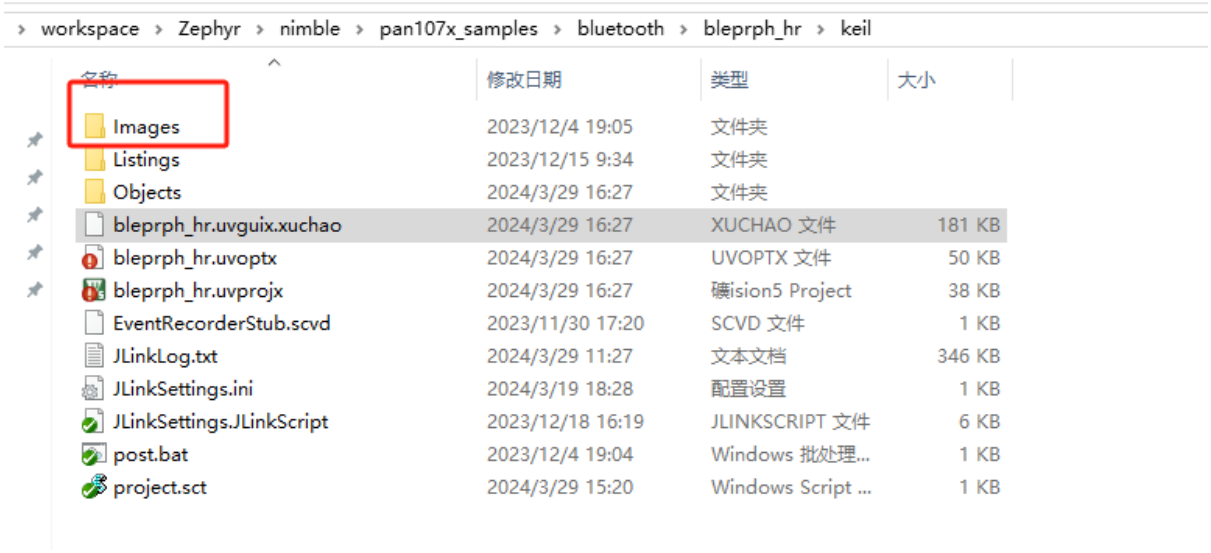
3.4 Backup dfu 模式

检测 backup 区域固件的完整性, 决定是否搬移到 image

4.5.5 4. uart 升级详解

升级的固件需要签名校验，默认 keil 编译的时候，在工程的同级目录 Images 下会自动生成 `ndk_app.signed.bin`。使用该功能依赖系统安装了 python 和 python 的库文件 numpy。如果系统已经安装 python，请执行下面的命令安装 numpy

```
python -m pip install numpy
```



4.1 检测并进入 uart 升级模式

1. 使能 uart dfu 功能，通过 BOOT_FROM_UART 宏进行使能
2. 编写 uart dfu 进入的 signal 函数，并将信号和槽连接，槽属于升级流程 BootLoader 已经支持，用户不需要修改。用户可以修改 signal 函数。默认如下

```
#if BOOT_FROM_UART
/* when detecting key1 down, the on_uart_dfu_enter function will be handled */
ss_connect(1, sig_key1_push_down, on_uart_dfu_enter);
#endif

/* user can implement a custom signal fucntion */
bool sig_key1_push_down(void)
{
        GPIO_SetMode(P2, BIT0, GPIO_MODE_INPUT);
        GPIO_EnablePullupPath(P2, BIT0);

        for (uint16_t i = 0; i < 1000; i++) {
                if (P20 == 1) {
                        return false;
                }
        }
        return true;
}
```

3. 下载 BootLoader 和应用程序，应用层程序需要配置 OTA_in_Bootloader 的模式

4.2 操作流程

1、烧录 boot，在 boot_config.c 里面配置

```
#define BOOT_FROM_UART          1
```

2、当前工程比如是 peripheral_hr 的工程，需要使用 uart 升级到需要的工程，比如 ble_central

3、使用工具 SecureCRT 进行升级

4、打开工具 SecureCRT 连接设备串口，串口波特率 921600

5、把板卡上 P20 接 GND，复位板卡，查看工具 SecureCRT 上 log 打印，一直输出 CCCCCCCCCC

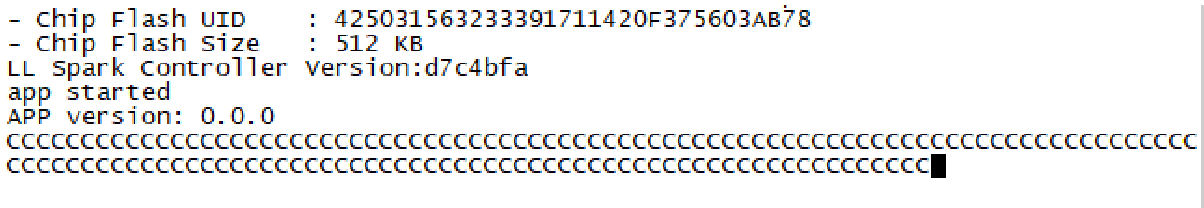


图 10: PAN107x UART DFU 进入升级模式

6、在工具 SecureCRT 的”传输”界面选择”发送 Xmodem(N)”，选择待升级工程的文件，位于 image 路径下：ndk_app.signed.bin，文件开始传输

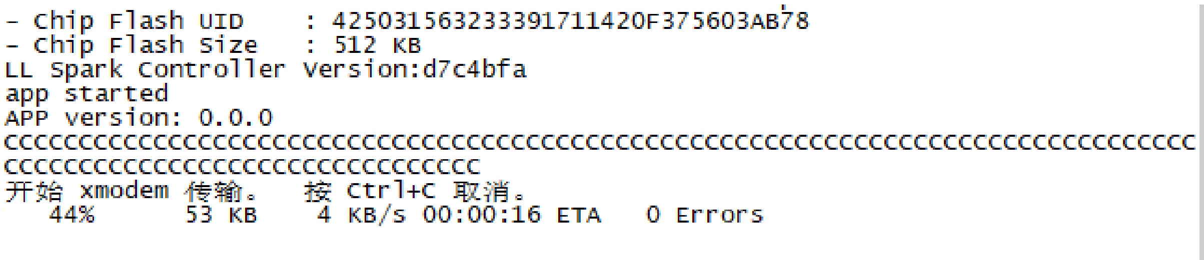


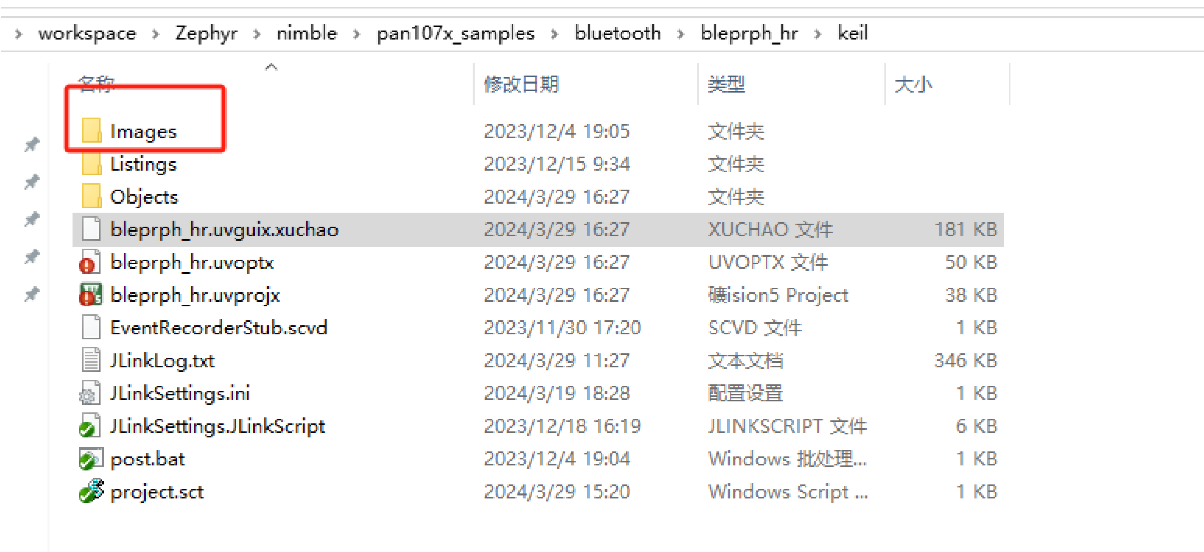
图 11: PAN107x UART DFU 传输文件

7、拔掉 GND，复位设备，查看串口 log，已经打印升级后的程序 log；

4.5.6 5 USB dfu 升级详解

升级的固件需要签名校验，默认 keil 编译的时候，在工程的同级目录 Images 下会自动生成 `ndk_app.signed.bin`。使用该功能依赖系统安装了 python 和 python 的库文件 numpy。如果系统已经安装 python，请执行下面的命令安装 numpy

```
python -m pip install numpy
```



5.1 检测并进入 usb 升级模式

1. 使能 usb dfu 功能，通过 `BOOT_FROM_USB` 宏进行使能

2. 编写 uart dfu 进入的 signal 函数，并将信号和槽连接，槽属于升级流程 BootLoader 已经支持，用户不需要修改。用户可以修改 signal 函数。默认如下

```
#if BOOT_FROM_USB
/* when dectecting key2 down, the on_usb_dfu_enter function will be handled */
ss_connect(2, sig_key2_push_down, on_usb_dfu_enter);
#endif

/* user can implement a custom signal fucntion */
bool sig_key2_push_down(void)
{

        GPIO_SetMode(P2, BIT1, GPIO_MODE_INPUT);
        GPIO_EnablePullupPath(P2, BIT1);

        for (uint16_t i = 0; i < 1000; i++) {
                if (P21 == 1) {
                        return false;
                }
        }
        return true;
}
```

3. 下载 BootLoader 和应用程序，应用层程序需要配置 OTA_in_Bootloader 的模式

5.2 操作流程

1、烧录 boot，在 boot_config.c 里面配置

```
#define BOOT_FROM_USB          1
```

2、当前工程比如是 peripheral_hr 的工程，需要使用 USB 升级到需要的工程，比如 ble_central

3、使用 SDK 的 05_TOOLS 里面的工具箱工具 pan107xToolBox V0.0.00x 进行升级

4、打开工具 pan107xToolBox V0.0.00x，选择”显示”>”DFU”，连接设备 USB 口

5、把板卡上 P21 接 GND，复位设备，查看工具 pan107xToolBox V0.0.00x 识别 USB 口

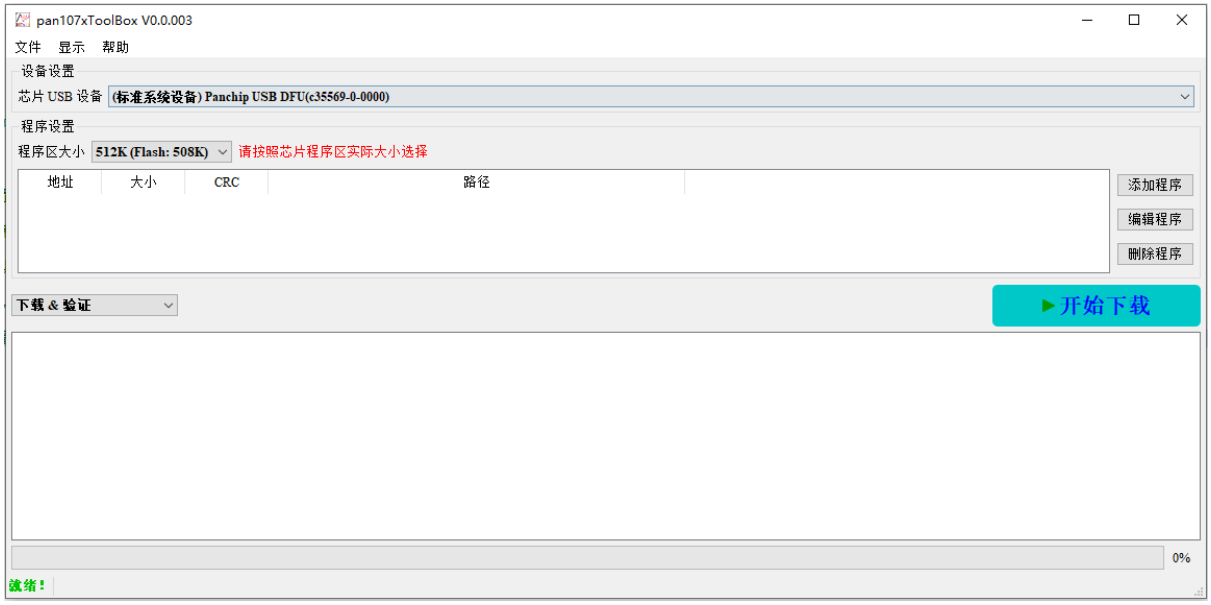


图 12: PAN107x USB DFU 识别 usb 口

6、在工具程序设置那里选择”添加程序”>”加载程序”，选择待升级工程的文件，位于 image 路径下:ndk_app.signed.bin，注意地址需要从 0x41000 开始

7、程序加载好之后，点击”开始下载”即可

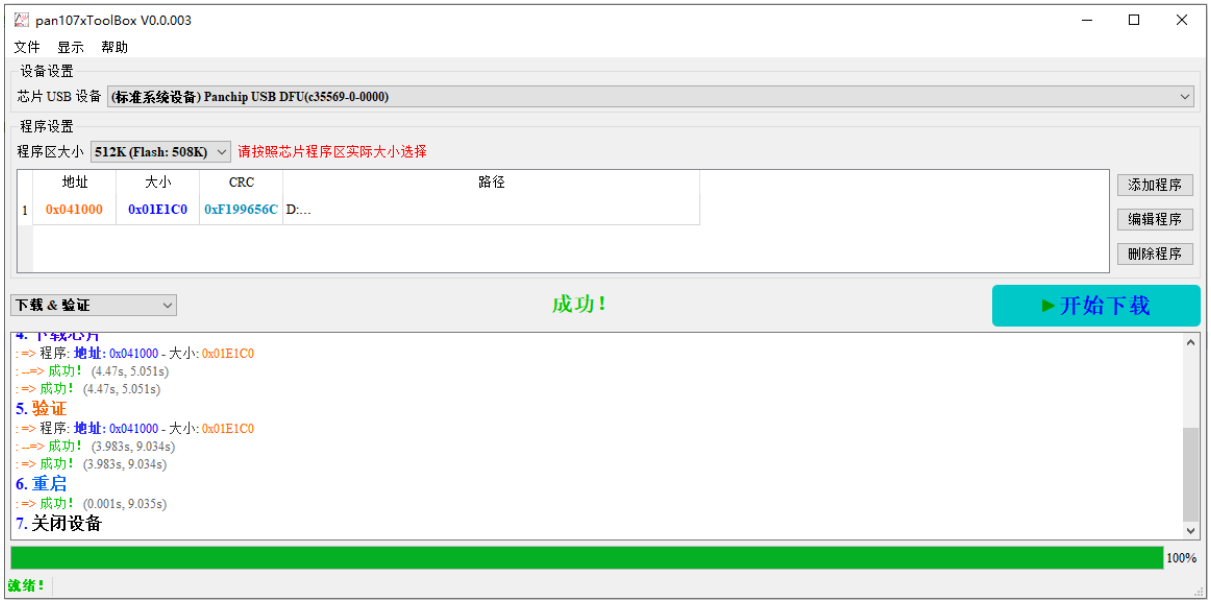


图 13: PAN107x USB DFU 升级成功

8、拔掉 GND 线，复位设备，查看串口 log，已经打印升级后的程序 log

4.5.7 6 PRF ota 升级详解

1、打开工具 Panchip 2.4G OTA V0.0.002.exe，选择 2.4g dongle 对应的串口，具体设置如下图所示：

2、上位机的使用说明参考上位机的帮助文档。

note：**多设备升级不进行校验，不能确保每个设备都能升级成功。有些设备如果升级不成功需要重新升级。**

4.6 NDK 常见问题（FAQs）

4.6.1 Q1：为什么我使用 JLink (SWD) 烧录一个工程后，无法（或很难）再次烧录？

正常来说，如果 JLink 接线没问题的话，您是可以反复烧录 App 工程的，若您发现在空板上使用 JLink 烧录 App 工程很容易成功，但再次烧录则不容易成功，则有可能是以下两个原因：

1. 您在 App 代码的初始化流程中修改了 P00/P01 引脚的功能，将其由默认的 SWD 功能切换为其他功能（如 GPIO），那么当试图使用 JLink 进行烧录的时候，SWD 通信将无法正常进行
2. 您的 App 工程会频繁地或长期地进入芯片的 DeepSleep 或 Standby 低功耗流程，在这两种低功耗模式下，SWD 通信将无法正常进行

若遇到上述情况，您可以尝试将 JLink 的 RESET 引脚接到 SoC 的 RESET 引脚上，然后再进行烧录，很多时候可以解决问题。若此方法仍然无法成功，那么您可以尝试以下两种方法：

1. 使用 Panchip 量产烧录工具 PANLINK，将芯片 Flash 全部擦除
2. 执行 JLink 脚本 ForceEraseVectorTable_PAN107x.bat（位于: `<pan1070-ndk>\05_TOOLS\`调试工具目录下），此脚本会尝试将 Flash 上的启动代码擦掉，若成功，后续即可正常使用 JLink 重新烧录

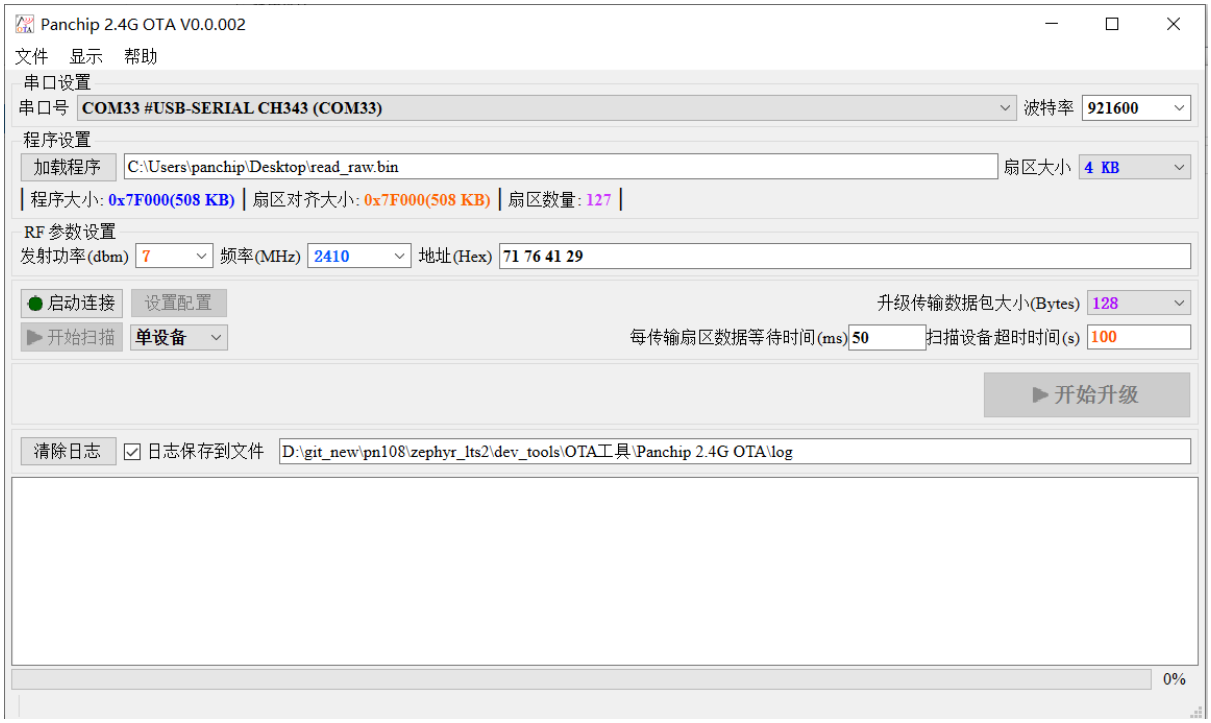


图 14: PAN107x PRF OTA 上位机

注: 使用上述两种方法擦除 Flash 时, SoC 的 RESET 引脚是一定要连接上 PANLINK/JLink 的

Chapter 5

量产测试

5.1 量产烧录

5.1.1 1. 芯片硬件系统说明

如果芯片按照我司提供的 PAN107x / PAN101x **硬件参考设计**设计的模块，烧录连接如表 1-1、表 1-2 所示。

J-Flash	连接	PAN107x / PAN101x 芯片模块
VTref 3.3V	<—>	VBAT
GND	<—>	GND
SWDIO	<—>	P01
SWDCLK	<—>	P00

PAN-LINK2.0	连接	PAN107x / PAN101x 芯片模块
VDD	<—>	VBAT
GND	<—>	GND
A2	<—>	RST
A3	<—>	P01
A4	<—>	P00

如果需要烧录**裸芯片**，没有任何外围器件的 PAN107x / PAN101x 芯片烧录连接如表 1-3、表 1-4 所示。

J-Flash	连接	PAN107x / PAN101x 裸芯片
VTref 3.3V	<—>	VCC_RF
GND	<—>	GND（注：QFN32: 33 **号引脚** (ePAD)）
SWDIO	<—>	P01
SWDCLK	<—>	P00

PAN-LINK2.0	连接	PAN107x / PAN101x 裸芯片
VDD	<—>	VCC_RF
GND	<—>	GND（注：QFN32: 33 **号引脚** (ePAD)）
A2	<—>	RST
A3	<—>	P01
A4	<—>	P00

5.1.2 2. 量产烧录工具

为配合 PAN-LINK2.0 烧录 PAN107x / PAN101x 芯片程序工具。

下载

2.1. 硬件准备

预先将 PAN-LINK2.0 通过 MiniUSB 线连接到 PC 电脑。

图 1: 图 2-1-1 PAN-LINK2.0 烧录器

图 2: 图 2-1-2 MiniUSB 连接线

如果 PAN-LINK2.0 固件程序不支持 PAN107x 芯片烧录，则需要根据提示自动更新升级。
或按照帮助文档方法更新 PAN-LINK2.0 固件程序。

2.1.1.PAN107x / PAN101x **芯片烧录接线** 注：PAN-LINK2.0 接口的 VCC 与 VIO 通过跳线帽短接。

PAN-LINK2.0 接口脚	连接	PAN107x / PAN101x 芯片脚
VDD	<—>	VDD
GND	<—>	GND
A1	<—>	RST
A3	<—>	P01
A4	<—>	P00

2.2. 工具界面

图 3: 图 2-2-1 烧录工具界面

如上图 2-2-1 所示为烧录工具界面。

1、在下载程序配置中的下载程序配置项右键点击加载程序，实现加载烧录程序功能。

2、通过点击擦除模式前面图标或右键选择更改烧录擦除模式。

3、根据需求选择设置其他下载配置。

4、选择下载模式，或直接默认下载到 PAN-LINK 后下载到芯片模式。

5、点击下载开始下载程序到芯片。

2.3. 查看帮助文档

通过烧录工具的**帮助**-> **查看帮助文档**或直接通过快捷键 F1, 打开查看帮助文档。

PAN-LINK2.0 程序更新方法、以及烧录工具的详细使用说明都在帮助文档中有详述。

5.2 RF TEST

图 4: 图 2-3-1 查看帮助文档

5.2.1 1 功能概述

本文主要介绍 PAN107x RF 测试固件的使用。

5.2.2 2 环境要求

- PAN107x EVB 若干块
- USB 转串口工具若干块
- 硬件接线：
  - 使用**杜邦线**连接 EVB 和 USB 转串口工具:
    - ICEK(P01) 与 USB 转串口工具 TX 连接
    - ICED(P00) 与 USB 转串口工具 RX 连接
- PAN107x ToolBox 下载
- USB TYPE-C 线

5.2.3 3 RF 测试固件说明

NO	固件说明	下载链接	更新日期
1	RF 性能测试 (发射功率、频偏、EVM、107 和 107 对测收包率等)	PAN107x RF 测试固件	2024-06-07
2	RF 信号采集测试 (RSSI)	PAN107x RF 信号采集固件	2024-04-07

5.2.4 4 演示说明

PAN107x EVB **板** PIN **脚接线说明：**

PAN107x EVB 板 GPIO	USB 转串口工具
ICEK_uart1_rx(P01)	UART_TX
ICED_uart1_tx(P00)	UART_RX
VBAT	VCC(3.3V)
GND	GND

4.1 RF 性能测试（串口模式）

RF 测试支持 2 种模式，第一种是采用标准仪器进行测量（cmw500），另一种采用 PAN107x tool box 工具进行射频测试。

4.1.1 标准射频测试

1. 下载 PAN107x RF 测试固件
2. 烧录.hex 程序，烧录流程可参考J-Flash 烧录方法 或Panlink 烧录方法 ，**烧录成功后板子重新上电**。
3. 连接串口，然后然后连接仪器（cmw500）的串口和射频接口

4.1.2 PAN107x Tool Box 工具进行射频测试

1. 用 panlink 或者 j-flash 烧录固件” PAN1080 RF 测试固件”。
2. PAN107x 的测试固件可在 PAN107x ToolBox 工具中导出，如下图所示，也可使用路径” 05_TOOLS\RF 测试固件\PAN107x RF 测试固件.zip”。
3. 烧录流程可参考J-Flash 烧录方法 或Panlink 烧录方法 ，**烧录成功后板子重新上电**。
4. 测试流程可参考PAN107x 工具箱用户指南中的第一章 RF 测试。

   uart 波特率:115200

典型的测试场景

- TX 测试，可以通过软件发送单载波和 dtm 数据包（下图是一个单载波的典型配置界面），通过频谱仪观察射频状况

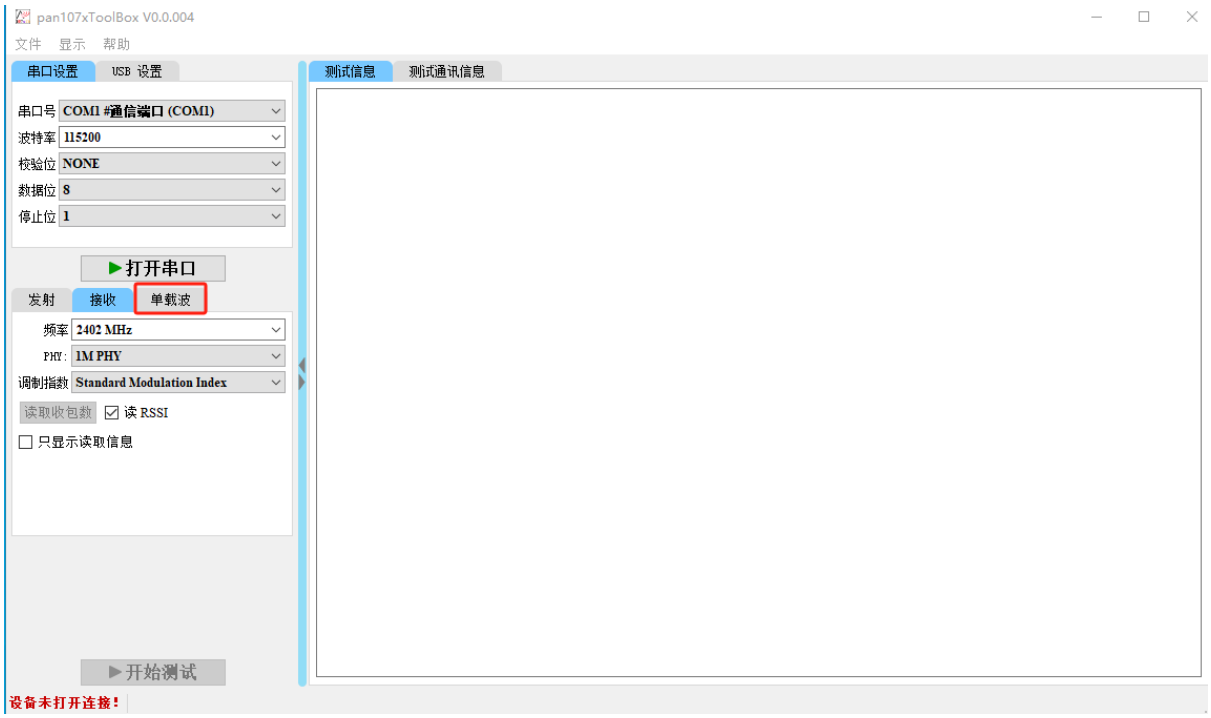


图 5: 图-1 单载波配置界面

- RX 测试可以打开 2 个软件，控制 2 个 107x 芯片，一个设置为 tx 模式，一个设置为 rx 模式，tx 端设置 tx counts 发送，rx 端打印收到的 counts，进而判断 rx 的质量和灵敏度。

4.4 RF 信号采集测试

1. 用 panlink 或者 j-flash 烧录固件” RSSIVIEWER_BAUD115200_P00RX_P01TX_V001.hex “。
2. 测试固件可在 PAN1080 ToolBox 工具中导出，也可使用路径” 5_Tools\RF 测试固件\PAN1080 RSSI VIEWER 测试固件.zip”。
3. 烧录流程可参考J-Flash 烧录方法 或Panlink 烧录方法 ，**烧录成功后板子重新上电**。

4. 测试流程可参考PAN107x 工具箱用户指南中的第四章 RF 信号采集。

5.3 JFlash 烧录

本文介绍使用 Segger J-FLash 工具烧录固件到 PAN107x SoC 的方法。

在使用 J-Flash 烧录工具前，请确保 J-Link 硬件与 PAN107x 芯片的 SWD 连接正常。

J-Flash 工具操作步骤如下：

1. 打开 PAN107x NDK/ZDK 中的 J-FLash 烧录工具

   在 `<PAN107x-DK>\05_TOOLS\调试工具\JLink-V644b` 目录下找到 “JFlash.exe”

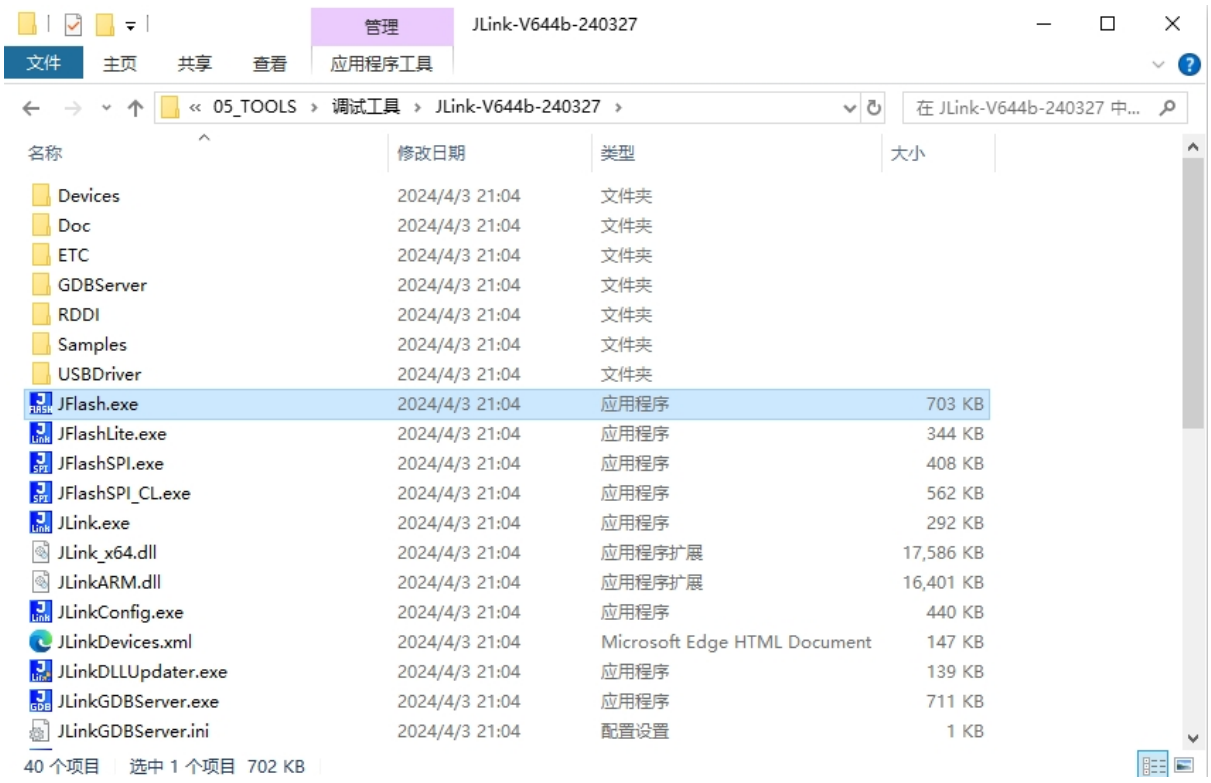


图 6: J-Flash 目录路径

2. 双击 `JFlash.exe` 打开软件，在欢迎界面选择 `Create a new project`，然后点击 `Start J-Flash` 按钮
3. 在新建工程界面，点击`...`，进入 Device 选择界面
4. 在 `Manufacturer` 中选择 `Panchip`，然后找到并选择名为 PAN107X 的 Device，点击 `OK` 按钮确认
5. 点击菜单栏 `Target - Connect`，连接目标芯片
6. 拖动待烧录固件至 J-Flash 软件右侧的空白区域
7. 点击菜单栏 `Target - Production Programming`（快捷键 `F7`），烧录程序至芯片
8. 烧录完成后，会有弹窗提示

   注：烧录完成后程序不会立刻执行，需要手动复位一下芯片，或重新给芯片上电。

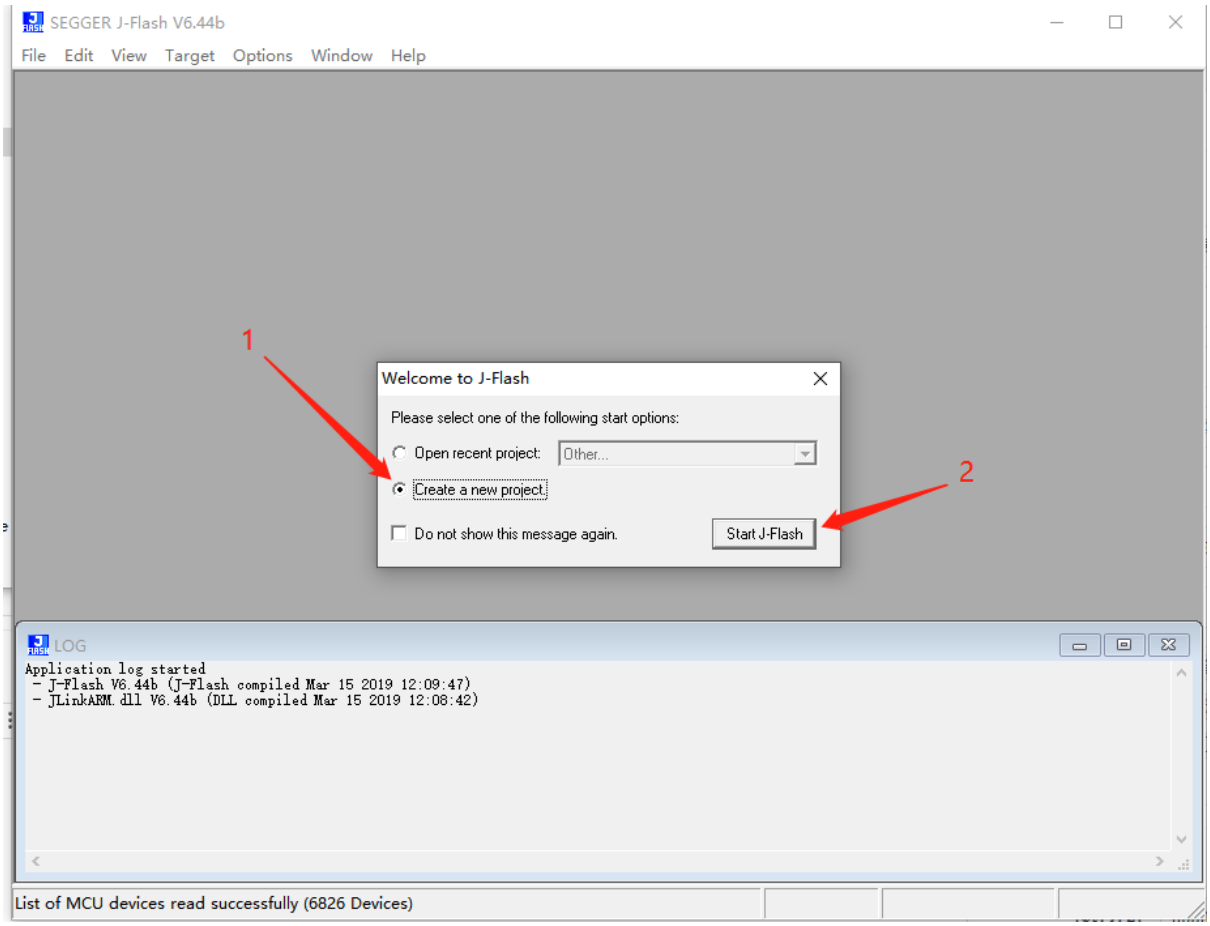


图 7: J-Flash 欢迎界面

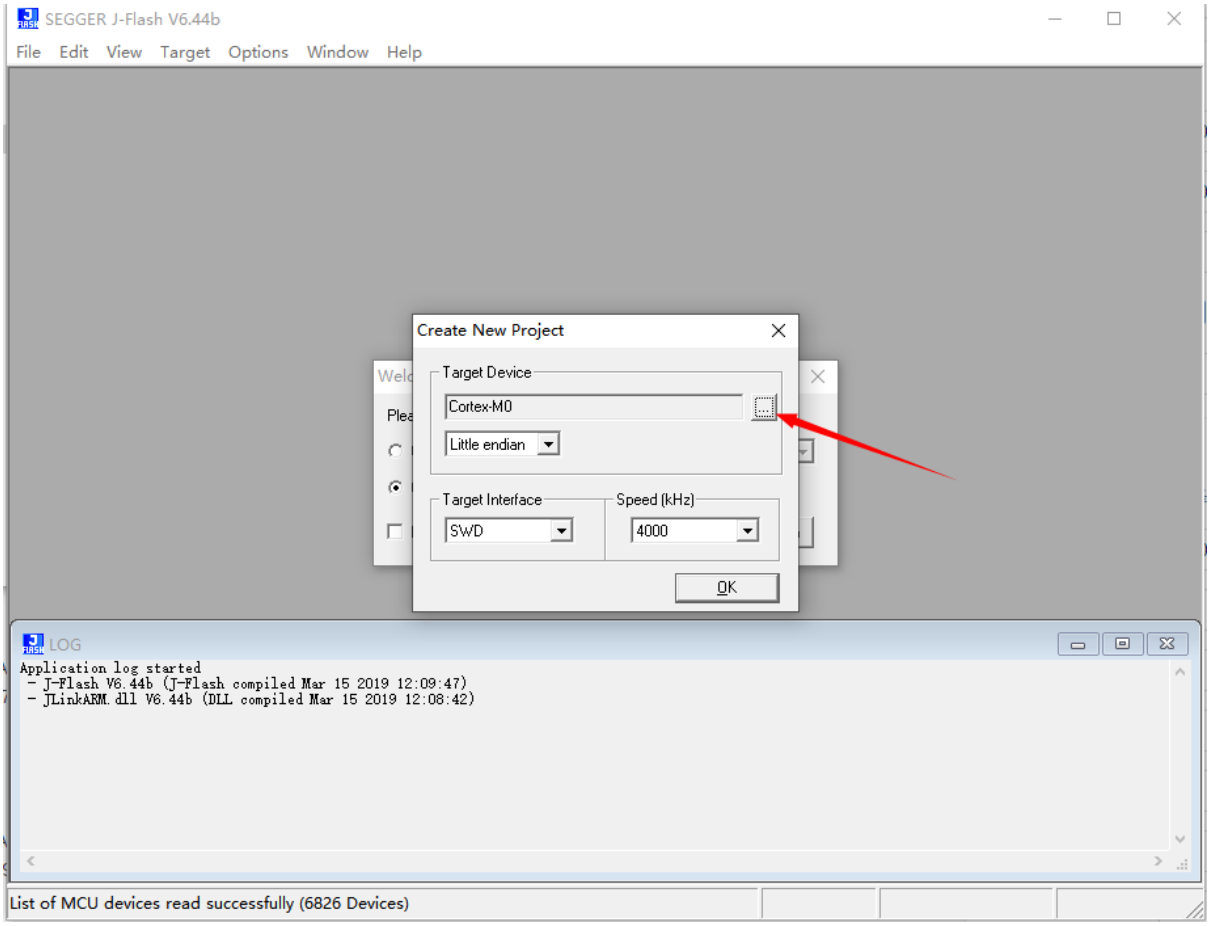

SEGGER J-Flash V6.44b
File Edit View Target Options Window Help
Create New Project
Target Device
Cortex-M0
Little endian
Target Interface
SWD
Speed (kHz)
4000
OK
LOG
Application log started
- J-Flash V6.44b (J-Flash compiled Mar 15 2019 12:09:47)
- JLinkARM.dll V6.44b (DLL compiled Mar 15 2019 12:08:42)
List of MCU devices read successfully (6826 Devices)

图 8: 新建工程界面

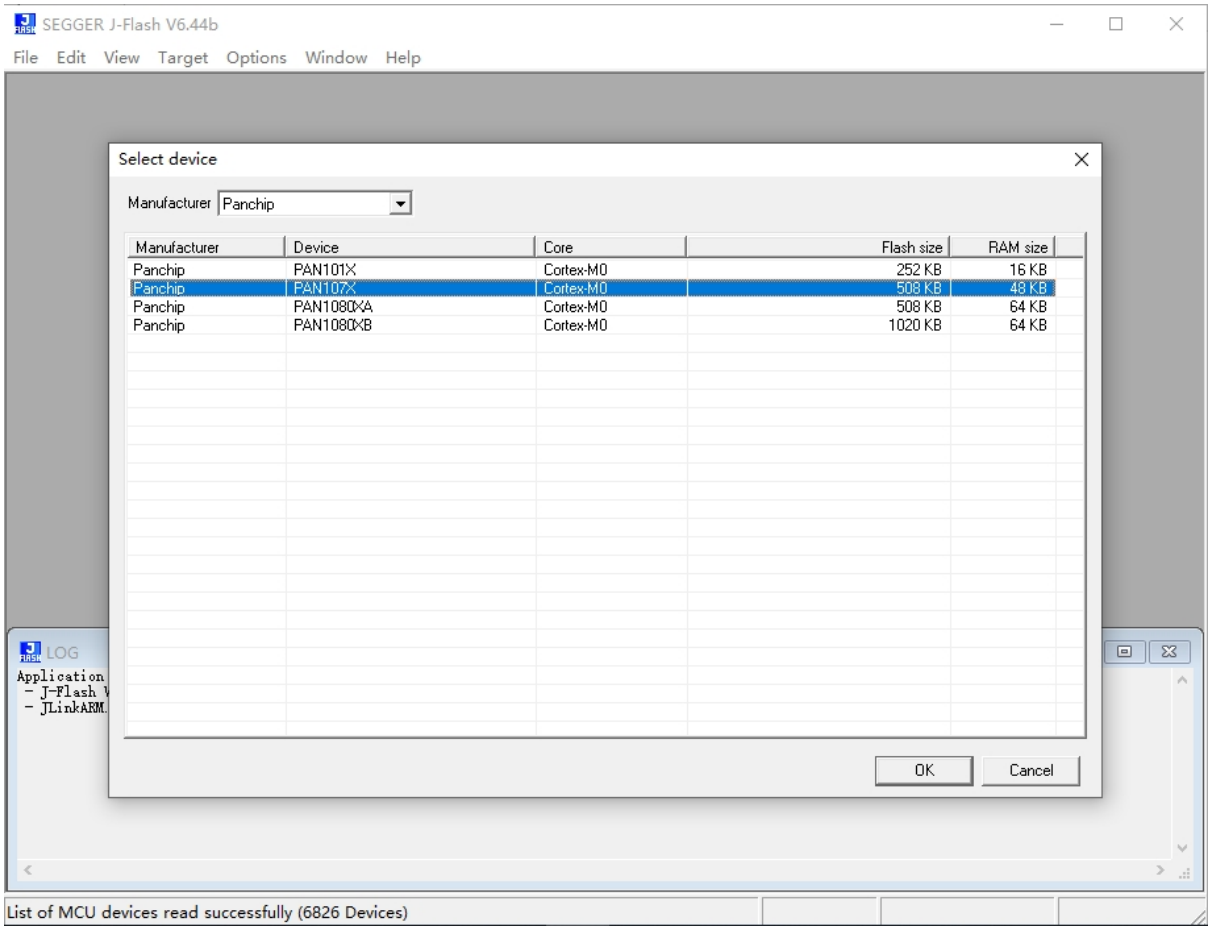


图 9: Device 选择界面

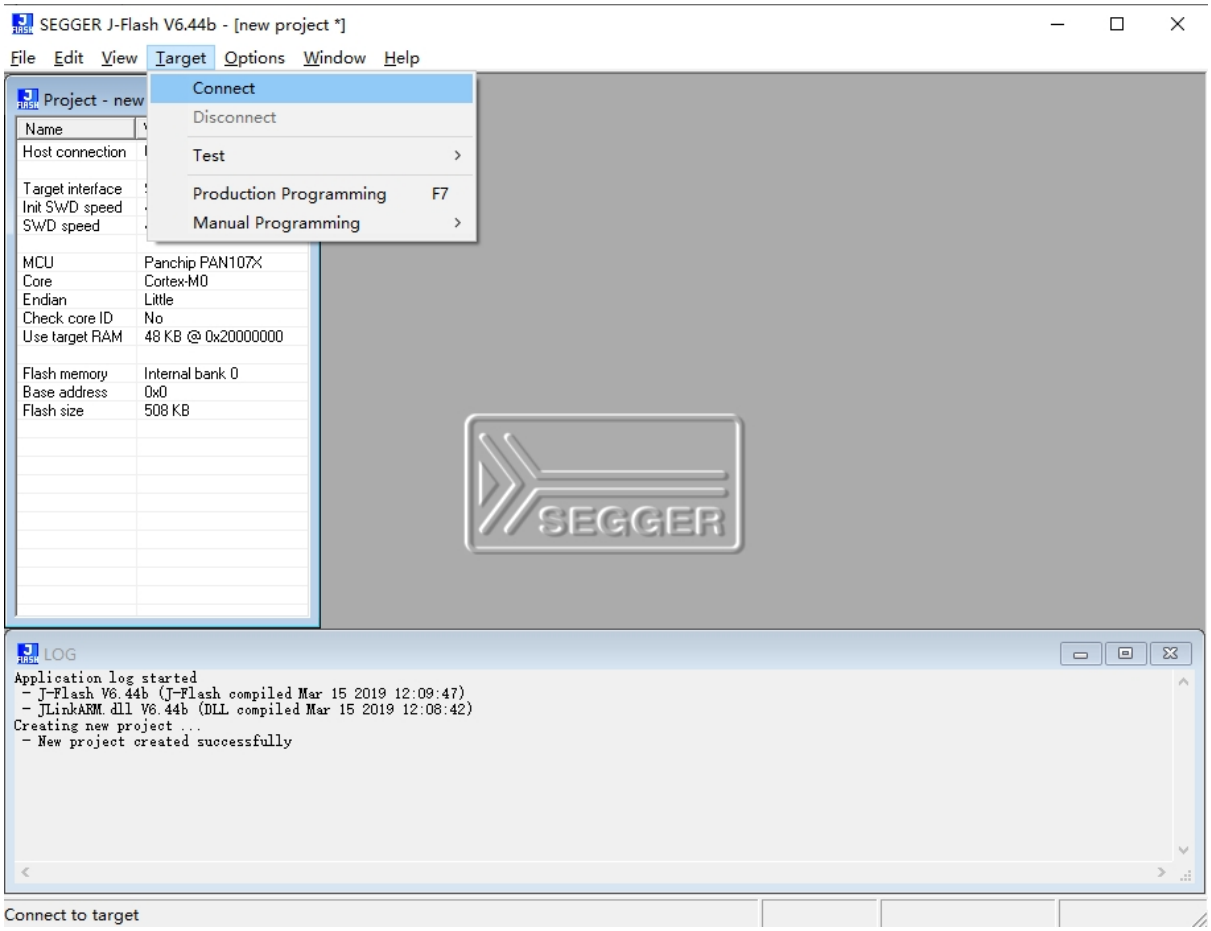

SEGGER J-Flash V6.44b - [new project *]
File Edit View Target Options Window Help
Project - new
Name
Host connection
Target interface
Init SWD speed
SWD speed
MCU Panchip PAN107X
Core Cortex-M0
Endian Little
Check core ID No
Use target RAM 48 KB @ 0x20000000
Flash memory Internal bank 0
Base address 0x0
Flash size 508 KB
Connect
Disconnect
Test
Production Programming F7
Manual Programming
SEGGER
LOG
Application log started
- J-Flash V6.44b (J-Flash compiled Mar 15 2019 12:09:47)
- JLinkARM.dll V6.44b (DLL compiled Mar 15 2019 12:08:42)
Creating new project ...
- New project created successfully
Connect to target

图 10: 连接目标芯片

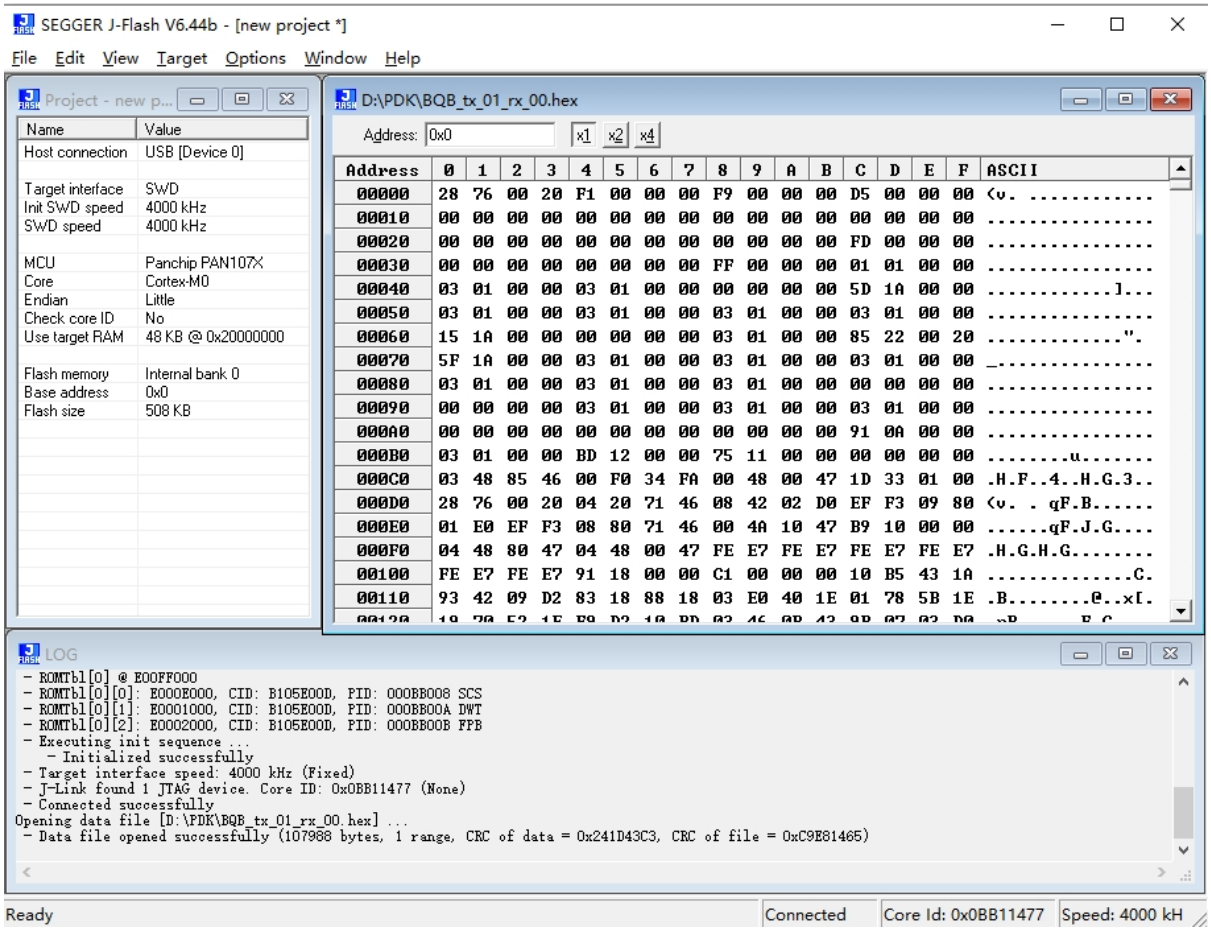

SEGGER J-Flash V6.44b - [new project *]
File Edit View Target Options Window Help
Project - new p...
Name Value
Host connection USB [Device 0]
Target interface SWD
Init SWD speed 4000 kHz
SWD speed 4000 kHz
MCU Panchip PAN107X
Core Cortex-M0
Endian Little
Check core ID No
Use target RAM 48 KB @ 0x20000000
Flash memory Internal bank 0
Base address 0x0
Flash size 508 KB
D:\PDK\BQB_tx_01_rx_00.hex
Address: 0x0
x1 x2 x4
Address 0 1 2 3 4 5 6 7 8 9 A B C D E F ASCII
00000 28 76 00 20 F1 00 00 00 F9 00 00 00 D5 00 00 00
00010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00020 00 00 00 00 00 00 00 00 00 00 00 00 FD 00 00 00
00030 00 00 00 00 00 00 00 00 FF 00 00 00 01 01 00 00
00040 03 01 00 00 03 01 00 00 00 00 00 00 5D 1A 00 00
00050 03 01 00 00 03 01 00 00 03 01 00 00 03 01 00 00
00060 15 1A 00 00 00 00 00 00 03 01 00 00 85 22 00 20
00070 5F 1A 00 00 03 01 00 00 03 01 00 00 03 01 00 00
00080 03 01 00 00 03 01 00 00 03 01 00 00 00 00 00 00
00090 00 00 00 00 03 01 00 00 03 01 00 00 03 01 00 00
000A0 00 00 00 00 00 00 00 00 00 00 00 00 91 0A 00 00
000B0 03 01 00 00 BD 12 00 00 75 11 00 00 00 00 00 00
000C0 03 48 85 46 00 F0 34 FA 00 48 00 47 1D 33 01 00
000D0 28 76 00 20 04 20 71 46 08 42 02 D0 EF F3 09 80
000E0 01 E0 EF F3 08 80 71 46 00 4A 10 47 B9 10 00 00
000F0 04 48 80 47 04 48 00 47 FE E7 FE E7 FE E7 FE E7
00100 FE E7 FE E7 91 18 00 00 C1 00 00 00 10 B5 43 1A
00110 93 42 09 D2 83 18 88 18 03 E0 40 1E 01 78 5B 1E
LOG
- ROMTbl[0] @ E00FF000
- ROMTbl[0][0]: E000E000, CID: B105E00D, PID: 000BB008 SCS
- ROMTbl[0][1]: E0001000, CID: B105E00D, PID: 000BB00A DWT
- ROMTbl[0][2]: E0002000, CID: B105E00D, PID: 000BB00B FPB
- Executing init sequence ...
- Initialized successfully
- Target interface speed: 4000 kHz (Fixed)
- J-Link found 1 JTAG device. Core ID: 0x0BB11477 (None)
- Connected successfully
Opening data file [D:\PDK\BQB_tx_01_rx_00.hex] ...
- Data file opened successfully (107988 bytes, 1 range, CRC of data = 0x241D43C3, CRC of file = 0xC9E81465)
Ready
Connected
Core Id: 0x0BB11477
Speed: 4000 kH

图 11: 载入待烧录固件

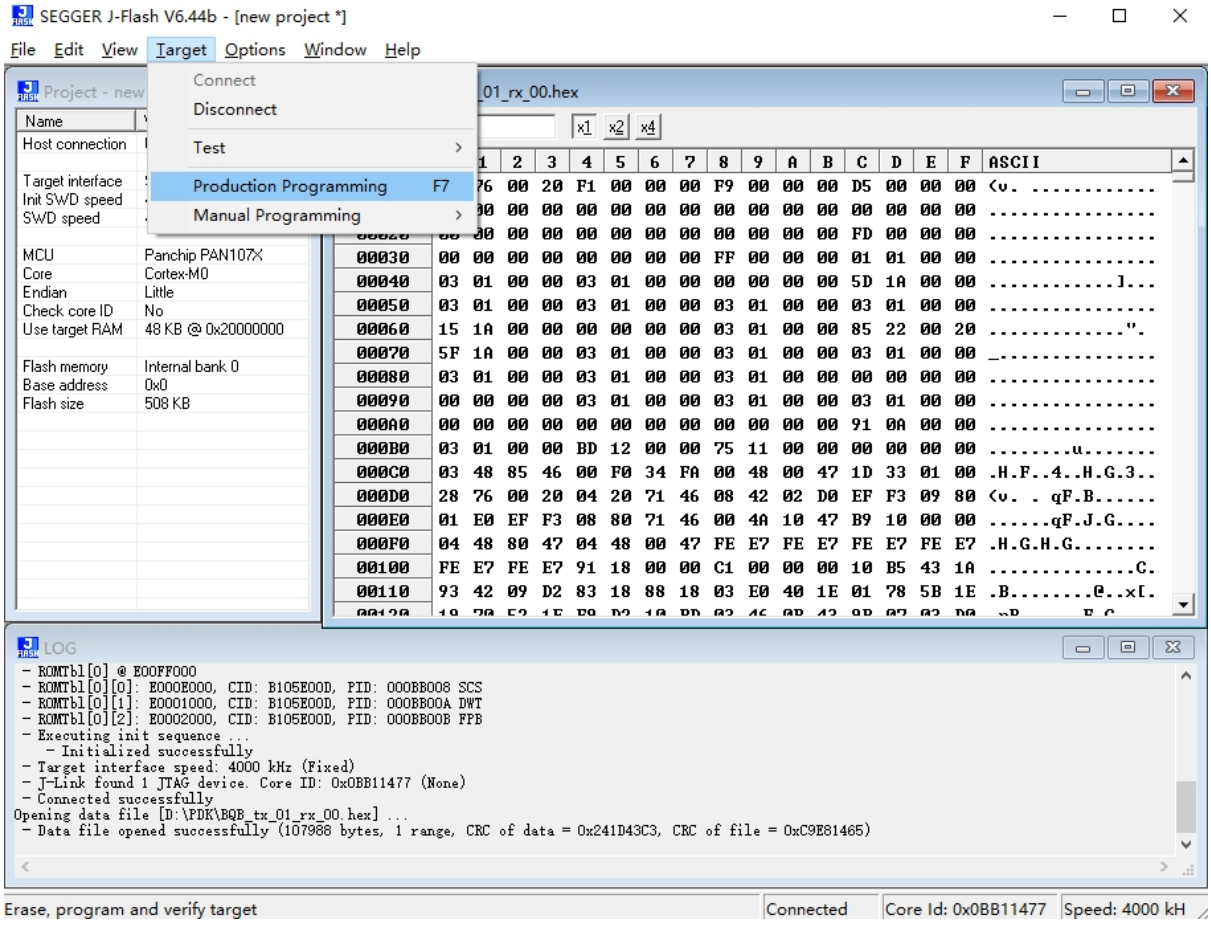

SEGGER J-Flash V6.44b - [new project *]
File Edit View Target Options Window Help
Connect
Disconnect
Test
Production Programming F7
Manual Programming
Project - new
Name
Host connection
Target interface
Init SWD speed
SWD speed
MCU Panchip PAN107X
Core Cortex-M0
Endian Little
Check core ID No
Use target RAM 48 KB @ 0x20000000
Flash memory Internal bank 0
Base address 0x0
Flash size 508 KB
_01_rx_00.hex
x1 x2 x4
1 2 3 4 5 6 7 8 9 A B C D E F ASCII
00030 00 00 00 00 00 00 00 00 FF 00 00 00 01 01 00 00
00040 03 01 00 00 03 01 00 00 00 00 00 00 5D 1A 00 00
00050 03 01 00 00 03 01 00 00 03 01 00 00 03 01 00 00
00060 15 1A 00 00 00 00 00 00 03 01 00 00 85 22 00 20
00070 5F 1A 00 00 03 01 00 00 03 01 00 00 03 01 00 00
00080 03 01 00 00 03 01 00 00 03 01 00 00 00 00 00 00
00090 00 00 00 00 03 01 00 00 03 01 00 00 03 01 00 00
000A0 00 00 00 00 00 00 00 00 00 00 00 00 91 0A 00 00
000B0 03 01 00 00 BD 12 00 00 75 11 00 00 00 00 00 00
000C0 03 48 85 46 00 F0 34 FA 00 48 00 47 1D 33 01 00
000D0 28 76 00 20 04 20 71 46 08 42 02 D0 EF F3 09 80
000E0 01 E0 EF F3 08 80 71 46 00 4A 10 47 B9 10 00 00
000F0 04 48 80 47 04 48 00 47 FE E7 FE E7 FE E7 FE E7
00100 FE E7 FE E7 91 18 00 00 C1 00 00 00 10 B5 43 1A
00110 93 42 09 D2 83 18 88 18 03 E0 40 1E 01 78 5B 1E
LOG
- ROMTbl[0] @ E00FF000
- ROMTbl[0][0]: E000E000, CID: B105E00D, PID: 000BB008 SCS
- ROMTbl[0][1]: E0001000, CID: B105E00D, PID: 000BB00A DWT
- ROMTbl[0][2]: E0002000, CID: B105E00D, PID: 000BB00B FPB
- Executing init sequence ...
- Initialized successfully
- Target interface speed: 4000 kHz (Fixed)
- J-Link found 1 JTAG device. Core ID: 0x0BB11477 (None)
- Connected successfully
Opening data file [D:\PDK\BQB_tx_01_rx_00.hex] ...
- Data file opened successfully (107988 bytes, 1 range, CRC of data = 0x241D43C3, CRC of file = 0xC9E81465)
Erase, program and verify target
Connected
Core Id: 0x0BB11477
Speed: 4000 kH

图 12: 开始烧录

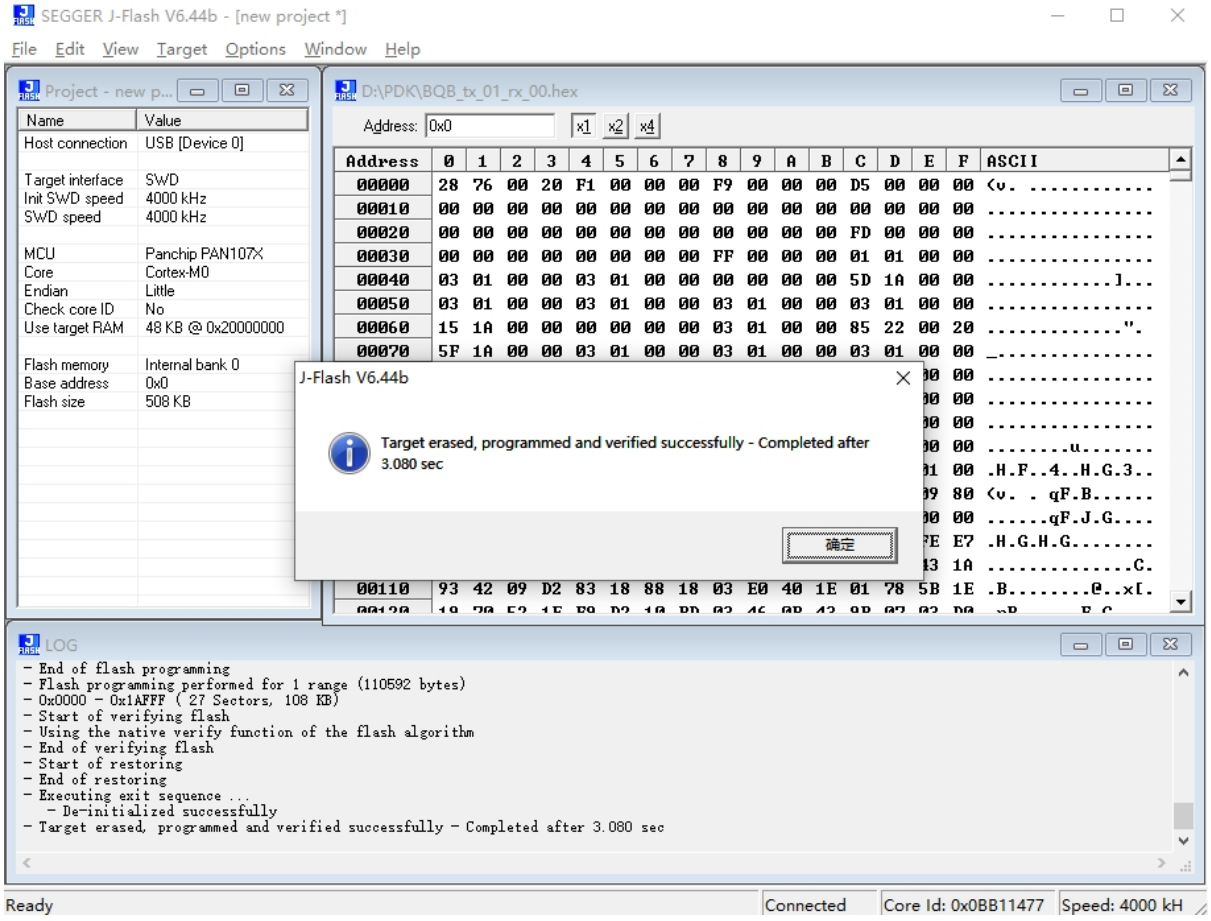

SEGGER J-Flash V6.44b - [new project *]
File Edit View Target Options Window Help
Project - new p...
Name
Value
Host connection
USB [Device 0]
Target interface
SWD
Init SWD speed
4000 kHz
SWD speed
4000 kHz
MCU
Panchip PAN107X
Core
Cortex-M0
Endian
Little
Check core ID
No
Use target RAM
48 KB @ 0x20000000
Flash memory
Internal bank 0
Base address
0x0
Flash size
508 KB
D:\PDK\BQB_tx_01_rx_00.hex
Address: 0x0
x1 x2 x4
Address 0 1 2 3 4 5 6 7 8 9 A B C D E F ASCII
00000 28 76 00 20 F1 00 00 00 F9 00 00 00 D5 00 00 00
00010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00020 00 00 00 00 00 00 00 00 00 00 00 00 FD 00 00 00
00030 00 00 00 00 00 00 00 00 FF 00 00 00 01 01 00 00
00040 03 01 00 00 03 01 00 00 00 00 00 00 5D 1A 00 00
00050 03 01 00 00 03 01 00 00 03 01 00 00 03 01 00 00
00060 15 1A 00 00 00 00 00 00 03 01 00 00 85 22 00 20
00070 5F 1A 00 00 03 01 00 00 03 01 00 00 03 01 00 00
00110 93 42 09 D2 83 18 88 18 03 E0 40 1E 01 78 5B 1E
J-Flash V6.44b
Target erased, programmed and verified successfully - Completed after 3.080 sec
确定
LOG
- End of flash programming
- Flash programming performed for 1 range (110592 bytes)
- 0x0000 - 0x1AFFF (27 Sectors, 108 KB)
- Start of verifying flash
- Using the native verify function of the flash algorithm
- End of verifying flash
- Start of restoring
- End of restoring
- Executing exit sequence ...
- De-initialized successfully
- Target erased, programmed and verified successfully - Completed after 3.080 sec
Ready
Connected
Core Id: 0x0BB11477
Speed: 4000 kH

图 13: 烧录完成

5.4 Panchip 2.4G OTA 工具

Panchip 2.4G OTA 是 Shanghai Panchip Microelectronics Co.,Ltd. 为 PAN1070 SDK 提供的开发工具集合，目前包含如下功能：

- 配合 2.4G 主机对连接的从机设备进行 OTA 升级

下载单 exe 程序的工具。

该程序为单独一个 exe 可执行文件；

启动时间相对较慢。

下载文件夹程序工具。

该程序为一个文件夹，里面的 exe 可执行程序需要依赖文件夹内的库文件。

启动时间较快。

5.4.1 1. OTA 升级

配合鼠标项目支持 2.4G OTA 的主机对从机设备进行 OTA 升级。

1.1. OTA 界面

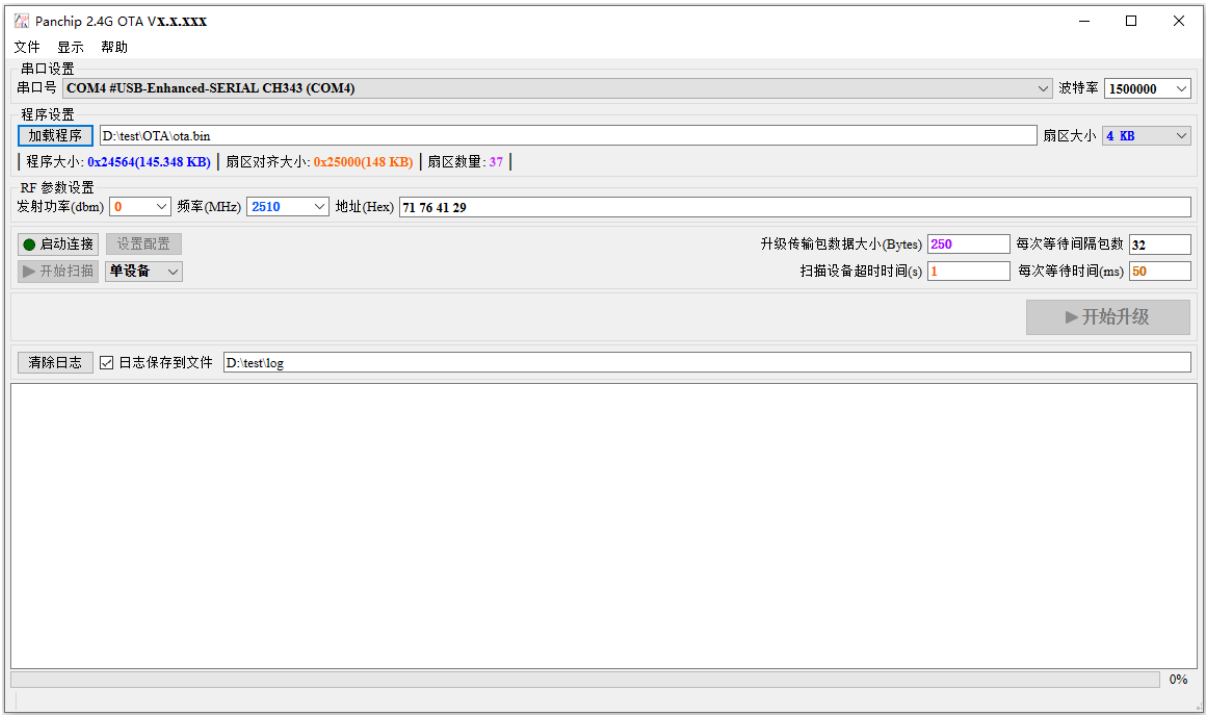


图 14: OTA 界面

1.2. 软件使用方法

1. 加载 OTA 程序文件，“*.bin” 格式程序文件。
2. 选择 2.4G 主机对应的串口号。
3. 启动连接，打开与主机的连接，并设置 RF 参数配置。
4. 如果修改了 RF 参数，需要点击设置配置，重新设置 RF 配置。

5. 设置选择扫描设备模式，点击扫描设备，必须成功扫描到设备才能进行 OTA 升级
6. 点击开始升级，进行 OTA 升级。

详细使用说明，可以点击软件菜单的 **帮助**-> **查看帮助文档**。

Chapter 6

开发工具

6.1 PAN107x Toolbox 工具箱

PAN107x Toolbox 是 Shanghai Panchip Microelectronics Co.,Ltd. 为 PAN1070 SDK 提供的开发工具集合，目前包含如下功能：

- 进行简单的 RF 测试
- 支持项目 DFU 固件升级
- PAN107x 芯片引脚规划
- 使用 PAN107x 芯片检测当前环境信号强度并显示

下载单 exe 程序的工具。

该程序为单独一个 exe 可执行文件；

启动时间相对较慢。

下载文件夹程序工具。

该程序为一个文件夹，里面的 exe 可执行程序需要依赖文件夹内的库文件。

启动时间较快。

6.1.1 功能界面选择

图 1: 显示切换功能界面

6.1.2 1. RF 测试

使用此功能，配合 PAN107x 芯片 RF 测试固件，通过串口通讯可以测试 PAN107x 芯片的**发射**、**接收收包数**、**单载波发射**等功能。

配合鼠标项目支持 USB 通信可以测试 PAN107x 芯片的**发射**、**接收收包数**、**单载波发射**等功能。

1.1. RF 测试界面

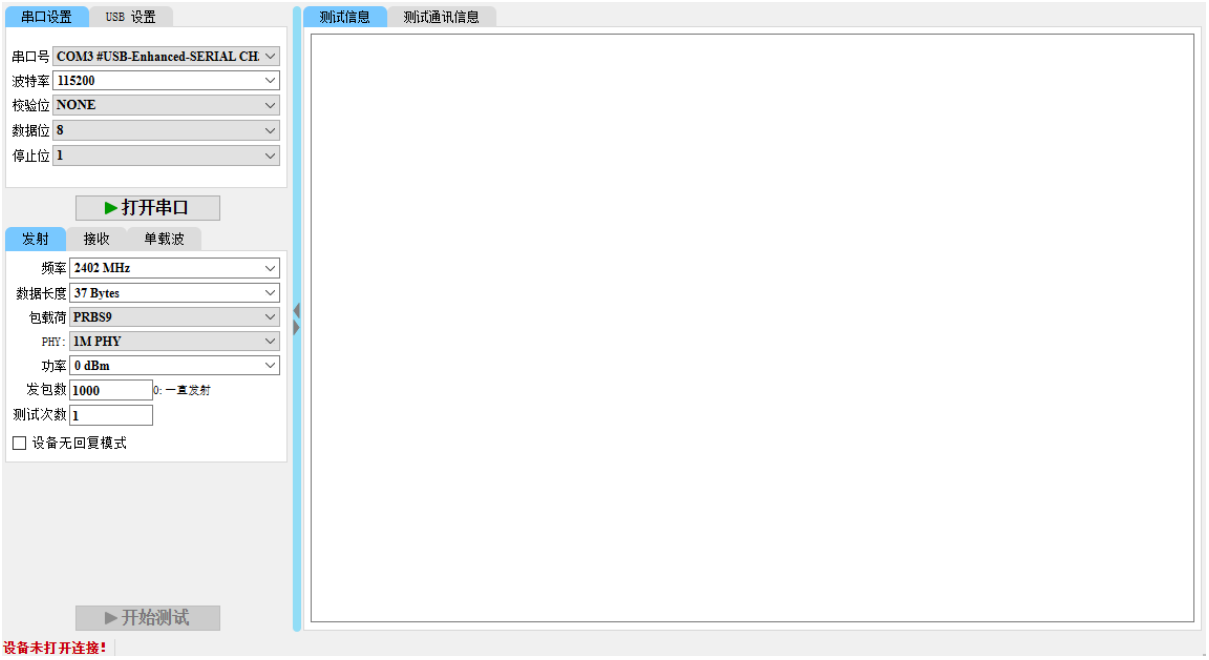


图 2: RF 测试界面

1.2. 软件使用方法

1. 从工具的**帮助**->RF **测试** Demo **程序**，**导出**或**下载**得到 RF 测试 Demo 程序固件，然后下载到 PAN107x 芯片。
2. 使用 USB **转串口设备**将 PAN107x **芯片**通过串口与电脑可以进行通讯连接。
3. 打开 PAN107xToolBox 工具的 RF **测试**界面。
4. 选择 USB **转串口设备**的串口打开连接，选择对应的测试模式，即可以进行测试。

详细使用说明，可以点击软件菜单的 **帮助**-> **查看帮助文档**。

6.1.3 2. 设备固件升级

使用此功能，可以通过 USB 实现固件升级。

2.1. 设备固件升级界面

2.2. 软件使用方法

1. 先将鼠标方案设备通过 USB 线连接到电脑。并确保设备进入 bootloader USB 通讯模式。
2. 打开 PAN107xToolBox 工具的 DFU 界面。
3. 选择添加固件程序，确保成功载入固件程序。
4. 确认选择的 USB **设备**为需要进行升级的设备。
5. 点击 “** 开始下载” 则会进入下载流程进行设备固件升级。

详细使用说明，可以点击**帮助**-> **查看帮助文档**。

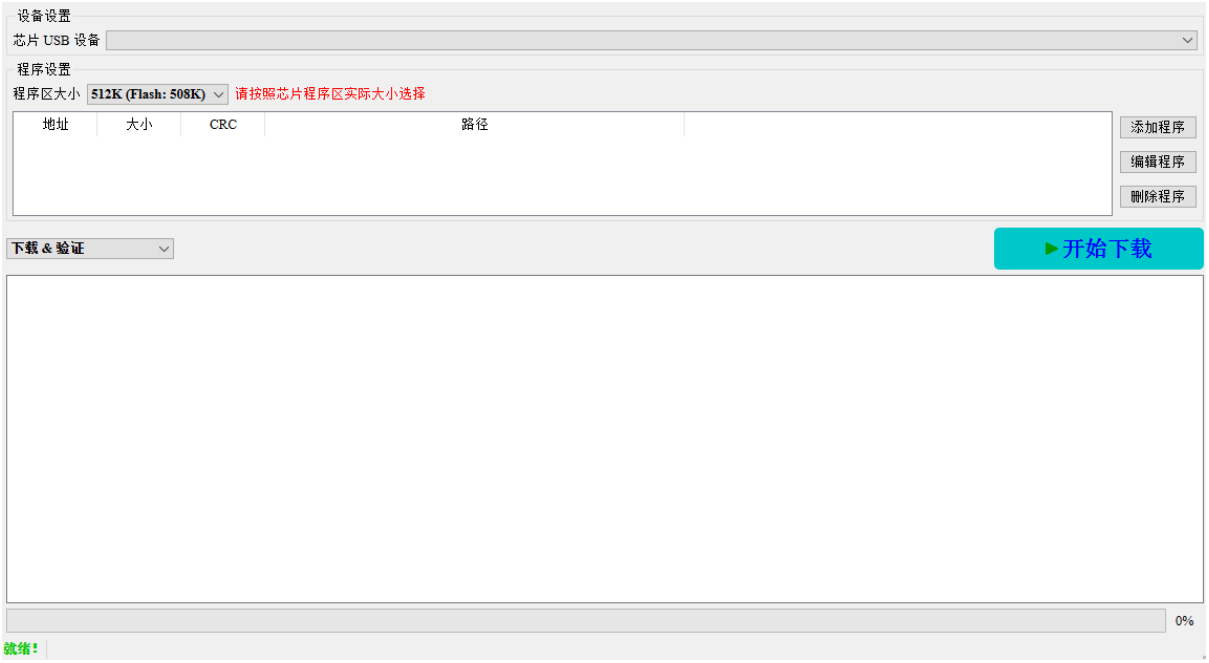


图 3: DFU 界面

6.1.4 3. 芯片引脚规划

使用此功能，方便用户快速查看特定型号芯片引脚功能。也可以对 PAN107x 特定型号芯片的引脚进行选择分配，导出分配报告，方便在应用中规划引脚。

3.1. 引出脚配置界面

3.2. 软件使用方法

1. 打开 PAN107xToolBox 工具的**引出脚配置**界面。
2. 选择使用的芯片**型号**。
3. 可以通过选择最左边的功能列表的功能，会在中间显示对应功能支持的配置引出脚选项。
4. 也可以直接在右边的芯片图示中选择对应的引脚的功能。
5. 选择完成，可以通过点击 PDF 按钮图标，则会生成配置报告 pdf 文档。

详细使用说明，可以点击**帮助**-> **查看帮助文档**。

6.1.5 4.RF 信号采集

使用此功能，配合 PAN107x 芯片 RF 信号采集固件，可以通过 PAN107x 采集当前环境中指定频点的信号强度并进行显示。

4.1. RF 信号采集界面

4.2. 软件使用方法

1. 需要预先下载对应 RF 信号采集固件到 PAN107x 芯片。
2. 使用 USB **转串口设备**将 PAN107x **芯片**通过串口与电脑可以进行通讯连接。
3. 打开 PAN107xToolBox 工具的 RF **信号采集**界面。

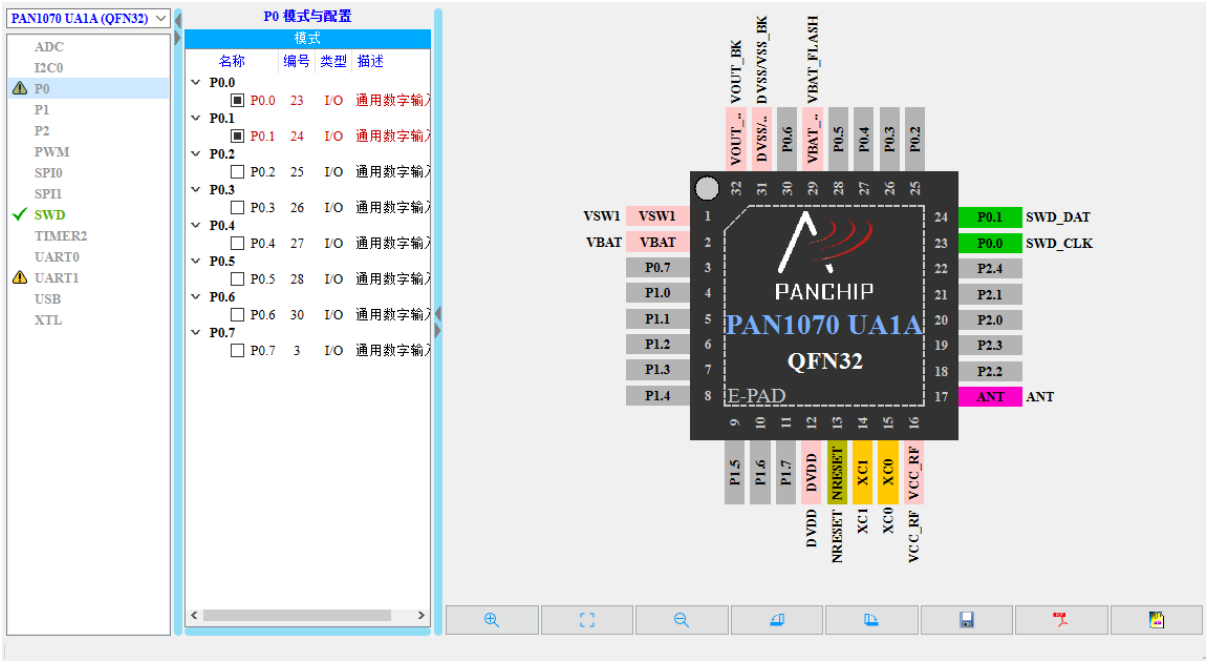


图 4: 引出脚配置界面

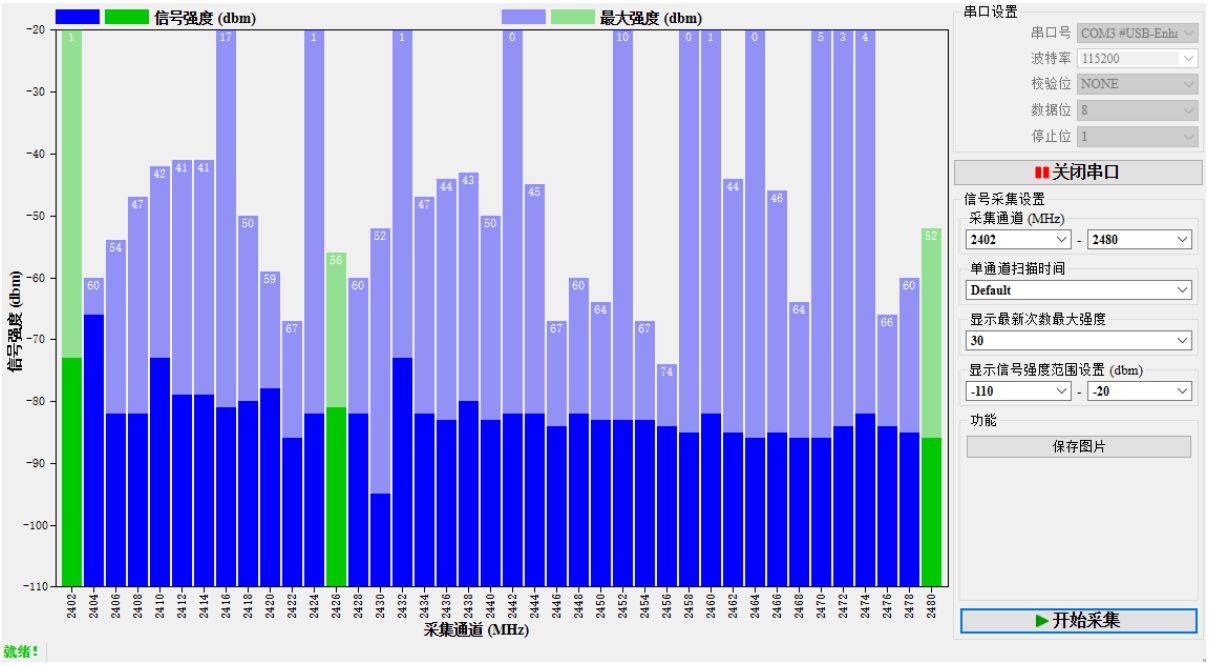


图 5: RF 测试界面

4. 选择 USB **转串口设备**的串口打开连接，选择需要采集的对应频点的信号强度，然后点击开始。

详细使用说明，可以点击软件菜单的 **帮助**-> **查看帮助文档**。

Chapter 7

其他文档

PAN107x SoC 相关的其他文档请参考：

- PAN107x BQB Test Report
- PAN1070 功耗测试报告
- PAN10xx 系列蓝牙兼容性测试报告

Chapter 8

更新日志

8.1 PAN10XX NDK v0.6.0

PAN10XX Nimble DK `v0.6.0` (2024-08-05) 已发布：

8.1.1 1. SDK

nimble

- 优化 Nimble 蓝牙 Host 层代码，新增 Mem Pool 机制，以支持 PAN101x 芯片小 Memroy 的场景
- 优化 Nimble 系统启动流程，将蓝牙初始化部分与 OS 初始化部分解耦，使得蓝牙功能关闭后，系统仍可正常运行
- 优化 Nimble DeepSleep 低功耗流程，并新增 Standby Mode 1 和 Standby Mode0 支持
- 修复 DeepSleep 唤醒瞬间出现大电流的问题
- 新增一些与低功耗相关的 API 接口（os_lp.h）：
  - `soc_lptmr_cycle_get()`
  - `soc_32k_clock_freq_get()`
  - `soc_lptmr_uptime_get_ms()`
  - `soc_busy_wait()`
  - `soc_reset_reason_get()`
  - `soc_stbm1_gpio_wakeup_src_get()`
  - `soc_enter_standby_mode_0()`
  - `soc_enter_standby_mode_1()`
- 修改 FreeRTOS `vApplicationIdleHook()` 函数的定义，使其有返回值，以满足低功耗场景的使用需求
- 更新 bootloader，优化 2.4G OTA 升级时间与稳定性

Panchip HAL

- Panchip Spark BLE Controller Library：
  - 优化时序以降低功耗
  - 优化 RF 性能

- Panchip PRF (2.4G Private RF) Library:
  - 新增 TRX 最小转换时间配置接口
  - 优化 B250K 通信
- BSP:
  - 更新量产芯片校准参数载入流程
  - 更新 DMA Driver，移除不必要的接口
  - 更新 FMC Driver，优化一些接口的实现以提示 Flash 操作的可靠性
  - 更新 I2C Driver，移除一些不必要的代码
  - 更新 Timer Driver，修复某些场景下 Timer0 无法正常使用的问题
  - 新增各个外设的 HAL 层 Driver，它们是基于各自底层 Driver 抽象出来的较上层的 Driver，简化了外设的使用方法

Samples

- bluetooth：
  - ble_multi_role:
    - 新增两主三从演示
  - bleprph_enc:
    - 支持 pan101x 芯片 (新增 pan101x 芯片工程)
  - bleprph_hr:
    - 新增非连接广播演示开关（默认不使能）
    - 新增更新 PHY 开关（默认不使能）
  - bleprph_throughput:
    - 新增手机测试方法，简化调试流程
- low_power:
  - deepsleep_gpio_key_wakeup (新增):
    - 演示 SoC 进入 DeepSleep 状态，并通过 GPIO 按键将其唤醒
  - deepsleep_gpio_pwm_wakeup (新增):
    - 演示 SoC 进入 DeepSleep 状态，使用外部 PWM 波形通过 GPIO 将其唤醒
  - deepsleep_pwm_waveform_generator (新增):
    - 演示 SoC 在 DeepSleep 状态下输出 PWM 波形，并使用 APB HW Timer0 定时唤醒并修改 PWM 波形周期和占空比
  - deepsleep_slptmr_wakeup (新增):
    - 演示 SoC 进入 DeepSleep 状态，并通过 SleepTimer 定时器将其唤醒
  - standby_m1_gpio_key_wakeup (新增):
    - 演示 SoC 进入 Standby Mode 1 状态，并通过 GPIO 按键将其唤醒
  - standby_m1_slptmr_wakeup (新增):
    - 演示 SoC 进入 Standby Mode 1 状态，并通过 SleepTimer 定时器将其唤醒
  - standby_m0_p02_key_wakeup (新增):
    - 演示 SoC 进入 Standby Mode 0 状态，并通过 WKUP (P02) 按键将其唤醒

  - multiple_wakeup_source (新增):
    * 演示 SoC 多种唤醒源、多种低功耗模式之间的切换
- os_debug:
  - os_rtt_logging (新增):
    * 演示使用 Segger Jlink RTT 的方式打印 Log 的方法
- peripheral:
  - adc_read_multiple_channels (新增):
    * 演示 ADC HAL Driver 的使用方法
  - gpio_digital_input_interrupt (新增):
    * 演示使用 GPIO HAL Driver 实现中断方式的 GPIO 输入检测功能
  - gpio_digital_input_polling (新增):
    * 演示使用 GPIO HAL Driver 实现查询方式的 GPIO 输入检测功能
  - gpio_output_open_drain (新增):
    * 演示使用 GPIO HAL Driver 实现 GPIO 开漏（Open-Drain）输出功能
  - gpio_output_push_pull (新增):
    * 演示使用 GPIO HAL Driver 实现 GPIO 推挽（Push-Pull）输出功能
  - gpio_simple_convenient_apis (新增):
    * 演示 GPIO 底层 Driver 中提供的几个简单好用的接口
  - i2c_master_dma_receive (新增):
    * 演示使用 I2C HAL Driver 实现 I2C Master 的 DMA 接收功能
  - i2c_master_int_send (新增):
    * 演示使用 I2C HAL Driver 实现 I2C Master 的中断发送功能
  - i2c_master_poll_send (新增):
    * 演示使用 I2C HAL Driver 实现 I2C Master 的查询发送功能
  - i2c_slave_dma_send (新增):
    * 演示使用 I2C HAL Driver 实现 I2C Slave 的 DMA 发送功能
  - i2c_slave_int_receive (新增):
    * 演示使用 I2C HAL Driver 实现 I2C Slave 的中断接收功能
  - i2c_slave_poll_receive (新增):
    * 演示使用 I2C HAL Driver 实现 I2C Slave 的查询接收功能
  - pwm (新增):
    * 演示 PWM HAL Driver 的使用方法
  - spi_master_dma_send_receive (新增):
    * 演示使用 SPI HAL Driver 实现 SPI Master DMA 方式先发后收功能
  - spi_master_int_send_receive (新增):
    * 演示使用 SPI HAL Driver 实现 SPI Master 中断方式先发后收功能
  - spi_master_poll_send_receive (新增):
    * 演示使用 SPI HAL Driver 实现 SPI Master 查询方式先发后收功能
  - spi_slave_dma_receive_send (新增):

        * 演示使用 SPI HAL Driver 实现 SPI Slave DMA 方式先收后发功能
    - spi_slave_int_receive_send (新增):
        * 演示使用 SPI HAL Driver 实现 SPI Slave 中断方式先收后发功能
    - spi_slave_poll_receive_send (新增):
        * 演示使用 SPI HAL Driver 实现 SPI Slave 查询方式先收后发功能
    - timer_basic (新增):
        * 演示 Timer HAL Driver 的计数（计时）功能
    - timer_capture (新增):
        * 演示 Timer HAL Driver 的输入捕获功能
    - uart_dma (新增):
        * 演示使用 UART HAL Driver 实现 UART DMA 方式收发数据功能
    - uart_fifo (新增):
        * 演示使用 UART HAL Driver 实现 UART 中断方式收发数据功能
    - wdt (新增):
        * 演示 WDT (Watchdog) HAL Driver 的使用方法
    - wwdt (新增):
        * 演示 WWDT (Window Watchdog) HAL Driver 的使用方法
- security:
    - fw_encryption (新增):
        * 演示芯片通过固件加密、硬件解密的机制保护 Flash 关键代码的方法
- solutions:
    - ble_accelerometer (新增):
        * 演示通过蓝牙自定义服务将 EVB 加速度传感器数据上报到对端的方法
    - ble_app_uart (新增):
        * 演示蓝牙从机串口透传功能，从机设备和手机或主机设备连接后可以和串口模块进行数据透传
    - ble_spi_tft_lcd (新增):
        * 演示使用蓝牙自定义复位将字符数据通过蓝牙传输到芯片中并显示在 EVB OLED 屏幕中的方法
    - ble_hid_selfie:
        * 支持 pan101x 芯片 (新增 pan101x 芯片工程)
    - ble_hid_uart_mult_roles:
        * 修复 UART1 作为串口透传功能不正常的问题
        * 修复从机 Notify 到主机的异常
        * 修复 Watchdog 使能后会异常复位的问题
- 其他：
    - 所有例程更新 configuration 配置，修复一些错误，并新增一些配置选项

8.1.2 2. HDK

- 新增 PAN1070UA1A 图纸、设计源文件、生产文件至 v1.5 版本

8.1.3 3. MCU

- 更新 ADC 例程:
  - 使能 ADC Buffer 以提升采样准确性
- 更新 PAN101x/PAN107x 芯片 Keil Flash 烧录算法（FLM）文件，提高稳定性 (位于 mcu_misc 目录)
- 底层 Driver 例程移除一些不必要的代码

8.1.4 4. DOC

- 更新文档中心主页，新增 PAN101x 规格书等下载链接，并优化一些表述
- 更新 `PAN10xx 硬件参考设计`文档
- 更新 `BLE MULTI ROLE` 例程文档
- 更新 `BLE Peripheral Throughput Test` 例程文档
- 新增 Low Power 低功耗相关例程文档 8 篇
- 新增 Peripheral 外设相关例程文档 25 篇
- 新增 Security 固件加密例程文档 1 篇
- 更新 `BLE Accelerometer` 方案例程文档
- 更新 `BLE APP UART` 方案例程文档
- 更新 `BLE HID Selfie` 方案例程文档
- 更新 `BLE Spi Tft Lcd` 方案例程文档
- 更新 `NDK App 开发指南`文档，更新 PAN1070 的功耗测试结果，并新增 PAN1010 的功耗测试结果
- 修复一些文档中的描述错误

8.1.5 5. TOOLS

- 更新 Panchip 2.4G OTA 工具至 `v0.0.004` 版本:
  - 修改通讯速率支持最大到 2000000 Hz
  - 添加 OTA 过程传输与等待配置
  - 修改 OTA 传输数据回复协议支持
- 更新量产烧录工具 `PAN10xx Download Tool` 至 `v0.0.005` 版本:
  - 修复功能-> 读取 PHY 问题
  - 更新 PAN-LINK 支持外部 AD1 IO 输出控制 RST 输出实现，控制芯片的 RST 脚功能
  - 更新下载加密信息功能，配合 SDK 发布的加密信息文件，实现芯片 Flash 程序加密
  - 修复烧录擦除芯片结果 log 显示错误问题
  - 修复下载特殊程序有失败的问题
- 更新调试工具目录:
  - 更新 JLink 工具中的 FLM 文件至最新版本

8.1.6 6. ISSUES

遗留问题

- `BUG #873`: 兼容问题–peripheral_ota—与小米手机 11 配合升级，小米 11 安装的 nRF Conenct 软件版本是 4.28 时，无法升级

8.2 PAN1070 NDK v0.5.0

PAN1070 Nimble DK `v0.5.0` (2024-06-07) 已发布：

注：PAN1070 NDK 现已兼容 PAN101x 系列芯片。

8.2.1 1. SDK

nimble

- 优化 Nimble Samples Configuration 配置选项
- 添加系统看门狗功能
- 优化例程 SRAM 资源消耗
- 新增对 PAN101x 芯片的支持
- 优化温度自动检测流程，修复与 App 层同时使用 ADC 产生冲突的问题

Panchip HAL

- Panchip Spark BLE Controller Library：
  - 更新 PHY 驱动，优化 RF 性能
  - 新增动态修改 Tx Power 接口
- Panchip PRF (2.4G Private RF) Library:
  - 更新 PHY 配置，优化 RF 性能
  - 更新 Tx Power
- BSP:
  - 更新 FT 校准信息载入流程，并节约一些 SRAM
  - 新增对 PAN101x 芯片的支持，包括 Driver 及 Flash 烧录算法等
  - 更新 ADC Driver，使其能够兼容不同 FT 版本的芯片
  - 更新 CLK Driver，优化 WDT/WWDT 时钟源选择接口
  - 更新 I2C Driver，修复 I2C 例程无法正常工作的问题
  - 更新 LP Driver，关闭 DeepSleep 状态下 Flash 的供电，以节约功耗
  - 新增 Power Driver，用于 Nimble 相关例程中根据当前温度动态修改芯片各个供电配置

Samples

- 蓝牙:
  - `bluetooth/bleprph_hr`
    * 支持 pan101x 芯片 (新增 pan101x 芯片工程)
  - `solutions/ble_rgb_light`
    * 支持 pan101x 芯片 (新增 pan101x 芯片工程)
- 其他:
  - `mcu_boot`: 更新 bootloader，优化 2.4G OTA 功能
  - 所有例程优化 configuration 配置框架

8.2.2 2. HDK

- 新增 PAN1010S9FA 核心板图纸

8.2.3 3. MCU

- 更新 ADC 例程:
  - 优化使用流程，使采样结果更准确
- 更新 PAN1070_PRF_TRX 开发指南.pdf 文档
- 新增 PAN101x 芯片 Keil Flash 烧录算法（FLM）文件 PAN101X_252KB_FLASH.FLM (位于 mcu_misc 目录)
- 所有例程:
  - 更新芯片校准信息载入流程及相关 Log 输出
  - 增加对 PAN101x 芯片的支持（源码与 PAN107x 共用，但新增 PAN101x Keil Project，注意有少数例程因 PAN101x 引脚限制无法支持）

8.2.4 4. DOC

- 更新文档中心主页，新增 PAN101x 相关内容介绍
- 更新 `NDK 快速入门指南`文档，并增加对 PAN101x 的描述
- 新增 `NDK Configuration` 配置开发指南文档
- 更新 `PAN107x EVB 介绍`文档，将其更名为 `PAN10xx EVB 介绍`，并新增对 PAN101x 相关介绍
- 更新 `PAN107x 硬件参考设计`文档，将其更名为 `PAN10xx 硬件参考设计`，并新增对 PAN101x 相关介绍
- 更新 `BLE Peripheral HR` 例程文档，新增对 PAN101x 芯片支持情况的描述
- 优化 `BLE RGB Light` 例程文档，新增对 PAN101x 芯片支持情况的描述
- 更新 `NDK Mcu Boot` 开发指南文档，新增生成签名文件的环境配置介绍
- 更新 量产烧录工具说明文档，增加对 PAN101x 芯片的描述
- 新增 `RF TEST` 说明文档，介绍 RF 测试固件的使用方法
- 新增 `JFlash` 烧录说明文档，介绍使用 Segger J-FLash 工具烧录固件到 PAN107x SoC 的方法
- 新增 `Panchip 2.4G OTA` 工具说明文档，介绍 2.4G OTA 的主机对从机设备进行 OTA 升级的方法

- 更新 NDK 常见问题（FAQs）文档，阐述某些情况下，芯片正常工作的时候，使用 JLink (SWD) 无法（或很难）再次烧录程序的原因及解决方法
- 更新所有文档中与特定芯片相关的描述，新增对 PAN101x 芯片支持情况的描述
- 修复一些表述上的问题

8.2.5 5. TOOLS

- 新增 Panchip 2.4G OTA 工具，用于配合 OTA 主机对从设备进行 OTA 升级
- 更新量产烧录工具 PAN10xx Download Tool 的介绍：
  - 新增对 PAN101x 支持情况介绍
- 更新 RF 测试固件至 v002，优化性能
- 更新调试工具目录：
  - 新增 ForceEraseVectorTable_PAN107x.bat 脚本，可擦除芯片 Flash 上的 Vector Table，阻止程序正常执行（详见开发指南/FAQs 文档相关说明）

8.2.6 6. ISSUES

新增问题

- BUG #873: 兼容问题–peripheral_ota—与小米手机 11 配合升级，小米 11 安装的 nRF Conenct 软件版本是 4.28 时，无法升级

遗留问题

- BUG #802: PRF OTA，带 OTA 作为 client，利用 ota 升级其他设备偶尔会失败

8.3 PAN1070 NDK v0.4.0

PAN1070 Nimble DK v0.4.0 (2024-04-03) 已发布：

8.3.1 1. SDK

nimble

- 优化 Keil 工程编译信息，清除编译警告
- 优化 app_config_spark.h 的配置选项和结构层次
- 优化 SoC Power Domain，进而优化功耗（支持定时检测温度并根据当前温度优化芯片 Power 配置）

Panchip HAL

- Panchip Spark BLE Controller Library：
  - 优化 RF 性能
  - 修复 RCL 作为低功耗时钟时的连接问题
  - 修复 RF PHY 问题
- Panchip PRF (2.4G Private RF) Library:

  - 更新 DCOC 校准流程
  - 修复频点设置接口 Bug
  - 更新 Tx Power 档位
  - 更新 g_250k deviation 为 170k
  - 优化读 rssi 接口, 增加 rssi 全局变量
- BSP:
  - 更新 FT 校准信息载入流程
  - 更新 ADC Driver，新增一些 API 接口，以简化 ADC 使用流程
  - 更新 CLK Driver，新增选择 PWM 时钟源的 API 接口
  - 更新 I2C Driver，修复潜在的问题
  - 更新 PWM Driver，新增一些易用的 API 接口
  - 更新 TIMER Driver，修复一些问题
  - 修复一些寄存器名称错误

Samples

- 蓝牙:
  - `bluetooth/ble_multi_role` (新增)
    - BLE 多主多从例程
  - `bluetooth\bleprph_throughput` (新增)
    - BLE 从机吞吐率例程
  - `bluetooth\bleprph_distance` (新增)
    - BLE 距离测试例程（心跳服务以及支持不同 phy 切换）
  - `bluetooth/peripheral_hr`
    - 修复多次断连后重连死机问题
- 方案:
  - `solutions/ble_vehicles_key`
    - 适配 RSSI 波形显示
    - 修复不同手机配对多次产生 cccd settings 条目不够最终导致音量调整失效的情况
- 其他:
  - `pan107x_mcu_boot`: 更新 bootloader，新增 2.4G OTA 功能

8.3.2 2. HDK

- 移除过期的测试板图纸

8.3.3 3. MCU

- 新增 PRF_OTA_CLIENT 例程:
  - 2.4G OTA 客户端工程，演示 2.4G OTA 功能
- 新增 PRF_TX_SAMPLE_UI 和 PRF_RX_SAMPLE_UI 例程:

  - 带屏幕显示的 2.4G 距离测试例程
- 移除 mcu_misc 目录下的旧版本 Keil Flash 烧录算法（FLM）文件，新增 PAN107X_508KB_FLASH.FLM
- 更新所有 Keil 工程默认使用的 FLM 文件

8.3.4 4. DOC

- 新增 `ble_multi_role` 例程文档
- 新增 `bleprph_distance` 例程文档
- 新增 `bleprph_throughput` 例程文档
- 新增 `mcu_samples_doc/PAN1070_PRF_UI 距离测试说明.pdf` 例程文档
- 更新 `ndk_develop_environment_intro` 介绍文档，更新 FLM 文件说明
- 更新 `ndk_mcu_boot` 开发指南文档，新增生成签名文件的环境配置介绍
- 优化文档目录架构，拆分了 NDK 和 ZDK 文档，使得文档架构更加清晰

8.3.5 5. TOOLS

- 更新工具箱工具 `PAN107x ToolBox` 至 `v0.0.004`:
  - 新增 USB 通信兼容
- 新增 RF 测试固件:
  - 新增 PAN107x RF 测试固件
  - 新增 PAN107x RSSI VIEWER 测试固件
- 新增 JLink v6.44b 软件
  - 支持 PAN107x 芯片的 Jlink 命令行调试，JFlash 烧录等

8.3.6 6. ISSUES

新增问题

- `BUG #802`: PRF OTA，带 OTA 作为 client，利用 ota 升级其他设备偶尔会失败

8.4 PAN1070 NDK v0.3.0

PAN1070 Nimble DK `v0.3.0` (2024-01-19) 已发布:

8.4.1 1. SDK

nimble

- 新增 Bootloader，并默认在各例程中使能，可通过 App 工程配置文件禁用
- 新增 SMP BT 子系统，以支持蓝牙 OTA 功能
- 更新 nimble ble host 一些细节
- 新增蓝牙低功耗定向优化配置，用于一些特殊的功耗测试场景

Panchip HAL

- Panchip Spark BLE Controller Library:
  - 优化 SRAM 占用
  - 优化 MD
  - 新增运动健康协议支持
  - 新增 DTM 支持
  - 优化 adv Rx timeout 至 60us
  - 优化 RF Post Tx Time
  - 更新 PHY 参数
  - 修复断连信息未及时清除问题
  - 修复 0x28 断连问题
- Panchip PRF (2.4G Private RF) Library:
  - 更新 PHY 参数
  - 完善一些 API 接口
- BSP:
  - 更新 FT 校准信息载入流程
  - 更新 ADC Driver，新增一些 API 接口，以简化 ADC 使用流程
  - 优化系统启动流程
  - 修复一些引脚定义错误
  - 修复 GPIO_DB 相关结构体名称错误的问题
  - 修复低功耗 Driver 的潜在问题
  - 移除一些不必要的代码以避免潜在的编译错误风险

演示例程

- 蓝牙:
  - `bluetooth/peripheral_hr_ota` (新增)
    - 演示蓝牙 OTA 功能
- 方案:
  - `solutions/ble_mouse` (新增)
    - BLE 鼠标方案
  - `solutions/multimode_mouse` (新增)
    - 多模鼠标方案
  - `solutions/multimode_mouse_dongle` (新增)
    - 多模鼠标配套 Dongle 方案
  - `solutions/ble_prf_sample` (新增)
    - BLE & 2.4G 双模方案
- 其他:
  - 所有例程均添加了 OTA 支持，并提供了 3 种编译和配置模式:

    * Bare Metal
    * OTA in Bootloder
    * OTA in App

8.4.2 2. HDK

- 新增 PAN107x EVB 底板图纸、设计源文件、生产文件 v1.1

8.4.3 3. MCU

- 更新 ADC 演示例程:
  - 优化 ADC Convert Test、VDD/4 Test、Temperature Test 流程，使用新的接口以简化使用
- 更新 CLK 演示例程:
  - 修复一些问题
- 更新 LP 演示例程:
  - 重命名例程名称为 LowPower
- 新增 PRF_Template_SAMPLE 例程:
  - 2.4G 模板工程，以方便用户快速创建自己的 2.4G 工程
- 其他:
  - 修复 GPIO_DB 相关结构体名称错误的问题
  - 修复例程生成的 Image 名称与预期不一致的问题

8.4.4 4. DOC

- 新增 `ble_mouse` 例程文档
- 新增 `multimode_mouse` 例程文档
- 新增 `multimode_mouse_dongle` 例程文档
- 新增 `ble_prf_sample` 例程文档
- 更新 `mcu_samples_doc/PAN1070_ADC 例程说明.pdf` 例程文档，以匹配工程最新的修改
- 新增 `ndk_mcu_boot` 开发指南文档，介绍 NDK 的 Bootloader
- 新增 `pan107x_evb_intro` 硬件资料文档，介绍 PAN107X EVB 相关内容
- 更新 `pan107x_hw_reference_design` 硬件参考设计文档，修改了一些具体描述
- 新增 `toOlbox_intro` 工具箱工具介绍文档

8.4.5 5. TOOLS

- 更新量产烧录工具 `PAN107x Download Tool` 至 `v0.0.002`:
  - 修复一些潜在问题
- 新增工具箱工具 `PAN107x ToolBox v0.0.003`:
  - 新增引出脚界面
  - 新增 RF 信号采集界面

8.5 PAN1070 NDK v0.2.0

PAN1070 Nimble DK `v0.2.0` (2023-11-19) 已发布：

8.5.1 1. SDK

nimble

- 更新 BLE Controller，优化一些内部流程并修复一些问题
- 新增获取 MAC 地址的接口

Panchip HAL

- 新增载入 Hardware Calibration 校准参数的接口
- 优化 WDT 接口，扩大 WDT Reset 的复位范围
- 更新 RF Lib，优化 2.4G 通信流程

演示例程

- `ble_cent_prph`（新增）：演示蓝牙主从一体功能
- `ble_central`（新增）：演示蓝牙主机功能
- `bleprph_hr`（新增）：演示蓝牙从机功能，包含 GATT 服务：HR (Heart Rate)，连接订阅服务后，会上报虚拟的心率值
- `bleprph_enc`（新增）：演示外设以及加密配对功能，可以和主机示例进行对测
- `ble_hid_selfie`（新增）：自拍解决方案，通过蓝牙 HID 控制手机拍照
- `ble_panchip_cte_beacon`（新增）：Panchip 蓝牙定位标签方案，通过发送特定的广播数据，实现蓝牙定位功能
- `ble_rgb_light`（新增）：蓝牙 RGB 灯控方案，演示 BLE RGB 灯与手机 APP 进行连接，通过 APP 控制 RGB 灯的亮度与颜色
- `ble_hid_uart_mult_roles`（新增）：蓝牙串口透传解决方案，演示蓝牙 hid 串口透传功能，支持 1 主 1 从
- `ble_vehicles_key>`（新增）：蓝牙车钥匙解决方案，演示基于 HID 服务的自动连接服务

8.5.2 2. HDK

- 新增 PAN1070 UA1A EVB 图纸、设计源文件、生产文件

8.5.3 3. MCU

- 更新 LP 低功耗例程，优化 CPU Retention and Remap 流程
- 更新 2.4G 例程及对应文档，演示更多的通信模式
- 更新各个底层 Driver 例程，增加初始化阶段载入芯片校准信息的流程

8.5.4 4. DOC

- 新增 `ble_cent_prph` 例程文档
- 新增 `ble_central` 例程文档
- 新增 `bleprph_enc` 例程文档
- 新增 `bleprph_hr` 例程文档
- 新增 `ble_hid_selfie` 例程文档
- 新增 `ble_hid_uart_mult_roles` 例程文档
- 新增 `ble_pcte_beacon` 例程文档
- 新增 `ble_rgb_light` 例程文档
- 新增 `ble_vehicles_key` 例程文档
- 新增 `NDK App` 开发指南文档
- 新增 `PAN107x` 硬件参考设计文档
- 新增 量产烧录说明文档

8.5.5 5. TOOLS

- 新增量产烧录工具 `PAN107x Download Tool`
- 新增 Testbox RF 测试固件

8.6 PAN1070 NDK v0.1.0

PAN1070 Nimble DK `v0.1.0` (2023-10-24) 已发布：

8.6.1 1. SDK

NDK 软件开发框架基于 Keil + FreeRTOS + NimBLE，其中：

- Keil 是 SDK 支持的软件开发环境
- FreeRTOS 是一个开源实时操作系统（RTOS），用于配合 NimBLE 实现蓝牙应用
- NimBLE 是一个开源低功耗蓝牙（BLE）5.1 协议栈，其实际上是 Apache Mynewt 项目的一部分

解决方案

- `esl`: ESL 价签方案演示例程，支持外部 SPI Flash 存储、EPD 墨水屏、低功耗模式、RF 通信等功能。

8.6.2 2. HDK

目前版本提供了如下硬件相关资料：

- PAN107B QFN40 测试板图纸、设计源文件、生产文件

8.6.3 3. MCU

目前版本提供了如下 MCU 裸机 Keil 例程及相关文档:

- ADC
- CLK
- CLKTRIM
- DebugProtect
- DMA
- EFUSE
- FMC
- GPIO
- I2C
- LP
- PRF_B250K_RX
- PRF_B250K_TX
- PWM
- SPI
- TIMER
- UART
- USB_HID
- WDT
- WWDT

8.6.4 4. DOC

目前版本提供了如下文档:

- NDK 快速入门指南
- NDK 开发环境介绍
- NDK 整体框架介绍
- Nimble 简介
- PAN107x 硬件参考设计指南
- ESL 电子货架标签方案例程说明
- MCU 底层外设驱动例程说明
- 低功耗开发指南
- NDK RAM 使用情况分析以及优化指南

8.6.5 5. TOOLS

目前版本提供了如下工具:

- 串口工具（PC 工具）
- Air Sync Debugger（手机测试软件安卓 APK）

- Google Home（手机测试软件安卓 APK）
- nRF Connect（手机测试软件安卓 APK）
- nRF Mesh（手机测试软件安卓 APK）
- Siliconlabs Bluetooth Mesh（手机测试软件安卓 APK）

8.6.6 6. 已知问题

- MCU USB_HID 例程暂未通过测试