

艾克姆科技

nRF52840 开发指南-上册

[基于 Nordic 蓝牙低功耗/802.15.4 Soc-nRF52840]

艾克姆科技飞字团队

[2018.12.1]

官方店铺: <https://acmemcu.taobao.com>

官方论坛: <http://930ebbs.com>

版权所有：艾克姆科技，引用请注明出处

本文档技术支持负责人：强光手电

[本文档以艾克姆科技 IK-52840DK 开发套件为硬件平台，通过原理分析和实验程序讲解以及实验演示，让读者以最短的时间掌握 nRF52840 的开发]

修订历史记录

Revision Records

日期 Date	版本 Version	编制 Written By	审核 Checked By	说明 Explanation
2017.2.2	A	强光手电	彭震	初建
2018.5.1	B	强光手电	彭震	以 SDK15.0 为基础重写编写。对章节进行了重新规划，对原理部分进行了更深入的分析。
2019.11.10	C	强光手电	彭震	更新为 SDK16.0 的库。

第一章：开发板硬件描述

IK-52840DK 是艾克姆科技设计的低功耗蓝牙系列开发套件之一，开发板以 Nordic 的 nRF52840-QIAA 为主芯片，nRF52840 是当前挪威奥斯陆（Nordic Semiconductor）推出的最高端的多协议芯片，其内核是 32 位 ARM® Cortex™-M4 处理器（带 FPU），64MHz，1MB 片内 Flash 和 256kB 片内 RAM、多种电源模式，尤其是支持 5V 电源供电，支持多协议，具备极低的功耗和优异的无线性能，完美适合于智能家居、面向支付和医疗用途的高级可穿戴应用，以及工业传感器和其它物联网(IoT)器件。

IK-52840DK 开发板采用了分离式的设计，板上设计了指示灯、按键、USB2.0 电路和 USB 转 UART、蜂鸣器、AD 采样电路、EEPROM、红外接收和发射电路、两种形式的触摸电路以及各种常用传感器、存储器及显示设备的专用接口，同时，也设计了跳线选择的电路，当我们不需要使用某个功能部件(如指示灯)时，可以通过跳线断开这部分电路，将这些 I/O 用于其他用途，极大地方便了我们灵活地使用 I/O。

1. 功能特点

- IK-52840DK 开发板完全兼容官方 nRF52840 开发板 nRF52DK (PCA10056)，根据用户需求的实际情况，增加了很多实用功能，如红外收发、两种类型的触摸按键、各种流行传感器接口、综合显示接口（支持 OLED、TFT 彩屏以及电子墨水屏）、TF 卡座、语音接口等。
- 开发板采用分离式的设计，nRF52840 模块可以方便的取下，nRF52840 芯片型号：nRF52840-QIAA (1M Flash, 256K Ram)。所有 IO 通过排针和排母的方式引出，开发板上的外设占用的 GPIO 可以通过短路帽连接或断开，极大地方便开发时对管脚的分配，同时，评估电流很方便，不拔下 nRF52840 模块通用可以测试电流。
- 开发板上设计了 NFC 天线接口(开发板配备了 NFC 天线)、USB 转串口电路、3V 有源蜂鸣器、光敏检测电路、触摸按键、38KHz 红外接收电路、电位器等功能部件。
- 开发板上设计 OLED、TFT 彩屏和电子墨水屏接口，无需接线即可使用 0.96/1.3 吋 OLED，1.8/2.2 吋 TFT 彩屏以及艾克姆科技发布的各种尺寸电子墨水屏，用户可根据需求选择合适的显示模块，避免接线的麻烦。
- 开发板上设计了各类流行传感器接口，无需接线即可使用 LIS3DH(百度手环开源项目用的加速度传感器)、LIS3DSH、MPU9250、PM2.5 传感器、MAX30100、MAX30102 心率传感器模块。这些传感器和蓝牙 BLE 配合，可拓展出各种应用，充分发挥自己的设计能力。
- 开发板上设计了外扩存储器接口，包含 TF 卡座（可插入各种容量的 TF 卡）和 Flash 模块接口（可接入 W25Q128 存储器以及 QSPI 接口的存储器）。

- 开发板上设计了语音模块接口，无需接线即可使用艾克姆科技发布的语音模块。
- 开发板上设计了ESP8266串口转Wifi模块接口，让蓝牙BLE进入网络，紧贴物联网应用。

2. 硬件原理

开发板实物图如下：

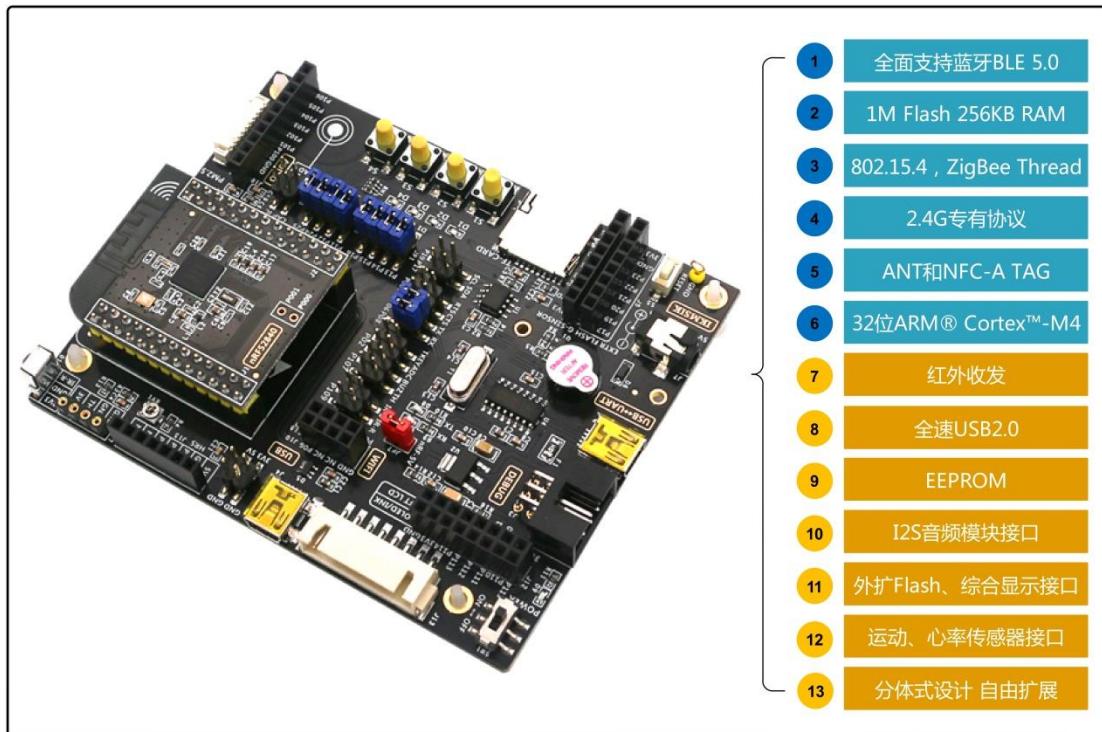


图 1-1：IK-52840DK 开发板实物图

❖ **特别说明：**如果您使用的传感器模块、显示模块、语音模块或存储模块不是从艾克姆科技购买的，接入开发板之前请务必检查模块信号和开发板的模块接口是否对应。

开发板硬件原理框图如下：

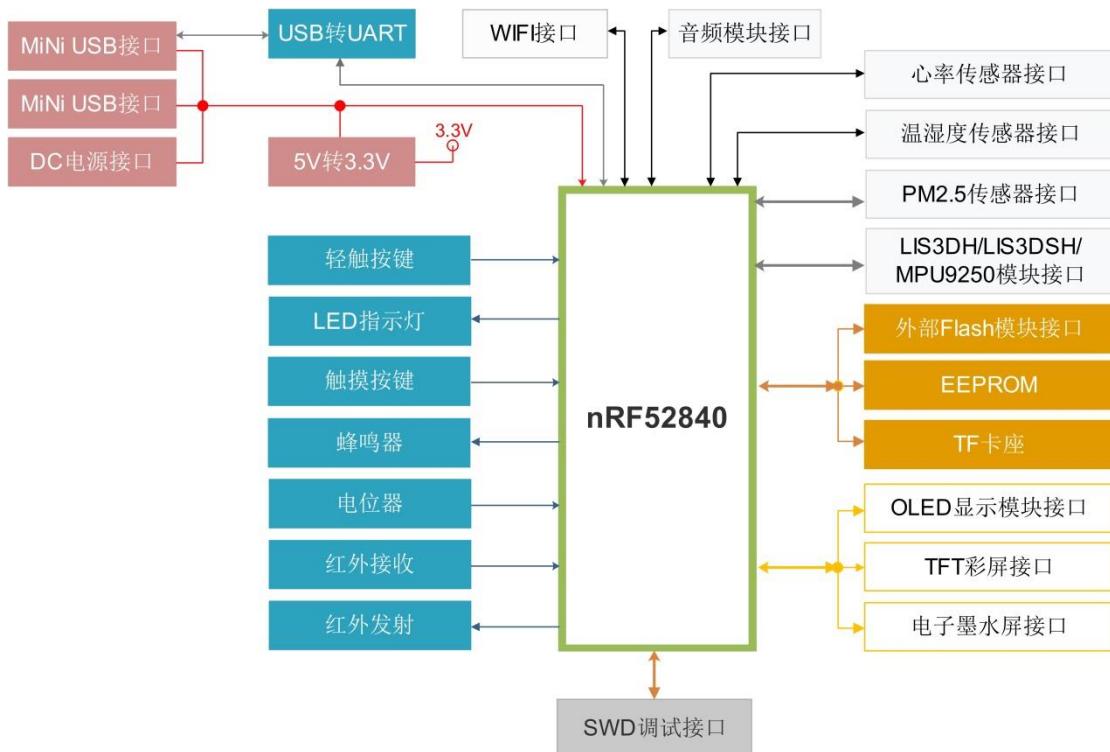


图 1-2: IK-52840DK 开发板硬件原理框图

2.1. nRF52840 特点

1. Bluetooth® 5, IEEE 802.15.4-2006, 2.4 GHz 收发器

- 接收灵敏度: BLE 1 Mbps 模式下 -95 dBm, 125 kbps BLE 模式 (远距模式) 下 -103 dBm
- 发射功率: (-20 ~ +8) dBm 可通过软件设置, 步进 4 dB。
- 兼容 Nordic 的 nRF52、nRF51、nRF24L 和 nRF24AP 系列芯片。
- 支持的数据速率
 - Bluetooth® 5: 2 Mbps、1 Mbps、500 kbps 和 125 kbps。
 - IEEE 802.15.4-2006: 250 kbps。
 - 专有 2.4 GHz: 2 Mbps、1 Mbps。
- 单端天线输出 (片上集成 balun)
- 128 位 AES/ECB/CCM/AAR 协处理器 (即时分组加密)
- TX (0 dBm) 时 4.8 mA 峰值电流, RX 4.6 mA 峰值电流。
- RSSI (1 dB 分辨率)。

2. Cortex-M4F 内核

- 32 位 ARM® Cortex®-M4 处理器 (带 FPU), 64 MHz。
- 强大的运算能力和浮点运算能力。
- 支持 SWD 调试。
- 片内 Flash 运行时 EEMBC CoreMark 分数 212。
- 片内 Flash 运行时 52 μA/MHz。
- DWT、ETM 和 ITM。

3. 超大的内存

片内 Flash 1M 字节，片内 RAM 256K 字节。

4. 灵活的电源管理

- 工作电源范围：1.7 V – 5.5 V。
- 片内 LDO 和 DC/DC 整流，具备自动低电流模式。
- 1.8~3.3V 整流输出为外部元件供电。
- 自动外设电源管理。
- 使用片内 64 MHz 振荡器实现快速唤醒。
- 0.4 μ A: 3V, System OFF 模式，RAM 数据不保持。
- 1.5 μ A: 3V, System ON 模式，RAM 数据不保持，RTC 唤醒。

5. 先进的片上接口

- USB 2.0 全速(12 Mbps)控制器。
- QSPI 32 MHz 接口。
- 高速 32 MHz SPI。
- Type 2 近场通讯 (NFC-A)。
- 支持触摸配对。
- 可编程外设互联 (PPI)。
- 48 个通用 I/O。
- EasyDMA。

6. Nordic SoftDevice 支持并发多协议。

7. 12 位、200 ksps ADC，8 个可编程增益的可配置通道。

8. 64 级比较器。

9. 15 级低功耗比较器，可将系统从 System OFF 模式唤醒。

10. 片内集成温度传感器。

11. 4 个 4 通道 PWM，带 EasyDMA。

12. 音频外设：I2S 和数字麦克风接口 (PDM)。

13. 5 个 32 位定时/计数器。

14. 4 个 SPI 主机/3 个 SPI 从机带 EasyDMA。

15. 2 个兼容 I2C 总线的 2 线主/从

16. 2 个 UART (CTS/RTS)，带 EasyDMA。

17. 正交解码器 QDEC。

18. 3 个实时计数器 (RTC)。

19. 封装 aQFN73, 7 x 7 mm。

20. 丰富的安全功能

■ ARM® TrustZone® Cryptocell 310 安全子系统。

- NIST SP800-90A 和 SP800-90B 兼容的随机数发生器。
- AES-128: ECB、CBC、CMAC/CBC-MAC、CTR、CCM/CCM*。
- Chacha20/Poly1305 AEAD 支持 128 和 256 位的密钥大小。

- SHA-1、SHA-2 高达 256 位。
 - 密钥相关的哈希运算消息认证码(HMAC)。
 - RSA 高达 2048 位的密钥大小。
 - SRP 高达 3072 位的密钥大小。
 - ECC 支持最常用的曲线，P-256 (secp256r1) 和 Ed25519/Curve25519 等等。
 - 使用派生密钥模型的应用程序密钥管理。
- Secure boot ready
- Flash 访问控制列表(ACL)。
 - Root-of-trust (RoT)。
 - Debug 控制和配置。
 - 访问端口保护(CTRL-AP)。
- 安全擦除。

2.2. 电源

开发板的工作电源可通过两种方式获取：

- USB (J8 和 J4): 通过 USB 接口给开发板供电，使用 MINI USB 数据线连接计算机和开发板后，计算机通过 USB 接口输出 5V 电源给开发板供电。
- 外部 DC 电源接口 (J7): 开发板上提供了一个外部电源的输入接口 J7，可以通过 5V 电源适配器供电。注意，一般情况下，使用计算机的 USB 输出的 5V 电源（每个 USB 口最大 500mA）给开发板供电完全足够，但是如果在开发板上连接了对电流需求较大的模块如 GPRS 模块等，需要单独给这些模块供电或者使用电源适配器通过外部 DC 电源接口供电。

2.3. 指示灯

开发板 LED 指示灯电路原理图如下：

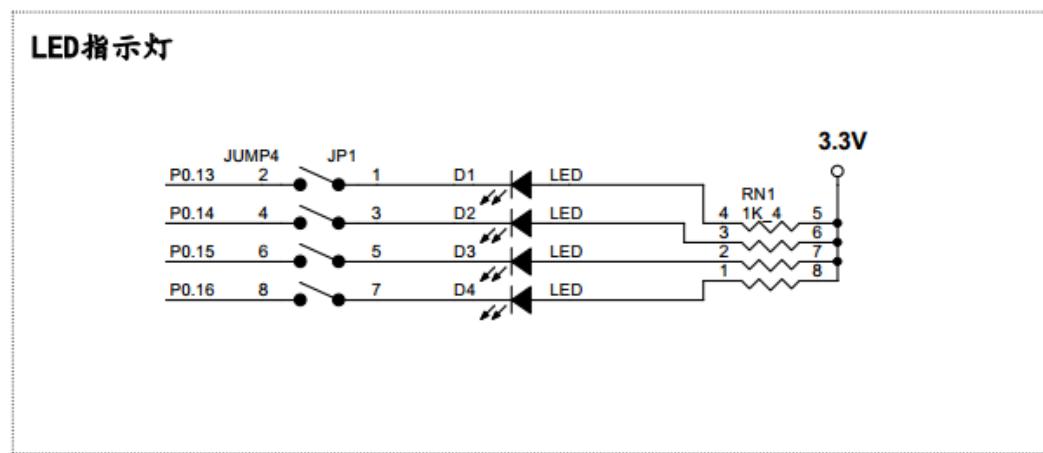


图 1-3: 指示灯电路

开发板上设计了四个用户指示灯 D1、D2、D3、D4，分别有 GPIO P0.13、P0.14、P0.15 和 P0.16 控制，当 GPIO 输出高电平时，LED 两端电压相等，LED 上没有电流流过，LED 处于灭状态，当 GPIO 输出低电平时，LED 两端存在正向压差，电流流过 LED，LED 被点亮。

◆ 特别说明：指示灯电路可以通过跳线断开或接通和 nRF52840 引脚之间的连接。

2.4. 按键和触摸按键

1. 轻触按键电路

开发板按键电路原理图如下：

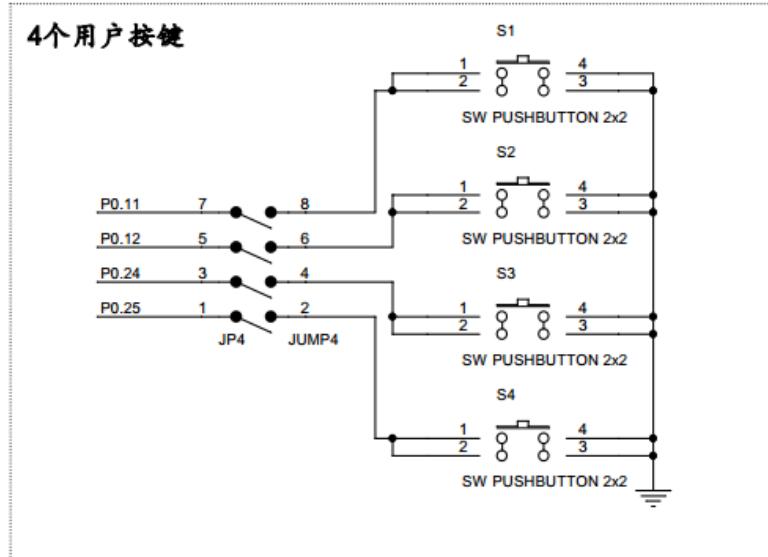


图 1-4: 轻触按键电路

开发板上设计了4个用户按键S1、S2、S3、S4，分别连接到GPIO P0.11、P0.12、P0.24和P0.25。GPIO用作输入时，需要通过软件打开GPIO的上拉电阻，用于确定IO口状态，当按键释放时由于上拉电阻的作用，IO输入为高电平，当按键按下时，IO输入为低电平。

2. 触摸按键

开发板触摸按键电路原理图如下：

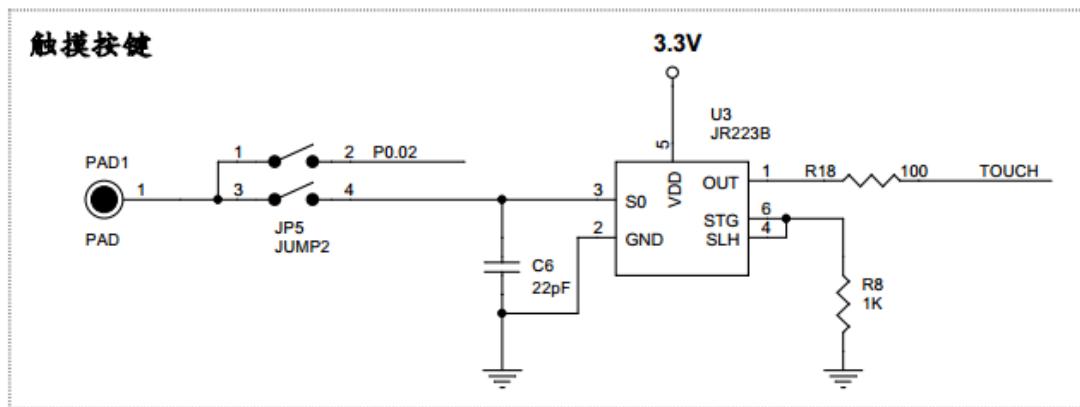


图 1-5: 触摸按键电路

触摸按键电路通过跳线JP5可以设置为使用触摸芯片检测和使用ADC采样电压检测。

■ 使用触摸芯片检测

JP5跳线的PIN3和PIN4短接，PAD连接到触摸芯片的输入引脚S0，由触摸芯片TTP223检测PAD上是否有触摸。

TTP223是电容式单键触摸按键IC，电压输入范围为2.0V~5.5V，它利用操作者的手指与触摸按键焊盘之间产生电荷电平来进行检测，通过监测电荷的微小变化来确定手指接近或者触摸到感应表面。没有任何机械部件，不会磨损，其感测部分可以放置到任何绝缘层（通

常为玻璃或塑料材料)的后面,很容易制成与周围环境相密封的键盘。

TTP223 的检测灵敏度可通过外部电容值(上图中的 C6)来调整。

■ 使用 ADC 采样电压检测

JP5 跳线的 PIN1 和 PIN2 短接, PAD 直接连接到 nRF52840 的模拟输入通道 AIN0(P0.02) nRF52840 通过采样 PAD 的电压判断 PAD 是否有触摸。

◆ 特别说明: 轻触按键和触摸按键电路可以通过跳线断开或接通和 nRF52840 引脚之间的连接。

2.5. 蜂鸣器

开发板蜂鸣器电路原理图如下:

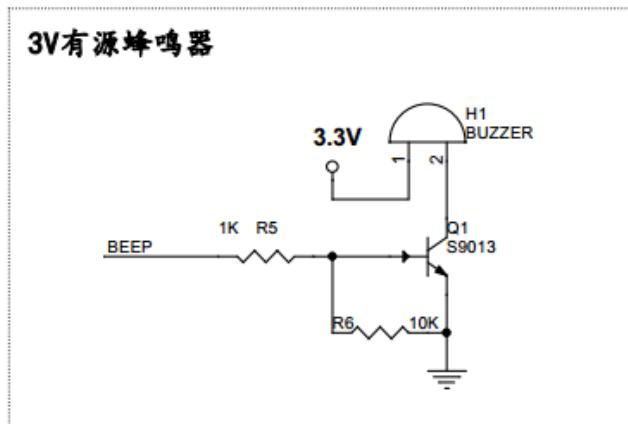


图 1-6: 蜂鸣器电路

开发板上使用的蜂鸣器是 3V 有源蜂鸣器。当 P1.07 输出高电平时,三极管导通,蜂鸣器鸣响。当 P1.07 输出低电平时,三极管截止,蜂鸣器停止鸣响。电路中的 R6 是为了保证三极管可靠的截止。

◆ 特别说明: 蜂鸣器电路可以通过跳线断开或接通和 nRF52840 引脚之间的连接。

2.6. 电位器检测

开发板电位器电路原理图如下:

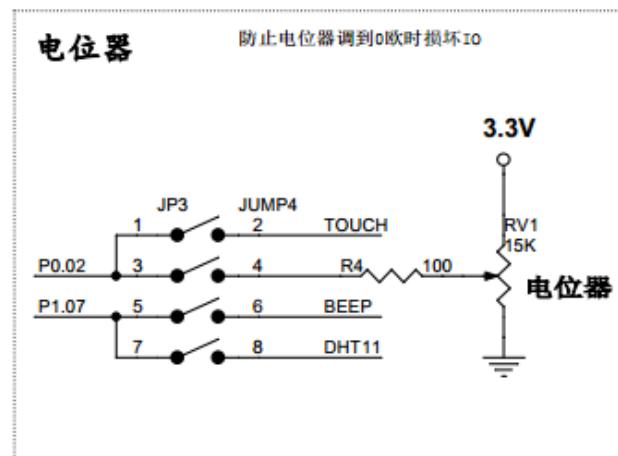


图 1-7: 电位器电路

电位器电路：

开发板上的电位器阻值为 15K，电位器抽头连接到 nRF52840 的引脚 P0.02（模拟输入通道 AIN0），通过旋转电位器上的旋钮调整抽头上的电压从而改变 P0.02 上的输入电压。

❖ 特别说明：电位器电路可以通过跳线断开或接通和 nRF52840 引脚之间的连接。

2.7. 红外接收和发射

开发板红外接收和发射电路原理图如下：

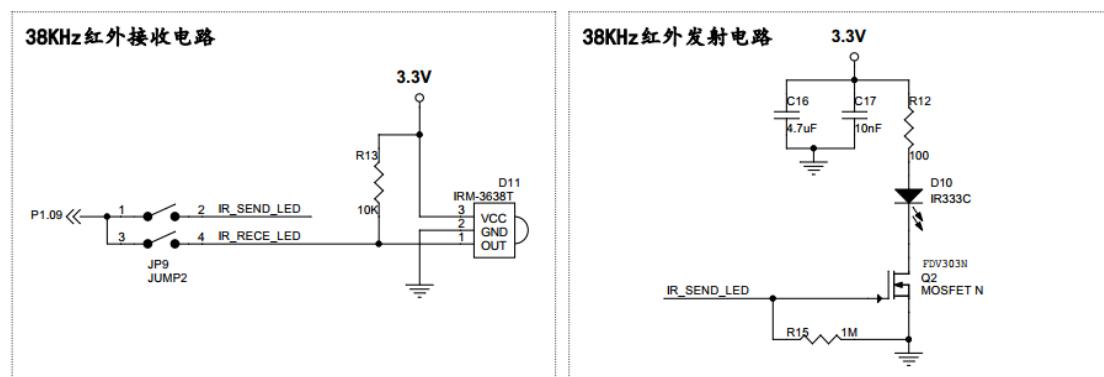


图 1-8：红外接收和发射电路

开发板上设计了载波频率 38KHz 的红外接收和发射电路，红外接收采用的是一体化接收头 IRM3836，IRM3836 集接收、放大、解调一体，接收到红外信号后进行处理并输出数字信号供单片机解析。发射电路采用的是红外发射管 IR333C，单片机可通过 PWM 或定时器生成 38KHz 载波和调制信号并通过红外发射管发射信号。

nRF52840 的 P1.09 通过跳线 JP9 选择连接红外接收或红外发射的电路，短接跳线 JP9 的 PIN3 和 PIN4 时，P1.09 连接到红外一体化接收头的信号输出引脚，工作时，通过软件解析红外一体化接收头输出的信号，将其转换为对应的编码。短接跳线 JP9 的 PIN1 和 PIN2 时，P1.09 连接到红外发送电路，通过 P1.09 输出红外编码信息并通过红外电路对外发射。

❖ 特别说明：红外接收和发射电路可以通过跳线断开或接通和 nRF52840 引脚之间的连接。

2.8. USB2.0 和 USB 转 UART

开发板 USB2.0 全速通讯电路和 USB 转 UART 电路原理图如下：

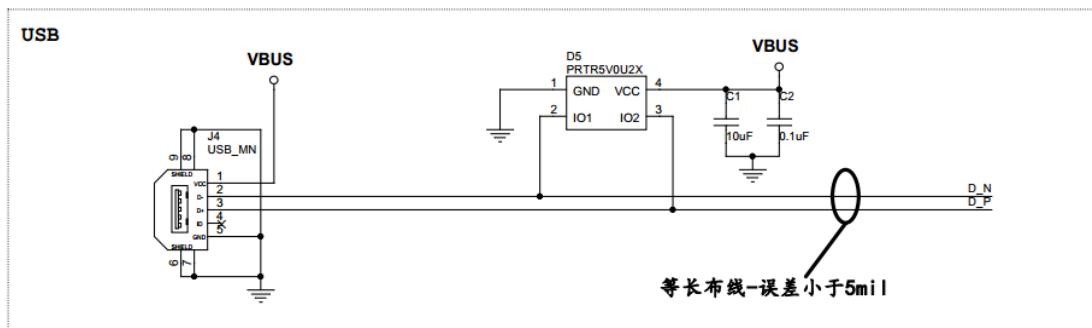


图 1-9: USB2.0 电路

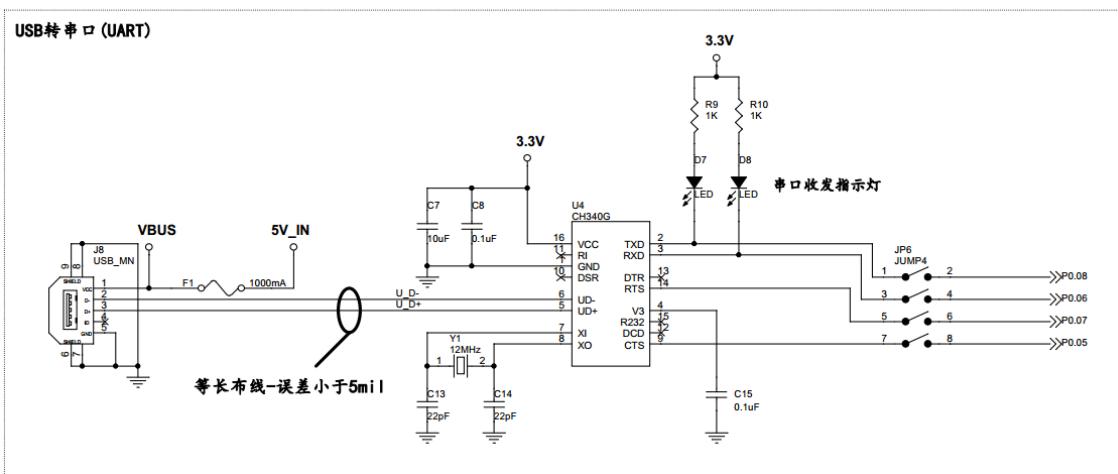


图 1-10: USB 转 UART 电路

其中 USB2.0 电路用于支持 USB2.0 全速通讯，USB 转 UART 采用 CH340 USB 转 UART 芯片，同时设计了 UART 硬件指示灯，方便从硬件角度观察串口通讯。

2.9. 音频模块接口

开发板 I2S 音频模块接口原理图如下：

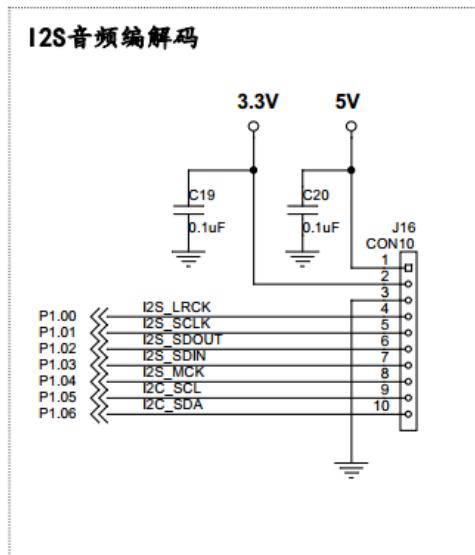


图 1-11: I2S 音频模块接口电路

I2S 音频模块接口包含 I2S 总线和 I2C 总线，以适应多种音频模块（很多音频模块需要通过 I2C 总线进行配置），可以实现音频播放和录音。

2.10. 显示模块接口

开发板显示模块接口原理图如下：

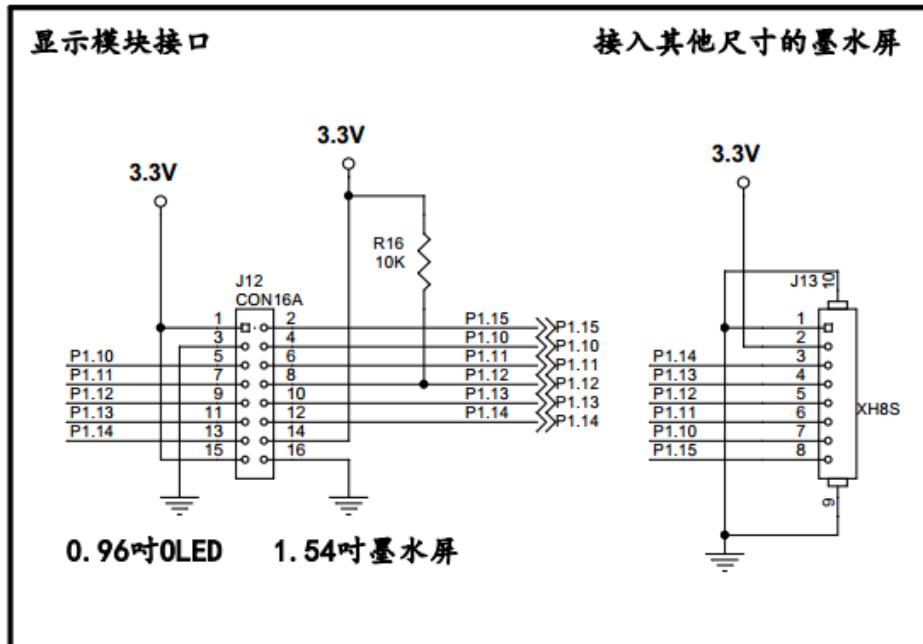


图 1-12：显示模块接口电路

开发板上设计 OLED、TFT 彩屏和电子墨水屏接口，其中 J12 右边是 0.96/1.3 英寸 OLED 显示模块和 1.54 英寸电子墨水屏接口，注意接入 OLED 显示模块时用了 7 孔（4、6、8、10、12、14、16），接入电子墨水屏使用 8 孔；J12 左边是 1.8 英寸 TFT 彩屏显示模块接口；J13 用于接入艾克姆科技发布的各种尺寸电子墨水屏。

2.11. 存储器

开发板外部存储相关原理图如下：

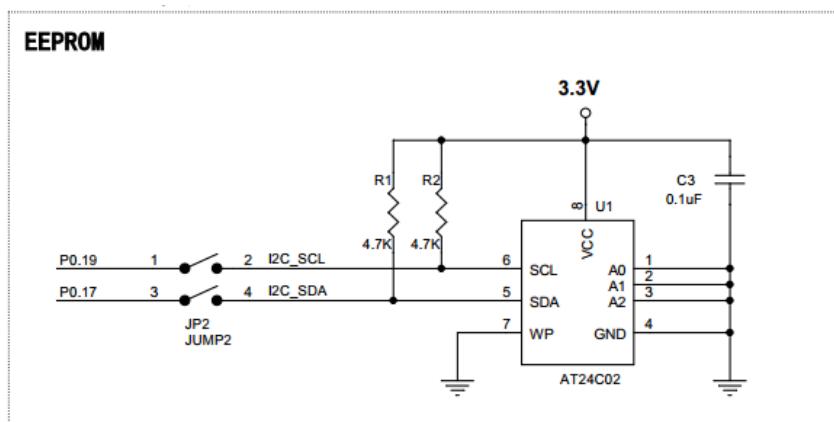


图 1-13：EEPROM 电路

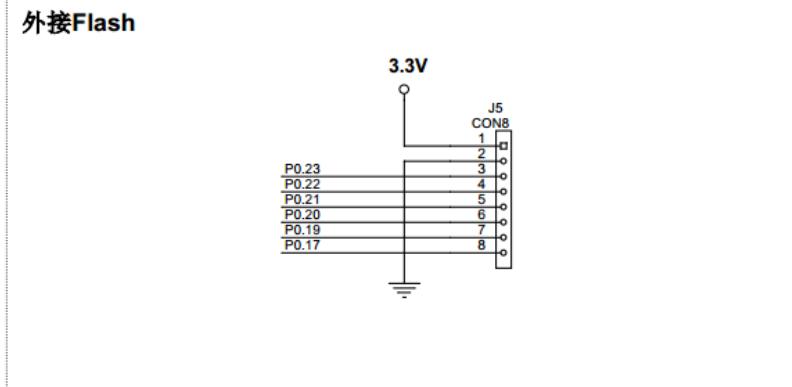


图 1-14: Flash 存储模块接口电路

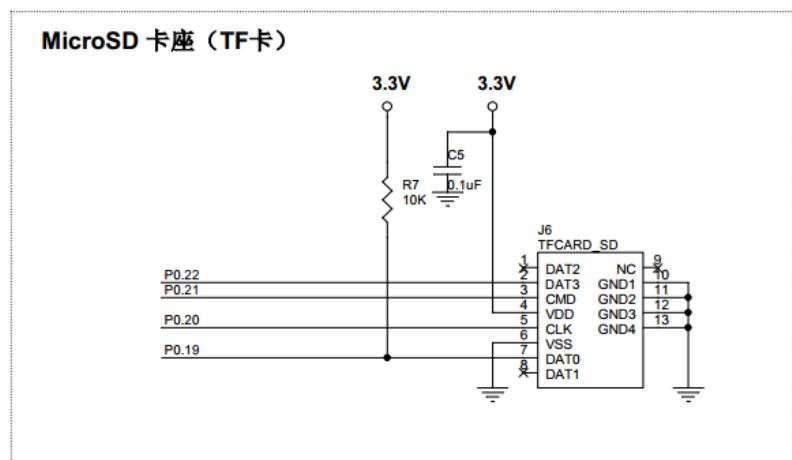


图 1-15: TF 卡电路

开发板上针对外扩存储，设计了 EEPROM (AT24C02)、外部 Flash 接口和 TF 卡座，其中外部 Flash 接口可以接入 W25Q128、铁电存储器和 QSPI 接口的存储器，TF 卡座可接入各种容量的 TF 卡。

2.12. Wifi 模块和 NFC 天线接口

开发板 Wifi 模块接口原理图如下：

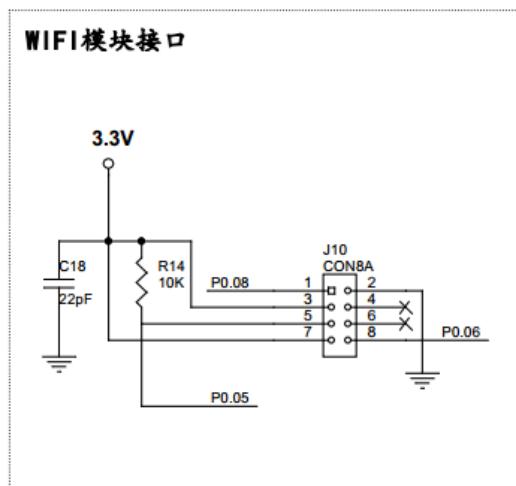


图 1-16: Wifi 模块接口电路

开发板上设计了 ESP8266 串口转 Wifi 模块接口，可直接安装 ESP-01 串口转 Wifi 模块和艾克姆科技的 ESP-12 串口转 WIFI 模块，实现 Wifi 通讯。

同时，开发板上也设计了 NFC 天线插座，可以插上 NFC 天线，实现 NFC 功能，在 BLE 应用中还可以通过 NFC 实现触摸配对，NFC 接口原理图如下。

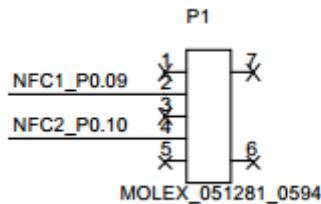


图 1-17: NFC 接口电路

NFC 占用了 nRF52840 的引脚 P0.09 和 P0.10，如果需要将这两个引脚作为普通 IO 使用，程序中需要进行相关配置。

2.13. 温湿度和 PM2.5 检测模块接口

开发板 PM2.5 传感器模块和 DHT11 温湿度传感器模块接口原理图如下：

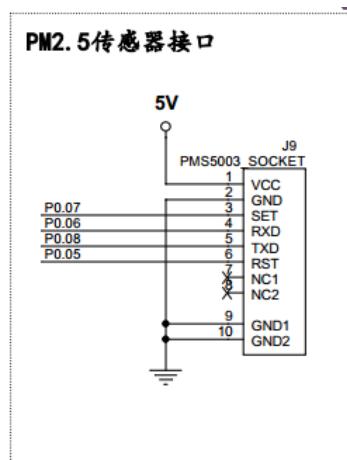


图 1-18: PM2.5 传感器接口电路

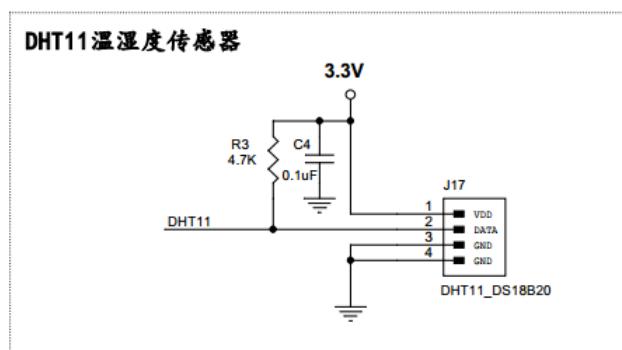


图 1-19: DHT11 温湿度传感器接口电路

开发板上设计了温湿度传感器 DHT11 和 PM2.5 检测模块的接口，可直接安装对应的模块实现温湿度检测和 PM2.5 检测的功能。

2.14. 运动及心率传感器接口

开发板运动传感器模块和心率传感器模块接口原理图如下：

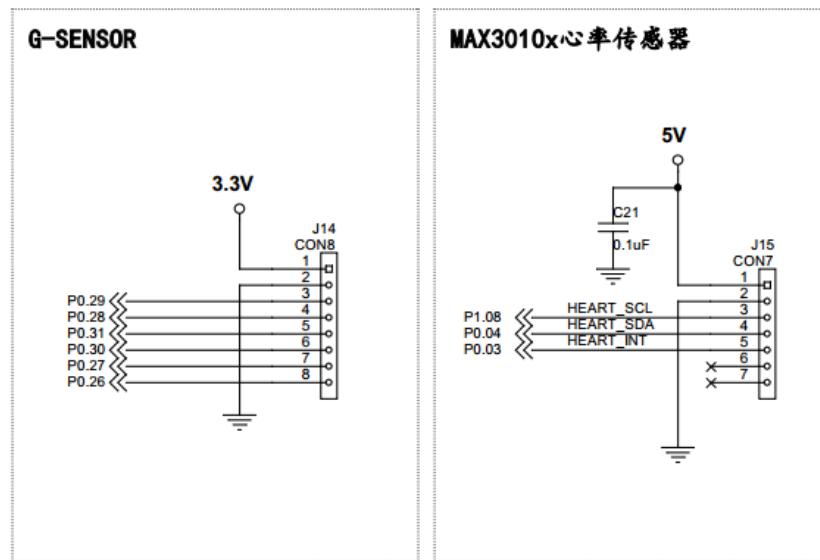


图 1-20：运动传感器和心率传感器模块接口电路

开发板上设计了运动传感器模块和心率传感器模块的专用接口，运动传感器模块接口可直接安装 LIS3DH(百度手环开源项目用的加速度传感器)模块、LIS3DSH 模块和 MPU9250 模块，心率传感器模块接口可安装 MAX30100 和 MAX30102 心率传感器模块。

2.15. 调试接口

开发板调试接口原理图如下：

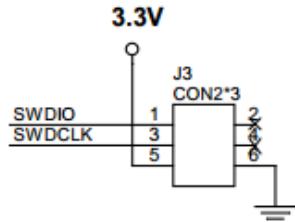


图 1-21：SWD 调试接口电路

开发板上设计了 SWD 调试接口电路，nRF52840 芯片 SWD 调试接口使用了 6 芯的连接器，实际只使用了 4 芯。SWD 调试接口信号定义如下表：

表 1-1：SWD 调试接口信号和引脚说明

序号	名称	对应 JLINK 引脚编号	类型	描述	对应的 nRF52840 引脚(QFAA)
1	SWDIO	7	输入/输出	双向数据线，主机和目标设备之间数据。	26
2	SWDCLK	9	输入	时钟，由主机产生。	25
3	Vref	1	输入	目标板电压检测，和目标板的工作电源连接。	工作电源
4	GND	GND	电源地	电源地。	电源地

3. 开发板硬件连接

开发板按照下图所示连接，其中：

- 仿真器：连接到计算机 USB 口，用于程序下载和仿真。
- 仿真器通过 JTAG-SWD 转接板和 6 芯排线连接到开发板。
- USB 数据线：连接计算机 USB 口，用于给开发板供电和串口通讯。
- 电源开关：拨到“ON”的位置，打开开发板电源。

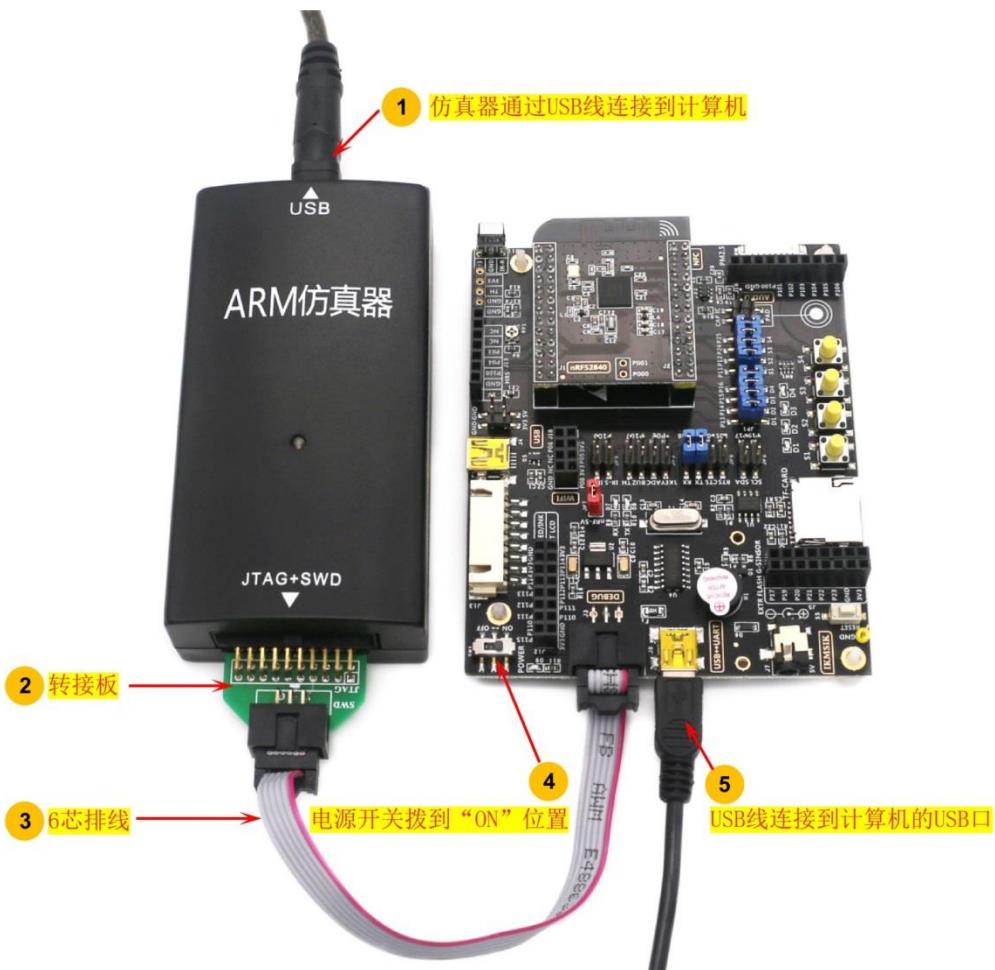


图 1-22：开发板硬件连接

第二章：开发工具

1. 硬件设备

表 2-1：硬件设备需求

名称	描述	需求
IK-52840DK 开发板	开发板	必须
JLINK 仿真器(推荐使用 V9)	仿真和下载程序	必须
JTAG-SWD 转接板、排线	JTAG 和 SWD 接口转接	必须
USB MINI 数据线	供电和串口通信	必须
IK-52Dongle	蓝牙 BLE 数据包捕获和协议分析	可选
传感器模块	实现各种额外的功能	可选

- ❖ 特别说明：IK-52Dongle 和传感器模块可以根据自己的需求配备。强烈建议配备一个 IK-52Dongle，IK-52Dongle 可以捕获和分析 BLE 的数据包，帮助我们理解 BLE 和在开发过程中分析问题，本书在讲解原理时很多地方都会用到 IK-52Dongle。可以这么说，不去分析数据包，很难深入地去理解 BLE。
- ❖ IK-52Dongle 的所需软件的安装，请参阅 IK-52Dongle 的配套资料。

2. 软件工具

表 2-2：软件工具需求

名称	描述	需求
win7/ win 10 系统	计算机系统	必须
MDK(版本不低于 5.25)集成开发环境	nRF52840 开发环境	必须
CH341SER.zip	USB 转串口驱动	必须
Nordic Semiconductor.nRF_DeviceFamily Pack.8.27.1.pack	nRF5_SDK_16.0 使用的 pack	必须
ARM.CMSIS.4.5.0.pack	SDK16.0 需要安装该版本	必须
WireShark	Dongle 需要安装的软件，和 nRF Sniffer 配合使用，抓包和协议分析	可选
nRF Connect	手机端软件，用于扫描、连接、读写设备等。	必须
nRF Toolbox	Nordic 手机端 APP 应用集，包含心率、接近等多个应用。	可选

第三章：搭建开发环境

1. 需要的工具软件

表 3-1：计算机需要安装的工具软件

序号	软件工具	描述
1	CH341SER.zip	USB 转串口驱动。
2	MDK5.27	Keil MDK 集成开发环境(版本不低于 5.25)。
3	NordicSemiconductor.nRF_DeviceFamilyPack.8.27.1.pack	nRF5_SDK_16.0 使用的 pack。
4	ARM.CMSIS.4.5.0.pack	SDK16.0 需要安装该版本
5	nRF-Command-Line-Tools_10_4_1_Installer_64.exe	Nordic 提供的 nRF5XX 开发工具链。
6	nRF5_SDK_16.0.0_98a08e2.zip	nRF5xx 软件开发包，无需安装，解压后即可使用。
7	nRF5_SDK_16.0.0_offline_doc.zip	SDK 离线文档，无需安装，解压后即可使用。

◆ 计算机需要安装的工具软件在资料包中的位置：

- 电脑端工具：“A 盘\6 - 搭建开发环境所用软件” 目录下。
- SDK 和 SDK 离线文档：“B 盘\6: SDK 和快速试验说明” 目录下。

表 3-2：手机端需要安装的 APP

序号	软件工具	描述
1	nRF Connect。	手机端测试 APP
2	nRF Toolbox。	手机端测试 APP

◆ 手机端需要安装的 APP 在资料包中的位置：

- 安卓系统：安装文件位于“A 盘\2 - 安卓 APP 安装文件 APK” 目录下。
- IOS 系统：苹果商店搜索 APP 下载安装。

2. 搭建开发环境

2.1. 安装串口驱动

开发板上设计了 USB 转串口电路，使用的 USB 转串口芯片是 CH340，使用前需要安装驱动。

注：如果计算机上已经安装了 CH340 的驱动，则无需再安装，可直接跳过此步骤。

CH340 和 CH341 的驱动一样，开发板配套资料包里面已经下载好了驱动，驱动的位置在开发板资料包的“资料盘(A 盘)\ 8 - 搭建开发环境所用软件\USB 驱动”目录下，驱动安装程序名称为“ch341ser.exe”。

1. 双击驱动安装程序“ch341ser.exe”，弹出驱动安装界面。
2. 点击【安装】安装驱动。

点击【安装】后，稍等片刻，驱动安装程序会弹出安装是否成功的提示，如提示安装成功，则表示驱动已经成功安装，关闭安装程序即可。



图 3-1：USB 转串口驱动安装界面

2.2. 安装 MDK5.27

2.2.1. Keil μ Vision 简介

Keil μ Vision 是 Keil 公司开发的一个集成开发环境，目前共有 μ Vision2、μ Vision3、μ Vision4 以及 μ Vision5 几个版本。2005 年 Keil 公司被 ARM 公司收购，2011 年 3 月 ARM 公司发布的最新集成开发环境 RealView MDK 开发工具中集成了最新版本的 Keil μ Vision4，其编译器、调试工具实现与 ARM 器件的最完美匹配。

本文档中使用的 MDK 版本是：5.27，打开后的主界面如图所示。

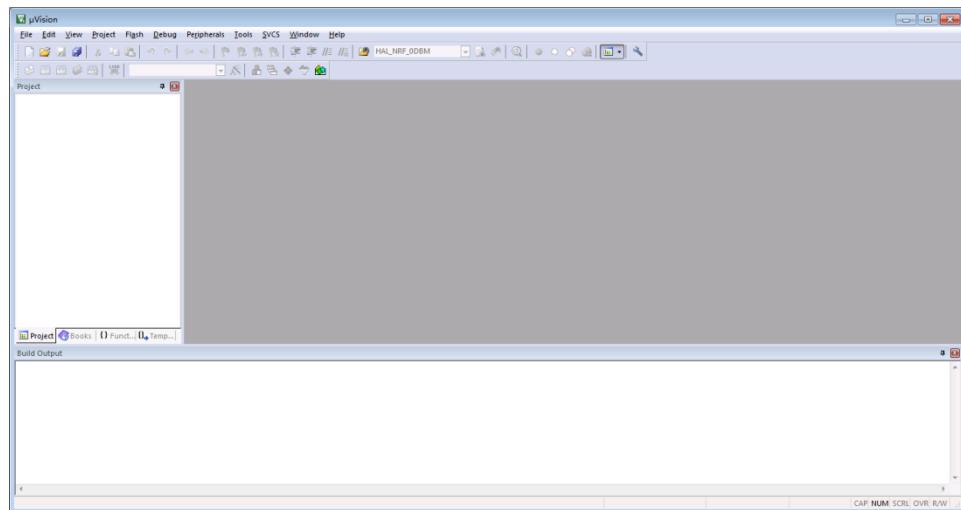


图 3-2：MDK5.27 主界面

2.2.2. MDK5.27 安装步骤

- 1) 双击 MDK5.27.exe，弹出 MDK-ARM V5.27 的安装向导，单击【Next】。



图 3-3: MDK5.23 安装

- 2) 勾选【I agree to ...】，然后点击【Next】，如下图所示。

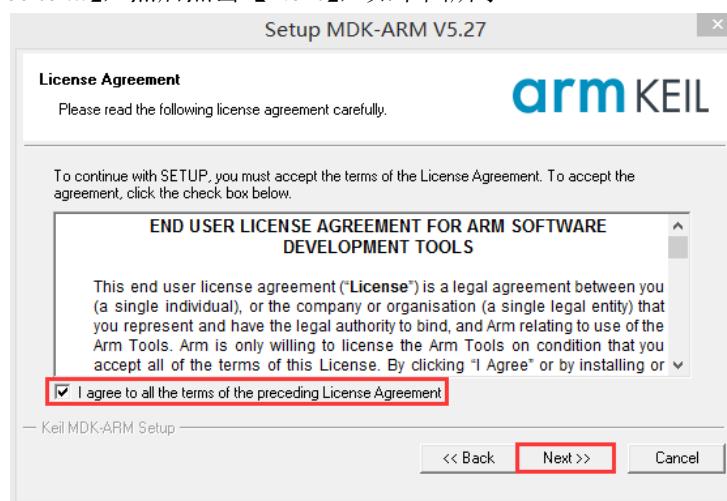


图 3-4: MDK5.277 安装

- 3) 选择安装路径，如下图所示。

此处，可以根据自己的需要选择安装路径，本文档设置的安装路径是默认安装路径，即安装在 C 盘。

◆ 注意：强烈建议安装在默认路径，这会省去一些不必要的麻烦。

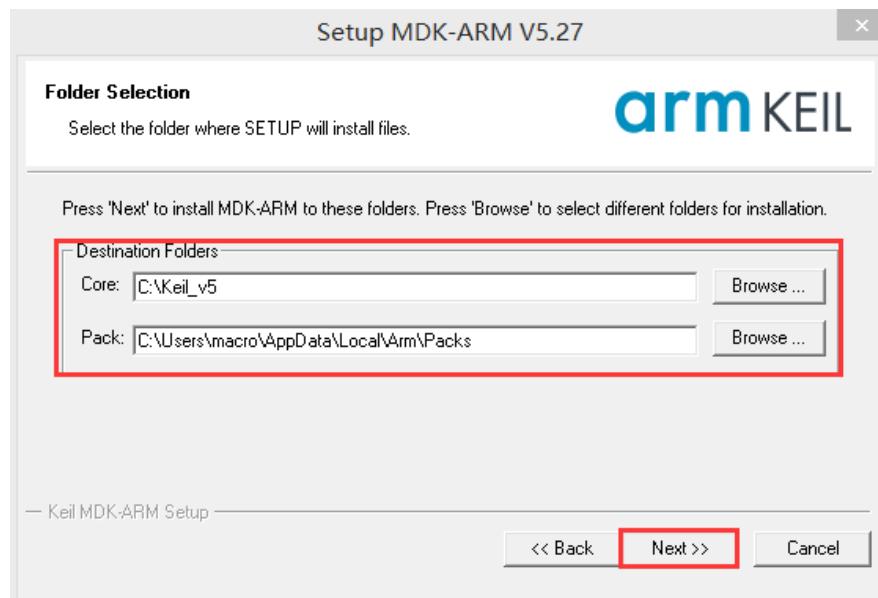


图 3-5: MDK5.27 安装

如果已经安装了 MDK5.27 之前的版本，会提示：是否备份旧的文件，这里可以根据需要选择是否备份。

4) 根据提示填入相关信息，然后点击【Next】，如下图所示。

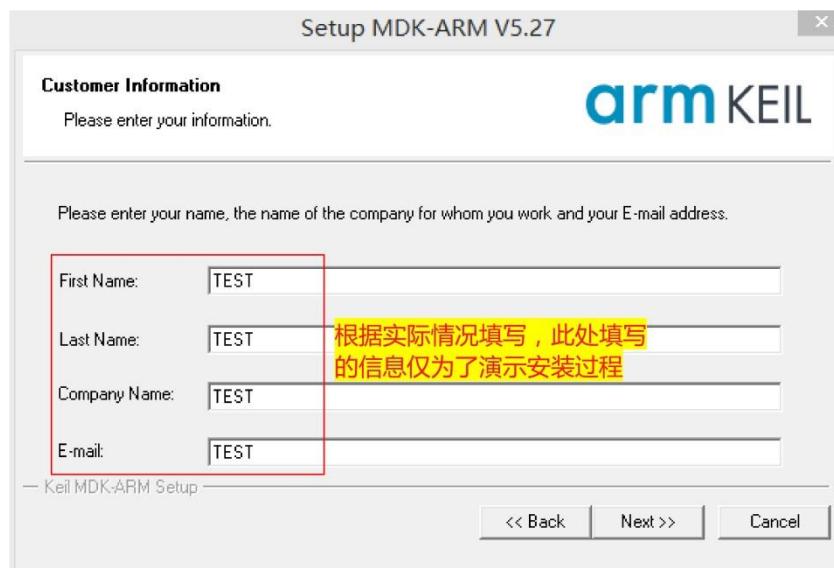


图 3-6: MDK5.27 安装

5) MDK5.23 开始安装，等待 MDK5.27 安装完成，如下图所示。

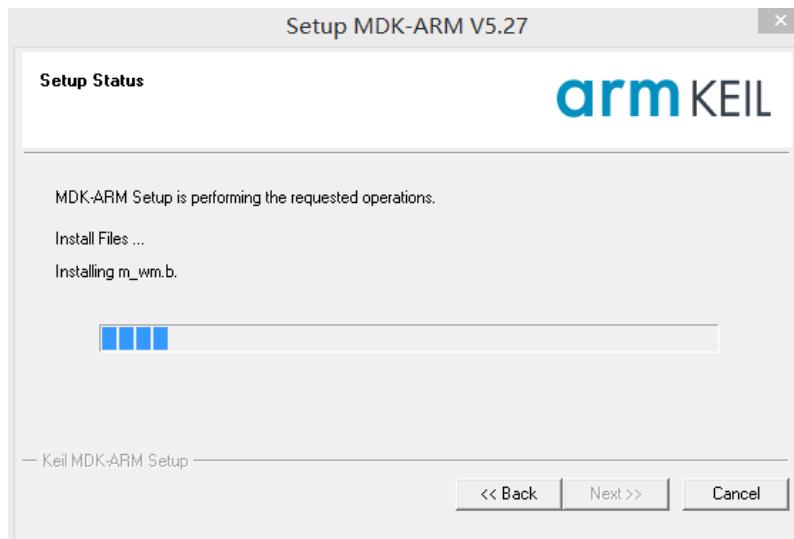


图 3-7: MDK5.27 安装

6) 选择安装, 如下图所示:



图 3-8: MDK5.27 安装

7) 点击【Finish】完成安装, 如下图所示。



图 3-9: MDK5.27 安装

8) 点击【Finish】出现如下界面，点击“OK”，如下图所示。

这里是更新 PACK，因为我们已经下载了 nRF52840 的 PACK 离线安装包，所以，这里不用在线安装，直接关闭即可。

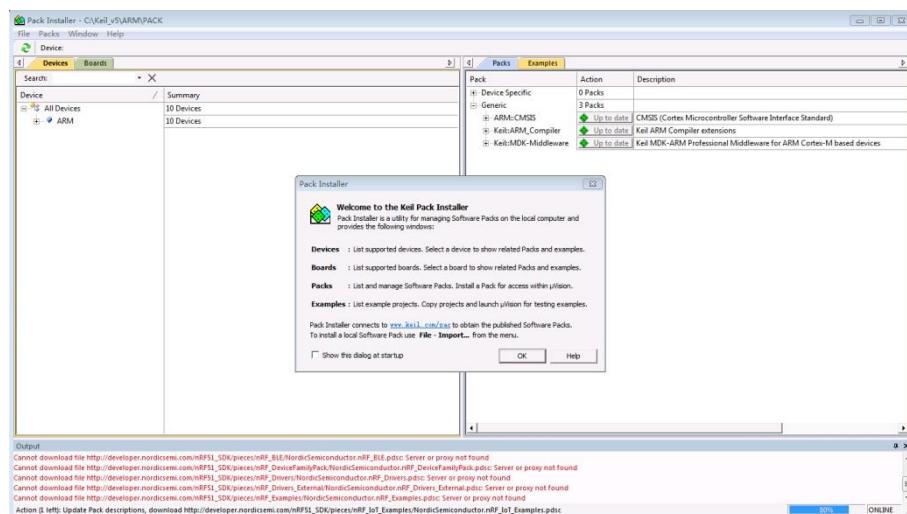


图 3-10: MDK5.23 安装

◆ 注意：MDK 安装完成后，还需要注册，未注册的是限制版本，有 32K 代码的限制。

2.3. 安装 pack

MDK5 相比于之前的版本，在架构上做了很大调整，增添了许多新的特性。MDK5 分成了 MDK 内核和 Software Pack 两部分，其内核部分仍然是包括编辑器、编译器、包安装和调试跟踪，而 Software Pack 则又包含 Device、CMSIS 和 MDK professional Midware。器件(Software Packs)与编译器(MDK core)分离的结构使得对 MDK5 的更新和维护更简单，我们可以根据自己的开发需要来选择安装对应的器件软件包，如我们开发 nRF52840，只需要安装 nRF52840 的软件包即可。

◆ 注：下面描述的是“NordicSemiconductor.nRF_DeviceFamilyPack.8.27.1.pack”的安装方法，其它版本的 Pack 包的安装方法和 Pack8.27.1 的完全一样。

◆ SDK16.0 需要安装的 pack: NordicSemiconductor.nRF_DeviceFamilyPack.8.27.1.pack。

1. 双击“NordicSemiconductor.nRF_DeviceFamilyPack.8.27.1.pack”，弹出安装向导，单击【Next】。

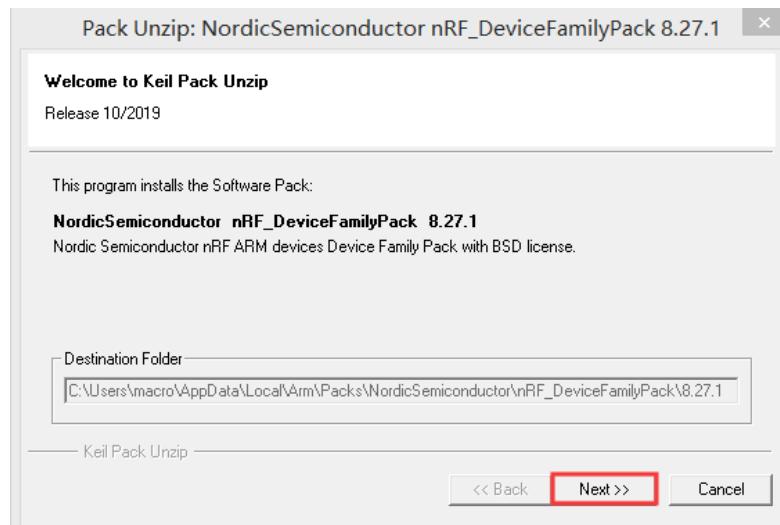


图 3-11: pack8.27.1 安装

2. 勾选【I agree to ...】，然后点击【Next】，如下图所示。

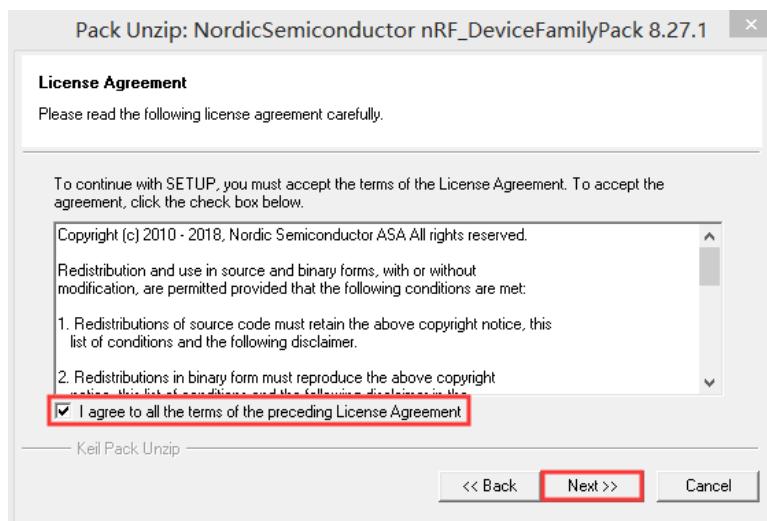


图 3-12: pack8.27.1 安装

3. 等待一段时间，出现如下界面后点击【Finish】结束安装。

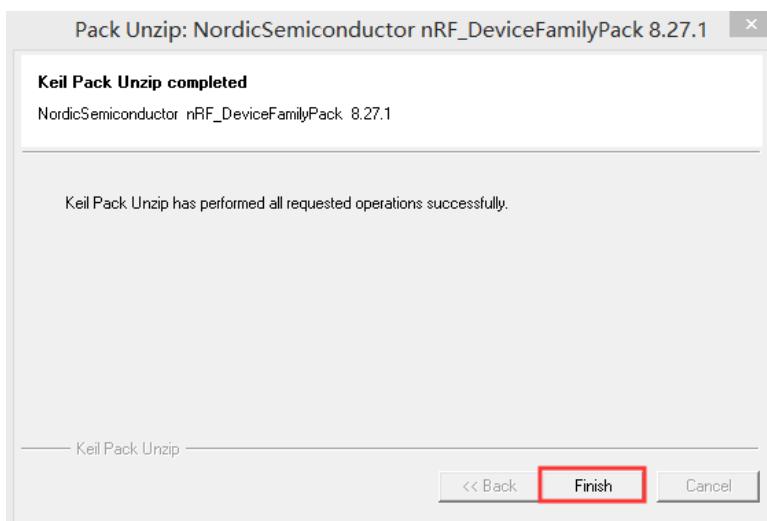


图 3-13: pack8.27.1 安装

2.4. 安装 CMSIS.4.5.0

1. 双击“ARM.CMSIS.4.5.0.pack”安装文件，打开 CMSIS 安装窗口，点击【NEXT】开始安装。

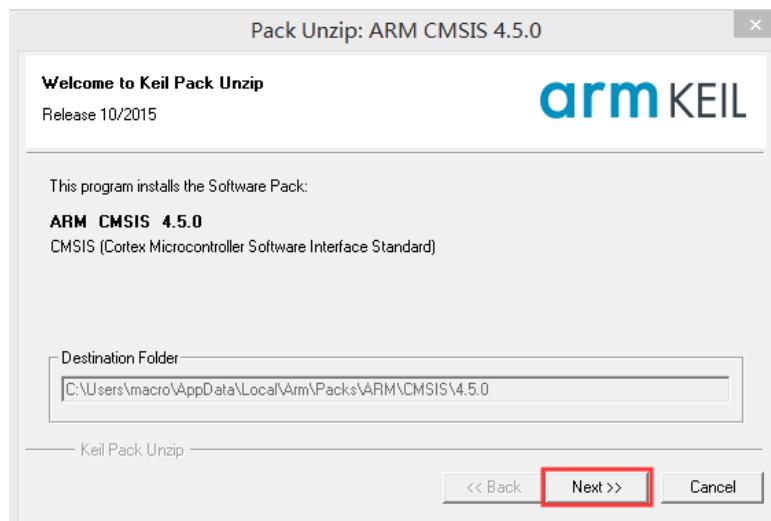


图 3-14: CMSIS 安装

2. 等待一段时间，出现如下界面后点击【Finish】结束安装。

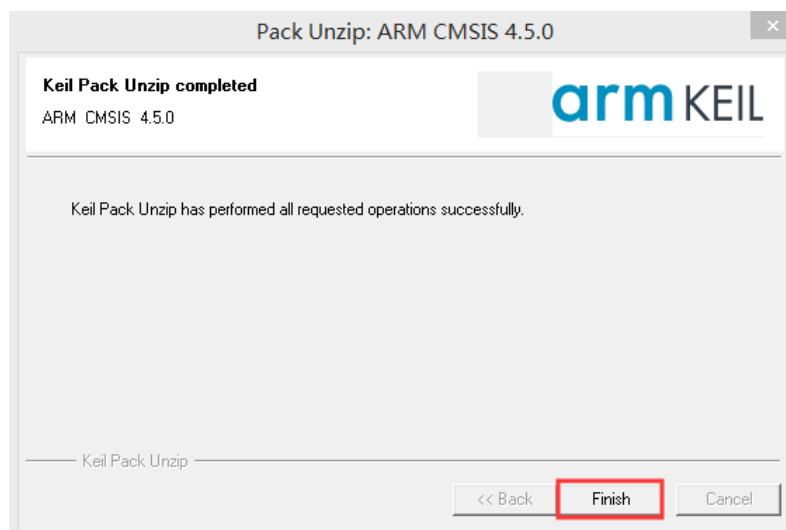


图 3-15: 完成 CMSIS 安装

2.5. 安装 Command-Line-Tools

1. 双击“nRF-Command-Line-Tools_10_4_1_Installer_64.exe”安装文件，打开安装窗口，点击【Install】开始安装。

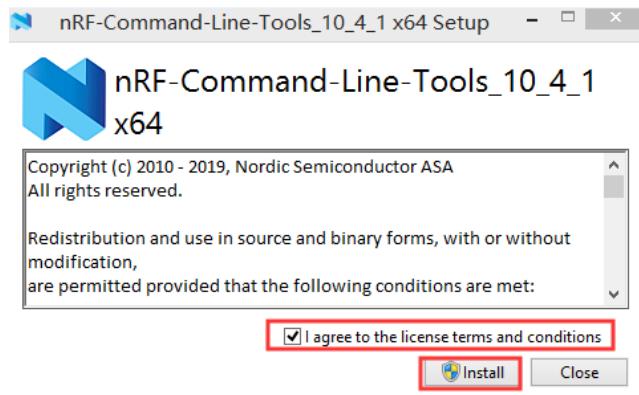


图 3-16: 安装 2. Command-Line-Tools

2. Command-Line-Tools 集成了 JLINK 驱动，安装时会弹出 JLINK 驱动安装窗口，点击【NEXT】，安装 JLINK 驱动。



图 3-17: JILNK 驱动安装

3. 点击【IAgree】，继续安装。

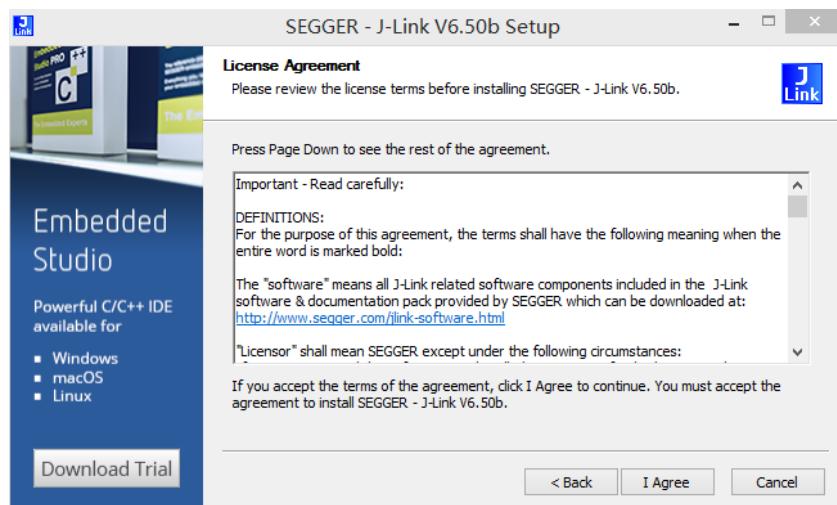


图 3-18: JILNK 驱动安装

4. 设置 JLINK 驱动的安装选项和安装路径。

这里可以根据自己的需要选择 JLINK 安装选项和路径，本文档设置的安装路径是默认安装路径，即安装在 C 盘。

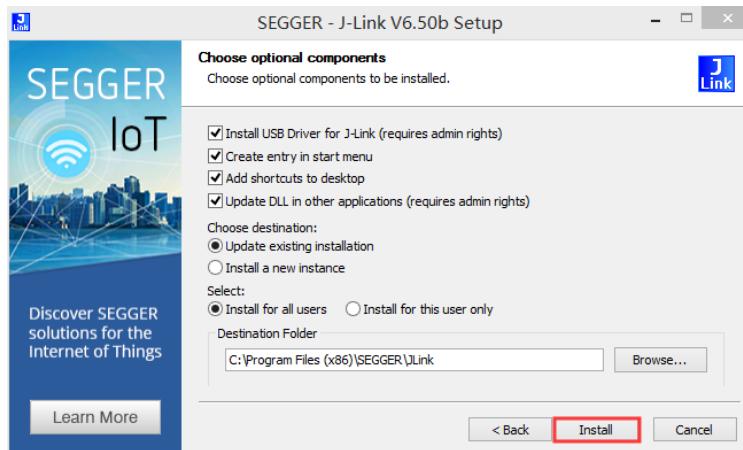


图 3-19: JILNK 驱动安装

5. 等待安装完成，弹出如下窗口，勾选需要更新 JLINK 驱动的集成开发环境，点击【OK】，如下图所示。

◆ 注意：这里一定要勾选，否则 JLINK 驱动安装后，开发环境如 keil 里面的驱动不会被更新。

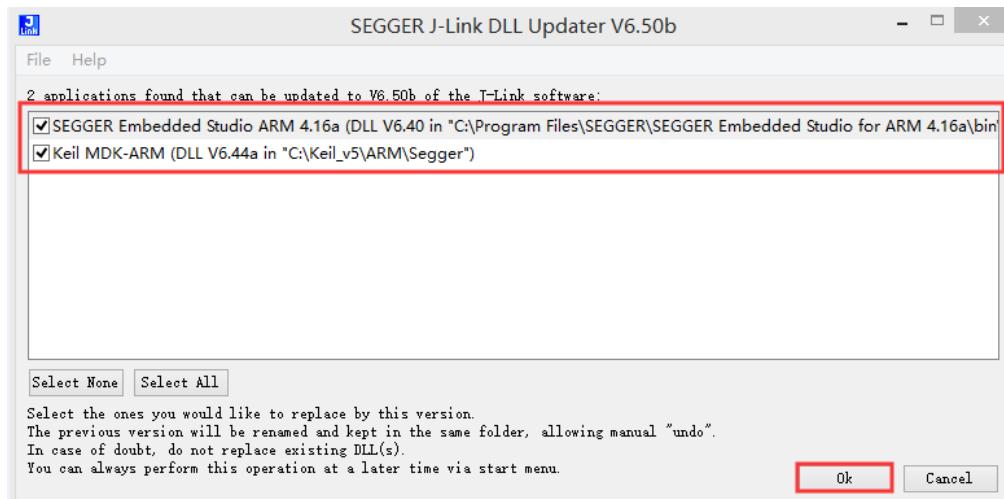


图 3-20: JILNK 驱动安装

6. 点击【Finish】，完成 J-LINK 驱动的安装。



图 3-21: 完成 J-ILNK 驱动安装

7. Command-Line-Tools 需要电脑上安装 Visual C++ 2017，如果电脑上没有安装，会弹出 Visual C++ 2017 安装窗口，去下图所示，勾选【我同意...】后点击安装。



图 3-22：安装 Visual C++ 2017

8. 等待一段时间，出现如下界面后点击【关闭】完成 Visual C++ 2017 的安装。

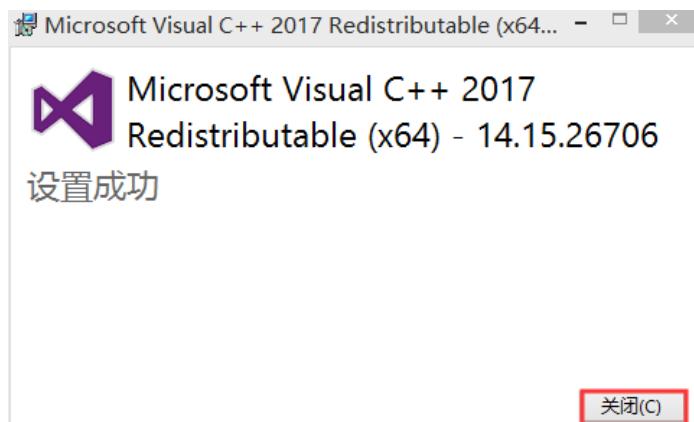


图 3-23：完成 Visual C++ 2017 安装

9. 最后，会弹出 Command-Line-Tools 安装窗口，点击【Install】安装。



图 3-24：安装 Command-Line-Tools

10. 等待一段时间，出现如下界面后点击【Finish】完成 Visual C++ 2017 的安装。

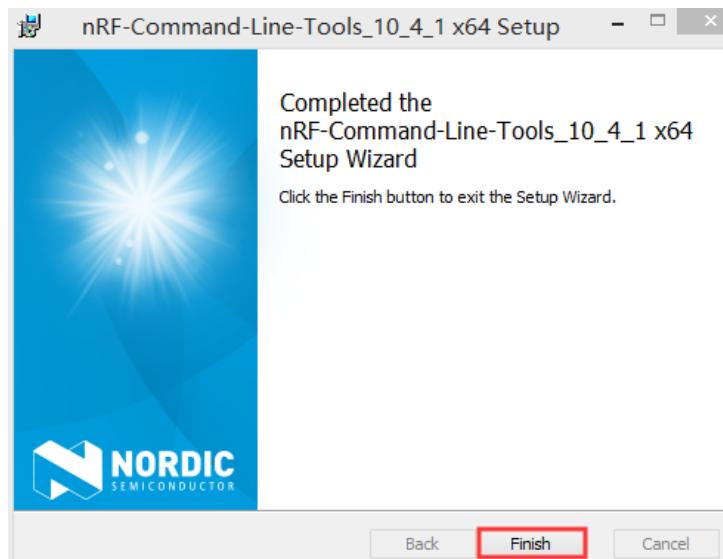


图 3-25：完成 Command-Line-Tools 安装

11. 点击【Close】完成软件的安装。

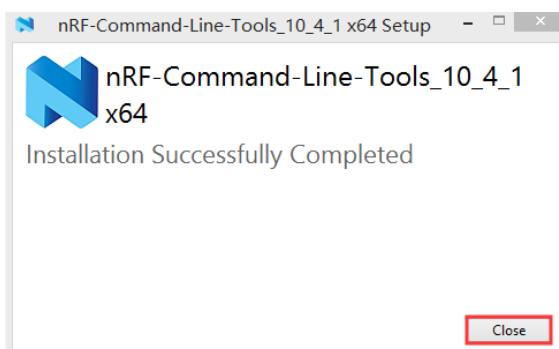


图 3-26：完成安装

2.6. 手机端 APP 安装

手机端用于测试的 APP 有两个：nRF Connect 和 nRF Toolbox，资料包里面已经下载好了安卓系统 APP 的安装文件，传到手机安装即可。对于苹果手机，需要自行去苹果商店下载。

nRF Connect 和 nRF Toolbox 的主要区别如下，调试的时候根据自己的需求来选择使用哪一个 APP，建议使用 nRF Connect。

❖ **注意：**手机端 APP 在蓝牙 BLE 的实验里面才会用到，这里我们先安装上，在开发指南下册中会用到。

■ nRF Connect 和 nRF Toolbox 的区别

- nRF Connect：手机端调试 BLE 的工具，它没有针对各个实验的图形界面，用户观察到的是原始的数据，nRF Connect 可以显示 RSSI 曲线图，可以同时连接多个从机，还可以“学习”从机后自己作为从机工作，功能非常强大，同时它的通用性和兼容性也很强，建议大家调试的时候手机端 APP 使用 nRF Connect。
- nRF Toolbox：手机端调试 BLE 的工具，nRF Toolbox 是针对部分实验的专用调试工具，

它的优点是针对支持的实验有专门的图形界面，如心率、健康体温计等，可以直观地看到具体的心率曲线、温度值。缺点是只支持部分实验，另外兼容性比 nRF Connect 差，有时会遇到无法发现服务的情况。

第四章：新建和配置裸机工程

1. SDK 库简介

Nordic 的 SDK，从 SDK15.0 开始，在文件结构上做了一些改变，相对于之前的版本增加了“integration”和“modules”两个文件夹，“nRF5_SDK_16.0.0_98a08e2.zip”解压后，文件目录如下：

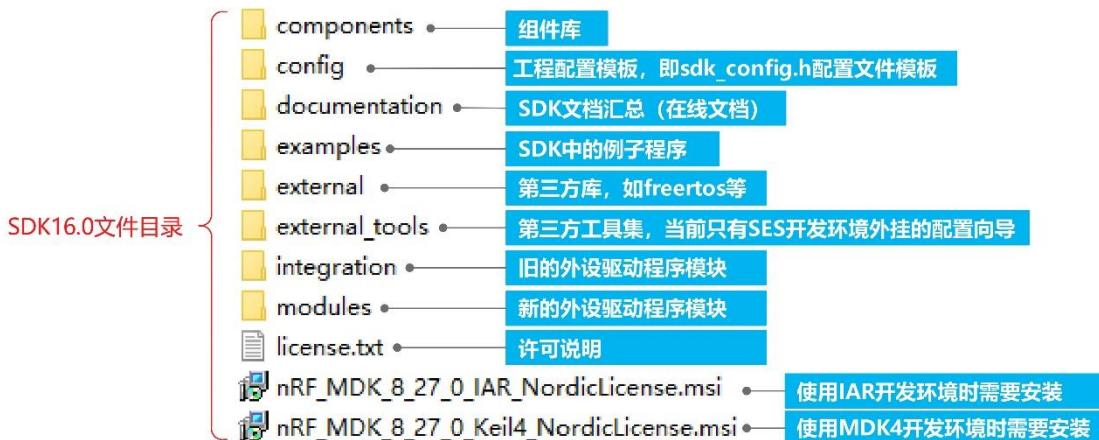


图 4-1: SDK16.0 文件目录

❖ 注意：SDK 中 MDK5 的工程不需要安装文件，直接用 MDK5 打开工程文件即可。本教程所讲解的例子使用的均为 MDK5 开发环境。

SDK16.0 中主要文件说明如下：

1. Components

Components 包含了开发时需要用到的各种库，无论我们开发的是 BLE 的应用还是 2.4G 的应用，Components 都是必不可少的，Components 的目录如下图所示。



图 4-2: Components 文件目录

2. Examples

Examples 文件夹中存放的是 SDK 中所有的例子源码，包含 BLE 从机、BLE 主机、BLE 主从一体、芯片外设等例子程序。

Examples 中的每个例子都包含了多个开发环境下的工程和多个芯片的工程，使用时要注意区别，下面以“蓝牙心率计”的 demo 为例说明如何打开 nRF52840 在 MDK5 开发环境下的工程。

“蓝牙心率计”的 demo 在 SDK 中的路径是“nRF5_SDK_16.0.0_98a08e2\examples\ble_peripheral\ble_app_hrs”，打开该路径后我们会看到几个名称为“pca10xxx”的文件夹，它们的含义如下表所示。如果该目录下有对应开发板型号的文件夹，表示该文件夹里面的工程可直接在开发板上运行，如该目录下有“pca10056”文件夹，表示该文件夹里面的工程可直接在 IK-52840DK 开发板上运行。

表 4-1：芯片对应的文件夹名称

名称	路径
pca10040	nRF52832 的工程 (pca10040 是 Nordic 的 nRF52832 开发板的型号)。
pca10040e	nRF52810 的工程。
pca10056	nRF52840 的工程 (pca10056 是 Nordic 的 nRF52840 开发板的型号)。
pca10056e	nRF52811 的工程。
pca10059	nRF52840 USB Dongle 的工程
pca10100	nRF52833 的工程 (pca10100 是 Nordic 的 nRF52833 开发板的型号)

打开 pca10056 目录后，继续往下操作：

- 因为我们使用的是 nRF52840，所以需要打开 pca10056 目录下的工程。
- nRF52840 对应的 BLE 的协议栈（SoftDevice）是 S140，所以需要打开 S140 目录下的工程。
- 我们要打开的是 MDK5 开发环境的工程，所以，需要打开 arm5_no_packs 目录下的工程。

综上所述，对于“蓝牙心率计”的 demo，MDK5 开发环境对应的工程文件是“nRF5_SDK_16.0.0_98a08e2\examples\ble_peripheral\ble_app_hrs\pca10056\s140\arm5_no_packs”目录下的“ble_app_hrs_pca10056_s140.uvprojx”，使用 MDK5 打开该工程文件即可。

3. Integration 和 moudles

Integration 中存放的是“旧版本的”芯片外设驱动程序，moudles 中存放的是芯片新版本的外设驱动程序。

在 SDK15.0 之前，芯片外设驱动都是存放在 Components 文件夹的 drivers_nrf 目录下，从 SDK15.0 开始，Nordic 开始重新编写外设驱动程序，并命名为“nrfx”，新的外设驱动存放在 moudles 目录下。“旧版本的”芯片外设驱动和“新版本的”芯片外设驱动可以同时整合使用，“旧版本的”芯片外设驱动存放在了 Integration 目录下，名称也改为“nrfx”。

2. 新建和配置工程

nRF52840 可以使用 MDK4、MDK5 和 IAR 集成开发环境来开发程序，在这些开发环境中，MDK5 是最常用的，本章主要描述如何在 MDK5 集成开发环境中新建工程和配置工程。

2.1. 新建工程模板

新建工程之前，我们先了解一下 MDK5 和 pack 的关系。MDK5 与之前的版本相比，最大的区别在于器件(Software Packs)与编译器(MDK core)分离。也就是说，我们安装好编译器(mdk_5xx.exe)以后，编译器里面是没有任何器件的。所以，我们需要根据自己具体开发来选择安装 pack，这样做的好处就是我们可以很灵活的来管理（下载、更新、移除）设备支持包和中间件更新包。

2.1.1. 规划工程目录

建立工程之前，我们需要先考虑一下工程文件的组织，也就是工程的目录。清晰的工程目录既方便我们管理工程中的各个文件，也方便日后的维护和移植，我们可以根据自己的习惯和喜好来建立自己的工程目录，但是也不要太随意，文件目录应该一目了然，目录中各个文件夹的名字要能准确地指示里面的内容。

下面是我们建立工程时使用的工程目录，供大家参考，其中：

- App 文件夹：用于存放 main.c 文件和我们自己编写的应用程序文件。
- Project 文件夹：
 - Config 文件夹：用于存放工程配置向导 (sdk_config.h)。
 - mdk5 文件夹：用于存放工程文件。
- doc 文件夹：用于存放说明之类的文档。
- components、integration、modules 等：从 SDK 中拷贝的库文件。

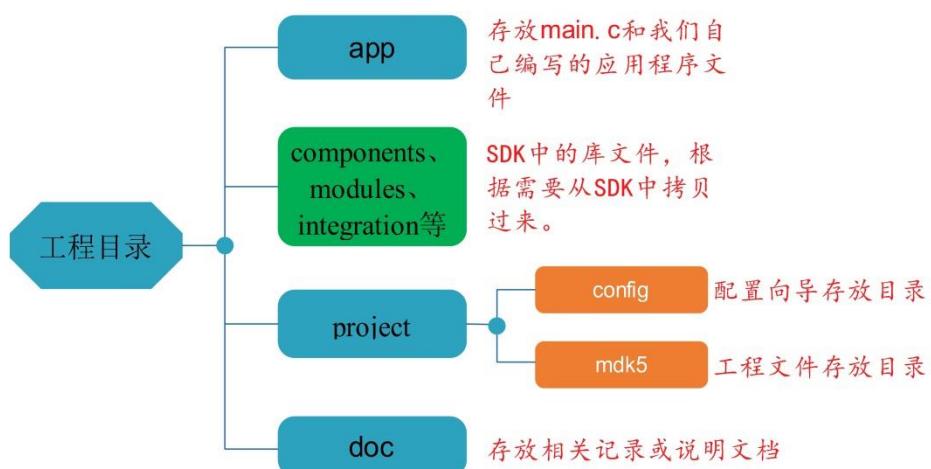


图 4-3：工程存放目录

2.1.2. 建立目录、拷贝库文件

按照上文中描述的工程目录新建用于存放工程各个模块的文件夹。先在 D 盘新建一个

名字为 ble_app_template 的文件夹，然后在这个文件夹下面新建 3 个名字分别为 project、doc 和 app 的文件夹，之后在 project 文件夹里面再新建 config 和 mdk5 两个文件夹，其中 config 文件夹用于存放工程配置向导 (sdk_config.h)，mdk5 文件夹用于存放工程文件。

之后，解压 SDK16.0，并将需要的库文件（包含 components、integration、modules 和 external 文件）拷贝到 BLE 工程模板目录下，如下图所示。

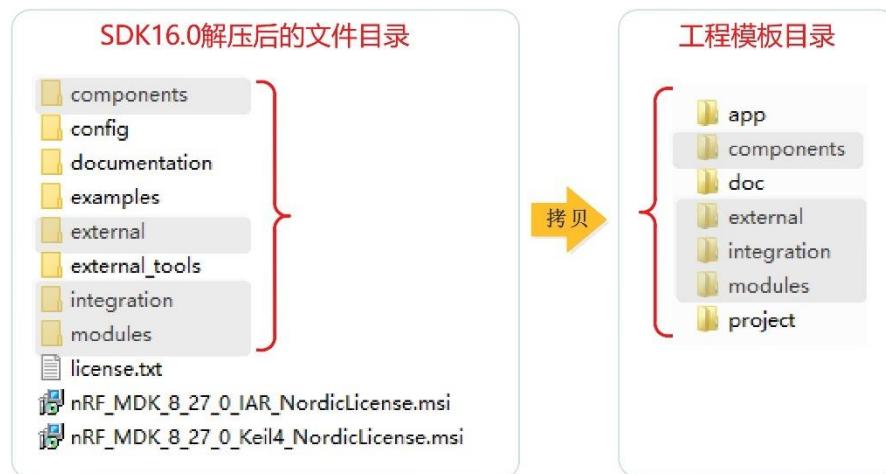


图 4-4：拷贝库文件

库文件可以根据需要去拷贝，也可以全部拷贝过来，全部拷贝占用的空间会大一些，但是用起来方便，以后添加功能时不需要再次去拷贝，故建议直接拷贝 SDK 中的 components、integration、moudles 和 external 文件夹到工程目录下。

2.1.3. 新建工程

建立工程存放文件夹和拷贝库文件之后，我们就可以开始建立工程了。

下面通过新建一个简单的裸机工程（不含 SoftDevice 的工程）：使用 GPIO 驱动指示灯闪烁来说明工程建立的基本过程，工程名取为：led_blinky，工程存放到 D 盘。

1. 启动 MDK，点击【Project】，在弹出的下拉菜单中选择【New uVision Project】。

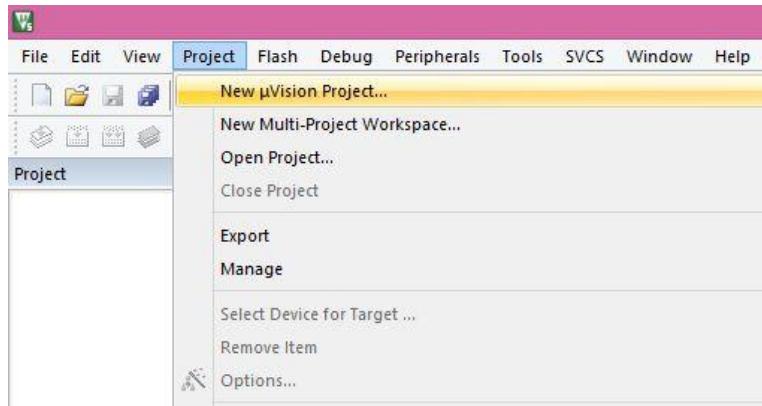


图 4-5：新建工程窗口

2. 设置工程名和工程保存路径，设置完成后点击【保存】。

- ❖ 工程路径和工程名设置注意事项：工程路径和工程名不能包含汉字字符(虽然有些计算机使用汉字字符没有问题，但是还是建议不要使用汉字字符，因为 MDK 对汉字字符的支持比较差)，同时路径不要过深，否则打开工程或仿真时可能会出现问题。



图 4-6：设置工程路径和工程名

保存后，工程名称是：led_blinky，工程保存路径是：“…\project\mdk5\led_blinky”。

3. 保存工程后，会弹出器件选择窗口，选择好器件后点击【确定】。

开发板上使用的 nRF52840 型号是：nRF52840-QIAA，所以，在下图的器件列表中需要选择这个型号。选中器件后，右边的文本框中会显示该器件的描述信息。

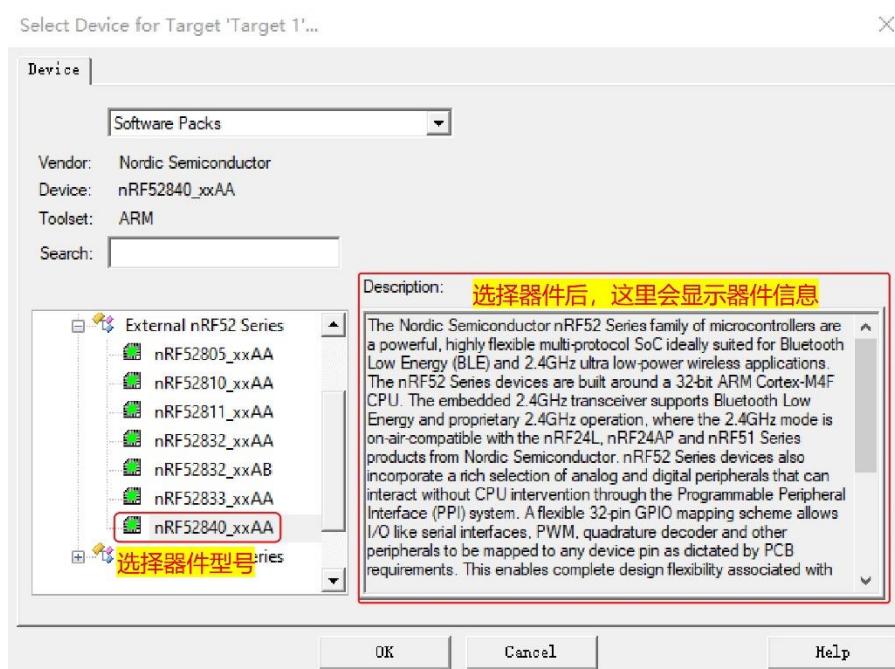


图 4-7：选择器件

4. 配置 RTE(Run-Time Environment)，选择完成后点击【OK】。

勾选两个必选项: CMSIS 中的 CORE 和 DEVICE 中的 StartUp。注意他们的版本号, SDK 版本不一样, 对应的 CMSIS 和 StartUp 版本号可能会不一样, 本文使用的 SDK 版本是 SDK16.0, SDK16.0 对应的 CORE 版本是 4.5.0, StartUp 版本是 8.27.1。

下图中, 我们可以看到 CORE 的版本只能选择 5.2.0, StartUp 版本只能选择 8.27.1, 这是因为安装 MDK5.27 时会自动安装 5.2.0 的 CORE, 而 MDK 新建工程时只会显示已安装的最新的 CORE 和 StartUp 版本, 所以, 这里 CORE 的版本选择 5.2.0, StartUp 版本选择 8.27.1。工程建立好了之后, 我们可以在工程配置中修改, 以保持和 SDK16.0 一致, 后面在工程配置章节会说明修改方法。

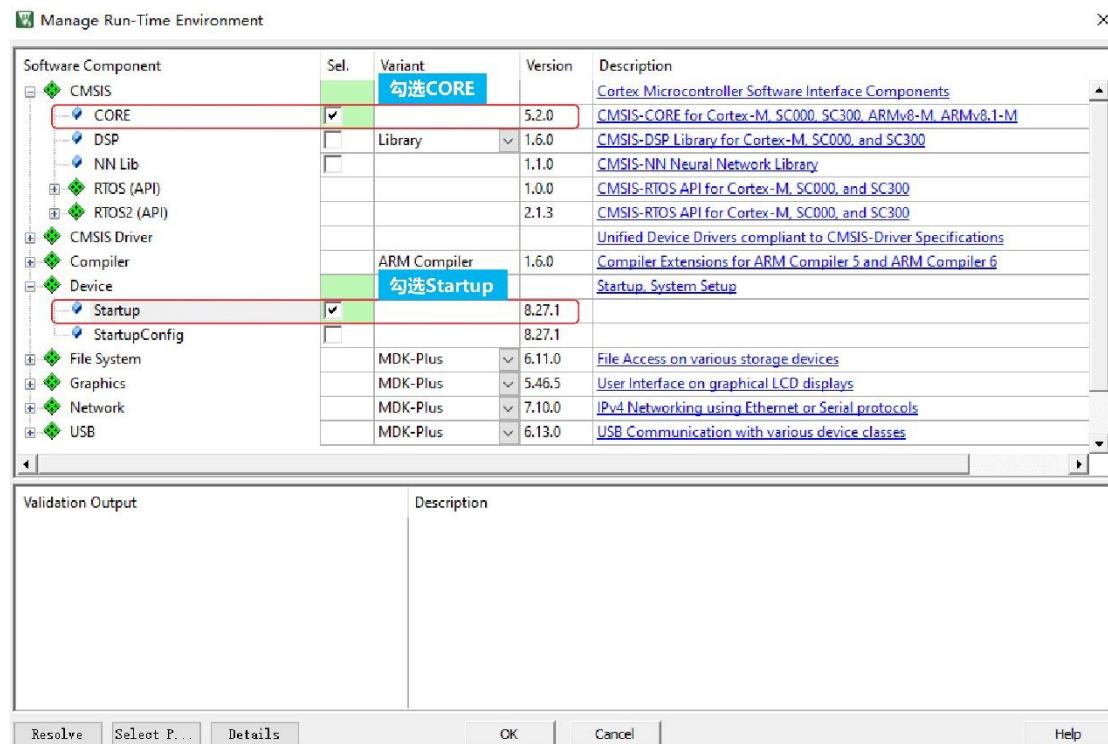


图 4-8: 配置 RTE

◆ 注意事项: 再次说明一下, 新建工程配置 RTE 时, 对于 CORE 和 StartUp, MDK 会自动选择最新的版本, 在这个步骤我们直接选择最新版本, 后面配置工程时再根据需要修改为自己需求的版本即可。

5. 管理 MDK 工程目录。

主要是添加组(也就是在 MDK 中添加文件夹)、修改组名称和软件包 (pack), 目的是为了目录清晰, 方便添加文件和管理文件。这里我们使用的工程目录是参考 SDK 里面的 BLE 工程的, 这么做的好处是和 SDK 保持一致, 以方便我们阅读 SDK 里面的参考代码。

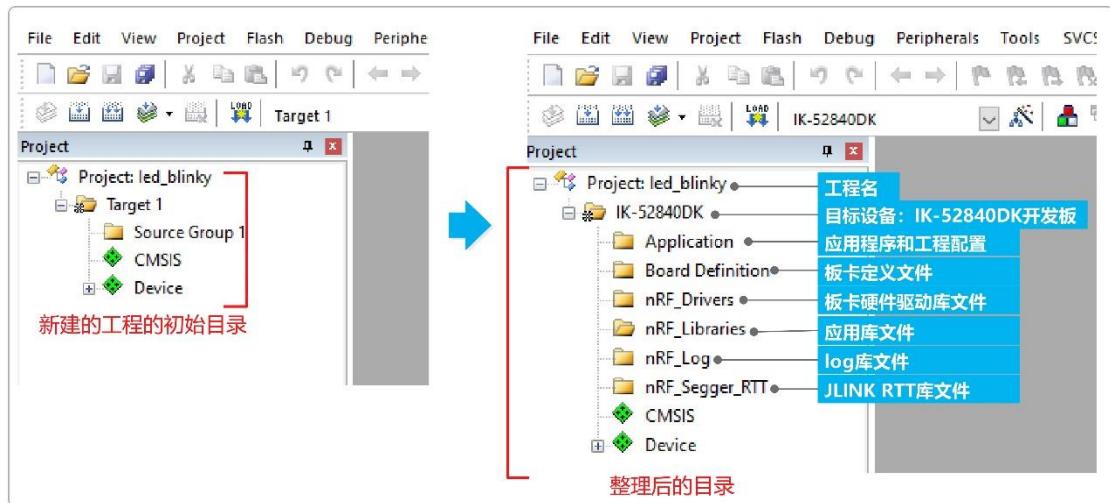


图 4-9：整理目录

1) 整理目录时会用到下面几个操作：

- 修改组名称：先选中需要修改名称的组，然后单击即可修改名称。**注意，不是双击。**
- 添加组：选中 Target(即整理后的目录中的 IK-52840DK)，右键选中“Add Group”，即可添加一个组。
- 调整组的顺序。

点击工程项目管理图标



图 4-10：打开工程项目管理窗口

按照下图所示根据自己的习惯调整组的顺序。

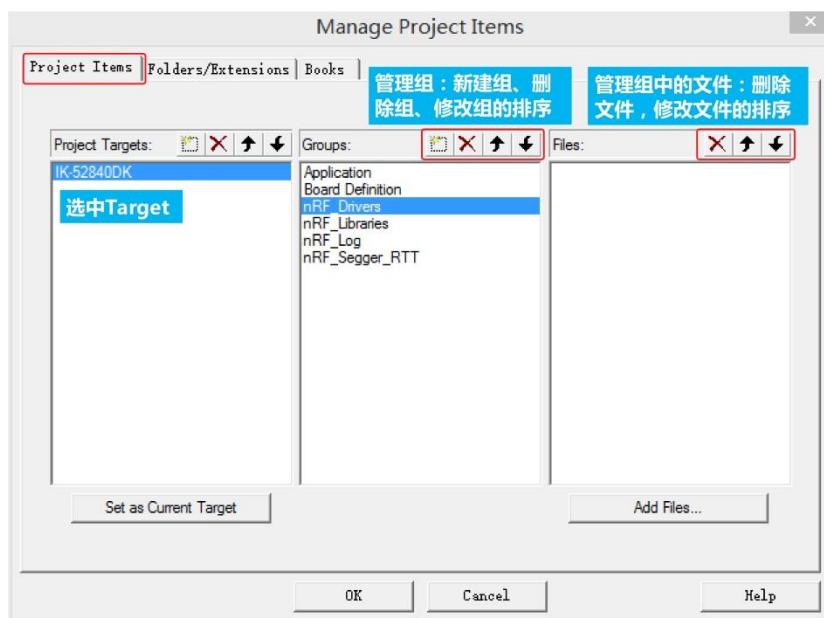


图 4-11：调整组的顺序

2) 管理软件包：这里不需要过多操作，我们可以在这修改 StartUp 的版本和选择是否自动使用安装的最新的软件包。

- 设置是否自动使用安装的最新的软件包

勾选“Use latest versions of all installed Software Packs”即可打开自动使用安装的最新的软件包，不勾选即可以使用指定的版本。

- 修改 StartUp 和 CORE 的版本

如我们的计算机中安装了 Nordic nRF5x 几个版本的器件包，就会显示多个版本的器件包，因为 SDK16.0 使用的 CMSIS 版本是 4.5，器件包版本是 8.27.1，所以我们要选择 CMSIS 的版本是 4.5，器件包的版本是 8.27.1，同时去掉自动选择最新版本的勾选，如下图所示。

点击下图所示的图标，打开软件包管理窗口。



图 4-12: 打开软件包管理窗口

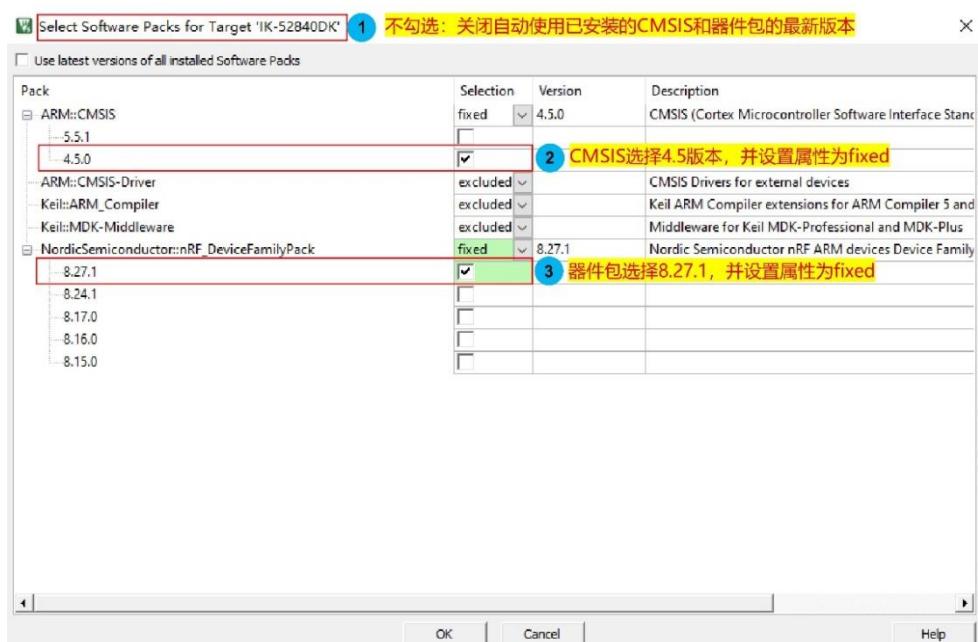


图 4-13: 修改 StartUp 和 CORE 的版本

这里设置下面 3 项，这么设置是为了让我们建立的工程使用确定的软件版本，这样就可以避免因为自动引用最新版本引起问题的风险。

- 关闭自动使用最新的软件包功能。
- CMSIS 版本设置为 4.5.0，并设置版本 fixed。
- pack 版本设置为 8.27.1，并设置版本 fixed。

2.1.4. 添加需要的库文件

根据程序实际使用的情况来添加库文件，裸机工程一般需要包含下面 5 个项目：

- 1) 驱动文件模块：包含底层硬件驱动和经过封装的上层驱动，如 GPIO、UART、SPI、APP 定时器等等，它的作用是给应用程序调用，实现各种功能。需要注意，部分比较简单的外设只有“.h”的头文件，而没有“.c”文件，如 GPIO，只有“nrf_gpio.h”头文件。

对于本例来说，因为我们只需要驱动 LED 指示灯，所以需要使用 GPIO 相关的库文件，即“nrf_gpio.h”，同时需要通过延时来实现 LED 闪烁的效果，所以还需要加入软件延时库文件，即“nrf_delay.h”。

- 2) 板卡定义模块：板卡定义用来定义具体板卡上的常用的外设，在这里主要是用来定义 LED 指示灯和按键，同时，它也包含了 LED 指示灯和按键常用的操作函数。
- 3) 应用库：包含各种应用的库文件，如 APP 定时器、CRC 校验、软件 FIFO、简易文件系统以及错误处理等等，“nRF_Libraries”组中加入的文件如下表所示。这里面尤其要注意错误处理模块：一个成熟的库，必定会包含错误处理机制，用于开发过程中跟踪错误，排查问题。错误处理机制对复杂的工程尤其有用，对于一个复杂的工程来说，如果运行时产生了错误，分析起来费时费力，这时可以通过错误代码确定问题类型，方便进一步分析。
- 4) 日志打印模块：用于打印调试信息，可以通过 UART 输出或者通过 RTT 打印信息。对于简单的工程，如本例的驱动 LED 闪烁，可以不使用日志打印模块，但是对于稍复杂一些的工程，日志打印模块都是“标配”，是不可缺少的一部分。
- 5) 工程配置向导：工程配置向导名称是“sdk_config.h”，它用于配置各个软件模块和外设，当我们在应用程序中使用某个软件模块或外设的时候，需要在配置向导中进行配置，即开启使用的外设或软件模块，同时配置向导也可以用来配置外设或软件模块的一些参数。接下来，我们将需要的库文件按添加到工程中。

1. 向工程中添加文件的方法

向工程中添加文件可以通过下面三种方式来添加。

- **方法 1：**双击组名打开添加文件窗口，导航到需要添加的文件的存放路径，添加即可。
- **方法 2：**选中组，右键选择“Add Existing Files To...”打开添加文件窗口，导航到需要添加的文件的存放路径，添加文件。
- **方法 3：**通过工程项目管理添加。点击工具栏中的图标，打开工程项目管理窗口，添加文件。如下图所示：

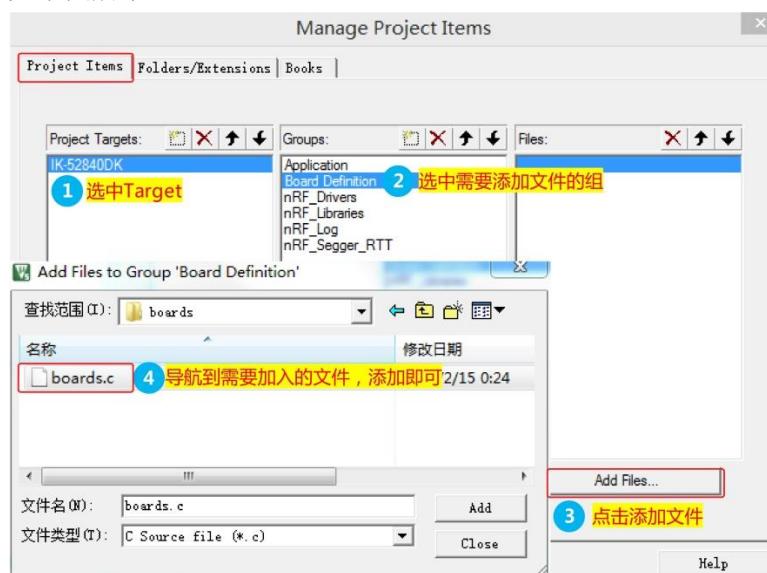


图 4-14：通过工程项目管理添加文件

2. 添加驱动文件

本例中只用到了 GPIO 和软件延时的驱动，而这两个驱动都是以头文件的方式提供的，所以 GPIO 没有需要加入的 “.c” 文件。

3. 添加板卡定义

板卡定义需要加入的文件如下表所示，注意，板卡定义对应的是具体的电路板，在这里对应的是 IK-52840DK 开发板。具体操作为将 “boards.c” 添加到工程的 “Board Definition” 组即可。

表 4-2: 板卡定义需要加入到工程的文件

文件名	路径
boards.c	..\..\components\boards

4. nRF_Drivers 组

“nRF_Drivers”组加入的是各种外设的驱动库文件和 Soc 相关的文件，如时钟、GPIOTE、UART 等等，本例中只用到了 GPIO 外设，而 GPIO 外设的驱动是以头文件的形式提供的，所以无需添加，这里只需添加 “nrfx_atomic.c” 文件（该程序模块实现了 C11 标准 stdatomic.h 的简化 API）即可。

表 4-3: “nRF_Drivers”组需要加入到工程的文件

文件名	路径
nrfx_atomic.c	.. \modules\nrfx\soc

5. nRF_Libraries 组

“nRF_Libraries”组加入的是各种应用的库文件，如 APP 定时器、CRC 校验、软件 FIFO、简易文件系统以及错误处理等等，“nRF_Libraries”组中加入的应用文件如下表所示。

表 4-3: 应用库文件

文件名	路径
nrf_fprintf.c	..\ external\fprintf
nrf_fprintf_format.c	..\ external\fprintf
nrf_ringbuf.c	..\ components\libraries\ringbuf
nrf_memobj.c	..\ components\libraries\memobj
nrf_atomic.c	..\ components\libraries\atomic
nrf_malloc.c	..\ components\libraries\malloc

错误处理模块也是添加到“nRF_Libraries”组的，错误处理模块所需的文件如下表所示，这些文件是固定的，我们只需在工程中加入这些文件即可使用错误处理功能。

表 4-3: 错误处理模块需要加入到工程的文件

文件名	路径
app_error.c	..\components\libraries\util
nrf_assert.c	..\components\libraries\util
app_util_platform.c	..\components\libraries\util

app_error_handler_keil.c	..\components\libraries\util
app_error_weak.c	..\components\libraries\util
nrf_strerror.c	..\components\libraries\strerror

6. nRF_Log 组

“nRF_Log”组中加入的文件如下表所示，Log 程序模块为程序提供日志打印功能。

表 2-7: “nRF_Log”组中加入的文件

文件名	路径
nrf_log_backend_rtt.c	..\components\libraries\log\src
nrf_log_backend_serial.c	..\components\libraries\log\src
nrf_log_backend_uart.c	..\components\libraries\log\src
nrf_log_default_backends.c	..\components\libraries\log\src
nrf_log_frontend.c	..\components\libraries\log\src
nrf_log_str_formatter.c	..\components\libraries\log\src

7. nRF_Segger_RTT 组

“nRF_Segger_RTT”组中加入的文件如下表所示，它们的作用是实现 JLINK-RTT Viewer 作为 Log 输出终端，打印 Log。

表 2-8: “nRF_Segger_RTT”组中加入的文件

文件名	路径
SEGGER_RTT.c	..\external\segger_rtt
SEGGER_RTT_Syscalls_KEIL.c	..\external\segger_rtt
SEGGER_RTT_printf.c	..\external\segger_rtt

2.1.5. 添加工程配置向导文件

工程配置向导文件名称是“sdk_config.h”，我们可以自己编写配置向导文件，不过更方便的方法是直接从 SDK 中拷贝配置向导文件，然后根据自己工程的需求进行修改。

SDK 中各个例子的工程都含有一个配置向导文件“sdk_config.h”，路径是“...\\pca10056\\s140\\config”，我们可以拷贝和我们自己的应用比较接近的例子的配置向导文件来进行修改。

具体操作为拷贝“sdk_config.h”文件到工程的“...\\project\\config”目录下，然后将它添加到工程的“Application”组，添加的时候要注意文件类型选择“All files”，如下图所示。

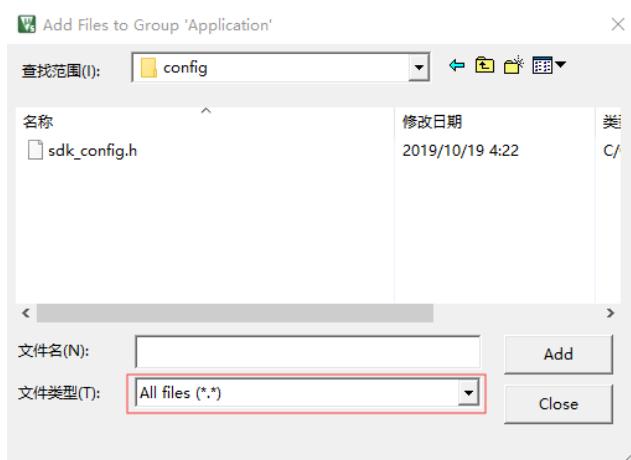


图 4-15：添加“sdk_config.h”文件

2.2. 配置工程

2.2.1. 配置“Target”选项卡

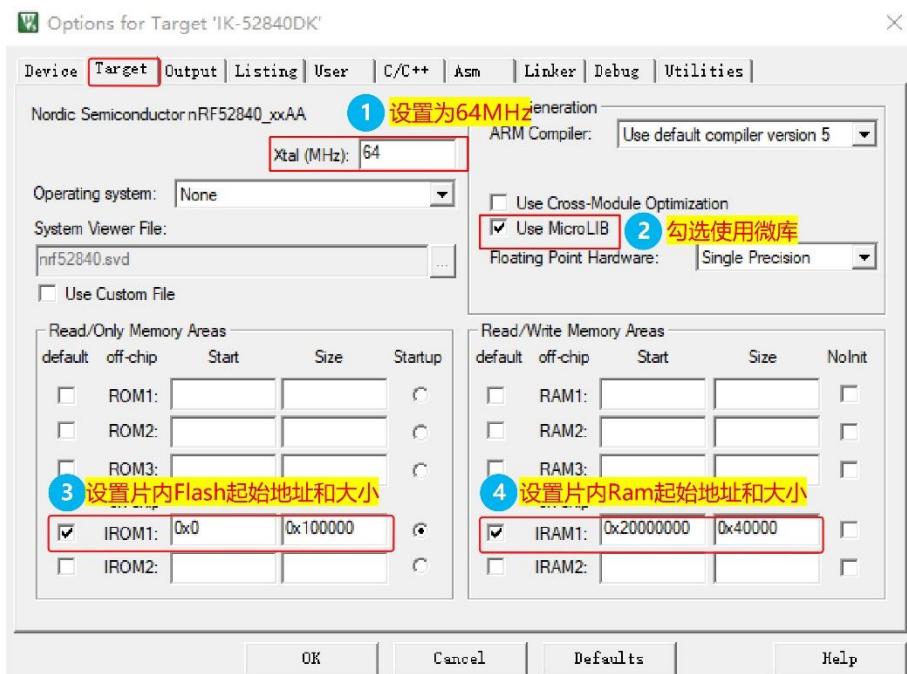


图 4-16：配置“Target”选项卡

主要配置下面几个项目：

- 晶振频率：晶振频率 Xtal 是用于软件仿真的，设置或不设置对硬件烧写和仿真都没有影响。这里设置为 64MHz。
- MicroLib 库：勾选使用 MicroLib 库。
Microlib 是缺省 C 库的备选库。为进一步改进基于 ARM 处理器的应用代码密度，RealView MDK 采用了新型 Microlib C 库（用于 C 的 ISO 标准运行时库的一个子集），并将其代码镜像降低最小以满足微控制器应用的需求。Microlib C 库进行了高度优化以使代码变得很小，可将运行时库代码大大降低。
- 内存设置

查阅芯片的数据手册可知 nRF52840-QIAA 片内 FLASH 是 1MB(对应的 16 进制是: 0x1000000), 片内 RAM 是 256KB(对应的 16 进制是: 0x40000), 设置如下:

片内 ROM 设置:

起始地址(16 进制)	大小(16 进制)
0x0	0x100000

片内 RAM 设置:

起始地址(16 进制)	大小(16 进制)
0x20000000	0x40000

2.2.2. 配置“Output”选项卡

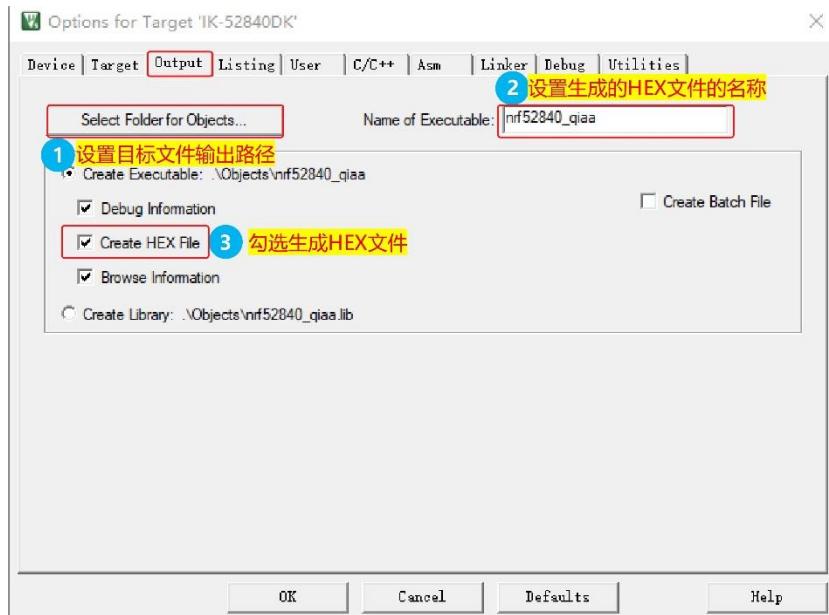


图 4-17: 配置“Output”选项卡

主要配置下面几个项目:

- 指定目标文件输出路径: 若无必要, 输出路径不用修改, 使用默认的即可。
- 设置生成 HEX 文件。

需要勾选“Create HEX File”, 使能生成 HEX 文件, 并设置一下输出的 HEX 文件的名称, 设置之后, 工程编译成功即可生成 HEX 文件, 生成的 HEX 文件位于指定的“目标文件输出路径”的目录下。

- HEX 文件名称: 输入生成的 HEX 文件的名称。

2.2.3. 配置“C/C++”选项卡

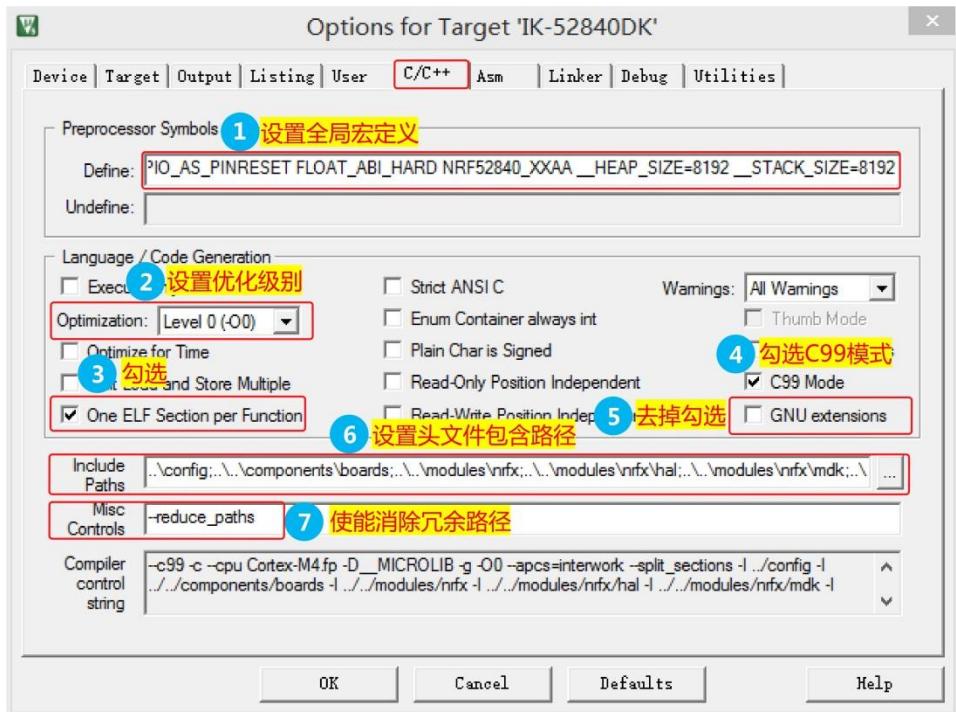


图 4-18: 配置“C/C++”选项卡

主要配置下面几个项目：

1. 全局宏定义

定义被整个工程使用的宏定义，如果有多个宏定义，用空格隔开。注意，这里定义的宏定义，对于整个工程有效。注意使用 Nordic 的库，需要加入下面几个宏定义：

- **BOARD_PCA10056:** 定义之后，工程会包含“pca10056.h”头文件，该头文件中定义了开发板外设如 led、按键、串口等使用的引脚。当然，我们也可以不包含这个头文件，自己来定义引脚。
- **FLOAT_ABI_HARD:** 浮点运算编译选项。
- **CONFIG_GPIO_AS_PINRESET:** 使能 P0.18 的复位功能，即加入该全局宏定义后，P0.18 不能作为普通 IO 使用，只能作为复位引脚。
- **NRF52840_XXAA:** 指定芯片型号是 nRF52 系列中的 nRF52840_QFAA。
- **BSP_DEFINES_ONLY:** 如果使用了 BSP，就需要加入这个全局宏定义。
- **__STACK_SIZE=8192:** 栈大小。
- **__HEAP_SIZE=8192:** 堆大小。

2. 设置优化级别“Optimization”

0 表示不优化，设置越大，优化级别越高。一般仿真调试的时候，优化级别设置为 Level 0（最低），调试完成后，设置为 Level 3（最高），以减小编译后的代码大小。

3. 勾选“One ELF Section per Function”。

One ELF Section per Function 的机制是将每一个函数作为一个优化的单元，而并非整个

文件作为参与优化的单元。该机制具有的这种优化功能特别重要，尤其是在对于生成的二进制文件大小有严格要求的场合。

One ELF Section per Function 对于一个大工程的优化效果尤其突出，对于小工程优化效果不是很明显。想象一下这样的一个应用场合：在 nRF52840 程序开发过程中，我们会使用 SDK 中的组件库“components”，我们加入组件库中的一个文件到工程并不表示我们会使用这个文件中所有的函数，这样，最后生成的二进制文件中就有可能包含众多的冗余函数，造成了存储空间的浪费，通过使用 One ELF Section per Function，即可在最后生成的二进制文件中将冗余函数排除掉，从而节省存储空间。

4. 启用“C99 Mode”

在组件库中，很多地方变量的声明放在了可执行语句的后面(C99 之前不允许这么做)，如果要使用组件库，就需要勾选这个选项，否则编译的时候会出现很多错误。

5. 设置头文件包含路径

当我们引用了一个头文件时，就需要告诉编译器这个头文件的路径，否则，编译器在编译时因为不知道头文件存放的路径，无法定位头文件，在编译时会产生错误。

头文件路径有几种设置方法，最常用的方式就是在 MDK 开发环境中设置(其他几种方式不方便，也不灵活，极少被使用，故在此不提)，即在工程配置的“C/C++”选项卡里面设置，添加头文件包含路径的方法如下。

- 点击“Options for Target...”快捷按钮（魔术棒）。



图 4-19：打开工程配置窗口

- 弹出的窗口中，切换到“C/C++”选项卡，点击“Include Paths”后的按钮。

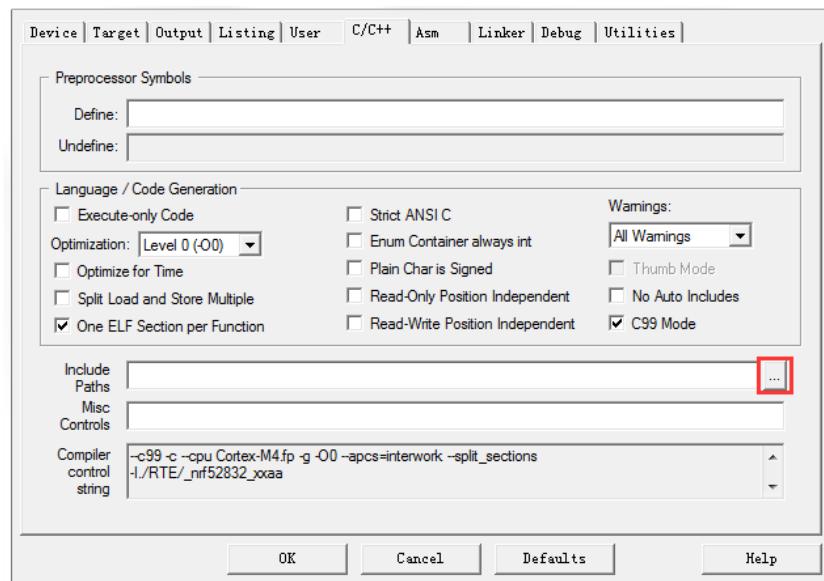


图 4-20：“C/C++”选项卡中设置头文件路径

- 弹出的窗口中，切换到“C/C++”选项卡，点击“Include Paths”后的按钮。

注：如果要设置多个头文件的路径，重复这一步里面的操作就可以了。

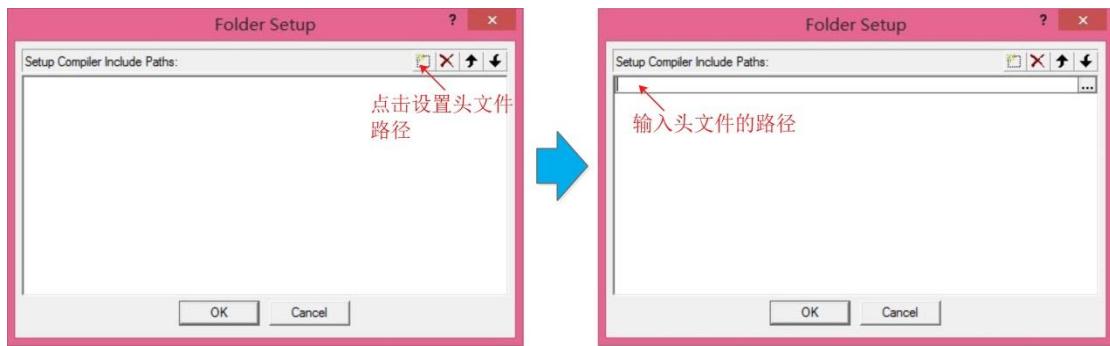


图 4-21：设置头文件路径

对于本例，设置好的头文件路径如下（注意：下图中并没有全部显示工程需要的头文件路径，读者可以打开“实验 6-1：GPIO 输出驱动 led 闪烁”的工程，然后打开该窗口，拖动右边的滚动条查看全部的头文件路径）。

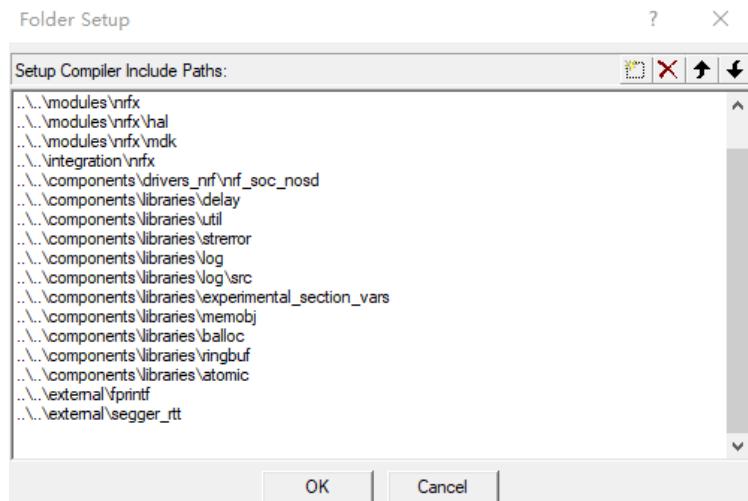


图 4-22：本例设置的头文件路径

❖ 路径的表示方法：

我们这里设置的头文件路径是相对路径，是相对于工程文件“led_blinky.uvprojx”的路径。使用相对路径的优点是：拷贝整个工程到其他盘，不会影响到工程，因为路径是相对的。相对路径中必须要掌握的几种路径表示方法：

- “..＼”：表示工程文件所在目录向上一级的目录，“..＼”可以连用，“..＼..＼” 表示工程文件所在目录向上两级的目录，以此类推。
- “.＼”：表示的是当前路径。

6. 设置消除冗余路径

Misc Controls 栏中填入“--reduce_paths”关键字，设置 MDK 自动消除冗余路径。

2.2.4. 配置“Linker”选项卡

“Linker”选项卡中需要配置的是忽略警告设置，BLE 工程编译的时候会产生 6330 的警告，这里我们在“Misc controls”栏加入“--diag_suppress 6330”忽略掉 6330 警告，如下

图所示。

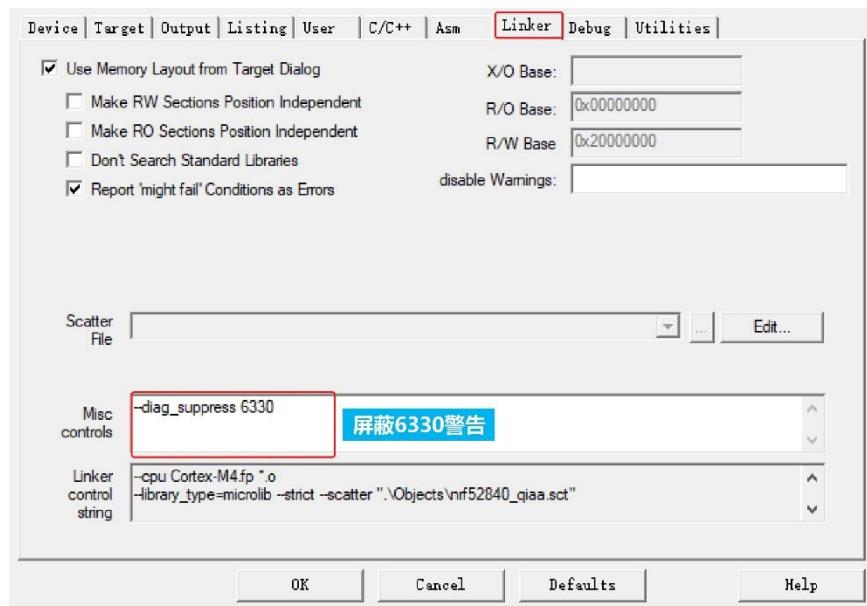


图 4-23：配置“Linker”选项卡

2.2.5. 配置“Debug”选项卡

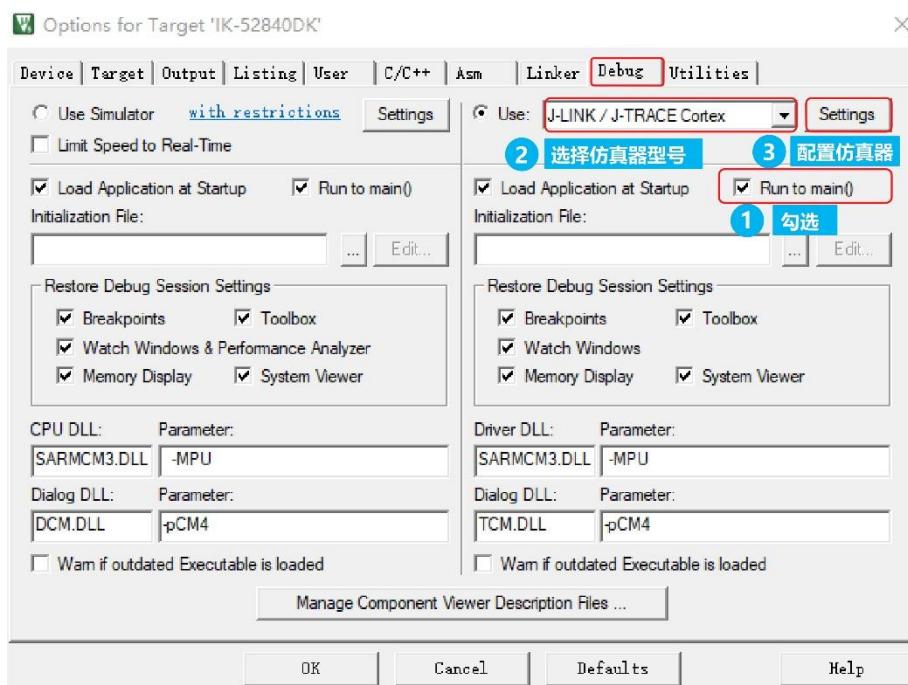


图 4-24：配置“Debug”选项卡

主要配置下面几个项目：

- Run to main(): 设置仿真时，程序是否自动运行到 main()函数，勾选后，仿真时，程序会自动运行到 main()函数。
- 仿真器设置: 本教程使用的仿真器是 JLINK，所以这里选择“J-LINK/J-TRACE Cortex”，之后，点击“Setting”按钮进入设置界面。(注意：设置时需要将 JLINK 仿真器连接到计算机)。

仿真器设置界面如下，设置时，先切换到“Debug”选项卡，将调试接口设置为“SW”。如下图所示。

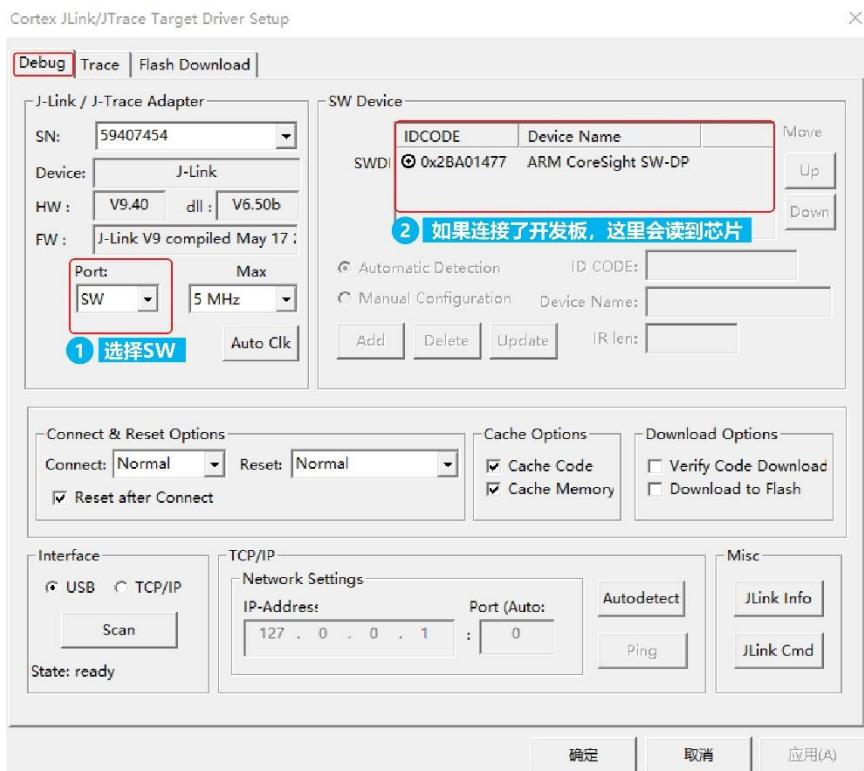


图 4-25：配置调试接口

切换到“Flash Download”选项卡，加载编程算法。

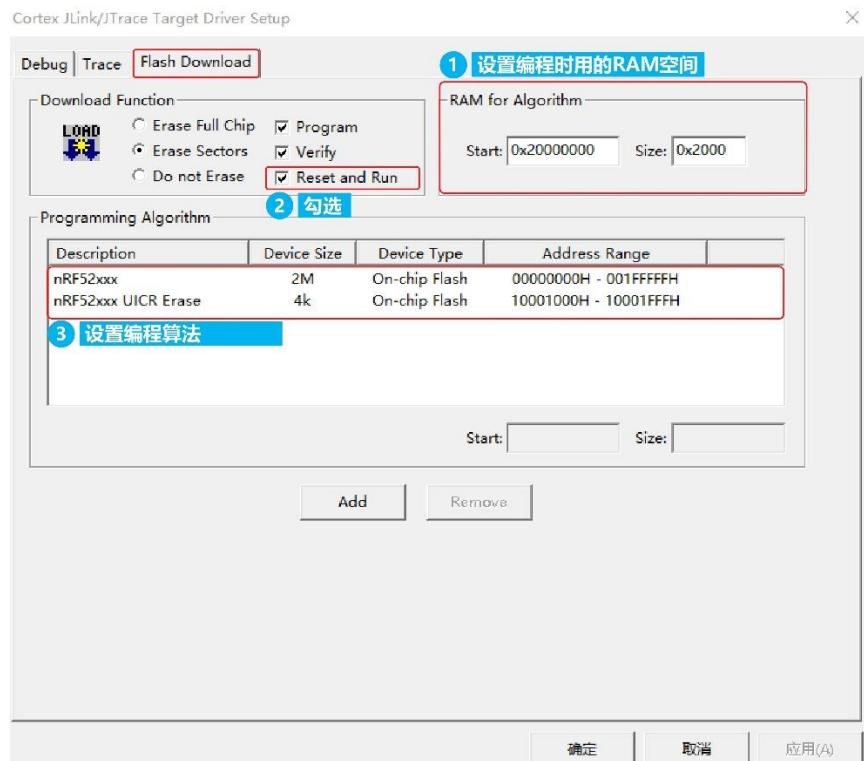


图 4-26：配置下载选项

“Flash Download”选项卡主要设置项目如下：

- **Reset and Run:** 下载完成后自动复位并运行程序，这一项很重要。有时候我们发现程序下载后需要断电重启后才能运行，原因有可能就是漏勾选了这一项。
- **RAM for Algorithm:** 指定用于烧写程序的 RAM 区域，一般使用默认设置即可。
- **program algorithm:** Flash 编程算法，一般建立工程选择芯片后会自动添加，如果没有，点击“ADD”按钮添加即可。

到这里，新建裸机工程和配置工程完成，接下来，我们加入驱动指示灯 D1 闪烁的代码，实现指示灯闪烁的功能。

2.3. 编写驱动 LED 代码

2.3.1. 新建 main.c 文件并添加到工程。

通常，工程都会包含一个“main.c”文件，用于存放 C 程序的入口函数（main()函数），所以，我们需要先新建一个“main.c”文件并添加到工程。

1. 执行“File→New”新建文件，如下图

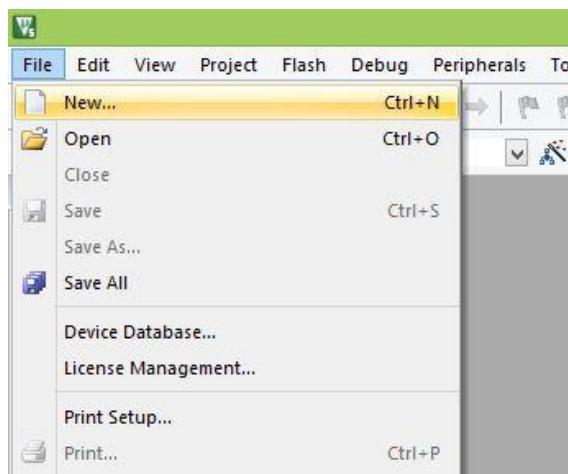


图 4-27：新建文件

2. 点击保存按钮“Save”将文件保存为“main.c”



图 4-28：保存文件

3. 将“main.c”文件添加到工程

保存“main.c”文件之后，还需要将“main.c”文件加入到工程中，双击组名 Application 打开添加文件窗口，导航到“main.c”文件的存放路径，添加即可。

2.3.2. 引用头文件

正如上文所说，我们需要用到“delay”和GPIO组件，并且在 main.c 中需要调用这两个组件中的函数，所以，需要在 main.c 中引入这两个头文件，如下图所示：

代码清单：引用头文件

```
1. #include <stdbool.h> //布尔变量定义的头文件，引用之后可以使用 true 和 false  
2. #include <stdint.h> //数据类型定义头文件  
3. #include "nrf_delay.h" //软件延时头文件  
4. #include "boards.h" //板卡定义头文件，该头文件包含了 GPIO 的头文件
```

❖ 注意：stdbool.h 和 stdint.h 是标准 C 库文件，所以引用的时候使用“<>”符号。对于自己编写的头文件，用“”引用即可。另外，本例因为代码比较简单，没有用到这 2 个头文件，但是因为这 2 个头文件在代码稍微多一点的例子里面都会用到，所以这里我们也加上对它们的引用。

2.3.3. 编写代码

main.c 文件中加入如下的代码，驱动 LED 指示灯 D1 以 200ms 的间隔闪烁。
❖ 注：此处是为了后续编译和下载程序的演示，关于指示灯驱动实验的教程详见后面的章节。

代码清单：驱动 led 指示灯闪烁

```
1. #include <stdbool.h>  
2. #include <stdint.h>  
3. #include "nrf_delay.h"  
4. #include "boards.h"  
5.  
6.  
7. int main(void)  
8. {  
9.     //配置用于驱动 LED 指示灯 D1 的引脚 P0.17 为输出  
10.    nrf_gpio_cfg_output(LED_1);  
11.  
12.    while(true)
```

```

13.    {
14.        //P0.17 输出高电平, D1 熄灭
15.        nrf_gpio_pin_set(LED_1);
16.        //软件延时 200ms
17.        nrf_delay_ms(200);
18.        //P0.17 输出低电平, D1 点亮
19.        nrf_gpio_pin_clear(LED_1);
20.        //软件延时 200ms
21.        nrf_delay_ms(200);
22.    }
23. }
```

2.4. 编译工程

点击 Rebuild 编译按钮（下图中的第 3 个），编译工程。



图 4-29：3 种编译方式

我们可以看到，编译按钮共有 3 个，从左到右分别是： Translate，Build，Rebuild 三个编译按钮，它们的区别如下：

- Translate（按钮 1）：是编译当前打开的活动文档。
- Build（按钮 2）：增量编译，编译工程中上次修改的文件及其它依赖于这些修改过的文件的模块，同时重新链接生成可执行文件。如果工程之前没编译链接过，它会直接调用 Rebuild All。
- Rebuild（按钮 3）：全部重新编译。

编译后，注意观察信息窗口输出的信息，若输出信息提示无错误表示编译成功。编译成功后即可将程序烧写到开发板中运行。

```

Build Output
compiling system_nrf52.c...
compiling main.c...
linking...
Program Size: Code=1532 RO-data=1056 RW-data=4 ZI-data=16484
".\Objects\led_blinky.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed: 00:00:03
```

图 4-30：编译信息

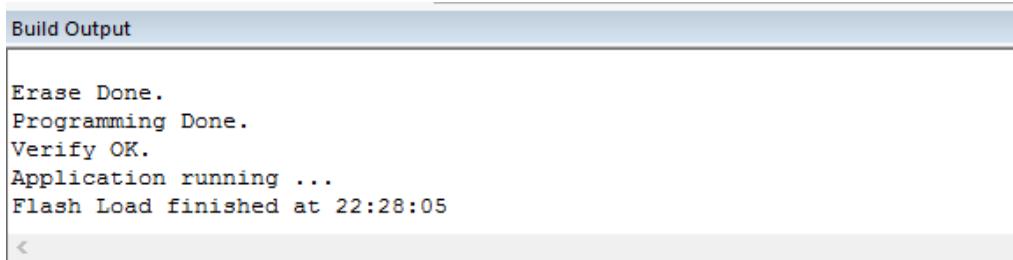
2.5. 程序烧写

点击下载按钮，如下图所示，将程序烧写到开发板。



烧写完成之后，注意观察信息窗口输出的信息，如下图所示，信息窗口输出信息提示：烧写成功、校验成功、运行程序。由此提示信息可以看出程序已经烧写成功，并且开始运行。

- ❖ 要注意一下“Application running ...”这个提示信息，这表示程序开始运行。如果没有这个提示，说明程序仅仅是烧写到开发板，并没有运行(下载后运行可以在 Debug 项目里面设置，前文已有说明)。有时候，我们会遇到程序烧写后不运行，但是复位或断电一下就可以正常运行了，这时，要注意一下烧写后有没有输出这个提示信息。



```

Build Output

Erase Done.
Programming Done.
Verify OK.
Application running ...
Flash Load finished at 22:28:05

```

图 4-31：烧写成功输出的提示信息

2.6. MDK 常用操作

1. 启用或关闭 MDK 每次启动时自动加载最近一次打开的工程

MDK 安装完成后，默认是开启了“启动时自动加载最近一次打开的工程”的功能，这个功能有时候挺麻烦，如果要关闭这个功能，按照下面的步骤操作。

- 1) 点击“配置”按钮，如下图所示：



图 4-32：打开 MDK 配置窗口

- 2) 在弹出的 MDK 配置窗口中切换到“other”选项卡，取消勾选“Open most recent project”即可关闭“启动时自动加载最近一次打开的工程”的功能。

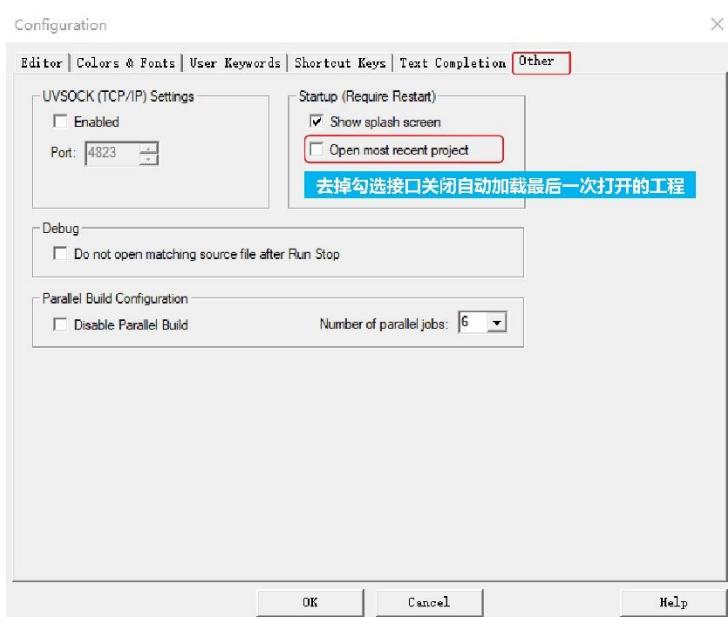


图 4-33：启用或关闭“Open most recent project”

2. 关闭/打开语法动态检查

❖ 注：建议不要关闭语法动态检查功能。

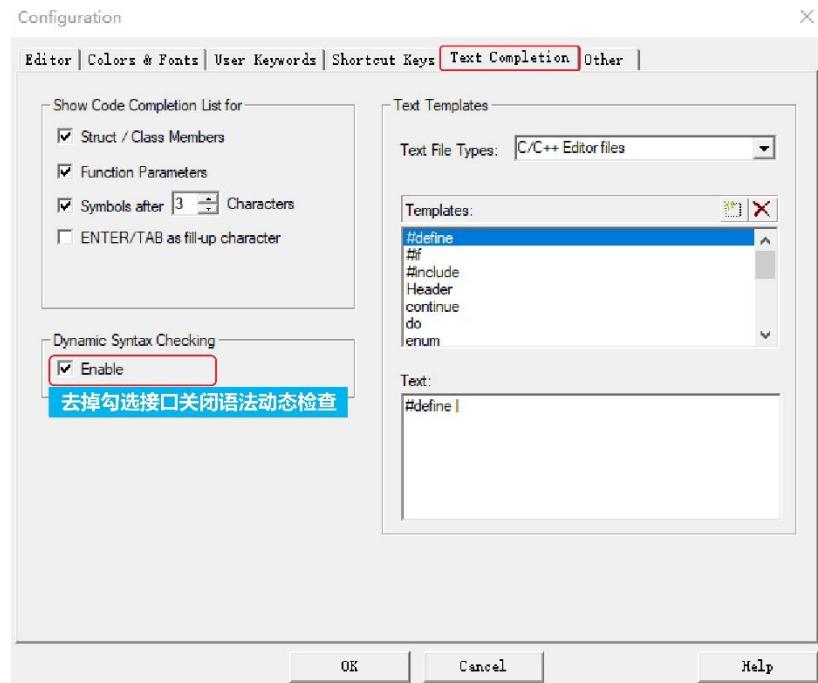


图 4-34: 关闭或打开语法动态检查

3. 删 除 pack 包

当我们在 MDK 中安装了多个 Pack 版本的时候，如果有些用不上的版本需要移植，可以按照下面的步骤移植不用的 Pack 版本。

1) 点击“Pack Installer”按钮，如下图所示：

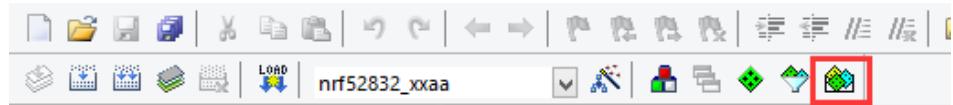


图 4-35: 打开“Pack Installer”

2) Pack Installer 窗口中，选中“选中 Nordic Semiconductor”，切换到 packs 选项卡，点击需要移除的 pack 后面的“Remove”按钮即可移除该版本的 packs。

❖ 注意：这里的操作是移除，而不是删除 packs，移除后“Remove”会变为“Unpack”，如果需要安装改版本的 packs，点击“Unpack”按钮即可安装。

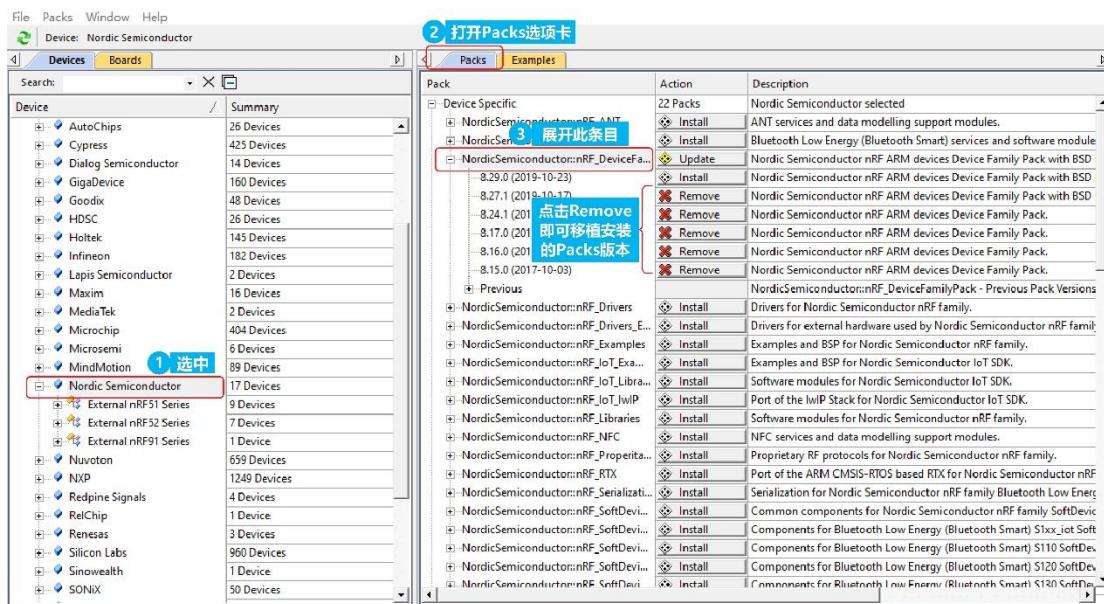


图 4-36: 移除 packs

4. 快速打开工程“map”

如下图所示，双击目标设备名称即可打开工程 map 文件。

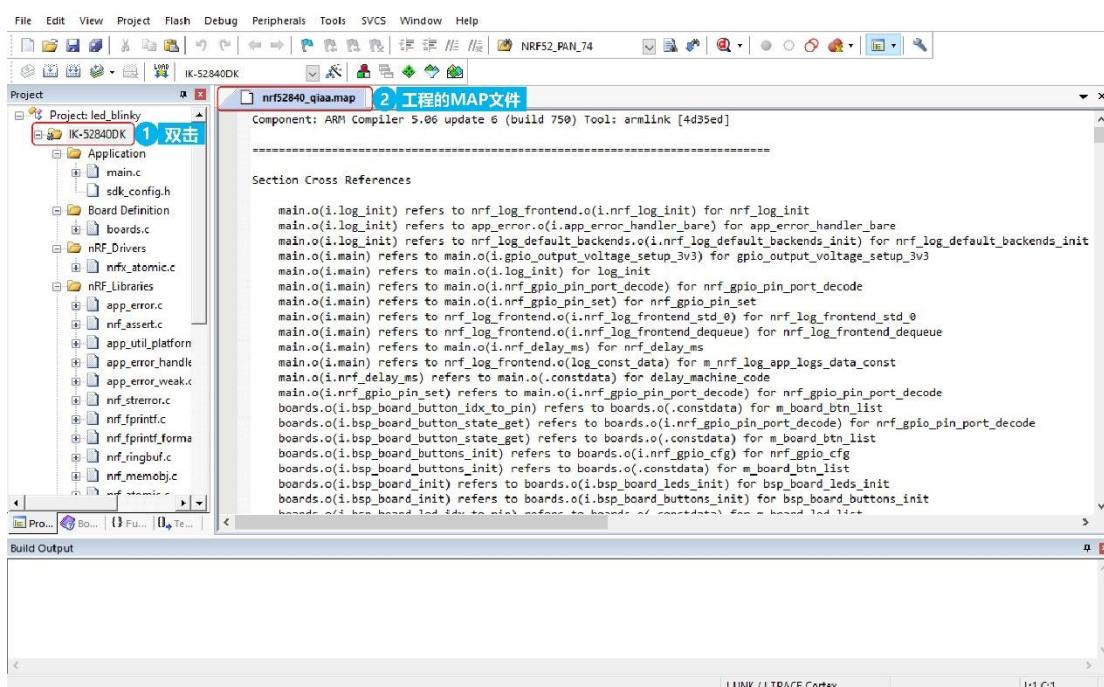


图 4-37: 打开工程 map 文件

5. 批量注释和批量取消注释

- 批量注释：先选中需要注释的代码，然后点击下图中红框内的按钮即可批量注释代码。



图 4-38: 批量注释

- 批量取消注释：先选中需要取消注释的代码，然后点击下图中红框内的按钮即可批量取消代码注释。



图 4-39：批量取消注释

第五章：程序下载

1. 下载方式概述

nF52840 常用的下载方式有 3 种，这 3 种方式都可以下载裸机程序和 BLE 程序他们的特点如下：

1. MDK 直接下载

可以在 MDK 中直接下载协议栈和应用程序，也可以仿真应用程序，优点是在 MDK 开发环境中直接操作，开发时使用这种方法下载比较方便。需要注意的是对于应用程序（BLE 的应用程序或者裸机程序）和 BLE 协议栈需要设置不同的编程算法，另外，只有使用 BLE 协议栈的编程算法才能全片擦除芯片。

2. J-Flash 下载

可以下载协议栈（协议栈 HEX 文件）和应用程序，不能仿真。优点是下载速度快，操作方便，适合开发时下载或者批量生产时下载。

3. nrfjprog 命令行下载

使用 J-Link 仿真器对 nRF5x 系列 SoC 进行编程的命令行工具，可以启用芯片的回读保护机制。

❖ 注：本节涉及到了蓝牙 BLE 程序的下载，这在《nF52840 开发指南-下册》中会用到。

2. 方式 1：MDK 直接下载

2.1. 下载裸机程序

裸机工程新建工程时会根据选择的芯片自动加载编程算法，而不需要自己去设置编程算法。一般我们会检查一下编程算法是不是正确的，步骤如下：

1. MDK 中点击魔术棒，如下图所示：



图 5-1：点击魔术棒

2. 在弹出的窗口中切换到“Utilities”选项卡，然后点击“Setting”，打开编程算法设置窗口

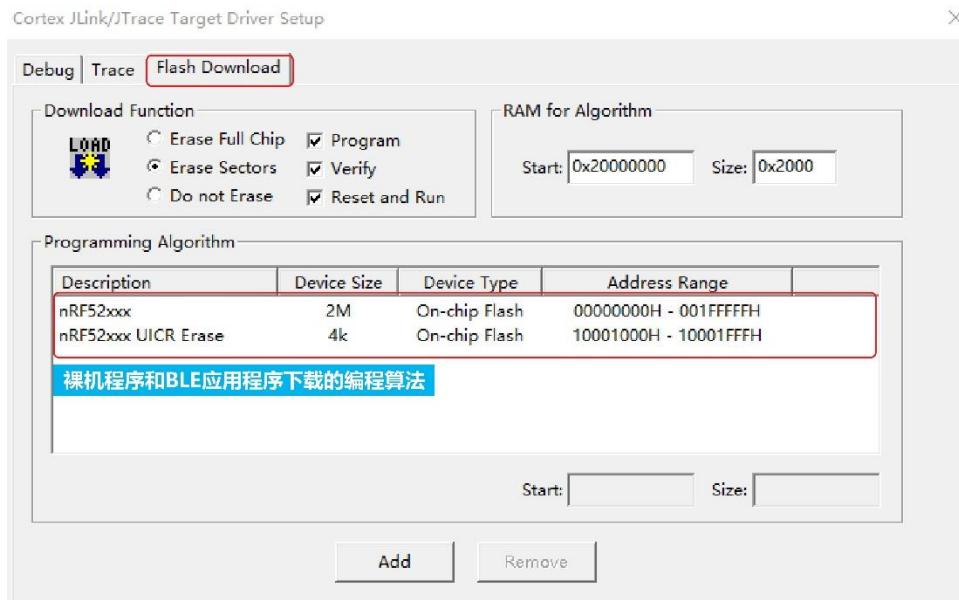


图 5-2 裸机程序和 BLE 应用程序下载的编程算法

编程算法设置好之后，编译工程，编译通过，点击下载按钮将程序下载到开发板运行，如下图所示：

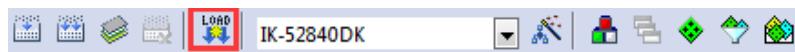


图 5-3：下载程序

2.2. 下载 BLE 程序

Nordic 的 BLE 程序的结构是协议栈和应用程序分开，所以下载需要分成两步进行，首先下载协议栈，然后下载应用程序。

本节以 nRF52840 蓝牙串口透传为例来说明如何使用 MDK 来下载协议栈、应用程序和全片擦除芯片。

2.2.1. 下载协议栈

1. MDK 中打开蓝牙串口透传工程

在 MDK5 中执行 “Project→Open Project” 打开 “nRF5_SDK_16.0.0_98a08e2\examples\ble_peripheral\ble_app_uart\pca10056\s140\arm5_no_packs” 目录下的工程 “ble_app_uart_pca10056_s140.uvproj”。

2. 切换到下载协议栈的工程

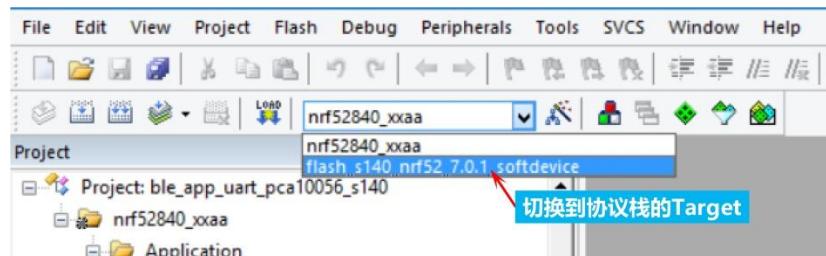


图 5-4：切换到下载协议栈的工程

3. MDK 中点击魔术棒，如下图所示：



图 5-5：点击魔术棒

4. 在弹出的窗口中切换到“Utilities”选项卡，然后点击“Setting”，打开编程算法设置窗口

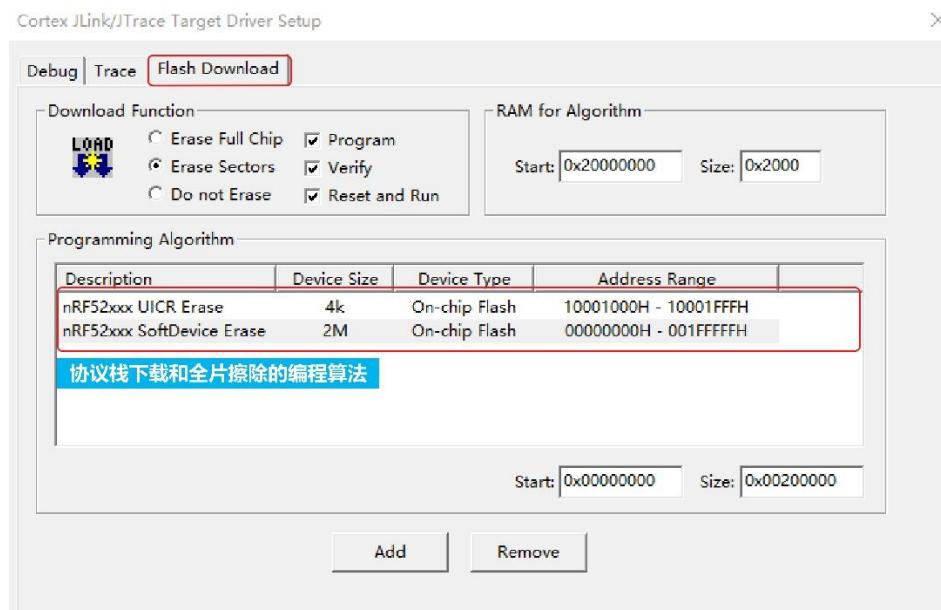


图 5-6：下载协议栈和全片擦除时的编程算法

5. 点击下载按钮，即可下载协议栈。

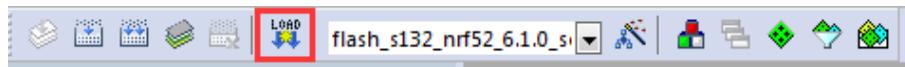


图 5-7：下载协议栈

❖ 特别注意：协议栈是预编译的，是以“HEX”文件格式提供的，无需编译工程，只需执行下载就可以了。

2.2.2. 下载应用程序

1. 切换到下载应用程序的工程

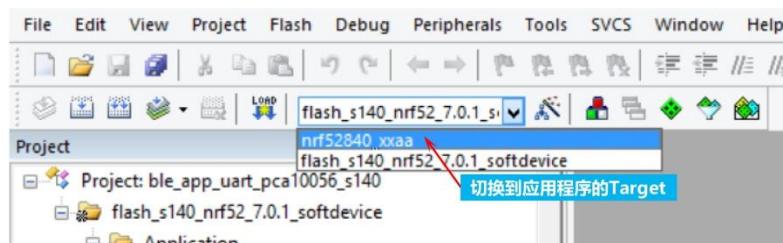


图 5-8：切换到下载应用程序的工程

2. 编程算法设置和裸机程序一样。
3. 编译工程后，点击下载按钮，即可下载应用程序。



图 5-9：下载应用程序

2.2.3. 全片擦除芯片

只有使用协议栈下载的编程算法才可以实现全片擦除，下载应用程序的编程算法是无法进行全片擦除的，这一点要特别注意，全片擦除的步骤如下。

1. 编程算法设置为协议栈编程算法

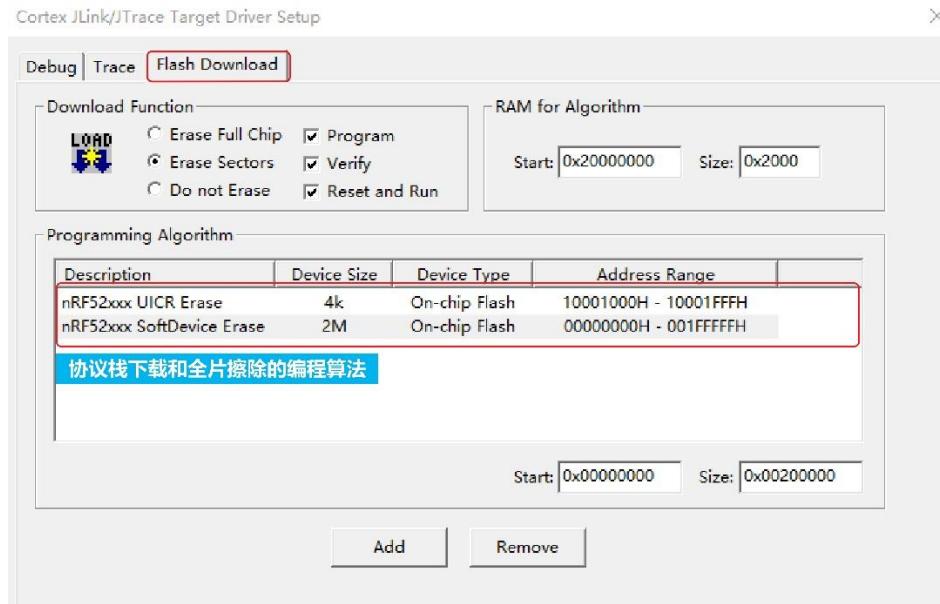


图 5-10：下载协议栈和全片擦除时的编程算法

2. 执行“Flash→Erase”，如下图所示。

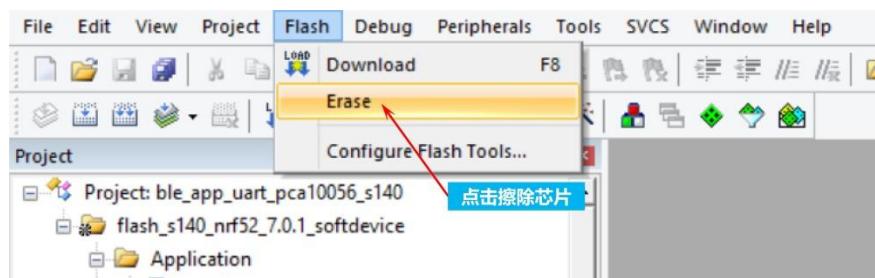


图 5-11：全片擦除芯片

- 如果是在裸机程序（没有使用蓝牙协议栈 Softdevice）中执行了全片擦除，擦除完成后，再将编程算法改为应用程序的编程算法。

2.3. 本节常见问题

1. 下载时出现 Error:Flash Download failed – “Cortex-M4” 的错误，如下图所示。

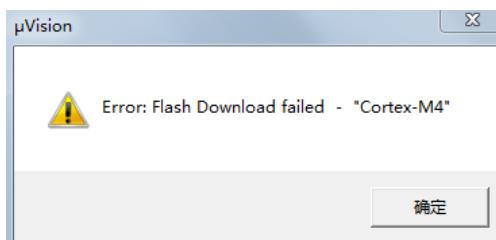


图 5-12：下载错误

原因 1：这是因为芯片中已经下载了协议栈，协议栈对占用的 Flash 空间进行了保护，并且

协议栈的地址是从 0 开始的，这时候来再下载地址从 0 开始的程序，就会出现这个现象。下面是产生这个问题最常见的两种错误操作和解决办法。

- 1) 芯片里面下载了协议栈，在没有进行全片擦除的情况下下载裸机程序。解决办法是全片擦除芯片即可。
- 2) 下载 BLE 工程时，因为修改了应用程序的工程，导致内存配置被恢复了（即 Flash 和 RAM 起始地址都恢复到了从 0 开始，这就和协议杂占用的 Flash 空间重合了，而协议栈对占用的 Flash 空间进行了保护，从而导致无法下载）。解决办法是按照操作错误之前的内存配置重新配置内存。

原因 2：没有加载编程算法

打开编程算法设置窗口，会看到编程算法栏为空，即没有设置编程算法。解决办法是按照前文所述设置编程算法即可。

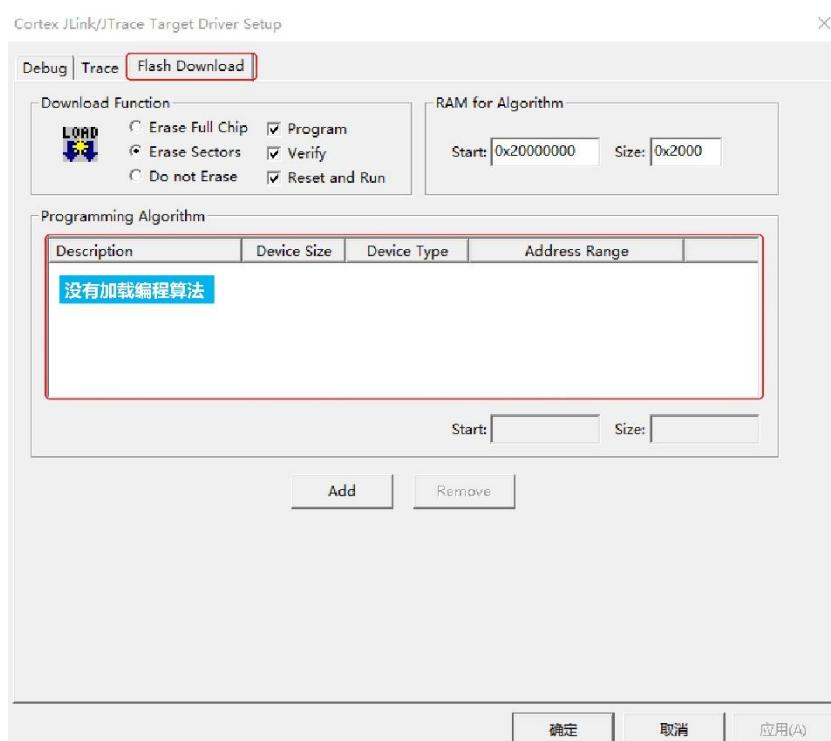


图 5-13：没有设置编程算法

2. 下载时出现 Overlapping of Algorithms at Address xxxxxxxxH 错误

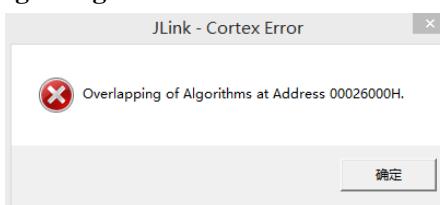


图 5-13：下载错误

原因：编程算法设置错误。

同时设置了下载协议栈和应用程序的编程算法，如下图所示，而下载协议栈的编程算法和下载应用程序的编程算法是不能同时使用的。解决办法是按照前文所述设置编程算法即可。

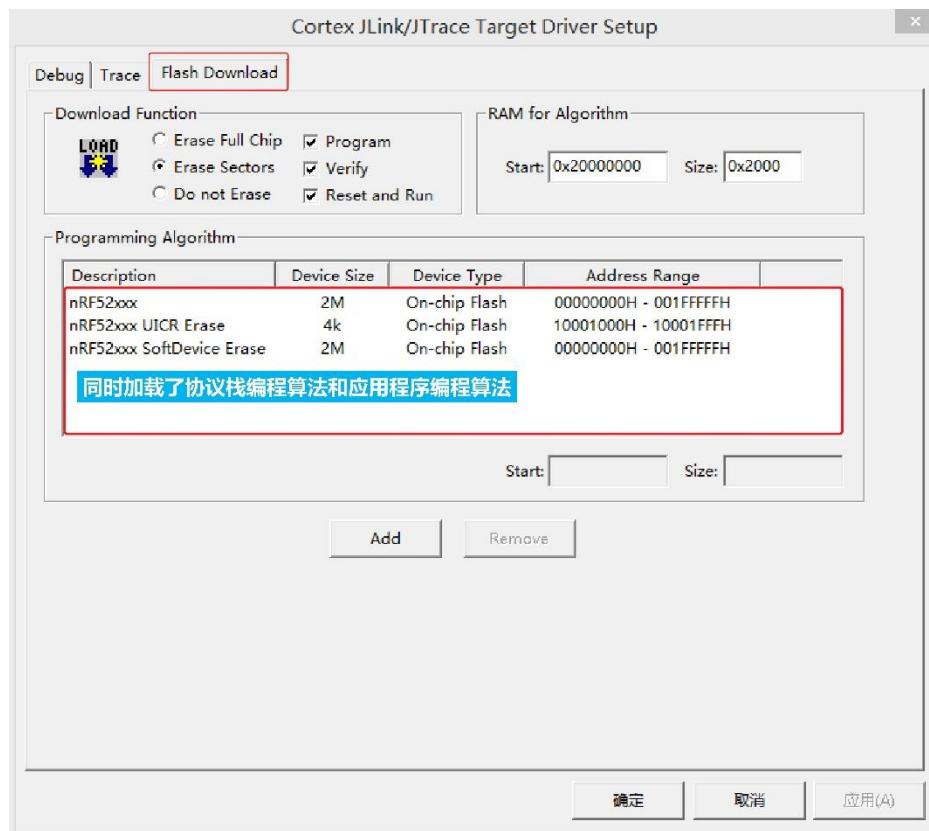


图 5-14：编程算法设置错误

3. 下载时出现 Err: Flash Download failed – Could not load file 错误

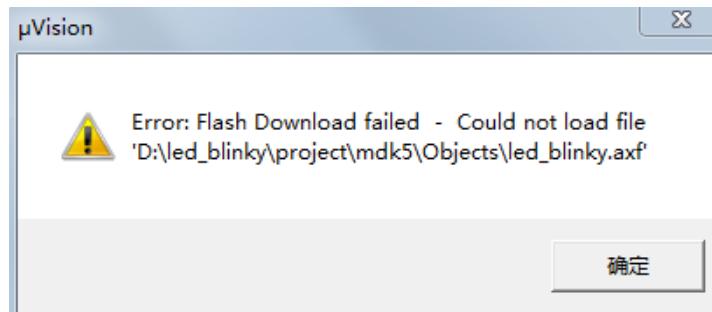


图 5-15：无法加载文件

原因：工程没有编译成功。

工程没有编译成功时执行下载会出现该错误提示，解决办法：编程成功后再执行下载。

3. 方式 2：使用 J-Flash 下载

3.1. J-Flash 简介

J-Flash 是 SEGGER (J-LINK 仿真器厂商) 发布的一款单独的 Flash ISP 烧写软件，支持将 HEX 和 BIN 格式文件烧写到单片机的 Flash。J-Flash 烧写速度极快，远远快于 Nordic 提供的 ISP 下载软件 nRFgo Studio。

J-Flash 集成在 J-LINK 驱动里面，当我们安装了 J-LINK 驱动后，也就安装了 J-Flash。安装 J-LINK 驱动后，打开 J-LINK 驱动安装的路径，可以看到 J-Flash 执行程序，如下图所示。

名称	修改日期	类型	大小
Devices	2019/11/10 18:39	文件夹	
Doc	2019/11/10 18:39	文件夹	
ETC	2019/11/10 18:39	文件夹	
GDBServer	2019/11/10 18:39	文件夹	
RDDI	2019/11/10 18:39	文件夹	
Samples	2019/11/10 18:39	文件夹	
USBDriver	2019/11/10 18:39	文件夹	
JFlash.exe	2019/9/7 0:00	应用程序	851 KB
JFlashLite.exe	2019/9/7 0:00	应用程序	184 KB
JFlashSPI.exe	2019/9/7 0:00	应用程序	559 KB
JFlashSPI_CL.exe	2019/9/7 0:00	应用程序	468 KB
JLink.exe	2019/9/7 0:00	应用程序	297 KB
JLink_x64.dll	2019/9/7 0:01	应用程序扩展	14,573 KB
JLinkARM.dll	2019/9/7 0:00	应用程序扩展	13,745 KB

图 5-16: J-Flash

Nordic 的 nRF5xx 系列 BLE 芯片都可以使用 J-Flash 烧写，本文以烧写 IK-52840DK 开发板为例，来说明如何使用 J-Flash。

3.2. 使用 J-Flash

3.2.1. 启动 J-Flash

双击 J-Flash 图标，启动 J-Flash，启动后，会弹出欢迎界面，如下图所示，这里可以打开之前的工程或者新建工程。

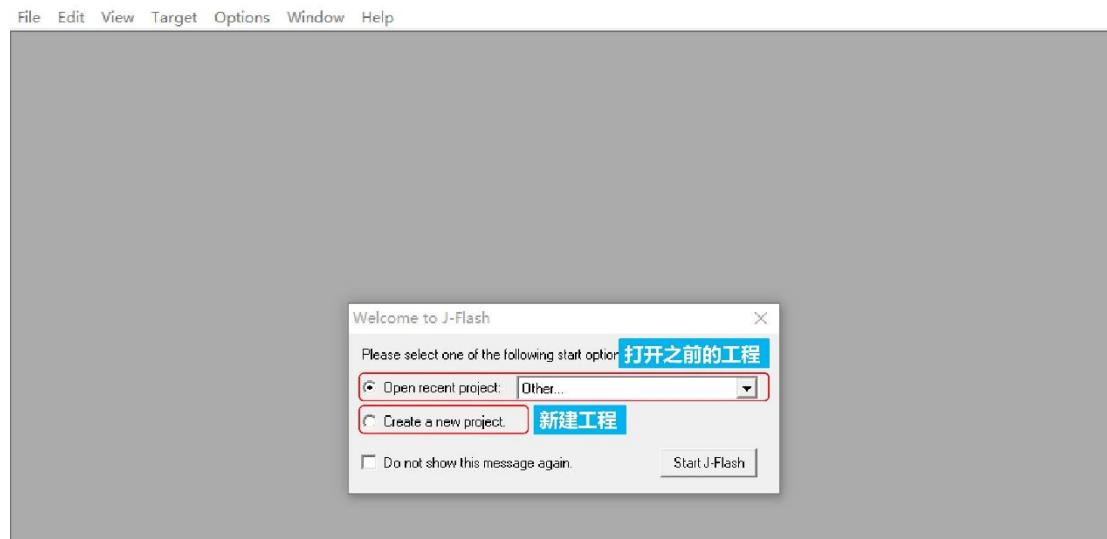


图 5-17: 启动 J-Flash

关闭欢迎窗口后，J-Flash 会自动新建一个工程，接下来我们看一下如何配置工程。

3.2.2. 配置工程

- 执行“Options→Project Settings”或者按下“ALT + F7”快捷键，打开工程配置窗口，如下图所示。

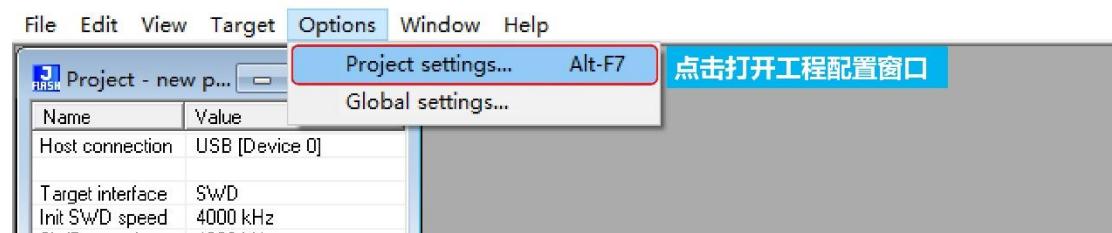


图 5-18：打开工程配置窗口

- 切换到“Target Interface”选项卡，设置目标设备接口为“SWD”，如下图所示。

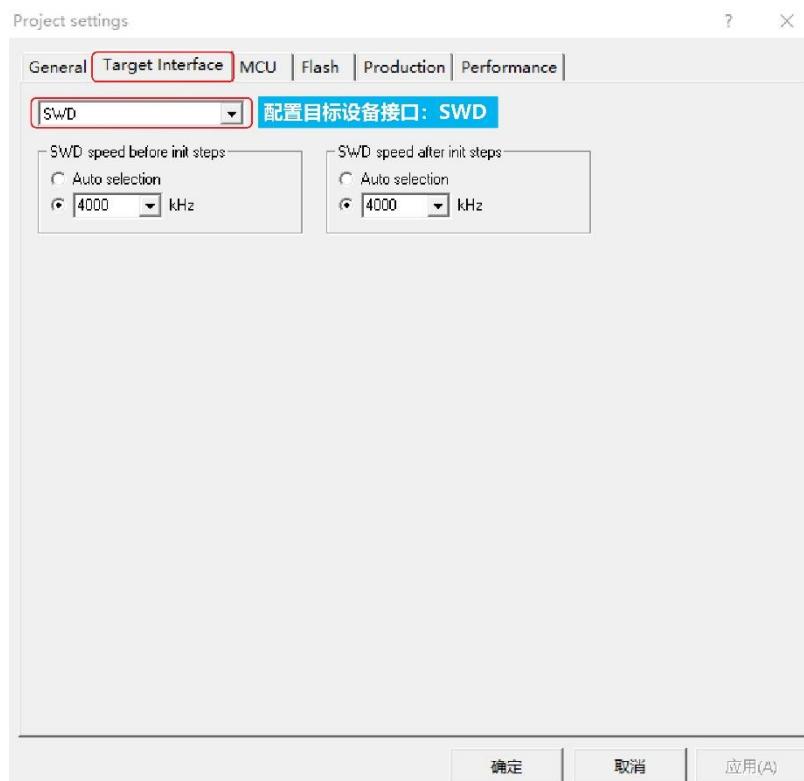


图 5-19：配置目标设备接口

- 切换到“MCU”选项卡，打开器件选择窗口，如下图所示。

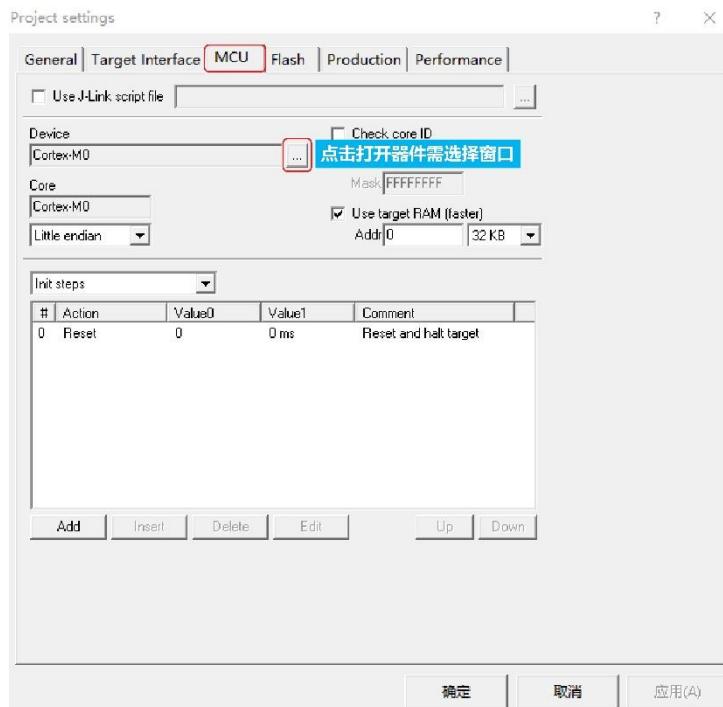


图 5-20：打开器件型号选择窗口

4. 选择器件型号，因为待烧写的开发板是 IK-52840DK，所用芯片是 nRF52840-QIAA，所以这里选择 nRF52840-xxAA，如下图所示。

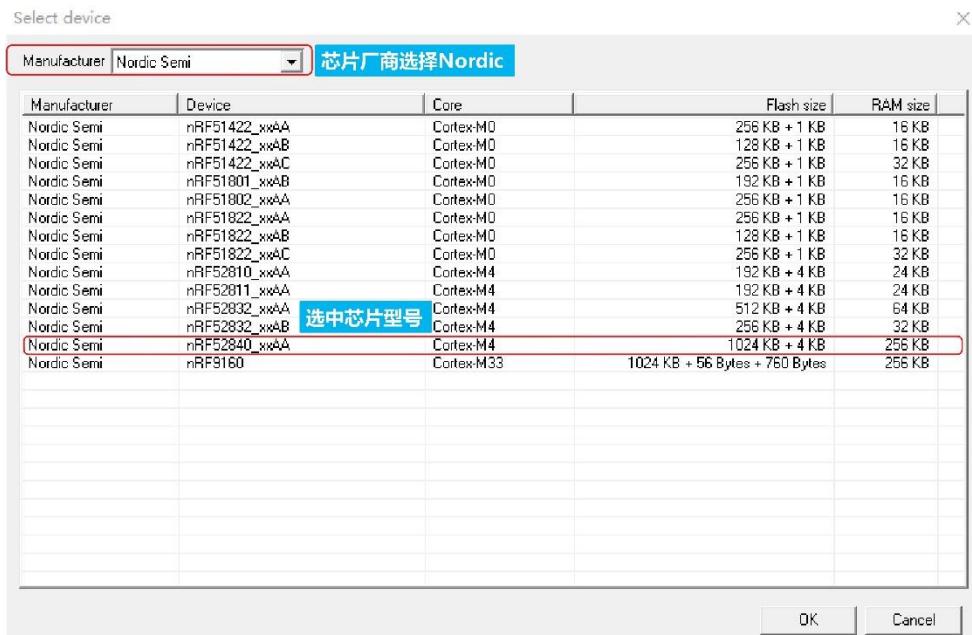


图 5-21：选择烧写的器件型号

3.2.3. 擦除芯片

执行“Target→Manual Programming→Erase Chip”或按下快捷键 F4，即可擦除芯片。注意：该擦除操作是全片擦除，即擦除片内所有 Flash 和掉电保持的寄存器。

3.2.4. 烧写和运行程序

1. 打开待烧写的文件。

执行“File→Open data file”，弹出打开文件窗口，导航到待烧写的文件的路径，打开待烧写文件。

2. 按下“F7”快捷键，烧写目标板。

3. 按下“F9”快捷键，运行烧写的程序。

❖ 注：“Target→Manual Programming”菜单下还可以执行如校验、读Flash等操作，读者可以使用一下，熟悉这些操作。

3.2.5. 关闭 J-Flash

关闭 J-Flash 时，会提示是否保存当前工程，这里我们可以保存本次配置的工程，这样，下次使用的时候，在欢迎界面直接打开保存的工程就可以了，而不用再次配置工程。

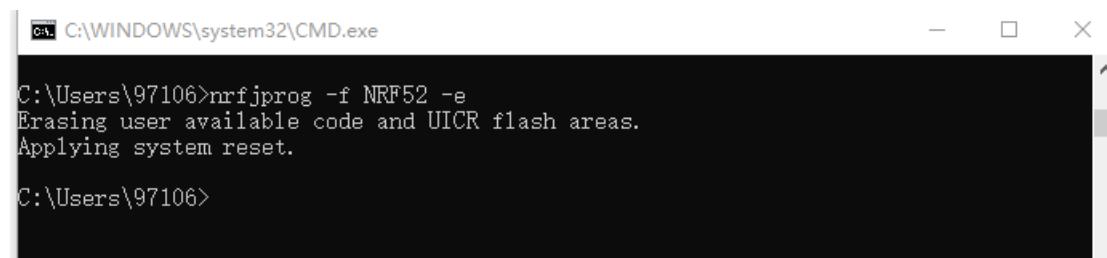
3.3. 方式 3：使用 nrfjprog 下载

- nrfjprog 是使用 J-Link 仿真器对 nRF5x 系列 SoC 进行编程的命令行工具，因此，需要使用 J-LINK 仿真器并安装 J-LINK 驱动。
- nrfjprog 是集成在 nRF-Command-Line-Tools 中的，在搭建开发环境时已经安装了 nRF-Command-Line-Tools，因此，无需再安装其他软件即可使用 nrfjprog。
- 用户除了可以用命令行实现下载、擦除等操作之外，还可以自己编写下载软件，引用 nrfjprog DLL “nrfjprog.dll”，开发符合自己需求的下载软件。

1. 全片擦除

命令：nrfjprog -f NRF52 -e

执行全片擦除后，会擦除片内所有 Flash 和掉电保持的寄存器，全片擦除操作如下图所示。



```
C:\WINDOWS\system32\cmd.exe
C:\Users\97106>nrfjprog -f NRF52 -e
Erasing user available code and UICR flash areas.
Applying system reset.

C:\Users\97106>
```

图 5-22：全片擦除

2. 下载

nrfjprog 的命令是可以组合的，对于程序下载，可以使用下面两种组合命令，注意他们的区别。

1) 仅下载+复位运行：芯片必须是空的（擦除过的），否则下载会失败。

命令：nrfjprog -f NRF52 --program G:\hex\blinky.hex -r

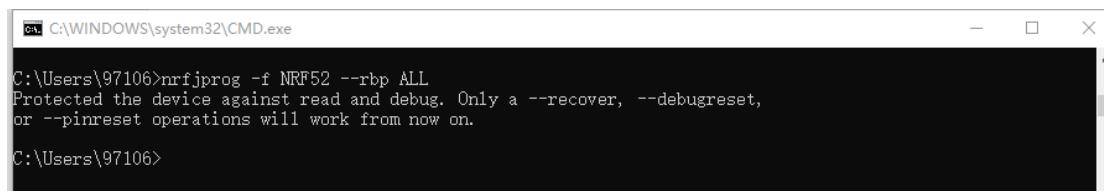
2) 全片擦除+下载+复位运行。

命令: nrfjprog -f NRF52 --program G:\hex\blinky.hex - -chiperase -r

3. 启用回读保护机制

命令: nrfjprog -f NRF52 --rbp ALL

该命令执行后, 无法使用仿真器通过 DEBUG 接口读出芯片的数据, 当我们发布产品的时候, 可以通过启用回读保护防止别人读取芯片中烧写的固件, 从而保护产品的权益不被侵犯。



```
C:\Users\97106>nrfjprog -f NRF52 --rbp ALL
Protected the device against read and debug. Only a --recover, --debugreset,
or --pinreset operations will work from now on.

C:\Users\97106>
```

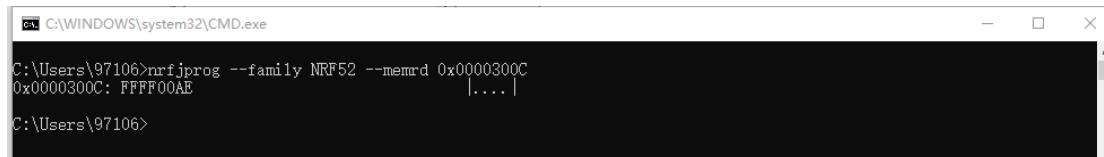
图 5-23: 启用回读保护

4. 读出 SoftDevice 的 FWID

只有已经下载了 SoftDevice 的芯片才可以读出 FWID。

- nRF51xx 系列命令: nrfjprog --family NRF51 --memrd 0x0000300C
- nRF52xx 系列命令: nrfjprog --family NRF52 --memrd 0x0000300C

示例: 读 SDK15.3 中 NRF52840 对应的 SoftDevice s140_nrf52_6.1.1 的 FWID, 如下图所示, 读出的数据是: 0xFFFF00AE, 表示协议栈的 FWID 是 0xAE, 根据 FWID 即可知道协议栈的版本 (FWID 在固件更新 DFU 中会用到)。



```
C:\Users\97106>nrfjprog --family NRF52 --memrd 0x0000300C
0x0000300C: FFFF00AE |....|
C:\Users\97106>
```

图 5-24: 读出 SoftDevice 的 FWID

5. 恢复: 全片擦除并禁用回读保护机制(如果启用)

命令: nrfjprog -f NRF52 -recover

该命令执行后, 会擦除片内所有 Flash 和掉电保持的寄存器。

6. 帮助列表

命令: nrfjprog -h

7. 读出版本(nrfjprog 和 JLINK 驱动的版本)

命令: nrfjprog -v

第六章：GPIO 输出驱动 LED

1. 学习目的

1. 掌握 LED 驱动电路的设计：控制方式、限流电阻的计算和确定。
2. 学习 nRF52840 GPIO 用作输出时相关寄存器的配置。
3. 掌握库函数中 GPIO 配置为输出的函数以及 GPIO 置位（输出高电平）、清除（输出低电平）和状态翻转函数的应用。
4. 编写驱动 LED 闪烁的程序，再此基础上，更进一步，编写流水灯的程序。

2. 硬件电路设计

LED(Light Emitting Diode)是发光二极管的简称，在很多设备上常用它来作为一种简单的人机接口，如网卡、路由器等通过 LED 向用户指示设备的不同工作状态。所以，我们习惯把这种用于指示状态的 LED 称为 LED 指示灯。

IK-52840DK 开发板上设计了 4 个 LED 指示灯，我们可以通过编程驱动 LED 指示灯点亮、熄灭、闪烁，从而达到状态指示的目的，LED 指示灯驱动电路如下图所示。

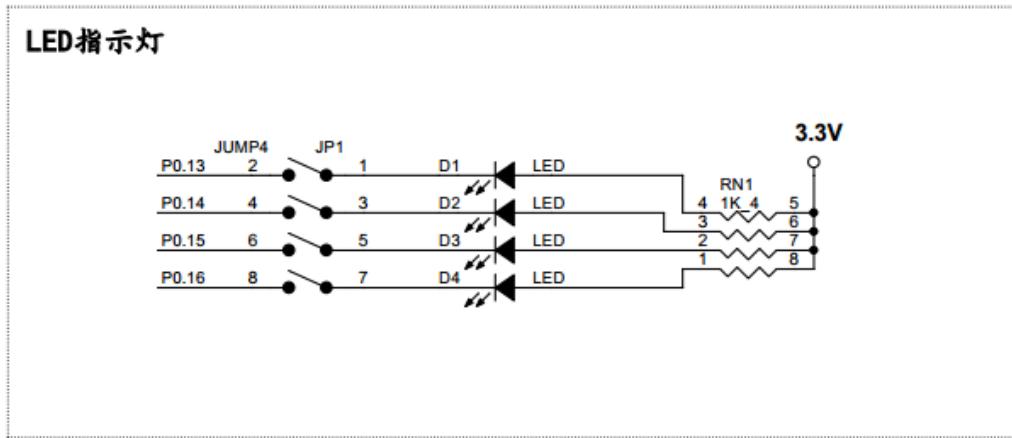


图 6-1: LED 指示灯驱动电路

4 个 LED 指示灯占用的 nRF52840 的引脚如下表：

表 6-1: LED 引脚分配

LED	颜色	引脚	说明
D1	蓝色	P0.13	独立 IO
D2	蓝色	P0.14	独立 IO
D3	蓝色	P0.15	独立 IO
D4	蓝色	P0.16	独立 IO

◆ 注：独立 IO 表示开发板没有其他的电路使用这个 IO。

LED 指示灯驱动电路是一个很常见、简单的电路，但它也是一个典型的单元电路，对于初学者来说，类似常用的典型电路必须要掌握，不但要知其然、还要知其所以然。

接下来，我们来分析一下这个简单的 LED 指示灯驱动电路。

LED 驱动电路设计的时候，需要我们考虑两个方面：控制方式和限流电阻的选取。

2.1. 控制方式

LED 指示灯控制方式分为高电平有效和低电平有效两种，高电平有效是单片机 IO 输出高电平时点亮 LED，低电平有效是单片机 IO 输出低电平时点亮 LED。

1. 低电平有效的控制方式

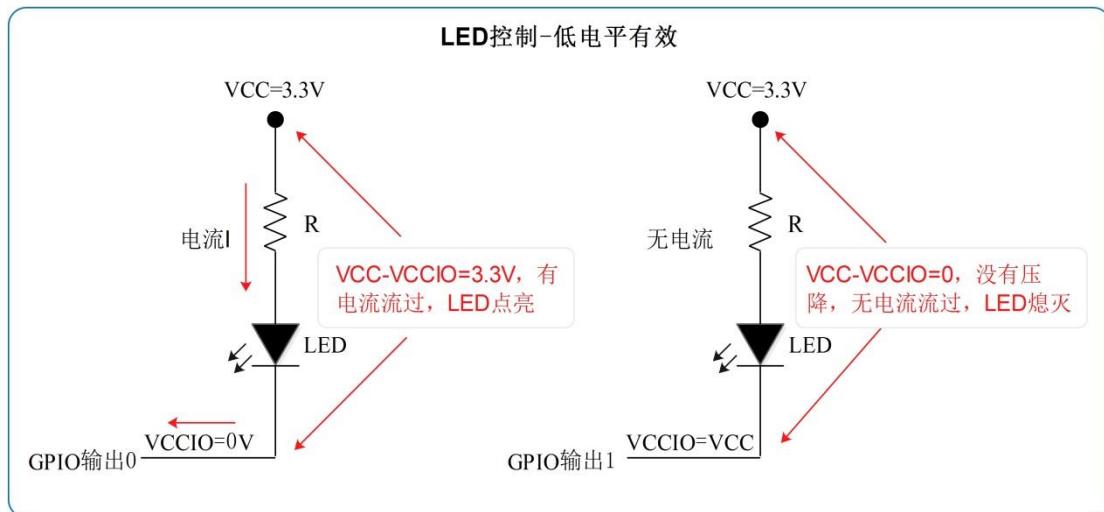


图 6-2: LED 控制-低电平有效原理

低电平有效控制方式中，当单片机的 GPIO 输出低电平(逻辑 0)的时候，LED 和电阻 R 上的压降等于 ($VCC-VCCIO = 3.3V$)，这时候，因为存在压降，同时，这个电路是闭合回环的，这就达到了电流产生的两个要素，LED 上会有电流流过，LED 被点亮。

当单片机的 GPIO 输出高电平(逻辑 1)的时候，LED 和电阻 R 上的压降等于 0V ($VCC-VCCIO = 0V$)，这时候，因为 LED 上没有压降，当然不会有电流流过，所以 LED 熄灭。

低电平有效控制方式中，电流经过限流电阻和 LED “灌入” GPIO，对于一般单片机来说，“灌电流”比较大，即低电平有效时驱动能力较强。

2. 高电平有效的控制方式

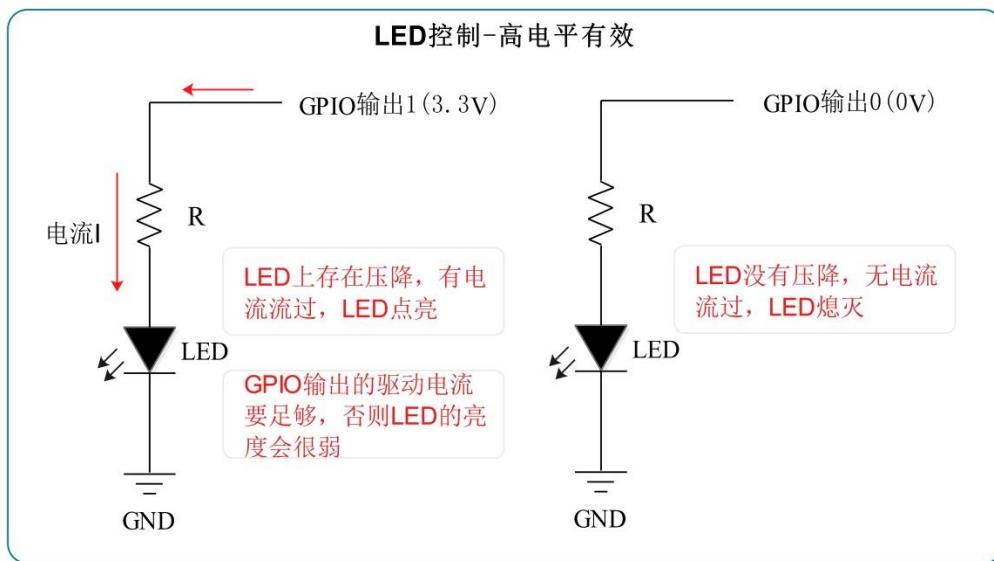


图 6-3: LED 控制-高电平有效原理

高电平有效控制方式中，由单片机的 GPIO 输出电流驱动 LED，当单片机的 GPIO 输出高电平(逻辑 1)的时候，LED 上存在压降，会有电流流过，这时 LED 被点亮，但要注意，单片机的 GPIO 要能提供足够的输出电流，否则，电流过小，会导致 LED 亮度很弱。

当单片机的 GPIO 输出地电平(逻辑 0)的时候，LED 和电阻 R 上的压降等于 0V，这时候，LED 上没有电流流过，LED 熄灭。

3. 选择哪种方式来控制 LED

大多数情况下，我们会选择使用低电平有效的控制方式，如开发板中的 LED 指示灯就是低电平有效。这是因为：单片机 IO 低电平时的灌入电流一般比高电平时的拉电流要大，能提供足够的电流驱动 LED，另外单片机上电或复位启动时，IO 口一般都是高阻输入，用低电平有效的控制方式可以确保 LED 在上电或复位启动时处于熄灭状态。

2.2. LED 限流电阻的选取

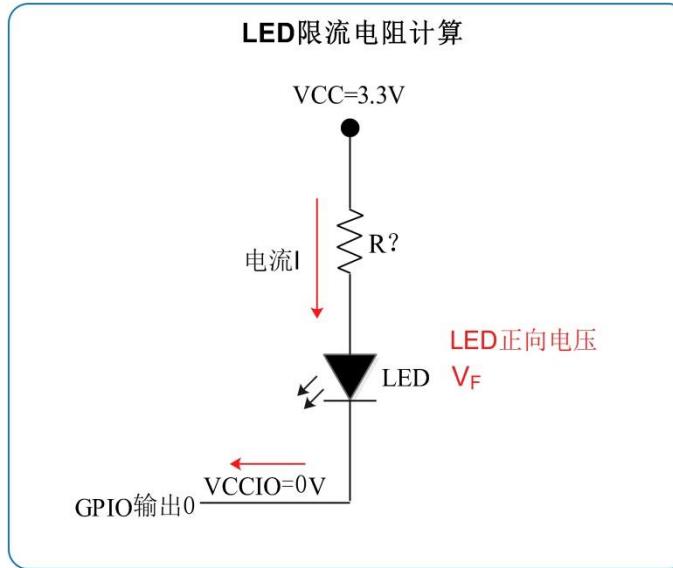


图 6-4: LED 限流电阻计算

由上图可以看出，LED 限流电阻的计算公式如下：

$$R = \frac{V_{CC} - V_F}{I} \Omega$$

其中，VCC=3.3V，V_F是 LED 的正向压降，LED 的数据手册都会给出正向电流为 20mA 时测试的 V_F的范围，下图是一款 0603 LED 的实物图和参数。在参数表中可以看到正向电流为 20 mA 时 V_F最小值是 1.6V，典型值是 2.0V，最大值是 2.6V。



0603 LED

参数名称 Parameter	符号 Symbol	条件 Condition	最小值 Min.	典型值 Typ.	最大值 Max.	单位 Unit
反向电流 Reverse Current	I _R	V _R =5V	-	-	10	μA
视角度 View Angle	2θ _{1/2}	-	-	130	-	deg.
正向电压 Forward Voltage	V _F		1.6	2.0	2.6	V
峰值波长 Peak Wavelength	λ _P	I _F =20mA 时 测试的V _F 范围		630		nm
主波长 Dominant Wavelength	λ _d		615	622	630	nm
半波宽度 Spectrum Radiation Bandwidth	Δλ		-	15	-	nm
光强 Luminous Intensity	I _V		80	120	220	med

图 6-5：LED 参数表

计算时 V_F的值可以用典型值来进行估算，对于电流，需要根据经验值和对 LED 亮度的要求相结合来确定，一般经验值是(2~5)mA，不过要注意，只要亮度符合自己的要求，电流低于 2mA 也没有任何问题。

电流为 2mA 时限流电阻值计算如下：

$$R = \frac{3.3 - 2.0}{0.002} \Omega = 650 \Omega$$

计算出的电阻是 650Ω，650Ω 不是一个常用的电阻值，所以我们需要选择一个电阻值为 650Ω 左右常用的电阻器，在 IK-52840DK 开发板上，我们选择的电阻值是最常用的 1K 的电阻器，选择限流电阻后，还需要实际测试观察 LED 的亮度是否符合自己的需求，经过实际测试观察，IK-52840DK 开发板上使用 1K 限流电阻时亮度符合我们的要求，由此，限流电阻的阻值确定为 1K。当然，如果我们觉得亮度不够，可以将电阻值适当减小一些，如使用 680Ω 或 510Ω 的电阻器作为限流电阻。

3. GPIO 输出驱动原理

3.1. 功能描述

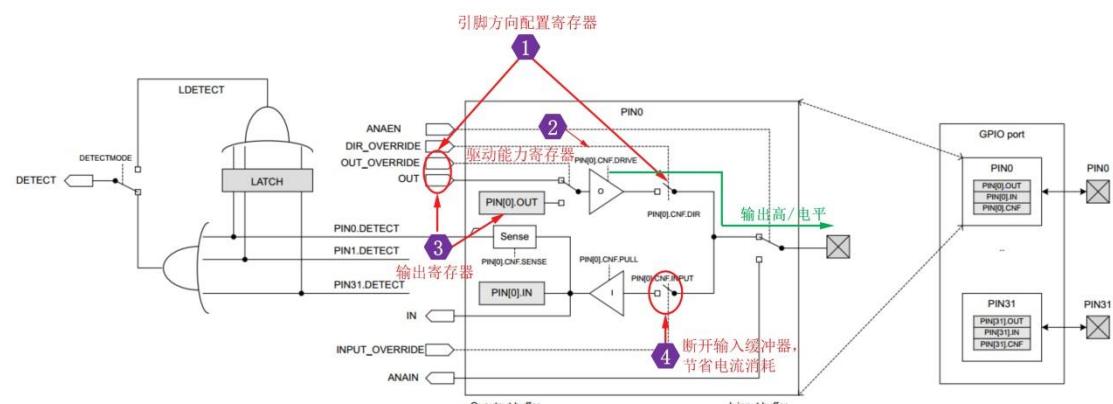
nRF52840 共有 48 个 GPIO，包含 P0 和 P1 端口，其中 P0 端口包含 P0.00~P0.31 共 32 个 GPIO，P1 端口包含 P1.00~P1.15 共 16 个 GPIO，每个 GPIO 均可以独立访问。

- 共 48 个 GPIO。
- 48 个 GPIO 中有 8 个 IO 可用作 SAADC、COMP 或 LPCOMP 的模拟输入通道。
- 可配置的输出驱动能力。
- 具有内部上拉和下拉电阻。
- 所有引脚的高电平或低电平触发均可唤醒系统。
- 所有的引脚都可以被 PPI 事件/任务系统使用。
- 所有的数字引脚都可以自由映射，这极大地增加了 PCB 的灵活性。
- SENSE 信号捕获的 GPIO 状态变化，可存储到 LATCH 寄存器。

nRF52840 的每个 GPIO 都都有一个引脚配置（PIN_CNF[n]，n=0...31）寄存器，每个引脚都可以通过自己的 PIN_CNF 独立配置，可配置的项目如下：

- 方向，即输入或输出。
- 输出驱动能力。
- 使能/禁用上拉和下拉电阻。
- 引脚感知功能（可以将系统从 system off 模式下唤醒）。
- 是否连接输入缓冲器。
- 模拟输入（针对特定的引脚）。

nRF52840 的 GPIO 的原理框图如下，从图中我们可以看到，当 GPIO 作为输出使用的时候，需要配置引脚方向寄存器、驱动能力寄存器，配置完成之后，通过写输出寄存器输出高/低电平（逻辑 1 和逻辑 0）。



引脚输出时涉及到的寄存器

- 1: 引脚方向配置寄存器，配置为输出。
- 2: 引脚驱动能力配置寄存器，配置引脚输出的驱动能力。
- 3: 引脚输入缓冲器，配置为断开输入缓冲器，以节省电流消耗，该寄存器初始状态是断开的。

图 6-6: GPIO 原理框图

引脚方向配置寄存器包含 *DIR* 和两个重写寄存器（ OVERRIDE）*DIRSET*、*DIRCLR*，写 *DIRSET* 和 *DIRCLR* 寄存器后，最终影响的是 *DIR* 寄存器中对应的位。如 *DIRSET* 寄存器的 0 位写入 1，那么 *DIR* 寄存器的 0 位会被重写为 1，如 *DIRCLR* 寄存器的 0 位写入 1，那么 *DIR* 寄存器的 0 位会被重写为 0。

3.2. 引脚的驱动能力

nRF52840 的 GPIO 的驱动能力是可以软件配置的，GPIO 输出高电平（逻辑 1）和低电平（逻辑 0）时可配置的驱动能力如下。

1. GPIO 输出高电平（逻辑 1）时的驱动能力

- 配置为标准驱动能力，VDD \geqslant 1.7V 时，最大 0.5mA。
- 配置为高驱动能力，VDD \geqslant 1.7V 时，最大 3mA。
- 配置为高驱动能力，VDD \geqslant 2.7V 时，最大 5mA。

2. GPIO 输出低电平（逻辑 0）时的驱动能力

- 配置为标准驱动能力，VDD \geqslant 1.7V 时，最大 0.5mA。
- 配置为高驱动能力，VDD \geqslant 1.7V 时，最大 3mA。
- 配置为高驱动能力，VDD \geqslant 2.7V 时，最大 5mA。

在我们设计的时候，需要注意虽然 GPIO 可配置驱动能力，但是不是能随意配置使用的，推荐的应用方式如下：芯片 GPIO 总电流（灌电流）不超过 15mA，拉电流不超过 14mA，如果只使用一个引脚，那么这个引脚可以使用 15mA 驱动电流。

4. 软件设计

4.1. GPIO 配置

nRF52840 的 P0 和 P1 端口的基址如下表所示。

表 6-2: SAADC 基址

外设名称	基址	描述
P0	0x50000000	GPIO P0.00~P0.31
P1	0x50000300	GPIO P1.00~P1.15

nRF52840 提供了 10 个用于操作 GPIO 的寄存器，如下表所示：

表 6-3: GPIO 相关寄存器

序号	寄存器名	读/写	功能描述
1	OUT	读/写	写 GPIO 端口，P0 端口对应 bit0~31 对应引脚 P0.00~P0.31，P1 端口对应 bit0~15 对应引脚 P1.00~P1.15。
2	OUTSET	读/写	引脚置位，P0 端口对应 bit0~31 对应引脚 P0.00~P0.31，P1 端口对应 bit0~15 对应引脚

			P1.00~P0.15。
3	OUTCLR	读/写	引脚清除, P0 端口对应 bit0~31 对应引脚 P0.00~P0.31, P1 端口对应 bit0~15 对应引脚 P1.00~P0.15。
4	IN	只读	读 GPIO 端口, P0 端口对应 bit0~31 对应引脚 P0.00~P0.31, P1 端口对应 bit0~15 对应引脚 P1.00~P0.15。
5	DIR	读/写	引脚方向配置, P0 端口对应 bit0~31 对应引脚 P0.00~P0.31, P1 端口对应 bit0~15 对应引脚 P1.00~P0.15。
6	DIRSET	读/写	引脚输出设置寄存器, 配置引脚为输出。
7	DIRCLR	读/写	引脚输出设置寄存器, 配置引脚为输入。
8	LATCH	读/写	引脚感知锁存寄存器。
9	DETECTMODE	读/写	DETECT 模式配置寄存器。
10	PIN_CNF[n] n=0~31	读/写	引脚配置寄存器。P0 端口对应 bit0~31 对应引脚 P0.00~P0.31, P1 端口对应 bit0~15 对应引脚 P1.00~P0.15。

1. 配置 IO 为输出

配置 IO 为输出的时候, 相关的寄存器有 DIR、DIRSET 和 PIN_CNF[n] n=0~31 寄存器, 这 3 个寄存器都是 32 位的, 只能按字访问, 其中 DIR 和 DIRSET 寄存器仅用来配置引脚方向。PIN_CNF[n] n=0~31 寄存器中的位 0 用来配置方向, 其它的位用来配置驱动能力等, 寄存器详细描述如下表所示。

- **DIR:** 引脚方向配置寄存器
- P0 端口: 可配置的 IO 是 P0.00~P0.31。
- P1 端口: 可配置的 IO 是 P1.00~P1.15。

表 6-4: DIR 寄存器

位	Field	RW	复位值	描述
位 0	PIN0	读/写	0	0: 设置引脚 Px.00 为输入。 1: 设置引脚 Px.00 为输出。
位 1	PIN1	读/写	0	0: 设置引脚 Px.01 为输入。 1: 设置引脚 Px.01 为输出。
...
位 31	PIN31	读/写	0	0: 设置引脚 Px.31 为输入。 1: 设置引脚 Px.31 为输出。

■ DIRSET: 引脚方向 (输出) 配置寄存器

- P0 端口: 可配置的 IO 是 P0.00~P0.31。
- P1 端口: 可配置的 IO 是 P1.00~P1.15。

表 6-5: DIRSET 寄存器

位	Field	RW	复位值	描述
位 0	PIN0	读/写	0	读, 0: 引脚 Px.00 方向为输入。 读, 1: 引脚 Px.00 方向输出。 写, 1: 设置引脚 Px.00 为输入, 写 0 无效。
位 1	PIN1	读/写	0	读, 0: 引脚 Px.01 方向为输入。 读, 1: 引脚 Px.01 方向输出。 写, 1: 设置引脚 Px.01 为输出, 写 0 无效。
...
位 31	PIN31	读/写	0	读, 0: 引脚 Px.31 方向为输入。 读, 1: 引脚 Px.31 方向输出。 写, 1: 设置引脚 Px.31 为输出, 写 0 无效。

■ PIN_CNF[n] (n=0~31): 引脚配置寄存器

- P0 端口: 可配置的 IO 是 P0.00~P0.31。
- P1 端口: 可配置的 IO 是 P1.00~P1.15。

表 6-6: PIN_CNF[n] (n=0~31)寄存器

位	Field	RW	复位值	描述
位 0	DIR	读/写	0	引脚方向配置, 等同于 DIR 寄存器。 0: 设置引脚为输入。 1: 设置引脚为输出。
位 1	INPUT	读/写	1	连接或断开输入缓冲器。 0: 连接输入缓冲器。 1: 断开输入缓冲器。
位 3~位 2	PULL	读/写	00	上拉/下拉电阻配置 0: 关闭上拉/下拉。 1: 开启下拉电阻。 3: 开启上拉电阻。
位 10~位 8	DRIVE	读/写	000	驱动能力配置 S0S1: 逻辑“0”标准驱动能力, 逻辑“1”标准驱

				动能力。
		1	H0S1:	逻辑“0”高驱动能力，逻辑“1”标准驱动能力。
		2	S0H1:	逻辑“0”标准驱动能力，逻辑“1”高驱动能力。
		3	H0H1:	逻辑“0”高驱动能力，逻辑“1”高驱动能力。
		4	D0S1:	逻辑“0”断开，逻辑“1”标准驱动能力。
		5	D0H1:	逻辑“0”断开，逻辑“1”高驱动能力。
		6	S0D1:	逻辑“0”标准驱动能力，逻辑“1”断开。
		7	H0D1:	逻辑“0”高驱动能力，逻辑“1”断开。
位 17~	SENSE	读/写	00	引脚感知机制
位 16				0: 关闭引脚感知。 2: 高电平感知。 3: 低电平感知。

由上表中的寄存器描述可以看出，配置一个 GPIO 为输出有多种方式：可以通过向 DIR 寄存器相应的位写 1 配置对应的引脚为输出，也可以通过向 DIRSET 寄存器相应的位写 1 配置对应的引脚为输出，还可以通过向 PIN_CNF[n]寄存器的位 0 写 1 配置对应的引脚为输出。

一般在配置引脚的时候，直接通过 PIN_CNF[n]寄存器来配置方向，因为配置引脚除了配置方向之外，PIN_CNF[n]寄存器中其它的位也要配置，所以直接配置 PIN_CNF[n]寄存器比较方便。如果我们仅仅是配置引脚方向（配置为输出）的话，通常使用 DIRSET 寄存器，这是因为 DIRSET 寄存器的各个位写 1 即配置对应的引脚为输出，而写零无效，这样在配置某个引脚为输出的时候直接向 DIRSET 对应的位写 1 即可，而不用担心会影响到其它其它引脚的方向配置（如果使用 DIR 寄存器，为了不影响其它引脚方向配置，就需要先读出端口方向配置，然后修改对应的位再写回去）。

2. GPIO 输出高/电平

GPIO 配置为输出后，就可以通过写寄存器进行控制。用于控制 GPIO 输出高/低电平的寄存器有 3 个：OUT、OUTSET 和 OUTCLR，这 3 个寄存器都是 32 位的，只能按字访问。

- OUT: 写 GPIO 端口寄存器
- P0 端口: 可配置的 IO 是 P0.00~P0.31。
- P1 端口: 可配置的 IO 是 P1.00~P1.15。

表 6-7: OUT 寄存器

位	Field	RW	复位值	描述
---	-------	----	-----	----

位 0	PIN0	读/写	0	0: 引脚 Px.00 输出低电平, 逻辑“0”。 1: 引脚 Px.00 输出高电平, 逻辑“1”。
位 1	PIN1	读/写	0	0: 引脚 Px.01 输出低电平, 逻辑“0”。 1: 引脚 Px.01 输出高电平, 逻辑“1”。
...
位 31	PIN31	读/写	0	0: 引脚 Px.31 输出低电平, 逻辑“0”。 1: 引脚 Px.31 输出高电平, 逻辑“1”。

- OUTSET: 引脚置位寄存器
- P0 端口: 可配置的 IO 是 P0.00~P0.31。
- P1 端口: 可配置的 IO 是 P1.00~P1.15。

表 6-8: OUTSET 寄存器

位	Field	RW	复位值	描述
位 0	PIN0	读/写	0	读, 0: 引脚 Px.00 输出的是低电平, 逻辑“0”。 读, 1: 引脚 Px.00 输出的是高电平, 逻辑“1”。 写, 1: 引脚 Px.00 输出高电平, 逻辑“1”, 写 0 无效。
位 1	PIN1	读/写	0	读, 0: 引脚 Px.01 输出的是低电平, 逻辑“0”。 读, 1: 引脚 Px.01 输出的是高电平, 逻辑“1”。 写, 1: 引脚 Px.01 输出高电平, 逻辑“1”, 写 0 无效。
...
位 31	PIN31	读/写	0	读, 0: 引脚 Px.31 输出的是低电平, 逻辑“0”。 读, 1: 引脚 Px.31 输出的是高电平, 逻辑“1”。 写, 1: 引脚 Px.31 输出高电平, 逻辑“1”, 写 0 无效。

- OUTCLR: 引脚清除寄存器
- P0 端口: 可配置的 IO 是 P0.00~P0.31。
- P1 端口: 可配置的 IO 是 P1.00~P1.15。

表 6-9: OUTCLR 寄存器

位	Field	RW	复位值	描述
位 0	PIN0	读/写	0	读, 0: 引脚 Px.00 输出的是低电平, 逻辑“0”。

			读, 1: 引脚 Px.00 输出的是高电平, 逻辑“1”。 写, 1: 引脚 Px.00 输出低电平, 逻辑“0”, 写 0 无效。
位 1	PIN1	读/写	0 读, 0: 引脚 Px.01 输出的是低电平, 逻辑“0”。 读, 1: 引脚 Px.01 输出的是高电平, 逻辑“1”。 写, 1: 引脚 Px.01 输出低电平, 逻辑“0”, 写 0 无效。
...
位 31	PIN31	读/写	0 读, 0: 引脚 Px.31 输出的是低电平, 逻辑“0”。 读, 1: 引脚 Px.31 输出的是高电平, 逻辑“1”。 写, 1: 引脚 Px.31 输出低电平, 逻辑“0”, 写 0 无效。

4.2. 库函数的应用

对于外设的寄存器，我们需要掌握它的原理，这能帮助我们更深入地掌握单片机外设。但是实际编程的时候，大可不必使用直接操作寄存器的方式来编程，这会降低我们编程的效率。相对来说，使用库函数会方便很多，库函数按照功能的实现封装了对寄存器的操作，避免了软件开发人员直接面对繁琐的寄存器，让软件开发人员可以将更多的精力放在应用开发上。

接下来，我们用库函数来实现 GPIO 的初始化和输出高/低电平驱动 LED 指示灯。

1. 定义引脚

nRF52840 共有 48 个 GPIO，分布于 P0 和 P1 端口，库函数中使用“0~47”表示 48 个 GPIO，但是这样看起来很不直观，所以库函数中专门提供了一个带参数的宏 NRF_GPIO_PIN_MAP 用来定义 GPIO。该宏有 2 个输入参数，其中“port”是 GPIO 端口 P0 或者 P1，“pin”是 GPIO 端口中具体的引脚。

```
#define NRF_GPIO_PIN_MAP(port, pin) (((port) << 5) | ((pin) & 0x1F))
```

使用 NRF_GPIO_PIN_MAP 定义引脚就很直观了，如：

- NRF_GPIO_PIN_MAP(0, 10): 端口 0 的 pin10，即 P0.10。
- NRF_GPIO_PIN_MAP(1, 10): 端口 1 的 pin10，即 P1.10。

2. 配置 GPIO 为输出

Nordic 的库中用于初始化 GPIO 为输出的库函数是 nrf_gpio_cfg_output() 和 nrf_gpio_range_cfg_output()，nrf_gpio_cfg_output() 用于配置单个的 GPIO 为输出，nrf_gpio_range_cfg_output() 用于配置连续编号的 GPIO 为输出，函数原型如下表所示：

表 6-10: nrf_gpio_cfg_output()函数

函数原型	<code>_STATIC_INLINE void nrf_gpio_cfg_output((uint32_t pin_number)</code>
函数功能	配置指定的 GPIO 为输出，输出 IO 配置为标准驱动能力。
参数	[in] pin_number: 引脚编号, 0~47, 其中: 0~31 对应 P0.00~P0.31, 32~47 对应 P1.00~P1.15。
返回值	无。

表 6-11: nrf_gpio_range_cfg_output()函数

函数原型	<code>_STATIC_INLINE void nrf_gpio_range_cfg_output((uint32_t pin_range_start, uint32_t pin_range_end)</code>
函数功能	配置连续编号的 GPIO 为输出，输出 IO 配置为标准驱动能力。注意引脚编号必须是连续的才可以使用该函数进行配置，如配置 P0.17~P0.20 为输出。
参数	[in] pin_range_start: 起始引脚编号。 [in] pin_range_end: 结束引脚编号。
返回值	无。

由上表可以看出，配置 GPIO 为输出的时候，只需要调用库函数，并将需要配置的引脚的编号作为函数的输入参数即可完成配置。

■ 应用示例：配置 P0.13 为输出

`nrf_gpio_cfg_output(13);`

函数中可以直接写“0~47”，更直观的是这样来写：

`nrf_gpio_cfg_output(NRF_GPIO_PIN_MAP(0, 13));`

■ 应用示例：配置 P0.13~P0.162 四个连续的 GPIO 为输出

可以使用 `nrf_gpio_cfg_output()` 函数逐个配置如下：

`nrf_gpio_cfg_output(NRF_GPIO_PIN_MAP(0, 13));`

`nrf_gpio_cfg_output(NRF_GPIO_PIN_MAP(0, 14));`

`nrf_gpio_cfg_output(NRF_GPIO_PIN_MAP(0, 15));`

`nrf_gpio_cfg_output(NRF_GPIO_PIN_MAP(0, 16));`

也可以使用 `nrf_gpio_range_cfg_output()` 配置如下：

```
nrf_gpio_range_cfg_output(NRF_GPIO_PIN_MAP(0, 13), NRF_GPIO_PIN_MAP(0, 16));
```

细心的读者在这里可能会注意到，GPIO 不是可以配置驱动能力和其它一些参数吗，这些参数在哪儿配置了？

我们打开输出配置函数，可以看到，在函数中使用了默认的标准参数配置了 GPIO 的其它项目。

代码清单：nrf_gpio_cfg_output 函数

```
1. __STATIC_INLINE void nrf_gpio_cfg_output(uint32_t pin_number)
2. {
3.     nrf_gpio_cfg(
4.         pin_number,
5.         NRF_GPIO_PIN_DIR_OUTPUT,
6.         NRF_GPIO_PIN_INPUT_DISCONNECT,
7.         NRF_GPIO_PIN_NOPULL,
8.         NRF_GPIO_PIN_S0S1,
9.         NRF_GPIO_PIN_NOSENSE);
10. }
```

上述代码执行后，对应的 GPIO 被配置为：

- 方向配置为输出。
- 输入缓冲器断开。
- 无上拉/下拉。
- 引脚电平感知关闭。
- 逻辑“0”标准驱动能力，逻辑“1”标准驱动能力。

对于 GPIO 配置为输出来说，绝大多数情况下，我们都会使用如上文所述默认的标准配置参数，如果对某个引脚不使用标准参数进行配置，可以使用 nrf_gpio_cfg() 函数来配置。

表 6-12: nrf_gpio_cfg() 函数

函数原型	<pre>_STATIC_INLINE void nrf_gpio_cfg (uint32_t pin_number, nrf_gpio_pin_dir_t dir, nrf_gpio_pin_input_t input, nrf_gpio_pin_pull_t pull, nrf_gpio_pin_drive_t drive, nrf_gpio_pin_sense_t sense)</pre>
函数功能	配置连续编号的 GPIO 为输出。注意引脚编号必须是连续的才可以使用该函数进行配置，如配置 P0.17~P0.20 为输出。

参 数	[in] pin_number: 引脚编号。 [in] dir: 方向配置。 [in] input: 是否连接输入缓冲器。 [in] pull: 上拉/下拉电阻配置。 [in] drive: 驱动能力配置。 [in] sense: 引脚感知配置。
返回值	无。

❖ 注意: nrf_gpio_cfg()函数参数较多, 使用起来比较麻烦, 如无必要, 强烈建议不使用该函数配置 IO。对于 GPIO 的特殊用途下如何配置, 在后续的章节中都会有详细描述。

3. GPIO 输出高/低电平(置位/清零)

Nordic 的库中用于 GPIO 输出的库函数常用的有:

- nrf_gpio_pin_set (): 置位指定的 GPIO, 即驱动指定编号的 GPIO 输出高电平(逻辑 1)。
- nrf_gpio_pin_clear (): 清除指定的 GPIO, 即驱动指定编号的 GPIO 输出低电平(逻辑 0)。
- nrf_gpio_pin_toggle (): 翻转指定编号的 GPIO 输出状态, 调用该函数后, 如原先输出的是高电平, 则翻转状态输出低电平, 反之也然。

函数原型如下表所示:

表 6-13: nrf_gpio_pin_set ()函数

函数原型	<code>_STATIC_INLINE void nrf_gpio_pin_set (uint32_t pin_number)</code>
函数功能	驱动指定的 GPIO 输出高电平。
参 数	[in] pin_number: 引脚编号, 0~31。其中: 0~31 对应 P0.00~P0.31,32~47 对应 P1.00~P1.15。
返回值	无。

表 6-14: nrf_gpio_pin_clear ()函数

函数原型	<code>_STATIC_INLINE void nrf_gpio_pin_clear (uint32_t pin_number)</code>
函数功能	驱动指定的 GPIO 输出低电平。
参 数	[in] pin_number: 引脚编号, 0~31。其中: 0~31 对应 P0.00~P0.31,32~47 对应 P1.00~P1.15。

返回值	无。
-----	----

表 6-15: nrf_gpio_pin_toggle()函数

函数原型	<code>_STATIC_INLINE void nrf_gpio_pin_toggle (uint32_t pin_number)</code>
函数功能	翻转指定的 GPIO 的输出状态。即：如原先输出的是高电平，则翻转状态输出低电平，原先输出的是低电平，则翻转状态输出高电平。
参数	[in] pin_number: 引脚编号，0~31。其中：0~31 对应 P0.00~P0.31,32~47 对应 P1.00~P1.15。
返回值	无。

■ 应用示例：驱动 P0.13 输出高电平

```
nrf_gpio_pin_set(13);
```

函数中可以直接写“0~47”，更直观的是这样来写：

```
nrf_gpio_pin_set(NRF_GPIO_PIN_MAP(0, 13));
```

■ 应用示例：驱动 P0.17 输出低电平

```
nrf_gpio_pin_clear(13);
```

函数中可以直接写“0~47”，更直观的是这样来写：

```
nrf_gpio_pin_clear(NRF_GPIO_PIN_MAP(0, 13));
```

■ 应用示例：翻转 P0.13 的输出状态

```
nrf_gpio_pin_toggle(13);
```

函数中可以直接写“0~47”，更直观的是这样来写：

```
nrf_gpio_pin_toggle(NRF_GPIO_PIN_MAP(0, 13));
```

4.3. 驱动 LED 闪烁实验

❖ 注：本节对应的实验源码是：“实验 6-1：GPIO 输出驱动 led 闪烁”。

4.3.1. 代码编写

知道了如何通过库函数驱动 GPIO 输出高低电平，驱动 LED 指示灯闪烁就很简单了，只需驱动 GPIO 以一定的间隔输出高低电平即可达到驱动 LED 指示灯闪烁的目的。编写代码的时候可以使用：反复“输出高电平→延时→输出低电平→延时”的方式实现，也可以通过：反复“翻转输出状态→延时”的方式实现。

代码清单：led 指示灯闪烁

```
1. /*****
2. * 描述 : main 函数
3. * 入参 : 无
4. * 返回值 : int 类型
5. *****/
6. int main(void)
7. {
8.     //配置用于驱动 LED 指示灯 D1 的管脚，即配置 P0.13 为输出
9.     nrf_gpio_cfg_output(LED_1);
10.    //LED 指示灯 D1 初始状态设置为熄灭，即引脚 P0.13 为输出高电平
11.    nrf_gpio_pin_set(LED_1);
12.
13.    while(true)
14.    {
15.        //P0.13 输出高电平，熄灭指示灯 D1
16.        nrf_gpio_pin_set(LED_1);
17.        //软件延时 200ms
18.        nrf_delay_ms(200);
19.        //P0.13 输出低电平，点亮指示灯 D1
20.        nrf_gpio_pin_clear(LED_1);
21.        //软件延时 200ms
22.        nrf_delay_ms(200);
23.
24.        //以下是用 nrf_gpio_pin_toggle 函数实现驱动 led 闪烁
25.        //翻转引脚 P0.13 的输出状态，即翻转指示灯 D1 的状态
26.        //nrf_gpio_pin_toggle(LED_1);
27.        //软件延时 200ms
28.        //nrf_delay_ms(200);
29.    }
30. }
```

4.3.2. 硬件连接

本实验需要使用 P0.13 驱动 LED 指示灯 D1，按照下图所示用跳线帽短接 P0.13。

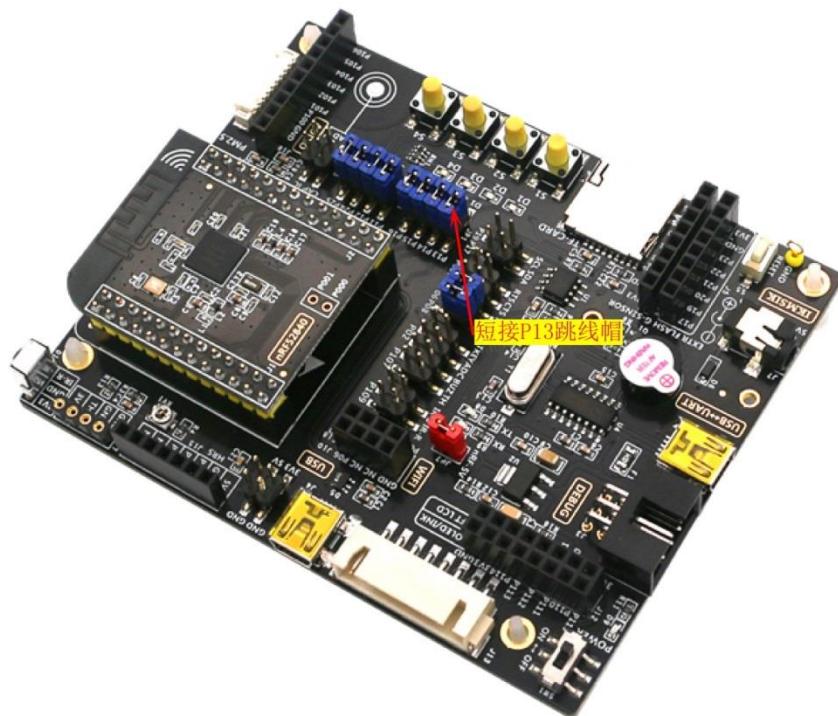


图 6-7: 开发板跳线帽短接 P0.13

4.3.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 6-1: GPIO 输出驱动 led 闪烁”，将解压后得到的文件夹“led_blinky”拷贝到合适的目录，如“D\ nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\led_blinky\project\mdk5”目录下的工程“led_blinky.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，开发板上的 D1 指示灯以 200ms 的间隔闪烁。

4.4. 流水灯实验

◆ 注：本节对应的实验源码是：“实验 6-2：流水灯(一般实现方式)”和“实验 6-3：流水灯(BSP 实现方式)”

4.4.1. 代码编写

流水灯就是按照一定的时间间隔有规律的轮流点亮 LED 指示灯，也就是 4 个 LED 对应

的引脚轮流输出高低电平，并加上一段时间的延时。但是我们要注意到一点：按照一般的方法是点亮一个 LED→延时→熄灭 LED→点亮下一个 LED…，这样虽然实现了功能，但是代码比较臃肿，尤其是 LED 数量比较多的时候。这时，我们可以定义一个数组，以 LED1~LED4 作为数组的元素，这样就可用 for 循环来操作 LED，编程上会方便很多。“boards.c”文件中已经使用这种方式写好了函数，使用的时候，我们直接调用即可。

程序清单：流水灯。一般的驱动方式

```
1.  ****
2.  * 描  述 : main 函数
3.  * 入  参 : 无
4.  * 返回值 : int 类型
5.  ****
6.  int main(void)
7.  {
8.      uint8_t i;
9.
10.     //配置用于驱动 LED 指示灯 D1 D2 D3 D4 的引脚脚，即配置 P0.13~P0.16 为输出
11.     nrf_gpio_range_cfg_output(LED_START, LED_STOP);
12.     //4 个 LED 初始状态设置为熄灭，即 P0.13~P0.16 输出高电平
13.     nrf_gpio_pin_set(LED_1);
14.     nrf_gpio_pin_set(LED_2);
15.     nrf_gpio_pin_set(LED_3);
16.     nrf_gpio_pin_set(LED_4);
17.
18.     while(true)
19.     {
20.         for(i=0;i<6;i++)
21.         {
22.             //指示灯 D1 状态翻转
23.             nrf_gpio_pin_toggle(LED_1);
24.             //软件延时 150ms
25.             nrf_delay_ms(150);
26.             nrf_gpio_pin_toggle(LED_2);
27.             nrf_delay_ms(150);
28.             nrf_gpio_pin_toggle(LED_3);
29.             nrf_delay_ms(150);
30.             nrf_gpio_pin_toggle(LED_4);
31.             nrf_delay_ms(150);
32.         }
33.
34.         //所有 LED 同时闪烁 2 次
35.         for(i=0;i<2;i++)
```

```
36.     {
37.         //P0.13 输出高电平, D1 熄灭
38.         nrf_gpio_pin_set(LED_1);
39.         nrf_gpio_pin_set(LED_2);
40.         nrf_gpio_pin_set(LED_3);
41.         nrf_gpio_pin_set(LED_4);
42.         //软件延时 150ms
43.         nrf_delay_ms(150);
44.         //P0.13 输出低电平, D1 点亮
45.         nrf_gpio_pin_clear(LED_1);
46.         nrf_gpio_pin_clear(LED_2);
47.         nrf_gpio_pin_clear(LED_3);
48.         nrf_gpio_pin_clear(LED_4);
49.         nrf_delay_ms(150);
50.     }
51.     //熄灭所有 LED
52.     nrf_gpio_pin_set(LED_1);
53.     nrf_gpio_pin_set(LED_2);
54.     nrf_gpio_pin_set(LED_3);
55.     nrf_gpio_pin_set(LED_4);
56.     //软件延时 500ms
57.     nrf_delay_ms(500);
58. }
59. }
```

“boards.c”文件中的 bsp 函数将 LED 指示灯按照数组的方式来组织，这样通过数组下标来访问各个 LED，对于稍复杂的应用操作起来会方便很多。

程序清单：定义 LED 数组

```
1. //LED 所用引脚宏定义
2. #define LEDS_NUMBER      4
3.
4. #define LED_1             NRF_GPIO_PIN_MAP(0,13)
5. #define LED_2             NRF_GPIO_PIN_MAP(0,14)
6. #define LED_3             NRF_GPIO_PIN_MAP(0,15)
7. #define LED_4             NRF_GPIO_PIN_MAP(0,16)
8. #define LED_START          LED_1
9. #define LED_STOP           LED_4
10.
11. #define LEDS_ACTIVE_STATE 0
12.
13. #define LEDS_LIST { LED_1, LED_2, LED_3, LED_4 }
```

//定义一个 LED 数组变量，之后即可通过数组下标操作 LED

1. static const uint8_t m_board_led_list[LEDS_NUMBER] = LEDS_LIST;

程序清单：流水灯。BSP 函数以数组的方式驱动

```
1. /*****
2. * 描述 : main 函数
3. * 入参 : 无
4. * 返回值 : int 类型
5. *****/
6. int main(void)
7. {
8.     uint8_t i;
9.
10.    //配置用于驱动 LED 指示灯 D1 D2 D3 D4 的引脚脚，即配置 P0.13~P0.16 为输出，并将 LED
11.    //的初始状态设置为熄灭
12.    bsp_board_init(BSP_INIT_LEDS);
13.
14.    while(true)
15.    {
16.        //使用 BSP 函数翻转 LED 指示灯状态
17.        for(i=0;i<6;i++)
18.        {
19.            for(int j = 0; j < LEDS_NUMBER; j++)
20.            {
21.                //翻转 LED 状态
22.                bsp_board_led_invert(j);
23.                //软件延时 150ms
24.                nrf_delay_ms(150);
25.            }
26.        }
27.
28.        //所有 LED 同时闪烁 2 次
29.        for(i=0;i<2;i++)
30.        {
31.            //P0.17 输出高电平，D1 熄灭
32.            bsp_board_leds_on();
33.            //软件延时 150ms
34.            nrf_delay_ms(150);
35.            bsp_board_leds_off();
36.            //软件延时 150ms
37.            nrf_delay_ms(150);
```

```
38.     }
39.     //软件延时 500ms
40.     nrf_delay_ms(500);
41. }
42. }
```

4.4.2. 硬件连接

本实验需要使用 P0.13 P0.14 P0.15 和 P0.16 驱动 LED 指示灯 D1 D2 D3 D4，按照下图所示用跳线帽短接 P0.13~P0.16。

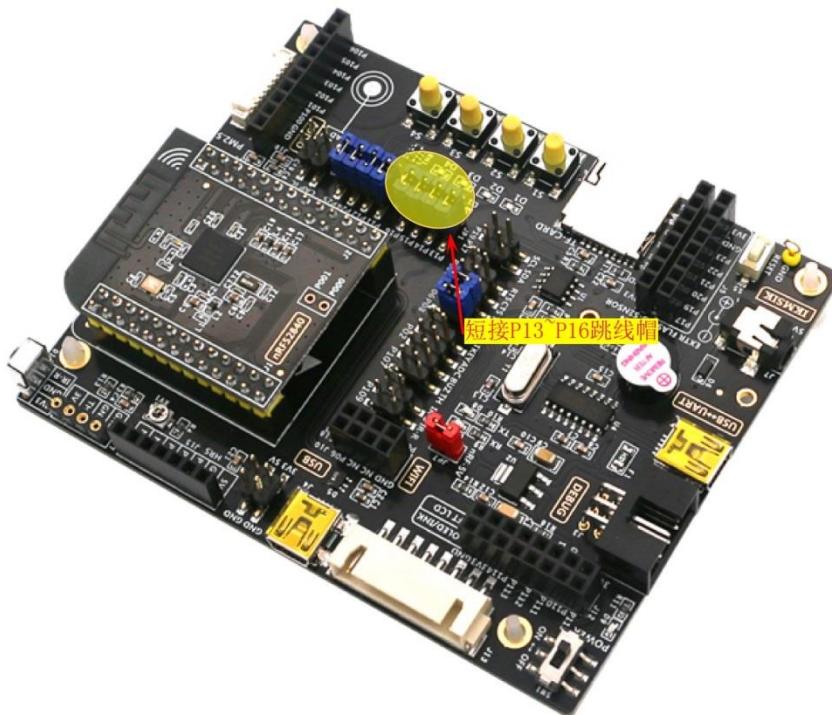


图 6-8：开发板跳线帽短接 P0.17~P0.20

4.4.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 6-2: 流水灯（一般实现方式）”（BSP 实现方式的试验解压“实验 6-3: 流水灯(BSP 实现方式)”), 将解压后得到的文件夹“leds_blinky”拷贝到合适的目录, 如“D\nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\leds_blinky\project\mdk5”目录下的工程“leds_blinky.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏, 观察编译的结果, 如果有错误, 修改程序, 直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真, 点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后, 开发板上的 4 个 LED 指示灯按照一定规律闪烁。

第七章：GPIO 输出驱动有源蜂鸣器

1. 学习目的

- 掌握有源蜂鸣器驱动电路的设计：控制方式、限流电阻的计算和确定。
- 了解有源蜂鸣器的特性以及和无源蜂鸣器的区别。

2. 硬件电路设计

开发板上使用的蜂鸣器是 3V 有源蜂鸣器，驱动电路如下图所示。有源蜂鸣器和无源蜂鸣器区别如下：

- 有源蜂鸣器：有源蜂鸣器内部带震荡源，所以只要一通电就会鸣响。
- 无源蜂鸣器：内部不带震荡源，用直流信号无法令其鸣响。必须用频率信号去驱动它。

所以，对于有源蜂鸣器来说，只要通电就会鸣响。为了实现控制有源蜂鸣器的鸣响，我们的电路中就需要一个“开关”来实现在有源蜂鸣器接通或断开电源。在开发板上，是使用 PNP 三极管 9013 来实现“开关”的功能的，通过 nRF52840 的引脚 P1.07 输出高低电平控制“开关”的接通和断开，即：

- 当 P1.07 输出逻辑 0 即低电平时，9013 基极电压约为 0V，基极没有电流，因此集电极也没有电流流过，三极管处于截止状态，即蜂鸣器和 GND 之间“断开”，蜂鸣器不鸣响。
- 当 P1.07 输出逻辑 1，即高电平时，9013 饱和导通，即蜂鸣器和 GND 之间“接通”，蜂鸣器鸣响。

电路中的 R6 是为了保证三极管可靠的截止。

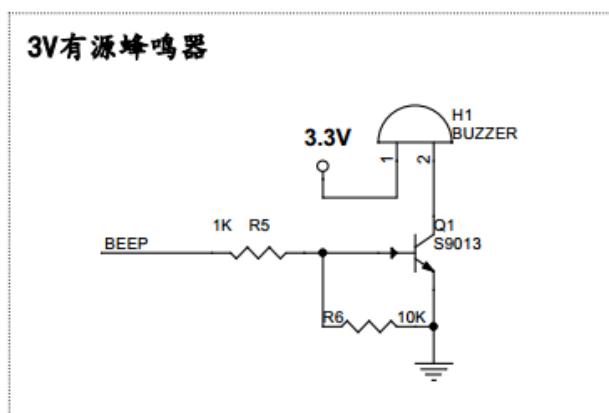


图 7-1：有源蜂鸣器驱动电路

有源蜂鸣器占用的 nRF52840 的引脚如下表：

表 7-1：有源蜂鸣器引脚分配

名称	引脚	说明
有源蜂鸣器	P0.04	和 DHT11 温湿度传感器检测公用

3. 软件设计

3.1. GPIO 配置

本实验中只需将 nRF52840 的引脚 P1.07 配置为输出，并输出高低电平即可，原理和 LED 试验一样，参见“第 6 章：GPIO 输出驱动 LED”中的描述即可。

3.2. 驱动蜂鸣器鸣响实验

◆ 注：本节对应的实验源码是：“实验 7-1：GPIO 输出驱动蜂鸣器”。

3.2.1. 代码编写

首先设置 GPIO 的输出电平（设置为 3.3V），然后使用库函数“`nrf_gpio_cfg_output()`”配置 P1.07 为输出（标准驱动能力），之后使用库函数“`nrf_gpio_pin_clear()`”设置 P1.07 输出低电平，以保证起始时蜂鸣器驱动电路三极管截止，即蜂鸣器不鸣响。

P1.07 配置完成后，主循环中每隔 500ms 翻转一次 P1.07 的状态，从而实现每隔 500ms 驱动蜂鸣器鸣响一次，鸣响时间也是 500ms。代码清单如下：

代码清单：驱动蜂鸣器鸣响

```
1. //定义驱动蜂鸣器的引脚 P1.07
2. #define BEEP_PIN NRF_GPIO_PIN_MAP(1,7)
3.
4. ****
5. * 描 述 : main 函数
6. * 入 参 : 无
7. * 返回值 : int 类型
8. ****
9. int main(void)
10. {
11.     //设置 GPIO 输出电压为 3.3V
12.     gpio_output_voltage_setup_3v3();
13.
14.     //配置用于驱动蜂鸣器的引脚，即配置 P1.07 为输出
15.     nrf_gpio_cfg_output(BEEP_PIN);
16.     //P1.07 输出低电平，即蜂鸣器停止鸣响
17.     nrf_gpio_pin_clear(BEEP_PIN);
18.
19.     while(true)
20.     {
21.         //翻转 P1.07 状态
```

```
22.     nrf_gpio_pin_toggle(BEEP_PIN);  
23.     //软件延时 500ms  
24.     nrf_delay_ms(500);  
25. }  
26. }
```

3.2.2. 硬件连接

本实验需要使用 P1.07 驱动 LED 指示灯 D1，按照下图所示用跳线帽短接 P0.13。

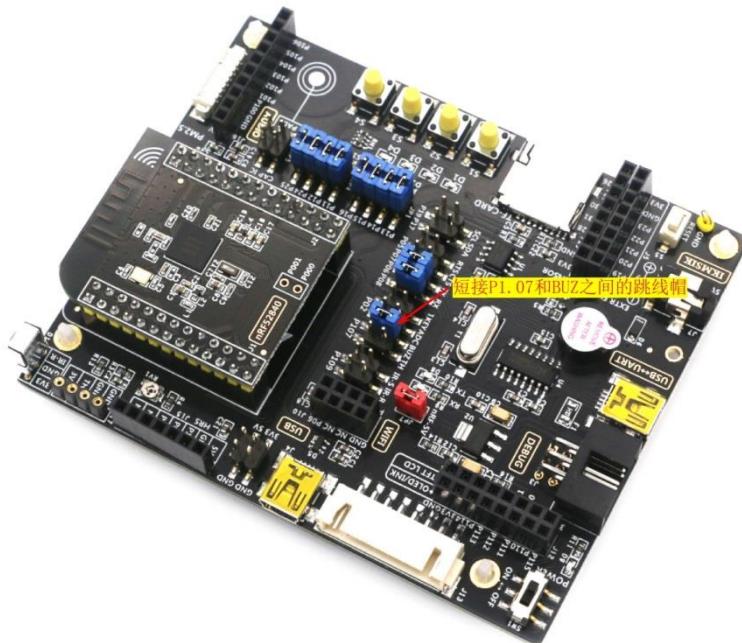


图 7-2：开发板跳线帽短接 P1.07

3.2.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 7-1: GPIO 输出驱动蜂鸣器”，将解压后得到的文件夹“beep”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\beep\project\mdk5”目录下的工程“beep.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，开发板上的蜂鸣器以 500ms 的间隔鸣响。

第 8 章：GPIO 输入按键检测

1. 学习目的

1. 学习轻触按键和触摸按键检测电路的设计以及硬件原理和区别。
2. 学习 nRF52840 GPIO 用作输入时相关寄存器的配置。
3. 掌握库函数中 GPIO 配置为输入的函数以及读取 GPIO 状态的函数。
4. 掌握编写轻触按键检测程序，检测 4 个按键状态。
5. 掌握编写触摸按键检测程序（触摸 IC 实现，ADC 实现的触摸按键在 SAADCA 章节后讲解），检测触摸按键状态。

2. 硬件电路设计

IK-52840DK 开发板上了使用了 4 个轻触键和 1 个触摸按键，触摸按键是通过触摸芯片实现的，输出的信号类型和轻触键一样也是开关量。对于 nRF52840 来说，检测轻触键和触摸按键并没有本质上的区别，都是通过读取按键连接的 GPIO 的状态来确定按键的状态。轻触键和触摸按键区别是他们本身的硬件原理不一样。

2.1.1. 轻触按键

轻触按键又称轻触开关，是电路中常用的一种开关元器件，也是一种常用的人机接口。广泛用于家电、数码产品、便携仪产品、电脑产品等电子设备中。

IK-52840DK 开发板上设计了 4 个轻触按键 S1、S2、S3、S4，分别连接到 nRF52840 的 GPIO P0.11 P0.12 P0.24 和 P0.25。我们可以通过读取轻触按键对应的 GPIO 的状态来判断该按键是否按下。

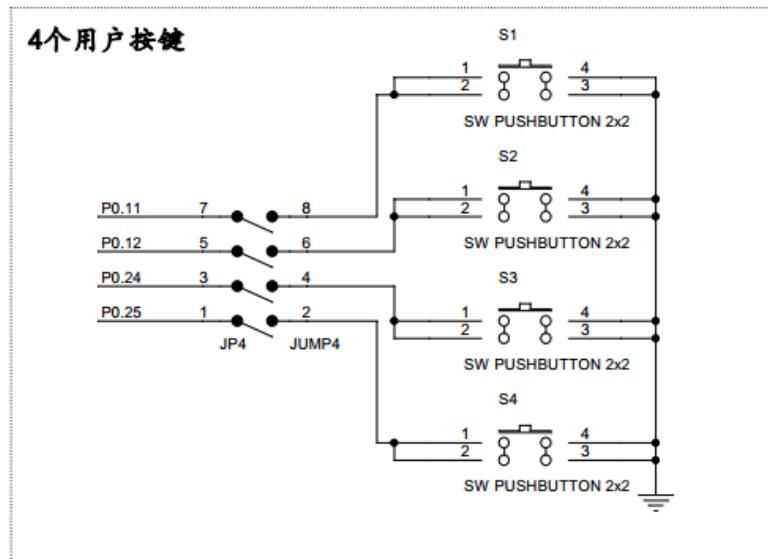


图 8-1：按键检测电路

开发板上 4 个轻触按键占用的 nRF52840 的引脚如下表：

表 8-1：轻触按键引脚分配

名称	引脚	说明
按键 S1	P0.11	独立 IO
按键 S2	P0.12	独立 IO
按键 S3	P0.24	独立 IO
按键 S4	P0.25	独立 IO

- ◆ 注：独立 IO 表示开发板没有其他的电路使用这个 IO。

轻触按键，顾名思义我们只需要施加很小的力量即可改变开关连接的状态。轻触按键在所需外力作用下（按下按键）触点导通，无外力作用时（释放按键）触点断开，如下图所示：

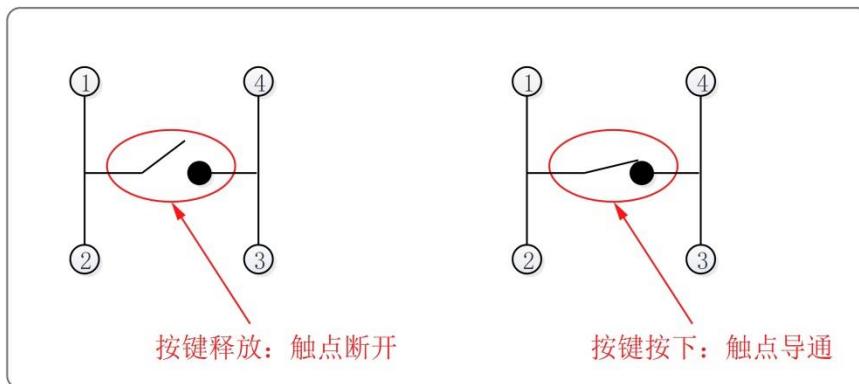


图 8-2：轻触开关原理

在轻触按键的电路中，轻触按键一端接 GND，当轻触按键按下时，轻触按键触点导通，轻触按键对应的引脚电平为低电平，这样，nRF52840 检测到按键对应的引脚为低电平时即可判断按键按下。当轻触按键释放时，轻触按键触点断开，由于对应的引脚开启了内部上拉电阻，所以轻触按键对应的引脚为高电平，这样，nRF52840 检测到按键对应的引脚为高电平时即可判断按键释放。

2.1.2. 触摸按键

开发板上设计了一路基于 TTP223 触摸芯片的触摸按键，TTP223 是电容式单键触摸按键 IC，电压输入范围为 2.0V~5.5V。TTP223 利用操作者的手指与触摸按键焊盘之间产生电荷电平来进行检测，通过监测电荷的微小变化来确定手指接近或者触摸到感应表面。没有任何机械部件，不会磨损，其感测部分可以放置到任何绝缘层（通常为玻璃或塑料材料）的后面，很容易制成与周围环境相密封的键盘。

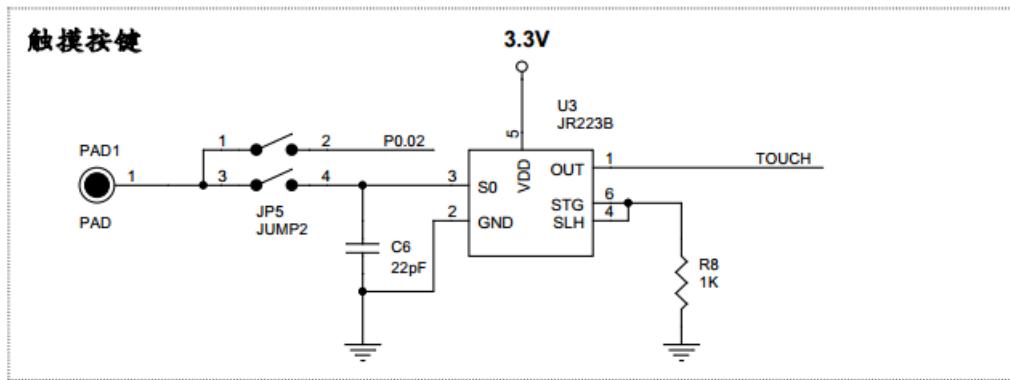


图 8-3：触摸按键检测电路

TTP223 的检测灵敏度可通过外部电容值(上图中的 C6)来调整。SLH 引脚用于设置 TTP223 的输出方式。

- 1) $SLH = 0$: 触摸时, TTP223 的 OUT 引脚输出高电平。
- 2) $SLH = 1$: 触摸时, TTP223 的 OUT 引脚输出低电平。

本电路中, $SLH = 0$, 所以触摸时 OUT 引脚输出高电平, 无触摸时 OUT 引脚输出低电平。注意一下, 这和轻触按键的电路输出刚好是反的, 轻触按键电路是按键按下时, 电路输出低电平, 无按键按下时电路输出高电平。

为什么触摸按键输出的信号不做成和轻触按键一样, 也是按键时输出低电平, 无按键时输出高电平?

这是因为: 作为开发板, 要方便用户测试, 2 种不同类型的输出方式更方便我们使用, 另外在后续的章节中我们还会用到按键信号的上升沿和下降沿, 这时就可以通过轻触键和触摸按键来获取, 而不需要另外接线。

电路中, 通过跳线 JP5 可以将 PAD 直接连接到 P0.02 引脚, P0.02 可以配置为 SAADC 输入通道 0, 所以可以通过 SAADC 直接检测触摸按键而不需要通过触摸 IC, 通过 SAADC 实现触摸按键在 SAADC 章节后讲解, 本章只讲解通过触摸 IC 实现触摸按键。

触摸按键占用的 nRF52840 的引脚如下表:

表 8-2: 触摸按键引脚分配

名称	引脚	说明
触摸按键	P0.02	和电位器共用 IO

2.1.3. 按键检测电路考虑因素

按键检测电路设计的时候, 需要我们考虑两个方面: 按键释放时 GPIO 口状态的确定和按键抖动。

- 1) 按键释放时 GPIO 口状态的确定

按键检测电路中，当按键释放后要能保证 IO 口电平是确定的，即按键释放时 IO 口固定为高电平或低电平，所以一般按键电路中，都会通过上拉电阻或下拉电阻来保证当按键释放时 IO 处于固定的电平。在上面的按键电路中，我们并没有看到电路中有上拉或下拉电阻，这是因为 nRF52840 的所有 GPIO 在片内都集成了上拉/下拉电阻，我们使用 GPIO 检测按键时只需通过软件打开上拉/下拉电阻就可以了，而不需要在片外增加上拉/下拉电阻。

2) 按键硬件消抖

对于按键硬件上的消抖，一般常用的方式是在按键上并接一个容值约 0.1uF 左右电容，利用电容两端的电压不能突变的特性，消除抖动时产生的毛刺电压。虽然电容可以起到消除抖动的作用，但是在考虑按键灵敏度的情况下，电容是无法完全消除抖动的，消除抖动还需要软件的配合。

本电路中，没有采用电容消抖的方式，而是通过软件消抖，对于检测按键状态已经完全足够。

3. GPIO 输入检测原理

nRF52840 的 48 个 GPIO 都可以通过对应的寄存器配置为输入。下面的 GPIO 原理框图展示了配置 GPIO 为输入时的逻辑。从图中我们可以看到，当 GPIO 作为输入使用的时候，需要配置引脚方向寄存器、输入缓冲寄存器以及上/下拉电阻，配置完成之后，即可通过 IN 寄存器读取当前 GPIO 的状态是高电平还是低电平（即逻辑 1 还是逻辑 0）。

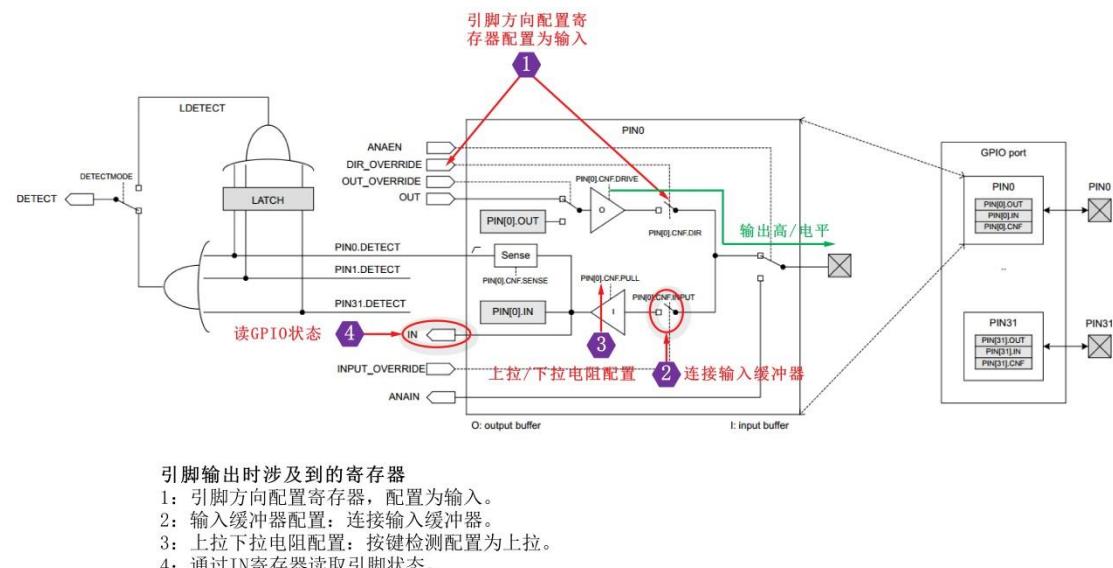


图 8-4: GPIO 输入原理

❖ **注: GPIO 除了输出驱动和输入检测，还有一个“感知”功能，“感知”功能和低功耗、唤醒密切相关，所以这一部分内容会放到低功耗相关的章节进行讲解。**

4. 软件设计

4.1. GPIO 配置

nRF52840 提供的 10 个用于操作 GPIO 的寄存器中，和输入相关的寄存器有方向配置寄存器 DIR 和 DIRCLR，引脚配置寄存器 PIN_CNF 以及 GPIO 端口状态寄存器 IN。使用 GPIO 进入输入检测的时候，毫无疑问首先需要配置 GPIO 输入相关的寄存器：

- 配置 GPIO 为输入。
- 连接输入缓冲器。
- 根据需求配置是否开启上拉或下拉电阻。

配置完成后，通过读取 IN 寄存器来获取 GPIO 当前的状态是高电平还是低电平。

3. 配置 IO 为输入

配置 IO 为输入的时候，相关的寄存器有 DIR、DIRCLR 和 PIN_CNF[n] n=0~31 寄存器，这 3 个寄存器都是 32 位的，只能按字访问，其中 DIR 和 DIRCLR 寄存器仅用来配置引脚方向。PIN_CNF 寄存器中的位 0 用来配置方向，其它的位用来配置、输入缓冲器、驱动能力等，DIR 和 PIN_CNF 寄存器在第六章中已经介绍过，不再赘述。这里我们来看一下配置输入的重写寄存器 DIRCLR。

- DIRCLR：引脚方向（输入）配置寄存器
- P0 端口：可配置的 IO 是 P0.00~P0.31。
 - P1 端口：可配置的 IO 是 P1.00~P1.15。

表 8-3: DIRCLR 寄存器

位	Field	RW	复位值	描述
位 0	PIN0	读/写	0	读，0：引脚 Px.00 方向为输入。 读，1：引脚 Px.00 方向输出。 写，1：设置引脚 Px.00 为输入，写0无效。
位 1	PIN1	读/写	0	读，0：引脚 Px.01 方向为输入。 读，1：引脚 Px.01 方向输出。 写，1：设置引脚 Px.01 为输入，写0无效。
...
位 31	PIN31	读/写	0	读，0：引脚 Px.31 方向为输入。 读，1：引脚 Px.31 方向输出。 写，1：设置引脚 Px.31 为输入，写0无效。

注意：虽然 DIR 和 DIRCLR 可以配置 GPIO 为输入，但是在配置引脚的时候，更常用的是直接通过 PIN_CNF[n] 寄存器来配置引脚方向，这样就不用操作多个寄存器了。

如果我们仅仅是配置引脚方向为输入的话，通常使用 DIRCLR 寄存器，这是因为 DIRCLR 寄存器的各个位写 1 即配置对应的引脚为输出，而写零无效，这样在配置某个引脚为输入的时候直接向 DIRCLR 对应的位写 1 即可，而不用担心会影响到其它其它引脚的方向配置（如果使用 DIR 寄存器，为了不影响其它引脚方向配置，就需要先读出端口方向配置，然后修改对应的位再写回去，操作比较麻烦）。

4. 连接输入缓冲器

是否连接输入缓冲器是由 PIN_CNF 寄存器中的 INPUT（即 PIN_CNF 寄存器的位 1）决定的。

- INPUT 写入 1：断开该 GPIO 的输入缓冲器。
- INPUT 写入 0：连接该 GPIO 的输入缓冲器。

◆ 注意：INPUT 的复位值是 1，即复位后，输入缓冲器是断开的。

5. 读 GPIO 状态

通过读 IN 寄存器即可获取 GPIO 状态，需要注意的是 IN 寄存器是 32 位的，所以读取的时候会一次读取 32 个 GPIO 的状态。

- P0 端口：可配置的 IO 是 P0.00~P0.31。
- P1 端口：可配置的 IO 是 P1.00~P1.15。

■ IN：读 GPIO 端口寄存器。

表 8-4: IN 寄存器

位	Field	RW	复位值	描述
位 0	PIN0	只读	0	读，0：引脚 Px.00 输入为低电平（逻辑“0”）。 读，1：引脚 Px.00 输入为高电平（逻辑“1”）。
位 1	PIN1	只读	0	读，0：引脚 Px.01 输入为低电平（逻辑“0”）。 读，1：引脚 Px.01 输入为高电平（逻辑“1”）。
...
位 31	PIN31	只读	0	读，0：引脚 Px.31 输入为低电平（逻辑“0”）。 读，1：引脚 Px.31 输入为高电平（逻辑“1”）。

4.2. 库函数的应用

了解了 GPIO 输入相关的寄存器后，接下来，我们看一下库配置 GPIO 为输入以及读取 GPIO 状态需要用到的库函数。

4. 配置 GPIO 为输入，连接缓冲器及上/下拉电阻配置

Nordic 的库中用于初始化 GPIO 为输入的库函数是 nrf_gpio_cfg_input () 和 nrf_gpio_range_cfg_input () , nrf_gpio_cfg_input () 用于配置单个的 GPIO 为输入 , nrf_gpio_range_cfg_input () 用于配置连续编号的 GPIO 为输入 , 函数原型如下表所示:

表 8-5: nrf_gpio_cfg_input () 函数

函数原型	<pre>_STATIC_INLINE void nrf_gpio_cfg_input (uint32_t pin_number, nrf_gpio_pin_pull_t pull_config)</pre>
函数功能	配置指定的 GPIO 为输入。
参数	<p>[in] pin_number: 引脚编号: 0~47, 其中: 0~31 对应 P0.00~P0.31,32~47 对应 P1.00~P1.15。。</p> <p>[in] pull_config: 上/下电阻配置。可以配置为上拉、下拉或无上/下拉电阻。对应的宏如下:</p> <ul style="list-style-type: none"> • NRF_GPIO_PIN_PULLUP: 上拉。 • NRF_GPIO_PIN_PULLDOWN: 下拉。 • NRF_GPIO_PIN_NOPULL: 无上/下拉电阻。
返回值	无。

表 8-6: nrf_gpio_range_cfg_input () 函数

函数原型	<pre>_STATIC_INLINE void nrf_gpio_range_cfg_input (uint32_t pin_range_start, uint32_t pin_range_end, nrf_gpio_pin_pull_t pull_config)</pre>
函数功能	配置连续编号的 GPIO 为输入。注意引脚编号必须是连续的才可以使用该函数进行配置，如配置 P0.11~P0.12 为输入。
参数	<p>[in] pin_range_start: 起始引脚编号。</p> <p>[in] pin_range_end: 结束引脚编号。</p> <p>[in] pull_config: 上/下电阻配置。可以配置为上拉、下拉或无上/下拉电阻。对应的宏如下:</p> <ul style="list-style-type: none"> • NRF_GPIO_PIN_PULLUP: 上拉。 • NRF_GPIO_PIN_PULLDOWN: 下拉。 • NRF_GPIO_PIN_NOPULL: 无上/下拉电阻。
返回值	无。

由上表可以看出，配置 GPIO 为输入的时候，只需要调用库函数，并将需要配置的引脚的编号和上/下电阻配置对应的宏作为函数的输入参数即可完成配置。

■ 应用示例：配置 P0.11 为输入，关闭上/下电阻配置、开启下拉和开启上拉的示例如下：

1. `nrf_gpio_cfg_input(NRF_GPIO_PIN_MAP(0, 11),NRF_GPIO_PIN_NOPULL); //配置 P0.11 为输入，无上/下拉电阻`
2. `nrf_gpio_cfg_input(NRF_GPIO_PIN_MAP(0, 11),NRF_GPIO_PIN_PULLDOWN); //配置 P0.11 为输入，开启下拉电阻`
3. `nrf_gpio_cfg_input(NRF_GPIO_PIN_MAP(0, 11),NRF_GPIO_PIN_PULLUP); //配置 P0.11 为输入，开启上拉电阻`

■ 应用示例：配置 P0.11~P0.12 两个个连续的 GPIO 为输入并开启上拉电阻
可以使用 `nrf_gpio_cfg_input()` 函数逐个配置如下：

1. `nrf_gpio_cfg_input(NRF_GPIO_PIN_MAP(0, 11),NRF_GPIO_PIN_PULLUP); //配置 P0.13 为输入，开启上拉电阻`
2. `nrf_gpio_cfg_input(NRF_GPIO_PIN_MAP(0, 12),NRF_GPIO_PIN_PULLUP); //配置 P0.14 为输入，开启上拉电阻`

也可以使用 `nrf_gpio_range_cfg_input()` 配置如下：

```
nrf_gpio_range_cfg_input (NRF_GPIO_PIN_MAP(0, 11), NRF_GPIO_PIN_MAP(0, 12),  
NRF_GPIO_PIN_PULLUP);
```

我们打开输入配置函数，可以看到，和输出配置函数一样，在输入配置函数中同样使用了默认的标准参数配置了 GPIO 的其它项目。

代码清单：nrf_gpio_cfg_input 函数

```
1. __STATIC_INLINE void nrf_gpio_cfg_input(uint32_t pin_number, nrf_gpio_pin_pull_t  
   pull_config)  
2. {  
3.     nrf_gpio_cfg(  
4.         pin_number,  
5.         NRF_GPIO_PIN_DIR_INPUT,  
6.         NRF_GPIO_PIN_INPUT_CONNECT,  
7.         pull_config,  
8.         NRF_GPIO_PIN_S0S1,  
9.         NRF_GPIO_PIN_NOSENSE);  
10. }
```

上述代码执行后，对应的 GPIO 被配置为：

- 方向配置为输入。
- 连接输入缓冲器。
- 开启上拉电阻。
- 逻辑“0”标准驱动能力，逻辑“1”标准驱动能力。
- 引脚电平感知关闭。

如果需要配置 GPIO 的所有参数，可以使用 nrf_gpio_cfg()函数，nrf_gpio_cfg()函数的用法可以参考第五章中的描述。

5. 读取 GPIO 状态

Nordic 的库中用于读取 GPIO 状态的库函数常用的有下面 2 个：

- nrf_gpio_pin_read(): 读取指定的 GPIO 的状态。
- nrf_gpio_port_in_read(): 读取端口所有的 GPIO 的状态，对于 nRF52840 来说，只有一个 P0 端口，调用该函数后，会读取 P0 端口所有 GPIO 的状态，即 P0.00~P0.31 这 32 个 GPIO 的状态。

表 8-7: nrf_gpio_pin_read()函数

函数原型	<code>_STATIC_INLINE uint32_t nrf_gpio_pin_read (uint32_t pin_number)</code>
函数功能	读取指定编号的引脚的状态。
参数	[in] pin_number: 引脚编号, 0~31。
返回值	0: 引脚状态是低电平。 1: 引脚状态是高电平。

表 8-8: nrf_gpio_port_in_read()函数

函数原型	<code>_STATIC_INLINE uint32_t nrf_gpio_port_in_read (NRF_GPIO_Type const * p_reg)</code>
函数功能	读取端口所有的 GPIO 的状态。
参数	[in] p_reg: 指向 GPIO 寄存器结构体。
返回值	GPIO 端口状态。

4.3. 轻触按键检测实验

❖ 注：本节对应的实验源码是：“实验 8-1：GPIO 输入按键检测”。

4.3.1. 代码编写

根据前文的描述，对于按键状态的检测，只需调用库函数 nrf_gpio_range_cfg_input() 配置按键连接的 GPIO（因为 4 个按键 S1~S4 的 GPIO 的编号是连续的，所以使用 nrf_gpio_range_cfg_input() 配置比较方便）。之后读取 GPIO 状态即可判断按键的状态是释放还是按下。

程序清单：轻触按键检测

```
1. /*****
2. * 描述 : main 函数
3. * 入参 : 无
4. * 返回值 : int 类型
5. *****/
6. int main(void)
7. {
8.     //设置 GPIO 输出电压为 3.3V
9.     gpio_output_voltage_setup_3v3();
10.
11.    //配置用于驱动 LED 指示灯 D1 D2 D3 D4 的引脚脚，即配置 P0.13~P0.16 为输出，并将 LED
12.    //的初始状态设置为熄灭
13.    //配置用于检测 4 个按键的 IO 位输入，并开启上拉电阻
14.    bsp_board_init(BSP_INIT_LEDS | BSP_INIT_BUTTONS);
15.
16.    while (true)
17.    {
18.        //检测按键 S1 是否按下
19.        if(nrf_gpio_pin_read(BUTTON_1) == 0)
20.        {
21.            //点亮 LED 指示灯 D1
22.            nrf_gpio_pin_clear(LED_1);
23.            while(nrf_gpio_pin_read(BUTTON_1) == 0); //等待按键释放
24.            //熄灭 LED 指示灯 D1
25.            nrf_gpio_pin_set(LED_1);
26.        }
27.        //检测按键 S2 是否按下
28.        if(nrf_gpio_pin_read(BUTTON_2) == 0)
29.        {
30.            nrf_gpio_pin_clear(LED_2);
31.            while(nrf_gpio_pin_read(BUTTON_2) == 0); //等待按键释放
32.            nrf_gpio_pin_set(LED_2);
33.        }
34.        //检测按键 S3 是否按下
35.        if(nrf_gpio_pin_read(BUTTON_3) == 0)
36.        {
```

```
37.         nrf_gpio_pin_clear(LED_3);
38.         while(nrf_gpio_pin_read(BUTTON_3) == 0); //等待按键释放
39.         nrf_gpio_pin_set(LED_3);
40.     }
41.     //检测按键 S4 是否按下
42.     if(nrf_gpio_pin_read(BUTTON_4) == 0)
43.     {
44.         nrf_gpio_pin_clear(LED_4);
45.         while(nrf_gpio_pin_read(BUTTON_4) == 0); //等待按键释放
46.         nrf_gpio_pin_set(LED_4);
47.     }
48. }
49. }
```

4.3.2. 硬件连接

本实验需要使用 P0.13 P0.14 P0.15 P0.16 驱动 4 个 LED 指示灯，P0.11 P0.12 P0.24 P0.25 检测 4 个按键的状态，需要用跳线帽短接这些引脚，如下图所示。

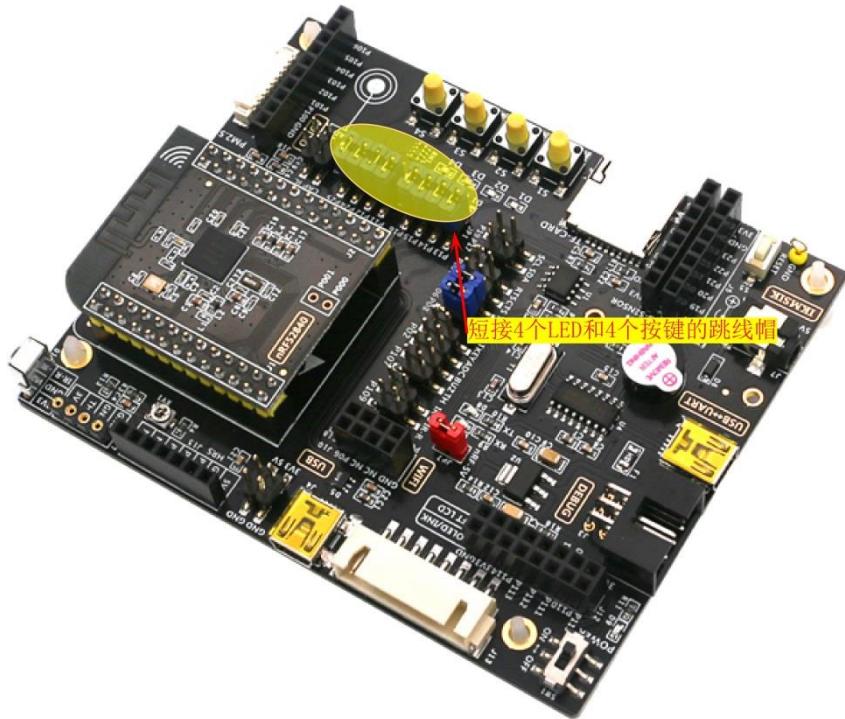


图 8-5：开发板跳线帽短接

4.3.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 8-1: GPIO 输入按键检测”，将解压后得到的文件夹“key”拷贝到合适的目录，如“D\NRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\key\project\mdk5”目录下的工程“key.uvproj”。

4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，分别按下开发板上的按键 S1~S4，对应的指示灯 D1~D4 会点亮，释放按键后，对应的指示灯熄灭。

4.4. 触摸按键检测实验

❖ 注：本节对应的实验源码是：“实验 8-2：GPIO 输入触摸按键检测”。

4.4.1. 代码编写

触摸按键的检测方式和轻触按键一样，不同的是轻触按键检测到低电平时认为按键按下，触摸按键检测到高电平时认为按键按下。

程序清单：触摸按键检测

```
1. /*****
2. * 描述 : main 函数
3. * 入参 : 无
4. * 返回值 : int 类型
5. *****/
6. int main(void)
7. {
8.     //配置用于驱动 LED 指示灯 D1 的引脚为输出，即配置 P0.13 为输出
9.     nrf_gpio_cfg_output(LED_1);
10.    //设置 LED 指示灯 D1 的初始状态为熄灭
11.    nrf_gpio_pin_set(LED_1);
12.
13.    //配置 P0.02 为输入，用于检测触摸按键状态
14.    nrf_gpio_cfg_input(TOUCH_PIN,NRF_GPIO_PIN_PULLUP);
15.
16.    while (true)
17.    {
18.        //检测触摸按键是否按下，注意触摸按键按下后，P0.02 为高电平
19.        if(nrf_gpio_pin_read(TOUCH_PIN) == 1)
20.        {
21.            //点亮 LED 指示灯 D1
22.            nrf_gpio_pin_clear(LED_1);
23.            //等待按键释放
24.        }
25.    }
26.
```

```
24.         while(nrf_gpio_pin_read(TOUCH_PIN) == 1);
25.             //熄灭 LED 指示灯 D1
26.             nrf_gpio_pin_set(LED_1);
27.         }
28.     }
29. }
```

4.4.2. 硬件连接

本实验需要使用 P0.13 驱动 LED 指示灯 D1, P0.02 检测触摸按键的状态，触摸 PAD 连接到触摸芯片，需要用跳线帽短接这些引脚，如下图所示。

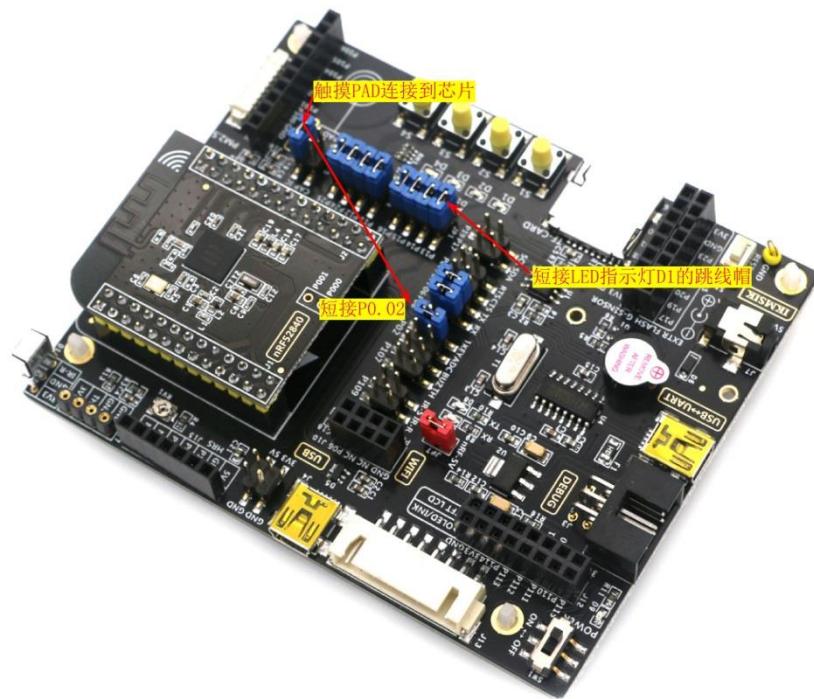


图 8-6：开发板跳线帽短接

4.4.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 8-2: GPIO 输入触摸按键检测”，将解压后得到的文件夹“touch_key”拷贝到合适的目录，如“D\ nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\touch_key\project\mdk5”目录下的工程“touch_key.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。

6. 程序运行后，用手指去接触触摸按键，指示灯 D1 点亮，手指拿开，指示灯 D1 熄灭。

第九章：寄存器类型和外设资源共享

1. 学习目的

1. 学习 nRF52840 的寄存器有哪些类型，以及区别。后续章节涉及到的复杂的外设都有多种类型，在学习之前，我们要先掌握寄存器类型这个知识点。
2. 了解 nRF52840 的外设之间的资源共享，这部分内容说明了哪些外设共用了资源。nRF52840 有很多外设，并不是所有的外设都可以同时使用，通过这一章节的学习，我们应掌握哪些外设可以同时使用，哪些外设同时只能使用一个。

2. 寄存器类型

nRF51 和 nRF52 的寄存器和一般的单片机有所差别，nRF51 和 nRF52 的寄存器分为下面的四种类型。

1. 任务寄存器 Task：可以由程序或事件触发，用于启动某一个任务。比如我们用 UART 发送数据时，就是通过触发 UART 的 STARTTX 任务来启动 UART 发送序列的。那么，任务寄存器是如何触发的？
 - 向任务寄存器的地址写“1”即可触发该任务。还是以 UART 的 STARTTX 任务为例来说明，UART 的 STARTTX 任务寄存器地址是：0x40002000，所以应用程序只需向这个地址写“1”即可触发 UART 的 STARTTX 任务。
2. 事件寄存器 Event：外设产生“事件”时，外设通过对应的事件寄存器和中断（如果该事件使能了中断）向 CPU 提供指示。如 UART 接收数据时，如果使能了 RXDRDY 事件中断，当 UART 接收到数据时，会产生中断，同时 RXDRDY 事件寄存器置位，CPU 通过中断和 RXDRDY 事件寄存器即可获知 UART 接收到数据。
3. 快捷方式寄存器：顾名思义，快捷方式寄存器用于提供一些快捷的操作而不用去配置多个寄存器。在 nRF52840 中，快捷方式寄存器的作用是连接同一个外设的事件和任务，当事件发生时，自动触发任务而不需要 CPU 参与。如定时器的计数值等于设定的比较值的时候，会产生比较事件，这时，需要将计数值清零重新开始计数，那么这种情况下就可以配置定时器的快捷方式寄存器，让比较事件自动清零计数值，而不使用软件去清零计数值。
4. 通用寄存器：和一般单片机的寄存器一样。

任务和事件一个很重要的用处是和 PPI（外设交叉互联）配合使用，通过配置 PPI，将某个事件和任务“连接”起来，连接后，当事件产生时会在硬件上自动触发任务而不需要 CPU 参与。这种机制能有效减少 CPU 的占用时间，降低 CPU 的负荷，同时也节省了中断资源，并且由于是硬件直接触发，硬件的速度快于软件速度，所以也提高了对事件的响应速度。

3. 外设资源共享

nRF52840 的每一个外设都分配了固定的大小为 0x1000 字节的地址空间，每个外设都有一个 ID，ID 和外设的对应关系是：从起始地址为 0x40000000 的外设开始编号，即起始地址为 0x40000000 的外设的 ID 是 0，起始地址为 0x40001000 的外设的 ID 是 1，依此类推。

具有相同 ID 的外设共享寄存器等资源，所以，具有相同 ID 的外设同时只能使用一个，比如 ID 为 2 的有 UARTE 和 UART，也就是 UARTE 和 UART 同时只能使用一个。

表 9-1：外设表

ID	基址	外设		功能描述
0	0x40000000	CLOCK	CLOCK	时钟控制。
0	0x40000000	POWER	POWER	电源控制。
1	0x40001000	RADIO	RADIO	2.4 GHz 无线电。
2	0x40002000	UART	UART0	通用异步收发器（不建议使用）。
2	0x40002000	UARTE	UARTE0	具有 EasyDMA 的通用异步收发器。
3	0x40003000	SPI	SPI0	SPI 主机 0（不建议使用）。
3	0x40003000	SPIM	SPIM0	SPI 主机 0。
3	0x40003000	SPIS	SPIS0	SPI 从机 0。
3	0x40003000	TWI	TWI0	TWI 主机 0（不建议使用）。
3	0x40003000	TWIM	TWIM0	TWI 主机 0。
3	0x40003000	TWIS	TWIS0	TWI 从机 0。
4	0x40004000	SPI	SPI1	SPI 主机 1（不建议使用）。
4	0x40004000	SPIM	SPIM1	SPI 主机 1。
4	0x40004000	SPIS	SPIS1	SPI 从机 1。
4	0x40004000	TWI	TWI1	TWI 主机 1（不建议使用）。
4	0x40004000	TWIM	TWIM1	TWI 主机 1。
4	0x40004000	TWIS	TWIS1	TWI 从机 0。
5	0x40005000	NFCT	NFCT	近场通讯标签。
6	0x40006000	GPIOE	GPIOE	GPIO 任务和事件。
7	0x40007000	SAADC	SAADC	模数转换器。
8	0x40008000	TIMER	TIMER0	定时器 0。

9	0x40009000	TIMER	TIMER1	定时器 1。
10	0x4000A000	TIMER	TIMER2	定时器 2。
11	0x4000B000	RTC	RTC0	实时计数器 0。
12	0x4000C000	TEMP	TEMP	片内温度传感器。
13	0x4000D000	RNG	RNG	随机数发生器。
14	0x4000E000	ECB	ECB	AES 电码本加密模式。
15	0x4000F000	AAR	AAR	加速地址解析器。
15	0x4000F000	CCM	CCM	AES CCM 模式加密。
16	0x40010000	WDT	WDT	看门狗定时器。
17	0x40011000	RTC	RTC1	实时计数器 1。
18	0x40012000	QDEC	QDEC	正交译码器。
19	0x40013000	COMP	COMP	通用比较器。
19	0x40013000	LPCOMP	LPCOMP	低功耗比较器。
20	0x40014000	EGU	EGU0	事件发生器单元 0。
20	0x40014000	SWI	SWI0	软件中断 0。
21	0x40015000	EGU	EGU1	事件发生器单元 1。
21	0x40015000	SWI	SWI1	软件中断 1。
22	0x40016000	EGU	EGU2	事件发生器单元 2。
22	0x40016000	SWI	SWI2	软件中断 2。
23	0x40017000	EGU	EGU3	事件发生器单元 3。
23	0x40017000	SWI	SWI3	软件中断 3。
24	0x40018000	EGU	EGU4	事件发生器单元 4。
24	0x40018000	SWI	SWI4	软件中断 4。
25	0x40019000	EGU	EGU5	事件发生器单元 5。
25	0x40019000	SWI	SWI5	软件中断 5。
26	0x4001A000	TIMER	TIMER3	定时器 3。
27	0x4001B000	TIMER	TIMER4	定时器 4。
28	0x4001C000	PWM	PWM0	脉冲宽度调制单元 0。

29	0x4001D000	PDM	PDM	脉冲密度调制 (数字麦克风接口)。
30	0x4001E000	ACL	ACL	访问控制列表。
30	0x4001E000	NVMC	NVMC	非易失性存储器控制器。
31	0x4001F000	PPI	PPI	可编程外设交叉互联。
32	0x40020000	MWU	MWU	内存监视单元。
33	0x40021000	PWM	PWM1	脉冲宽度调制单元 1。
34	0x40022000	PWM	PWM2	脉冲宽度调制单元 2。
35	0x40023000	SPI	SPI2	SPI 主机 2 (不建议使用)。
35	0x40023000	SPIM	SPIM2	SPI 主机 2。
35	0x40023000	SPIS	SPIS2	SPI 从机 2。
36	0x40024000	RTC	RTC2	实时计数器 2。
37	0x40025000	I2S	I2S	I2S 总线。
38	0x40026000	FPU	FPU	FPU 中断。
39	0x40027000	USBD	USBD	通用串行总线设备。
40	0x40028000	UARTE	UARTE1	具有 EasyDMA 的通用异步收发器。
41	0x40029000	QSPI	QSPI	外部存储器接口。
45	0x4002D000	PWM	PWM3	脉冲宽度调制单元 3。
47	0x4002F000	SPIM	SPIM3	SPI 主机 3。
0	0x50000000	GPIO	GPIO	通用输入输出 (不建议使用)。
0	0x50000000	GPIO	P0	通用输入输出端口 0。
0	0x50000300	GPIO	P1	通用输入输出端口 1。
42	0x5002A000	CRYPTOCELL	CRYPTOCELL	CryptoCell 子系统控制接口。
N/A	0x10000000	FICR	FICR	工厂信息配置。
N/A	0x10001000	UICR	UICR	用户信息配置。

第十章：串口数据收发

1. 学习目的

1. 掌握 USB 转串口电路的原理和设计。
2. 学习 nRF52840 的 UART 和 UARTE 原理、区别以及相关的寄存器（nRF52840 片内集成了 1 个 UART 和 2 个 UARTE）。
3. 掌握库函数中串口初始化、发送和接收数据相关的函数。
4. 编写驱动串口收发程序，在此基础上，更进一步，编写串口发送命令控制 4 个 LED 指示灯亮灭的程序。

2. 硬件电路设计

IK-52840DK 开发板上设计了 USB 转串口电路，它的主要作用有 2 个：

- 1) USB 转串口通讯，通过 USB 数据线连接到计算机的 USB 口即可使用串口通讯功能。
- 2) 开发板供电通过 USB 可以为开发板供电（计算机 USB 可以提供 500mA 的电流）。

USB 转串口电路如下图所示，串口接收和发送的管脚上均连接了 LED 指示灯，收发数据时指示灯会闪烁，这样，更方便我们从硬件的角度观察串口有没有在进行数据收发。电路中的自恢复保险丝用于保护开发板和计算机 USB 口。

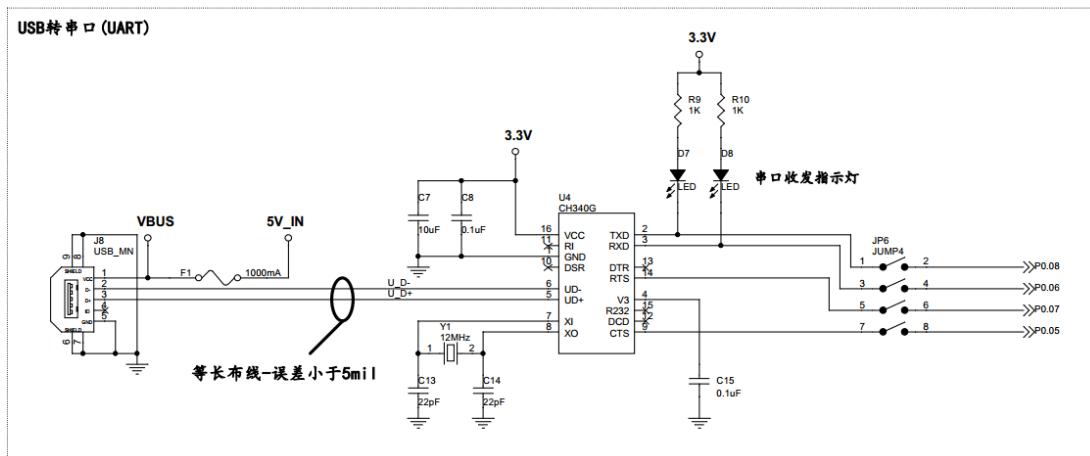


图 10-1: USB 转串口电路

USB 转串口电路采用的 USB 转串口芯片是 CH340，该芯片特点如下：

- 全速 USB 设备接口，兼容 USB V2.0。
- 标准 USB 打印口，用于升级原并口打印机，兼容相关的 USB 规范。
- 支持 IEEE-1284 规范的双向通讯，支持单向和双向传输打印机。
- 由于是通过 USB 转换的打印口，所以只能做到应用层兼容，而无法绝对相同。
- 软件兼容 CH341，可以直接使用 CH341 的驱动程序。

- 支持 5V 电源电压和 3.3V 电源电压甚至 3V 电源电压。
- 采用无铅封装，兼容 RoHS，引脚兼容 CH341。

另外，CH340 芯片内置了 USB 上拉电阻，所以我们直接将 UD+和 UD-引脚连接 USB 总线上即可，而不用在芯片外部加上拉电阻。同时，CH340 芯片也内置了电源上电复位电路，不需要另外增加外部复位电路。

串口占用的 nRF52840 的引脚如下表：

表 10-1：串口引脚分配

名称	引脚	说明
UART_RX	P0.08	串口接收，和 PM2.5 传感器接口、WIFI 模块接口共用 IO。
UART_TX	P0.06	串口发送，和 PM2.5 传感器接口、WIFI 模块接口共用 IO。
UART_RTS	P0.05	串口硬件流控 RTS，和 PM2.5 传感器接口、WIFI 模块接口共用 IO。
UART_CTS	P0.07	串口硬件流控 CTS，和 PM2.5 传感器接口共用 IO。

3. UART 原理

3.1. 功能描述

nRF52840 片内集成了 1 个 UART 外设，其主要特性如下。

- 全双工。
- 自动的流控。
- 奇偶校验并自动产生校验位。
- 1 位停止位。

nRF52840 的 UART 通过 TXD 和 RXD 寄存器发送和接收数据，UART 的原理框图如下图所示：

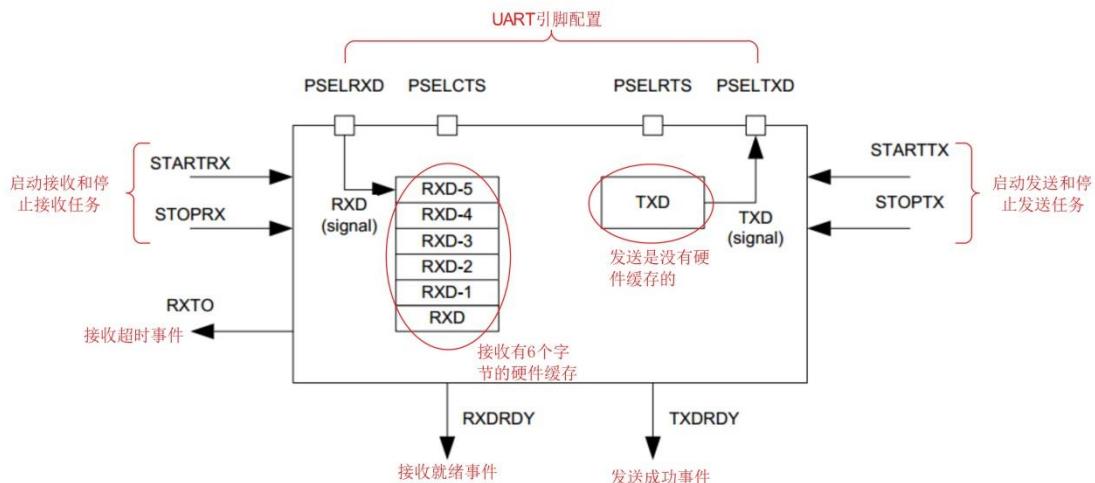


图 10-2：UART 原理框图

从上图中，我们可以看到，UART 共有 4 个引脚配置寄存器，通过这 4 个寄存器可以将

UART 的 4 个信号映射到任何一个物理引脚。UART 的发送通过 STARTTX 任务启动和停止，发送成功后，会产生 TXDRDY 事件。UART 的接收通过 STARTRX 和 STOPRX 任务启动和停止，当数据由硬件 FIFO 提取到 RXD 寄存器时，产生 RXDRDY 事件。

3.2. 配置引脚

nRF52840 的 UART 支持硬件流控，共有 4 个信号：RXD、CTS、TXD 和 RTS。其中 TXD 和 RXD 信号用于发送和接收数据，RTS 和 CTS 用于硬件流控。

这 4 个信号可以通过下表所示的寄存器映射到 nRF52840 的任何一个物理引脚，如可以通过配置 PSELRXD 寄存器将 RXD 映射到 P0.06，即使用 P0.06 作为 UART 的接收引脚，也可以通过配置 PSELRXD 寄存器将 RXD 映射到 P0.08 或其它任何一个引脚。但是需要注意的是：nRF52840 只有 1 个 UART 外设，所以同时只能使用一个 UART，另外，同一时刻只能有一个信号映射到同一个引脚，如不能同时将 P0.08 配置为串口接收和发送。

表 10-2：串口引脚配置

寄存器	说明
PSELRXD	配置串口接收引脚
PSELCTS	配置串口发送引脚
PSELRTS	配置串口硬件流控 RTS 引脚
PSELTXD	配置串口硬件流控 CTS 引脚

当系统处于 OFF 模式时，为了保证 UART 连接的引脚电平正确，UART 的连接的引脚应按照下表所示来配置。

表 10-3：系统处于 OFF 模式时 UART 引脚配置

UART 引脚	方向	输出值
RXD	输入	--
CTS	输入	--
RTS	输出	逻辑 1，即高电平
TXD	输出	逻辑 1，即高电平

3.3. UART 发送

UART 通过触发 STARTTX 任务（即向任务寄存器 STARTTX 中写入 1）启动发送序列，数据通过写入到 TXD 寄存器发送，每次发送一个字节，之后 TXD 中的数据逐个传送到物理线路。当 TXD 中的数据传送完成之后，UART 产生 TXDRDY 事件，这时可以向 TXD 寄存器写入下一个要发送的数据，如下图“关闭硬件流控部分”的时序。

如果要停止 UART 发送，触发 STOPTX 任务（即向任务寄存器 STOPTX 中写入 1）即

可，触发后，UART 发送会立即停止。

一旦使能了串口硬件流控，向 TXD 中写入数据后，如果 CTS 信号有效（CTS 为低电平，允许发送），TXD 中的数据会被传送到物理线路，否则，发送会被挂起直到 CTS 信号有效，如下图“使能硬件流控部分”。

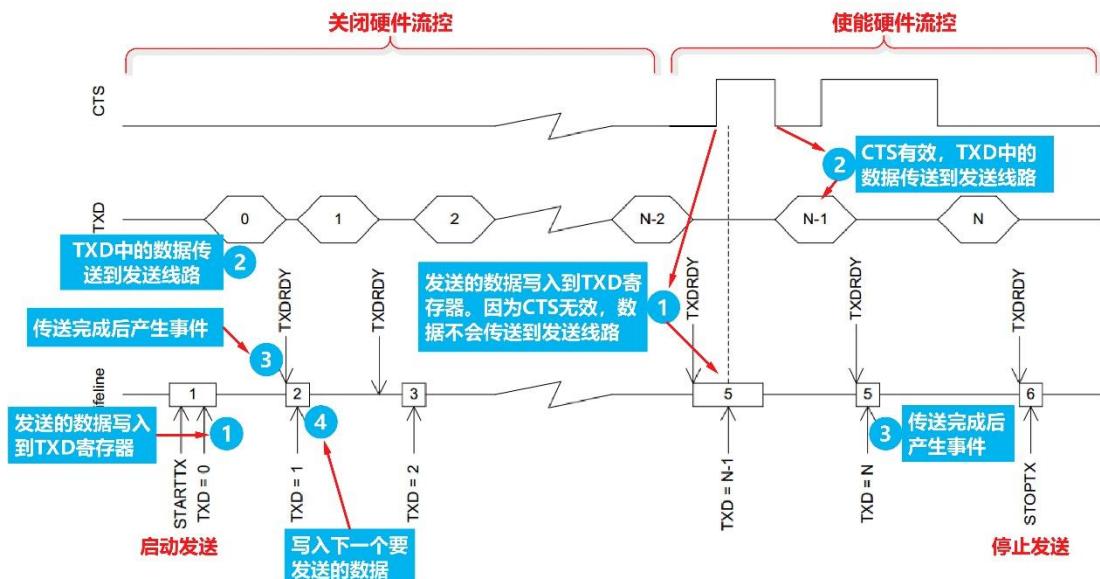


图 10-3: UART 发送时序

3.4. UART 接收

UART 通过触发 STARTRX 任务（即向任务寄存器 STARTRX 中写入 1）启动接收序列，UART 接收有 6 个字节的硬件 FIFO，在接收的数据被覆盖之前可以存储 6 个字节。

UART 接收的数据通过 RXD 寄存器读出，当 RXD 寄存器中的数据被读出后，FIFO 中的数据会移动到 RXD 寄存器。

如果使能了硬件流控，UART 会在 FIFO 剩余 4 个字节空间时失效 RTS 信号，这时 UART 的 FIFO 存在 4 个空闲字节空间，也就是对端发送设备在 RTS 信号失效后可以发送 4 个字节数据而不会导致接收数据被覆盖。从这里我们也可以看出，为了确保数据不被覆盖，发送端必须在 RTS 信号失效后发送 4 个字节数据内停止发送。

RTS 信号在 FIFO 为空时有效，即当 CPU 读出所有接收数据后，RTS 信号有效。如果触发 STOPRX 任务（即向任务寄存器 STOPRX 中写入 1）停止接收序列，RTS 信号失效，但是这时候 UART 还有能力接收 4~5 个字节数据。存在这种可能是因为 STOPRX 任务触发后，UART 能够根据波特率的配置继续接收一段时间内数据，在这段时间结束后，UART 产生超时事件（RXTO 事件）。

为了防止数据丢失，在每个 RXDRDY 事件产生后，RXD 数据只能读取一次。

为了保证 CPU 可以通过 RXDRDY 寄存器检测到所有的 RXDRDY 事件，RXDRDY 事件寄存器必须在读取 RXD 寄存器之前清除，这么做的原因是一旦读出 TXD 中的数据，UART 允许写入新的数据，这时候 FIFO 可能为空，因此在读取 RXD 后会立即产生新的事件。

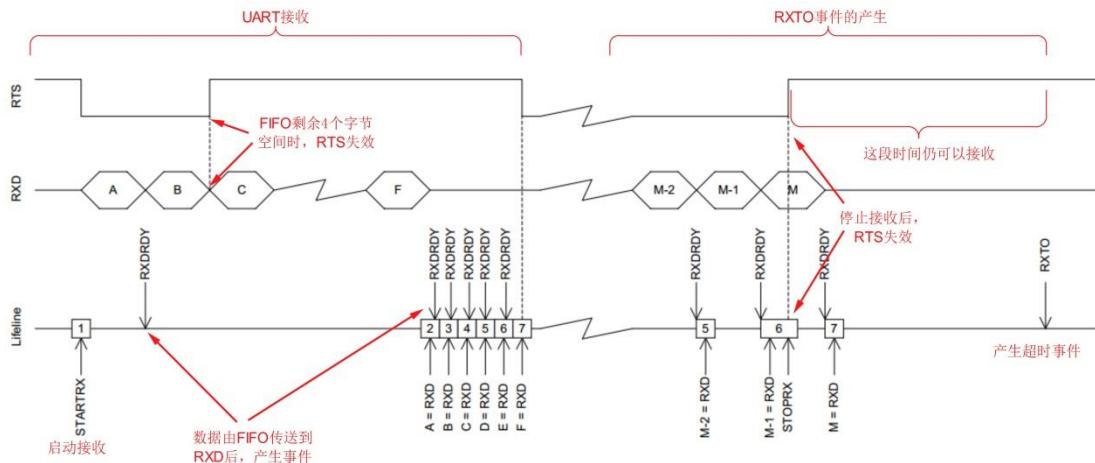


图 10-4: UART 接收时序

3.5. UART 挂起

触发 SUSPEND 任务（即向任务寄存器 SUSPEND 中写入 1）可以挂起 UART，挂起会同时影响到 UART 的发送和接收，即 UART 发送器会停止发送，接收器会停止接收。UART 挂起后，通过触发 STARTTX 任务和 STARTRX 任务即可恢复 UART 的发送和接收。

触发 SUSPEND 任务后，TXD 寄存器中如果有正在发送的数据，UART 会在发送完后挂起。触发 SUSPEND 任务后，UART 接收器的工作方式和触发 STOPRX 任务时一样。

3.6. 错误产生条件

UART 可以捕获硬件上产生额定错误并触发相应事件，下列两种情形会导致错误事件的产生：

- 结束位没有被正确识别。
- RXD 一直被拉低，并且被拉低的时间超过一帧数据的长度。

3.7. 奇偶校验

当使能自动奇偶校验后，发送和接收的 TXD 和 RXD 会分别自动生成奇偶校验位。

4. UART 寄存器

nRF52840 的 UART 的基址如下表所示。

表 10-2: SAADC 基址

外设名称	基址	描述
UART0	0x40002000	通用异步收发器。

UART 相关的寄存器，如下表所示：

表 10-4: UART 相关寄存器

序号	寄存器名	偏移地址	功能描述
任务寄存器			
1	TASKS_STARTRX	0x000	启动 UART 接收。
2	TASKS_STOPRX	0x004	停止 UART 接收。
3	TASKS_STARTTX	0x008	启动 UART 发送。
4	TASKS_STOPTX	0x00C	停止 UART 发送。
5	TASKS_SUSPEND	0x01C	挂起 UART。
事件寄存器			
1	EVENTS_CTS	0x100	CTS 有效（低电平），允许发送。
2	EVENTS_NCTS	0x104	CTS 无效（高电平），不允许发送。
3	EVENTS_RXDRDY	0x108	从 RXD 线路接收到一个字节数据。
4	EVENTS_TXDRDY	0x11C	一字节数据传送至发送线路。
5	EVENTS_ERROR	0x124	检测到错误。
6	EVENTS_RXTO	0x144	接收超时。
快捷方式寄存器			
1	SHORTS	0x200	快捷方式
通用寄存器			
1	INTENSET	0x304	中断使能。
2	INTENCLR	0x308	中断禁止。
3	ERRORSRC	0x480	错误原因。
4	ENABLE	0x500	使能 UART。
5	PSELRTS	0x508	RTS 引脚配置。
6	PSELTXD	0x50C	TXD 引脚配置。
7	PSELCTS	0x510	CTS 引脚配置。
8	PSELRXD	0x514	RXD 引脚配置。
9	RXD	0x518	数据接收。
10	TXD	0x51C	数据发送。
11	BAUDRATE	0x524	UART 波特率配置。

12	CONFIG	0x56C	UART 硬件流控和奇偶校验配置。
----	--------	-------	-------------------

■ SHORTS: 快捷方式寄存器

表 10-5: SHORTS 寄存器

位	Field	RW	复位值	描述
位 3	CTS_STARTRX	读/写	0	CTS 事件和 STARTRX 任务之间的快捷方式 0: 禁止快捷方式。 1: 使能快捷方式。
位 4	NCTS_STOPRX	读/写	0	NCTS 事件和 STOPRX 任务之间的快捷方式 0: 禁止快捷方式。 1: 使能快捷方式。

■ INTENSET: 中断使能寄存器

表 10-6: INTENSET 寄存器

位	Field	RW	复位值	描述
位 0	CTS	读/写	0	写 “1” 使能 CTS 事件中断。 写, 1: 使能 CTS 事件中断。 读, 0: CTS 事件中断已禁止。 读, 1: CTS 事件中断已使能。
位 1	NCTS	读/写	0	写 “1” 使能 NCTS 事件中断。 写, 1: 使能 NCTS 事件中断。 读, 0: NCTS 事件中断已禁止。 读, 1: NCTS 事件中断已使能。
位 2	RXDRDY	读/写	0	写 “1” 使能 RXDRDY 事件中断。 写, 1: 使能 RXDRDY 事件中断。 读, 0: RXDRDY 事件中断已禁止。 读, 1: RXDRDY 事件中断已使能。
位 7	TXDRDY	读/写	0	写 “1” 使能 TXDRDY 事件中断。 写, 1: 使能 TXDRDY 事件中断。 读, 0: TXDRDY 事件中断已禁止。 读, 1: TXDRDY 事件中断已使能。

位 9	ERR	读/写	0	写“1”使能 ERR 事件中断。 写, 1: 使能 ERR 事件中断。 读, 0: ERR 事件中断已禁止。 读, 1: ERR 事件中断已使能。
位 17	RXTO	读/写	0	写“1”使能 RXTO 事件中断。 写, 1: 使能 RXTO 事件中断。 读, 0: RXTO 事件中断已禁止。 读, 1: RXTO 事件中断已使能。

■ INTENCLR: 中断禁止寄存器

表 10-7: INTENCLR 寄存器

位	Field	RW	复位值	描述
位 0	CTS	读/写	0	写“1”禁止 CTS 事件中断。 写, 1: 禁止 CTS 事件中断。 读, 0: CTS 事件中断已禁止。 读, 1: CTS 事件中断已使能。
位 1	NCTS	读/写	0	写“1”禁止 NCTS 事件中断。 写, 1: 禁止 NCTS 事件中断。 读, 0: NCTS 事件中断已禁止。 读, 1: NCTS 事件中断已使能。
位 2	RXDRDY	读/写	0	写“1”禁止 RXDRDY 事件中断。 写, 1: 禁止 RXDRDY 事件中断。 读, 0: RXDRDY 事件中断已禁止。 读, 1: RXDRDY 事件中断已使能。
位 7	TXDRDY	读/写	0	写“1”禁止 TXDRDY 事件中断。 写, 1: 禁止 TXDRDY 事件中断。 读, 0: TXDRDY 事件中断已禁止。 读, 1: TXDRDY 事件中断已使能。
位 9	ERR	读/写	0	写“1”禁止 ERR 事件中断。 写, 1: 禁止 ERR 事件中断。 读, 0: ERR 事件中断已禁止。 读, 1: ERR 事件中断已使能。
位 17	RXTO	读/写	0	写“1”禁止 RXTO 事件中断。

		写, 1: 禁止 RXTO 事件中断。 读, 0: RXTO 事件中断已禁止。 读, 1: RXTO 事件中断已使能。
--	--	---

■ ERRORSRC: 错误原因寄存器

表 10-8: ERRORSRC 寄存器

位	Field	RW	复位值	描述
位 0	OVERRUN	读/写	0	Overrun 错误。接收到新数据的起始位时, 前面的数据仍然在 RXD 中。(前面的数据丢失) 0: 无错误。 1: 有错误。
位 1	PARITY	读/写	0	校验错误, 使能硬件奇偶校验的情况下接收到了含有错误校验位的数据。 0: 无错误。 1: 有错误。
位 2	FRAMING	读/写	0	帧错误 接收到字符中的所有位之后, 在串行数据输入上未检测到有效的停止位。 0: 无错误。 1: 有错误。
位 3	BREAK	读/写	0	串行数据输入为“0”的时长超过一帧数据的时长(无奇偶校验时数据帧长度是 10 位, 有奇偶校验时数据帧长度是 11 位) 0: 无错误。 1: 有错误。

■ ENABLE: UART 使能寄存器

表 10-9: ENABLE 寄存器

位	Field	RW	复位值	描述
位 3~位 0	ENABLE	读/写	0	使能或禁止 UART 0: 禁止 UART。 4: 使能 UART。

■ PSELRTS: RTS 引脚配置寄存器

表 10-10: PSELRTS 寄存器

位	Field	RW	复位值	描述
位 4~位 0	PSELRTS	读/写	0xF	[0~31]: 配置 RTS 信号连接的物理引脚, 0~31 对应引脚 P0.00~P0.31。 0xFFFFFFFF: 不连接。
位 5	PORT	读/写	1	GPIO 端口。0=端口 P0, 1=端口 P1。
位 31	CONNECT	读/写	1	1: RTS 不连接到该引脚。 0: RTS 连接到该引脚。

■ PSELTxD: TXD 引脚配置寄存器

表 10-11: PSELTxD 寄存器

位	Field	RW	复位值	描述
位 4~位 0	PSELTxD	读 / 写	0xF	[0~31]: 配置 TXD 信号连接的物理引脚, 0~31 对应引脚 P0.00~P0.31。 0xFFFFFFFF: 不连接。
位 5	PORT	读 / 写	1	GPIO 端口。0=端口 P0, 1=端口 P1。
位 31	CONNECT	读 / 写	1	1: TXD 不连接到该引脚。 0: TXD 连接到该引脚。

■ PSELCTS: CTS 引脚配置寄存器

表 10-12: PSELCTS 寄存器

位	Field	RW	复位值	描述
位 4~位 0	PSELCTS	读/写	0xF	[0~31]: 配置 CTS 信号连接的物理引脚, 0~31 对应引脚 P0.00~P0.31。 0xFFFFFFFF: 不连接。
位 5	PORT	读/写	1	GPIO 端口。0=端口 P0, 1=端口 P1。
位 31	CONNECT	读/写	1	1: CTS 不连接到该引脚。 0: CTS 连接到该引脚。

■ PSELRXD: RXD 引脚配置寄存器

表 10-13: PSELRXD 寄存器

位	Field	RW	复位值	描述
位 4~位 0	PSELRXD	读 / 写	0xF	[0~31]: 配置 RXD 信号连接的物理引脚, 0~31 对应引脚 P0.00~P0.31。 0xFFFFFFFF: 不连接。
位 5	PORT	读 / 写	1	GPIO 端口。0=端口 P0, 1=端口 P1。
位 31	CONNECT	读 / 写	1	1: RXD 不连接到该引脚。 0: RXD 连接到该引脚。

■ RXD: 数据接收寄存器

表 10-14: RXD 寄存器

位	Field	RW	复位值	描述
位 8~位 0	RXD	只读	0	UART 接收的数据经硬件 FIFO 传送到 RXD, 供 CPU 读出。

■ TXD: 数据接收寄存器

表 10-15: TXD 寄存器

位	Field	RW	复位值	描述
位 8~位 0	TXD	只写	0	UART 数据发送寄存器。

■ BAUDRATE: 波特率配置寄存器

表 10-16: BAUDRATE 寄存器

位	Field	RW	复位值	描述
位 0~位 31	BAUDRATE	只读	0x04000000	波特率。 0x0004F000: 1200bps (实际值: 1205bps)。 0x0009D000: 2400bps (实际值: 2396 bps)。 0x0013B000: 4800bps (实际值: 4808 bps)。 0x00275000: 9600bps (实际值: 9598 bps)。 0x003B0000: 14400bps (实际值: 14414 bps)。 0x004EA000: 19200bps (实际值: 19208 bps)。 0x0075F000: 28800bps (实际值: 28829 bps)。

	0x00800000: 31250bps。
	0x009D5000: 38400bps (实际值: 38462 bps)。
	0x00E50000: 56000bps (实际值: 55944 bps)。
	0x00EBF000: 57600bps (实际值: 57762 bps)。
	0x013A9000 76800bps (实际值: 76923)。
	0x01D7E000: 115200bps (实际值: 115942)。
	0x03AFB000: 230400bps (实际值: 231884)。
	0x04000000: 250000bps。
	0x075F7000: 460800bps (实际值: 470588)。
	0x0EBED000: 921600bps (实际值: 941176)。
	0x10000000: 1Mbps。

■ CONFIG: UART 硬件流控和奇偶校验配置

表 10-17: CONFIG 寄存器

位	Field	RW	复位值	描述
位 0	HWFC	读/写	0	硬件流控 0: 禁止。 1: 使能。
位 3~ 位 1	PARITY	读/写	0	奇偶校验 0x0: 禁止。 0x7: 使能。

5. UARTE 原理

5.1. 功能描述

UARTE 和 UART 共享内存和资源, UARTE 和 UART 的区别是 UARTE 通过是 EasyDMA 进行数据收发, UARTE 的主要特性如下。

- 全双工。
- 自动的硬件流控。
- 奇偶校验并自动产生校验位。
- EasyDMA。
- 串口波特率最高 1 Mbps。
- 支持传输间隔自动进入 IDLE (使用硬件流控时)。
- 1 位/2 位停止位。
- LSB 数据格式。

nRF52840 的 UARTE 的原理框图如下图所示，从图中我们可以看到，和 UART 不同的是 UARTE 通过 EasyDMA 发送和接收数据(下图中红色字体标注的是和 UART 不同的部分，至于引脚配置等都和 UART 一样，参考 UART 章节即可)。

发送的数据事先存放于片内 RAM (发送缓存)，发送时由 EasyDMA 负责从 RAM 数据并传送到物理线路。接收时，数据由物理线路进入硬件 FIFO，之后由 EasyDMA 将数据写入到 RAM。由此可见，UARTE 在接收和发送的过程中，CPU 并没有过多参与，所以使用 UARTE 可有效降低 CPU 的负荷。另外，因为数据均存放于 RAM，所以对于大批量数据传输，UARTE 更适合。

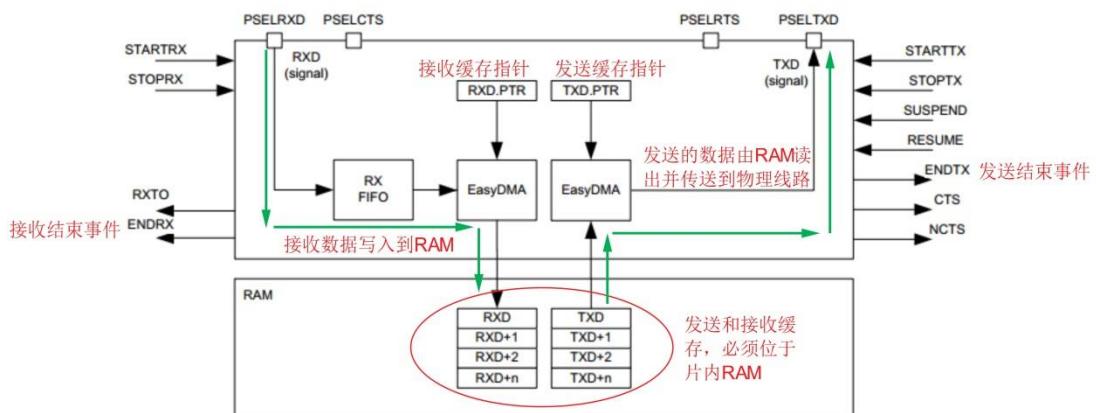


图 10-5: UARTE 原理框图

使用 UARTE 收发数据时，如果发送指针 TXD.PTR 和接收指针 RXD.PTR 没有指向发送缓存或接收缓存，EasyDMA 传输会出现硬件错误或者造成 RAM 中其他数据被破坏。

TXD.PTR、RXD.PTR、TXD.MAXCNT 和 RXD.MAXCNT 这些寄存器是双缓存的，在接收到 RXSTARTED 或 TXSTARTED 事件后即可更新这些寄存器用于下次传输。UARTE 通过 ENDRX 或 ENDTX 事件指示 EasyDMA 已经结束从 RAM 中存取数据。

❖ 注意：使用串口时必须使用外部晶体振荡器，以获取稳定的时钟精度，从而保证通讯稳定。

5.2. UARTE 发送

UARTE 发送时，需要先将待发送的数据写入到发送缓存(发送缓存必须位于片内 RAM)，然后将缓存的地址写入 TXD.PTR 寄存器，并将发送的数据的字节数写入到 TXD.MAXCNT 寄存器，之后触发 STARTTX 任务（即向任务寄存器 STARTTX 中写入 1）启动发送序列。

发送启动后，发送缓存中的数据通过 DMA 传送到 TXD 寄存器中发送，每发送一个字节数据（即一个字节数据由 TXD 寄存器传送到物理线路）产生一次 TXDRDY 事件，当发送数据字节数达到 TXD.MAXCNT 寄存器中定义的字节数时，UARTE 自动停止发送并产生 ENDTX 事件。

如果要停止 UARTE 发送，触发 STOPTX 任务（即向任务寄存器 STOPTX 中写入 1）即可，触发后，UARTE 停止发送并产生 TXSTOPPED 事件。

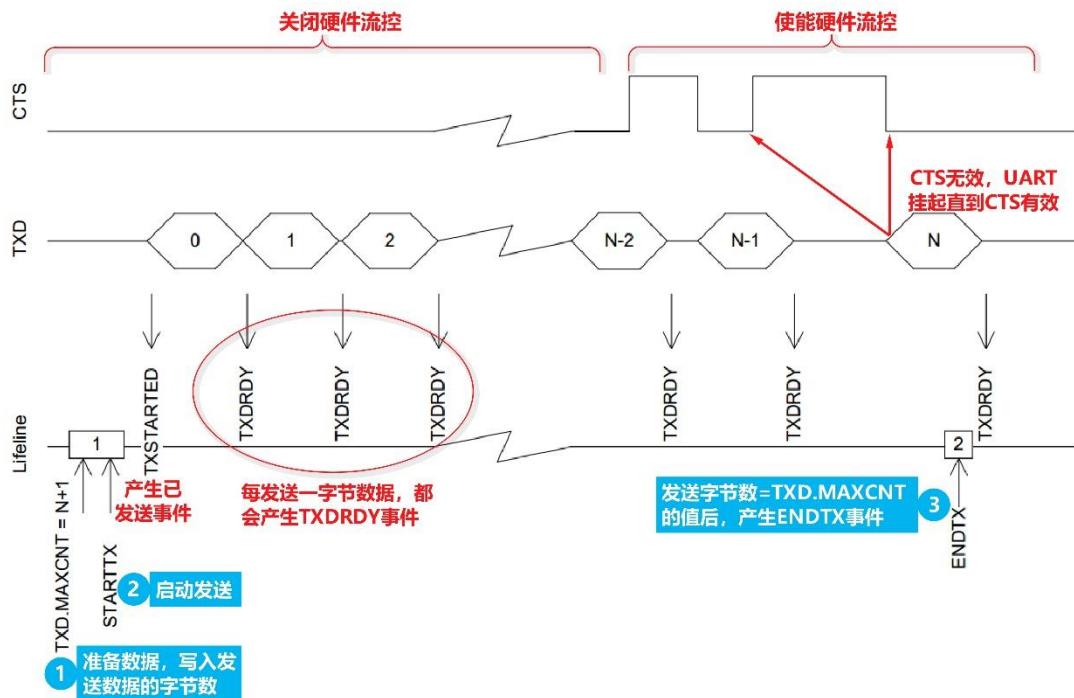


图 10-6: UARTE 发送时序

5.3. UARTE 接收

和 UART一样，UARTE 也是通过触发 STARTRX 任务（即向任务寄存器 STARTRX 中写入 1）启动接收，UARTE 接收时，数据由物理线路进入硬件 FIFO，之后由 EasyDMA 将数据写入到应用程序在 RAM 中定义的接收缓存。接收缓存是应用程序在 RAM 开辟的一段存储空间，EasyDMA 和接收缓存之间是通过 RXD.PTR 寄存器和 RXD.MAXCNT 寄存器联系起来的，换句话说就是 EasyDMA 通过 RXD.PTR 寄存器知道向 RAM 中写入数据的起始地址，通过 RXD.MAXCNT 寄存器知道应该写入多少个字节。

所以，对于应用程序来说，在启动接收前必须要做的是：

- RAM 中开辟一段内存作为接收缓存。
- 将接收缓存的起始地址写入到 RXD.PTR 寄存器。
- 将期望一次接收的字节数写入到 RXD.MAXCNT 寄存器，也就是期望接收多少个字节时产生 ENDRX 事件。

RXD.PTR 和 RXD.MAXCNT 寄存器是双缓存寄存器，在 RXSTARTED 事件产生后即可立刻更新 RXD.PTR 和 RXD.MAXCNT 寄存器中的数值而不会影响当前接收，这样我们就很容易实现数据的持续接收和处理。

UARTE 从 RXD 线路上每接收完一个字节数据，在数据写入到 RAM 前产生一次 RXDRDY 事件。

ENDRX 事件产生后，可以通过 RXD.AMOUNT 寄存器查看从上一次 ENDRX 事件到当前有多少个新数据写入到接收缓存。

5.4. 错误产生条件

UARTE 可以捕获硬件上产生额定错误并触发相应事件，下列两种情形会导致错误事件的产生：

- 结束位没有被正确识别。
- RXD 一直被拉低，并且被拉低的时间超过一帧数据的长度。

错误事件不会停止 UARTE 接收，如果是奇偶校验错误，接收的数据仍然会被存储到数据 RAM，随后的数据也同样会被接收。如果是帧错误（停止位错误），错误数据不会被存储到数据 RAM，随后的数据会正常存储到数据 RAM。

5.5. 低功耗

当系统进入低功耗模式不需要 UARTE 外设时，可以通过停止和禁用 UARTE 外设，从而获得更低的电流消耗。

STOPTX 和 STOPRX 并不是总是需要去执行（因为外设可能已经停止了），但是如果已经触发了 STOPTX 和 STOPRX，软件会一直等到接收到 TXSTOPPED 和/或 RXTO 事件响应后才能通过 ENABLE 寄存器关闭外设。

6. UARTE 寄存器

UARTE 和 UART 共享内存资源，如寄存器等，下表中黑色字体部分是 UARTE 和 UART “共有”的寄存器，这些“共有”的寄存器具有相同的内存地址，UARTE 和 UART 均可使用这些寄存器。红色字体部分是 UARTE 和 UART 各自拥有的寄存器，注意它们共有的部分寄存器有某些位是不一样的，在这里没有列出。

表 10-18：UARTE 相关寄存器

序号	寄存器名	偏移地址	功能描述
任务寄存器			
1	TASKS_STARTRX	0x000	启动 UART 接收
2	TASKS_STOPRX	0x004	停止 UART 接收
3	TASKS_STARTTX	0x008	启动 UART 发送
4	TASKS_STOPTX	0x00C	停止 UART 发送
5	TASKS_FLUSHRX	0x02C	将接收 FIFO 中的数据移入接收缓存。仅 UARTE 具有该任务。
事件寄存器			
1	EVENTS_CTS	0x100	CTS 有效（低电平），允许发送。
2	EVENTS_NCTS	0x104	CTS 无效（高电平），不允许发送。
3	EVENTS_RXDRDY	0x108	从 RXD 线路接收到一个字节数据，但该数据

			可能尚未传输到接收缓存。
4	EVENTS_ENDRX	0x110	接收缓存满，仅 UARTE 具有该寄存器。
5	EVENTS_TXDRDY	0x11C	一字节数据传送至发送线路。
6	EVENTS_ENDTX	0x120	最后一字节数据被发送，仅 UARTE 具有该寄存器。
7	EVENTS_ERROR	0x124	检测到错误。
8	EVENTS_RXTO	0x144	接收超时。
9	EVENTS_RXSTARTE D	0x14C	UART 接收已经启动，仅 UARTE 具有该寄存器。
10	EVENTS_TXSTARTE D	0x150	UART 发送已经启动，仅 UARTE 具有该寄存器。
11	EVENTS_TXSTOPPE D	0x158	发送已停止，仅 UARTE 具有该寄存器。
快捷方式寄存器			
1	SHORTS	0x200	快捷方式。
通用寄存器			
1	INTEN	0x300	使能或禁止中断，仅 UARTE 具有该寄存器。
2	INTENSET	0x304	中断使能。
3	INTENCLR	0x308	中断禁止。
4	ERRORSRC	0x480	错误原因。
5	ENABLE	0x500	使能 UART。
6	PSELRTS	0x508	RTS 引脚配置。
7	PSELTXD	0x50C	TXD 引脚配置。
8	PSELCTS	0x510	CTS 引脚配置。
9	PSELRXD	0x514	RXD 引脚配置。
10	BAUDRATE	0x524	UART 波特率配置。
11	RXD.PTR	0x534	接收数据指针。
12	RXD.MAXCNT	0x538	接收缓存的最大字节数。

13	RXD.AMOUNT	0x53C	最后一次传送的字节数。
14	TXD.PTR	0x544	发送数据指针。
15	TXD.MAXCNT	0x548	发送缓存的最大字节数。
16	TXD.AMOUNT	0x54C	最后一次传送的字节数。
17	CONFIG	0x56C	UART 硬件流控和奇偶校验配置。

下面的寄存器描述仅描述了 UARTE 和 UART 不一样的寄存器，其他的寄存器和 UART 一样，参考 UART 的寄存器描述。

■ INTEN：中断使能/禁止寄存器

表 10-19: INTEN 寄存器

位	Field	RW	复位值	描述
位 0	CTS	读/写	0	使能或禁止 CTS 事件中断。 0: 使能。 1: 禁止。
位 1	NCTS	读/写	0	使能或禁止 NCTS 事件中断。 0: 使能。 1: 禁止。
位 2	RXDRDY	读/写	0	使能或禁止 RXDRDY 事件中断。 0: 使能。 1: 禁止。
位 4	ENDRX	读/写	0	使能或禁止 ENDRX 事件中断。 0: 使能。 1: 禁止。
位 7	TXDRDY	读/写	0	使能或禁止 TXDRDY 事件中断。 0: 使能。 1: 禁止。
位 8	ENDTX	读/写	0	使能或禁止 ENDTX 事件中断。 0: 使能。 1: 禁止。
位 9	ERR	读/写	0	使能或禁止 ERR 事件中断。 0: 使能。 1: 禁止。
位 17	RXTO	读/写	0	使能或禁止 RXTO 事件中断。

				0: 使能。 1: 禁止。
位 19	RXSTARTED	读/写	0	使能或禁止 RXSTARTED 事件中断。 0: 使能。 1: 禁止。
位 20	TXSTARTED	读/写	0	使能或禁止 TXSTARTED 事件中断。 0: 使能。 1: 禁止。
位 22	TXSTOPPED	读/写	0	使能或禁止 TXSTOPPED 事件中断。 0: 使能。 1: 禁止。

■ RXD.PTR: 接收数据指针寄存器

表 10-20: RXD.PTR 寄存器

位	Field	RW	复位值	描述
位 31~	PTR	读/写	0	数据指针。
位 0				

■ RXD.MAXCNT: 接收缓存的最大字节数配置寄存器

表 10-21: RXD.MAXCNT 寄存器

位	Field	RW	复位值	描述
位 16~	MAXCNT	读/写	0	接收缓存的最大字节数。
位 0				

■ RXD.AMOUNT: 最后一次传送的字节数查询寄存器

表 10-22: RXD.AMOUNT 寄存器

位 16~	Field	RW	复位值	描述
位 0				
位 16~	AMOUNT	只读	0	最后一次传送的字节数。
位 0				

- TXD.PTR: 发送数据指针寄存器

表 10-23: TXD.PTR 寄存器

位	Field	RW	复位值	描述
位 31~	PTR	读/写	0	数据指针。
位 0				

- TXD.MAXCNT: 接收缓存的最大字节数配置寄存器

表 10-5: TXD.MAXCNT 寄存器

位	Field	RW	复位值	描述
位 16~	MAXCNT	读/写	0	发送缓存的最大字节数。
位 0				

- RXD.AMOUNT: 最后一次传送的字节数查询寄存器

表 10-24: RXD.AMOUNT 寄存器

位	Field	RW	复位值	描述
位 16~	AMOUNT	只读	0	最后一次传送的字节数。
位 0				

7. 软件设计

7.1. 库函数的应用

Nordic 的库中，UART 和 UARTE 公用一个程序模块，称为 APP UART，APP UART 中封装了对 UART 和 UARTE 操作，库函数中通过条件编译区分应用程序使用 UART 和 UARTE。下文中，为了方便描述，我们将 UART 和 UARTE 统称为串口。

7.1.1. 串口初始化

SDK 中定义了一个专门用于初始化串口的宏：APP_UART_FIFO_INIT，该宏展开后如下：

代码清单：APP_UART_FIFO_INIT 宏

```

1. #define APP_UART_FIFO_INIT(P_COMM_PARAMS, RX_BUF_SIZE, TX_BUF_SIZE, EVT_HANDLER, IRQ_PRIO,
                           ERR_CODE) \
2.   do \
3.   { \
4.     app_uart_buffers_t buffers; \
5.     static uint8_t      rx_buf[RX_BUF_SIZE]; \

```

```

6.     static uint8_t      tx_buf[TX_BUF_SIZE];      \
7.                                         \
8.     buffers.rx_buf      = rx_buf;                  \
9.     buffers.rx_buf_size = sizeof (rx_buf);          \
10.    buffers.tx_buf      = tx_buf;                  \
11.    buffers.tx_buf_size = sizeof (tx_buf);          \
12.    ERR_CODE = app_uart_init(P_COMM_PARAMS, &buffers, EVT_HANDLER, IRQ_PRIO); \
13. } while (0)

```

APP_UART_FIFO_INIT 宏封装了 app_uart_init() 函数，同时它也定义了串口接收和发送的缓存，串口接收和发送缓存的大小由应用程序配置。

- 1) 行 4：定义 app_uart_buffers_t 结构体变量 buffers，buffers 用来保存发送和接收的数据。
app_uart_buffers_t 结构体声明如下：

代码清单：app_uart_buffers_t 结构体

```

1. typedef struct
2. {
3.     uint8_t * rx_buf;        //指向接收缓存
4.     uint32_t rx_buf_size;   //接收缓存大小
5.     uint8_t * tx_buf;        //指向发送缓存
6.     uint32_t tx_buf_size;   //发送缓存大小
7. } app_uart_buffers_t;

```

- 2) 行 5：接收缓存数组，串口接收的数据实际的存放位置。
- 3) 行 6：发送缓存数组，串口发送的数据实际的存放位置。
- 4) 行 8~行 11：buffers 结构体成员变量赋值，即将发送和接收数组的首地址以及数组的大小赋值给 buffers 结构体相应的成员变量。
- 5) 行 12：调用库函数 app_uart_init() 初始化串口。

APP_UART_FIFO_INIT 宏是一个带参数宏，共有 5 个输入参数和 1 个输出参数，应用程序调用该宏时需要提供如下的 5 个输入参数，同时，应用程序可以通过输出参数获取宏执行的结果，即错误代码。

- 输入参数 1：P_COMM_PARAMS：指向串口的通讯参数配置结构体。

应用程序需要定义一个 app_uart_comm_params_t 结构体变量，并初始化该结构体变量的成员变量，即配置串口通讯参数，app_uart_comm_params_t 结构体的声明如下：

代码清单：app_uart_comm_params_t 结构体

```

1. typedef struct
2. {
3.     uint32_t    rx_pin_no;    //RxD 信号连接的引脚
4.     uint32_t    tx_pin_no;    //TxD 信号连接的引脚

```

```

5.     uint32_t      rts_pin_no;    //RTS 信号连接的引脚，仅在硬件流控使能时有效
6.     uint32_t      cts_pin_no;    //CTS 信号连接的引脚，仅在硬件流控使能时有效
7.     app_uart_flow_control_t flow_control; //硬件流控配置：使能或禁止
8.     bool          use_parity;   //奇偶校验。false：禁止奇偶检验，true：使能奇偶检验
9.     uint32_t      baud_rate;    //波特率配置
10. } app_uart_comm_params_t;

```

- 输入参数 2: RX_BUF_SIZE: 接收缓存大小。
- 输入参数 3: TX_BUF_SIZE: 发送缓存大小。
- 输入参数 4: EVT_HANDLER: 串口事件句柄。
- 输入参数 5: IRQ_PRIO: 中断优先级。
- 输出参数 ERR_CODE: 错误代码，用于向应用程序反馈串口初始化的结果。

APP_UART_FIFO_INIT 宏最终调用 app_uart_init() 函数初始化串口，app_uart_init() 函数原型如下：

表 10-25: app_uart_init ()函数

函数原型	<code>uint32_t app_uart_init (const app_uart_comm_params_t * p_comm_params, app_uart_buffers_t * p_buffers, app_uart_event_handler_t error_handler, app_irq_priority_t irq_priority)</code>
函数功能	初始化串口程序模块。
参 数	<p>[in] p_comm_params: 指向串口通讯参数配置结构体。</p> <p>[in] p_buffers: 指向发送和接收缓存，设置为 NULL 表示不使用缓存：</p> <p>[in] error_handler: 串口事件回调函数。</p> <p>[in] irq_priority: 串口中断优先级</p>
返回值	<ul style="list-style-type: none"> • NRF_SUCCESS: 初始化成功。 • NRF_ERROR_INVALID_LENGTH: 提供的接收或发送缓存的大小不是 2 的整数倍。 • NRF_ERROR_NULL: 接收或发送缓存中的一个为 NULL。 ◆ 注意：下面的错误是在使能硬件流控的情况下，初始化串口模块时可能会返回的错误代码。如果没有使能硬件流控，下面的错误永远不会发生。 • NRF_ERROR_INVALID_STATE: 注册串口程序模块时，GPIOE 模块状态无效。 • NRF_ERROR_INVALID_PARAM: 注册串口程序模块时，提供了无效的回调函数。

- NRF_ERROR_NO_MEM: GPIO模块没有空闲的通道可以使用。

7.1.2. 串口事件处理

初始化串口时，应用程序需要提供串口事件回调函数，串口事件回调函数在串口初始化时注册，当串口事件产生时，串口程序模块会调用串口事件回调函数，在串口事件回调函数即可处理串口的事件。APP UART 定义了 5 种串口事件，它们定义在枚举变量 app_uart_evt_type_t 中，如下所示：

代码清单：app_uart_evt_type_t 结构体声明

```

1. typedef enum
2. {
3.     //串口接收到数据，数据已经存入软件 FIFO 并可以通过 ref app_uart_get 函数读取数据
4.     APP_UART_DATA_READY,
5.     //串口软件 FIFO 程序模块发生错误，错误代码存在 app_uart_evt_t.data.error_code
6.     APP_UART_FIFO_ERROR,
7.     //串口通讯错误，错误代码存在 app_uart_evt_t.data.error_communication
8.     APP_UART_COMMUNICATION_ERROR,
9.     APP_UART_TX_EMPTY,           //发送 FIFO 中的数据已经全部发送完
10.    APP_UART_DATA,             //串口接收到数据，该事件仅用于不使用软件 FIFO
11. } app_uart_evt_type_t;

```

◆ 注意：APP_UART_DATA 事件仅在不使用软件 FIFO 的情况下使用，若使用了软件 FIFO，该事件永远不会发生。

串口事件回调函数的编写需要遵循相应的格式，串口事件回调函数的格式如下：

```

void 函数名称(app_uart_evt_t * p_event)
{
    //这里写事件处理代码。
    //示例：判断串口接收事件
    if (p_event->evt_type == APP_UART_DATA_READY)
    {
        //功能代码，如保存数据等。
    }
}

```

7.1.3. 启用 UARTE

串口使用 UART 还是 UARTE，是在“sdk_config.h”文件中配置，如下图所示，当我们勾选“NRFX_UARTE_ENABLED”后，启用 UARTE。

这里实际就是使用了条件编译，当宏定义“NRFX_UARTE_ENABLED”有效时，串口程序模块通过条件编译语句会将串口配置为 UARTE，从而程序编译后，执行的是 UARTE

的功能，反之亦然。

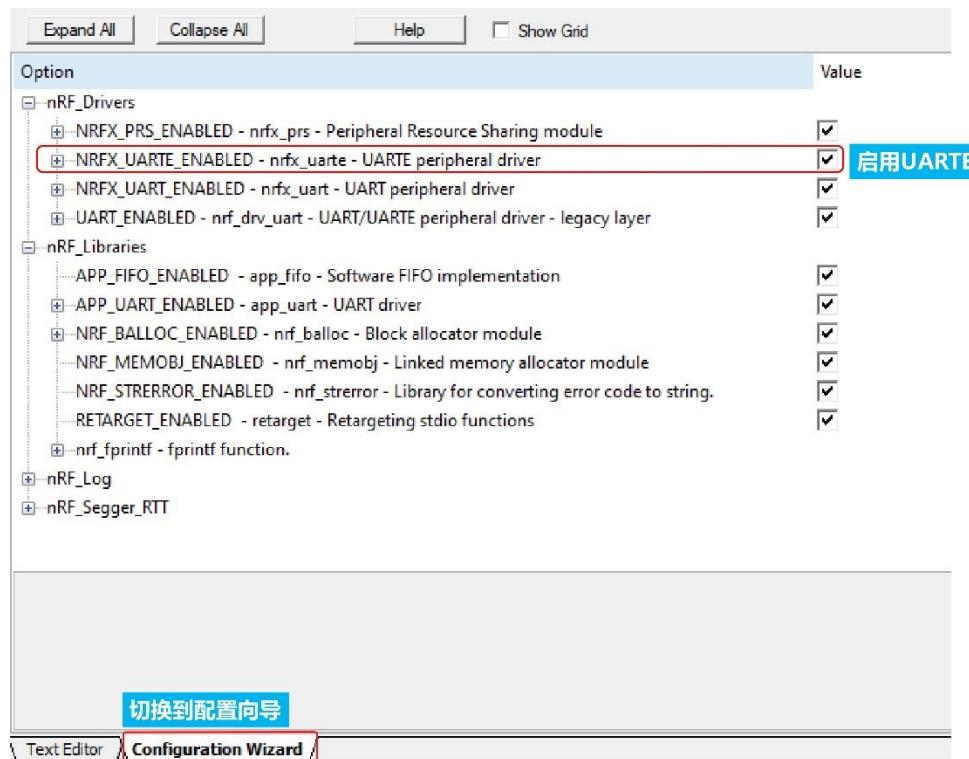


图 10-7：“sdk_config.h”文件中配置是否使用 UARTE

7.1.4. 发送和接收数据

APP UART 通过 `app_uart_put()` 函数发送数据，通过 `app_uart_get()` 函数接收数据，这两个函数都是单字节操作的，每次只能处理一个字节，函数原型如下表所示。:

表 10-26: `app_uart_put()` 函数

函数原型	<code>uint32_t app_uart_put(uint8_t byte)</code>
函数功能	串口发送一个字节，该函数是非堵塞的，它仅将数据写入到发送缓存，而不是等待数据发送完成。也就是当调用该函数时返回 <code>NRF_SUCCESS</code> 表示待发送的数据成功写入到发送缓存，而不是表示数据已经成功发送。
参 数	[in] <code>byte</code> : 待发送的 1 字节数据。
返回值	<ul style="list-style-type: none"> <code>NRF_SUCCESS</code>: 待发送的 1 字节数据成功写入发送缓存。 <code>NRF_ERROR_NO_MEM</code>: 发送缓存中没有剩余空间来存放待发送的数据。在硬件流控开启的情况下，当 CTS 信号无效的时候过长导致发送缓存满时也会返回 <code>NRF_ERROR_NO_MEM</code>。 <code>NRF_ERROR_INTERNAL</code>: 串口驱动上报错误。

表 10-27: `app_uart_get()` 函数

函数原型	<code>uint32_t app_uart_get(uint8_t * p_byte)</code>
函数功能	串口从接收缓存中读取一个字节数据。如果接收缓存为空，返回错误代

	码 NRF_ERROR_NOT_FOUND，另外，app_uart 每次向接收缓存中添加第一个字节数据时都会产生事件。
参 数	[in] p_byte: 指向保存读出的数据的变量地址。
返回值	<ul style="list-style-type: none"> NRF_SUCCESS 读出一个字节数据并保存到 p_byte 指向的地址。 NRF_ERROR_NOT_FOUND: 接收缓存中没有可读取的数据。

7.2. 串口数据收发实验

使用 APP UART，需要在工程中加入相关的文件以及引用头文件和路径配置等，接下来，我们在“实验 5-3：流水灯(BSP 实现方式)”工程的基础上加入 APP UART 程序模块并实现数据的收发。

❖ 注：本节对应的实验源码是：“实验 10-1：串口数据收发（查询方式）”。

7.2.1. 添加需要的文件

APP UART 需要加入的文件如下表所示。

表 10-28: APP UART 需要加入的文件

文件名	SDK 中的目录	描述
nrf_drv_uart.c	..\..\integration\nrfx\legacy\	旧 UART 驱动程序。
nrfx_uart.c	..\..\modules\nrfx\drivers\src\	新 UART 驱动程序。
nrfx_uarte.c	..\..\modules\nrfx\drivers\src\	新 UART 驱动程序。
app_fifo.c	..\..\components\libraries\fifo\	软件 FIFO 程序模块。
app_uart_fifo.c	..\..\components\libraries\uart\	使用软件 FIFO 的串口库文件。
nrfx_prs.c	..\..\modules\nrfx\drivers\src\prs\	外设资源共享程序模块。
retarget.c	..\..\components\libraries\uart\	串口重定向文件，加入该文件后即可使用 printf 函数。

7.2.2. 头文件引用和路径设置

■ 需要引用的头文件

因为在“main.c”文件中使用了 APP UART，所以需要引用下面的头文件。

代码清单：接收 UART 需要引用的头文件

```

1. #include "app_uart.h"
2. #if defined (UART_PRESENT)
3. #include "nrf_uart.h"
4. #endif
5. #if defined (UARTE_PRESENT)
6. #include "nrf_uarte.h"

```

7. #endif

■ 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

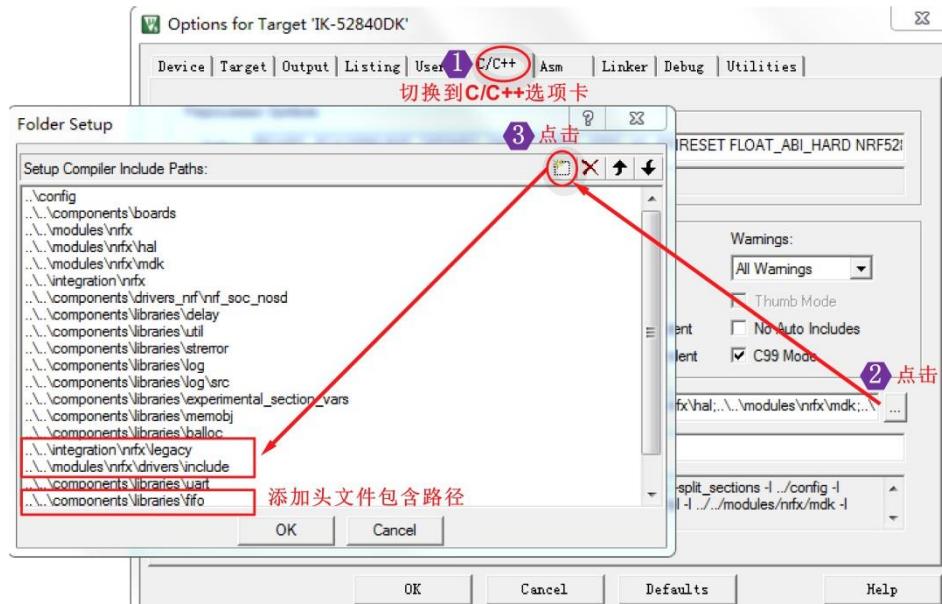


图 10-8：添加头文件包含路径

APP UART 需要添加的头文件路径如下表：

表 10-29：头文件包含路径

序号	路径
1	..\\..\\integration\\nrfx\\legacy
2	..\\..\\modules\\nrfx\\drivers\\include
3	..\\..\\components\\libraries\\uart
4	..\\..\\components\\libraries\\fifo

7.2.3. 工程配置

打开“sdk_config.h”文件，加入串口配置，如下图所示，编辑内容时切换到“Text Editor”进行编辑(编辑的内容参考实验的源码，因此代码很长，此处不贴出代码，参见“sdk_config.h”文件的(52~490)行、(498~518)行和(588~593))，编辑完成后，切换到“Configuration Wizard”，即可观察到配置的具体项目。

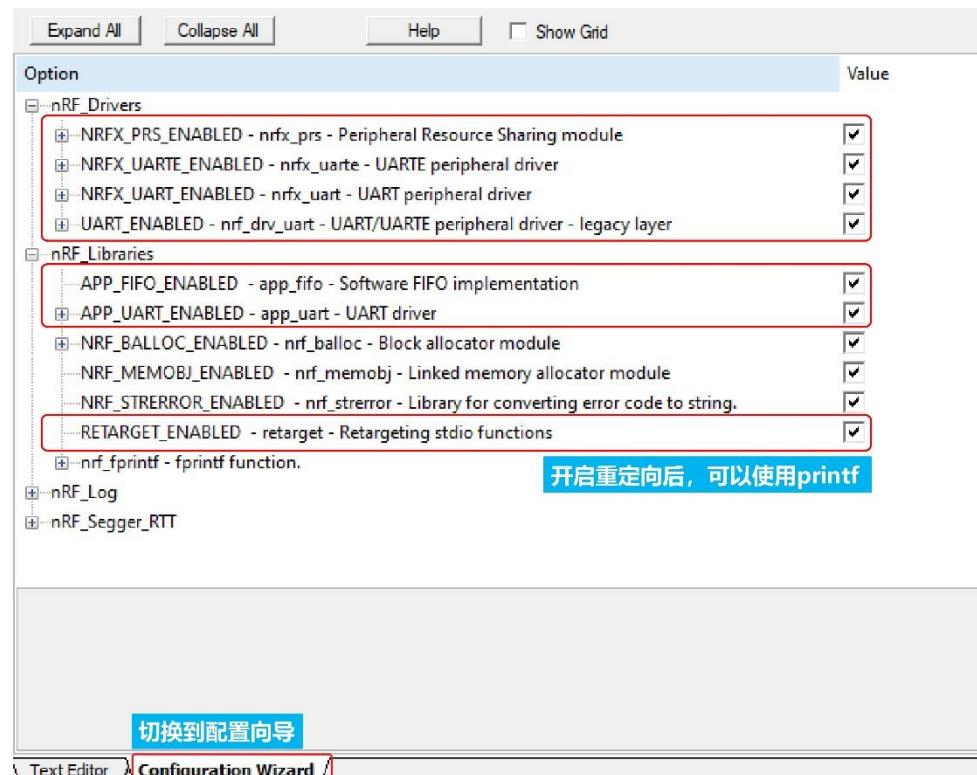


图 10-1：工程配置

7.2.4. 代码编写

根据上文描述，我们使用 APP UART 编写串口收发程序的流程是：“定义串口通讯参数配置结构体并设置参数→初始化串口→“sdk_config.h”文件中指定使用 UART 还是 UARTE→串口数据收发”，例子的程序清单如下。

代码清单：接收和发送缓存大小定义和串口事件回调函数

```

1. #define UART_TX_BUF_SIZE 256          //串口发送缓存大小(字节数)
2. #define UART_RX_BUF_SIZE 256          //串口接收缓存大小(字节数)
3.
4. //串口事件回调函数, 该函数中判断事件类型并进行处理
5. void uart_error_handle(app_uart_evt_t * p_event)
6. {
7.     //通讯错误事件
8.     if (p_event->evt_type == APP_UART_COMMUNICATION_ERROR)
9.     {
10.         APP_ERROR_HANDLER(p_event->data.error_communication);
11.     }
12.     //FIFO 错误事件
13.     else if (p_event->evt_type == APP_UART_FIFO_ERROR)
14.     {
15.         APP_ERROR_HANDLER(p_event->data.error_code);
16.     }
17. }
```

❖ 注意：串口事件回调函数 `uart_error_handle` 里面有个 `err`，并不表示这个事件回调函数只能处理错误事件，而是因为这个例子里面只需要处理错误事件，所以回调函数的名字取为 `uart_error_handle`。

代码清单：串口初始化配置

```
1. //串口配置
2. void uart_config(void)
3. {
4.     uint32_t err_code;
5.
6.     //定义串口通讯参数配置结构体并初始化
7.     const app_uart_comm_params_t comm_params =
8.     {
9.         RX_PIN_NUMBER, //定义 uart 接收引脚
10.        TX_PIN_NUMBER, //定义 uart 发送引脚
11.        //定义 uart RTS 引脚，流控关闭后虽然定义了 RTS 和 CTS 引脚，但是驱动程序会忽略，不会配
12.        //置这两个引脚，两个引脚仍可作为 IO 使用
13.        RTS_PIN_NUMBER,
14.        CTS_PIN_NUMBER, //定义 uart CTS 引脚
15.        APP_UART_FLOW_CONTROL_DISABLED, //关闭 uart 硬件流控
16.        false, //禁止奇偶检验
17.        NRF_UART_BAUDRATE_115200 //uart 波特率设置为 115200bps
18.    };
19.    //初始化串口，注册串口事件回调函数
20.    APP_UART_FIFO_INIT(&comm_params,
21.                        UART_RX_BUF_SIZE,
22.                        UART_TX_BUF_SIZE,
23.                        uart_error_handle,
24.                        APP_IRQ_PRIORITY_LOWEST,
25.                        err_code);
26.
27.    APP_ERROR_CHECK(err_code);
28. }
```

初始化配置后，即可进行数据收发，本例是在主循环中查询 FIFO 是否有接收到的数据，如果有数据，将数据原样通过串口输出。因为配置了串口重定向（工程中加入了“retarget.c”文件，“ `sdk_config.h`”文件中也开启了 `RETARGET_ENABLED`），所以也可以使用 `printf` 输出数据。

这里说的查询是指查询串口接收软件 FIFO，实际串口数据接收是中断接收的，接收后的数据会存入到软件 FIFO。

代码清单：串口数据收发

```
1. ****
```

```
2. * 描述 : main 函数
3. * 入参 : 无
4. * 返回值 : int 类型
5. ****
6. int main(void)
7. {
8.     //配置 UICR_REGOUT0 寄存器, 设置 GPIO 输出 3.3V 电压
9.     gpio_output_voltage_setup_3v3();
10.
11.    //初始化 log 程序模块, 本例中使用 RTT 作为输出终端打印信息
12.    log_init();
13.
14.    //初始化开发板上的 4 个 LED, 即将驱动 LED 的 GPIO 配置为输出,
15.    bsp_board_init(BSP_INIT_LEDS);
16.    //初始化串口
17.    uart_config();
18.
19.    //LOG 打印启动信息
20.    NRF_LOG_INFO("uart echo example started");
21.    NRF_LOG_FLUSH();
22.
23.    while(true)
24.    {
25.        uint8_t cr;
26.        //查询是否接收到数据
27.        while (app_uart_get(&cr) != NRF_SUCCESS);
28.        //将接收的数据原样发回
29.        while (app_uart_put(cr) != NRF_SUCCESS);
30.
31.        //使用 printf 发数据
32.        printf("%c", cr);
33.    }
34. }
```

7.2.5. 硬件连接

本实验需要使用 P0.13~P0.16 驱动 LED 指示灯 D1~D4, P0.06 和 P0.08 作为串口通讯引脚, 按照下图所示短接跳线帽。

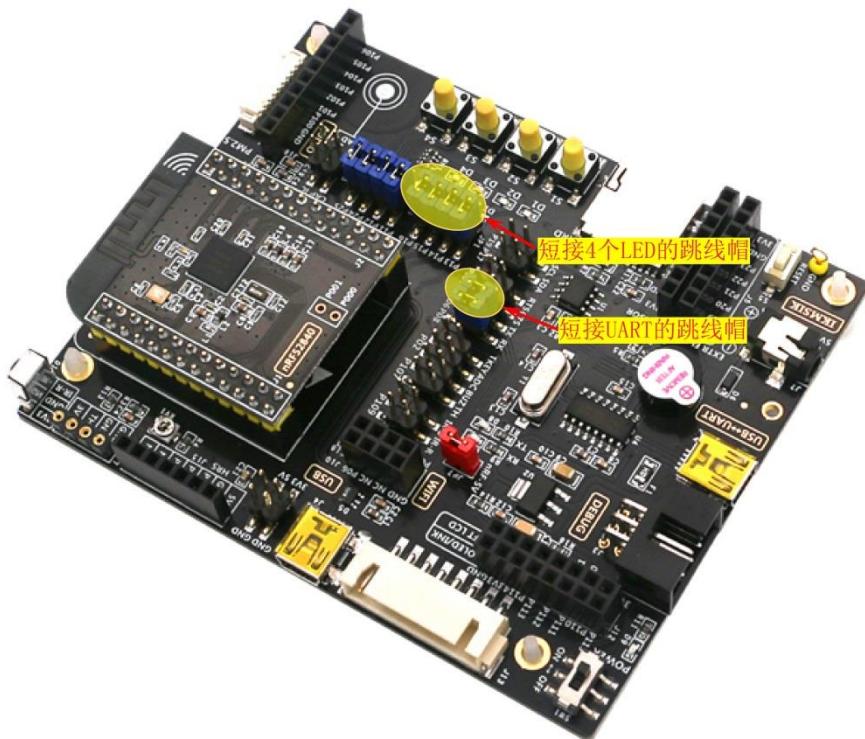


图 10-9：开发板跳线帽短接

7.2.6. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 10-1：串口数据收发（查询方式）”，将解压后得到的文件夹“uart_echo”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 Keil MDK5。
3. 在 MDK5 中执行“Project→Open Project” 打开“…\uart_echo\project\mdk5” 目录下的工程“uart_echo.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，通过串口调试助手发送数据，开发板接收到数据后会原样返回数据。串口调试助手波特率设置为 115200bps。

7.3. 串口发送和接收事件实验

本实验在“实验 10-1：串口数据收发（查询方式）”的基础上修改，在串口事件回调函数里面增加了对 APP_UART_TX_EMPTY 和 APP_UART_DATA_READY 的处理。

❖ 注：本节对应的实验源码是：“实验 10-2：串口发送和接收事件”。

7.3.1. 代码编写

上一个实验是在主循环中查询串口接收 FIFO 获取接收的数据，本实验在串口事件回调函数中直接判断事件类型，在 APP_UART_DATA_READY 事件中接收数据。

我们在串口事件回调函数中增加判断 APP_UART_TX_EMPTY 事件和 APP_UART_DATA_READY 事件的代码，当串口接收到数据时，产生 APP_UART_DATA_READY 事件，翻转指示灯 D1 的状态，即通过 D1 指示接收事件。当串口发送数据完成时，即软件 FIFO 为空时，产生 APP_UART_TX_EMPTY 事件，翻转指示灯 D2 的状态，即通过 D2 指示发送完成事件。这样，通过指示灯 D1 和 D2 的状态即可观察串口发送和接收事件的产生。

程序清单：串口事件回调函数

```
1. //串口事件回调函数，该函数中判断事件类型并进行处理
2. void uart_error_handle(app_uart_evt_t * p_event)
3. {
4.     uint8_t cr;
5.
6.     //通讯错误事件
7.     if (p_event->evt_type == APP_UART_COMMUNICATION_ERROR)
8.    {
9.        APP_ERROR_HANDLER(p_event->data.error_communication);
10.    }
11.    //FIFO 错误事件
12.    else if (p_event->evt_type == APP_UART_FIFO_ERROR)
13.    {
14.        APP_ERROR_HANDLER(p_event->data.error_code);
15.    }
16.    //串口接收事件
17.    else if (p_event->evt_type == APP_UART_DATA_READY)
18.    {
19.        //翻转指示灯 D1 状态，指示串口接收事件
20.        nrf_gpio_pin_toggle(LED_1);
21.        //从 FIFO 中读取数据
22.        app_uart_get(&cr);
23.        //串口输出数据
24.        printf("%c",cr);
25.    }
26.    //串口发送完成事件
27.    else if (p_event->evt_type == APP_UART_TX_EMPTY)
28.    {
29.        //翻转指示灯 D2 状态，指示串口发送完成事件
30.        nrf_gpio_pin_toggle(LED_2);
31.    }
}
```

32. }

7.3.2. 硬件连接

同“实验 10-1：串口数据收发（查询方式）”实验。

7.3.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 8-2：串口发送和接收事件”，将解压后得到的文件夹“uart_event”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\uart_event\project\mdk5”目录下的工程“uart_event.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，通过串口调试助手向开发板发送数据，开发板会返回数据，同时开发板上的指示灯 D1 在接收数据（APP_UART_DATA_READY 事件）时翻转状态，指示灯 D2 在发送数据（APP_UART_TX_EMPTY 事件）时翻转状态。

7.4. 串口控制 LED 指示灯实验

本实验在“实验 10-1：串口数据收发（查询方式）”的基础上修改。前面 2 个实验中我们学习了串口数据的收发和事件，这一节我们来看一下如何解析简单命令。

我们定义的 LED 指示灯控制的命令格式如下：

- 指示灯控制命令格式：“D’ + 指示灯标号（1、2、3、4）+ ‘#’。
- 命令示例：控制指示灯 D1 点亮“D1#”。

开发板接收到一个命令包后，对命令包进行解析，并根据解析结果点亮相应的 LED 指示灯。

◆ 注：本节对应的实验源码是：“实验 10-3：串口控制 LED 亮灭”。

7.4.1. 代码编写

程序清单：串口接收命令控制 LED 亮灭

```
1. int main(void)  
2. {
```

```
3.     uint32_t err_code;
4.     uint8_t RxCnt = 0;           //UART 接收字节数
5.     uint8_t UartRxBuf[RXBUFF_LEN]; //UART 接收缓存
6.     uint8_t cr;   //定义一个变量保存接收到的数据
7.
8.     //初始化开发板上的 4 个 LED，即将驱动 LED 的 GPIO 配置为输出,
9.     bsp_board_init(BSP_INIT_LEDS);
10.    //定义串口通讯参数配置结构体并初始化
11.    const app_uart_comm_params_t comm_params =
12.    {
13.        RX_PIN_NUMBER, //定义 uart 接收引脚
14.        TX_PIN_NUMBER, //定义 uart 发送引脚
15.        //定义 uart RTS 引脚, 注意流控关闭后虽然定义了 RTS 和 CTS 引脚, 但是不起作用
16.        RTS_PIN_NUMBER,
17.        CTS_PIN_NUMBER, //定义 uart CTS 引脚
18.        APP_UART_FLOW_CONTROL_DISABLED, //关闭 uart 硬件流控
19.        false, //禁止奇偶检验
20.        NRF_UART_BAUDRATE_115200 //uart 波特率设置为 115200bps
21.    };
22.    //初始化串口, 注册串口事件回调函数
23.    APP_UART_FIFO_INIT(&comm_params,
24.                        UART_RX_BUF_SIZE,
25.                        UART_TX_BUF_SIZE,
26.                        uart_error_handle,
27.                        APP_IRQ_PRIORITY_LOWEST,
28.                        err_code);
29.    //检查串口初始化返回的错误代码
30.    APP_ERROR_CHECK(err_code);
31.
32.    while(true)
33.    {
34.        //查询是否接收到数据
35.        while(app_uart_get(&cr) != NRF_SUCCESS);
36.
37.        //数据长度小于 3, 并且不是结束符, 保存接收的数据
38.        if((cr != '#') && (RxCnt < 3))
39.        {
40.            UartRxBuf[RxCnt++] = cr;
41.        }
42.        else
43.        {
44.            //如果接收数据长度大于 3 个字节, 表示接收出错, 清零接收计数
45.            if(RxCnt >= 3)
46.            {
```

```
47.         RxCnt = 0;
48.     }
49.     else
50.     {
51.         //检查数据是否合法
52.         if((UartRxBuf[0] == 'D') || (UartRxBuf[0] == 'd'))
53.         {
54.             //减去 48, 得到 ASCII 对应的十进制数值
55.             switch(UartRxBuf[1]-48)
56.             {
57.                 case 1:
58.                     LEDS_OFF(LEDS_MASK);           //熄灭所有 LED 指示灯
59.                     nrf_gpio_pin_clear(LED_1); //点亮 D1 指示灯
60.                     RxCnt = 0;                  //串口接收计数清零
61.                     break;
62.
63.                 case 2:
64.                     LEDS_OFF(LEDS_MASK);           //熄灭所有 LED 指示灯
65.                     nrf_gpio_pin_clear(LED_2); //点亮 D2 指示灯
66.                     RxCnt = 0;                  //串口接收计数清零
67.                     break;
68.
69.                 case 3:
70.                     LEDS_OFF(LEDS_MASK);           //熄灭所有 LED 指示灯
71.                     nrf_gpio_pin_clear(LED_3); //点亮 D3 指示灯
72.                     RxCnt = 0;                  //串口接收计数清零
73.                     break;
74.
75.                 case 4:
76.                     LEDS_OFF(LEDS_MASK);           //熄灭所有 LED 指示灯
77.                     nrf_gpio_pin_clear(LED_4); //点亮 D4 指示灯
78.                     RxCnt = 0;                  //串口接收计数清零
79.                     break;
80.
81.             default:
82.                 break;
83.             }
84.         }
85.     }
86. }
87. }
88. }
```

7.4.2. 硬件连接

同“实验 8-1：串口数据收发”实验。

7.4.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 10-3：串口控制 LED 亮灭”，将解压后得到的文件夹“uart_led”拷贝到合适的目录，如“D\ nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\uart_led\project\mdk5”目录下的工程“uart_led.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，串口调试助手发送栏输入“D1#”、“D2#”、“D3#”或“D4#”，点击发送按钮发送数据，观察开发板上的指示灯的亮灭，应和发送的指示灯编号一致。

第十一章：GPIO 任务和事件(GPIOTE)

1. 学习目的

1. 学习 GPIOTE (GPIO 任务和事件) 的原理和用途。
2. 掌握 GPIOTE 库函数的应用，熟悉 GPIOTE 常用的库函数。
3. 掌握 GPIOTE 输出的配置，以及通过任务触发引脚输出高电平、低电平和翻转状态。
清楚使用 GPIOTE 程序模块时，什么情况下通过 GPIOTE 操作引脚，什么情况下通过 GPIO 操作引脚。
4. 掌握 GPIOTE 输入的配置，以及检测引脚状态变化并产生事件。
5. 掌握 GPIOTE PORT 的配置，使用 GPIOTE PORT 检测多个引脚的状态变化以及 PORT 事件产生时如何获取产生事件的引脚号。

2. GPIOTE 原理

2.1. GPIOTE 功能

在我们学习 GPIO 的时候，我们会发现 nRF52840 的 GPIO 只能作为通用的输入输出使用，它作为输入时是无法产生中断的，那么，如果我们需要使用引脚输入产生中断的功能，该如何实现？

这时，我们需要用到 nRF52840 的 GPIOTE 外设。Nordic 的 nRF51 和 nRF52 系列芯片在 GPIO 的基础上引入了任务和事件 (GPIOTE)，可以通过任务和事件来操作 IO，任务是针对输出的，它可以让 IO 输出执行不同的动作，事件是针对输入的，IO 状态的改变会置位事件寄存器，从而产生事件中断。应用程序可以配置中断使能寄存器，使能或关闭事件中断。GPIOTE 能让我们更方便地去操作 IO，同时，他还可以和 PPI 结合，有效减少 CPU 的参与、降低 CPU 的负担。

nRF52840 的 GPIOTE 共有 8 个通道，每个通道都可以分配给一个引脚，分配的引脚可以配置为任务模式或事件模式。需要注意的是：不能将某个引脚同时分配给多个 GPIOTE 通道，否则会导致无法预料的错误。

■ GPIOTE 每个通道都有 SET、CLR 和 OUT 任务，其中 SET 和 CLR 任务让引脚执行固定的操作，即触发 SET 任务引脚输出高电平，触发 CLR 任务引脚输出低电平，OUT 任务可以通过配置让引脚执行下面 3 种操作：

- 置位（输出高电平）。
- 清除（输出低电平）。
- 翻转。

■ GPIOTE 每个通道的事件都可以配置为由以下的 3 种输入状态中的一种来产生：

- 上升沿。

- 下降沿。
- 任意电平跳变。

2.2. 引脚任务和事件

任务和事件通过 CONFIG[n](n=0~7) 寄存器配置, 每个 CONFIG[n]寄存器可配置一个对应编号的 OUT[n] (n=0~7)任务寄存器和 IN[n]事件寄存器。OUT[n]用于写引脚, IN[n]由引脚状态变化触发。

当通过 CONFIG 寄存器配置某个引脚由 SET、CLR 和 OUT 任务寄存器或 IN[n]事件寄存器控制后, 该引脚就只能被 GPIOE 模块写操作, 正常的 GPIO 写入无效, 即一旦配置 OUT[n]任务或 IN[n]事件控制某个引脚, 那么该引脚的输出值只能通过 GPIOE 模块操作, 使用 GPIO 的寄存器操作会被忽略。

当 GPIOE 通道被配置用于操作一个任务引脚时, 这个引脚的初始状态是可以配置的, 可以通过配置 CONFIG[n]寄存器中的 OUTINIT 来设置引脚初始化状态为高电平或是低电平。

当在同一个 GPIOE 通道中同时(即同一个时钟周期内)触发了有冲突的任务, 那么任务按照下表所示的优先级执行。

表 11-1: 任务执行优先级

优先级(由高到低)	任务
1	OUT
2	CLEAR
3	SET

2.3. PORT 事件

GPIOE 除了 8 个通道外, 还包含一个 PORT 事件。PORT 事件是多个引脚通过 GPIO DETECT 信号产生的事件, DETECT 信号的上升沿产生 PORT 事件 (DETECT 信号是芯片内部处理的逻辑, 并不表示引脚只能在上升沿时触发 PORT 事件, 引脚可以配置为上升沿、下降沿或电平变化产生 PORT 事件)

PORT 事件由 nRF52840 的所有的 IO 共享, 即 48 个 IO 均可以触发 PORT 事件。PORT 事件使用了 GPIO 外设的感知机制, 也就是如果使用 PORT 事件, 必须操作 GPIO 外设, 在 GPIO 外设中打开引脚的感知功能, 但是要注意 PORT 事件寄存器是属于 GPIOE 外设的。PORT 不需要任何时钟或其他电源相关的外设, 即使外设处于空闲状态, PORT 事件也可以使能。

当 CPU 和所有外设处于 IDLE 状态时, PORT 事件可以将 CPU 从 WFI 或 WFE 类型休眠中, 以及 CPU 空闲模式下唤醒, 即在系统 ON 模式下实现最低的功耗。

PORT 事件也可以作为唤醒源, 将系统从 system off 模式唤醒, 但是本质上还是使用了 GPIO 外设的感知机制来唤醒系统。

配置 PORT 事件时, 为了防止 PORT 事件产生虚假的中断, 用户应首先禁止 PORT 事件中断, 然后再配置 PORT 事件, 之后通过清零 EVENTS_PORT 寄存器清除事件, 最后使能 PORT 事件中断。

3. GPIOE 寄存器

nRF52840 的 GPIOE 的基址如下表所示。

表 11-2: GPIOE 基址

外设名称	基址	描述
GPIOE	0x40006000	GPIO 任务和事件。

nRF52840 GPIOE 的寄存器如下表所示：

表 11-3: GPIO 相关寄存器

序号	寄存器名	偏移地址	功能描述
任务寄存器			
1	OUT[0]	0x000	对 CONFIG[0].PSEL 指定的引脚进行写操作的任务。引脚动作由 CONFIG[0].POLARITY 中的配置决定。
2	OUT[1]	0x004	对 CONFIG[1].PSEL 指定的引脚进行写操作的任务。引脚动作由 CONFIG[1].POLARITY 中的配置决定。
3	OUT[2]	0x008	对 CONFIG[2].PSEL 指定的引脚进行写操作的任务。引脚动作由 CONFIG[2].POLARITY 中的配置决定。
4	OUT[3]	0x00C	对 CONFIG[3].PSEL 指定的引脚进行写操作的任务。引脚动作由 CONFIG[3].POLARITY 中的配置决定。
5	OUT[4]	0x010	对 CONFIG[4].PSEL 指定的引脚进行写操作的任务。引脚动作由 CONFIG[4].POLARITY 中的配置决定。
6	OUT[5]	0x014	对 CONFIG[5].PSEL 指定的引脚进行写操作的任务。引脚动作由 CONFIG[5].POLARITY 中的配置决定。
7	OUT[6]	0x018	对 CONFIG[6].PSEL 指定的引脚进行写操作的任务。引脚动作由 CONFIG[6].POLARITY 中的配置决定。
8	OUT[7]	0x01C	对 CONFIG[7].PSEL 指定的引脚进行写操作的任务。引脚动作由 CONFIG[7].POLARITY 中的配置决定。
9	TASKS_SET[0]	0x030	对 CONFIG[0].PSEL 指定的引脚进行写操作的任务。引脚动作作为输出高电平。
10	TASKS_SET[1]	0x034	对 CONFIG[1].PSEL 指定的引脚进行写操作的任务。引脚动作作为输出高电平。
11	TASKS_SET[2]	0x038	对 CONFIG[2].PSEL 指定的引脚进行写操作的任务。引脚动作作为输出高电平。

12	TASKS_SET[3]	0x03C	对 CONFIG[3].PSEL 指定的引脚进行写操作的任务。引脚动作为输出高电平。
13	TASKS_SET[4]	0x040	对 CONFIG[4].PSEL 指定的引脚进行写操作的任务。引脚动作为输出高电平。
14	TASKS_SET[5]	0x044	对 CONFIG[5].PSEL 指定的引脚进行写操作的任务。引脚动作为输出高电平。
15	TASKS_SET[6]	0x048	对 CONFIG[6].PSEL 指定的引脚进行写操作的任务。引脚动作为输出高电平。
16	TASKS_SET[7]	0x04C	对 CONFIG[7].PSEL 指定的引脚进行写操作的任务。引脚动作为输出高电平。
17	TASKS_CLR[0]	0x060	对 CONFIG[0].PSEL 指定的引脚进行写操作的任务。引脚动作为输出低电平。
18	TASKS_CLR[1]	0x064	对 CONFIG[1].PSEL 指定的引脚进行写操作的任务。引脚动作为输出低电平。
19	TASKS_CLR[2]	0x068	对 CONFIG[2].PSEL 指定的引脚进行写操作的任务。引脚动作为输出低电平。
20	TASKS_CLR[3]	0x06C	对 CONFIG[3].PSEL 指定的引脚进行写操作的任务。引脚动作为输出低电平。
21	TASKS_CLR[4]	0x070	对 CONFIG[4].PSEL 指定的引脚进行写操作的任务。引脚动作为输出低电平。
22	TASKS_CLR[5]	0x074	对 CONFIG[5].PSEL 指定的引脚进行写操作的任务。引脚动作为输出低电平。
23	TASKS_CLR[6]	0x078	对 CONFIG[6].PSEL 指定的引脚进行写操作的任务。引脚动作为输出低电平。
24	TASKS_CLR[7]	0x07C	对 CONFIG[7].PSEL 指定的引脚进行写操作的任务。引脚动作为输出低电平。
事件寄存器			
1	EVENT_IN[0]	0x100	CONFIG[0].PSEL 指定的引脚产生的事件。
2	EVENT_IN[1]	0x104	CONFIG[1].PSEL 指定的引脚产生的事件。
3	EVENT_IN[2]	0x108	CONFIG[2].PSEL 指定的引脚产生的事件。
4	EVENT_IN[3]	0x10C	CONFIG[3].PSEL 指定的引脚产生的事件。
5	EVENT_IN[4]	0x110	CONFIG[4].PSEL 指定的引脚产生的事件。
6	EVENT_IN[5]	0x114	CONFIG[5].PSEL 指定的引脚产生的事件。
7	EVENT_IN[6]	0x118	CONFIG[6].PSEL 指定的引脚产生的事件。
8	EVENT_IN[7]	0x11C	CONFIG[7].PSEL 指定的引脚产生的事件。
9	PORT	0x17C	多个使能了 SENSE 机制的输入引脚产生的事件。
通用寄存器			

1	INTENSET	0x304	使能中断。
2	INTENCLR	0x308	禁用中断。
3	CONFIG[0]	0x510	OUT[0]、SET[0]和 CLR[0]任务和 IN[0]事件的配置。
4	CONFIG[1]	0x514	OUT[1]、SET[1]和 CLR[1]任务和 IN[1]事件的配置。
5	CONFIG[2]	0x518	OUT[2]、SET[2]和 CLR[2]任务和 IN[2]事件的配置。
6	CONFIG[3]	0x51C	OUT[3]、SET[3]和 CLR[3]任务和 IN[3]事件的配置。
7	CONFIG[4]	0x520	OUT[4]、SET[4]和 CLR[4]任务和 IN[4]事件的配置。
8	CONFIG[5]	0x524	OUT[5]、SET[5]和 CLR[5]任务和 IN[5]事件的配置。
9	CONFIG[6]	0x528	OUT[6]、SET[6]和 CLR[6]任务和 IN[6]事件的配置。
10	CONFIG[7]	0x52C	OUT[7]、SET[7]和 CLR[7]任务和 IN[7]事件的配置。

■ INTENSET：中断使能/禁止寄存器

表 11-4: INTENSET 寄存器

位	Field	RW	复位值	描述
位 0	IN0	读/写	0	写“1”使能 IN[0]事件中断。写“0”无效。 读，0: IN[0]事件中断已禁止。 读，1: IN[0]事件中断已使能。
位 1	IN1	读/写	0	写“1”使能 IN[1]事件中断。写“0”无效。 读，0: IN[1]事件中断已禁止。 读，1: IN[1]事件中断已使能。
位 2	IN2	读/写	0	写“1”使能 IN[2]事件中断。写“0”无效。 读，0: IN[2]事件中断已禁止。 读，1: IN[2]事件中断已使能。
位 3	IN3	读/写	0	写“1”使能 IN[3]事件中断。写“0”无效。 读，0: IN[3]事件中断已禁止。 读，1: IN[3]事件中断已使能。
位 4	IN4	读/写	0	写“1”使能 IN[4]事件中断。写“0”无效。 读，0: IN[4]事件中断已禁止。 读，1: IN[4]事件中断已使能。
位 5	IN5	读/写	0	写“1”使能 IN[5]事件中断。写“0”无效。 读，0: IN[5]事件中断已禁止。 读，1: IN[5]事件中断已使能。

位 6	IN6	读/写	0	写“1”使能 IN[6]事件中断。写“0”无效。 读，0: IN[6]事件中断已禁止。 读，1: IN[6]事件中断已使能。
位 7	IN7	读/写	0	写“1”使能 IN[7]事件中断。写“0”无效。 读，0: IN[7]事件中断已禁止。 读，1: IN[7]事件中断已使能。
位 31	PORT	读/写	0	写“1”使能 PORT 事件中断。写“0”无效。 读，0: PORT 事件中断已禁止。 读，1: PORT 事件中断已使能。

■ INTENCLR: 中断使能/禁止寄存器

表 11-5: INTENCLR 寄存器

位	Field	RW	复位值	描述
位 0	IN0	读/写	0	写“1”禁止 IN[0]事件中断。写“0”无效。 读，0: IN[0]事件中断已禁止。 读，1: IN[0]事件中断已使能。
位 1	IN1	读/写	0	写“1”禁止 IN[1]事件中断。写“0”无效。 读，0: IN[1]事件中断已禁止。 读，1: IN[1]事件中断已使能。
位 2	IN2	读/写	0	写“1”禁止 IN[2]事件中断。写“0”无效。 读，0: IN[2]事件中断已禁止。 读，1: IN[2]事件中断已使能。
位 3	IN3	读/写	0	写“1”禁止 IN[3]事件中断。写“0”无效。 读，0: IN[3]事件中断已禁止。 读，1: IN[3]事件中断已使能。
位 4	IN4	读/写	0	写“1”禁止 IN[4]事件中断。写“0”无效。 读，0: IN[4]事件中断已禁止。 读，1: IN[4]事件中断已使能。
位 5	IN5	读/写	0	写“1”禁止 IN[5]事件中断。写“0”无效。 读，0: IN[5]事件中断已禁止。 读，1: IN[5]事件中断已使能。
位 6	IN6	读/写	0	写“1”禁止 IN[6]事件中断。写“0”无效。

				读, 0: IN[6]事件中断已禁止。 读, 1: IN[6]事件中断已使能。
位 7	IN7	读/写	0	写“1”禁止 IN[7]事件中断。写“0”无效。 读, 0: IN[7]事件中断已禁止。 读, 1: IN[7]事件中断已使能。
位 31	PORT	读/写	0	写“1”禁止 PORT 事件中断。写“0”无效。 读, 0: PORT 事件中断已禁止。 读, 1: PORT 事件中断已使能。

■ CONFIG: 配置寄存器

CONFIG 寄存器共有 8 个，对应 GPIOTE 的 8 个通道，CONFIG[0]对应通道 0，依此类推。CONFIG 寄存器用于配置 OUT[n]、SET[n]、CLR[n]任务和 IN[n]事件。

表 11-6: CONFIG 寄存器

位	Field	RW	复位值	描述
位 1~位 0	MODE	读/写	0	0: 禁止。PSEL 寄存器定义的引脚不再用于 GPIOTE 模块。 1: 事件模式。PSEL 寄存器定义的引脚配置为输入，当 POLARITY 寄存器指定的操作在该引脚发生后，产生 IN[n]事件。 3: 任务模式。PSEL 寄存器定义的引脚配置为输出，触发 SET[n]、CLR[n]或 OUT[n]任务会执行 POLARITY 寄存器中指定的动作。一旦引脚配置为任务模式后，该引脚只能通过 GPIOTE 模块访问，而不能通过 GPIO 模块访问。
位 12~位 8	PSEL	读/写	0	[0~31]: 配置和 SET[n]、CLR[n]、OUT[n]任务，IN[1]事件连接的引脚。0~31 对应 P0.00~P0.31。
位 17~位 16	POLARITY	读/写	0	任务模式时：触发 OUT[n]任务执行输出操作。事件模式时：输入操作产生 IN[n]事件。 0: 任务模式下触发 OUT[n]任务无效，事件模式下引脚活动也不会产生 IN[n]事件。 1: 任务模式：触发 OUT[n]任务引脚输出高电平（逻辑“1”），事件模式：上升沿产

				生 IN[n] 事件。
				2: 任务模式: 触发 OUT[n] 任务引脚输出低电平 (逻辑 “0”), 事件模式: 下降沿产生 IN[n] 事件。
				3: 任务模式: 触发 OUT[n] 任务引脚输出翻转, 事件模式: 任意电平变化产生 IN[n] 事件。
位 20	OUTINIT	读/写	0	任务模式时: GPIOTE 通道输出的初始值。 事件模式时: 无影响。 0: 引脚初始值为低电平 (逻辑 “0”)。 1: 引脚初始值为低电平 (逻辑 “1”)。

4. 软件设计

4.1. 库函数的应用

Nordic 的 SDK 中提供了 GPIOTE 的驱动文件, 使用 GPIOTE 驱动可以方便地操作 GPIO 和 GPIOTE 外设、配置和控制输出和输入引脚。输出引脚可以配置为手动控制或通过 GPIOTE 任务控制, 输入引脚可以配置为高精度和低精度/低功耗两种模式, 区别如下:

- **高精度:** 使用一个独立的 GPIOTE 通道事件来检测引脚上的变化, 如果一个引脚被配置受控于这种模式, 那么需要使用高频时钟。
- **低精度/低功耗:** 使用 PORT 事件来检测引脚上的变化, 一个 PORT 事件可用于多个引脚, 因此, 它不是精确的, 它不能用来跟踪高速引脚的变化。如果引脚被配置为这种模式, 那么系统在睡眠的时候可以关闭高频时钟, 所以, 该模式功耗更低。

用于驱动引脚输出的 GPIOTE 任务或者用于在输入引脚电平变换时产生事件的任务/事件通道数量是受限制的, 驱动程序会管理这些通道, 用户是决定不了使用哪个通道的, 也就是说通道的分配由驱动程序完成, 用户不能指定使用哪一个具体的通道。

当我们调用 `nrf_drv_gpiote_in_init()` 或者 `nrf_drv_gpiote_out_init()` 函数时, 驱动程序会分配空闲的通道给应用程序使用, 如果所有的通道已经用完, 那么函数会返回一个错误代码。注意只有当 `nrf_drv_gpiote_out_config_t` 中为任务指定了具体的引脚, 调用 `nrf_drv_gpiote_out_init` 函数才会分配通道。

如果要释放已分配的通道, 调用 `nrf_drv_gpiote_in_uninit()` 函数或者 `nrf_drv_gpiote_out_uninit()` 函数即可。

4.2. GPIOTE 输出流程

GPIOTE 输出的操作流程如下图所示, 其中初始化 GPIOTE 模块和初始化 GPIOTE 输出

引脚是必须要有的，接下来可以选择是否使能该引脚的 GPIOTE 任务模式，任务模式使能之后，只能通过 GPIOTE 任务触发输出，如果不使能 GPIOTE 任务模式，则通过写 GPIO 寄存器控制其输出。

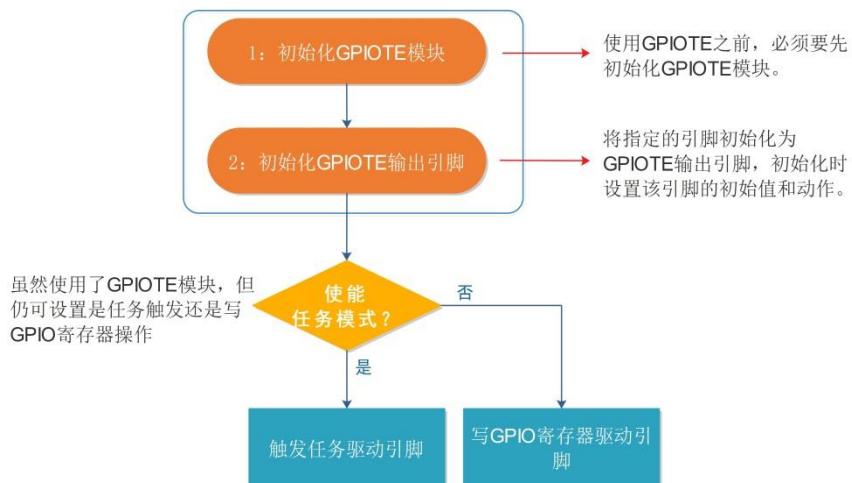


图 11-1：GPIOTE 输出应用步骤

注意：GPIOTE 输出一般不单独使用，因为单独使用时，操作和普通 GPIO 差不多，同样需要程序来操作它，不同的是普通 GPIO 通过 GPIO 的寄存器操作，GPIOTE 通过任务寄存器触发，单独使用时和普通 GPIO 一样同样需要软件干预，体现不了他的优势。GPIOTE 应用的时候一般和 PPI（可编程外设互联）一起用，通过其他外设的事件来触发 GPIOTE 的任务，如使用定时器比较事件等等，这样，整个过程就可以由硬件自动完成，无需软件干预，从而简化程序的流程、降低 CPU 的负担。

1. 初始化 GPIOTE 模块

Nordic 的 SDK 是模块化设计的，使用 GPIOTE 模块时，先要初始化 GPIOTE 模块，初始化很简单，调用初始化函数 nrf_drv_gpiote_init 即可，函数原型如下：

表 11-7：nrf_drv_gpiote_init ()函数

函数原型	<code>ret_code_t nrf_drv_gpiote_init (void)</code>
函数功能	初始化 GPIOTE 模块。仅支持静态配置，以防止用户自定义共享资源。
参 数	无。
返回值	<code>NRF_SUCCESS</code> : 初始化成功。

需要注意的是：GPIOTE 模块在一个应用程序中和很多其他的程序模块共享资源，所以在一个应用程序中 GPIOTE 模块只能初始化一次。

2. 初始化 GPIOTE 输出引脚

初始化 GPIOTE 输出引脚通过调用 nrf_drv_gpiote_out_init() 函数来完成，该函数接收两个输入参数：引脚号和 GPIOTE 输出初始化结构体，引脚号指定将要配置的引脚的编号(0~31)，GPIOTE 输出初始化结构体 nrf_drv_gpiote_out_config_t 包含下面的三项内容：

- 1) 引脚的初始状态：高电平或是低电平。

- 高电平: NRF_GPIOTE_INITIAL_VALUE_HIGH。
- 低电平 NRF_GPIOTE_INITIAL_VALUE_LOW。
- 2) 引脚动作: 任务触发后引脚执行的动作。
 - 置位(高电平): NRF_GPIOTE_POLARITY_LOTOHI。
 - 清除(低电平): NRF_GPIOTE_POLARITY_HITOLO。
 - 翻转: NRF_GPIOTE_POLARITY_TOGGLE。
- 3) 是否为 GPIOTE 任务引脚: 如果设置成 GPIOTE 任务引脚, 驱动程序才会去配置前面两项(初始状态和引脚动作)并且为该引脚分配 GPIOTE 通道, 否则, 将该引脚按照普通 GPIO 处理。需要注意, 即使将引脚配置为 GPIOTE 任务引脚, 也需要通过 nrf_drv_gpiote_out_task_enable()函数使能该引脚的任务触发功能之后, 触发任务才对该引脚无效。

GPIOTE 初始化输出引脚的函数有初始化函数 nrf_drv_gpiote_out_init ()和反初始化函数 nrf_drv_gpiote_out_uninit (), 函数原型如下:

表 11-8: nrf_drv_gpiote_out_init ()函数

函数原型	<pre>ret_code_t nrf_drv_gpiote_out_init (nrf_drv_gpiote_pin_t pin, nrf_drv_gpiote_out_config_t const * p_config)</pre>
函数功能	初始化 GPIOTE 输出引脚。输出引脚初始化配置中会设定引脚的动作(低电平到高电平、高电平到低电平或者翻转状态)和初始状态(高电平或低电平), 如果没有空闲的 GPIOTE 通道, 将会返回一个错误代码。
参数	[in] pin: 引脚号。 [in] p_config: 指向输出初始化配置结构体。
返回值	<ul style="list-style-type: none"> ● NRF_SUCCESS: 初始化成功。 ● NRF_ERROR_INVALID_STATE: 驱动没有初始化或者指定的引脚已经被占用。 ● NRF_ERROR_NO_MEM: 没有空闲的 GPIOTE 通道。

表 11-9: nrf_drv_gpiote_out_uninit ()函数

函数原型	<pre>void nrf_drv_gpiote_out_uninit(nrf_drv_gpiote_pin_t pin)</pre>
函数功能	释放 GPIOTE 输出引脚。如果该输出引脚被使用, 驱动程序会释放此 GPIOTE 通道。
参数	[in] pin: 引脚号。
返回值	无。

3. 使能任务模式

调用 `nrf_drv_gpiote_out_init()` 函数后，指定的引脚被配置为 GPIOTE 输出引脚，但是 `nrf_drv_gpiote_out_init()` 函数并没有使能该引脚的任务模式，即没有配置 `CONFIG[n].MODE` 为任务模式。所以，应用程序需要使能任务模式，即配置 `CONFIG[n].MODE = 3`，之后才能通过 GPIOTE 操作引脚。

使能任务模式通过调用 `nrf_drv_gpiote_out_task_enable()` 函数实现，只有使能任务触发后，我们才能通过触发 GPIOTE 的任务让引脚执行动作。使能任务模式和禁止任务模式的函数原型如下：

表 11-10: `nrfx_gpiote_out_task_enable()` 函数

函数原型	<code>void nrfx_gpiote_out_task_enable (nrfx_gpiote_pin_t pin)</code>
函数功能	使能 GPIOTE 输出引脚的任务模式。一旦调用该函数使能引脚的任务模式后，该引脚只能使用 GPIOTE 操作，而不能使用 GPIO 操作。
参数	[in] pin: 引脚号。
返回值	无。

表 11-11: `nrfx_gpiote_out_task_disable()` 函数

函数原型	<code>void nrfx_gpiote_out_task_disable (nrfx_gpiote_pin_t pin)</code>
函数功能	禁止 GPIOTE 输出引脚的任务模式。禁止后，使用 GPIO 操作该引脚，触发 GPIOTE 任务对该引脚无效。
参数	[in] pin: 引脚号。
返回值	无。

4. 触发任务驱动引脚

如果使能了引脚的任务模式，通过下面三个函数触发 GPIOTE 任务，驱动引脚动作，函数的原型如下表所示：

- 1) `nrf_drv_gpiote_set_task_trigger()` 函数：触发 GPIOTE 的 TASKS_SET 任务，触发后，引脚输出高电平。
- 2) `nrf_drv_gpiote_clr_task_trigger()` 函数：触发 GPIOTE 的 TASKS_CLR 任务，触发后，引脚输出低电平。
- 3) `nrf_drv_gpiote_out_task_trigger()` 函数：触发 GPIOTE 的 TASKS_OUT 任务，每触发一次，引脚状态翻转一次。

表 11-12: `nrf_drv_gpiote_set_task_trigger()` 函数

函数原型	<code>void nrf_drv_gpiote_set_task_trigger (nrf_drv_gpiote_pin_t pin)</code>
函数功能	手动触发 GPIOTE SET 任务，触发后，对应的引脚输出高电平。注意：

	这里说的手动触发指的是通过程序来触发，也就是向 TASKS_SET 寄存器中写入“1”。
参 数	[in] pin: 引脚号。
返回值	无。

表 11-13: nrf_drv_gpiote_clr_task_trigger ()函数

函数原型	void nrf_drv_gpiote_clr_task_trigger (nrf_drv_gpiote_pin_t pin)
函数功能	手动触发 GPIOTE CLR 任务，触发后，对应的引脚输出低电平。注意：这里说的手动触发指的是通过程序来触发，也就是向 TASKS_CLR 寄存器中写入“1”。
参 数	[in] pin: 引脚号。
返回值	无。

表 11-14: nrf_drv_gpiote_out_task_trigger ()函数

函数原型	void nrf_drv_gpiote_out_task_trigger (nrf_drv_gpiote_pin_t pin)
函数功能	手动触发 GPIOTE OUT 任务，触发后，对应的引脚输出状态翻转。注意：这里说的手动触发指的是通过程序来触发，也就是向 TASKS_OUT 寄存器中写入“1”。
参 数	[in] pin: 引脚号。
返回值	无。

如果没有使能引脚的任务模式，通过下面三个函数驱动引脚动作，函数的原型如下表所示：

- 1) nrfx_gpiote_out_set ()函数：驱动指定的引脚输出高电平。
- 2) nrfx_gpiote_out_clear()函数：驱动指定的引脚输出低电平。
- 3) nrfx_gpiote_out_toggle()函数：驱动指定的引脚输出状态翻转。

表 11-15: nrfx_gpiote_out_set ()函数

函数原型	void nrfx_gpiote_out_set(nrfx_gpiote_pin_t pin)
函数功能	置位指定的 GPIOTE 输出引脚，即该引脚输出高电平。若使能了引脚的任务模式，该函数无效。
参 数	[in] pin: 引脚号。
返回值	无。

表 11-16: nrfx_gpiote_out_clear ()函数

函数原型	void nrfx_gpiote_out_clear(nrfx_gpiote_pin_t pin)
函数功能	清零指定的 GPIOTE 输出引脚，即该引脚输出低电平。若使能了引脚的任务模式，该函数无效。
参数	[in] pin: 引脚号。
返回值	无

表 11-17: nrfx_gpiote_out_toggle()函数

函数原型	void nrfx_gpiote_out_toggle(nrfx_gpiote_pin_t pin)
函数功能	翻转指定的 GPIOTE 输出引脚，即该引脚原先为高电平，则输出低电平，原先为低电平，则输出高电平。若使能了引脚的任务模式，该函数无效。
参数	[in] pin: 引脚号。
返回值	无。

4.3. GPIOTE 输入流程

GPIOTE 输入的操作流程如下图所示，包含初始化 GPIOTE 程序模块、初始化 GPIOTE 输入引脚、使能事件模式。

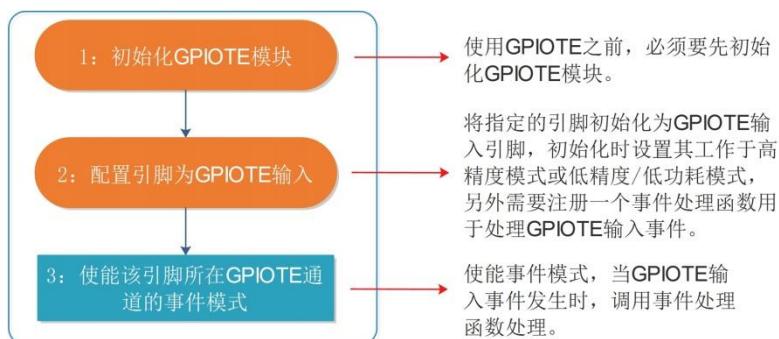


图 11-2: GPIOTE 输入应用步骤

1. 初始化 GPIOTE 模块

和 GPIOTE 输入时一样，调用初始化函数 `nrf_drv_gpiote_init()` 即可。注意即使配置多个引脚，部分引脚配置为输出，部分配置为输入，也只需要调用一次 `nrf_drv_gpiote_init()` 初始化 GPIOTE 模块。

2. 初始化 GPIOTE 输入引脚

初始化 GPIOTE 输入引脚通过调用 `nrf_drv_gpiote_in_init()` 函数来完成，该函数接收三个输入参数：引脚号、GPIOTE 输入初始化结构体和事件句柄，相对于 GPIOTE 输出引脚的初始化，函数参数多了一个事件句柄，当 GPIOTE 检测到引脚电平变化后，会产生事件，这时 GPIOTE 程序模块会自动调用这个注册的事件句柄来处理事件。

引脚号指定将要配置的引脚的编号(0~31)，GPIOTE 输入初始化结构体 `nrf_drv_gpiote_in_config_t` 包含下面的 4 项内容：

- 1) Sense: 配置引脚的 Sense 功能, 可配置为下面。
 - 高电平到低电平的变化产生事件: NRF_GPIOTE_POLARITY_HITOLO。
 - 低电平到高电平的变化产生事件: NRF_GPIOTE_POLARITY_LOTOHI。
 - 任意电平变化产生事件: NRF_GPIOTE_POLARITY_TOGGLE。
- 2) is_watcher: 是否连接输入缓冲器, 设置为“true”表示连接输入缓冲器。
- 3) pull: 是否开启引脚的上拉电阻, 设置为“true”表示打开上拉。
- 4) hi_accuracy: 是否为高精度模式, 设置为“true”表示使用高精度模式。

nrf_drv_gpiote_in_init ()函数原型如下:

表 11-18: nrf_drv_gpiote_in_init ()函数

函数原型	<pre>ret_code_t nrf_drv_gpiote_in_init (nrf_drv_gpiote_pin_t pin, nrf_drv_gpiote_in_config_t const * p_config, nrf_drv_gpiote_evt_handler_t evt_handler)</pre>
函数功能	<p>初始化 GPIOTE 输入引脚。输入引脚可以工作于下面两种模式:</p> <ul style="list-style-type: none"> • 高精度: 需要使用高频时钟。 • 低精度/低功耗: 无需高频时钟。 <p>初始化的时候, 可以指定 GPIOTE 输入引脚的工作模式, 如果工作于高精度模式, 驱动程序会尝试为其分配一个 GPIOTE 通道, 如果 GPIOTE 通道全部被占用, 则会返回一个错误代码。低精度/低功耗模式下使用引脚的 SENSE 功能, 在这种情况下, 某一时刻只能检测到一个活动的引脚。</p>
参数	<p>[in] <code>pin</code>: 引脚号。</p> <p>[in] <code>p_config</code>: 指向输入初始化配置结构体。</p> <p>[in] <code>evt_handler</code>: 事件句柄。</p>
返回值	<ul style="list-style-type: none"> • NRF_SUCCESS: 初始化成功。 • NRF_ERROR_INVALID_STATE: 驱动没有初始化或者指定的引脚已经被占用。 • NRF_ERROR_NO_MEM: 没有空闲的 GPIOTE 通道。

3. 使能事件模式

初始化 GPIOTE 输入引脚时为指定的引脚分配了 GPIOTE 通道, 但是并没有配置该引脚的模式即没有配置 CONFIG[n].MODE, 使能事件模式就是设置 CONFIG[n].MODE = 1, 即将其配置为事件模式, 当其检测到引脚电平变化时(初始化 GPIOTE 输入引脚时 Sense 项配置了哪种变化可以产生事件)即产生事件。

表 11-19: nrfx_gpiote_in_event_enable ()函数

函数原型	<pre>void nrfx_gpiote_in_event_enable (</pre>
------	---

	<pre>nrfx_gpiote_pin_t pin, bool int_enable)</pre>
函数功能	使能 GPIOTE 输入引脚。如果引脚已配置为高精度模式，该函数使能 IN_EVENT 事件模式，否则，使能 GPIO 的感知机制。注意，PORT 事件被多个引脚共享，所以中断总是使能的。
参数	[in] pin: 引脚号。 [in] int_enable: true: 使能中断。对高精度引脚总是有效。
返回值	无。

表 11-20: nrfx_gpiote_out_task_disable ()函数

函数原型	<pre>void nrfx_gpiote_out_task_disable (nrfx_gpiote_pin_t pin)</pre>
函数功能	禁止 GPIOTE 输出引脚的任务模式。禁止后，使用 GPIO 操作该引脚，触发 GPIOTE 任务对该引脚无效。
参数	[in] pin: 引脚号。
返回值	无。

4.4. GPIOTE 通道输出试验

本实验在“实验 6-3: 流水灯(BSP 实现方式)”的基础上修改。将 P0.13(指示灯 D1)配置为 GPIOTE 输出引脚，并使能其任务模式。在程序中修改输出引脚的动作，分别将其设置为高电平、低电平和翻转，之后触发任务，观察 P0.13 引脚的输出，即观察指示灯 D1 的状态。

❖ 注：本节对应的实验源码是：“实验 11-1：GPIOTE 通道输出”。

4.4.1. 添加需要的文件

使用 GPIOTE 程序模块需要加入的文件如下表所示。

表 11-21: GPIOTE 需要加入的文件

文件名	SDK 中的目录	描述
nrfx_gpiote.c	...\\modules\\nrfx\\drivers\\src	GPIOTE 驱动文件。

4.4.2. 头文件引用和路径设置

■ 需要引用的头文件

因为在“main.c”文件中使用了 GPIOTE 程序模块，所以“main.c”文件需要引用下面

的头文件。

```
#include "nrf_drv_gpiote.h"
```

■ 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，切换到“C/C++”选项卡，按照下图所示添加头文件包含路径。

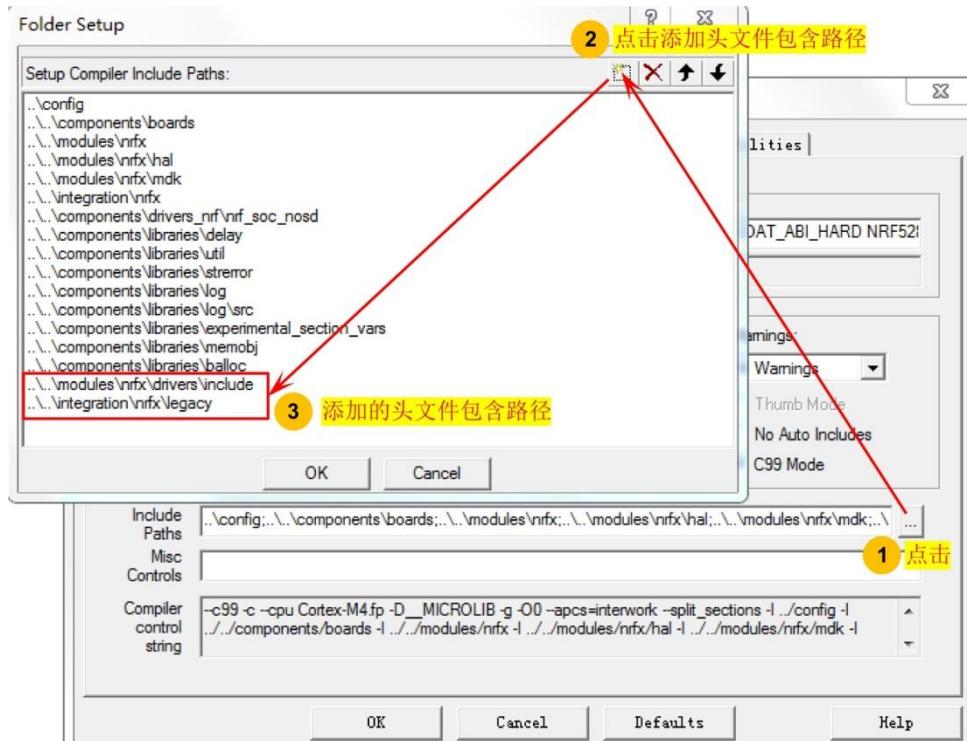


图 11-3：添加头文件包含路径

GPIOTE 需要添加的头文件路径如下表：

表 11-22：头文件包含路径

序号	路径
1	..\..\modules\nrfx\drivers\include
2	..\..\integration\nrfx\legacy

4.4.3. 工程配置

打开“ `sdk_config.h`”文件，加入串口配置，如下图所示，编辑内容时切换到“Text Editor”进行编辑（编辑的内容参考实验的源码，因此代码很长，此处不贴出代码，参见“ `sdk_config.h`”文件的（52~157）行），编辑完成后，切换到“Configuration Wizard”，即可观察到配置的具体项目：

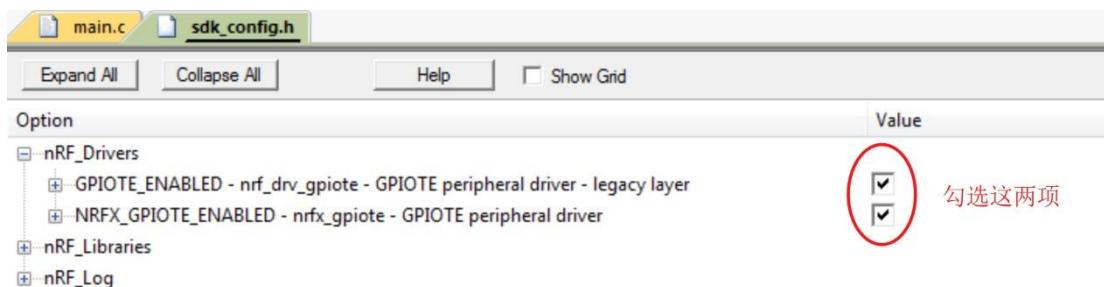


图 11-4: 工程配置

4.4.4. 代码编写

本例演示了使用 GPIOTE 程序模块时，通过 GPIOTE 和 GPIO 操作引脚的方式和区别。所以，我们在程序定义一个宏定义 USE_AS_GPIO，作为条件编译的判断条件，当 USE_AS_GPIO 定义后，程序虽然配置了 GPIOTE，但是因为没有使能引脚的任务模式，所以仍然需要用 GPIO 来操作引脚。当 USE_AS_GPIO 没有定义时，程序中执行 nrf_drv_gpiote_out_task_enable() 函数使能引脚的任务模式，任务模式使能后，只能使用 GPIOTE 操作引脚，使用 GPIO 操作引脚无效。

代码清单：宏定义 USE_AS_GPIO

```
1. //定义后, nrf_drv_gpiote_out_task_enable 函数不会执行, 即不使能引脚的任务模式。这时使
2. //用 GPIO 操作引脚 ;屏蔽该宏定义后, 引脚的任务模式使能, 这时, 只能用 GPIOTE 操作引脚。
3. //#define USE_AS_GPIO
```

代码清单：根据宏定义 USE_AS_GPIO 配置并使用 P0.13 驱动 LED

```
1. ****
2. * 描述 : main 函数
3. * 入参 : 无
4. * 返回值 : int 类型
5. ****
6. int main(void)
7. {
8.     ret_code_t err_code;
9.     //设置 GPIO 输出电压为 3.3V
10.    gpio_output_voltage_setup_3v3();
11.    //初始化 GPIOTE 程序模块
12.    err_code = nrf_drv_gpiote_init();
13.    APP_ERROR_CHECK(err_code);
14.
15.    //定义 GPIOTE 输出初始化结构体, 并对其成员变量赋值
16.    nrf_drv_gpiote_out_config_t config = GPIOTE_CONFIG_OUT_TASK_TOGGLE(true);
```

```
17.  
18.    //初始化 GPIOTE 输出引脚  
19.    err_code = nrf_drv_gpiote_out_init(LED_1, &config);  
20.    APP_ERROR_CHECK(err_code);  
21.  
22.    #if !defined(USE_AS_GPIO)  
23.        //使能引脚 LED_1(P0.13)所在 GPIOTE 通道的任务模式，一旦使能任务模式后，只能用 GPIOTE  
24.        //操作该引脚  
25.        nrf_drv_gpiote_out_task_enable(LED_1);  
26.    #endif  
27.  
28.    while(true)  
29.    {  
30.        #if !defined(USE_AS_GPIO)  
31.            //任务触发驱动引脚 P0.13 状态翻转，即指示灯 D1 翻转状态  
32.            nrf_drv_gpiote_out_task_trigger(LED_1);  
33.            nrf_delay_ms(150);  
34.        #else  
35.            //使用 GPIO 操作引脚 P0.13 输出高电平，D1 熄灭  
36.            nrfx_gpiote_out_set(LED_1);  
37.            nrf_delay_ms(500);  
38.            //使用 GPIO 操作引脚 P0.13 输出低电平，D1 点亮  
39.            nrfx_gpiote_out_clear(LED_1);  
40.            nrf_delay_ms(500);  
41.        #endif  
42.    }  
43. }
```

4.4.5. 硬件连接

本实验需要使用 P0.13 驱动 LED 指示灯 D1，按照下图所示用跳线帽短接 P0.13。

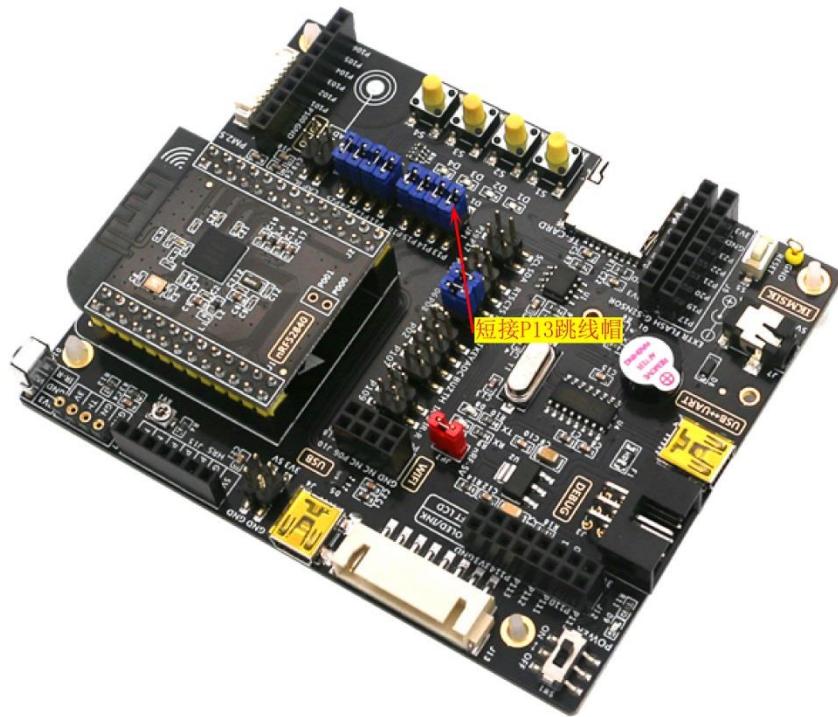


图 11-5：开发板跳线帽短接

4.4.6. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 11-1: GPIOTE 输出”，将解压后得到的文件夹“gpiote_output”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project” 打开“…\gpiote_output\project\mdk5” 目录下的工程“gpiote_output.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，每 150ms 触发一次 OUT 任务驱动 D1 指示灯状态翻转一次，可以观察到开发板上的指示灯 D1 以 150ms 的间隔闪烁。
7. 取消“#define USE_AS_GPIO”的屏蔽，重新编译程序并下载到开发板运行，这时，GPIOTE 程序模块用 GPIO 每 500ms 驱动指示灯 D1 状态翻转一次，可以观察到开发板上的指示灯 D1 以 500ms 的间隔闪烁。

4.5. GPIOE 通道输入试验

本实验在“实验 11-1: GPIOTE 输出”的基础上修改。将 P0.11(按键 S1)配置为 GPIOTE 输入引脚，使能其事件模式。在程序中修改引脚的 Sense 配置，分别将其设置为高电平到

低电平变化、低电平到高电平变化和任意电平变化，GPIOTE 事件回调函数中翻转指示灯 D1 的状态，之后按动 S1 按键，观察指示灯 D1 的状态。

❖ 注：本节对应的实验源码是：“实验 11-2：GPIOTE 通道输入”。

4.5.1. 代码编写

GPIOTE 输入和输出一样，首先初始化 GPIOTE 程序模块，之后为引脚分配 GPIOTE 通道并配置通道的参数，配置完成之后使能该通道的事件模式。

代码清单：配置 P0.11 为 GPIOTE 输入

```
1. /*****
2. * 描述 : main 函数
3. * 入参 : 无
4. * 返回值 : int 类型
5. *****/
6. int main(void)
7. {
8.     //设置 GPIO 输出电压为 3.3V
9.     gpio_output_voltage_setup_3v3();
10.
11.    //初始化开发板上的 4 个 LED，即将驱动 LED 的 GPIO 配置为输出，
12.    bsp_board_init(BSP_INIT_LEDS);
13.    ret_code_t err_code;
14.
15.    //初始化 GPIOTE 程序模块
16.    err_code = nrf_drv_gpiote_init();
17.    APP_ERROR_CHECK(err_code);
18.    //以下代码配置 P0.11 作为 GPIOTE 输入，并分别配置引脚上升沿、下降沿和任意电平变化产生
19.    //事件。
20.    //高电平到低电平变化产生事件，即下降沿产生事件
21.    nrf_drv_gpiote_in_config_t in_config = GPIOTE_CONFIG_IN_SENSE_HITOLO(true);
22.
23.    //低电平到高电平变化产生事件，即上升沿产生事件
24.    //nrf_drv_gpiote_in_config_t in_config = GPIOTE_CONFIG_IN_SENSE_LOTOHI(true);
25.    //任意电平变化产生事件
26.    //nrf_drv_gpiote_in_config_t in_config = GPIOTE_CONFIG_IN_SENSE_TOGGLE(true);
27.
28.    //开启 P0.11 引脚的上拉电阻
29.    in_config.pull = NRF_GPIO_PIN_PULLUP;
30.    //配置该引脚为 GPIOTE 输入
31.    err_code = nrf_drv_gpiote_in_init(BUTTON_1, &in_config, in_pin_handler);
32.    APP_ERROR_CHECK(err_code);
33.    //使能该引脚所在 GPIOTE 通道的事件模式
34.    nrf_drv_gpiote_in_event_enable(BUTTON_1, true);
```

```
35.  
36.     while(true)  
37.     {  
38.     }  
39. }
```

事件模式使能之后，引脚上的输入状态若发生了和配置内容一致的变化，则会产生事件，所以应用程序需要提供事件回调函数用于处理事件。事件回调函数在 GPIOTE 初始化时会注册到 GPIOTE 模块，供 GPIOTE 模块调用。

GPIOTE 事件回调函数中，可以读出是哪一个引脚产生了事件，也可以读出引脚的状态变化类型（上升沿、下降沿还是任意电平变化）。

代码清单：GPIOTE 事件回调函数

```
1. /*****  
2. * 描述 : main 函数  
3. * 入参 : 无  
4. * 返回值 : int 类型  
5. *****/  
6. int main(void)  
7. {  
8.     //设置 GPIO 输出电压为 3.3V  
9.     gpio_output_voltage_setup_3v3();  
10.  
11.    //初始化开发板上的 4 个 LED，即将驱动 LED 的 GPIO 配置为输出，  
12.    bsp_board_init(BSP_INIT_LEDS);  
13.    //GPIOTE 事件处理函回调函数，事件回调函数里面可以获取 pin 编号和引脚动作  
14.    void in_pin_handler(nrf_drv_gpiote_pin_t pin, nrf_gpiote_polarity_t action)  
15.    {  
16.        //事件由按键 S1 产生  
17.        if(pin == BUTTON_1)  
18.        {  
19.            //翻转指示灯 D1 的状态  
20.            nrf_gpio_pin_toggle(LED_1);  
21.        }  
22.        //判断引脚动作  
23.        if(action == NRF_GPIOTE_POLARITY_HITOLO)nrf_gpio_pin_toggle(LED_2);  
24.        else if(action == NRF_GPIOTE_POLARITY_LOTOHI)nrf_gpio_pin_toggle(LED_3);  
25.        else if(action == NRF_GPIOTE_POLARITY_TOGGLE)nrf_gpio_pin_toggle(LED_4);  
26.    }
```

4.5.2. 硬件连接

本实验需要使用 P0.13~P0.16 驱动 LED 指示灯 D1~D4，P0.11 检测轻触按键 S1 输入，按照下图所示用短接跳线帽。

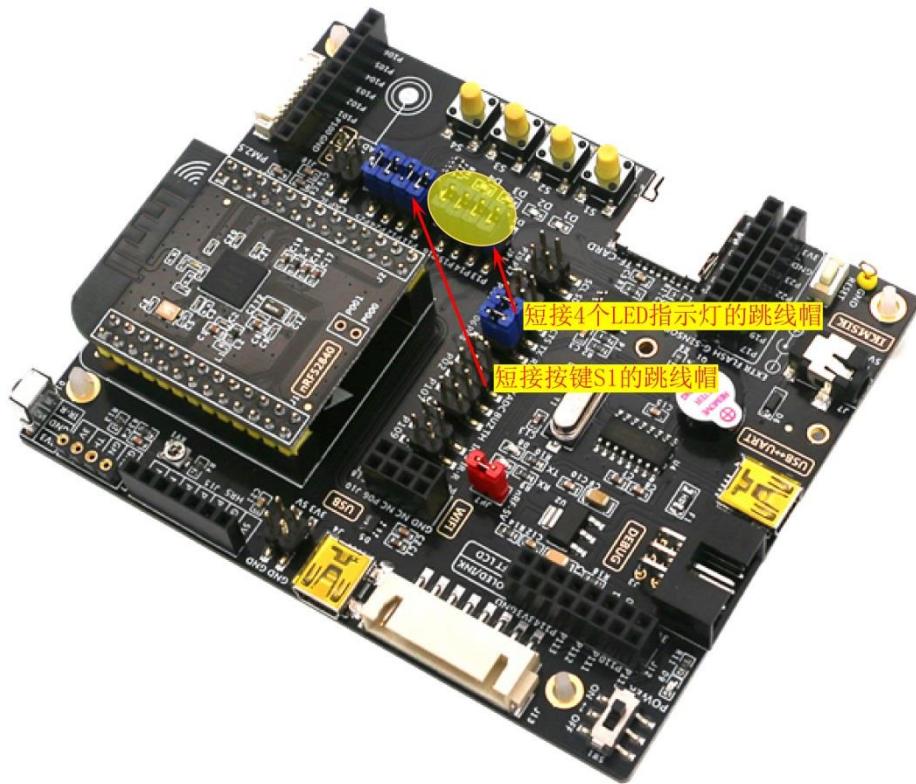


图 11-6：开发板跳线帽短接

4.5.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码”目录下的压缩文件“实验 11-2: GPIOTE 通道输入”，将解压后得到的文件夹“gpiote_input”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\gpiote_input\project\mdk5”目录下的工程“gpiote_input.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，按动 S1 按键，每按动一次即产生一个高电平到低电平变化的事件，可以看到每按动一次开发板上的指示灯 D1 状态翻转一次，同时 LED 指示灯 D2 的状态也翻转一次，表示 GPIOTE 事件回调函数中当前产生的 GPIOTE 事件是高电平到低电平变化产生的事件。
7. 将 GPIOTE 配置分别修改为低电平到高电平和任意电平变化产生事件，编译后运行，按动 S1 按键，D1 状态翻转，GPIOTE 事件回调函数中根据产生的事件翻转 D3 或 D4 状态。

■ **思考题 1：**如何使用 GPIOTE 通道（不是 PORT）检测多个引脚输入？编写使用 GPIOTE

通道检测 4 个按键输入的程序，要求下降沿触发事件。

4.6. GPIOTE PORT 输入试验

本实验在“实验 11-2：GPIOTE 通道输入”的基础上修改。使用 GPIOTE PORT 事件检测引脚输入状态变化。

nRF52840 的 GPIOTE 共有 8 个通道，可以检测 8 路输入，如果我们需要检测的输入超过 8 路，使用 GPIOTE 通道是无法实现的，这时，我们可以使用 GPIOTE 的 PORT 来完成检测。PORT 事件由 nRF52840 的所有的 IO 共享，所以使用 PORT 最多可以检测 48 路引脚输入的变化。

GPIOTE PORT 相对于 GPIOTE 通道的优缺点如下：

- PORT 事件是 48 个 IO 共有的事件，所以 PORT 事件产生后，需要查询是哪一个引脚产生的。
- PORT 的时钟源来自于 32.768KHz 低频时钟，所以它的功耗更低，但是反应速度没有 GPIOTE 通道快速。

本例中，我们用 PORT 来检测 5 路输入的状态：4 个轻触按键和 1 个触摸按键。

❖ 注：本节对应的实验源码是：“实验 11-3：GPIOTE PORT 输入”。

4.6.1. 代码编写

SDK 中已经定义好了 4 个按键的标号：BUTTON_1~ BUTTON_4，但是触摸按键是没有定义的，这里我们需要自己定义触摸按键。

代码清单：定义触摸按键引脚编号

```
1. //定义触摸按键的引脚编号 P0.02
2. #define BUTTON_TOUCH NRF_GPIO_PIN_MAP(0,2)
```

初始化部分，我们需要对 5 个按键分别进行初始化，初始化配置宏的参数设置为 false，即低精度。

按键 BUTTON_1~ BUTTON_4 是低电平有效，所以设置引脚为：下降沿产生事件，触摸按键是高电平有效，所以设置为：上升沿产生事件。

代码清单：初始化 GPIOTE PORT

```
1. ****
2. * 描述 : main 函数
3. * 入参 : 无
4. * 返回值 : int 类型
5. ****
6. int main(void)
7. {
8.     //设置 GPIO 输出电压为 3.3V
9.     gpio_output_voltage_setup_3v3();
```

```
10.  
11. //初始化开发板上的 4 个 LED，即将驱动 LED 的 GPIO 配置为输出,  
12. bsp_board_init(BSP_INIT_LEDS);  
13. net_code_t err_code;  
14. //初始化 GPIOTE 程序模块  
15. err_code = nrf_drv_gpiote_init();  
16. APP_ERROR_CHECK(err_code);  
17. //定义 GPIOTE 配置结构体，配置为下降沿触发（按键是低电平有效），低精度(即使用  
18. //GPIOTE PORT)  
19. nrf_drv_gpiote_in_config_t in_config_hitlo = GPIOTE_CONFIG_IN_SENSE_HITOLO(false);  
20. //开启引脚的上拉电阻  
21. in_config_hitlo.pull = NRF_GPIO_PIN_PULLUP;  
22.  
23. //配置引脚 P0.11 为 GPIOTE 输入  
24. err_code = nrf_drv_gpiote_in_init(BUTTON_1, &in_config_hitlo, in_pin_handler);  
25. APP_ERROR_CHECK(err_code);  
26.  
27. //配置引脚 P0.12 为 GPIOTE 输入  
28. err_code = nrf_drv_gpiote_in_init(BUTTON_2, &in_config_hitlo, in_pin_handler);  
29. APP_ERROR_CHECK(err_code);  
30.  
31. //配置引脚 P0.24 为 GPIOTE 输入  
32. err_code = nrf_drv_gpiote_in_init(BUTTON_3, &in_config_hitlo, in_pin_handler);  
33. APP_ERROR_CHECK(err_code);  
34.  
35. //配置引脚 P0.25 为 GPIOTE 输入  
36. err_code = nrf_drv_gpiote_in_init(BUTTON_4, &in_config_hitlo, in_pin_handler);  
37. APP_ERROR_CHECK(err_code);  
38.  
39. //配置 P0.02 作为 GPIOTE 输入，高电平到低电平变化产生事件，即下降沿产生事件,低精度(即使用  
40. //GPIOTE PORT)  
41. nrf_drv_gpiote_in_config_t in_config_lothi = GPIOTE_CONFIG_IN_SENSE_LOTOHI(false);  
42. //开启引脚的上拉电阻  
43. in_config_lothi.pull = NRF_GPIO_PIN_PULLUP;  
44. //配置该引脚为 GPIOTE 输入  
45. err_code = nrf_drv_gpiote_in_init(BUTTON_TOUCH, &in_config_lothi, in_pin_handler);  
46. APP_ERROR_CHECK(err_code);  
47.  
48. //注意这里的使能函数，对于 GPIOTE 通道，使能的是通道的事件模式，对于 PORT，使能的是引脚的感知功能  
49. //使能 P0.11 感知功能  
50. nrf_drv_gpiote_in_event_enable(BUTTON_1, true);  
51. //使能 P0.12 感知功能  
52. nrf_drv_gpiote_in_event_enable(BUTTON_2, true);  
53. //使能 P0.24 感知功能
```

```
54.     nrf_drv_gpiote_in_event_enable(BUTTON_3, true);
55.     //使能 P0.25 感知功能
56.     nrf_drv_gpiote_in_event_enable(BUTTON_4, true);
57.     //使能 P0.02 感知功能
58.     nrf_drv_gpiote_in_event_enable(BUTTON_TOUCH, true);
59.
60.     while(true)
61.     {
62.     }
63. }
```

GPIOE 事件回调函数中，可以读取产生事件的引脚号，应用程序从而知道是 PORT 中的哪个引脚产生了事件。这里，我们通过 LED 指示产生事件的引脚。

- BUTTON_1 (按键 S1): 翻转 LED 指示灯 D1 的状态。
- BUTTON_2 (按键 S2): 翻转 LED 指示灯 D2 的状态。
- BUTTON_3 (按键 S3): 翻转 LED 指示灯 D3 的状态。
- BUTTON_4 (按键 S4): 翻转 LED 指示灯 D4 的状态。
- 触摸按键: 同时翻转 LED 指示灯 D1~D4 的状态。

代码清单：GPIOE 事件回调函数

```
1. //GPIOE 事件处理函回调函数，事件回调函数里面可以获取 pin 编号和引脚状态变化
2. void in_pin_handler(nrf_drv_gpiote_pin_t pin, nrf_gpiote_polarity_t action)
3. {
4.     //事件由按键 S1 产生，即按键 S1 按下
5.     if(pin == BUTTON_1)
6.     {
7.         //翻转指示灯 D1 的状态
8.         nrf_gpio_pin_toggle(LED_1);
9.     }
10.    //事件由按键 S2 产生，即按键 S2 按下
11.    else if(pin == BUTTON_2)
12.    {
13.        //翻转指示灯 D2 的状态
14.        nrf_gpio_pin_toggle(LED_2);
15.    }
16.    //事件由按键 S3 产生，即按键 S3 按下
17.    else if(pin == BUTTON_3)
18.    {
19.        //翻转指示灯 D3 的状态
20.        nrf_gpio_pin_toggle(LED_3);
21.    }
22.    //事件由按键 S4 产生，即按键 S4 按下
```

```

23.     else if(pin == BUTTON_4)
24.     {
25.         //翻转指示灯 D4 的状态
26.         nrf_gpio_pin_toggle(LED_4);
27.     }
28.     //事件由触摸按键产生，即按键 S1 按下
29.     else if(pin == BUTTON_TOUCH)
30.     {
31.         //点亮 D1~D4
32.         nrf_gpio_pin_toggle(LED_1);
33.         nrf_gpio_pin_toggle(LED_2);
34.         nrf_gpio_pin_toggle(LED_3);
35.         nrf_gpio_pin_toggle(LED_4);
36.     }
37. }
```

代码编写完成后，我们还需要修改“sdk_config”文件中的PORT引脚数量，本例中我们使用了5个引脚，所以需要将数量改为5，否则程序是无法运行的。如下图所示：

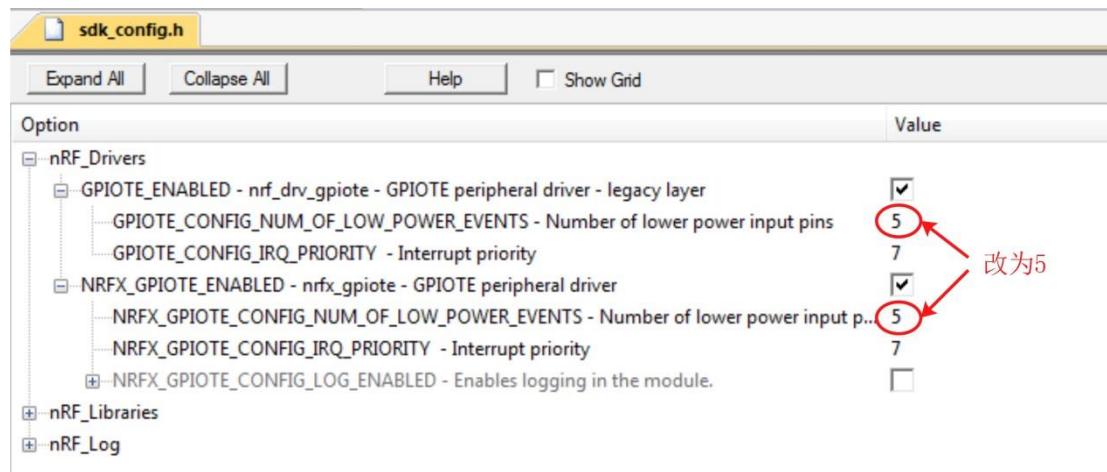


图 11-7：修改 PORT 引脚数量

4.6.2. 硬件连接

本实验需要使用 P0.13~P0.16 驱动 LED 指示灯 D1~D4，P0.11 P0.12 P0.24 P0.25 检测轻触按键输入，P0.02 检测触摸按键输入，按照下图所示用跳线帽短接。

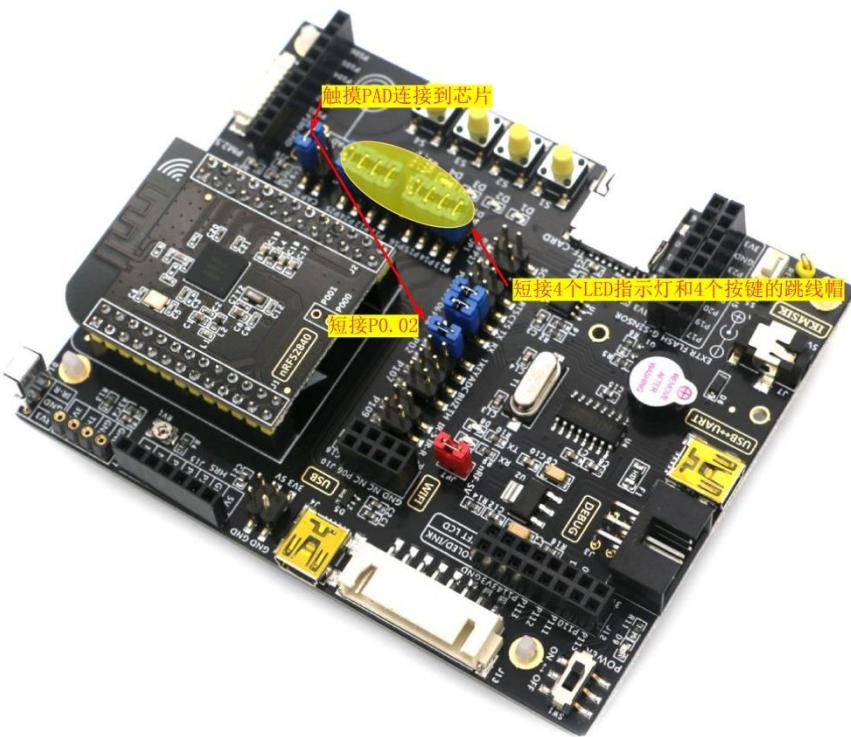


图 11-8：开发板跳线帽短接

4.6.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码”目录下的压缩文件“实验 11-3: GPIO PORT 输入”，将解压后得到的文件夹“gpiote_port_input”拷贝到合适的目录，如“D\ nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\gpiote_port_input\project\mdk5”目录下的工程“gpiote_port_input.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，按动 S1~S4 按键，每按动一次对应的指示灯 D1~D4 状态翻转一次，触碰触摸按键，每触碰一次，4 个指示灯状态同时翻转一次。

第十二章： SysTick

1. 学习目的

1. 学习 SysTick（系统节拍）定时器的原理。
2. 掌握 SysTick 库函数的应用。
3. 了解开发 nRF52840 应用时，什么情况下适合使用 SysTick。

2. SysTick 概述

SysTick 是 Cortex-M 处理器的外设，称为系统定时器，Cortex-M 处理器之所以集成 SysTick，是为了将其用作 OS 的时钟节拍，这样 Cortex-M 处理器之间就可以保持统一，方便 OS 的管理和移植。

SysTick 是一个 24 位的向下递减的计数器，计数器每计数一次的时间为 $1/SYSCLK$ ，对于 nRF52840 来说，系统时钟为 64M。当重装载数值寄存器的值递减到 0 的时候，SysTick 就产生一次中断，并装入重载值重新开始递减。

在 BLE 的例子里面，很少使用 SysTick，这是因为 nRF52840 实现的很多应用中，对功耗的要求相对较高，所以一般不使用 SysTick，而是使用功耗更低的实时计数器 RTC。对于 nRF52840 开发，下面两种情况适合使用 SysTick。

- 对功耗要求不高的 OS 系统。
- 对功耗要求不高，只需实现简单的定时，对定时精度要求较高。

3. SysTick 寄存器

SysTick 有 4 个寄存器，如下表所示。

表 12-1: SysTick 寄存器

序号	寄存器名	地址	描述
1	CTRL	0xE000E010	SysTick 控制及状态寄存器。
2	LOAD	0xE000E014	SysTick 重装载数值寄存器。
3	VAL	0xE000E018	SysTick 当前数值寄存器。
4	CALIB	0xE000E01C	SysTick 校准数值寄存器。

■ CTRL: SysTick 控制及状态寄存器

表 12-2: CTRL 寄存器

位段	名称	类型	复位值	描述
位 16	COUNTFLAG	读/写	0	如果在上次读取本寄存器后，SysTick 已经

			计到了 0，则该位为 1。读取寄存器值或清除当前寄存器值后会清零。
位 2	CLKSOURCE	读/写 0	时钟源选择位，0=外部时钟源（STCLK），1=内部时钟源。
位 1	TICKINT	读/写 0	1=SysTick 倒数计数到 0 时产生异常请求。 0= SysTick 倒数计数到 0 时不产生异常请求。
位 0	ENABLE	读/写 0	SysTick 定时器使能。

■ LOAD: SysTick 重装载数值寄存器

表 12-3: LOAD 寄存器

位段	名称	类型	复位值	描述
位 23~0	LOAD	读/写 0		计数值递减到零时，将被重装载的值。

■ VAL: SysTick 当前数值寄存器

表 12-4: VAL 寄存器

位段	名称	类型	复位值	描述
位 23~0	CURRENT	读/写 0		读取时返回 SysTick 当前计数值，写入任何数值都会清零寄存器，同时也会清零 SysTick 控制及状态寄存器中的 COUNTFLAG 标志。

■ CTRL: SysTick 控制及状态寄存器

表 12-5: CTRL 寄存器

位段	名称	类型	复位值	描述
位 31	NOREF	读/写 0		如果在上次读取本寄存器后，SysTick 已经计到了 0，则该位为 1。读取寄存器值或清除当前寄存器值后会清零。
位 30	SKEW	读/写 0		时钟源选择位，0=外部时钟源（STCLK），1=内部时钟源。
位 23~0	TENMS	读/写 0		1=SysTick 倒数计数到 0 时产生异常请求。 0= SysTick 倒数计数到 0 时不产生异常请求。

4. 软件设计

4.1. 库函数的应用

Nordic 的 SysTick 库实现的是无中断、阻塞式的定时，同时扩展了定时的时长。使用 SysTick 库函数实现定时时，只需调用 `nrfx_systick_init()` 函数初始化并启动 SysTick，之后调用 `nrfx_systick_delay_ms()` 函数和 `nrfx_systick_delay_us()` 函数实现定时即可，这 3 个库函数的原型如下表所示。

表 12-6: `nrfx_systick_init()` 函数

函数原型	<code>void nrfx_systick_init(void)</code>
函数功能	配置 SysTick 为一个无中断自由运行的定时器并启动该定时器。
参数	无。
返回值	无。

表 12-7: `nrfx_systick_delay_ms()` 函数

函数原型	<code>void nrfx_systick_delay_ms(uint32_t ms)</code>
函数功能	阻塞式延时，单位毫秒。该延时函数消除了 SysTick 最高可能延时值的限制。
参数	[in] ms: 延时的毫秒数。
返回值	无。

表 12-8: `nrfx_systick_delay_us()` 函数

函数原型	<code>void nrfx_systick_delay_us(uint32_t us)</code>
函数功能	阻塞式延时，单位微秒。
参数	[in] us: 延时的微秒数。
返回值	无。

4.2. SysTick 定时闪烁指示灯试验

本实验在“实验 6-3：流水灯(BSP 实现方式)”的基础上修改。使用 SysTick 实现 500ms 定时，并驱动 LED 指示灯 D1 状态翻转，从而达到闪烁 D1 的目的。

❖ 注：本节对应的试验源码是：“实验 12-1：SysTick 定时闪烁指示灯”。

4.2.1. 添加需要的文件

使用 SysTick 时需要加入的文件如下表所示。

表 11-21: SysTick 需要加入的文件

文件名	SDK 中的目录	描述
nrfx_systick.c	...\\modules\\nrfx\\drivers\\src	nrfx_systick 驱动文件。

4.2.2. 头文件引用和路径设置

■ 需要引用的头文件

因为在“main.c”文件中使用了 SysTick，所以“main.c”文件需要引用下面的头文件。

```
#include "nrf_drv_systick.h"
```

■ 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

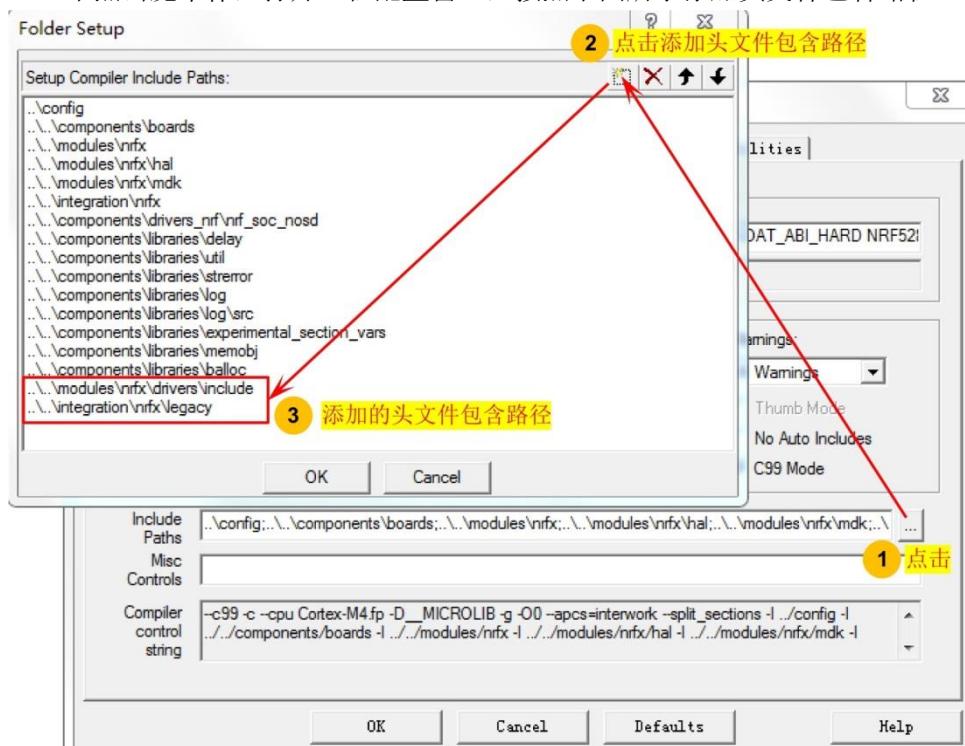


图 12-1: 添加头文件包含路径

SysTick 需要添加的头文件路径如下表：

表 12-2: 头文件包含路径

序号	路径
1	...\\modules\\nrfx\\drivers\\include
2	...\\..\\integration\\nrfx\\legacy

4.2.3. 工程配置

打开“sdk_config.h”文件，加入串口配置，如下图所示，编辑内容时切换到“Text Editor”进行编辑(编辑的内容参考实验的源码，因此代码很长，此处不贴出代码，参见“sdk_config.h”文件的（52~64）行)，编辑完成后，切换到“Configuration Wizard”，即可观察到配置的具

体项目：

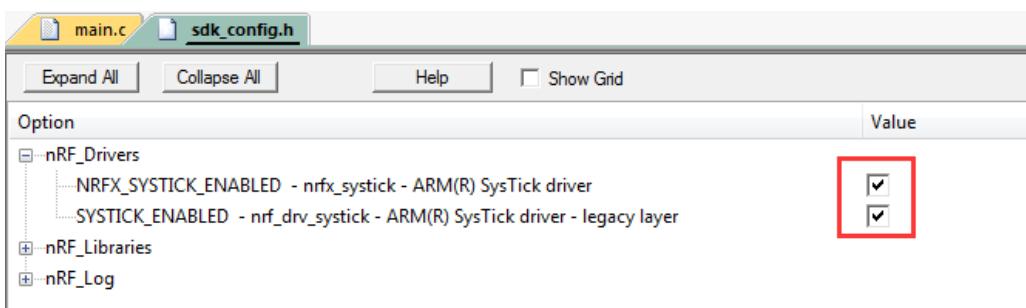


图 12-3：工程配置

4.2.4. 代码编写

本例中首先将 P0.13(指示灯 D1)配置为输出，用于驱动 LED 指示灯 D1，然后调用 nrfx_systick_init() 函数 初始化并启动 SysTick，之后在主循环中调用库函数 nrfx_systick_delay_ms()实现 500ms 定时，并驱动指示灯 D1 状态翻转，即实现指示灯 D1 以 500ms 的间隔闪烁。

代码清单：SysTick 实现 500ms 定时

```

1. /*****
2. * 描 述 : main 函数
3. * 入 参 : 无
4. * 返回值 : int 类型
5. *****/
6. int main(void)
7. {
8.     //配置 P0.13 为输出，用于驱动 D1
9.     nrf_gpio_cfg_output(LED_1);
10.    //将 D1 的初始状态设置为熄灭
11.    nrf_gpio_pin_set(LED_1);
12.    //配置 SysTick 为一个无中断自由运行的定时器并启动该定时器
13.    nrfx_systick_init();
14.
15.    while(true)
16.    {
17.        //翻转指示灯 D1 的状态
18.        nrf_gpio_pin_toggle(LED_1);
19.        //SysTick 延时 500ms
20.        nrfx_systick_delay_ms(500);
21.    }
22. }
```

4.2.5. 硬件连接

本实验需要使用 P0.13 驱动 LED 指示灯 D1，按照下图所示用跳线帽短接 P0.13。

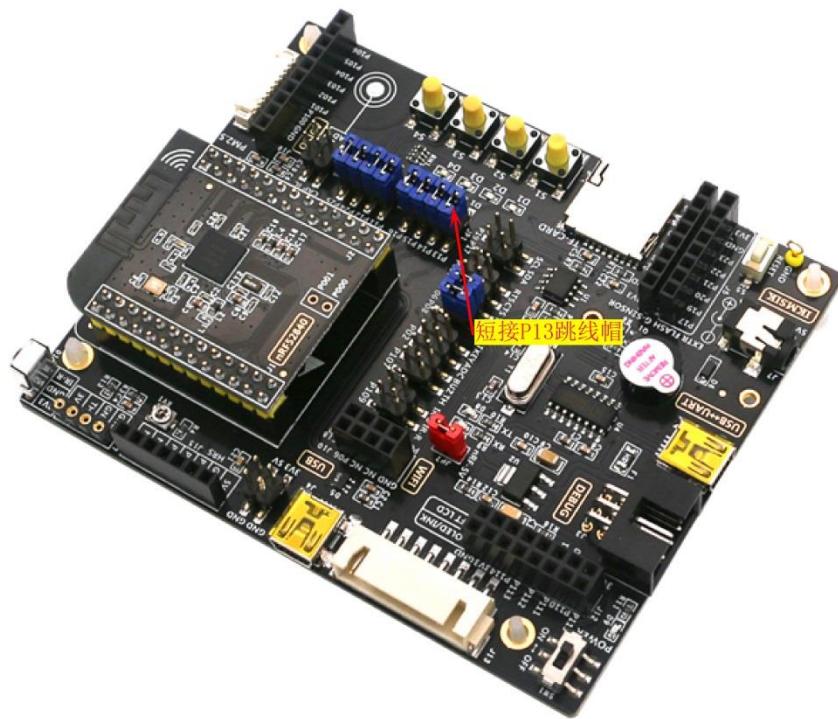


图 12-4：开发板跳线帽短接

4.2.6. 试验步骤

1. 解压“…\3：开发指南（上册）配套实验源码\”目录下的压缩文件“实验 12-1：SysTick 定时闪烁指示灯”，将解压后得到的文件夹“led_blinky_systick”拷贝到合适的目录，如“D\ nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project” 打开“…\led_blinky_systick\project\mdk5” 目录下的工程“led_blinky_systick.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，每 500ms 翻转一次 D1 指示灯的状态，可以观察到开发板上的指示灯 D1 以 500ms 的间隔闪烁。

第十三章：定时/计数器

1. 学习目的

1. 学习 Timer 的原理，理解定时和计数的区别。
2. 掌握 Timer 库函数的应用：Timer 工作于定时模式和计数模式的操作流程及所用库函数的功能。
3. 掌握 Timer 工作于计数模式时计数值的读取方式。

2. TIMER 原理

定时器几乎是每个单片机必有的重要外设之一，可用于定时、精确延时、计数等等，在检测、控制领域有广泛应用。

Timer 运行时不占用 CPU 时间，配置好之后，可以与 CPU 并行工作，实现精确的定时和计数，并且可以通过软件控制其是否产生中断，使用起来灵活方便。

讲解 nRF52840 的 Timer 之前，我们有必要先了解一下定时器和计数器的区别。

■ 定时器和计数器的区别：

定时器和计数器实际都是通过计数器来计数，定时器是对周期不变的脉冲计数（一般来自于系统时钟），由计数的个数和脉冲的周期即可计算出时间，同时，通过一个给定的预期值（即比较值，对应预期的计数值，也就是预期时间），当计数值达到预期值时产生中断，这样就实现了定时，应用程序通过设置不同的预期值实现不同时长的定时。

计数器是对某一事件进行计数，这个事件每发生一次，计数值加/减 1，而这个事件的产生可能是没有规律的。也就是计数器的用途是对事件的发生次数进行计数，由计数值来反映事件产生的次数。

nRF52840 共有 5 个 32 位的 Timer，它们都可以工作于定时和计数两种模式，并且 Timer 的时钟频率和位宽都可以配置，Timer 的原理框图如下。

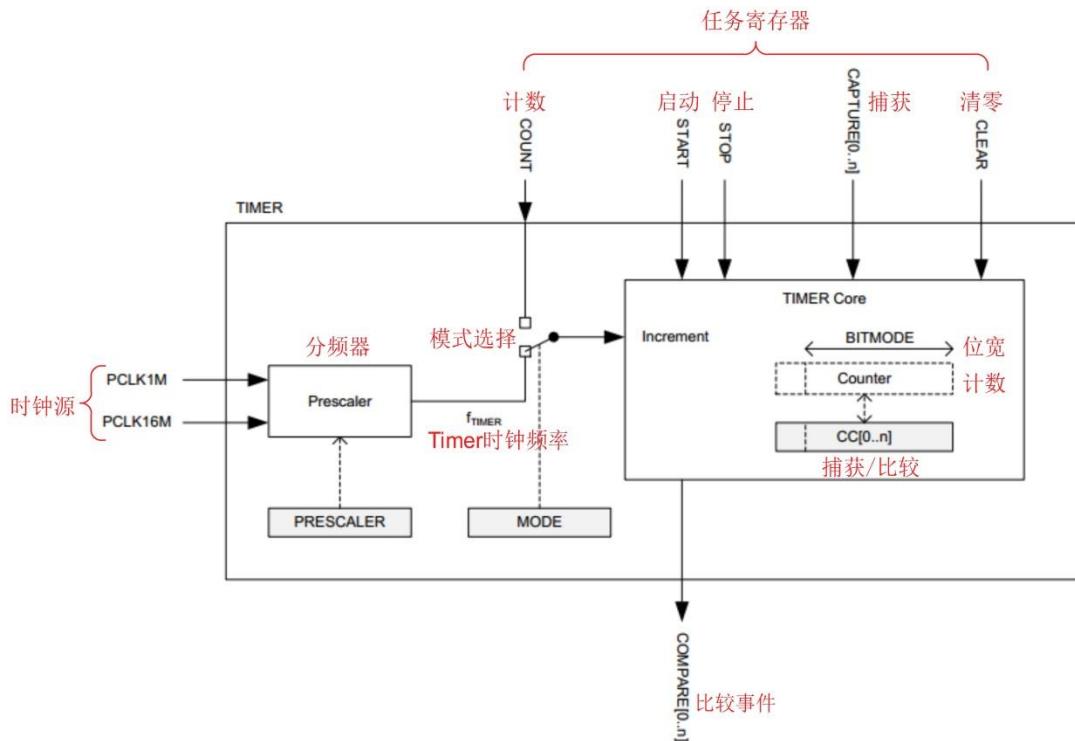


图 13-1: Timer 原理框图

■ Timer 时钟源

Timer 使用的时钟源来自 PCLK1M 或 PCLK16M，系统会根据设置的 Timer 时钟频率 f_{TIMER} 自动选择使用哪一个时钟源，即时钟源的设置是无需软件操作的，当我们设置了 Timer 时钟频率后，系统会根据时钟频率自动选择时钟源，选择的依据如下：

- 当 $f_{\text{TIMER}} > 1 \text{ MHz}$ 时，定时器模块使用 PCLK16M 作为时钟源。
- 当 $f_{\text{TIMER}} \leq 1 \text{ MHz}$ 时，定时器模块使用 PCLK1M 代替 PCLK16M，从而降低功耗。

■ Timer 时钟频率

Timer 的时钟频率 f_{TIMER} 计算公式如下，无论时钟源使用的是 PCLK1M 或 PCLK16M，时钟频率计算时都以 16 MHz 为基准。

$$f_{\text{TIMER}} = 16 \text{ MHz} / (2^{\text{PRESCALER}})$$

分频系数可设置的范围是 0~9，根据上式可计算出时钟频率，时钟频率决定了系统使用的时钟源，下表列出了分频系数的值对应的 Timer 时钟频率和时钟源：

表 13-1: 分频系数的值对应的 Timer 时钟频率

分频系数	时钟频率	时钟源
0	16 MHz	PCLK16M
1	8 MHz	PCLK16M
2	4 MHz	PCLK16M
3	2 MHz	PCLK16M

4	1 MHz	PCLK1M
5	500kHz	PCLK1M
6	250kHz	PCLK1M
7	125kHz	PCLK1M
8	62.5kHz	PCLK1M
9	31.25 kHz	PCLK1M

■ Timer 位宽

nRF52840 的 5 个定时器最大位宽都是 32 位，位宽可以通过 BITMODE 寄存器配置，可配置的位宽有 8 位、16 位、24 位和 32 位。

■ Timer 启动和停止

Timer 可以工作于定时或计数模式，无论在哪种模式下，Timer 都是通过触发 START 任务启动，通过触发 STOP 任务寄存器来停止的。Timer 停止后，触发 START 任务寄存器可以让它再次启动，启动后，定时器将继续从它之前被停止时的值继续计数。

■ 定时和计数

定时模式下，Timer 内部计数寄存器的值在每个时钟脉冲加 1，此模式下，触发任务寄存器 COUNT 不会影响到计数寄存器的值。

计数模式下，每触发一次 COUNT 任务，Timer 内部计数寄存器的值加 1，此模式下，Timer 的时钟频率和分频系数没有使用。

❖ 注意：设置分频系数(PRESCALER 寄存器)和位宽(BITMODE 寄存器)时，必须要停止 Timer 后再设置，否则将会导致不可预知的后果。

■ 溢出和清零

当 Timer 的计数寄存器的值增加到最大值时，计数寄存器溢出，这时计数寄存器的数值会清零并自动从零开始计数。另外，触发任务寄存器 CLEAR，计数寄存器的值也会被清零。

■ 捕获

TIMER 中的每个 CC (比较/匹配) 寄存器都对应一个捕获任务 TASKS_CAPTURE，每次触发任务寄存器 CAPTURE[n]，捕获值会被存储到对应的 CC[n]寄存器。

■ 比较

TIMER 中的每个 CC (比较/匹配) 寄存器都对应一个比较事件 COMPARE，当计数值增加到和 CC[n]寄存器中的数值一样时，产生对应的比较事件 COMPARE[n]。

- 任务延迟和任务优先级
- 任务延迟: Timer 启动之后, 如果触发 CLEAR、COUNT 和 STOP 任务, 这些任务会确保在 PCLK16M 的一个时钟周期内生效。
- 任务优先级: 如果在同一时刻, 也就是在 PCLK16M 的一个时钟周期内同时触发 START 任务和 STOP 任务, STOP 会优先执行。

3. TIMER 寄存器

nRF52840 的 5 个定时器中, Timer0~Timer2 有 4 个捕获/比较寄存器(CC 寄存器), Timer3 和 Timer4 有 6 个 CC 寄存器, 如下表所示:

表 13-2: Timer 基址和拥有的 CC 寄存器数量

外设名	基址	CC 寄存器
TIMER 0	0x40008000	4 个 CC 寄存器 CC[0..3]
TIMER 1	0x40009000	4 个 CC 寄存器 CC[0..3]
TIMER 2	0x4000A000	4 个 CC 寄存器 CC[0..3]
TIMER 3	0x4001A000	6 个 CC 寄存器 CC[0..5]
TIMER 4	0x4001B000	6 个 CC 寄存器 CC[0..5]

表 13-3: 定时器相关寄存器

序号	寄存器名	偏移地址	功能描述
任务寄存器			
1	TASKS_STOP	0x004	停止任务寄存器, 触发后 TIMER 停止。
2	TASKS_COUNT	0x008	计数任务寄存器, 触发后计数值加 1(仅用于计数模式)。
3	TASKS_CLEAR	0x00C	清零任务寄存器, 触发后计数值清零。
4	TASKS_SHUTDOWN	0x010	关闭 TIMER。
5	TASKS_CAPTURE[0]	0x040	触发后, 捕获值将会被拷贝到 CC[0]寄存器。
6	TASKS_CAPTURE[1]	0x044	触发后, 捕获值将会被拷贝到 CC[1]寄存器。
7	TASKS_CAPTURE[2]	0x048	触发后, 捕获值将会被拷贝到 CC[2]寄存器。
8	TASKS_CAPTURE[3]	0x04C	触发后, 捕获值将会被拷贝到 CC[3]寄存器。
9	TASKS_CAPTURE[4]	0x050	触发后, 捕获值将会被拷贝到 CC[4]寄存器。

10	TASKS_CAPTURE[5]	0x054	触发后，捕获值将会被拷贝到 CC[5]寄存器。
事件寄存器			
1	EVENTS_COMPARE[0]	0x140	CC[0]比较匹配事件
2	EVENTS_COMPARE[1]	0x144	CC[1]比较匹配事件
3	EVENTS_COMPARE[2]	0x148	CC[2]比较匹配事件
4	EVENTS_COMPARE[3]	0x14C	CC[3]比较匹配事件
5	EVENTS_COMPARE[4]	0x150	CC[4]比较匹配事件
快捷方式寄存器			
1	SHORTS	0x200	快捷方式寄存器
通用寄存器			
1	INTENSET	0x304	使能中断寄存器
2	INTENCLR	0x308	禁用中断寄存器
3	MODE	0x504	TIMER 模式配置寄存器
4	BITMODE	0x508	TIMER 位宽配置寄存器
5	PRESCALER	0x510	TIMER 分频系数配置寄存器
6	CC[0]	0x540	捕获/比较寄存器[0]
7	CC [1]	0x544	捕获/比较寄存器[1]
8	CC [2]	0x548	捕获/比较寄存器[2]
9	CC [3]	0x54C	捕获/比较寄存器[3]
10	CC [4]	0x550	捕获/比较寄存器[4]
11	CC [5]	0x554	捕获/比较寄存器[5]

■ SHORTS: 快捷方式寄存器

表 13-4: SHORTS 寄存器

位	Field	RW	复位值	描述
位 0	COMPARE0_CLEAR	读/写	0	COMPARE0 事件和 CLEAR 任务之间的快捷方式。 0: 禁止快捷方式。

			1: 使能快捷方式。
位 1	COMPARE1_CLEAR	读/写 0	COMPARE1 事件和 CLEAR 任务之间的快捷方式。 0: 禁止快捷方式。 1: 使能快捷方式。
位 2	COMPARE2_CLEAR	读/写 0	COMPARE2 事件和 CLEAR 任务之间的快捷方式。 0: 禁止快捷方式。 1: 使能快捷方式。
位 3	COMPARE3_CLEAR	读/写 0	COMPARE3 事件和 CLEAR 任务之间的快捷方式。 0: 禁止快捷方式。 1: 使能快捷方式。
位 4	COMPARE4_CLEAR	读/写 0	COMPARE4 事件和 CLEAR 任务之间的快捷方式。 0: 禁止快捷方式。 1: 使能快捷方式。
位 5	COMPARE5_CLEAR	读/写 0	COMPARE5 事件和 CLEAR 任务之间的快捷方式。 0: 禁止快捷方式。 1: 使能快捷方式。
位 8	COMPARE0_STOP	读/写 0	COMPARE0 事件和 STOP 任务之间的快捷方式。 0: 禁止快捷方式。 1: 使能快捷方式。
位 9	COMPARE1_STOP	读/写 0	COMPARE1 事件和 STOP 任务之间的快捷方式。 0: 禁止快捷方式。 1: 使能快捷方式。
位 10	COMPARE2_STOP	读/写 0	COMPARE2 事件和 STOP 任务之间的快捷方式。 0: 禁止快捷方式。 1: 使能快捷方式。

位 11	COMPARE3_STOP	读/写 0	COMPARE3 事件和 STOP 任务之间的快捷方式。 0: 禁止快捷方式。 1: 使能快捷方式。
位 12	COMPARE4_STOP	读/写 0	COMPARE4 事件和 STOP 任务之间的快捷方式。 0: 禁止快捷方式。 1: 使能快捷方式。
位 13	COMPARE5_STOP	读/写 0	COMPARE5 事件和 STOP 任务之间的快捷方式。 0: 禁止快捷方式。 1: 使能快捷方式。

■ INTENSET: 中断使能寄存器

表 13-5: INTENSET 寄存器

位	Field	RW	复位值	描述
位 16	COMPARE0	读/写	0	写“1”使能 COMPARE0 事件中断。 写, 1: 使能。 读, 0: COMPARE0 事件中断已禁止。 读, 1: COMPARE0 事件中断已使能。
位 17	COMPARE1	读/写	0	写“1”使能 COMPARE1 事件中断。 写, 1: 使能。 读, 0: COMPARE1 事件中断已禁止。 读, 1: COMPARE1 事件中断已使能。
位 18	COMPARE2	读/写	0	写“1”使能 COMPARE2 事件中断。 写, 1: 使能。 读, 0: COMPARE2 事件中断已禁止。 读, 1: COMPARE2 事件中断已使能。
位 19	COMPARE3	读/写	0	写“1”使能 COMPARE3 事件中断。 写, 1: 使能。 读, 0: COMPARE3 事件中断已禁止。 读, 1: COMPARE3 事件中断已使能。
位 20	COMPARE4	读/写	0	写“1”使能 COMPARE4 事件中断。

				写, 1: 使能。 读, 0: COMPARE4 事件中断已禁止。 读, 1: COMPARE4 事件中断已使能。
位 21	COMPARE5	读/写	0	写 “1” 使能 COMPARE5 事件中断。 写, 1: 使能。 读, 0: COMPARE5 事件中断已禁止。 读, 1: COMPARE5 事件中断已使能。

■ INTENCLR: 中断禁止寄存器

表 13-6: INTENCLR 寄存器

位	Field	RW	复位值	描述
位 16	COMPARE0	读/写	0	写 “1” 禁止 COMPARE0 事件中断。 写, 1: 禁止。 读, 0: COMPARE0 事件中断已禁止。 读, 1: COMPARE0 事件中断已使能。
位 17	COMPARE1	读/写	0	写 “1” 禁止 COMPARE1 事件中断。 写, 1: 禁止。 读, 0: COMPARE1 事件中断已禁止。 读, 1: COMPARE1 事件中断已使能。
位 18	COMPARE2	读/写	0	写 “1” 禁止 COMPARE2 事件中断。 写, 1: 禁止。 读, 0: COMPARE2 事件中断已禁止。 读, 1: COMPARE2 事件中断已使能。
位 19	COMPARE3	读/写	0	写 “1” 禁止 COMPARE3 事件中断。 写, 1: 禁止。 读, 0: COMPARE3 事件中断已禁止。 读, 1: COMPARE3 事件中断已使能。
位 20	COMPARE4	读/写	0	写 “1” 禁止 COMPARE4 事件中断。 写, 1: 禁止。 读, 0: COMPARE4 事件中断已禁止。 读, 1: COMPARE4 事件中断已使能。
位 21	COMPARE5	读/写	0	写 “1” 禁止 COMPARE5 事件中断。 写, 1: 禁止。

		读, 0: COMPARE5 事件中断已禁止。 读, 1: COMPARE5 事件中断已使能。
--	--	--

■ MODE: Timer 模式配置寄存器

表 13-7: MODE 寄存器

位	Field	RW	复位值	描述
位 1~	MODE	读/写	0	Timer 模式配置
位 0				0: 定时器。 1: 计数器。 2: 低功耗计数器。

■ BITMODE: Timer 位宽配置寄存器

表 13-8: BITMODE 寄存器

位	Field	RW	复位值	描述
位 1~	BITMODE	读/写	0	Timer 位宽配置
位 0				0: 16 位。 1: 8 位。 2: 24 位。 3: 32 位。

■ PRESCALER: 分频系数配置寄存器

表 13-9: PRESCALER 寄存器

位	Field	RW	复位值	描述
位 3~	PRESCALER	读/写	0	Timer 分频系数配置
位 0				取值范围[0~9]。

■ CC[n] (n=0~5): 捕获/比较寄存器

表 13-10: CC[n] (n=0~5) 寄存器

位	Field	RW	复位值	描述
位 32~	CC	读/写	0	捕获/比较值。
位 0				寄存器中的有效位数和位宽寄存器 BITMODE 配置的位数一致。例如位宽配

置为 8 位，CC 寄存器只有位 0~7 有效。

4. 软件设计

4.1. 库函数的应用(定时器)

Timer 外设可工作于定时和计数两种模式，定时模式下要设置捕获/比较寄存器的比较值，这个比较值对应的就是定时的时间，可以同时设置多个捕获/比较寄存器的比较值。Timer 启动之后，计数值在每个时钟频率递增，当计数值达到比较值的时候，即定时时间到了的时候，产生比较匹配事件，在事件处理函数中加入相应的处理代码即可实现定时处理任务的目的，如我们要实现定时翻转指示灯的状态，让指示灯闪烁，那么在事件处理函数中加入翻转指示灯状态的代码即可实现。

Timer 工作于定时模式时的应用流程如下(使用库函数)：

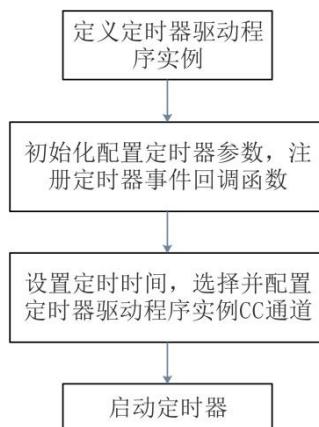


图 13-2: Timer 作为定时器时的应用步骤

4.1.1. 定时器驱动程序实例

Nordic 的定时器库函数要求我们在使用定时器的时候，需要先定义定时器驱动程序实例，驱动程序实例数据结构如下，该结构体有 3 个成员变量，其中 “`p_reg`” 用于关联到一个具体的定时器外设 (Timer0~Timer4)，“`instance_id`” 记录驱动程序实例在定时器驱动程序中的索引，因为驱动程序管理了 5 个定时器外设，所以需要通过索引来标志各个定时器。

“`cc_channel_count`” 是该定时器的捕获/比较通道数量。

代码清单：驱动程序实例数据结构

```

1. typedef struct
2. {
3.     NRF_TIMER_Type * p_reg;           //指向 Timer 外设定义结构体
4.     uint8_t         instance_id;      //驱动程序实例索引
5.     uint8_t         cc_channel_count; //捕获/比较通道数量
6. } nrfx_timer_t;
  
```

定时器库函数中专门提供了一个用于初始化定时器驱动程序实例的宏 NRFX_TIMER_INSTANCE，该宏的输入参数“id”的数值和定时器外设编号对应，如“id”为0，该宏对应的就是定时器0。

代码清单：定时器驱动程序实例初始化宏

```

1. #define NRFX_TIMER_INSTANCE(id) \
2. { \
3.     .p_reg          = NRFX_CONCAT_2(NRF_TIMER, id), \
4.     .instance_id    = NRFX_CONCAT_3(NRFX_TIMER, id, _INST_IDX), \
5.     .cc_channel_count = NRF_TIMER_CC_CHANNEL_COUNT(id), \
6. }
```

- 定时器驱动程序实例定义示例：定义一个名称为 MY_TIMER 的定时器驱动程序实例，该实例使用硬件定时器 Timer0。

```
const nrfx_timer_t MY_TIMER = NRF_DRV_TIMER_INSTANCE(0);
```

4.1.2. 定时器初始化

定义了定时器驱动程序实例后，接下来要做的是初始化定时器，包含配置定时器的参数和注册定时器事件回调函数。定时器初始化是通过调用函数 nrfx_timer_init ()来完成的，函数原型如下：

表 13-11: nrfx_timer_init ()函数

函数原型	<pre>ret_code_t nrfx_timer_init (nrf_drv_timer_t const *const p_instance, nrf_drv_timer_config_t const * p_config, nrf_timer_event_handler_t timer_event_handler)</pre>
函数功能	Timer 初始化函数。
参数	<p>[in] p_instance: 指向定时器驱动程序实例结构体的指针。</p> <p>[in] p_config: 定时器配置结构体，如果是 NULL，使用默认配置参数。</p> <p>[in] timer_event_handler: 事件句柄，不能为 NULL。</p>
返回值	<ul style="list-style-type: none"> • NRF_SUCCESS: 初始化成功。 • NRF_ERROR_INVALID_STATE: 该定时器驱动程序实例已经初始化。 • NRF_ERROR_INVALID_PARAM: 没有提供事件句柄。

nrfx_timer_init ()函数有3个输入参数，其中 p_instance 指向我们定义的定时器驱动程序实例，p_config 指向定时器配置结构体，timer_event_handler 指向定时器事件回调函数。所以在调用 nrfx_timer_init ()函数之前，我们还需要提供定时器配置结构体和定时器事件回调

函数。

1. 配置结构体

驱动程序提供了一个定时器配置结构体 nrfx_timer_config_t，该结构体包含了定时器配置所用的参数，其中时钟频率 frequency 一旦设置后，驱动程序会根据设置的值计算对应的分频系数并将计算得出的分频系数写入到分频系数配置寄存器，也就是应用程序只需关心需要的定时器时钟频率是多少，而不用关心如何设置分频系数。定时器配置结构体声明如下：

代码清单：定时器配置结构体

```
1. typedef struct
2. {
3.     nrf_timer_frequency_t frequency;           //时钟频率
4.     nrf_timer_mode_t      mode;                 //模式
5.     nrf_timer_bit_width_t bit_width;            //位宽
6.     uint8_t               interrupt_priority; //中断优先级
7.     void *                p_context;             //上下文，传递给中断处理函数
8. } nrfx_timer_config_t;
```

一般地，在定义定时器配置结构体时，会使用宏 NRFX_TIMER_DEFAULT_CONFIG 初始化配置结构体，该宏定义了定时器的默认配置，这些配置参数定义在“sdk_config.h”文件中，我们可以根据需要修改“sdk_config.h”文件中的配置参数。

代码清单：定时器配置结构体

```
1. #define NRFX_TIMER_DEFAULT_CONFIG \
2. {                                     \
3.     .frequency  = (nrf_timer_frequency_t)NRFX_TIMER_DEFAULT_CONFIG_FREQUENCY, \
4.     .mode        = (nrf_timer_mode_t)NRFX_TIMER_DEFAULT_CONFIG_MODE,          \
5.     .bit_width   = (nrf_timer_bit_width_t)NRFX_TIMER_DEFAULT_CONFIG_BIT_WIDTH, \
6.     .interrupt_priority = NRFX_TIMER_DEFAULT_CONFIG_IRQ_PRIORITY,  \
7.     .p_context    = NULL  \
8. }
```

打开“sdk_config.h”文件，找到定时器部分，可以看到定时器参数的定义，下面是 NRFX_TIMER_DEFAULT_CONFIG_FREQUENCY 的定义，共有 10 个可配置的值（参见表 13-1），可以看到默认配置的是 0，即 16 MHz。如果我们需要配置为 8MHz，将配置值改为 1 即可。

代码清单：时钟频率的定义

```
1. // <0=> 16 MHz
2. // <1=> 8 MHz
3. // <2=> 4 MHz
4. // <3=> 2 MHz
5. // <4=> 1 MHz
6. // <5=> 500 kHz
```

```

7. // <6=> 250 kHz
8. // <7=> 125 kHz
9. // <8=> 62.5 kHz
10. // <9=> 31.25 kHz
11.
12. #ifndef NRFX_TIMER_DEFAULT_CONFIG_FREQUENCY
13. #define NRFX_TIMER_DEFAULT_CONFIG_FREQUENCY 0
14. #endif

```

- 定时器配置结构体定义示例：定义一个名称为 my_timer_cfg 的定时器配置结构体并初始化。

```
nrfx_timer_config_t my_timer_cfg = NRF_DRV_TIMER_DEFAULT_CONFIG;
```

2. 定时器事件回调函数

定时器事件回调函数的编写格式如下，事件回调函数是在定时器初始化的时候注册给驱动程序的。当比较匹配事件产生后，事件回调函数被执行，在事件回调函数中我们可以判断产生了哪些事件，并添加需要执行的功能代码。

代码清单：定时器事件回调函数

```

1. void timer_led_event_handler(nrf_timer_event_t event_type, void* p_context)
2. {
3.     //判断产生了哪个事件
4.     switch (event_type)
5.     {
6.         //CC[0]比较匹配事件
7.         case NRF_TIMER_EVENT_COMPARE0:
8.             /* 添加功能代码 */
9.             break;
10.            /* 添加 CC[1]~ CC[5]比较匹配事件*/
11.        default:
12.            break;
13.    }
14. }

```

定时器可产生的事件有 6 个，即比较匹配事件 CC[0]~CC[5]，其中 CC[4] 和 CC[5] 仅 Timer3 和 Timer4 拥有，这些事件定义在 nrf_timer_event_t 中。

代码清单：定时器事件

```

1. typedef enum
2. {
3.     //CC[0]比较匹配事件
4.     NRF_TIMER_EVENT_COMPARE0 = offsetof(NRF_TIMER_Type, EVENTS_COMPARE[0]),
5.     //CC[1]比较匹配事件

```

```

6.     NRF_TIMER_EVENT_COMPARE1 = offsetof(NRF_TIMER_Type, EVENTS_COMPARE[1]),
7. //CC[2]比较匹配事件
8.     NRF_TIMER_EVENT_COMPARE2 = offsetof(NRF_TIMER_Type, EVENTS_COMPARE[2]),
9. //CC[3]比较匹配事件
10.    NRF_TIMER_EVENT_COMPARE3 = offsetof(NRF_TIMER_Type, EVENTS_COMPARE[3]),
11. #if defined(TIMER_INTENSET_COMPARE4_Msk) || defined(__NRFX_DOXYGEN__)
12. //CC[4]比较匹配事件, 仅 Timer3 和 Timer4 拥有
13.    NRF_TIMER_EVENT_COMPARE4 = offsetof(NRF_TIMER_Type, EVENTS_COMPARE[4]),
14. #endif
15. #if defined(TIMER_INTENSET_COMPARE5_Msk) || defined(__NRFX_DOXYGEN__)
16. //CC[5]比较匹配事件, 仅 Timer3 和 Timer4 拥有
17.    NRF_TIMER_EVENT_COMPARE5 = offsetof(NRF_TIMER_Type, EVENTS_COMPARE[5]),
18. #endif
19. } nrf_timer_event_t;

```

4.1.3. 配置定时时间和 CC 通道

对于 nRF52840 的定时器来说，工作于定时模式时，设置定时时间也就是设置 CC 寄存器的值，CC 寄存器的值设置后，当定时器的计数值达到 CC 寄存器的值后，即定时时间到达，这时会产生比较匹配事件。

CC 寄存器的值对应的是 Timer 的 ticks(时钟滴答)数，对于给定的定时时间，需要进行计算其对应的 ticks 数量，然后将计算得到的 ticks 数量写入到 CC 寄存器。

1. 计算 ticks

我们计算 ticks 时需要用到下表中的资源，其中时钟频率是通过配置分频系数寄存器设置的，计算公式是 $16 \text{ MHz} / (2^{\text{PRESCALER}})$ ，位宽决定了 CC 寄存器的有效位数，也就是决定了 CC 寄存器可设置的最大值，对应的就是最大定时时间。

表 13-12：计算 ticks 时需要用的资源

序号	资源	作用
1	时钟频率	定时器时钟频率由分频系数决定的，可配置的值有 10 种，参见表 13-1。时钟频率决定了 1 秒包含多个 ticks，也就是时钟频率可以计算出 1 个 tick 占用多长时间。
2	位宽	位宽决定了定时器最长的定时时间。

- 计算示例：时钟频率设置为 16MHz，定时时间 100ms 时，ticks 数值是多少？

$$\text{ticks 数值} = \frac{100}{1000} \div \frac{1}{16 \times 10^6} = 1600000$$

接下来，我们来看一下时钟频率确定后，位宽是如何决定最大定时时间的。

时钟频率设置后，一个 tick 对应的时间也就确定了，上面的示例中，时钟频率设置为 16MHz，那么 1 个 tick 对应的时间是 $(1/1600000)$ 秒，即 0.625us。注意到 CC 寄存器的有效位数是由位宽决定的，如果位宽设置为 8 位，那么 CC 寄存器最大能写入的数值是 255，

也就是最大定时时间是：

$$\text{最大定时时间} = 255 \times 0.625\mu\text{s} = 159.375\mu\text{s}$$

如果位宽设置为 32 位，按照上式可以计算出，最大定时时间为 2684 秒。所以，在配置定时时间的时候，不仅要注意时钟频率的配置，还要注意位宽的配置。

我们使用定时器程序模块的时候，ticks 是不需要我们自己来计算的，驱动程序提供了计算 ticks 的函数 nrfx_timer_ms_to_ticks，该函数会根据我们配置的定时器的参数计算（时钟频率、位宽）出对应的 ticks。

表 13-13: nrfx_timer_extended_compare () 函数

函数原型	<code>_STATIC_INLINE uint32_t nrfx_timer_ms_to_ticks (nrfx_timer_t const *const p_instance, uint32_t time_ms)</code>
函数功能	扩展比较模式下设置 Timer 通道。。
参 数	[in] <code>p_instance</code> : 指向定时器驱动程序实例结构体。 [in] <code>time_ms</code> : 定时时间，单位 ms。
返回值	Ticks 数量。

■ 应用示例：计算定时时间 100ms 对应的 ticks

```
uint32_t time_ticks;  
time_ticks = nrfx_timer_ms_to_ticks(&MY_TIMER, 100);
```

其中，MY_TIMER 是我们定义的定时器驱动程序示例，函数执行后，返回的 ticks 数量保存到变量 `time_ticks`。

2. 配置 CC 通道

Ticks 计算完成后，接下来我们要做的是配置 CC 通道，也就是选择一个 CC 通道来实现定时，配置 CC 通道使用的库函数是 nrfx_timer_extended_compare，原型如下表所示。该函数指定了使用哪个 CC 通道，并将比较值写入到通道的 CC 寄存器，同时设置快捷方式和是否使能中断。

表 13-14: nrfx_timer_extended_compare () 函数

函数原型	<code>void nrfx_timer_extended_compare (nrfx_timer_t const * const p_instance, nrfx_timer_cc_channel_t cc_channel, uint32_t cc_value, nrfx_timer_short_mask_t timer_short_mask, bool enable_int</code>
------	---

)
函数功能	选择并设置定时器驱动程序 CC 通道。
参数	<p>[in] <code>p_instance</code>: 指向定时器驱动程序实例。</p> <p>[in] <code>cc_channel</code>: CC 通道。</p> <p>[in] <code>cc_value</code>: CC 比较值。</p> <p>[in] <code>timer_short_mask</code>: 快捷方式。</p> <p>[in] <code>enable_int</code>: 开启或关闭比较通道的中断。</p>
返回值	<ul style="list-style-type: none"> • <code>NRF_SUCCESS</code>: 初始化成功。 • <code>NRF_ERROR_INVALID_STATE</code>: 该定时器驱动程序实例已经初始化。 • <code>NRF_ERROR_INVALID_PARAM</code>: 没有提供事件句柄。

快捷方式可配置的有下面 2 种类型：

- 1) 比较匹配事件产生时停止定时器：如果我们每次启动定时器后，只需要定时器执行一次定时，可以使用这个快捷方式，当定时时间到达后，产生事件，定时器自动停止。
- 2) 比较匹配事件产生时清零计数值：如果我们需要定时器重复定时，就需要用到这个快捷方式，这样，每次计数值计数到比较值（我们写入到 CC 寄存器中的数值）时，定时器自动清零计数值并重新开始计数。

可配置的快捷方式定义在 `nrf_timer_short_mask_t` 中，如下所示。

代码清单：快捷方式定义

```

1. typedef enum
2. {
3.     //COMPARE0 事件和 STOP 任务之间的快捷方式: COMPARE0 事件产生后自动停止定时器
4.     NRF_TIMER_SHORT_COMPARE0_STOP_MASK = TIMER_SHORTS_COMPARE0_STOP_Msk,
5.     //COMPARE1 事件和 STOP 任务之间的快捷方式: COMPARE0 事件产生后自动停止定时器
6.     NRF_TIMER_SHORT_COMPARE1_STOP_MASK = TIMER_SHORTS_COMPARE1_STOP_Msk,
7.     //COMPARE2 事件和 STOP 任务之间的快捷方式: COMPARE0 事件产生后自动停止定时器
8.     NRF_TIMER_SHORT_COMPARE2_STOP_MASK = TIMER_SHORTS_COMPARE2_STOP_Msk,
9.     //COMPARE3 事件和 STOP 任务之间的快捷方式: COMPARE0 事件产生后自动停止定时器
10.    NRF_TIMER_SHORT_COMPARE3_STOP_MASK = TIMER_SHORTS_COMPARE3_STOP_Msk,
11.
12. #if defined(TIMER_INTENSET_COMPARE4_Msk) || defined(__NRFX_DOXYGEN__)
13.     //COMPARE4 事件和 STOP 任务之间的快捷方式: COMPARE0 事件产生后自动停止定时器，仅
14.     //Timer3 和 Timer4 拥有
15.     NRF_TIMER_SHORT_COMPARE4_STOP_MASK = TIMER_SHORTS_COMPARE4_STOP_Msk,
16. #endif
17.

```

```

18. #if defined(TIMER_INTENSET_COMPARE5_Msk) || defined(__NRFX_DOXYGEN__)
19.     //COMPARE5 事件和 STOP 任务之间的快捷方式: COMPARE0 事件产生后自动停止定时器, , 仅
20.     //Timer3 和 Timer4 拥有
21.     NRF_TIMER_SHORT_COMPARE5_STOP_MASK = TIMER_SHORTS_COMPARE5_STOP_Msk,
22. #endif
23.
24.     //COMPARE0 事件和 CLEAR 任务之间的快捷方式: COMPARE0 事件产生后自动清零计数值
25.     NRF_TIMER_SHORT_COMPARE0_CLEAR_MASK = TIMER_SHORTS_COMPARE0_CLEAR_Msk,
26.     //COMPARE1 事件和 CLEAR 任务之间的快捷方式: COMPARE0 事件产生后自动清零计数值
27.     NRF_TIMER_SHORT_COMPARE1_CLEAR_MASK = TIMER_SHORTS_COMPARE1_CLEAR_Msk,
28.     //COMPARE2 事件和 CLEAR 任务之间的快捷方式: COMPARE0 事件产生后自动清零计数值
29.     NRF_TIMER_SHORT_COMPARE2_CLEAR_MASK = TIMER_SHORTS_COMPARE2_CLEAR_Msk,
30.     //COMPARE3 事件和 CLEAR 任务之间的快捷方式: COMPARE0 事件产生后自动清零计数值
31.     NRF_TIMER_SHORT_COMPARE3_CLEAR_MASK = TIMER_SHORTS_COMPARE3_CLEAR_Msk,
32. #if defined(TIMER_INTENSET_COMPARE4_Msk) || defined(__NRFX_DOXYGEN__)
33.     //COMPARE4 事件和 CLEAR 任务之间的快捷方式: COMPARE0 事件产生后自动清零计数值, 仅
34.     //Timer3 和 Timer4 拥有
35.     NRF_TIMER_SHORT_COMPARE4_CLEAR_MASK = TIMER_SHORTS_COMPARE4_CLEAR_Msk,
36. #endif
37. #if defined(TIMER_INTENSET_COMPARE5_Msk) || defined(__NRFX_DOXYGEN__)
38.     //COMPARE5 事件和 CLEAR 任务之间的快捷方式: COMPARE0 事件产生后自动清零计数值, 仅
39.     //Timer3 和 Timer4 拥有
40.     NRF_TIMER_SHORT_COMPARE5_CLEAR_MASK = TIMER_SHORTS_COMPARE5_CLEAR_Msk,
41. #endif
42. } nrf_timer_short_mask_t;

```

- 应用示例: 配置定时器驱动程序实例 MY_TIMER 的 CC 通道 0, 比较值设置为 time_ticks, 打开比较事件清零计数值的快捷方式, 使能 CC 通道 0 事件中断。

```
nrfx_timer_extended_compare(&MY_TIMER, NRF_TIMER_CC_CHANNEL0,
                            time_ticks, NRF_TIMER_SHORT_COMPARE0_CLEAR_MASK, true);
```

4.1.4. 启动/关闭定时器

定时器配置完成之后, 调用 `nrfx_timer_enable` 函数即可启动定时器开始计时, `nrfx_timer_enable` 函数中通过触发定时器的 START 任务启动定时器。

`nrfx_timer_disable` 函数用于关闭定时器, 函数中通过触发定时器的 SHUTDOWN 任务关闭定时器。前文中讲述定时器寄存器的时候, 提到过定时器有 STOP 任务, 触发后可以停止寄存器, 那么这里为什么不用 STOP 任务来停止定时器, 而是用 SHUTDOWN 关闭定时器? 原因是: nRF52840 芯片硬件存在 BUG, 触发 STOP 任务停止寄存器后, 电流不会降低, 所以, 为了降低电流, 用 SHUTDOWN 任务关闭定时器。

表 13-15: `nrfx_timer_enable()` 函数

函数原型	<code>void nrfx_timer_enable (nrf_drv_timer_t const *const p_instance)</code>
函数功能	扩展比较模式下设置 Timer 通道。
参数	[in] <code>p_instance</code> : 指向定时器驱动程序实例。
返回值	无。

表 13-16: `nrfx_timer_disable()` 函数

函数原型	<code>void nrfx_timer_enable (nrf_drv_timer_t const *const p_instance)</code>
函数功能	关闭定时器。注意该函数是触发 <code>TASKS_SHUTDOWN</code> 任务关闭定时器，而不是触发 <code>STOP</code> 任务停止寄存器。
参数	[in] <code>p_instance</code> : 指向定时器驱动程序实例。
返回值	无。

■ 应用示例：启动定时器

```
nrfx_timer_enable(&MY_TIMER);
```

■ 应用示例：关闭定时器

```
nrfx_timer_disable(&MY_TIMER);
```

4.2. 库函数的应用(计数器)

Timer 工作于计数模式时的应用流程如下(使用库函数)，其中“定义定时器驱动程序实例”、“启动定时器”和定时模式完全一样，参考定时模式中的描述即可。

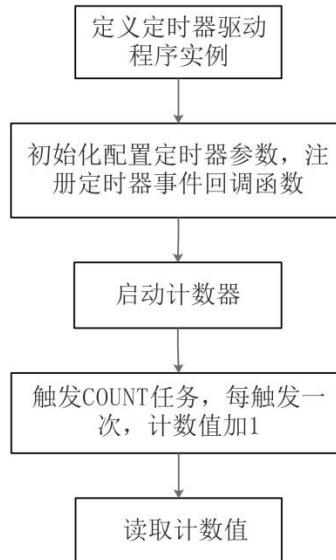


图 13-3：Timer 作为计数器时的应用步骤

4.2.1. 计数模式时的初始化

计数模式时的初始化和定时模式的初始化主要有下面 3 个区别：

1. Timer 配置为计数模式

初始化定时器的时候，会定义定时器配置结构体，并使用初始化配置宏 NRF_DRV_TIMER_DEFAULT_CONFIG 对配置结构体进行初始化。初始化配置宏默认设置 Timer 为定时模式，在这里，我们需要将它修改为计数模式。修改的时候，一般不直接修改初始化配置宏中的 Timer 模式配置，而是对定时器配置结构体的模式配置“mode”进行重写，即将“mode”重写为计数模式，如下所示。

代码清单：配置 Timer 为计数模式

```

1. // 定义定时器配置结构体，并使用默认配置参数初始化结构体
2. nrfx_timer_config_t timer_cfg = NRF_DRV_TIMER_DEFAULT_CONFIG;
3.
4. // Timer0 配置为计数模式
5. timer_cfg.mode = NRF_TIMER_MODE_COUNTER;
  
```

2. 定时器事件回调函数

计数模式因为不会产生事件，因此不需要回调函数，但是定时器初始化函数 nrfx_timer_init() 明确要求必须提供回调函数，也就是它的输入参数 timer_event_handler 不能为“NULL”，所以我们提供一个如下所示的空回调函数用于定时器初始化时注册。

```
void my_timer_event_handler(nrf_timer_event_t event_type, void* p_context){ }
```

3. 计数模式不需要计算定时时间，也不需要去配置 CC 通道。

4.2.2. 计数

计数模式下通过触发 COUNT 任务增加计数值，COUNT 任务有 2 种触发方式：

- 手动触发，手动触发时调用库函数 nrfx_timer_increment()，函数原型如下表所示，该函数中会触发 COUNT 任务将计数值加 1。
- PPI（可编程外设互联）触发：通过将 PPI 连接 COUNT 任务和其它事件连接，当事件产生时自动触发 COUNT 任务。

表 13-17: nrfx_timer_increment() 函数

函数原型	<code>void nrfx_timer_increment((nrf_drv_timer_t const *const p_instance)</code>
函数功能	触发 COUNT 任务，计数值加 1。
参数	[in] <code>p_instance</code> : 指向定时器驱动程序实例。
返回值	无。

4.2.3. 读取计数值

从前文的描述中我们可以看到，nRF52840 的定时器的计数值是保存在计数器 COUNTER 中的，但是定时器并没有提供相应的寄存器，也就是计数器 COUNTER 是程序员不可见的，我们无法直接从计数器 COUNTER 中读取计数值。那么，我们应该如何操作才能读出计数值？

计数值实际是通过 CC[n] (n=0~5) 寄存器读出的，读取计数值的时候，需要先复制再读取。即先触发 TASKS_CAPTURE[n] (n=0~5) 任务，将计数值复制到对应的 CC[n] (n=0~5) 寄存器，之后应用程序从 CC 寄存器中读出计数值，如下图所示。

◆ 注意：仅 Timer3 和 Timer4 有 CC[4]和 CC[5]寄存器。所以对于 Timer0~Timer2 不能触发 TASKS_CAPTURE[4]和 TASKS_CAPTURE[5]任务。

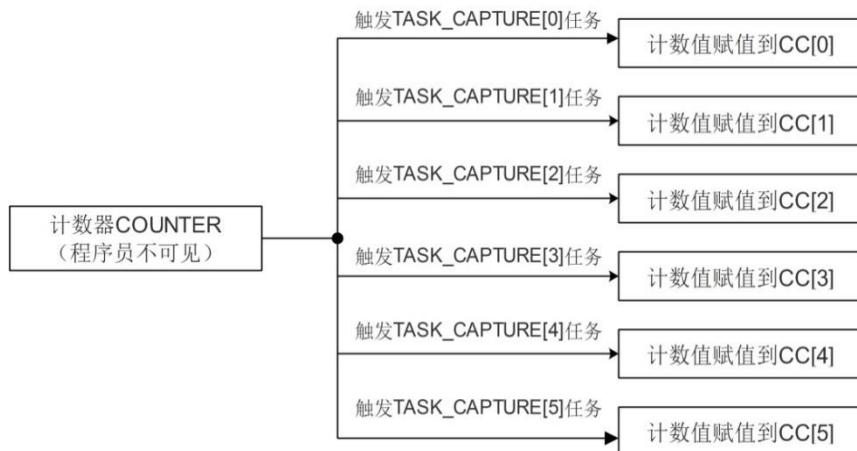


图 13-4: 读计数值

用于读取计数值的库函数是 `nrfx_timer_capture()`, 该函数会触发 `TASKS_CAPTURE` 任务将计数值从 `COUNTER` 复制到指定的 `CC` 寄存器, 然后从 `CC` 寄存器中读出计数值并作为函数的返回值, 应用程序调用该函数后, 通过返回值即可获取计数值。

表 13-18: `nrfx_timer_capture()` 函数

函数原型	<code>uint32_t nrfx_timer_capture((nrfx_timer_t const *const p_instance, nrf_timer_cc_channel_t cc_channel)</code>
函数功能	触发 <code>TASKS_CAPTURE</code> 任务, 读出计数值。
参数	[in] <code>p_instance</code> : 指向定时器驱动程序实例。 [in] <code>cc_channel</code> : 复制计数值的 <code>CC</code> 通道。
返回值	计数值。

4.3. 定时实验

本实验在“实验 6-1: GPIO 输出驱动 led 闪烁”的基础上修改, 配置 Timer0 为定时模式, 定时时间为设置 200ms, 即每 200ms 产生一次比较匹配事件, 比较匹配事件回调函数中翻转 LED 指示灯 D1 的状态, 从而达到驱动指示灯 D1 以 200ms 间隔闪烁的目的。

❖ 注: 本节对应的实验源码是:“实验 13-1: 定时器定时驱动 led 闪烁”。

4.3.1. 添加需要的文件

定时器需要加入的文件如下表所示。

表 13-19: 定时器需要加入的文件

文件名	SDK 中的目录	描述
<code>nrfx_timer.c</code>	<code>..\..\modules\nrfx\drivers\src\</code>	定时器驱动程序。

4.3.2. 头文件引用和路径设置

1. 需要引用的头文件

因为在“`main.c`”文件中使用了定时器, 所以需要引用下面的头文件。

```
#include "nrf_drv_timer.h"
```

2. 需要添加的头文件包含路径

MDK 中点击魔术棒, 打开工程配置窗口, 切换到“C/C++”选项卡, 按照下图所示添加头文件包含路径。

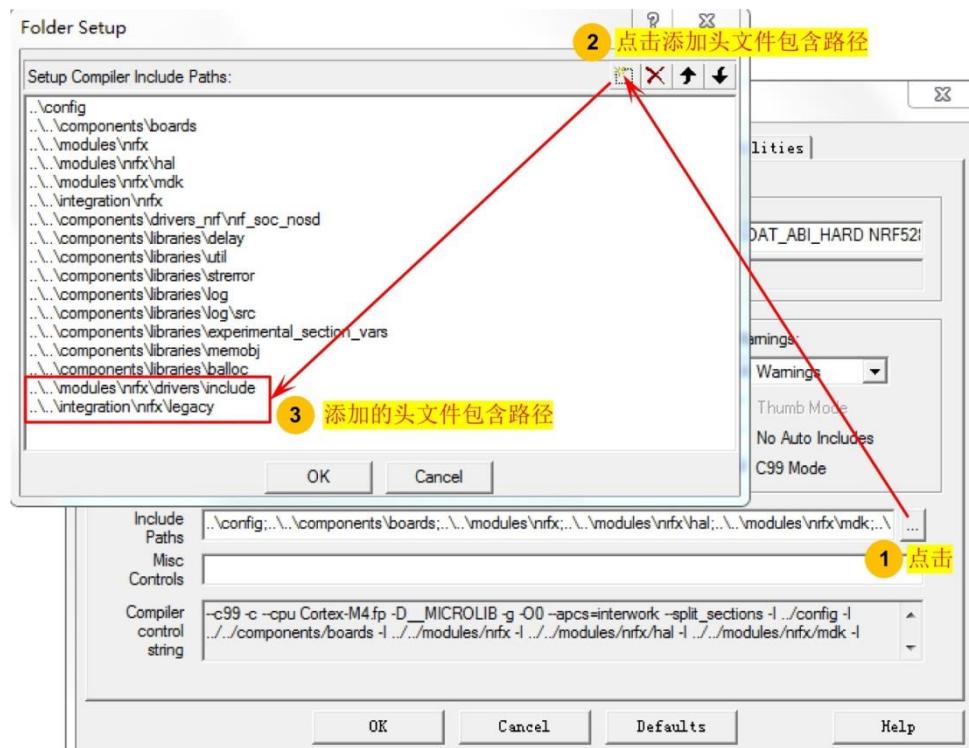


图 13-5：添加头文件包含路径

使用定时器需要添加的头文件路径如下表：

表 13-20：头文件包含路径

序号	路径
1	..\..\integration\nrfx\legacy
2	..\..\modules\nrfx\drivers\include

4.3.3. 工程配置

打开“sdk_config.h”文件，加入定时器配置，如下图所示，编辑内容时切换到“Text Editor”进行编辑（编辑的内容参考实验的源码，因此代码很长，此处不贴出代码，参见“sdk_config.h”文件的(492~731))，编辑完成后，切换到“Configuration Wizard”，即可观察到配置的具体项目：

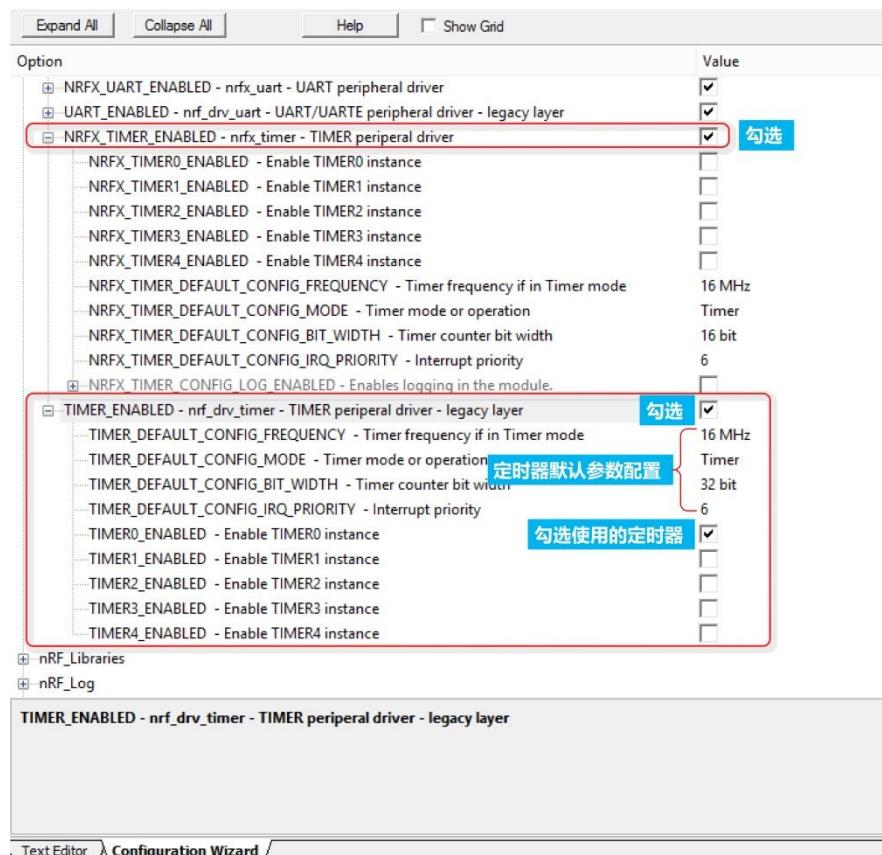


图 13-6：工程配置

4.3.4. 代码编写

根据上文描述，我们使用定时器实现定时的时候，首先定义一个定时器驱动程序实例，程序清单如下，因为本例是定时器驱动 LED，所以为了直观，定时器驱动程序实例命名为 TIMER_LED。

代码清单：定义定时器驱动程序实例

```
1. // 定义 Timer0 的驱动程序实例。驱动程序实例的 ID 对应 Timer 的 ID，如
2. // NRF_DRV_TIMER_INSTANCE(0) 对应 Timer0
3. const nrfx_timer_t TIMER_LED = NRF_TIMER_INSTANCE(0);
```

接下来的是完成定时器初始化的配置，即配置定时器参数（时钟频率、位宽），设置定时时间和配置 CC 通道以及注册定时器事件回调函数。

代码清单：初始化定时器

```
1. // 定时器初始化
2. void timer_init(void)
3. {
4.     uint32_t err_code = NRF_SUCCESS;
5.     uint32_t time_ms = 200; // 定义时间 200ms
6.     uint32_t time_ticks;
7.
```

```

8. //定义定时器配置结构体，并使用默认配置参数初始化结构体
9. nrfx_timer_config_t timer_cfg = NRFX_TIMER_DEFAULT_CONFIG;
10.
11. //初始化定时器，初始化时会注册 timer_led_event_handler 事件回调函数
12. err_code = nrfx_timer_init(&TIMER_LED, &timer_cfg, timer_led_event_handler);

13. APP_ERROR_CHECK(err_code);
14.
15. //定时时间(单位 ms)转换为 ticks
16. time_ticks = nrfx_timer_ms_to_ticks(&TIMER_LED, time_ms);
17. //设置定时器捕获/比较通道及该通道的比较值，使能通道的比较中断
18. nrfx_timer_extended_compare(
19.     &TIMER_LED, NRF_TIMER_CC_CHANNEL0, time_ticks, NRF_TIMER_SHORT_COMPARE0
20.     _CLEAR_MASK, true);
21. }

```

定时器初始化时需要注册定时器事件回调函数，所以应用程序需要提供该回调函数。

代码清单：定时器事件回调函数

```

1. //Timer 事件回调函数
2. void timer_led_event_handler(nrf_timer_event_t event_type, void* p_context)
3. {
4.     switch (event_type)
5.     {
6.         //因为我们配置的是使用 CC 通道 0，所以事件回调函数中判断 NRF_TIMER_EVENT_COMPARE0
7.         //事件
8.         case NRF_TIMER_EVENT_COMPARE0:
9.             //翻转指示灯 D1 状态
10.            nrf_gpio_pin_toggle(LED_1);
11.            break;
12.
13.        default:
14.            break;
15.    }
16. }

```

最后，在主函数中调用定时器初始化函数初始化定时器，初始化后启动定时器，之后定时器开始工作。

代码清单：主函数

```

1. int main(void)
2. {

```

```
3. //配置用于驱动 LED 指示灯 D1 的管脚，即配置 P0.13 为输出
4. nrf_gpio_cfg_output(LED_1);
5. //LED 指示灯 D1 初始状态设置为熄灭，即引脚 P0.13 为输出高电平
6. nrf_gpio_pin_set(LED_1);
7. timer_init(); //初始化定时器
8. nrfx_timer_enable(&TIMER_LED); //启动定时器
9.
10. while(true)
11. {
12.
13. }
14. }
```

4.3.5. 硬件连接

本实验需要使用 P0.13 驱动 LED 指示灯 D1，按照下图所示用跳线帽短接 P0.13。

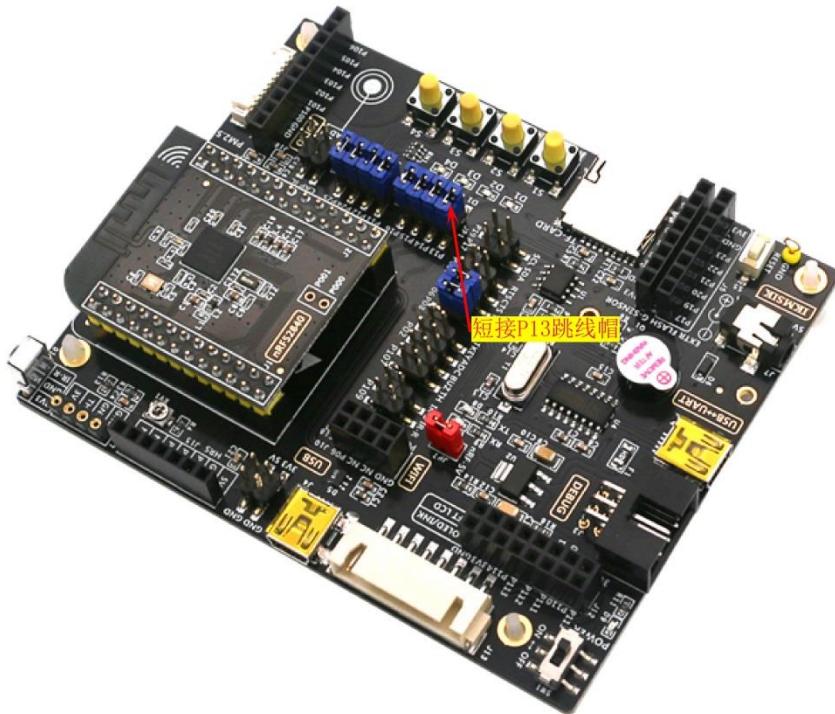


图 13-7：开发板跳线帽短接

4.3.6. 试验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 13-1：定时器定时驱动 led 闪烁”，将解压后得到的文件夹“timer_led”拷贝到合适的目录，如“D\ nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\timer_led\project\mdk5”目录下的工程“timer_led.uvproj”。

4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，定时器 0 每 200ms 产生一次比较匹配事件，事件回调函数中翻转指示灯 D1 的状态，可以观察到开发板上的指示灯 D1 以 200ms 的间隔闪烁。

- **思考题 1：**如何使用定时器 2 实现本试验的功能，即实现定时器 2 定时 200ms 取反一次指示灯 D1 的状态？

4.4. 计数实验

本实验在“实验 10-1：串口数据收发”的基础上修改，配置 Timer0 为计数模式，程序中每隔 1 秒触发一次 Timer0 的 TASKS_COUNT 任务，触发后，计数值加 1，之后读出计数值并通过串口打印出计数值。

❖ 注：本节对应的实验源码是：“实验 13-2：定时器计数”。

4.4.1. 代码编写

本实验中的添加文件、头文件设置等和“实验 13-1”完全一样，参考“实验 13-1”的部分的内容即可。

代码部分我们主要来看一下 Timer 的初始化和读取计数值。初始化 Timer 为计数模式的代码清单如下，这里面我们重写了配置结构体中的定时器模式“mode”，将“mode”配置为计数模式。

代码清单：初始化 Timer 为计数模式

```
1. void timer_init(void)
2. {
3.     uint32_t err_code = NRF_SUCCESS;
4.
5.     //定义定时器配置结构体，并使用默认配置参数初始化结构体
6.     nrfx_timer_config_t timer_cfg = NRF_DRV_TIMER_DEFAULT_CONFIG;
7.
8.     //Timer0 配置为计数模式
9.     timer_cfg.mode = NRF_TIMER_MODE_COUNTER;
10.
11.    //初始化定时器，定时器工作于计数模式时，没有事件，所以无需回调函数，但是 nrfx_timer_init
12.    //函数说明中要求必须提供回调函数，这里我们提供一个空的回调函数用于注册。
13.    //另外，经过实际测试，不提供回调函数，即参数设置为 NULL 也是可以的
14.    err_code = nrfx_timer_init(&TIMER_COUNTER, &timer_cfg, timer_event_handler);
15.    //err_code = nrfx_timer_init(&TIMER_COUNTER, &timer_cfg, NULL);
```

```
15. APP_ERROR_CHECK(err_code);  
16. }
```

计数值读取的方式是：先复制再读取，代码清单如下：

代码清单：触发计数，读取计数值

```
1. while(true)  
2. {  
3.     //触发 COUNTER 任务，定时器计数值加 1  
4.     nrfx_timer_increment(&TIMER_COUNTER);  
5.  
6.     //获取计数值  
7.     timVal = nrfx_timer_capture(&TIMER_COUNTER,NRF_TIMER_CC_CHANNEL2);  
8.     //串口打印计数值  
9.     printf("conuter value: %d\r\n", timVal);  
10.    //延时 1 秒  
11.    nrf_delay_ms(1000);  
12.    //LED 指示灯 D1 状态取反，指示程序的运行  
13.    nrf_gpio_pin_toggle(LED_1);  
14. }
```

4.4.2. 硬件连接

本实验需要使用 P0.13 驱动 LED 指示灯 D1，P0.06 和 P0.08 作为串口通讯引脚，按照下图所示短接跳线帽。

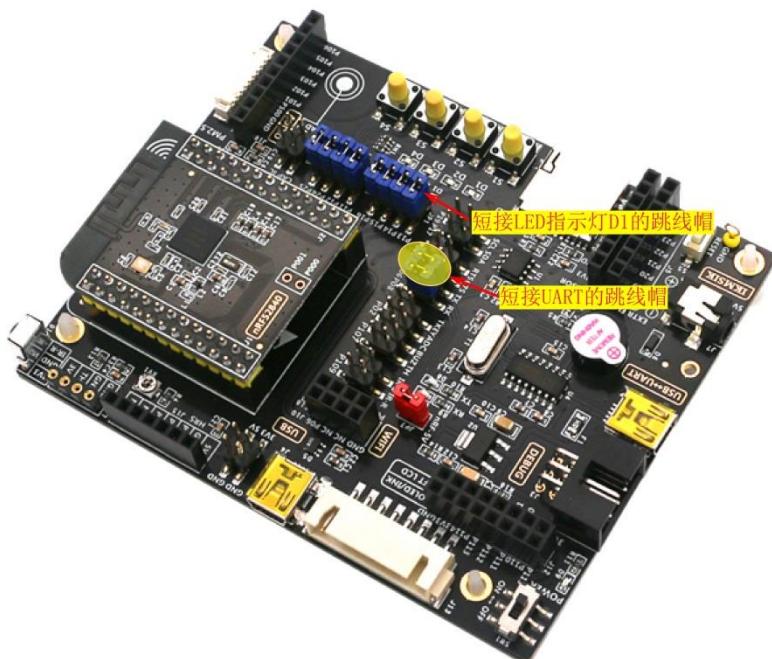


图 13-8：开发板跳线帽短接

4.4.3. 试验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 13-2：定时器计数”，将解压后得到的文件夹“timer_counter”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\timer_counter\project\mdk5”目录下的工程“timer_counter.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qfaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，定时器 0 每秒触发一次 TASKS_COUNT 任务，之后读出计数值并通过串口打印出计数值，在串口调试助手上我们可以观察到计数值每秒加 1。

- **思考题 2：**如何使用定时器 0 的 CC[1]寄存器（本实验使用了 CC[0]寄存器）读取计数值？

第十四章：可编程外设互联 PPI

1. 学习目的

1. 掌握 PPI（可编程外设互联）的原理。
2. 掌握通过外设的库函数获取外设的事件和任务寄存器地址的方法。
3. 掌握 PPI 连接事件和任务、PPI 次级任务以及 PPI 组的应用及使用库函数时的步骤和编程。

2. PPI 原理

2.1. 什么是 PPI

PPI 是可编程外设互联（Programmable Peripheral Interconnect）的缩写，PPI 的作用是提供一个硬件通道，将不同外设的事件和任务“连接”在一起，当事件产生时由硬件自动触发事件“连接”的任务。

PPI 机制对实时应用来说非常重要，它的执行过程由硬件自动完成，省去了原先需要 CPU 参与的步骤，这一方面降低了 CPU 的负荷，另一方面保证了事件触发到任务执行的及时性。

下面我们对比一下一般方式和 PPI 方式实现“定时器定时驱动 LED 闪烁”的不同之处，这更有利于我们直观的了解 PPI 的特点。

- 一般的实现方法：定时时间达到后，产生比较匹配事件，CPU 响应事件中断后，执行事件回调函数，在事件回调函数中驱动 LED，显然，CPU 参与到了这个过程，这里面响应事件中断以及执行功能代码都会占用 CPU 的时间，毫无疑问，这不但增加了 CPU 的负荷，同时也降低了事件触发到任务执行的及时性。
- 使用 PPI 的实现方法：通过 PPI 的一个硬件通道“连接”定时器的比较匹配事件和 GPIOE 的输出任务，定时时间达到后，产生比较匹配事件，这时，比较匹配事件会自动触发 GPIOE 的输出任务驱动 LED，在这个过程中，没有 CPU 的参与，当然不会增加 CPU 的负担，并且，事件触发到任务执行完全由速度更快的硬件直接完成，显然它的效率比一般实现方式更高。

那么，PPI 是如何实现事件和任务之间的“连接”的？

nRF52840 的 PPI 有多个通道，每个通道都包含一个 EEP（Event End-Point：事件端点）和一个 TEP（Task End-Point：任务端点），使用 PPI 连接外设事件和外设任务的时候，将外设事件寄存器的地址写入到 PPI 通道的 EEP，将外设任务寄存器的地址写入到 PPI 通道的 TEP，然后使能该 PPI 通道即可实现外设事件和外设任务的连接，如下图所示：

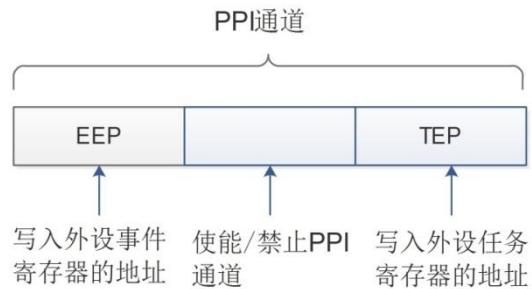


图 14-1: PPI 连接外设的事件和任务

2.2. PPI 原理框图

nRF52840 共有 32 个 PPI 通道，编号为 0~31，其中有 12 个通道(通道 20~31)已经被预编程，剩余的 20 个通道（通道 0~19）是用户可编程的，PPI 的原理框图如下：

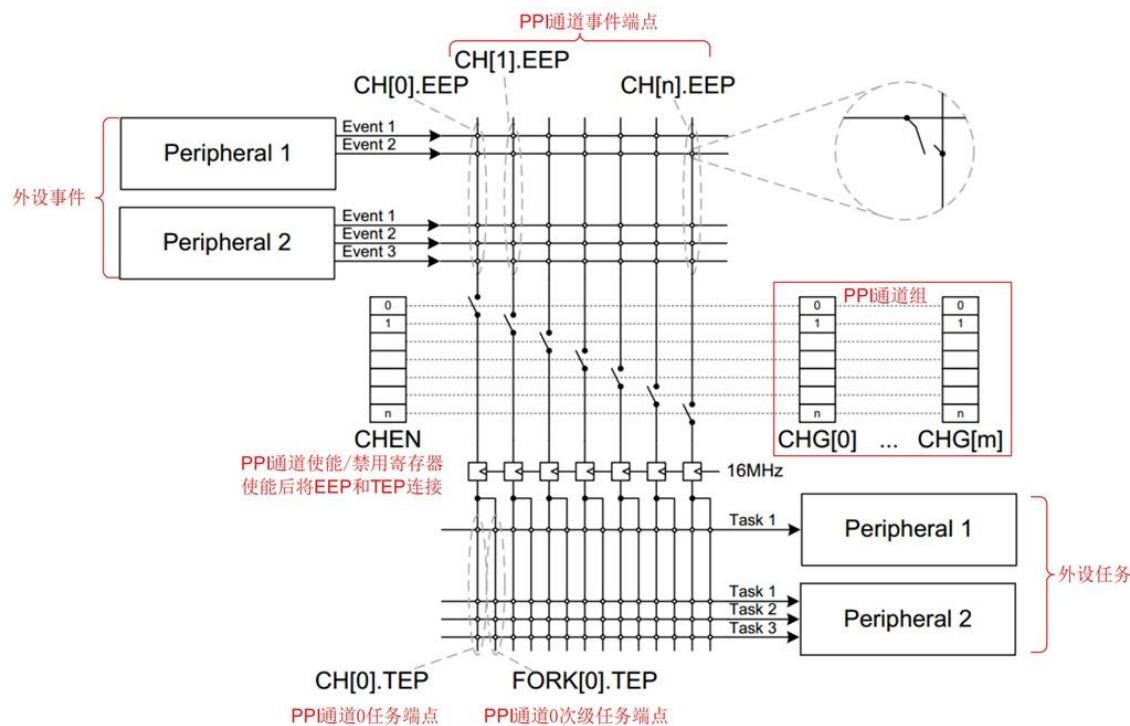


图 14-2: nRF52840 的 PPI 原理框图

从上图中可以看到，一个 PPI 通道由 3 个端点寄存器组成：1 个 EEP 和 2 个 TEP，各个 PPI 通道的事件端点和任务端点之间是“断开的”，当我们把不同外设的事件寄存器和任务寄存器的地址分别写入到一个 PPI 通道的事件端点和任务端点，然后通过 PPI 通道控制寄存器使能该 PPI 通道后，该事件端点和任务端点即通过 PPI 通道连接，PPI 开始工作。

每一个 PPI 通道上，信号都被同步到 16MHz 时钟，这样就避免了内部建立和保持时间的影响。因此，与 16 MHz 时钟同步的事件会被延迟一个时钟周期，而其他异步事件将被延迟一个 16 MHz 的时钟周期。

每个 PPI 通道的 TEP 都包含一个次级任务端点，如 CH[0].TEP 的次级任务端点是 FORK[0].TEP，次级任务端点和任务端点同时被触发，次级任务端点关联的任务通过 FORK 寄存器组配置。

这里需要注意一下，PPI 的控制寄存器有 2 种：

- CHEN 寄存器：这个寄存器的各个位对应 PPI 的通道编号(Bit.16~ Bit.19 为空，因为 PPI 没有编号 16~19 的通道)，如 Bit.0 对应 PPI 通道 0。如果位的值被写为 0，即禁用对应的 PPI 通道，如果位的值被写为 1，即使能对应的 PPI 通道，可以看到，这个寄存器可以使能或禁用 PPI 通道。
- CHENSET 和 CHENCLR 寄存器：这两个寄存器和 CHEN 寄存器一样，也是各个位对应 PPI 的通道编号，和 CHEN 不同的是 CHENSET 只能通过位写 1 使能对应的 PPI 通道(写 0 无效)，而 CHENCLR 只能通过位写 1 禁用对应的 PPI 通道(写 0 无效)。

另外，还需要注意一下：PPI 的任务寄存器也可以像其他外设寄存器一样，通过 PPI 来触发，如 PPI 的组使能寄存器 CHG0EN。

2.3. 预编程 PPI 通道

PPI 中的通道 20~通道 31 已经被预编程，即这些 PPI 通道已经分配给了特定的外设事件和外设任务，CPU 不能对这些通道进行配置，但是 CPU 可以把它们加入到 PPI 组，也可以使能或禁止这些 PPI 通道。也就是说，在程序中，我们不能修改这些 PPI 通道的 EEP 和 TEP，但是我们可以通过配置将这些通道加入到某个 PPI 组，并且还可以像操作其他的 PPI 通道一样使能/禁用这些 PPI 通道，预编程的 PPI 通道如下表所示。

表 14-1：预编程 PPI 通道

通道号	EEP	TEP
20	TIMER0->EVENTS_COMPARE[0]	RADIO->TASKS_TXEN
21	TIMER0->EVENTS_COMPARE[0]	RADIO->TASKS_RXEN
22	TIMER0->EVENTS_COMPARE[1]	RADIO->TASKS_DISABLE
23	RADIO->EVENTS_BCMATCH	AAR->TASKS_START
24	RADIO->EVENTS_READY	CCM->TASKS_KSGEN
25	RADIO->EVENTS_ADDRESS	CCM->TASKS_CRYPT
26	RADIO->EVENTS_ADDRESS	TIMER0->TASKS_CAPTURE[1]
27	RADIO->EVENTS_END	TIMER0->TASKS_CAPTURE[2]
28	RTC0->EVENTS_COMPARE[0]	RADIO->TASKS_TXEN
29	RTC0->EVENTS_COMPARE[0]	RADIO->TASKS_RXEN
30	RTC0->EVENTS_COMPARE[0]	TIMER0->TASKS_CLEAR
31	RTC0->EVENTS_COMPARE[0]	TIMER0->TASKS_START

2.4. PPI 组

nRF52840 共有 6 个 PPI 组 CHG[0]~CHG[5]，PPI 通道可以加入到 PPI 组中集中管理。PPI 组的作用是实现同时使能或禁止组里面的 PPI 通道。

举个例子，如果 PPI 组 CHG[0] 中加入了 3 个 PPI 通道，触发 TASKS_CHG[0].EN 任务后，这 3 个 PPI 通道会同时被使能。触发 TASKS_CHG[0].DIS 任务后，这 3 个 PPI 通道会同时被禁止。

◆ 注意，当一个 PPI 通道同时属于两个 PPI 组：PPI 组 m 和 PPI 组 n 时，如果 CHG[m].EN 和 CHG[n].DIS 同时发生，那么 EN 将被优先执行。

3. PPI 寄存器

PPI 只有任务寄存器和通用寄存器，它是没有事件寄存器的，所以 PPI 是产生不了事件的。

表 14-2: PPI 相关寄存器

序号	寄存器名	偏移地址	功能描述
任务寄存器			
1	CHG[0].EN	0x000	使能 PPI 通道组 0。
2	CHG[0].DIS	0x004	禁用 PPI 通道组 0。
3	CHG[1].EN	0x008	使能 PPI 通道组 1。
4	CHG[1].DIS	0x00C	禁用 PPI 通道组 1。
5	CHG[2].EN	0x010	使能 PPI 通道组 2。
6	CHG[2].DIS	0x014	禁用 PPI 通道组 2。
7	CHG[3].EN	0x018	使能 PPI 通道组 3。
8	CHG[3].DIS	0x01C	禁用 PPI 通道组 3。
9	CHG[4].EN	0x020	使能 PPI 通道组 4。
10	CHG[4].DIS	0x024	禁用 PPI 通道组 4。
11	CHG[5].EN	0x028	使能 PPI 通道组 5。
12	CHG[5].DIS	0x02C	禁用 PPI 通道组 5。
通用寄存器			
1	CHEN	0x500	PPI 通道使能/禁止寄存器。
2	CHENSET	0x504	PPI 通道使能寄存器。

3	CHENCLR	0x508	PPI 通道禁止寄存器。
4	CH[n].EE (n=0~19)	0x510~	PPI 通道 n 事件端点。
5	CH[n].TEP (n=0~19)	0x5AC	PPI 通道 n 任务端点。
6	CHG[n] (n=0~5)	0x800~ 0x814	PPI 通道组 n。
7	FORK[n].TEP (n=0~31)	0x910~ 0x98C	PPI 通道 n 次级任务端点。

■ CHEN: PPI 通道使能/禁止寄存器

CHEN 寄存器用于使能或禁止 PPI 通道。它的各个位对应 PPI 通道编号，写“1”时，对应的 PPI 通道使能，写“0”时，对应的 PPI 通道禁止，注意：预编程 PPI 通道同样可以使能或禁止。

表 14-3: CHEN 寄存器

位	Field	RW	复位值	描述
位 n (n=0~31)	CHn (n=0~31)	读/写	0	使能/禁止 PPI 通道 n (n=0~31)。 1: 使能。 0: 禁止。

■ CHENSET: PPI 通道使能寄存器

CHENSET 寄存器用于使能 PPI 通道。它的各个位对应 PPI 通道编号，位的值写为“1”时使能该通道，写入“0”无效。注意：预编程 PPI 通道同样可以使能或禁用。

表 14-4: CHENSET 寄存器

位	Field	RW	复位值	描述
位 n (n=0~31)	CHn (n=0~31)	读/写	0	写“1”使能 PPI 通道 n (n=0~31)，写“0”无效。 写，1: 使能。 读，0: 通道 n 已禁止。 读，1: 通道 n 已使能。

■ CHENCLR: PPI 通道禁止寄存器

CHENCLR 寄存器用于禁止 PPI 通道。它的各个位对应 PPI 通道编号，位的值写为“1”时禁止该通道，写入“0”无效。注意：预编程 PPI 通道同样可以使能或禁用。

表 14-5: CHENCLR 寄存器

位	Field	RW	复位值	描述
---	-------	----	-----	----

位 n (n=0~31)	CHn (n=0~31) 读/写 0	写“1”禁止 PPI 通道 n (n=0~31), 写“0”无效。 写, 1: 禁止。 读, 0: 通道 n 已禁止。 读, 1: 通道 n 已使能。
--------------	--------------------	--

■ CH[n].EEP (n=0~19): PPI 通道 n 事件端点寄存器

CH[n].EEP 是 PPI 事件端点寄存器, 其中 n={0…19}, 用于写入外设事件寄存器的地址。

表 14-6: CH[n].EEP (n=0~19) 寄存器

位	Field	RW	复位值	描述
位 0~位 31	EEP	读/写	0	事件寄存器地址。注意 EEP 只能写入事件寄存器的地址, 不能写入其他类型寄存器如通用寄存器的地址。

■ CH[n].TEP (n=0~19): PPI 通道 n 任务端点寄存器

CH[n].TEP 是 PPI 任务端点寄存器, 其中 n={0…19}, 用于写入外设任务寄存器的地址。

表 14-7: CH[n].TEP (n=0~19) 寄存器

位	Field	RW	复位值	描述
位 0~位 31	TEP	读/写	0	任务寄存器地址。注意 TEP 只能写入任务寄存器的地址, 不能写入其他类型寄存器如通用寄存器的地址。

■ CHG[n]寄存器 (n=0~5): PPI 通道组 n

CHG[n]寄存器用于 PPI 通道组配置, 其中 n={0…5}。它的各个位对应 PPI 通道编号, 位的值写“1”时表示该通道加入组, 写“0”表示该通道不加入组。注意: 通道 20~31 虽然已经预编程, 但是同样是可以加入组的。

表 14-8: CHENCLR 寄存器

位	Field	RW	复位值	描述
位 m (m=0~31)	CHm (m=0~31)	读/写	0	设置 PPI 组 n 是否包含 PPI 通道 m。 0: 不包含。 1: 包含。

■ FORK[n].TEP 寄存器 (n=0~31): PPI 通道 n 次级任务端点

FORK[n].TEP (n=0~31) 是 PPI 通道 n (n=0~31) 的次级任务端点寄存器，用于写入外设任务寄存器的地址。

表 14-9: FORK [n].TEP (n=0~31) 寄存器

位	Field	RW	复位值	描述
位 0~	TEP	读/写	0	任务寄存器地址。
位 31				

4. 软件设计

4.1. 库函数的应用

PPI 的应用相对比较简单，使用 PPI 库函数的时候，需要初始化 PPI 程序模块，之后向 PPI 驱动程序申请 PPI 通道，申请成功后，应用程序得到驱动程序分配的 PPI 通道，然后将需要连接的事件和任务的地址分别写入到该 PPI 通道的 EEP 和 TEP 并使能该 PPI 通道，这时，PPI 通道就配置完成、开始工作了。如事件端点 EEP 中的外设事件产生就会自动触发任务端点 TEP 中的任务。PPI 应用的流程如下图所示：

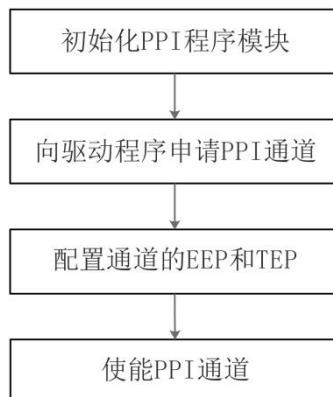


图 14-3: PPI 的应用步骤

4.1.1. 初始化 PPI 程序模块

使用 PPI 库函数时，第一步需要做的是调用 `nrf_drv_ppi_init()` 函数初始化 PPI 程序模块，该函数会检查用于标志 PPI 程序模块状态的变量 `m_drv_state`，如果 PPI 程序模块未初始化，那么设置 `m_drv_state` 为已初始化并返回“初始化成功”，如果 PPI 程序模块已经初始化，则返回“PPI 已被初始化”，并将返回结果交由错误处理程序进行处理。`nrf_drv_ppi_init()` 函数原型如下表所示。

表 14-10: `nrf_drv_ppi_init()` 函数

函数原型	<code>ret_code_t nrf_drv_ppi_init(void)</code>
函数功能	初始化 PPI 程序模块。
参 数	无。

返回值	<ul style="list-style-type: none"> • NRF_SUCCESS: 初始化成功。 • NRF_ERROR_MODULE_ALREADY_INITIALIZED: PPI 程序模块已经初始化。
-----	---

■ 初始化 PPI 程序模块示例:

```
err_code = nrf_drv_ppi_init();
```

4.1.2. 申请 PPI 通道

首先明确一点，PPI 通道是由 PPI 驱动程序来管理的，驱动程序会记录已使用的和空闲的 PPI 通道。应用程序使用 PPI 的时候，自己不能指定使用哪个 PPI 通道，应用程序需要向驱动程序申请空闲的 PPI 通道。

PPI 通道的申请通过 nrfx_ppi_channel_alloc() 函数完成，该函数会查找空闲的 PPI 通道，并将找到的第一个空闲 PPI 通道传递给函数的输出参数 p_channel，这样，应用就得到了驱动程序分配的 PPI 通道。nrfx_ppi_channel_alloc() 函数原型如下表所示：

表 14-11: nrf_drv_ppi_init() 函数

函数原型	<pre>nrfx_err_t nrfx_ppi_channel_alloc (nrf_ppi_channel_t * p_channel)</pre>
函数功能	分配未用的 PPI 通道。调用该函数后，该函数会将第一个查找到的未用的 PPI 通道分配给应用程序。
参 数	[out] p_channel: 指向保存驱动程序分配的 PPI 通道的变量。
返回值	<ul style="list-style-type: none"> • NRF_SUCCESS: PPI 通道分配成功。 • NRF_ERROR_NO_MEM: 没有空闲的 PPI 通道供分配。

■ 应用程序申请 PPI 通道示例：首先定义变量 my_ppi_channel 保存申请到的 PPI 通道，然后调用 nrfx_ppi_channel_alloc() 函数向 PPI 驱动程序申请通道。

```
// 定义保存 PPI 通道的变量
nrf_ppi_channel_t my_ppi_channel;
// 申请 PPI 通道
err_code = nrfx_ppi_channel_alloc(&my_ppi_channel);
```

4.1.3. 配置 PPI 通道的 EEP 和 TEP 及次级任务

1. 配置 PPI 通道的 EEP 和 TEP

应用程序得到 PPI 驱动程序分配的 PPI 通道后，接下来要做的就是将我们需要连接的事件和任务的地址分别写入到该通道的 EEP 和 TEP，即配置 PPI 通道的 EEP 和 TEP。

配置通道的 EEP 和 TEP 需要用到 nrfx_ppi_channel_assign()函数，该函数原型如下表所示。

表 14-12: nrfx_ppi_channel_assign()函数

函数原型	<pre>nrfx_err_t nrfx_ppi_channel_assign (nrf_ppi_channel_t channel, uint32_t eep, uint32_t tep)</pre>
函数功能	设置 PPI 通道的事件端点和任务端点。
参数	<p>[in] channel: 需要配置事件和任务端点的 PPI 通道。</p> <p>[in] eep: 事件端点地址。</p> <p>[in] tep: 任务端点地址。</p>
返回值	<ul style="list-style-type: none"> • NRF_SUCCESS: PPI 通道的事件和任务端点设置成功。 • NRF_ERROR_INVALID_STATE: 该 PPI 通道没有被分配给用户。 • NRF_ERROR_INVALID_PARAM: 该 PPI 通道是用户不可编程的通道。

nrfx_ppi_channel_assign()函数有 3 个输入参数，其中“channel”是应用程序向驱动程序申请的 PPI 通道，申请成功后，PPI 通道也就确定了，“eep”和“tep”是我们需要连接的事件和任务的地址，那么我们怎么知道事件和任务的地址是多少？

我们说的事件和任务实际指的是外设的事件寄存器和任务寄存器，寄存器都是有各自的地址的，这里只需要把需要连接的事件寄存器和任务寄存器的地址写入到“eep”和“tep”就可以了。事件寄存器和任务寄存器的地址，我们可以通过两种方式获取：

1) 通过查数据手册得到事件和任务的地址：

各个外设的事件寄存器地址和任务寄存器地址可以从 nRF52840 的数据手册中查到，寄存器的地址都是基址加上偏移地址，如 GPIOE 通道 0 的 IN 事件，打开 nRF52840 的数据手册，在 GPIOE 的寄存器章节可以看到 GPIOE 的基址是 0x40006000，如下图所示：

Base address	Peripheral	Instance	Description
0x40006000	GPIOE	GPIOE	GPIO Tasks and Events

图 14-4: GPIOE 基址

GPIOE 通道 0 的 IN 事件的偏移地址是 0x100，如下图所示，由此可以得到 GPIOE 通道 0 的 IN 事件的地址是：。

EVENTS_IN[0]	0x100	Event generated from pin specified in CONFIG[0].PSEL
EVENTS_IN[1]	0x104	Event generated from pin specified in CONFIG[1].PSEL
EVENTS_IN[2]	0x108	Event generated from pin specified in CONFIG[2].PSEL
EVENTS_IN[3]	0x10C	Event generated from pin specified in CONFIG[3].PSEL
EVENTS_IN[4]	0x110	Event generated from pin specified in CONFIG[4].PSEL
EVENTS_IN[5]	0x114	Event generated from pin specified in CONFIG[5].PSEL
EVENTS_IN[6]	0x118	Event generated from pin specified in CONFIG[6].PSEL
EVENTS_IN[7]	0x11C	Event generated from pin specified in CONFIG[7].PSEL

图 14-5: GPIOTE 通道 0 的 IN 事件的偏移地址

所以, GPIOTE 通道 0 的 IN 事件的实际地址是: 基址+偏移地址 = 0x40006000+0x100 = 0x40006100。

2) 通过调用外设的获取事件和任务地址的库函数来获取地址

通过数据手册来查找事件和任务寄存器的地址很麻烦, 费时费力, 更简单的方法是调用外设获取事件和任务地址的库函数来获取这些地址。

nRF52840 的外设驱动库都提供了获取该外设事件和任务寄存器地址的库函数, 我们可以直接使用这些库函数来获取地址, 下面是使用库函数获取外设的事件和任务地址的示例。

■ 获取引脚 BUTTON_1 所用的 GPIOTE 输入事件的地址

```
My_address = nrfx_gpiote_in_event_addr_get(BUTTON_1);
```

■ 获取 Timer0 的 COMPARE[0]事件的地址

```
My_address = nrf_drv_timer_event_address_get(&timer0, NRF_TIMER_EVENT_COMPARE0);
```

■ 获取引脚 LED_1 所用的 GPIOTE 输出任务的地址

```
My_address = nrfx_gpiote_out_task_addr_get(LED_1));
```

2. 配置 PPI 通道的次级任务端点

nRF52840 的每个 PPI 通道均有且只有一个次级任务端点, 次级任务端点的作用是让一个事件可以同时触发两个不同的任务。次级任务端点没有专门的“使能/禁止寄存器”, 当次级任务端点中写入外设任务寄存器的地址后, 次级任务端点“使能”, 写入“0”后, 次级任务端点禁止。PPI 库中配置 PPI 通道次级任务端点的函数是 nrfx_ppi_channel_fork_assign(), 其原型如下表所示。

表 14-13: nrfx_ppi_channel_fork_assign()函数

函数原型	<pre>nrfx_err_t nrfx_ppi_channel_fork_assign (nrf_ppi_channel_t channel, uint32_t fork_tep)</pre>
函数功能	设置 PPI 通道的次级任务端点。。

参 数	[in] channel : 需要配置事件和任务端点的 PPI 通道。 [in] tep : 次级任务端点地址, 等于 0 表示清除。
返回值	<ul style="list-style-type: none"> • NRF_SUCCESS: PPI 通道的事件和任务端点设置成功。 • NRF_ERROR_INVALID_STATE: 该 PPI 通道没有被分配给用户。 • NRF_ERROR_INVALID_PARAM: 该 PPI 通道是用户不可编程的通道。 • NRF_ERROR_NOT_SUPPORTED: 函数不支持此功能, 如 nRF51822 是没有次级任务端点的, 如果在 nRF51822 的应用程序中配置次级任务端点, 会返回此错误代码。

4.1.4. 使能 PPI 通道

PPI 通道配置了 EEP 和 TEP 之后, 调用 nrfx_ppi_channel_enable()函数使能该 PPI 通道, 使能之后, PPI 即开始工作。

表 14-14: nrfx_ppi_channel_enable()函数

函数原型	nrfx_err_t nrfx_ppi_channel_enable (nrf_ppi_channel_t channel)
函数功能	使能指定的 PPI 通道。
参 数	[in] channel : 需要使能的 PPI 通道。
返回值	<ul style="list-style-type: none"> • NRF_SUCCESS: PPI 通道使能成功。 • NRF_ERROR_INVALID_STATE: 该 PPI 通道没有被分配给用户。 • NRF_ERROR_INVALID_PARAM: 该 PPI 通道是用户不可编程的通道。

4.1.5. 使用 PPI 通道组

使用 PPI 通道组时, 首先需要申请 PPI 组, 申请成功后, 将需要加入 PPI 组的 PPI 通道加入到 PPI 组, 之后可以通过使能或禁止 PPI 组来实现同时使能或禁止加入到 PPI 组的 PPI 通道。

1. 申请 PPI 组

应用程序申请 PPI 和申请 PPI 通道的过程一样, PPI 组也是由 PPI 驱动程序来管理的, 应用程序通过 nrfx_ppi_group_alloc()函数向 PPI 驱动程序申请 PPI 组, 该函数会查找空闲的 PPI 组, 并将查找到的第一个空闲 PPI 组传递给函数的输出参数 p_group, 这样, 应用就得到了驱动程序分配的 PPI 组。nrfx_ppi_group_alloc()函数原型如下表所示:

表 14-15: nrfx_ppi_group_alloc()函数

函数原型	<pre>nrfx_err_t nrfx_ppi_group_alloc (nrf_ppi_channel_group_t * p_group)</pre>
函数功能	分配未用的 PPI 组。调用该函数后，该函数会将第一个查找到的未用的 PPI 组分配给应用程序。
参数	[out] <code>p_group</code> : 指向保存驱动程序分配的 PPI 组的变量。
返回值	<ul style="list-style-type: none"> • NRF_SUCCESS: PPI 组分配成功。 • NRF_ERROR_NO_MEM: 没有空闲的 PPI 组供分配。

2. PPI 通道加入组和移出组

应用程序申请到 PPI 通道后，接下来可以通过 `nrfx_ppi_channel_include_in_group()` 函数将 PPI 通道加入到 PPI 组。如果需要从 PPI 组移出 PPI 通道，使用 `nrfx_ppi_channel_remove_from_group()` 函数即可将指定的 PPI 通道移出 PPI 组，PPI 通道加入和移出 PPI 组的函数原型如下表所示。

表 14-16: nrfx_ppi_channel_include_in_group()函数

函数原型	<pre>_STATIC_INLINE nrfx_err_t nrfx_ppi_channel_include_in_group (nrf_ppi_channel_t channel, nrf_ppi_channel_group_t group)</pre>
函数功能	指定的 PPI 通道加入指定的 PPI 组。
参数	<p>[in] <code>channel</code>: 加入组的 PPI 通道。</p> <p>[in] <code>group</code>: 指定的 PPI 组。</p>
返回值	<ul style="list-style-type: none"> • NRF_SUCCESS: PPI 通道加入 PPI 组成功。 • NRFX_ERROR_INVALID_PARAM: PPI 组不是有效的组或 PPI 通道不是有效的 PPI 通道。 • NRFX_ERROR_INVALID_STATE: PPI 组未分配。

表 14-17: nrfx_ppi_channel_remove_from_group()函数

函数原型	<pre>_STATIC_INLINE nrfx_err_t nrfx_ppi_channel_remove_from_group (nrf_ppi_channel_t channel, nrf_ppi_channel_group_t group)</pre>
------	--

函数功能	从指定的 PPI 组中移出指定的 PPI 通道。
参数	<p>[in] channel: 移出组的 PPI 通道。</p> <p>[in] group: 指定的 PPI 组。</p>
返回值	<ul style="list-style-type: none"> • NRF_SUCCESS: PPI 通道移出 PPI 组成功。 • NRFX_ERROR_INVALID_PARAM: PPI 组不是有效的组或 PPI 通道不是有效的 PPI 通道。 • NRFX_ERROR_INVALID_STATE: PPI 组未分配。

3. PPI 组的使能和禁止

PPI 组的使能和禁止通过调用库函数 `nrfx_ppi_group_enable()` 和 `nrfx_ppi_group_disable()` 完成，调用后，PPI 组里面所有的 PPI 通道都会被使能或禁止，组使能和禁止函数如下表所示。

表 14-18: `nrfx_ppi_group_enable()` 函数

函数原型	<pre>nrfx_err_t nrfx_ppi_group_enable (nrf_ppi_channel_group_t group)</pre>
函数功能	使能指定的 PPI 组。
参数	[in] group : 需要使能的 PPI 组。
返回值	<ul style="list-style-type: none"> • NRF_SUCCESS: PPI 组使能成功。 • NRFX_ERROR_INVALID_PARAM: PPI 组不是有效的组。 • NRFX_ERROR_INVALID_STATE: PPI 组未分配。

表 14-19: `nrfx_ppi_group_disable()` 函数

函数原型	<pre>nrfx_err_t nrfx_ppi_group_disable (nrf_ppi_channel_group_t group)</pre>
函数功能	禁止指定的 PPI 组。
参数	[in] group : 需要禁止的 PPI 组。
返回值	<ul style="list-style-type: none"> • NRF_SUCCESS: PPI 组禁止成功。 • NRFX_ERROR_INVALID_PARAM: PPI 组不是有效的组。 • NRFX_ERROR_INVALID_STATE: PPI 组未分配。

4.2. PPI 连接 GPIOTE 事件和任务实验

本实验在“实验 11-2: GPIOTE 通道输入”的基础上修改。试验原理如下图所示。配置 P0.11(按键 S1 连接的引脚)为 GPIOTE 输入引脚，并使能其事件模式，配置 P0.13(指示灯 D1 连接的引脚)为 GPIOTE 输出引脚，并使能其任务模式，应用程序申请 PPI 通道后，将 P0.11 所在 GPIOTE 通道的 IN 事件地址写入到 PPI 通道的 EEP，P0.13 所在 GPIOTE 通道的 OUT 任务地址写入到 PPI 通道的 TEP，这样，按键后产生的 IN 事件会触发 GPIOTE OUT 任务，OUT 任务通过 P0.13 驱动指示灯 D1 翻转状态。

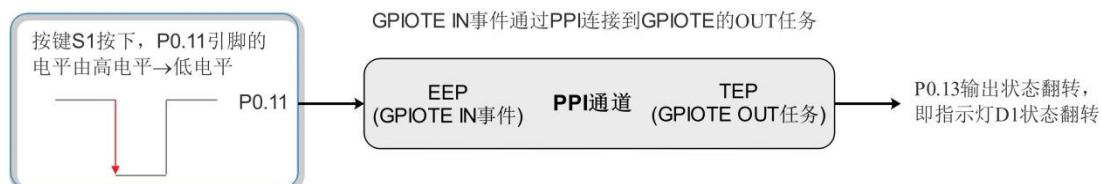


图 14-6：实验原理

❖ 注：本节对应的实验源码是：“实验 14-1：PPI 连接 GPIOTE 事件和任务”。

4.2.1. 添加需要的文件

使用 GPIOTE 程序模块需要加入的文件如下表所示。

表 14-20： GPIOTE 需要加入的文件

文件名	SDK 中的目录	描述
nrfx_ppi.c	...\\modules\\nrfx\\drivers\\src	新 PPI 驱动文件。
nrf_drv_ppi.c	...\\integration\\nrfx\\legacy	旧 PPI 驱动文件

4.2.2. 头文件引用和路径设置

1. 需要引用的头文件

因为在“main.c”文件中使用了 PPI 程序模块，所以“main.c”文件需要引用下面的头文件。

```
#include "nrf_drv_ppi.h"
```

2. 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，切换到“C/C++”选项卡，按照下图所示添加头文件包含路径。

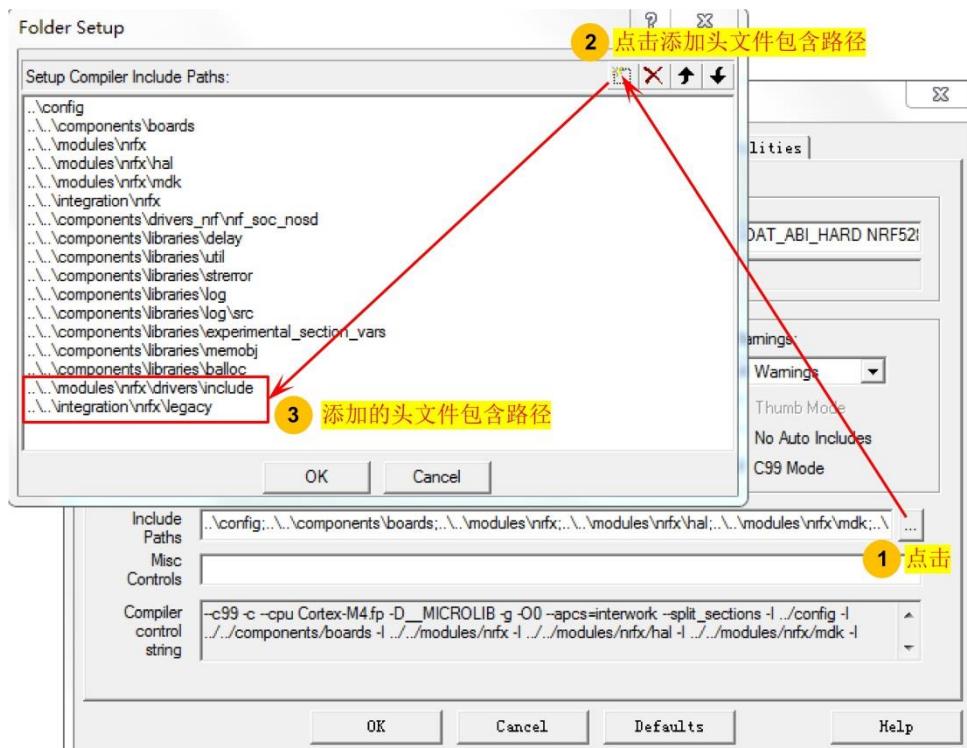


图 14-7: 添加头文件包含路径

GPIOTE 需要添加的头文件路径如下表:

表 14-21: 头文件包含路径

序号	路径
1	..\..\modules\nrfx\drivers\include
2	..\..\integration\nrfx\legacy

4.2.3. 工程配置

打开“ `sdk_config.h`”文件, 加入串口配置, 如下图所示, 编辑内容时切换到“Text Editor”进行编辑(编辑的内容参考实验的源码, 因此代码很长, 此处不贴出代码, 参见“ `sdk_config.h`”文件的(159~215)行和(702~707)行), 编辑完成后, 切换到“Configuration Wizard”, 即可观察到配置的具体项目:

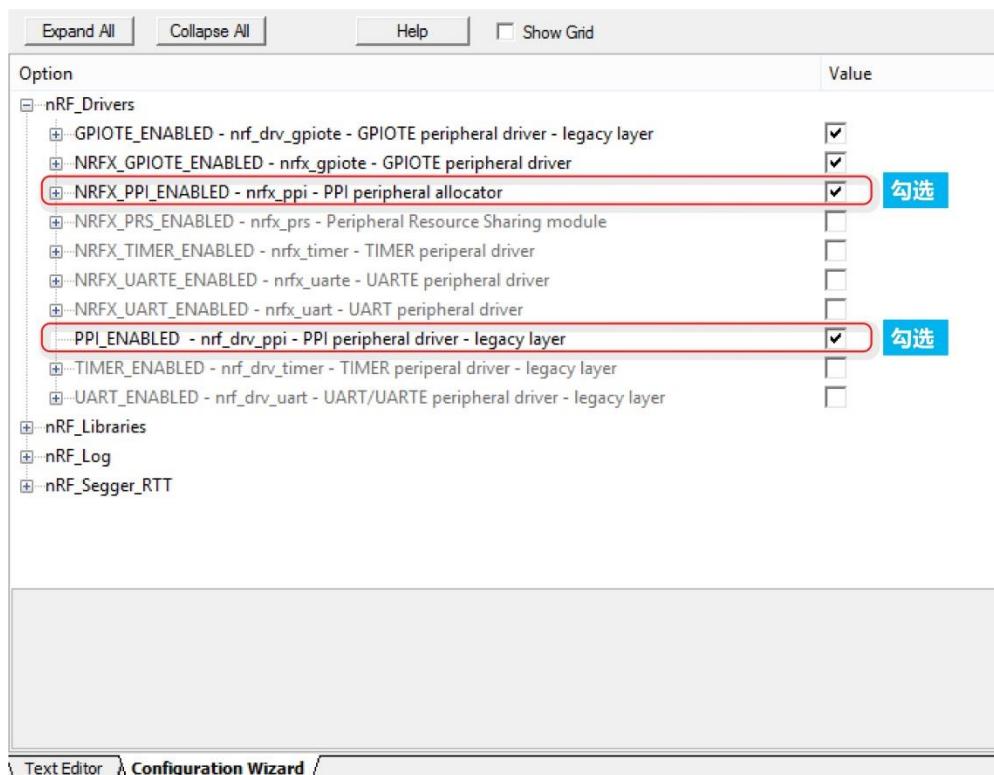


图 14-8：工程配置

4.2.4. 代码编写

使用 PPI 时，首先需要定义一个全局变量 my_ppi_channel 用来保存应用程序向 PPI 驱动程序申请的 PPI 通道，该变量对应的就是具体的 PPI 通道，代码清单如下。

代码清单：定义保存 PPI 通道的变量

```
4. //该变量用来保存应用程序向驱动程序申请的 PPI 通道编号
5. nrf_ppi_channel_t my_ppi_channel;
```

接下来配置 GPIOTE IN 事件和 GPIOTE OUT 任务，即配置 P0.11 为 GPIOTE 输入引脚，配置 P0.13 为 GPIOTE 输出引脚，并使能它们的事件和任务模式，配置完成后，P0.11 和 P0.13 会被分配具体的 GPIOTE 通道，GPIOTE 通道确定后，P0.11 和 P0.13 对应的 IN 事件和 OUT 任务的地址也就确定了。

代码清单：初始化配置 GPIOTE

```
1. void gpiote_init(void)
2. {
3.     ret_code_t err_code;
4.
5.     //初始化 GPIOTE 程序模块
6.     err_code = nrf_drv_gpiote_init();
7.     APP_ERROR_CHECK(err_code);
8.
```

```
9.     //定义 GPIOTE 输出初始化结构体，并对其成员变量赋值
10.    nrf_drv_gpiote_out_config_t out_config = NRFX_GPIOTE_CONFIG_OUT_TASK_TOGGLE(true);
11.    //初始化 GPIOTE 输出引脚，初始化时会分配一个 GPIOTE 通道
12.    err_code = nrfx_gpiote_out_init(LED_1, &out_config);
13.    APP_ERROR_CHECK(err_code);
14.    //使能引脚 LED_1(P0.13)所在 GPIOTE 通道的任务触发
15.    nrf_drv_gpiote_out_task_enable(LED_1);
16.
17.    //以下代码配置 P0.11 作为 GPIOTE 输入，下降沿产生事件
18.    nrf_drv_gpiote_in_config_t in_config = NRFX_GPIOTE_CONFIG_IN_SENSE_HITOLO(true);
19.
20.    //开启 P0.11 引脚的上拉电阻
21.    in_config.pull = NRF_GPIO_PIN_PULLUP;
22.    //配置该引脚为 GPIOTE 输入，因为使用 PPI 连接，所以不需要注册事件回调函数
23.    err_code = nrfx_gpiote_in_init(BUTTON_1, &in_config, NULL);
24.    APP_ERROR_CHECK(err_code);
25.    //使能该引脚所在 GPIOTE 通道的事件模式
26.    nrf_drv_gpiote_in_event_enable(BUTTON_1, true);
27. }
```

gpiote_init()函数中我们确定了 IN 事件和 OUT 任务的地址，地址确定后即可配置 PPI，即初始化 PPI 程序模块，申请 PPI 通道，申请成功后配置 PPI 通道的 EEP 和 TEP（将 IN 事件和 OUT 任务的地址分别写入 EEP 和 TEP），之后使能 PPI 通道。申请 PPI 通道的时候注意对函数的返回值的判断，下面的代码中是通过错误处理单元 APP_ERROR_CHECK 来判断的。

代码清单：初始化配置 PPI

```
1. void ppi_config(void)
2. {
3.     uint32_t err_code = NRF_SUCCESS;
4.
5.     //初始化 PPI 程序模块
6.     err_code = nrf_drv_ppi_init();
7.     APP_ERROR_CHECK(err_code);
8.
9.     //申请 PPI 通道，注意 PPI 通道的分配是由驱动函数完成的，分配的通道号保存到
10.    //my_ppi_channel
11.    err_code = nrfx_ppi_channel_alloc(&my_ppi_channel);
12.    //检查返回值，确定是否申请到了空闲的 PPI 通道
13.    APP_ERROR_CHECK(err_code);
14.    //设置 PPI 通道 my_ppi_channel 的 EEP 和 TEP
15.    err_code = nrfx_ppi_channel_assign(my_ppi_channel,
16.                                      nrfx_gpiote_in_event_addr_get(BUTTON_1),
17.                                      nrfx_gpiote_out_task_addr_get(LED_1));
```

```
18. APP_ERROR_CHECK(err_code);
19. //使能 PPI 通道
20. err_code = nrfx_ppi_channel_enable(my_ppi_channel);
21. APP_ERROR_CHECK(err_code);
22. }
```

4.2.5. 硬件连接

本实验需要使用 P0.13 驱动 LED 指示灯 D1, P0.11 检测 S1 按键的状态，按照下图所示用短接跳线帽。

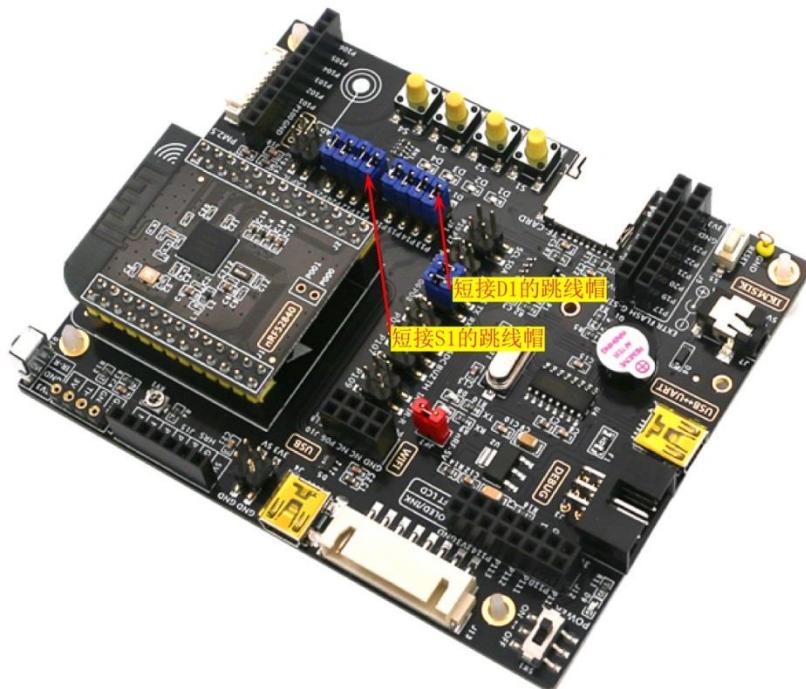


图 14-9：开发板跳线帽短接

4.2.6. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码”目录下的压缩文件“实验 14-1: PPI 连接 GPIO 事件和任务”，将解压后得到的文件夹“ppi_gpiote”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ppi_gpiote\project\mdk5”目录下的工程“ppi_gpiote.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。

6. 程序运行后，每按一次 S1 按键，LED 指示灯 D1 的状态翻转一次（注意：因为本例没有对按键进行软件消抖，所以按动一次可能因为按键抖动的原因产生多次有效输入，导致指示灯状态翻转几次）。
- **思考题 1：**如何同时使用多路 PPI 通道？编写同时使用 4 路 PPI 通道的程序，实现按下按键 S1~S4 后翻转指示灯 D1~D4 状态的功能。
- **思考题 2：**本例使用 PPI 连接 GPIOTE 的 IN 和 OUT 事件，如果使用 PPI 连接 Timer0 比较匹配事件和 GPIOTE 的 OUT 事件，如何实现？

4.3. PPI 次级任务实验

本实验在“实验 13-1：PPI 连接 GPIOTE 事件和任务”的基础上修改。试验原理如下图所示。配置 P0.11(按键 S1 连接的引脚)为 GPIOTE 输入引脚，并使能其事件模式，配置 P0.13(指示灯 D1 连接的引脚) 和 P0.14(指示灯 D2 连接的引脚)为 GPIOTE 输出引脚，并使能其任务模式，应用程序申请 PPI 通道后，将 P0.11 所在 GPIOTE 通道的 IN 事件地址写入到 PPI 通道的 EEP，P0.13 所在 GPIOTE 通道的 OUT 任务地址写入到 PPI 通道的 TEP，P0.14 所在 GPIOTE 通道的 OUT 任务地址写入到 PPI 通道的次级任务端点，这样，按键后产生的 IN 事件会触发两路 GPIOTE OUT 任务，两路 OUT 任务通过 P0.13 和 P0.14 驱动指示灯 D1 和 D2 翻转状态。

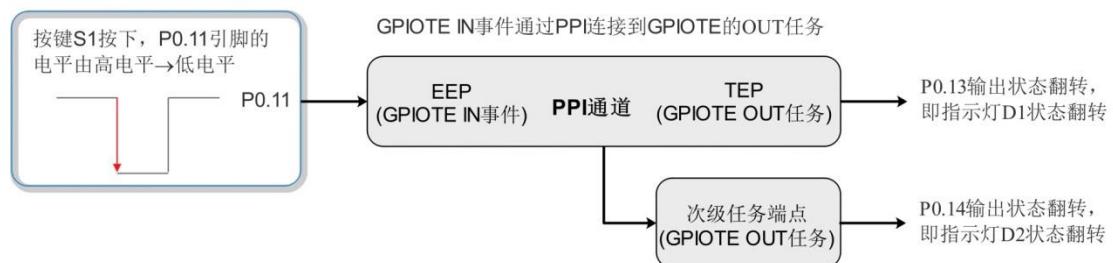


图 14-10：实验原理

❖ 注：本节对应的实验源码是：“实验 14-2：PPI 次级任务”。

4.3.1. 代码编写

和“实验 14-1：PPI 连接 GPIOTE 事件和任务”相比，本实验修改了下面 2 个地方：

- 增加了配置 P0.14(指示灯 D2 连接的引脚)为 GPIOTE 输出引脚，代码清单如下。

代码清单：初始化 P0.14(指示灯 D2 连接的引脚)为 GPIOTE 输出引脚

```

1. //初始化 P0.14(指示灯 D2 连接的引脚)为 GPIOTE 输出引脚，初始化时会分配一个 GPIOTE 通道
2. err_code = nrfx_gpiote_out_init(LED_2, &out_config);
3. APP_ERROR_CHECK(err_code);
4.
5. //使能引脚 P0.14 所在 GPIOTE 通道的任务
  
```

```
6. nrf_drv_gpiote_out_task_enable(LED_2);
```

2. 增加了配置 PPI 通道的次级任务端点，即将 P0.14 所在 GPIOE 通道的 OUT 任务地址写入到 PPI 通道的次级任务端点，代码清单如下。

代码清单：配置 PPI 通道的次级任务端点

```
1. //配置 PPI 通道的次级任务端点
2. err_code = nrfx_ppi_channel_fork_assign(my_ppi_channel,
3.                                         nrf_drv_gpiote_out_task_addr_get(LED_2));
4.
5. //使能 PPI 通道
6. err_code = nrfx_ppi_channel_enable(my_ppi_channel);
7. APP_ERROR_CHECK(err_code);
```

4.3.2. 硬件连接

本实验需要使用 P0.13 和 P0.14 驱动 LED 指示灯 D1 和 D2, P0.11 检测 S1 按键的状态，按照下图所示短接跳线帽。

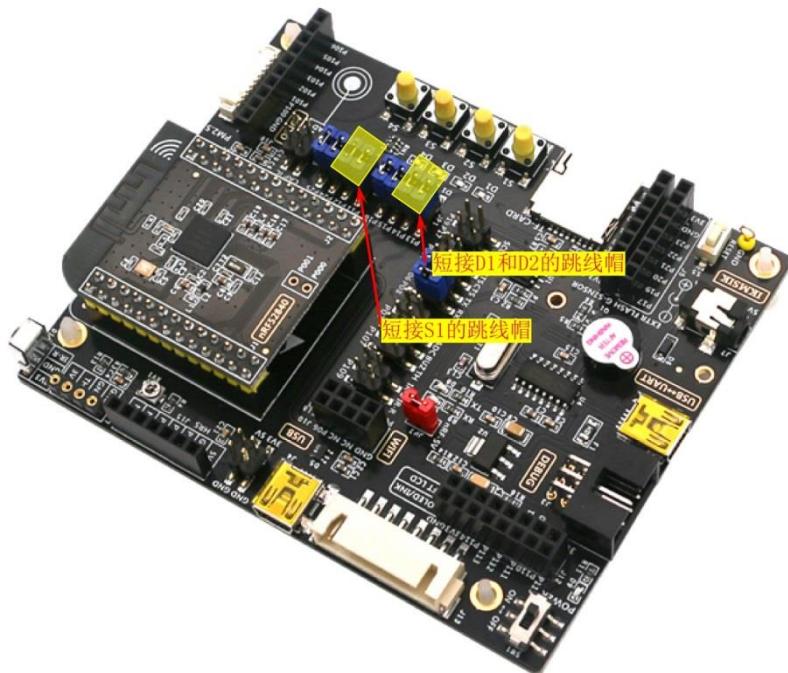


图 14-11：开发板跳线帽短接

4.3.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 13-2: PPI 次级任务”，将解压后得到的文件夹“ppi_fork”拷贝到合适的目录，如“D\ nRF52840”。

2. 启动 MDK5.23。
3. 在 MDK5 中执行 “Project→Open Project” 打开 “…\ppi_fork\project\mdk5” 目录下的工程 “ppi_fork.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件 “nrf52840_qiaa.hex” 位于工程目录下的 “Objects” 文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，每按一次 S1 按键，LED 指示灯 D1 和 D2 的状态翻转一次（注意：因为本例没有对按键进行软件消抖，所以按动一次可能因为按键抖动的原因产生多次有效输入，导致指示灯状态翻转几次）。

4.4. PPI 组实验

本实验在“实验 13-1：PPI 连接 GPIO 事件和任务”的基础上修改。实验中使用 2 个 PPI 通道，并将 2 个 PPI 通道加入到同一个 PPI 组，按键 S3 用来使能 PPI 组，按键 S4 用来禁止 PPI 组。当按下 S3 按键时 PPI 组使能，即 2 个 PPI 通道使能，当按下 S4 按键时 PPI 组禁止，即 2 个 PPI 通道禁止。

❖ 注：本节对应的实验源码是：“实验 13-3：PPI 组”。

4.4.1. 代码编写

本实验在“实验 13-1：PPI 连接 GPIO 事件和任务”的基础上增加了以下内容。

1. 本例中使用了 2 个 PPI 通道和 1 个 PPI 通道组，所以要定义 2 个 nrf_ppi_channel_t 变量，用来保存应用程序向驱动程序申请的 PPI 通道编号的变量，1 个 nrf_ppi_channel_group_t 变量，保存应用程序向驱动程序申请的 PPI 组的编号，代码清单如下。

代码清单：定义保存 PPI 通道和通道组的变量

```
1. //该变量用来保存应用程序向驱动程序申请的 PPI 通道编号  
2. nrf_ppi_channel_t my_ppi_channel1;  
3. nrf_ppi_channel_t my_ppi_channel2;  
4. //该变量用来保存应用程序向驱动程序申请的 PPI 组  
5. nrf_ppi_channel_group_t my_ppi_group;
```

2. 配置 P0.14(按键 S1 连接的引脚)为 GPIO 事件输入引脚，配置 P0.18(指示灯 D2 连接的引脚)为 GPIO 事件输出引脚，代码清单如下。

代码清单：GPIO 事件输入输出引脚配置

```
1. //初始化 P0.14(指示灯 D2 连接的引脚)为 GPIO 事件输出引脚, 初始化时会分配一个 GPIO 通道
```

```
2.     err_code = nrfx_gpiote_out_init(LED_2, &out_config);
3.     APP_ERROR_CHECK(err_code);
4.
5.     //使能引脚 P0.14 所在 GPIOTE 通道的任务触发
6.     nrf_drv_gpiote_out_task_enable(LED_2);
7.
8.     //配置 P0.12 引脚为 GPIOTE 输入，因为使用 PPI 连接，所以不需要注册事件回调函数
9.     err_code = nrfx_gpiote_in_init(BUTTON_2, &in_config, NULL);
10.    APP_ERROR_CHECK(err_code);
11.
12.    //使能 P0.12 引脚所在 GPIOTE 通道的事件模式
13.    nrf_drv_gpiote_in_event_enable(BUTTON_2, true);
```

3. 申请 PPI 通道“my_ppi_channel2”，并连接 BUTTON_2 事件和 LED_2 任务，代码清单如下。

代码清单：申请 PPI 通道连接 BUTTON_2 事件和 LED_2 任务

```
1. //申请 PPI 通道，注意 PPI 通道的分配是由驱动函数完成的，分配的通道号保存到
2. //my_ppi_channel2
3. err_code = nrfx_ppi_channel_alloc(&my_ppi_channel2);
4. //检查返回值，确定是否申请到了空闲的 PPI 通道
5. APP_ERROR_CHECK(err_code);
6. //设置 PPI 通道 my_ppi_channel 的 EEP 和 TEP
7. err_code = nrfx_ppi_channel_assign(my_ppi_channel2,
8.                                     nrfx_gpiote_in_event_addr_get(BUTTON_2),
9.                                     nrfx_gpiote_out_task_addr_get(LED_2));
10. APP_ERROR_CHECK(err_code);
```

4. 申请 PPI 组“my_ppi_group”，并将 PPI 通道“my_ppi_channel1”和“my_ppi_channel2”加入到 PPI 组，代码清单如下。

代码清单：申请和配置 PPI 组

```
1. //申请 PPI 组，注意 PPI 组的分配是由驱动函数完成的，分配的组号保存到 my_ppi_group
2. err_code = nrfx_ppi_group_alloc(&my_ppi_group);
3. APP_ERROR_CHECK(err_code);
4.
5. //PPI 通道 my_ppi_channel 加入到 PPI 组 my_ppi_group
6. err_code = nrfx_ppi_channel_include_in_group(my_ppi_channel1, my_ppi_group);
7. APP_ERROR_CHECK(err_code);
8. //PPI 通道 my_ppi_channel2 加入到 PPI 组 my_ppi_group
9. err_code = nrfx_ppi_channel_include_in_group(my_ppi_channel2, my_ppi_group);
10. APP_ERROR_CHECK(err_code);
```

```
11. //使能 PPI 组 my_ppi_group  
12. err_code = nrfx_ppi_group_enable(my_ppi_group);  
13. APP_ERROR_CHECK(err_code);
```

5. 检测到按键 S3 按下后，使能 PPI 组并点亮指示灯 D3 指示 PPI 组已使能，检测到按键 S4 按下后，禁止 PPI 组并点亮指示灯 D4 指示 PPI 组已禁止，代码如下。

代码清单：使能和禁止 PPI 组

```
1. //检测按键 S3 是否按下  
2. if(nrf_gpio_pin_read(BUTTON_3) == 0)  
3. {  
4.     //D3 点亮, D4 熄灭, 指示: PPI 组使能  
5.     nrf_gpio_pin_clear(LED_3);  
6.     nrf_gpio_pin_set(LED_4);  
7.     while(nrf_gpio_pin_read(BUTTON_3) == 0){} //等待按键释放  
8.     //使能 PPI 组 my_ppi_group  
9.     err_code = nrfx_ppi_group_enable(my_ppi_group);  
10.    APP_ERROR_CHECK(err_code);  
11. }  
12. //检测按键 S4 是否按下  
13. if(nrf_gpio_pin_read(BUTTON_4) == 0)  
14. {  
15.     //D4 点亮, D3 熄灭, 指示: PPI 组禁止  
16.     nrf_gpio_pin_clear(LED_4);  
17.     nrf_gpio_pin_set(LED_3);  
18.     while(nrf_gpio_pin_read(BUTTON_4) == 0){} //等待按键释放  
19.     //禁止 PPI 组 my_ppi_group  
20.     err_code = nrfx_ppi_group_disable(my_ppi_group);  
21.     APP_ERROR_CHECK(err_code);  
22. }
```

4.4.2. 硬件连接

本实验需要使用 P0.13~P0.16 驱动 4 个 LED 指示灯，P0.11 P0.12 P0.24 P0.25 检测 4 个按键的状态，需要用跳线帽短接这些引脚，如下图所示。

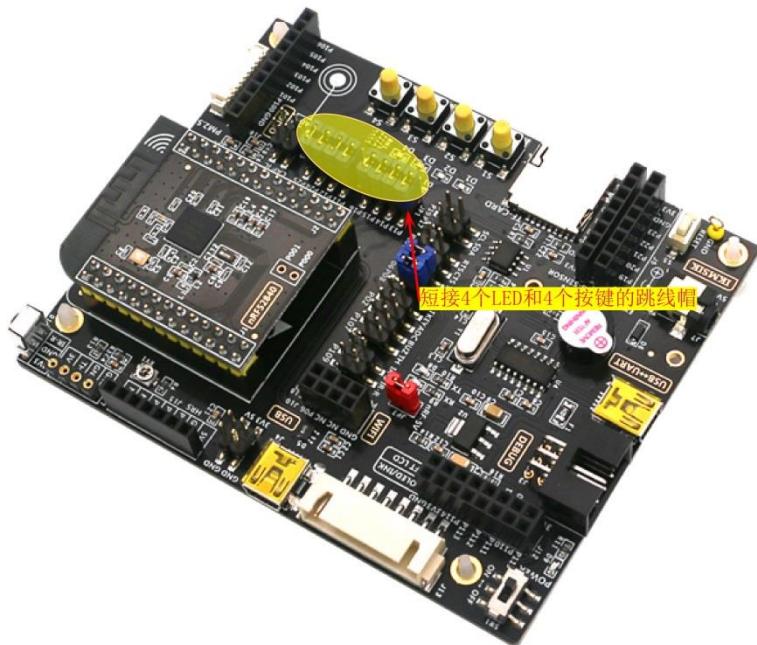


图 14-12：开发板跳线帽短接

4.4.3. 实验步骤

1. 解压“…\3: 开发指南(上册)配套实验源码\”目录下的压缩文件“实验 13-3: PPI 组”，将解压后得到的文件夹“ppi_group”拷贝到合适的目录，如“D\ nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\ppi_group\project\mdk5”目录下的工程“ppi_group.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序启动运行后，PPI 组是没有使能的，这时候按动按键 S1 和 S2 均不会翻转指示灯 D1 和 D2 的状态。按一次 S3 按键，使能 PPI 组，D3 点亮表示 PPI 组已使能，这时候每按动一次按键 S1 或 S2，对应的 D1 和 D2 的状态翻转一次。按一次 S4 按键，禁止 PPI 组，D4 点亮表示 PPI 组已禁止。由此可以看到，执行使能 PPI 组操作会将 PPI 组里面所有的 PPI 通道使能，执行禁止 PPI 组操作会将 PPI 组里面所有的 PPI 通道禁止。

第十五章：非易失性存储器控制器 NVMC

1. 学习目的

1. 掌握 nRF52840-QIAA 片内 Flash 分布。
2. 掌握 Flash 读写和擦除的步骤。

2. NVMC 原理

1. 片内 Flash 分布

开发板上用的 nRF52840 芯片型号是：nRF52840-QIAA，其片内的 Flash 如下表所示：

表 15-1: nRF52840-QIAA 片内 Flash

Flash 总大小	页数	页大小
1M 字节	256	4 k 字节

nRF52840 的 Flash 地址范围是：0x0000 0000~0x0010 0000，共 1M 字节，如下图所示，Flash 区域分为 256 个 4 k 字节大小的页面，擦除操作最小的单位是页面。

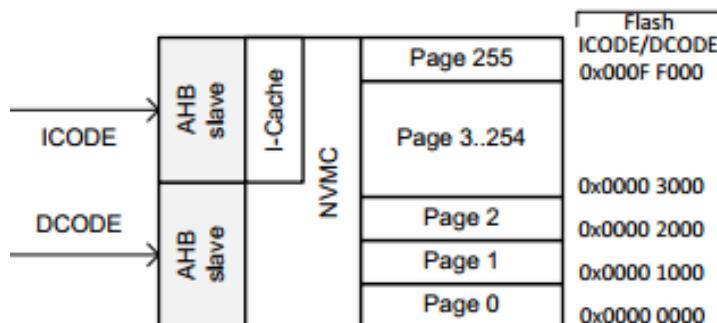


图 15-1: nRF52840 片内 Flash 分布

2. NVMC

nRF52840 的非易失性存储器是通过 NVMC 来管理的，NVMC 是 Non-volatile memory controller（非易失性存储器控制器）的缩写，NVMC 提供了写或擦除 Flash 区域和 UICR（用户信息配置寄存器）的功能。

写 Flash 之前必须通过写寄存器 CONFIG.WEN 来使能 Flash 的写操作，同样，擦除 Flash 之前必须通过写寄存器 CONFIG.EEN 来使能 Flash 的擦除操作。并且，对 Flash 进行操作的时候要特别注意，不能同时使能 Flash 的写和擦除，否则将导致无法预测的后果。

1) 写 Flash

- 使能 Flash 的写后，每次写入一个字(32 位)到字对齐的地址上。
- NVMC 只能将 Flash 中的各个位由“1”写成“0”，而不能将“0”写成“1”，这也是为什么 Flash 写之前要进行擦除操作，擦除操作可将 Flash 的各个位写成“1”。

- 写 Flash 时，只允许写字操作，写入字节或半字都会导致硬件错误，所以，当我们写入的数据不够一个字时，应该按照一个字来处理。
- 写一个字所需要的时间是： t_{WRITE} ，在写的过程中，CPU 是被挂起的。
- 2) 擦除 Flash 的页面
 - 使能 Flash 擦除后，可通过 ERASEPAGE 寄存器来按页擦除 Flash。
 - 擦除页面时候，向 ERASEPAGE 寄存器写入这个页面的起始地址即可擦除该页面。
 - 擦除一个页面所需要的时间是： $t_{ERASEPAGE}$ ，在擦除的过程中，CPU 是被挂起的。
- 3) 写用户信息配置寄存器(UICR)
 - 用户信息配置寄存器 UICR 的写入方式和 Flash 一样。
 - 写 UICR 后，只有复位后新的值才会生效。
 - 写 UICR 所需要的时间是： t_{WRITE} ，在写的过程中，CPU 是被挂起的。
- 4) 擦除用户信息配置寄存器(UICR)
 - 使能擦除后，通过 ERASEUICR 寄存器来擦除 UICR。
 - 擦除 UICR 后，UICR 所有的位被设置成“1”。
 - 擦除 UICR 所需要的时间是： $t_{ERASEPAGE}$ ，在擦除的过程中，CPU 是被挂起的。
- 5) 全片擦除(Erase all)
 - 使能擦除后，通过 ERASEALL 寄存器一次全部擦除 Flash 和 UICR。
 - 全片擦除所需要的时间是： $t_{ERASEALL}$ ，在擦除的过程中，CPU 是被挂起的。
- 6) 访问端口保护行为

启用访问端口保护后，将阻止部分 NVMC 功能，以防止故意或无意擦除 UICR

表 15-2: NVMC 保护

端口访问 保护	CTRL-AP ERASEALL	NVMC ERASEPAGE	NVMC ERASEPAGE PARTIAL	NVMC ERASEALL	NVMC ERASEUICR
使能	允许	允许	允许	允许	允许
关闭	允许	允许	允许	允许	阻止

7) 页面部分擦除(Erase all)

NVMC 具有部分擦除的功能，该功能可将页擦除时间拆分为更短的片段，因此可用于防止在时间要求严格的应用中出现更长的 CPU 停顿。部分擦除仅适用于 Flash 中的代码区域，不适用于 UICR。

当擦除使能后，部分擦除 Flash 页面通过写 ERASEPAGEPARTIAL 开始，部分擦除 Flash 页面的持续时间可在 ERASEPAGEPARTIALCFG 中配置。当擦除时间到达 $t_{ERASEPAGE}$ 后，一个 Flash 页面被擦除。使用 ERASEPAGEPARTIAL N 次，使 $N * ERASEPAGEPARTIALCFG \geq t_{ERASEPAGE}$ ，其中 $N * ERASEPAGEPARTIALCFG$ 给出累计（总）擦除时间，每次累积擦除时间达到 $t_{ERASEPAGE}$ ，计为一个擦除周期。

擦除完成后，Flash 中所有的位设置为“1”，在页面部分擦除的过程中，如果 CPU 从 Flash 中执行代码，CPU 是被挂起的

8) I-Cache

NVMC 的ICODE 总线具有 2048 字节的 I-Cache(指令高速缓存), Cache 命中是从 Cache 中提取指令, 它具有 0 个等待状态延迟。Cache 缺失是指指令不存在于 Cache 中, 需要从 Flash 中取指令, 这种情况下所需花费的等待状态数取决于 CPU 的频率。

使能 Cache 可以提高 CPU 的性能, 降低功耗, 因为它可以减少等待周期和访问 Flash 的次数, 当然, 这取决于 Cache 的命中率。Cache 使能后, 本身也会消耗电流, 如果减少等待周期和访问 Flash 的次数降低的平均电流大于 Cache 自身消耗的电流, 那么执行程序代码的平均电流将减少。Cache 禁止后, 将不再消耗电流, 同时 Cache 中的内容也不会保存。

可以通过 *ICACHECNF* 寄存器启用 Cache 分析来分析程序的 Cache 性能, 使能 Cache 分析后, *IHIT* 和 *IMISS* 寄存器分别用来记录 Cache 命中和未命中的次数, *IHIT* 和 *IMISS* 寄存器在达到最大值后会清零重新计数, 所以分析时间不宜过长, 否则无法获得正确的值。

Flash 是有使用寿命的, nRF52840 片内 Flash 可擦除/写的次数是 10000 次。写和擦除所需的时间如下表所示。

表 15-3: 写和擦除所需的时间

符号	描述	最小值	典型值	最大值	单位
tWRITE	写一个字	—	—	41	μs
tERASEPAGE	擦除一个页面	—	—	85	ms
tERASEALL	全部擦除	—	—	169	ms
nENDURANCE	每个页面擦除次数	10000			

◆ 写和擦除所需的时间参数适用于使用 HFxo 时, 使用 HFINT 时, 时序根据 HFINT 精度而变化。

3. NVMC 寄存器

表 15-3: NVMC 相关寄存器

序号	寄存器名	偏移地址	功能描述
通用寄存器			
1	READY	0x400	就绪标志寄存器。
2	CONFIG	0x504	配置寄存器。
3	ERASEPAGE	0x508	页擦除寄存器。
4	ERASEPCR1	0x508	代码区域页面擦除寄存器, 该寄存器等同于 ERASEPAGE 寄存器 (不推荐使用)。
5	ERASEALL	0x50C	全片擦除 (包含 UICR) 寄存器。

6	ERASEPCR0	0x510	代码区域页面擦除寄存器，该寄存器等同于 ERASEPAGE 寄存器（不推荐使用）。
7	ERASEUICR	0x514	UICR 擦除寄存器。
8	ICACHECNF	0x540	I-Code cache 配置寄存器
9	IHIT	0x548	I-Code cache 命中计数器。
10	IMISS	0x54C	I-Code cache 缺失计数器。

■ READY: 就绪标志寄存器

READY 寄存器用来标志 NVMC 是否就绪。

表 15-4: READY 寄存器

位	Field	RW	复位值	描述
位 0	READY	只读	0	NVMC 就绪/忙。 0: NVMC 忙：NVMC 正在实行写或擦除操作。 1: NVMC 就绪。

■ CONFIG: 配置寄存器

表 15-5: CONFIG 寄存器

位	Field	RW	复位值	描述
位 1~位 0	WEN	读/写	0	Flash 访问方式配置。 0: 只读。 1: 使能写。 2: 使能擦除

■ ERASEPAGE: Flash 页擦除寄存器

ERASEPAGE 寄存器用来擦除 Flash 区域的页面，向 ERASEPAGE 寄存器写入需要擦除的页面的起始地址即可擦除该页面，擦除页面之前必须置位 CONFIG.EEN。如果擦除的页面地址超过了 nRF52840 Flash 的地址范围将会导致不可预料的错误。

表 15-6: ERASEPAGE 寄存器

位	Field	RW	复位值	描述
位 0~位 31	ERASEPAGE	读/写	0	Flash 访问方式配置。 0: 只读。 1: 使能写。 2: 使能擦除

■ ERASEALL: Flash 全片擦除寄存器

表 15-7: ERASEALL 寄存器

位	Field	RW	复位值	描述
位 0~位 31	ERASEALL	读/写	0	全片擦除, 包含 UICR。全片擦除前必须配置 CONFIG 寄存器使能擦除操作。 0: 无操作。 1: 开始全片擦除。

■ ERASEUICR: UICR 擦除寄存器

表 15-8: ERASEUICR 寄存器

位	Field	RW	复位值	描述
位 0	ERASEUICR	读/写	0	擦除 UICR。擦除前必须配置 CONFIG 寄存器使能擦除操作。 0: 无操作。 1: 开始擦除 UICR。

■ ICACHECNF: I-Code cache 配置寄存器

ICACHECNF 寄存器用来使能/禁止 Cache 和 Cache 分析。

表 15-9: ICACHECNF 寄存器

位	Field	RW	复位值	描述
位 0	CACHEEN	读/写	0	CACHE 使能。 0: 禁止 CACHE。所有缓存条目无效。 1: 使能 CACHE。
位 8	CACHEPROFEN			CACHE 分析使能。 0: 禁止。 1: 使能。

■ IHIT: Cache 命中计数寄存器

IHIT 寄存器记录 Cache 命中次数, 当达到最大值时自动清零重新开始计数。

表 15-10: IHIT 寄存器

位	Field	RW	复位值	描述
位 0~位 31	HITS	读/写	0	Cache 命中次数。

■ IMISS: Cache 缺失计数寄存器

MISSES 寄存器记录 Cache 缺失次数, 当达到最大值时自动清零重新开始计数。

表 15-11: IMISS 寄存器

位	Field	RW	复位值	描述
---	-------	----	-----	----

位 0~位 31	MISSES	读/写	0	Cache 缺失次数。
----------	--------	-----	---	-------------

4. 软件设计

4.1. 库函数的应用(定时器)

Flash 操作需要按照下面几个步骤进行，包含规划写入数据的 Flash 空间、擦除 Flash、写 Flash 以及读 Flash。

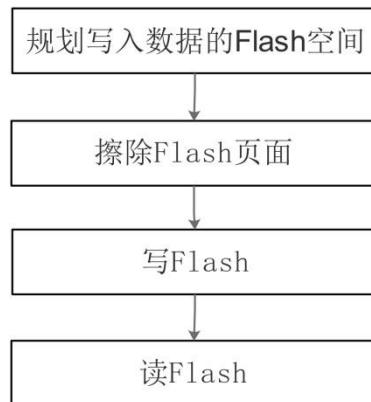


图 15-2: Flash 读写操作步骤

4.1.1. 规划写入数据的 Flash 空间

操作 Flash 之前，我们需要确定使用的 Flash 地址空间，也就是确定写入数据的起始地址和大小，并根据这些信息计算出所需占用的 Flash 页面，因为读写 Flash 的时候需要用到起始地址和大小，擦除 Flash 的时候需要页面。

4.1.2. 擦除页

Flash 页面擦除使用的库函数是 `nrf_nvmc_page_erase()`，该函数的实参必须是待擦除的 Flash 页面的起始地址，函数中将实参中的 Flash 页面起始地址写入 `ERASEPAGE` 寄存器，从而擦除该页面，函数原型如下表所示。

表 15-12: `nrf_nvmc_page_erase()` 函数

函数原型	<code>void nrf_nvmc_page_erase(uint32_t address)</code>
函数功能	页擦除函数。
参数	[in] <code>address</code> : 页起始地址。
返回值	无。

- Flash 页面擦除示例: 擦除第 127 页(第 127 页的起始地址是: $127 \times 4 \times 1024 = 0x0007F000$)。

```
nrf_nvmc_page_erase(0x0007F000);
```

4.1.3. 写

写 Flash 的最小单位是字，写 Flash 的库函数是 nrf_nvmc_write_word()，该函数将一个字写入到 Flash 中指定的地址，注意 Flash 写之前必须擦除，函数原型如下表所示。

表 15-13: nrf_nvmc_write_word()函数

函数原型	<code>void nrf_nvmc_write_word (uint32_t address, uint32_t value)</code>
函数功能	向 Flash 指定地址写入一个字。
参数	[in] <code>address</code> : 写入的地址。 [in] <code>value</code> : 写入到 Flash 的数据。
返回值	无。

- 写 Flash 示例：向 Flash 的地址 0x0007F000 写入数据 “0x12345678”
`nrf_nvmc_write_word(addr,0x12345678);`

4.1.4. 读

Flash 中的数据需要通过指针来读取。读取的过程是先定义指针变量，然后将该指针指向数据存放的地址，这样就可以获取 Flash 中该地址上的数据。

- 读 Flash 示例：从地址 0x0007F000 读取一个字。

```
1. uint32_t * pdat; //定义指针变量
2. pdat = (uint32_t *)0x0007F000; //指针指向地址 0x0007F000
3. printf("%x ", *pdat); //打印出读出的数据
```

上述代码中，因为 pdat 是指针变量，0x0007F000 是一个常数，所以要将 0x0007F000 强制转换为指针类型后赋值给 pdat。

4.2. Flash 读写实验

本实验在“实验 10-1：串口数据收发”的基础上修改。程序中检测 S1 按键的状态，当 S1 按键按下后，擦除 Flash 的第 127 页，然后向第 127 页起始地址 0x0007F000 写入数据 “0x12345678”，之后读出数据并通过串口输出。

❖ 注：本节对应的试验源码是：“实验 15-1：Flash 读写”。

4.2.1. 添加需要的文件

读写 Flash 需要加入的文件如下表所示。

表 15-14: GPIOE 需要加入的文件

文件名	SDK 中的目录	描述
nrf_nvmc.c	...\\modules\\nrfx\\hal	NVMC 驱动文件。

4.2.2. 头文件引用和路径设置

3. 需要引用的头文件

因为在“main.c”文件中使用了 NVMC 驱动程序，所以“main.c”文件需要引用下面的头文件。

```
#include "nrf_nvmc.h"
```

4. 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

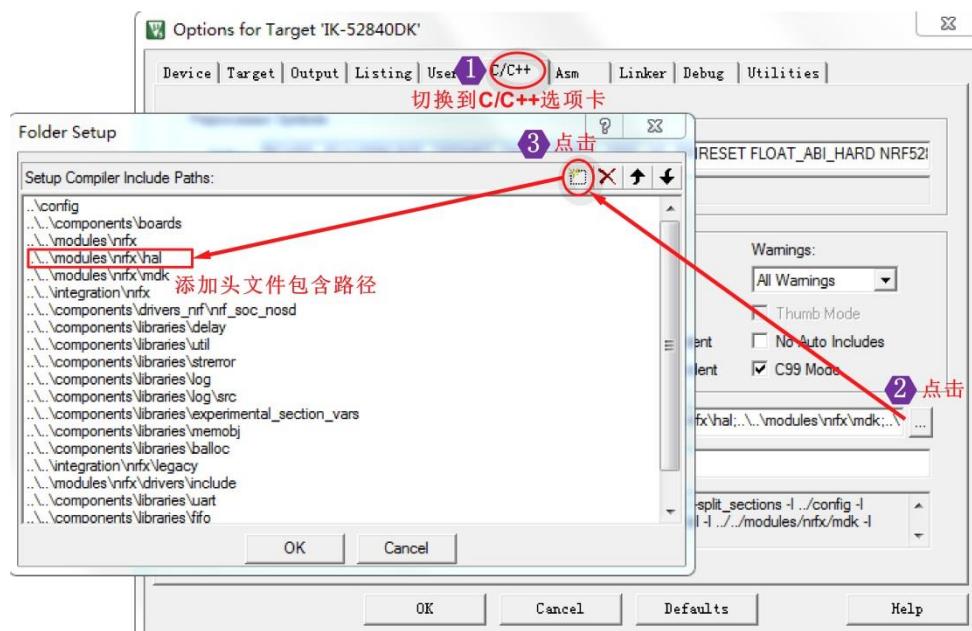


图 15-3: 添加头文件包含路径

NVMC 需要添加的头文件路径如下表：

表 15-15: 头文件包含路径

序号	路径
1	...\\modules\\nrfx\\hal

4.2.3. 代码编写

根据上文描述可知，执行 Flash 写操作时，首先要规划写入数据的 Flash 空间，本例中写入的起始地址是：0x0007F000，大小是一个字。写入之前先要擦除写入地址所在的页面，

本例中 0x0007F000 是第 127 页面的起始地址，所以擦除第 127 页。而读 Flash 数据的时候，通过指针读出数据即可，程序清单如下。

代码清单：Flash 数据读取

```
1. int main(void)
2. {
3.     uint32_t    addr;
4.     //定义读取 Flash 的指针
5.     uint32_t * pdat;
6.
7.     //初始化开发板上的 4 个 LED，即将驱动 LED 的 GPIO 配置为输出，
8.     bsp_board_init(BSP_INIT_LEDS);
9.
10.    //配置 P0.13 为输入，检测按键 S1 状态
11.    nrf_gpio_cfg_input(BUTTON_1,NRF_GPIO_PIN_PULLUP);
12.    //串口初始化
13.    uart_config();
14.
15.    while(true)
16.    {
17.        //检测按键 S1 是否按下
18.        if(nrf_gpio_pin_read(BUTTON_1) == 0)
19.        {
20.            //D3 点亮，D4 熄灭，指示：PPI 组使能
21.            nrf_gpio_pin_clear(LED_1);
22.            while(nrf_gpio_pin_read(BUTTON_1) == 0){} //等待按键释放
23.
24.            // 待擦除页的起始地址
25.            addr = 0x0007F000;
26.            // 擦除页
27.            nrf_nvmc_page_erase(addr);
28.            //向地址 0x0007F000 写入一个字 0x12345678
29.            nrf_nvmc_write_word(addr,0x12345678);
30.            // 读出数据并通过串口打印出数据
31.            pdat = (uint32_t *)addr;
32.
33.            pdat = (uint32_t *)0x0007F000;
34.            printf("0x%08x was read from flash\r\n", *pdat);
35.            nrf_gpio_pin_set(LED_1);
36.        }
37.    }
38. }
```

4.2.4. 硬件连接

本实验需要使用 P0.13 驱动 LED 指示灯 D1, P0.11 检测 S1 按键的状态, P0.06 和 P0.08 作为串口通讯引脚, 按照下图所示短接跳线帽。

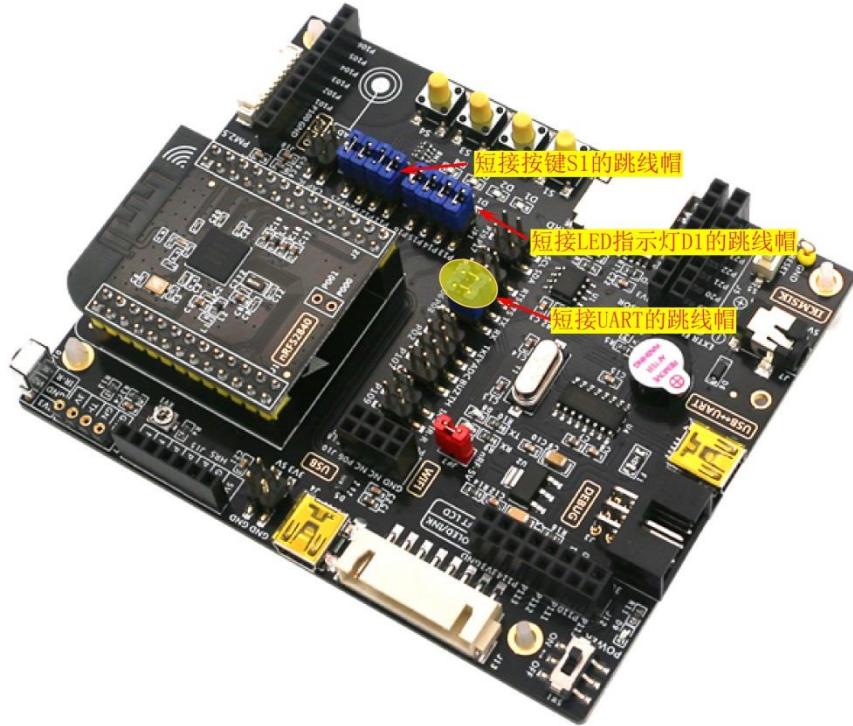


图 15-4: 开发板跳线帽短接

4.2.5. 试验步骤

1. 解压 “…\3: 开发指南（上册）配套实验源码\” 目录下的压缩文件 “实验 15-1: Flash 读写”, 将解压后得到的文件夹 “flash_rw” 拷贝到合适的目录, 如 “D\nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行 “Project→Open Project” 打开 “…\flash_rw\project\mdk5” 目录下的工程 “flash_rw.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏, 观察编译的结果, 如果有错误, 修改程序, 直到编译成功为止。编译后生成的 HEX 文件 “nrf52840_qiaa.hex” 位于工程目录下的 “Objects” 文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真, 点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后, 按下 S1 按键, D1 点亮, 指示按键 S1 已按下, 接着擦除 Flash 的第 127 页, 并向该页起始地址 0x0007F000 写入数据 “0x12345678”, 之后读出数据并通过串口输出。

第十六章：看门狗定时器

1. 学习目的

1. 学习看门狗定时器作用和原理。
2. 掌握看门狗定时器库函数的应用：操作流程及参数的配置，尤其是看门狗定时器的动作和超时时间的配置和意义。

2. 看门狗定时器原理

1. 看门狗定时器的作用

看门狗定时器（WDT: Watchdog Timer）的作用是在发生软件故障时（如程序陷入死循环或者程序跑飞），强制复位单片机，让单片机重新运行程序。

看门狗定时器本质上是一个计数器，只不过这个计数器的作用是固定的，一旦计数值递增达到设定的值（向上计数）或者计数值递减到 0（向下计数），即“超时”时，看门狗定时器产生复位信号，复位系统。

程序正常运行时，会在看门狗定时器“超时”前清零计数值（向上计数）或重装计数值（向下计数），俗称“喂狗”，这样就保证了看门狗定时器永不会“超时”，而一旦程序运行出现故障，无法正常“喂狗”时，看门狗定时器最终会“超时”复位系统。

2. nRF52840 看门狗定时器原理

nRF52840 的片内集成了一个看门狗定时器，该看门狗定时器是倒计数器（向下计数），装载计数值后，计数值在看门狗时钟频率下不断减 1，当计数值减少到 0 时产生 TIMEOUT 事件，TIMEOUT 事件会自动复位系统。

程序中可通过配置 INTENSET 和 INTENCLR 寄存器来控制 TIMEOUT 事件是否产生中断，如果使能了 TIMEOUT 事件中断，看门狗定时器会在 TIMEOUT 事件产生后推迟 2 个 32.768KHz 时钟周期后复位系统。

看门狗定时器通过触发 TASKS_START 任务启动，看门狗定时器启动时，如没有其他 32.768KHz 时钟源提供时钟，看门狗定时器会强制打开片内 32.768KHz RC 振荡器。

看门狗定时器超时周期取决于 CRV 寄存器中写入的重装计数值和看门狗定时器的时钟频率，计算公式如下：

$$\text{超时时间} = \frac{\text{CRV} + 1}{32768} \text{ 秒}$$

看门狗定时器有 8 个独立的重装请求寄存器 RR[0]~RR[7]，重装计数值时向使能的重装请求寄存器写入 0x6E524635 即可。

看门狗定时器一旦启动后，无法停止，但是，可以通过配置看门狗定时器的 CONFIG

寄存器，使其在 CPU 睡眠期间，或是仿真器将 CPU 挂起的时候自动暂停。默认情况下，看门狗定时器会在 CPU 睡眠期间，或是仿真器将 CPU 暂停的时候保持运行。

看门狗定时器的配置必须在启动前完成，一旦启动后，将不能再对 WDT 的 CRV、RREN 和 CONFIG 等寄存器进行配置，而只有在看门狗定时器复位后才可以重新配置。需要注意的是并不是所有的复位源都可以复位看门狗定时器，下表列出了哪些复位源可以复位看门狗定时器。

表 16-1：复位源和看门狗定时器复位

复位源	能否复位看门狗定时器
CPU lockup	不能
软件复位	不能
System OFF 唤醒后复位	不能
看门狗定时器复位	能
外部引脚复位	能
上电复位	能
掉电复位	能

3. WDT 寄存器

nRF52840 片内集成了一个看门狗定时器，其基址和寄存器如下表所示。

表 16-2：WDT 基址

外设名称	基址
WDT	0x40010000

表 16-3：WDT 寄存器

序号	寄存器名	偏移地址	功能描述
任务寄存器			
1	TASKS_START	0x000	启动 WDT.
事件寄存器			
1	EVENTS_TIMEOUT	0x100	WDT 超时事件。
通用寄存器			
1	INTENSET	0x304	使能中断。
2	INTENCLR	0x308	禁止中断。
3	RUNSTATUS	0x400	运行状态。
4	REQSTATUS	0x404	请求状态。

5	CRV	0x504	计数器重装值。
6	RREN	0x508	重装请求寄存器使能。
7	CONFIG	0x50C	配置寄存器。
8	RR[0]	0x600	重装请求 0。
9	RR[1]	0x604	重装请求 1。
10	RR[2]	0x608	重装请求 2。
11	RR[3]	0x60C	重装请求 3。
12	RR[4]	0x610	重装请求 4。
13	RR[5]	0x614	重装请求 5。
14	RR[6]	0x618	重装请求 6。
15	RR[7]	0x61C	重装请求 7。

■ INTENSET: 中断使能寄存器

表 16-4: INTENSET 寄存器

位	Field	RW	复位值	描述
位 0	TIMEOUT	读/写	0	写“1”使能 TIMEOUT 事件中断。 写, 1: 使能。 读, 0: TIMEOUT 事件中断已禁止。 读, 1: TIMEOUT 事件中断已使能。

■ INTENCLR: 中断禁止寄存器

表 16-5: INTENCLR 寄存器

位	Field	RW	复位值	描述
位 0	TIMEOUT	读/写	0	写“1”禁止 TIMEOUT 事件中断。 写, 1: 禁止。 读, 0: TIMEOUT 事件中断已禁止。 读, 1: TIMEOUT 事件中断已使能。

■ RUNSTATUS: WDT 运行指示寄存器

表 16-6: RUNSTATUS 寄存器

位	Field	RW	复位值	描述
位 0	RUNSTATUS	只读	0	指示 WDT 是否运行。 0: WDT 未运行。 1: WDT 已运行。

■ REQSTATUS: 请求状态寄存器

表 16-7: REQSTATUS 寄存器

位	Field	RW	复位值	描述
位 0	RR0	只读	1	RR0 寄存器请求状态 0: RR0 寄存器未使能, 或者已经请求重装。 1: RR0 寄存器已使能, 并且还未请求重装。
位 1	RR1	只读	0	RR1 寄存器请求状态 0: RR1 寄存器未使能, 或者已经请求重装。 1: RR1 寄存器已使能, 并且还未请求重装。
位 2	RR2	只读	0	RR2 寄存器请求状态 0: RR2 寄存器未使能, 或者已经请求重装。 1: RR2 寄存器已使能, 并且还未请求重装。
位 3	RR3	只读	0	RR3 寄存器请求状态 0: RR3 寄存器未使能, 或者已经请求重装。 1: RR3 寄存器已使能, 并且还未请求重装。
位 4	RR4	只读	0	RR4 寄存器请求状态 0: RR4 寄存器未使能, 或者已经请求重装。 1: RR4 寄存器已使能, 并且还未请求重装。
位 5	RR5	只读	0	RR5 寄存器请求状态 0: RR5 寄存器未使能, 或者已经请求重装。 1: RR5 寄存器已使能, 并且还未请求重装。
位 6	RR6	只读	0	RR6 寄存器请求状态 0: RR6 寄存器未使能, 或者已经请求重装。 1: RR6 寄存器已使能, 并且还未请求重装。
位 7	RR7	只读	0	RR7 寄存器请求状态 0: RR7 寄存器未使能, 或者已经请求重装。 1: RR7 寄存器已使能, 并且还未请求重装。

■ CRV：计数器重装值寄存器

表 16-8: CRV 寄存器

位	Field	RW	复位值	描述
位 31~位 0	CRV	读/写	0xFFFFFFFF	计数器重装值，范围：[0x0000000F..0xFFFFFFFF]。

■ CC[n] (n=0~5): 捕获/比较寄存器

表 16-9: CC[n] (n=0~5) 寄存器

位	Field	RW	复位值	描述
位 32~位 0	CC	读/写	0	捕获/比较值。 寄存器中的有效位数和位宽寄存器 BITMODE 配置的位数一致。例如位宽配置为 8 位，CC 寄存器只有位 0~7 有效。

4. 软件设计

4.1. 库函数的应用

使用 WDT 时需要初始化 WDT 程序模块，之后向 WDT 驱动程序申请喂狗通道，可以申请一个，也可以申请多个（WDT 共有 8 个喂狗通道），申请成功后，启动看门狗。看门狗启动后，WDT 计数值开始递减，程序需要在 WDT 计数值减到 0 之前执行喂狗操作，重装计数值。WDT 的应用流程如下图所示。

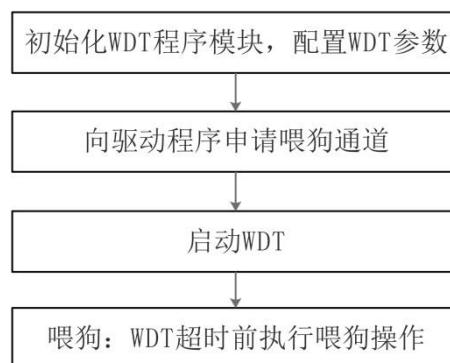


图 16-1: WDT 的应用步骤

4.1.1. 看门狗定时器初始化

WDT 初始化通过调用 nrfx_wdt_init() 函数完成，该函数中会初始化 WDT 程序模块，即 WDT 程序模块初始化标志设置为已初始化，同时该函数会配置 WDT 的动作、重装值和中断优先级。nrfx_wdt_init() 函数原型如下：

表 16-10: nrfx_wdt_init()函数

函数原型	nrfx_err_t nrfx_wdt_init (nrfx_wdt_config_t const * p_config, nrfx_wdt_event_handler_t wdt_event_handler)
函数功能	初始化看门狗定时器。
参数	[in] p_config : 指向看门狗定时器初始化配置结构体。 [in] wdt_event_handler : 事件句柄, 不能为 NULL。
返回值	初始化成功返回 NRF_SUCCESS 否则返回错误代码。

nrfx_wdt_init() 函数有 2 个输入参数, 其中 **p_config** 指向 WDT 配置结构体, **wdt_event_handler** 指向 WDT 事件回调函数。所以在调用 nrfx_wdt_init ()函数之前, 我们还需要提供 WDT 配置结构体和事件回调函数。

3. 配置结构体

驱动程序提供了一个 WDT 配置结构体 nrfx_wdt_config_t, 该结构体包含了定时器配置所用的参数, WDT 配置结构体声明如下, 其中 **reload_value** 的单位是毫秒, WDT 初始化函数 nrfx_wdt_init() 中会将 **reload_value** 计算为实际的重装值并写入到 CRV 寄存器。

代码清单: WDT 配置结构体

```

1. typedef struct
2. {
3.     //CPU 睡眠和挂起时 WDT 动作
4.     nrf_wdt_behaviour_t behaviour;
5.     //WDT 超时时间(单位 ms), 初始化函数中会将超时时间计算为重装值
6.     uint32_t            reload_value;
7.     //WDT 中断优先级
8.     uint8_t             interrupt_priority;
9. } nrfx_wdt_config_t;
```

一般地, 在定义定时器配置结构体时, 会使用宏 **NRFX_WDT_DEFAULT_CONFIG** 初始化配置结构体, 该宏定义了 WDT 的默认配置, 这些配置参数定义在“ **sdk_config.h**”文件中, 我们可以根据需要修改“ **sdk_config.h**”文件中的配置参数。

代码清单: 定时器配置结构体

```

1. #define NRFX_WDT_DEFAULT_CONFIG \
2. { \
3.     .behaviour      = (nrf_wdt_behaviour_t)NRFX_WDT_CONFIG_BEHAVIOUR, \
4.     .reload_value   = NRFX_WDT_CONFIG_RELOAD_VALUE, \
5.     .interrupt_priority = NRFX_WDT_CONFIG_IRQ_PRIORITY, \

```

6. }

其中，WDT 动作配置项 NRFX_WDT_CONFIG_BEHAVIOUR 有下表所示的 4 个可配置的值。看门狗超时时间 NRFX_WDT_CONFIG_RELOAD_VALUE 可配置的范围是（1~131072000）ms。

表 16-11: WDT 可配置的动作

值	动作
1	CPU 睡眠时运行，CPU 挂起时暂停。
8	CPU 睡眠时暂停，CPU 挂起时运行。
9	CPU 睡眠时暂停，CPU 挂起时暂停。
0	CPU 睡眠时运行，CPU 挂起时运行。

4. WDT 事件回调函数

WDT 事件回调函数的编写格式如下，需要注意的是事件回调函数中不能加入过多的功能代码，因为 WDT 中断中可花费的最长的时间只有 2 个 32.768KHz 时钟周期，之后系统就会复位。

代码清单：WDT 事件回调函数

```
1. void wdt_event_handler(void)
2. {
3.     //功能代码
4. }
```

4.1.2. 申请喂狗通道

看门狗定时器有 8 个独立的重装请求寄存器 RR[0]~RR[7]，每个重装请求寄存器都可以用来喂狗，重装请求寄存器的分配由驱动程序管理，申请喂狗通道就是向驱动程序申请一个重装请求寄存器。

应用程序是通过调用 nrfx_wdt_channel_alloc() 函数向驱动程序申请喂狗通道的，该函数按照重装请求寄存器索引顺序将重装请求寄存器分配给应用程序，函数原型如下。

表 16-12: nrfx_wdt_channel_alloc() 函数

函数原型	nrfx_err_t nrfx_wdt_channel_alloc (nrfx_wdt_channel_id * p_channel_id)
函数功能	扩展比较模式下设置 Timer 通道。
参数	[out] p_channel_id: 指向保存驱动程序分配的喂狗通道的变量。
返回值	初始化成功返回 NRF_SUCCESS 否则返回错误代码。

■ 应用示例：申请喂狗通道

```
nrfx_wdt_channel_id m_channel_id;
err_code = nrfx_wdt_channel_alloc(&m_channel_id);
```

其中，变量 `m_channel_id` 用来保存应用程序申请的喂狗通道。

4.1.3. 启动看门狗

应用程序通过调用 `nrfx_wdt_enable()` 函数启动看门狗，启动后，看门狗即开始递减计数，程序中至少要使用一个喂狗通道在看门狗定时器超时前喂狗，以避免看门狗定时器复位系统。`nrfx_wdt_enable()` 函数原型如下表所示。

表 16-13: `nrfx_wdt_enable()` 函数

函数原型	<code>void nrfx_wdt_enable (void)</code>
函数功能	启动看门狗定时器。一旦使能后，程序中至少要使用一个喂狗通道在看门狗定时器超时前喂狗，以避免看门狗定时器复位系统。
参数	[in] <code>p_channel_id</code> : 喂狗通道。
返回值	无。

4.1.4. 喂狗

喂狗通过 `nrfx_wdt_channel_feed()` 函数完成，该函数向指定的喂狗通道（重装请求寄存器）写入 0x6E524635，重装计数值。

表 16-14: `nrfx_wdt_channel_feed()` 函数

函数原型	<code>void nrfx_wdt_channel_feed (void)</code>
函数功能	喂狗。该函数在指定的喂狗通道执行喂狗操作。
参数	无。
返回值	无。

4.2. 看门狗试验

本实验在“实验 6-1：GPIO 输出驱动 led 闪烁”的基础上修改，配置 WDT 超时时间为 2000ms，即 2000ms 内不执行喂狗操作，系统复位。

为了观察到复位现象，程序启动后，D1 闪烁 4 次，指示系统启动，之后初始化并启动 WDT，每按动一次 S1 按键执行一次喂狗，如果连续在 2 秒内按动 S1 按键喂狗，系统不会复位，如果 2 秒内不按动 S1 按键喂狗，系统会复位重新启动，可以看到 D1 闪烁 4 次。

WDT 超时后，会产生 TIMEOUT 事件中断，中断中可花费的最长的时间是 2 个 32.768KHz 时钟周期，时间极短，为了观察到 TIMEOUT 事件中断，在启动 WDT 之前熄灭

4个LED, TIMEOUT事件函数中点亮4个LED,这样,WDT超时后,我们会看到4个LED微弱地闪烁一次,指示产生了TIMEOUT事件中断。

❖ 注:本节对应的试验源码是:“实验16-1: WDT”。

4.2.1. 添加需要的文件

定时器需要加入的文件如下表所示。

表 16-15: WDT 需要加入的文件

文件名	SDK 中的目录	描述
nrfx_wdt.c	..\\..\\modules\\nrfx\\drivers\\src	定时器驱动程序。
nrf_drv_clock.c	..\\..\\integration\\nrfx\\legacy	时钟配置驱动(旧版本)。
nrfx_clock.c	..\\..\\modules\\nrfx\\drivers\\src	时钟配置驱动(新版本)。

4.2.2. 头文件引用和路径设置

1. 需要引用的头文件

因为在“main.c”文件中使用了定时器,所以需要引用下面的头文件。

```
#include "nrfx_wdt.h"
#include "nrf_drv_clock.h"
```

2. 需要添加的头文件包含路径

MDK 中点击魔术棒, 打开工程配置窗口, 按照下图所示添加头文件包含路径。

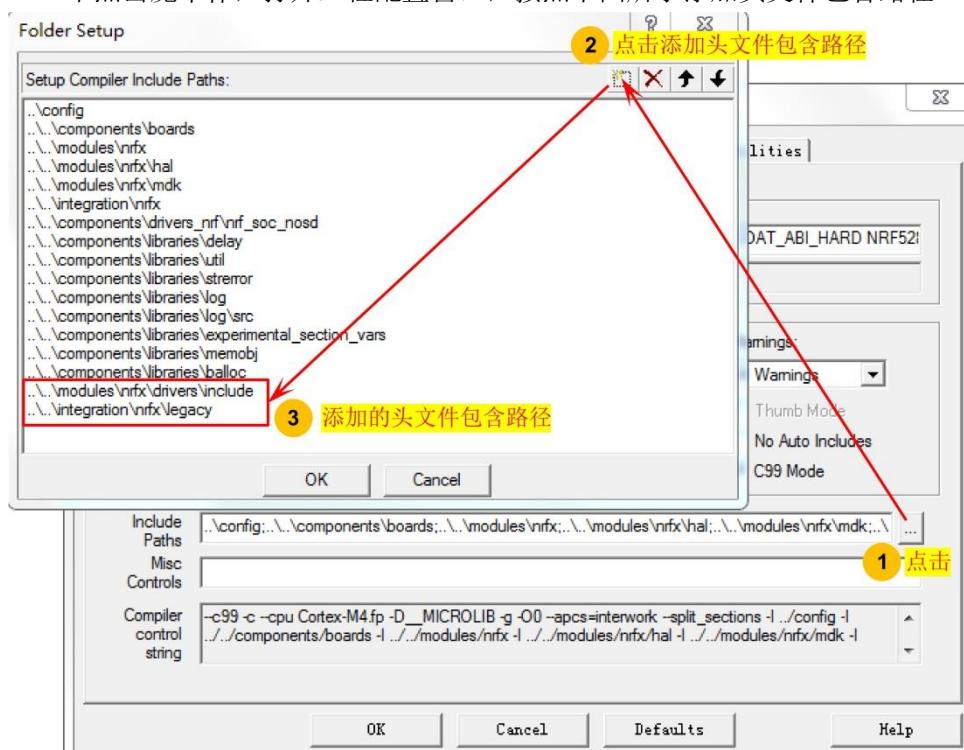


图 16-2: 添加头文件包含路径

使用定时器需要添加的头文件路径如下表:

表 16-16: 头文件包含路径

序号	路径
1	..\..\integration\nrfx\legacy
2	..\..\modules\nrfx\drivers\include

4.2.3. 工程配置

打开“sdk_config.h”文件，加入看门狗需要的配置，如下图所示，编辑内容时切换到“Text Editor”进行编辑（编辑的内容参考实验的源码，因此代码很长，此处不贴出代码，参见“sdk_config.h”文件的（52~320）行），编辑完成后，切换到“Configuration Wizard”，即可观察到配置的具体项目：

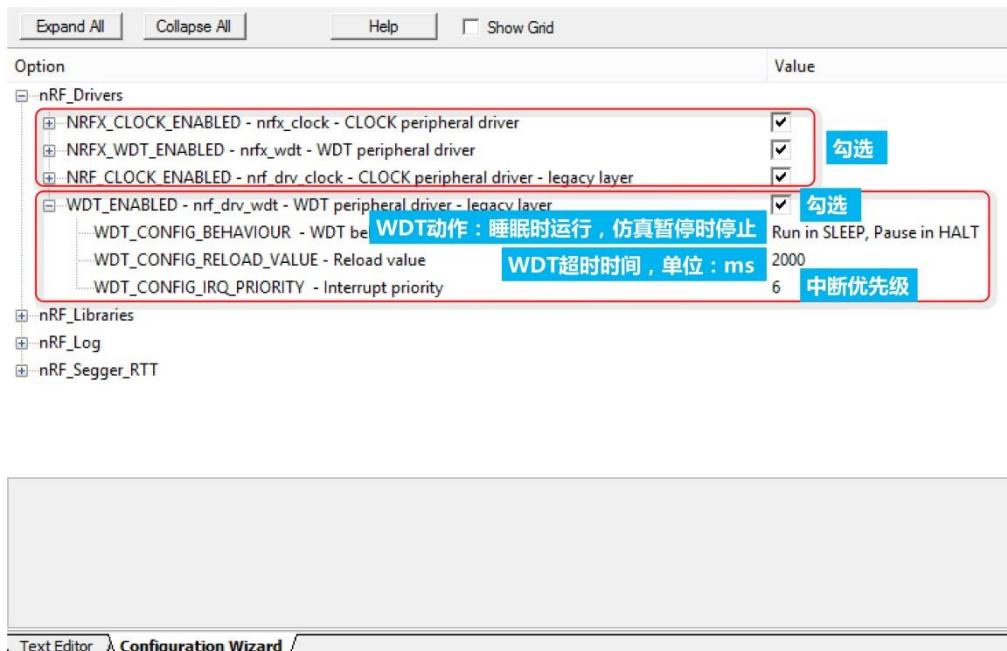


图 13-8: 工程配置

4.2.4. 代码编写

首先定义变量 m_channel_id，用于保存应用程序向驱动程序申请的喂狗通道。应用程序可以申请多个喂狗通道，每个喂狗通道都需要定义一个 nrfx_wdt_channel_id 类型变量来保存。这里，我们申请了一个喂狗通道用于执行喂狗。

代码清单：定义变量，保存申请的喂狗通道

1. //保存申请的喂狗通道
2. nrfx_wdt_channel_id m_channel_id;

接下来的是完成 WDT 初始化的配置，配置参数 WDT 动作、超时时间和中断优先级可根据需求在配置向导“sdk_config.h”里修改，初始化完成后，启动 WDT。

代码清单：初始化并启动 WDT

```
1. //看门狗初始化， 初始化完成后会启动看门狗，看门狗一旦启动后就无法停止
2. void wdt_init(void)
3. {
4.     uint32_t err_code = NRF_SUCCESS;
5.
6.     //定义 WDT 配置结构体并使用
7.     nrfx_wdt_config_t config = NRFX_WDT_DEFAULT_CONFIG;
8.     //初始化 WDT
9.     err_code = nrfx_wdt_init(&config, wdt_event_handler);
10.    APP_ERROR_CHECK(err_code);
11.    //申请喂狗通道，也就是使用哪个
12.    err_code = nrfx_wdt_channel_alloc(&m_channel_id);
13.    APP_ERROR_CHECK(err_code);
14.    //启动 WDT
15.    nrfx_wdt_enable();
16. }
```

WDT 初始化时需要注册 WDT 事件回调函数，所以应用程序需要提供该回调函数。

代码清单：WDT 事件回调函数

```
1. //WDT 中断中可花费的最长的时间是 2 个 32.768KHz 时钟周期，之后系统复位
2. void wdt_event_handler(void)
3. {
4.     //点亮 4 个 LED。因为最长只有 2 个 32.768KHz 时钟周期的时间，所以程序运行后我们会
5.     //看到 4 个 LED 微弱地闪烁一次，也就是 4 个 LED 刚点亮，系统就复位了，复位后，4 个
6.     //LED 熄灭
7.     bsp_board_leds_on();
8. }
9.
```

最后，在主函数中加入系统启动指示，初始化并启动 WDT，之后循环查询按键 S1 状态，检测到按键 S1 按下后，执行喂狗操作。

代码清单：主函数

```
1. int main(void)
2. {
3.     uint8_t i;
4.     uint32_t err_code = NRF_SUCCESS;
5.
6.     //配置 P0.13 为输入，检测按键 S1 状态
7.     nrf_gpio_cfg_input(BUTTON_1, NRF_GPIO_PIN_PULLUP);
```

```
8.  
9.     //配置 32.768KHz 低频时钟，使用外部 32.768KHz 晶体。如果不配置低频时钟的话，  
10.    //WDT 会强制打开内部低频 RC 时钟  
11.    err_code = nrf_drv_clock_init();  
12.    APP_ERROR_CHECK(err_code);  
13.    nrf_drv_clock_lfcclk_request(NULL);  
14.  
15.    //初始化开发板上的 4 个 LED，即将驱动 LED 的 GPIO 配置为输出，  
16.    bsp_board_init(BSP_INIT_LEDS);  
17.  
18.    //闪烁指示灯 D1 表示系统启动  
19.    for(i=0;i<8;i++)  
20.    {  
21.        bsp_board_led_invert(0);  
22.        nrf_delay_ms(150);  
23.    }  
24.    //熄灭 4 个 LED 指示灯，当 WDT 超时后，产生 TIMEOUT 事件中断，在中断事件回调函数  
25.    //里面点亮 4 个 LED，这样，当 WDT 超时后会观察到 4 个 LED 微地闪烁一次  
26.    bsp_board_leds_off();  
27.    //初始化并启动 WDT  
28.    wdt_init();  
29.  
30.    while(true)  
31.    {  
32.        //查询按键 S1 是否按下，检查到 S1 按下后，执行一次喂狗操作  
33.        if(nrf_gpio_pin_read(BUTTON_1) == 0)  
34.        {  
35.            //D1 点亮指示按键 S1 按下  
36.            nrf_gpio_pin_clear(LED_2);  
37.            //喂狗  
38.            nrfx_wdt_channel_feed(m_channel_id);  
39.            //等待按键释放  
40.            while(nrf_gpio_pin_read(BUTTON_1) == 0){}  
41.            nrf_gpio_pin_set(LED_2);  
42.        }  
43.    }  
44. }  
45.
```

4.2.5. 硬件连接

本实验需要使用 P0.13 P0.14 P0.15 P0.16 驱动 4 个 LED 指示灯，P0.11 检测 S1 按键的状态，需要用跳线帽短接这些引脚，如下图所示。



图 16-4：开发板跳线帽短接

4.2.6. 试验步骤

1. 解压“…\3: 开发指南（上册）配套试验源码”目录下的压缩文件“实验 16-1: WDT”，将解压后得到的文件夹“wdt”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\wdt\project\mdk5”目录下的工程“wdt.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，可以观察到 D1 指示灯闪烁 4 次，之后熄灭。连续 2 秒内按动 S1 按键，系统不会复位。2 秒内不按动 S1 按键，WDT 超时，系统复位重新启动，D1 闪烁 4 次。在 WDT 超时复位系统的过程中，可以看到 4 个 LED 微弱闪烁一次，指示了 WDT 超时后产生了 TIMEOUT 事件中断。

- **思考题 1：**如何将 WDT 超时时间配置为 5 秒？仿真程序的时候，如果暂停程序运行，WDT 会继续运行吗？为什么？
- **思考题 2：**看门狗启动后可以通过软件停止吗？
- **思考题 3：**什么情况下可以对看门狗重新配置？

第十七章：模数转换 SAADC

1. 学习目的

1. 掌握单端采样和差分采样的使用。
2. 掌握通过 SAADC 配置的参数和采样数据计算实际电压值的方法。
3. 掌握 SAADC 采样时双缓存的原理和应用。

2. SAADC 原理

实际应用中，我们经常需要将模拟量转换为数字量供 CPU 处理，如电池电压检测、温度检测等等，对于 CPU 来说，它能处理的是数字量，所以，需要通过 A/D 转换(模数转换)将时间连续、幅值也连续的模拟量转换为时间离散、幅值也离散的数字信号，从而实现 CPU 对模拟信号的处理，能够实现 A/D 转换功能的电路称之为模数转换器(ADC: Analog-to-digital converter)。

ADC 的结构和实现原理有多种方式，常见的 ADC 的类型有积分型、逐次逼近型、并行比较型/串并行型、 $\Sigma - \Delta$ 调制型等。nRF52840 集成的是逐次逼近型 ADC，称为 SAADC (Successive approximation analog-to-digital converter)。SAADC 是利用二分法逐步比较，在有效精度范围内找到最接近输入模拟信号的数字量。由此可见，这种结构的 ADC 要完成一次转换，至少要比较 N 次，所以其转换速度较慢，同时电路结构也比较简单，功耗较低，适用于便携式、穿戴式等低功耗应用领域。

2.1. 主要特征

nRF52840 的 SAADC 主要特征如下：

- 1) 8/10/12 位分辨率，使用过采样可达到 14 位分辨率。
- 2) 共 8 个输入通道。
 - 单端输入时使用 1 个通道，2 个通道可组成差分输入。
 - 单端和差分输入时均可配置为扫描模式。
- 3) 满量程输入范围(0 to VDD)。
- 4) 每个通道均可单独设置参考电压。
 - VDD。
 - 内部参考。
- 5) 可以通过软件触发采样任务启动采样，也可以使用低功耗的 32.768kHz RTC 或更加精确 1/16MHz 定时器通过 PPI 触发采样任务，从而使能 SAADC 具备非常灵活的采样频率。
- 6) SAADC 支持单次模式和扫描模式：
 - 单次模式一次采样一个通道。
 - 扫描模式按照顺序采样一系列通道。通道之间的采样延迟是 $t_{ack} + t_{conv}$ ，各个通道

之间的采样延迟可能不一样，因为 t_{ack} 是软件可配置的。

- 7) 通过 EasyDMA 可以直接将采样结果保存到 RAM。
- 8) 无需外部定时器即可实现连续采样。
- 9) 可配置通道输入负载电阻。
- 10) 具备采样值门限监测功能。

nRF52840 的数字引脚是可以自由映射的，但是模拟输入的引脚是固定的，如下表所示，ADC 和 COMP、LPCOMP 等其它外设共用模拟输入 AIN0-AIN7，但是这些外设功能需要分配到不同的引脚，不建议为两个外设选择相同的模拟输入引脚。

表 17-1：模拟输入通道

模拟输入通道	GPIO	引脚编号
AIN0	P0.02	4
AIN1	P0.03	5
AIN2	P0.04	6
AIN3	P0.05	7
AIN4	P0.28	40
AIN5	P0.29	41
AIN6	P0.30	42
AIN7	P0.31	43

nRF52840 的 ADC 支持多达 8 个外部模拟输入通道，可以工作于软件控制的单次模式，也可以工作于采样速率可配置的扫描模式。模拟输入通道可以配置为 8 个单端输入或 4 组差分输入或者单端、差分混合输入(一部分模拟输入通道用作单端输入，一部分用作差分输入)。每一个 ADC 采样通道都可以配置为 AIN0 ~ AIN7 或 VDD，所有通道都可以进行单次或者连续采样，多个通道可以使用扫描模式按照通道顺序采样，另外，ADC 通道也可以过采样以提高噪声性能。

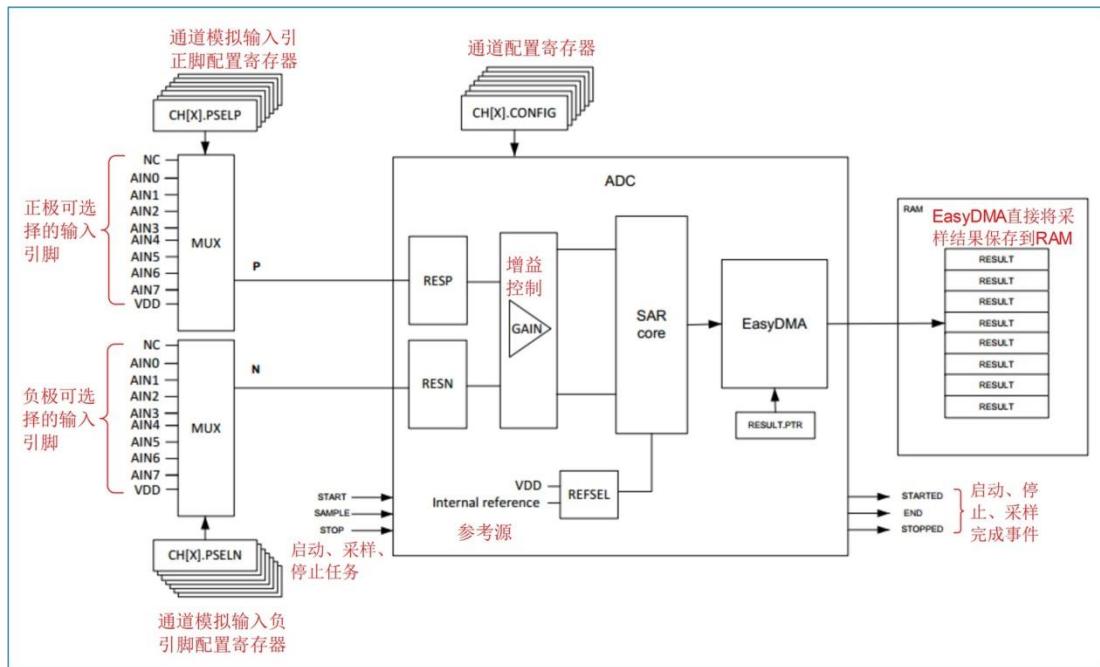


图 17-1：SAADC 原理框图

从上面的原理框图可以看到，芯片内部，nRF52840 的 ADC 本质上是一个差分模数转换器，包含正极输入 RESP 和负极输入 RESN，他们的引脚都可以从 AIN0~AIN7 中选择。默认情况下，ADC 的配置的是单端输入(即 CH[n].CONFIG 寄存器中的 MODE = 0)，此时，芯片内部将 ADC 负极输入短接到地。

使用单端模式时，是将内部地和外部待测电压的参考地假设为一样来考虑的，但是地弹噪声会导致 ADC 产生误差，如果这个误差超过我们能接受的范围，建议使用差分输入。在参考源上，相对于 nRF51822，nRF52840 取消了外部参考源，只能使用内部参考源。

使用 ADC 时，通过 CH[n].PSEL 和 CH[n].PSELN 配置通道的模拟输入引脚，通过 CH[n].CONFIG 配置 ADC 的参数，包括：模拟输入端口负载电阻、增益、参考电压、采样时间、单端或差分输入模式等。

ADC 输出的采样值取决于 CH[n].CONFIG 和 RESOLUTION 寄存器配置的参数，采样值计算公式如下：

$$\text{采样值} = [V(P) - V(N)] \times \frac{\text{GAIN}}{\text{REFERENCE}} \times 2^{(\text{RESOLUTION} - m)} \quad \text{式 17-1}$$

其中：

- **V(P)**: ADC 通道正极。
- **V(N)**: ADC 通道负极。
- **GAIN**: CH[n].CONFIG 寄存器中设置的增益 GAIN。
- **REFERENCE**: 参考电压。
- **m**: 如果 ADC 配置为单端模式，m = 0，如果 ADC 配置为差分模式，m = 1。

2.2. 工作模式

nRF52840 的 ADC 有 3 种工作模式：单次模式、连续模式和扫描模式。各个模式的具体描述如下：

1. 单次模式(one-shot mode)

nRF52840 没有专门用来配置 ADC 为单次模式的寄存器，当我们通过配置 CH[n].PSELP、CH[n].PSELN 和 CH[n].CONFIG 只使能一个 ADC 通道的时候，ADC 工作于单次模式。触发采样任务后，ADC 开始采样输入电压，采样时间通过 CH[n].CONFIG.TACQ 配置。EVENTS_DONE 事件表明了一次采样的完成。单次采样模式下，在没有过采样发生的情况下 EVENTS_RESULTDONE 事件等同于 EVENTS_DONE 事件，注意在实际采样数据通过 EasyDMA 被保存到 RAM 之前，这两个事件都会产生。

2. 连续模式(continuous mode)

可以通过下面两种方式实现连续模式：

- 使用 ADC 内部定时器实现定时采样，ADC 有一个 SAMPLERATE 寄存器，该寄存器可以配置为 Timer，配置 SAMPLERATE 的比较值 SAMPLERATE.CC 即可实现定时采样。这种方式下，触发一次采样任务即可对所有使能的通道进行采样。
- 使用 nRF52840 的通用定时器定时通过 PPI 触发采样，实现连续采样。这种方式不能算是 ADC 本身的连续采样功能，因为它是借助了其他外设实现的连续采样。

采样速率由 SAMPLERATE.CC 控制，定时时间长，采样速率慢，定时时间短，采样速率快，需要注意的是，使用连续采样模式的时候，采样速率应符合下面的公式：

$$f_{SAMPLE} < \frac{1}{[t_{ACQ} + t_{CONV}]}$$

使用 SAMPLERATE 的 Timer 实现的连续采样不能和扫描模式结合使用，连续采样模式下只能使用一个通道。

EVENTS_DONE 事件表明了一次采样的完成，连续采样模式下，在没有过采样发生的情况下 EVENTS_RESULTDONE 事件等同于 EVENTS_DONE 事件，注意在实际采样数据通过 EasyDMA 被保存到 RAM 之前，这两个事件都会产生。

3. 扫描模式(scan mode)

nRF52840 同样没有专门用来配置 ADC 为扫描模式的寄存器，那么，扫描模式是怎么进入的？

当我们使能一个 ADC 通道，ADC 工作于单次模式，当使能的通道数量大于 1 个，ADC 进入扫描模式。扫描模式下，采样所有通道花费的总时间如下式：

$$\text{总时间} < \text{Sum}(\text{CH}[x].t_{ACQ} + t_{conv}), x = 0.. \text{使能的通道数量}$$

EVENTS_DONE 事件表明了一次采样的完成，扫描模式下，在没有过采样发生的情况下

下 EVENTS_RESULTDONE 事件等同于 EVENTS_DONE 事件，注意在实际采样数据通过 EasyDMA 被保存到 RAM 之前，这两个事件都会产生。

3. SAADC 寄存器

SAADC 的基址和寄存器如下表所示。

表 17-2: SAADC 基址

外设名称	基址
SAADC	0x40007000

表 17-3: SAADC 相关寄存器

序号	寄存器名	偏移地址	功能描述
任务寄存器			
1	TASKS_START	0x000	启动 ADC 并在 RAM 中准备采样结果存放缓存区。
2	TASKS_SAMPLE	0x004	进行一次采样。如果启用扫描模式，所有通道都将被采样。
3	TASKS_STOP	0x008	停止 ADC 并终止任何正在进行的转换。
4	TASKS_CALIBRATE	0x00C	启动偏移自动校准。
事件寄存器			
1	EVENTS_STARTED	0x100	ADC 已启动。
2	EVENTS_END	0x104	ADC 已填满采样缓存区
3	EVENTS_DONE	0x108	一次转换任务完成，取决于 ADC 模式，将数据保存到 RAM 可能需要进行多次转换。
4	EVENTS_RESULTDONE	0x10C	采样结果已准备好转移到 RAM。
5	EVENTS_CALIBRATEDONE	0x110	校准完成。
6	EVENTS_STOPPED	0x114	ADC 已停止事件。
7	EVENTS_CH[0].LIMITH	0x118	采样结果大于等于 CH[0].LIMIT.HIGH。
8	EVENTS_CH[0].LIMITL	0x11C	采样结果小于等于 CH[0].LIMIT.LOW。
9	EVENTS_CH[1].LIMITH	0x120	采样结果大于等于 CH[1].LIMIT.HIGH。
10	EVENTS_CH[1].LIMITL	0x124	采样结果小于等于 CH[1].LIMIT.LOW。
11	EVENTS_CH[2].LIMITH	0x128	采样结果大于等于 CH[2].LIMIT.HIGH。
12	EVENTS_CH[2].LIMITL	0x12C	采样结果小于等于 CH[2].LIMIT.LOW。

13	EVENTS_CH[3].LIMITH	0x130	采样结果大于等于 CH[3].LIMIT.HIGH。
14	EVENTS_CH[3].LIMITL	0x134	采样结果小于等于 CH[3].LIMIT.LOW。
15	EVENTS_CH[4].LIMITH	0x138	采样结果大于等于 CH[4].LIMIT.HIGH。
16	EVENTS_CH[4].LIMITL	0x13C	采样结果小于等于 CH[4].LIMIT.LOW。
17	EVENTS_CH[5].LIMITH	0x140	采样结果大于等于 CH[5].LIMIT.HIGH。
18	EVENTS_CH[5].LIMITL	0x144	采样结果小于等于 CH[5].LIMIT.LOW。
19	EVENTS_CH[6].LIMITH	0x148	采样结果大于等于 CH[6].LIMIT.HIGH。
20	EVENTS_CH[6].LIMITL	0x14C	采样结果小于等于 CH[6].LIMIT.LOW。
21	EVENTS_CH[7].LIMITH	0x150	采样结果大于等于 CH[7].LIMIT.HIGH。
22	EVENTS_CH[7].LIMITL	0x15C	采样结果小于等于 CH[7].LIMIT.LOW。
通用寄存器			
1	INTEN	0x300	使能或禁止中断
2	INTENSET	0x304	使能中断
3	INTENCLR	0x308	禁止中断
4	STATUS	0x400	ADC 状态寄存器
5	ENABLE	0x500	使能或禁用 ADC。
6	CH[0].PSELP	0x510	通道 CH[0]正极输入引脚选择。
7	CH[0].PSELN	0x514	通道 CH[0]负极输入引脚选择。
8	CH[0].CONFIG	0x518	通道 CH[0]输入配置。
9	CH[0].LIMIT	0x51C	通道 CH[0]事件监视高/低门限值。
10	CH[0].PSELP	0x510	通道 CH[0]正极输入引脚选择。
11	CH[0].PSELN	0x514	通道 CH[0]负极输入引脚选择。
12	CH[1].CONFIG	0x520	通道 CH[1]输入配置。
13	CH[1].LIMIT	0x524	通道 CH[1]事件监视高/低门限值。
14	CH[1].PSELP	0x528	通道 CH[1]正极输入引脚选择。
15	CH[1].PSELN	0x52C	通道 CH[1]负极输入引脚选择。
16	CH[2].CONFIG	0x530	通道 CH[2]输入配置。
17	CH[2].LIMIT	0x534	通道 CH[2]事件监视高/低门限值。

18	CH[2].PSELP	0x538	通道 CH[2]正极输入引脚选择。
19	CH[2].PSELN	0x53C	通道 CH[2]负极输入引脚选择。
20	CH[3].CONFIG	0x540	通道 CH[3]输入配置。
21	CH[3].LIMIT	0x544	通道 CH[3]事件监视高/低门限值。
22	CH[3].PSELP	0x548	通道 CH[3]正极输入引脚选择。
23	CH[3].PSELN	0x54C	通道 CH[3]负极输入引脚选择。
24	CH[4].CONFIG	0x550	通道 CH[4]输入配置。
25	CH[4].LIMIT	0x554	通道 CH[4]事件监视高/低门限值。
26	CH[4].PSELP	0x558	通道 CH[4]正极输入引脚选择。
27	CH[4].PSELN	0x55C	通道 CH[4]负极输入引脚选择。
28	CH[5].CONFIG	0x560	通道 CH[5]输入配置。
29	CH[5].LIMIT	0x564	通道 CH[5]事件监视高/低门限值。
30	CH[5].PSELP	0x568	通道 CH[5]正极输入引脚选择。
31	CH[5].PSELN	0x56C	通道 CH[5]负极输入引脚选择。
32	CH[6].CONFIG	0x570	通道 CH[6]输入配置。
33	CH[6].LIMIT	0x574	通道 CH[6]事件监视高/低门限值。
34	CH[6].PSELP	0x578	通道 CH[6]正极输入引脚选择。
35	CH[6].PSELN	0x57C	通道 CH[6]负极输入引脚选择。
36	CH[7].CONFIG	0x580	通道 CH[7]输入配置。
37	CH[7].LIMIT	0x584	通道 CH[7]事件监视高/低门限值。
38	CH[7].PSELP	0x588	通道 CH[7]正极输入引脚选择。
39	CH[7].PSELN	0x58C	通道 CH[7]负极输入引脚选择。
40	RESOLUTION	0x5F0	ADC 分辨率配置
41	OVERSAMPLE	0x5F4	过采样配置，注意 OVERSAMPLE 不能和扫描模式结合使用
42	SAMPLERATE	0x5F8	采样速率配置，控制正常或连续模式的采样速率。
43	RESULT.PTR	0x62C	数据指针。
44	RESULT.MAXCNT	0x630	传输的最大缓存字数。

45	RESULT.AMOUNT	0x634	自上次启动后传输的缓存字数。
----	---------------	-------	----------------

■ INTEN：中断使能/禁止寄存器

INTEN 寄存器用于使能或禁止中断。位的值写 0 时禁止对应的中断，位的值写 1 时使能对应的中断。注意 INTEN 寄存器既可以使能也可以禁止中断。

表 17-4: INTEN 寄存器

位	Field	RW	复位值	描述
位 0	STARTED	读/写	0	使能/禁止 STARTED 事件中断。 0: 禁止。 1: 使能。
位 1	END	读/写	0	使能/禁止 END 事件中断。 0: 禁止。 1: 使能。
位 2	DONE	读/写	0	使能/禁止 DONE 事件中断。 0: 禁止。 1: 使能。
位 3	RESULTDONE	读/写	0	使能/禁止 RESULTDONE 事件中断。 0: 禁止。 1: 使能。
位 4	CALIBRATEDONE	读/写	0	使能/禁止 CALIBRATEDONE 事件中断。 0: 禁止。 1: 使能。
位 5	STOPPED	读/写	0	使能/禁止 STOPPED 事件中断。 0: 禁止。 1: 使能。
位 6、8、 10、12、 14、16、 18、20	CHnLIMITH	读/写	0	使能/禁止 CH[0].LIMITH(n=0~7)事件中 断。 0: 禁止。 1: 使能。
位 7、9、 11、13、 15、17、 19、21	CHnLIMITL	读/写	0	使能/禁止 CH[0].LIMITL(n=0~7)事件中 断。 0: 禁止。 1: 使能。

■ INTENSET：中断使能寄存器

INTENSET 寄存器用于使能中断。位的值写为 1 时使能对应的中断，写入 0 无效。注意 INTENSET 寄存器只能用来使能中断。

表 17-5: INTENSET 寄存器

位	Field	RW	复位值	描述
位 0	STARTED	读/写	0	写“1”使能 STARTED 事件中断，写“0”无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。
位 1	END	读/写	0	写“1”使能 STARTED 事件中断，写“0”无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。
位 2	DONE	读/写	0	写“1”使能 DONE 事件中断，写“0”无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。
位 3	RESULTDONE	读/写	0	写“1”使能 RESULTDONE 事件中断，写“0”无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。
位 4	CALIBRATEDONE	读/写	0	写“1”使能 CALIBRATEDONE 事件中断，写“0”无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。
位 5	STOPPED	读/写	0	写“1”使能 STOPPED 事件中断，写“0”无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。

位 6、8、 10、12、 14、16、 18、20	CHnLIMITH	读/写 0	写“1”使能 CHnLIMITH 事件中断，写“0”无效。 写，1：使能。 读，0：已禁止。 读，1：已使能。
位 7、9、 11、13、 15、17、 19、21	CHnLIMITL	读/写 0	写“1”使能 CHnLIMITL 事件中断，写“0”无效。 写，1：使能。 读，0：已禁止。 读，1：已使能。

■ INTENCLR：中断禁止寄存器

INTENCLR 寄存器用于禁止中断。位的值写为 1 时禁止对应的中断，写入 0 无效。注意 INTENCLR 寄存器只能用来禁止中断。

表 17-6: INTENCLR 寄存器

位	Field	RW	复位值	描述
位 0	STARTED	读/写 0	写“1”禁止 STARTED 事件中断，写“0”无效。 写，1：禁止。 读，0：已禁止。 读，1：已使能。	
位 1	END	读/写 0	写“1”禁止 STARTED 事件中断，写“0”无效。 写，1：禁止。 读，0：已禁止。 读，1：已使能。	
位 2	DONE	读/写 0	写“1”禁止 DONE 事件中断，写“0”无效。 写，1：禁止。 读，0：已禁止。 读，1：已使能。	
位 3	RESULTDONE	读/写 0	写“1”禁止 RESULTDONE 事件中断，写“0”无效。 写，1：禁止。	

			读, 0: 已禁止。 读, 1: 已使能。
位 4	CALIBRATEDONE	读/写 0	写“1”禁止 CALIBRATEDONE 事件中断, 写“0”无效。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。
位 5	STOPPED	读/写 0	写“1”禁止 STOPPED 事件中断, 写“0”无效。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。
位 6、8、 10、12、 14、16、 18、20	CHnLIMITH	读/写 0	写“1”禁止 CHnLIMITH 事件中断, 写“0”无效。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。
位 7、9、 11、13、 15、17、 19、21	CHnLIMITL	读/写 0	写“1”禁止 CHnLIMITL 事件中断, 写“0”无效。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。

■ STATUS: ADC 状态寄存器

STATUS 寄存器是一个只读寄存器, 用来指示 ADC 是否处于“忙”状态。

表 17-7: STATUS 寄存器

位	Field	RW	复位值	描述
位 0	STATUS	只读	0	0: ADC 就绪, 当前没有正在进行的 ADC 转换。 1: ADC 忙, 正在进行 ADC 转换。

■ ENABLE: ADC 使能/禁止寄存器

ENABLE 寄存器是用于使能或禁用 ADC。当 ADC 使能后, ADC 获取由 CH[n].PSELP 和 CH[n].PSELN 寄存器中描述的模拟输入引脚的访问权。

表 17-8: ENABLE 寄存器

位	Field	RW	复位值	描述
位 0	ENABLE	读/写	0	0: 禁止 ADC。 1: 使能 ADC。

■ CH[n].PSELP (n=0~7): 通道正极输入配置寄存器

CH[n] (n=0~7)寄存器用于配置 SAADC 通道正极输入连接的模拟输入通道。

表 17-9: CH[n].PSELP 寄存器

位	Field	RW	复位值	描述
位 0	PSELP	读/写	0	ADC 模拟输入正极通道。 0: NC: 不连接。 1: AIN0: 模拟输入通道 0。 2: AIN1: 模拟输入通道 1。 3: AIN2: 模拟输入通道 2。 4: AIN3: 模拟输入通道 3。 5: AIN4: 模拟输入通道 4。 6: AIN5: 模拟输入通道 5。 7: AIN6: 模拟输入通道 6。 8: AIN7: 模拟输入通道 7。 9: VDD。

■ CH[n].PSELN (n=0~7): 通道负极输入配置寄存器

通道 CH[n] (n=0~7) 寄存器用于配置 SAADC 通道负极输入连接的模拟输入通道。

表 17-10: CH[n].PSELN 寄存器

位	Field	RW	复位值	描述
位 0	PSELN	读/写	0	ADC 模拟输入负极通道。 0: NC: 不连接。 1: AIN0: 模拟输入通道 0。 2: AIN1: 模拟输入通道 1。 3: AIN2: 模拟输入通道 2。 4: AIN3: 模拟输入通道 3。 5: AIN4: 模拟输入通道 4。 6: AIN5: 模拟输入通道 5。 7: AIN6: 模拟输入通道 6。 8: AIN7: 模拟输入通道 7。

	9: VDD。
--	---------

■ CH[n]. CONFIG (n=0~7): 通道配置寄存器

CH[n] . CONFIG (n=0~7) 寄存器用于 SAADC 通道的参数。

表 17-11: CH[n]. CONFIG 寄存器

位	Field	RW	复位值	描述
位 1~位 0	RESP	读/写	0	通道正极电阻配置。 0: 旁路电阻梯 1: 下拉到 GND 2: 上拉到 VDD 3: 设置输入为: VDD/2。
位 1~位 0	RESN	读/写	0	通道负极电阻配置。 0: 旁路电阻梯 1: 下拉到 GND 2: 上拉到 VDD 3: 设置输入为: VDD/2。
位 10~位 8	GAIN	读/写	0	增益控制。 0: 1/6。 1: 1/5。 2: 1/4。 3: 1/3。 4: 1/2。 5: 1。 6: 2。 7: 4。
位 12	REFSEL	读/写	0	参考源选择。 0: 内部参考源(0.6V) 1: VDD/4
位 18~位 16	TACQ	读/写	2	采样时间, ADC 用来采样输入电压的时间。 0: 3us。 0: 5us。 0: 10us。 0: 15us。

			0: 20us。 0: 40us。
位 20	MODE	读/写 0	单端/差分模式配置 0: 单端模式, PSELN 将被忽略, ADC 负极输入在内部短接到 GND。 1: 差分模式。
位 24	BURST	读/写 0	使能突发模式。 0: 禁止突发模式, 使用普通模式。 1: 使能突发模式, SAADC 以其最快速度进行 $2^{\text{OVERSAMPLE}}$ 次采样, 并将采样结果的平均值传送到数据 RAM。

■ CH[n]. LIMIT (n=0~7): 通道采样值上/下限监测配置寄存器

CH[n] . LIMIT (n=0~7) 寄存器用于配置通道采样值上/下限监测, 当采样值超过设置的上/下限时产生事件。

表 17-12: CH[n]. LIMIT 寄存器

位	Field	RW	复位值	描述
位 15~位 0	LOW	读/写	0x8000	[-32768~+32767]: 低门限。
位 31~位 16	HIGH	读/写	0x7FFF	[-32768~+32767]: 高门限

■ RESOLUTION: 分辨率配置寄存器

RESOLUTION 用于配置 SAADC 的分辨率, 配置的值对所有通道有效。

表 17-13: RESOLUTION 寄存器

位	Field	RW	复位值	描述
位 2~0	RESOLUTION	读/写	0	配置分辨率 0: 8 位。 1: 10 位。 2: 12 位。 3: 14 位。。

■ OVERSAMPLE: 过采样配置寄存器

OVERSAMPLE 寄存器用于配置过采样, 过采样不能和扫描模式结合使用。

表 17-14: OVERSAMPLE 寄存器

位	Field	RW	复位值	描述
位 3~0	OVERSAMPLE	读/写	0	过采样控制。 0: 旁路过采样。 1: 过采样 2x。 2: 过采样 4x。 3: 过采样 8x。 4: 过采样 16x。 5: 过采样 32x。 6: 过采样 64x。 7: 过采样 128x。 8: 过采样 256x。

■ SAMPLERATE: 采样速率配置寄存器

SAMPLERATE 寄存器用于配置普通和连续采样时的采样速率。

表 17-15: SAMPLERATE 寄存器

位	Field	RW	复位值	描述
位 10~0	CC	读/写	0	[80..2047]: 捕获比较值, 采样速率=16 MHz/CC。
位 12	MODE	读/写	0	采样速率控制模式配置。 0: 采样速率由采样任务控制。这句话的意思是：应用程序触发采样任务的频度决定了采样速率。 1: 采样速率由本地定时器控制，也就是使用 CC 控制采样速率。

■ RESULT.PTR: EasyDMA 地址指针

表 17-16: RESULT.PTR 寄存器

位	Field	RW	复位值	描述
位 0~31	PTR	读/写	0	地址指针。

■ RESULT.MAXCNT: EasyDMA 计数

RESULT.MAXCNT 用于配置 EasyDMA 传输的最大缓存字数。

表 17-17: RESULT.MAXCNT 寄存器

位	Field	RW	复位值	描述
位 0~14	MAXCNT	读/写	0	传输的最大缓存字数。

- **RESULT_AMOUNT:** 自上次启动后传输的缓存字数。

表 17-18: RESULT_AMOUNT 寄存器

位	Field	RW	复位值	描述
位 14~0	AMOUNT	只读	0	自上次启动后传输的缓存字数。该寄存器的数值可以在 END 或 STOPPED 事件后读出。

4. 软件设计

4.1. 库函数的应用

SAADC 的驱动包含 HAL 和驱动层：

1. HAL: (Hardware Access Layer)硬件访问层，HAL 可以理解为驱动的最底层，它直接和硬件交互，存取硬件寄存器。
2. 驱动层：驱动层在 HAL 的基础上封装，它提供了 ADC 轮询采样，缓存处理，并且具备阻塞和非阻塞两种工作模式。驱动层提供了 API 函数供应用程序调用，我们在使用 ADC 库函数时主要就是调用驱动层的 API 来实现功能。

■ **ADC 库的关键特性:**

- 阻塞功能：用于单通道单次采样。
- 非阻塞功能：用于触发所有使能的通道进行转换，并将结果保存到指定的缓存。
- 双缓存：可以建立两个缓存，在第一个缓存写满后立即转到新的缓存。

❖ 注意：如果使能了过采样，只能使能一个通道，如果没有满足这个条件，使用 nrf_drv_saadc_channel_init 函数初始化通道时会返回错误。

SAADC 的每一个通道都可以独立配置，通道的配置和使能通过调用 nrfx_saadc_channel_init() 函数完成。配置通道时可以使用驱动程序中定义的两个包含默认参数的宏来进入配置，这两个宏定义分别对应单端输入模式和差分输入模式。

- NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_SE：单端输入模式。
- NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_DIFFERENTIAL：差分输入模式。

SAADC 的应用流程如下图所示，首先需要初始化 SAADC 程序模块，接着配置需要用到的 SAADC 通道和注册事件回调函数，之后是缓存配置，应用程序可以使用双缓存或者不使用缓存，最后执行 ADC 采样，ADC 采样有阻塞和非阻塞两种工作模式，应用程序根据需求调用对应的库函数执行阻塞和非阻塞采样。

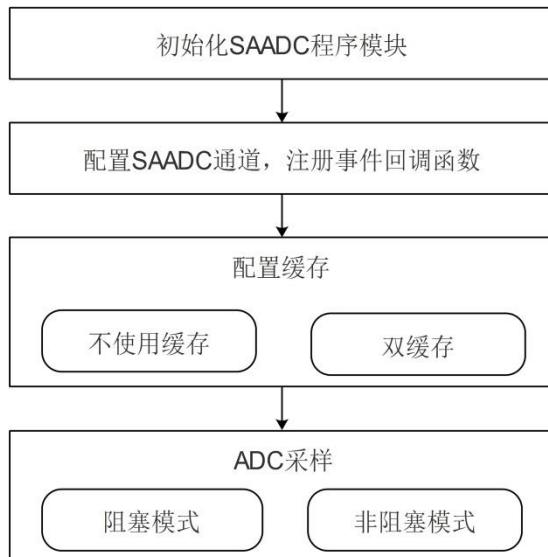


图 17-2: SAADC 应用步骤

4.1.1. 初始化 SAADC

SAADC 初始化的库函数是 nrf_drv_saadc_init() 函数，该函数配置 SAADC 的共有参数，包含分辨率、过采样、中断优先级和低功耗模式，函数原型如下表所示。

表 17-19: nrf_drv_saadc_init() 函数

函数原型	<code>_STATIC_INLINE ret_code_t nrf_drv_saadc_init((nrf_drv_saadc_config_t const * p_config, nrf_drv_saadc_event_handler_t event_handler)</code>
函数功能	初始化 SAADC。
参 数	[in] <code>p_config</code> : 指向 SAADC 配置结构体，如果指向 NULL，则使用默认的 SAADC 配置宏。 [in] <code>event_handler</code> : 应用程序提供的事件回调函数。
返回值	无。

一般地，调用 nrf_drv_saadc_init() 函数初始化 SAADC 时，会将 `p_config` 指向 NULL，这样，我们就不需要自己定义配置结构体，`nrf_drv_saadc_init()` 函数中会定义配置结构体并使用 SAADC 默认配置宏 NRFX_SAADC_DEFAULT_CONFIG 配置 SAADC，同时该宏中的参数定义在“ `sdk_config.h`”文件中，需要修改 SAADC 配置时，在“ `sdk_config.h`”文件中修改即可。

代码清单: nrf_drv_saadc_init() 函数

```

1. __STATIC_INLINE ret_code_t nrf_drv_saadc_init(  
2.                                     nrf_drv_saadc_config_t const * p_config,
  
```

```

3.                                     nrf_drv_saadc_event_handler_t event_handler)
4. {
5.     //p_config 指向了 NULL, 使用默认配置宏初始化 SAADC
6.     if (p_config == NULL)
7.     {
8.         //定义 SAADC 配置结构体, 并使用 NRFX_SAADC_DEFAULT_CONFIG 宏初始化
9.         static const nrfx_saadc_config_t default_config = NRFX_SAADC_DEFAULT_CONFIG;
10.
11.        //p_config 指向 SAADC 配置结构体
12.        p_config = &default_config;
13.    }
14.    //初始化 SAADC
15.    return nrfx_saadc_init(p_config, event_handler);
16. }

```

SAADC 默认配置宏 NRFX_SAADC_DEFAULT_CONFIG 代码清单如下:

代码清单：宏 NRFX_SAADC_DEFAULT_CONFIG

```

1. #define NRFX_SAADC_DEFAULT_CONFIG \
2. { \
3.     .resolution  = (nrf_saadc_resolution_t)NRFX_SAADC_CONFIG_RESOLUTION, \
4.     .oversample  = (nrf_saadc_oversample_t)NRFX_SAADC_CONFIG_OVERSAMPLE, \
5.     .interrupt_priority = NRFX_SAADC_CONFIG_IRQ_PRIORITY, \
6.     .low_power_mode      = NRFX_SAADC_CONFIG_LP_MODE \
7. }

```

宏 NRFX_SAADC_DEFAULT_CONFIG 配置的 4 个项目意义如下:

- resolution: 分辨率。
- oversample: 使能/禁止过采样。
- interrupt_priority: 中断优先级。
- low_power_mode: 使能/禁止低功耗模式。

它们的初始化值位于“sdk_config.h”文件中，使用 SAADC 时必须在“sdk_config.h”文件中启用 SAADC 并设置这些参数的值。

初始化 SAADC 时还需要提供事件回调函数，供 SAADC 事件调用，事件回调函数的格式如下，在事件回调函数中我们可以判断产生了哪些事件，并添加需要执行的功能代码。

代码清单：SAADC 事件回调函数

```

1. void saadc_callback(nrf_drv_saadc_evt_t const * p_event)
2. {
3.     //判断事件
4.     if (p_event->type == NRFX_SAADC_EVT_DONE)

```

```

5. {
6.     /* 功能代码*/
7. }
8. }
```

SAADC 可产生的事件有 3 个，这些事件定义在 nrfx_saadc_evt_type_t 中。

代码清单：SAADC 事件

```

1. typedef enum
2. {
3.     NRFX_SAADC_EVT_DONE,           //采样值填满缓存时产生该事件
4.     NRFX_SAADC_EVT_LIMIT,         //采样值达到门限值时产生该事件
5.     NRFX_SAADC_EVT_CALIBRATEDONE //校准完成后产生该事件
6. } nrfx_saadc_evt_type_t;
```

4.1.2. 配置 SAADC 通道

SAADC 初始化函数中配置了 SAADC 共有的参数，接下来需要配置使用的 SAADC 通道的参数。配置 SAADC 通道使用的库函数是 nrfx_saadc_channel_init() 函数，该函数原型如下表所示。

表 17-20: nrfx_saadc_channel_init() 函数

函数原型	nrfx_err_t nrfx_saadc_channel_init (uint8_t channel, nrf_saadc_channel_config_t const *const p_config)
函数功能	初始化 SAADC 通道。该函数配置和使能采样通道。
参 数	[in] channel : ADC 通道号。 [in] p_config : 指向配置结构体。
返回值	NRF_SUCCESS: 初始化成功。 NRF_ERROR_INVALID_STATE: SAADC 未初始化。 NRF_ERROR_NO_MEM: 待初始化的通道已经被分配过了。

nrf_drv_saadc_init() 函数有 2 个输入参数，其中，channel 是 SAADC 的通道号，取值范围是 0~7，p_config 指向通道配置结构体，所以，应用程序调用该函数配置通道时需要指定通道号，并提供已初始化参数的通道配置结构体。

5. SAADC 通道配置结构体

SAADC 驱动程序提供了一个通道配置结构体 nrf_saadc_channel_config_t，该结构体包

含了通道配置所有参数，其声明如下：

代码清单：SAADC 通道配置结构体 nrf_saadc_channel_config_t

```

1. typedef struct
2. {
3.     nrf_saadc_resistor_t    resistor_p; //通道正极电阻
4.     nrf_saadc_resistor_t    resistor_n; //通道负极电阻
5.     nrf_saadc_gain_t        gain;      //增益
6.     nrf_saadc_reference_t   reference; //参考电压
7.     nrf_saadc_acqtime_t    acq_time; //采样时间
8.     nrf_saadc_mode_t       mode;      //工作模式
9.     nrf_saadc_burst_t      burst;     //BURST 模式
10.    nrf_saadc_input_t     pin_p;     //通道正极连接的模拟引脚
11.    nrf_saadc_input_t     pin_n;     //通道负极连接的模拟引脚
12. } nrf_saadc_channel_config_t;

```

针对单端采样和差分采样，驱动程序提供了两个默认参数配置宏 NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_SE 和 NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_DIFFERENTIAL 用于初始化通道配置结构体。

代码清单：单端采样配置宏

```

1. #define NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_SE(PIN_P) \
2. { \
3.     //旁路通道正极电阻梯 \
4.     .resistor_p = NRF_SAADC_RESISTOR_DISABLED, \
5.     //旁路通道负极电阻梯 \
6.     .resistor_n = NRF_SAADC_RESISTOR_DISABLED, \
7.     //增益: 1/6 \
8.     .gain       = NRF_SAADC_GAIN1_6, \
9.     //内部参考源(0.6V) \
10.    .reference = NRF_SAADC_REFERENCE_INTERNAL, \
11.    //采样时间: 10us \
12.    .acq_time  = NRF_SAADC_ACQTIME_10US, \
13.    //单端采样 \
14.    .mode       = NRF_SAADC_MODE_SINGLE_ENDED, \
15.    //禁止 BURST 模式 \
16.    .burst      = NRF_SAADC_BURST_DISABLED, \
17.    //通道正极连接模拟输入引脚 PIN_P \
18.    .pin_p      = (nrf_saadc_input_t)(PIN_P), \
19.    //通道负极不连接模拟输入引脚 \
20.    .pin_n      = NRF_SAADC_INPUT_DISABLED \
21. }

```

代码清单：差分采样配置宏

```

1. #define NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_DIFFERENTIAL(PIN_P, PIN_N) \
2. { \
3.     //旁路通道正极电阻梯 \
4.     .resistor_p = NRF_SAADC_RESISTOR_DISABLED, \
5.     //旁路通道负极电阻梯 \
6.     .resistor_n = NRF_SAADC_RESISTOR_DISABLED, \
7.     //增益: 1/6 \
8.     .gain       = NRF_SAADC_GAIN1_6, \
9.     //内部参考源(0.6V) \
10.    .reference = NRF_SAADC_REFERENCE_INTERNAL, \
11.    //采样时间: 10us \
12.    .acq_time  = NRF_SAADC_ACQTIME_10US, \
13.    //差分采样 \
14.    .mode      = NRF_SAADC_MODE_DIFFERENTIAL, \
15.    //通道正极连接模拟输入引脚 PIN_P \
16.    .pin_p     = (nrf_saadc_input_t)(PIN_P), \
17.    //通道负极连接模拟输入引脚 PIN_N \
18.    .pin_n     = (nrf_saadc_input_t)(PIN_N) \
19. }

```

一般地，我们在定义通道配置结构体时会根据通道的采样模式使用对应的初始化配置宏进行初始化，如果需要修改通道的配置参数，我们会重写配置参数而不会在初始化宏里面修改。

- SAADC 通道配置示例：配置 SAADC 通道 2，单端采样。

```

nrf_saadc_channel_config_t channel_config =
    NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_AIN2);
err_code = nrfx_saadc_channel_init(0, &channel_config);

```

- SAADC 通道配置示例：配置 SAADC 通道 2，差分采样，重写增益为 1/2。

```

nrf_saadc_channel_config_t channel_config =
    NRF_DRV_SAADC_DEFAULT_CHANNEL_CONFIG_DIFFERENTIAL(NRF_SAADC_INPUT_AIN2,
    NRF_SAADC_INPUT_AIN0);
//增益重写为 1/2
channel_config = NRF_SAADC_GAIN1_2;
err_code = nrfx_saadc_channel_init(0, &channel_config);

```

4.1.3. 配置缓存

SAADC 驱动程序具备采样数据缓存功能，并且支持双缓存，当然我们也可以不使用缓存，不使用缓存时，每次采样完成后应用程序需要及时读取采样数据，否则，下一次采样完

成后，之前的采样数据就会被覆盖，为了防止数据被覆盖，CPU 需要频繁去读取采样值，这显然不利于大批量采样和采样数据的处理，所以，在实际应用中，我们一般都会使用缓存。

SAADC 配置缓存的库函数是 nrfx_saadc_buffer_convert()，该函数每次配置一个缓存，所以配置为双缓存时，需要在 SAADC 初始化时调用 2 次，函数原型如下表所示。

表 17-21: nrfx_saadc_buffer_convert()函数

函数原型	<pre>nrfx_err_t nrfx_saadc_buffer_convert (nrf_saadc_value_t * buffer, uint16_t size)</pre>
函数功能	<p>配置 SAADC 缓存。如果 ADC 处于 Idle 状态，该函数会为转换设置好 Easy DMA，之后 ADC 就绪并等待采样任务。</p> <ul style="list-style-type: none"> 如果设置了一个缓存，并且转换正在进行，这时候再次调用该函数，新的缓存将会被加入队列。当第一个缓存写满时，驱动程序将开始填充队列中的缓存。 如果在第一个缓存被写满之前或者正在校准时再次调用该函数，将会返回错误。
参数	<p>[in] buffer: 指向采样结果存放缓存。</p> <p>[in] size: 采样缓存大小（字数）。</p>
返回值	<p>NRF_SUCCESS: 采样转换成功。</p> <p>NRF_ERROR_BUSY: 如果驱动程序已经有了两个缓存或者正在校准。</p>

调用 nrfx_saadc_buffer_convert()函数配置缓存之前，需要定义用来存放采样数据的缓存数组，因为 SAADC 使用 EasyDMA，所以缓存数组需要位于片内 RAM 空间，即定义缓存数组的时候要加上“static”关键字。

一般为了方便管理，缓存数组会定义成 2 维数组，此处，为了直观，我们定义成 2 个单独的数组 m_buffer_pool_1 和 m_buffer_pool_2，数组长度为 SAMPLES_BUFFER_LEN。

代码清单：定义 SAADC 采样缓存数组

```
1. static nrf_saadc_value_t      m_buffer_pool_1[SAMPLES_BUFFER_LEN];
2. static nrf_saadc_value_t      m_buffer_pool_2[SAMPLES_BUFFER_LEN];
```

之后，在 SAADC 初始化时，调用 2 次 nrfx_saadc_buffer_convert()函数分别配置缓存 m_buffer_pool_1 和 m_buffer_pool_2。

代码清单：配置缓存

```
1. //配置缓存 m_buffer_pool_1, 等待应用程序启动采样
2. err_code = nrfx_saadc_buffer_convert(m_buffer_pool_1, SAMPLES_BUFFER_LEN);
```

```

3. APP_ERROR_CHECK(err_code);
4. //配置缓存 m_buffer_pool_2, 等待应用程序启动采样
5. err_code = nrfx_saadc_buffer_convert(m_buffer_pool_2, SAMPLES_BUFFER_LEN);
6. APP_ERROR_CHECK(err_code);

```

SAADC 的缓存配置使用起来很简单, 双缓存的配置都是通过 nrfx_saadc_buffer_convert() 函数完成, 但是 SAADC 中最不好理解的也是缓存的配置和工作流程, 接下来, 我们来分析一下双缓存的工作流程。

1. SAADC 初始化时缓存的配置

SAADC 驱动程序中的控制块 m_cb 有 2 个指针: 一级缓存指针 “p_buffer” 和二级缓存指针 “p_secondary_buffer”, 分别用来指向缓存 1 和缓存 2。SAADC 采样的数据永远存储一级缓存指针指向的缓存, 驱动程序负责完成两个缓存的轮询使用。

初始化时, 第一次调用 nrfx_saadc_buffer_convert() 后, 一级缓存指针指向缓存 1 “m_buffer_pool_1”, 一级缓存计数保存缓存 1 的大小。同时, 配置了 SAADC 的 EasyDMA, 即一级缓存指针赋值给 EasyDMA 的地址指针 “RESULT.PTR”, 一级缓存计数赋值给 “RESULT.MAXCNT”。这样, 启动采样后, 采样数据会存储到一级缓存指针, 即缓存数组 m_buffer_pool_1。

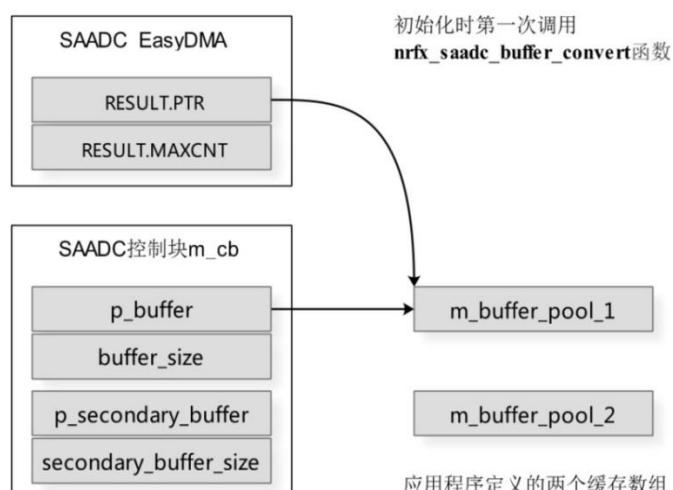


图 17-3: 第一次调用缓存配置函数

第二次调用 nrfx_saadc_buffer_convert() 后, 二级缓存指针指向缓存 2 “m_buffer_pool_2”, 二级缓存计数保存缓存 2 的大小。

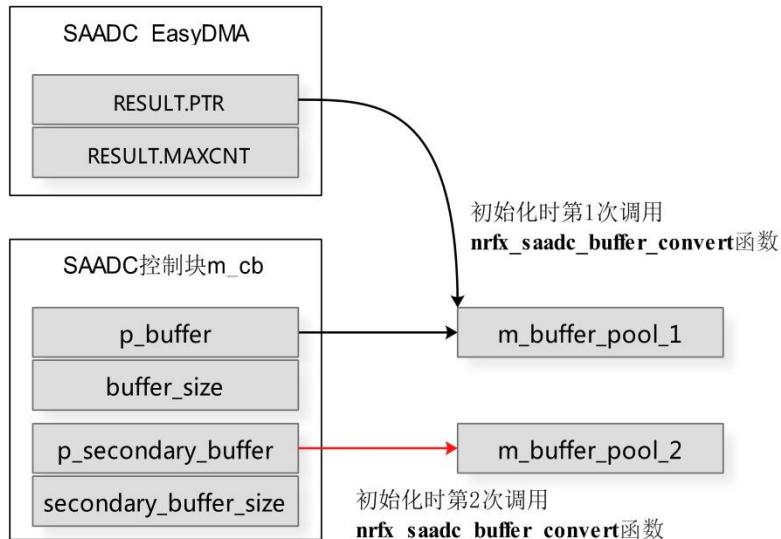


图 17-4: 第二次调用缓存配置函数

2. 双缓存的轮询使用

初始化时双缓存配置好后，EasyDMA 首先使用的是缓存 1。启动采样后，当采样数据填满缓存“m_buffer_pool_1”时（采样数据的数量等于 RESULT.MAXCNT），产生 EVENTS_END 事件中断，中断服务函数中首先将“m_buffer_pool_1”的地址赋值给 SAADC 事件结构体，并声明事件类型为“NRFX_SAADC_EVT_DONE”，之后进入事件回调函数。应用程序在事件回调函数中获取“NRFX_SAADC_EVT_DONE”事件，并通过事件结构体获取采样数据。

代码清单：SAADC 中断服务函数中对事件结构体赋值

```

1. evt.type          = NRFX_SAADC_EVT_DONE;
2. evt.data.done.p_buffer = (nrf_saadc_value_t *)m_cb.p_buffer;
3. evt.data.done.size     = m_cb.buffer_size;
  
```

中断服务程序中将缓存 1 的信息赋值给事件结构体后，接着将控制块 m_cb 的一级缓存指针指向缓存 2 “m_buffer_pool_2”，一级缓存计数保存缓存 2 的大小，同时将二级缓存指针设置为 NULL，代码清单如下。

代码清单：SAADC 中断服务函数中切换缓存

```

1. m_cb.p_buffer      = m_cb.p_secondary_buffer;
2. m_cb.buffer_size    = m_cb.secondary_buffer_size;
3. m_cb.p_secondary_buffer = NULL;
  
```

这时，虽然设置了一级缓存指针指向缓存 2，但是并没有设置 EasyDMA，同时缓存 1 中的采样数据也没有取走。接下来，中断服务程序中执行回调，向应用程序递交事件。应用程序在 SAADC 事件回调函数中再次调用 nrfx_saadc_buffer_convert() 函数，将缓存 1 的地址赋值给二级缓存指针，并将一级缓存指针和一级缓存计数赋值给 EasyDMA 的地址指针和计数寄存器，之后读取缓存 1 的数据，这样就完成了 2 个缓存的轮询使用。

4.1.4. 启动 SAADC 采样

启动 SAADC 采样有 2 个库函数 `nrfx_saadc_sample_convert()` 和 `nrfx_saadc_sample()`，分别用于堵塞和非堵塞采样。

1. `nrfx_saadc_sample_convert()`

用于堵塞模式采样，使用该函数时，应用程序不需要初始化和使能 ADC 通道，该函数会根据通道的配置信息启动一次 ADC 采样，并等待采样完成，返回采样结果，函数原型如下表所示。

表 17-22: `nrfx_saadc_sample_convert()` 函数

函数原型	<code>nrfx_err_t nrfx_saadc_sample_convert((uint8_t channel, nrf_saadc_value_t * p_value)</code>
函数功能	该函数工作于堵塞模式，用于对指定通道执行单次 ADC 采样：根据通道的配置信息启动一次 ADC 采样，并等待采样完成，返回采样结果。 <ul style="list-style-type: none">• 如果 ADC 忙，该函数返回“ADC 忙”，采样不会被执行。• 调用该函数之前，应用程序不需要初始化和使能 ADC 通道，函数内部会在采样之前使能 ADC 通道。
参数	[in] <code>channel</code> : ADC 通道号。 [in] <code>p_value</code> : 指向用于保存 ADC 采样结果的变量。
返回值	<code>NRF_SUCCESS</code> : 采样转换成功。 <code>NRF_ERROR_BUSY</code> : ADC 忙。

2. `nrfx_saadc_sample()`

`nrfx_saadc_sample()` 函数仅启动 SAADC 采样，该函数用于非堵塞模式下启动采样。

表 17-23: `nrfx_saadc_sample()` 函数

函数原型	<code>nrfx_err_t nrfx_saadc_sample(void)</code>
函数功能	启动 SAADC 采样。
参数	无。
返回值	<code>NRF_SUCCESS</code> : SAADC 已启动。 <code>NRF_ERROR_INVALID_STATE</code> : SAADC 处于 IDLE 状态。

4.2. 堵塞模式-单端输入采样实验

本实验在“实验 10-1：串口数据收发”的基础上修改。配置 nRF52840 的 SAADC 通道 0 为单端采样，通道正极连接模拟输入引脚 AIN0 (即引脚 P0.02)，采样电位器抽头电压，程

序中使用阻塞采样函数 `nrfx_saadc_sample_convert()` 每 500 毫秒执行一次电压采样，采样结果通过串口输出。

❖ 注：本节对应的实验源码是：“实验 17-1：SAADC 堵塞模式-单端输入采样”。

4.2.1. 添加需要的文件

SAADC 需要加入的文件如下表所示。

表 17-24：GPIOE 需要加入的文件

文件名	SDK 中的目录	描述
<code>nrfx_saadc.c</code>	<code>..\modules\nrfx\drivers\src</code>	ADC 驱动文件。

4.2.2. 头文件引用和路径设置

1. 需要引用的头文件

因为在“main.c”文件中使用了 SAADC 驱动程序，所以“main.c”文件需要引用下面的头文件。

```
#include "nrf_drv_saadc.h"
```

2. 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

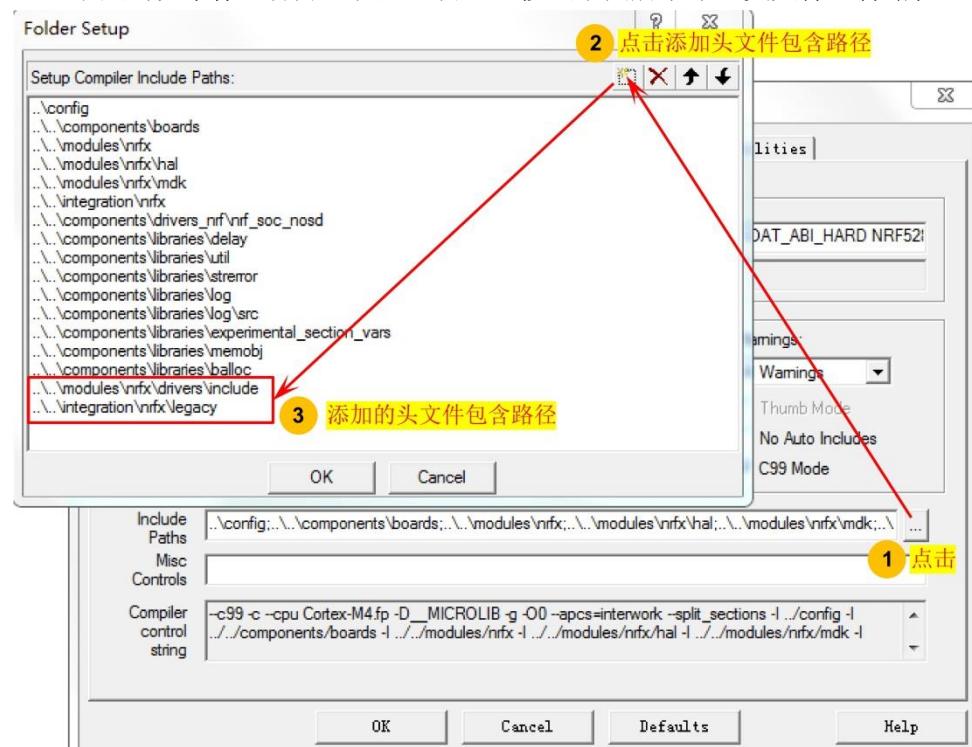


图 17-5：添加头文件包含路径

SAADC 需要添加的头文件路径如下表：

表 17-25: 头文件包含路径

序号	路径
1	..\..\modules\nrfx\drivers\include
2	..\..\integration\nrfx\legacy

4.2.3. 工程配置

打开“sdk_config.h”文件，加入 SAADC 需要的配置，如下图所示，编辑内容时切换到“Text Editor”进行编辑（编辑的内容参考实验的源码，因此代码很长，此处不贴出代码，参见“sdk_config.h”文件的（144~306）行），编辑完成后，切换到“Configuration Wizard”，即可观察到配置的具体项目：

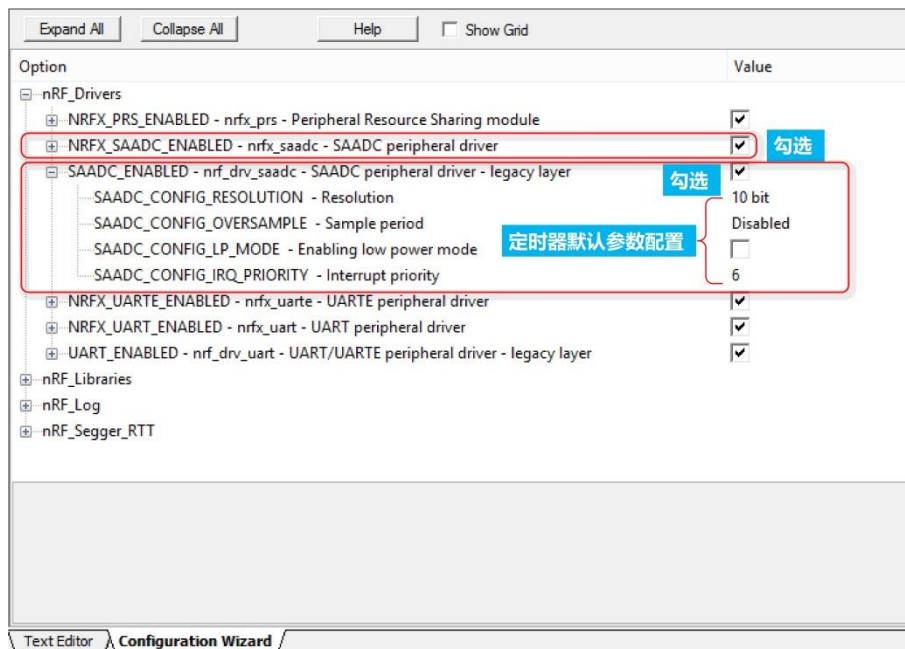


图 17-8: 工程配置

可以看到，“sdk_config.h”文件里面配置了 SAADC 的共有参数，这里配置了 10 位分辨率、禁止过采样，中断优先级为 6。

4.2.4. 代码编写

本例要实现的是采样电位器电压（电位器抽头连接到模拟输入通道 0）。首先需要调用 nrf_drv_saadc_init() 函数初始化 SAADC 程序模块和 SAADC 的共有参数，之后定义通道配置结构体并使用单端采样配置宏初始化结构体，因为使用的是模拟输入通道 2，所以单端采样配置宏的实参是 NRF_SAADC_INPUT_AIN0，程序清单如下。

代码清单：SAADC 初始化函数

1. //初始化 SAADC，配置使用的 SAADC 通道的参数
2. **void** saadc_init(**void**)
3. {
4. ret_code_t err_code;

```

5.     //定义 ADC 通道配置结构体，并使用单端采样配置宏初始化,
6.     //NRF_SAADC_INPUT_AIN0 是使用的模拟输入通道
7.     nrf_saadc_channel_config_t channel_config =
8.         NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_AIN0);
9.     //初始化 SAADC，注册事件回调函数。
10.    err_code = nrf_drv_saadc_init(NULL, saadc_callback);
11.    APP_ERROR_CHECK(err_code);
12.    //初始化 SAADC 通道 0，注意函数参数的 0 表示的是该 SAADC 通道在 SAADC 驱动 SAADC 管理
13.    //数组中的位置
14.    err_code = nrfx_saadc_channel_init(0, &channel_config);
15.    APP_ERROR_CHECK(err_code);
16.
17. }
```

本例中是调用阻塞采样函数 `nrftx_saadc_sample_convert()`执行堵塞采样，该函数启动 SAADC 后会一直等待采样完成后退出，所以不需要通过 SAADC 事件回调函数判断采样是否完成，这里我们提供一个空的 SAADC 事件回调函数给初始化函数 `nrf_drv_saadc_init()`即可。

代码清单：SAADC 事件回调函数

```
1. void saadc_callback(nrf_drv_saadc_evt_t const * p_event){}
```

主函数中调用 `saadc_init()`函数初始化 SAADC，之后在主循环里面每隔 500ms 调用一次 `nrftx_saadc_sample_convert()`执行一次 ADC 采样，采样值和计算后的电压值通过串口输出。

代码清单：采样电位器电压

```

1. int main(void)
2. {
3.     uint32_t err_code;
4.     nrf_saadc_value_t saadc_val;
5.
6.     //初始化开发板上的 4 个 LED，即将驱动 LED 的 GPIO 配置为输出,
7.     bsp_board_init(BSP_INIT_LEDS);
8.     //串口初始化
9.     uart_config();
10.    //初始化 SAADC
11.    saadc_init();
12.
13.    while(true)
14.    {
15.        //启动一次 ADC 采样（阻塞模式）。
16.        nrftx_saadc_sample_convert(0,&saadc_val);
17.        printf("Sample value = %d\r\n", saadc_val);
```

```
18.  
19.     //串口输出采样值计算得到的电压值。电压值 = 采样值 * 3.6 /1024  
20.     printf("Voltage = %.3fV\r\n", saadc_val * 3.6 /1024);  
21.     //延时 500ms，方便观察 SAADC 采样数据  
22.     nrf_delay_ms(500);  
23.     //翻转开发板上的指示灯 D1 的状态，指示一次采样的完成  
24.     nrf_gpio_pin_toggle(LED_1);  
25. }  
26. }
```

4.2.5. 硬件连接

本实验需要使用 P0.13 驱动 LED 指示灯 D1，P0.02 连接电位器，P0.06 和 P0.08 作为串口通讯引脚，按照下图所示短接跳线帽。

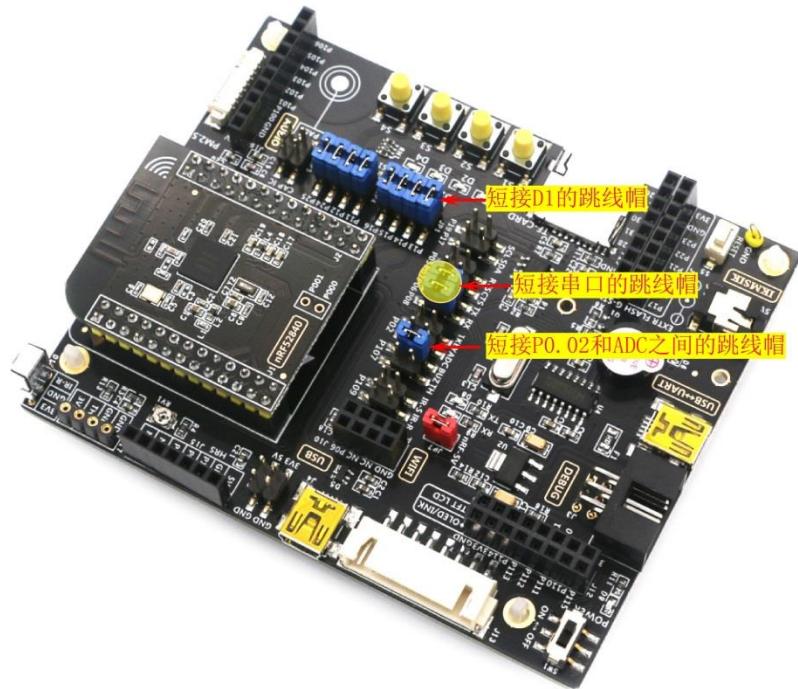


图 17-6：开发板跳线帽短接

4.2.6. 实验步骤

1. 解压“…\3: 开发指南（上册）配套试验源码\”目录下的压缩文件“实验 17-1: SAADC 堵塞模式-单端输入采样”，将解压后得到的文件夹“saadc_single_blocking”拷贝到合适的目录，如“D\ nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\saadc_single_blocking\project\mdk5”目录下的工程“saadc_single_blocking.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qfia.hex”位于工程目录下

的“Objects”文件夹中。

5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，每 500ms 采样一次电位器抽头电压，并通过串口输出采样值和计算后的电压值，用螺丝刀旋转电位器旋钮，可以看到电压值的变化。

4.3. 堵塞模式-差分输入采样实验

本实验在“实验 17-1：SAADC 堵塞模式-单端输入采样”的基础上修改。配置 nRF52840 的 SAADC 通道 0 为差分采样，通道正极连接 VDD，通道负极连接模拟输入引脚 AIN0 (即引脚 P0.02)，采样 VDD 和电位器抽头电压的差值，程序中使用阻塞采样函数 nrfx_saadc_sample_convert() 每 500 毫秒执行一次电压采样，采样结果通过串口输出。

❖ 注：本节对应的实验源码是：“实验 17-2：SAADC 堵塞模式-差分输入采样”。

4.3.1. 代码编写

本例要实现的是差分采样 VDD 和电位器电压的差值，在 SAADC 配置函数 saadc_init() 中我们需要将 SAADC 通道配置为差分采样，也就是在定义通道配置结构体时使用差分配置宏 NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_DIFFERENTIAL 初始化结构体，程序清单如下。

代码清单：SAADC 初始化函数

```
1. void saadc_init(void)
2. {
3.     ret_code_t err_code;
4.     // 定义 ADC 通道配置结构体，并使用差分采样配置宏初始化,
5.     // 因为要采样的是 VDD 和电位器抽头电压（模拟输入 NRF_SAADC_INPUT_AIN2）的差值
6.     // 所以差分采样配置宏的通道正极为：NRF_SAADC_INPUT_VDD，通道负极为：
7.     // NRF_SAADC_INPUT_AIN2
8.     nrf_saadc_channel_config_t channel_config =
9.         NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_DIFFERENTIAL(NRF_SAADC_INPUT_VDD,
10.                                                       NRF_SAADC_INPUT_AIN2);
11.    // 初始化 SAADC，注册事件回调函数。
12.    err_code = nrf_drv_saadc_init(NULL, saadc_callback);
13.    APP_ERROR_CHECK(err_code);
14.    // 初始化 SAADC 通道 0，注意函数参数的 0 表示的是该 SAADC 通道在 SAADC 驱动 SAADC 管理数
15.    // 组中的位置
16.    err_code = nrfx_saadc_channel_init(0, &channel_config);
17.    APP_ERROR_CHECK(err_code);
18. }
```

对于差分采样，还需要注意采样值计算为电压值时和单端采样是有差别的，差分采样时“式 16-1”中的 $m=1$ ，而单端模式时等于 0。所以计算时，对于 10 位分辨率，差分采样是除以 512 (2^{10-1})，单端采样是除以 1024 (2^{10})。

代码清单：采样值计算为电压值

```
1. //串口输出采样值计算得到的电压值。电压值 = 采样值 * 3.6 /2^(10-1)
2. printf("Differential Voltage = %.3fV\r\n", saadc_val * 3.6 /512);
```

4.3.2. 硬件连接

同“实验 17-1：SAADC 堵塞模式-单端输入采样”。

4.3.3. 实验步骤

1. 解压“…\3：开发指南（上册）配套试验源码\”目录下的压缩文件“实验 17-2：SAADC 堵塞模式-差分输入采样”，将解压后得到的文件夹“saadc_diff_blocking”拷贝到合适的目录，如“D\ nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\saadc_diff_blocking\project\mdk5”目录下的工程“saadc_diff_blocking.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qfaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，每 500ms 采样一次电位器抽头电压，并通过串口输出采样值和计算后的电压值，用螺丝刀旋转电位器旋钮，可以看到电压值的变化。。

4.4. 非堵塞模式-双缓存采样实验

前面的两个实验使用的是堵塞模式，采样完成后直接读取数据，没有使用缓存保存采样数据。但是在实际应用中，我们更希望 SAADC 工作在非堵塞模式，并且采样数据自动保存到我们定义的缓存里面，同时，可以配置采样多少个数据后向应用程序提交事件。接下来，我们使用 SAADC 库编写代码实现这些功能。

本实验在“实验 17-1：SAADC 堵塞模式-单端输入采样”的基础上修改。配置 nRF52840 的 SAADC 通道 0 为单端采样，通道正极连接模拟输入通道 0（对应引脚 P0.02，连接的是电位器抽头），程序中配置双缓存用于存放采样数据，每 500ms 使用 nrfx_saadc_sample() 函数启动一次采样，采样结果通过串口输出。

❖ 注：本节对应的实验源码是：“实验 17-3：SAADC 非堵塞模式-双缓存采样”。

4.4.1. 代码编写

因为本例中使用双缓存，所以需要先定义缓存数组，这里定义的双缓存是二维数组。同时，为了观察事件产生是采样次数之间的关系，定义了变量 m_adc_evt_counter 用来记录事件产生的次数。

代码清单：定义缓存数组

```
1. //定义 SAADC 采样缓存数组大小，这个参数极其重要，它决定了 SAADC 采样多少个数据后产生事件中断
2. #define SAMPLES_BUFFER_LEN 4
3.
4. //定义 SAADC 采样缓存数组，这里定义成 2 维数组，也可以定义成 2 个一维数组，它们本质上是一样的
5. static nrf_saadc_value_t      m_buffer_pool[2][SAMPLES_BUFFER_LEN];
6. //static nrf_saadc_value_t      m_buffer_pool_1[SAMPLES_BUFFER_LEN];
7. //static nrf_saadc_value_t      m_buffer_pool_2[SAMPLES_BUFFER_LEN];
8. //定义一个 32 的变量，用来保存进入事件回调函数的次数。
9. static uint32_t      m_adc_evt_counter;
```

SAADC 初始化时，调用两次 nrfx_saadc_buffer_convert() 函数配置缓存，程序清单如下。
代码清单：SAADC 初始化函数

```
1. void saadc_init(void)
2. {
3.     ret_code_t err_code;
4.     //定义 ADC 通道配置结构体，并使用单端采样配置宏初始化
5.     //因为要采样的是电位器抽头电压（模拟输入 NRF_SAADC_INPUT_AIN0），
6.     //所以单端采样配置宏的通道正极为：NRF_SAADC_INPUT_AIN0
7.     nrf_saadc_channel1_config_t channel_config =
8.     NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_AIN0);
9.     //初始化 SAADC，注册事件回调函数。
10.    err_code = nrf_drv_saadc_init(NULL, saadc_callback);
11.    APP_ERROR_CHECK(err_code);
12.    //初始化 SAADC 通道 0，注意函数参数的 0 表示的是该 SAADC 通道在 SAADC 驱动 SAADC 管理
13.    //数组中的位置
14.    err_code = nrfx_saadc_channel_init(0, &channel_config);
15.    APP_ERROR_CHECK(err_code);
16.
17.    //配置缓存 1，将缓存 1 地址赋值给 SAADC 驱动程序中的控制块 m_cb 的一级缓存指针
18.    err_code = nrfx_saadc_buffer_convert(m_buffer_pool[0], SAMPLES_BUFFER_LEN);
19.
20.    APP_ERROR_CHECK(err_code);
//配置缓存 2，将缓存 1 地址赋值给 SAADC 驱动程序中的控制块 m_cb 的二级缓存指针
```

```
21.     err_code = nrfx_saadc_buffer_convert(m_buffer_pool[1], SAMPLES_BUFFER_LEN);  
22.     APP_ERROR_CHECK(err_code);  
23. }
```

SAADC 事件回调函数中，调用 nrfx_saadc_buffer_convert() 函数设置 EasyDMA，即将控制块的一级缓存指针和一级缓存计数赋值给 EasyDMA 的地址指针和计数器。同时将当前存储采样数据的缓存的地址和数据长度赋值给控制块的二级缓存指针和二级缓存计数，之后，应用程序读取采样数据，本例中，我们读出采样数据后通过串口输出数据。

事件回调函数中 m_adc_evt_counter 记录了事件产生的次数，本例中我们使用了 SAADC 的一路通道，所以每次采样得到一个采样数据，应用程序每 500ms 执行一次采样，缓存大小设置的是 5，所以需要执行 5 次采样才能填满一个缓存，也就是每执行 5 次采样约 2.5 秒会产生一次 NRF52840_SAADC_EVT_DONE 事件。

代码清单：SAADC 事件回调函数

```
1. void saadc_callback(nrfx_saadc_evt_t const * p_event)  
2. {  
3.     float val; //保存 SAADC 采样数据计算的实际电压值  
4.  
5.     if (p_event->type == NRF52840_SAADC_EVT_DONE)  
6.     {  
7.         ret_code_t err_code;  
8.         //设置好缓存，为下一次采样准备  
9.         err_code = nrfx_saadc_buffer_convert(p_event->data.done.p_buffer,  
10.                                              SAMPLES_BUFFER_LEN);  
11.        APP_ERROR_CHECK(err_code);  
12.        int i;  
13.        //串口输出 ADC 采样值。  
14.        printf("\r\nADC event number: %d\r\n", (int)m_adc_evt_counter);  
15.        for (i = 0; i < SAMPLES_BUFFER_LEN; i++)  
16.        {  
17.            //如果直接输出采样结果，使用这个代码  
18.            printf("Sample value: %d    ", p_event->data.done.p_buffer[i]);  
19.            //串口输出采样值计算得到的电压值。电压值 = 采样值 * 3.6 /2^10  
20.            val = p_event->data.done.p_buffer[i] * 3.6 /1024;  
21.            printf("Voltage = %.3fV\r\n", val);  
22.        }  
23.        //事件次数加 1  
24.        m_adc_evt_counter++;  
25.        printf("\r\n");  
26.    }  
27. }
```

主循环中，每 500ms 调用 nrfx_saadc_sample()函数执行一次采样，这样，每 2.5 秒产生一次 NRF_SAADC_EVT_DONE 事件。

代码清单：主循环

```
1. while(true)
2. {
3.     //启动一次 ADC 采样。
4.     nrfx_saadc_sample();
5.
6.     //翻转开发板上的指示灯 D1 的状态，指示一次采样的完成
7.     nrf_gpio_pin_toggle(LED_1);
8.     //延时 500ms，方便观察 SAADC 采样数据
9.     nrf_delay_ms(500);
10. }
```

4.4.2. 硬件连接

同“实验 17-1：SAADC 堵塞模式-单端输入采样”。

4.4.3. 实验步骤

1. 解压“…\3：开发指南（上册）配套试验源码\”目录下的压缩文件“实验 17-3：SAADC 非堵塞模式-双缓存采样”，将解压后得到的文件夹“saadc_single_nonblocking_db”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\saadc_single_nonblocking_db\project\mdk5”目录下的工程“saadc_single_nonblocking_db.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，每 500ms 采样一次电位器抽头电压，本例中使用了 SAADC 的通道 0，缓存大小设置的是 5，所以，事件产生的间隔是：500ms×5=2.5s。串口调试助手中我们可以观察到：每 2.5 秒输出一次数据，输出的数据中包含 5 次采样的数据，事件计数每次加 1。用螺丝刀旋转电位器旋钮，可以看到电压值的变化。

```

ADC event number: 0
Sample value is: 426 Voltage = 1.498V
Sample value is: 423 Voltage = 1.487V
Sample value is: 423 Voltage = 1.487V
Sample value is: 424 Voltage = 1.491V
Sample value is: 422 Voltage = 1.484V

ADC event number: 1
Sample value is: 422 Voltage = 1.484V
Sample value is: 425 Voltage = 1.494V
Sample value is: 425 Voltage = 1.494V
Sample value is: 424 Voltage = 1.491V
Sample value is: 426 Voltage = 1.498V

```

图 17-7：串口调试助手接收的数据

4.5. 电池电压采样实验

穿戴式或便携式设备中，电池是必不可少的一个部件，常用的电池有纽扣电池和锂离子电池，下图是两种纽扣电池 CR2032 和 CR2450。



图 17-8：纽扣电池

CR 系列的纽扣电池一般电压范围是(2.0~3.6)V，nRF52840 的工作电压范围也是(2.0~3.6)V，所以，使用 CR 系列的纽扣电池供电时，可以直接将电池连接到芯片的供电引脚，这种情况下，我们通过 SAADC 采样芯片的供电电压的值就等于电池的电压值。

常用的锂离子电池一般额定电压是 3.7V，最高电压 4.2V，对于这种电池，不能直接通过 SAADC 进行采样，需要使用分压器(如电阻等)先分压，然后对分压后的电压进行采样，再通过分压系数计算出电池的电压。这种情况下，直接用上文的实验 1、实验 2 或实验 3 都可以实现，所不同的就是需要通过分压系数计算一下实际的电池电压。

本例演示的是使用 SAADC 对芯片的供电电压进行采样，即使用纽扣电池供电时的 SAADC 采样方法。因为，直接对芯片供电电压进行采样时，可以不用任何模拟输入引脚，也就是不需要进行任何的连接，即可实现对供电电压进行采样，这种方式在很多场合下用处很大，并且也很方便(因为不需要使用任何 IO 和连线)。

本实验在“实验 17-3：SAADC 非堵塞模式-双缓存采样”的基础上修改。对于采样芯片 VDD 电压，我们只需在 SAADC 初始化函数中将 SAADC 通道 0 的通道正极连接 VDD

即可。

❖ 注：本节对应的实验源码是：“实验 17-4：电池电压采样”。

4.5.1. 代码编写

本例要采样的是纽扣电池的电压，也就是采样芯片 VDD，所以只需在 SAADC 初始化函数 `saadc_init()` 中修改单端配置宏 `NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_SE` 的实参，将其改为 `NRF_SAADC_INPUT_VDD` 即可，程序清单如下。

代码清单：SAADC 初始化函数

```
1. ret_code_t err_code;
2. //定义 ADC 通道配置结构体，并使用单端采样配置宏初始化
3. //因为要采样的是芯片的 VDD，所以单端采样配置宏的通道正极为：NRF_SAADC_INPUT_VDD
4. nrf_saadc_channel_config_t channel_config =
5.     NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_VDD);
```

4.5.2. 硬件连接

本实验需要使用 P0.13 驱动 LED 指示灯 D1，P0.06 和 P0.08 作为串口通讯引脚，按照下图所示短接跳线帽。

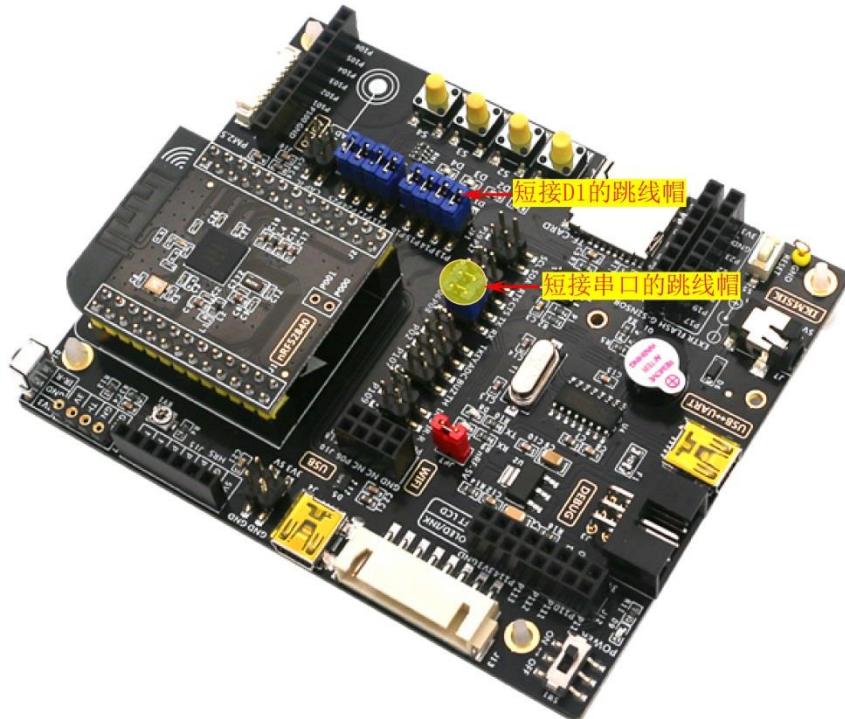


图 17-9：开发板跳线帽短接

4.5.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套试验源码\”目录下的压缩文件“实验 17-4：电池

- 电压采样”，将解压后得到的文件夹“saadc_battery”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.23。
 3. 在 MDK5 中执行 “Project→Open Project” 打开 “…\saadc_battery\project\mdk5” 目录下的工程 “saadc_battery.uvproj”。
 4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件 “nrf52840_qiaa.hex” 位于工程目录下的 “Objects” 文件夹中。
 5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
 6. 程序运行后，每 500ms 采样一次芯片 VDD，串口调试助手每 2.5 秒输出一次数据，输出的数据中包含 5 次采样的数据。

4.6. 门限监测实验

本实验在“实验 17-3: SAADC 非堵塞模式-双缓存采样”的基础上修改。配置 nRF52840 的 SAADC 通道 0 的通道正极连接模拟输入 AIN0 (即引脚 P0.02)，采样电位器抽头电压，程序中使用双缓存，缓存大小设置为 1，程序每 500ms 调用 nrfx_saadc_sample 函数启动一次采样，采样结果通过串口输出，程序中分别设置通道 0 的门限值如下，观察门限事件产生的条件：

- 只设置上限：门限值 300，当采样值超过该数值后产生 NRF_DRV_SAADC_EVT_LIMIT 事件。
 - 只设置下限：门限值 100，当采样值低于该数值后产生 NRF_DRV_SAADC_EVT_LIMIT 事件。
 - 同时设置上下限：高门限值 300，低门限值 100，当采样值超过上限值或低于下限值时都会产生 NRF_DRV_SAADC_EVT_LIMIT 事件。
- ❖ 注意：门限值是可以设置为负数的，如使用差分输入时，采样结果可能为负值，这时可以设置负值门限。
- ❖ 注：本节对应的实验源码是：“实验 17-5：门限监测”。

4.6.1. 代码编写

增加门限监测时，需要在 SAADC 初始化函数 saadc_init()中加入设置门限的代码。这里我们分别设置通道 0 的上限、下限和上/下限，程序中默认使用的是上限，下限和上/下限是屏蔽的，测试下限和上/下限时取消程序屏蔽即可，程序清单如下。

代码清单：SAADC 初始化函数中加入门限设置

```
1. void saadc_init(void)
2. {
3.     ret_code_t err_code;
4.     //定义 ADC 通道配置结构体，并使用单端采样配置宏初始化
```

```

5. //因为要采样的是电位器抽头电压，所以单端采样配置宏的通道正极为: RF_SAADC_INPUT_AIN0
6. nrf_saadc_channel_config_t channel_config =
7.     NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_AIN0);
8. //初始化 SAADC，注册事件回调函数。
9. err_code = nrf_drv_saadc_init(NULL, saadc_callback);
10. APP_ERROR_CHECK(err_code);
11.
12. //只设置上限值：设置为 300
13. nrfx_saadc_limits_set(0,NRFX_SAADC_LIMITL_DISABLED,300);
14.
15. //只设置下限值：设置为 100
16. //nrfx_saadc_limits_set(0,100,NRFX_SAADC_LIMITH_DISABLED);
17.
18. //同时设置上/下限值：上限值设置为 300，下限值设置为 100
19. //nrfx_saadc_limits_set(0,100,300);
20.
21. //初始化 SAADC 通道 0，注意函数参数的 0 表示的是该 SAADC 通道在 SAADC 驱动 SAADC 管理数组
22. //中的位置
23. err_code = nrfx_saadc_channel_init(0, &channel_config);
24. APP_ERROR_CHECK(err_code);
25.
26. //配置缓存 1，将缓存 1 地址赋值给 SAADC 驱动程序中的控制块 m_cb 的一级缓存指针
27. err_code = nrfx_saadc_buffer_convert(m_buffer_pool[0], SAMPLES_BUFFER_LEN);
28. APP_ERROR_CHECK(err_code);
29. //配置缓存 2，将缓存 1 地址赋值给 SAADC 驱动程序中的控制块 m_cb 的二级缓存指针
30. err_code = nrfx_saadc_buffer_convert(m_buffer_pool[1], SAMPLES_BUFFER_LEN);
31. APP_ERROR_CHECK(err_code);
32. }

```

设置门限后，当采样值达到门限值时会产生 NRF_DRV_SAADC_EVT_LIMIT 事件，SAADC 事件回调函数中可以加入处理门限事件的代码，本例在门限事件中通过串口输出提示信息，代码清单如下。

代码清单：门限事件

```

1. //门限事件
2. if(p_event->type == NRF_DRV_SAADC_EVT_LIMIT)
3. {
4.     printf("Limit Event\r\n");
5. }

```

4.6.2. 硬件连接

同“实验 17-1：SAADC 堵塞模式-单端输入采样”。

4.6.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套试验源码\”目录下的压缩文件“实验 17-5：电压门限监测”，将解压后得到的文件夹“saadc_limit”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\saadc_limit\project\mdk5”目录下的工程“saadc_limit.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，每 500ms 采样一次电位器抽头电压，本例中默认设置了门限的上限值为 300，所以，当采样值超过 300 后，会产生 NRF_DRV_SAADC_EVT_LIMIT，串口会输出信息“Limit Event”，用螺丝刀旋转电位器旋钮，改变电位器抽头电压，观察上限时的门限事件。分别设置门限监测的下限和上/下限（下限和上/下限设置的代码默认是屏蔽的，测试时取消屏蔽即可），旋转电位器旋钮，改变电位器抽头电压，观察下限和上/下限时的门限事件。

第十八章：片内温度传感器

1. 学习目的

1. 掌握片内温度传感器的使用步骤。
2. 掌握片内温度传感器测量值计算为温度值的方法。

2. TEMP 原理

nRF52840 片内集成了一个温度传感器，用来测量芯片温度。片内温度传感器具有线性补偿功能，测量温度时，应用程序可以通过配置相关寄存器，实现线性补偿。片内温度传感器主要特性如下：

- 温度测量范围大于或等于芯片的工作温度。
- 温度测量分辨率：0.25 °C。

片内温度传感器通过触发 START 任务启动，当温度测量完成后，产生 DATARDY 事件，这时可以通过 TEMP 寄存器读取片内温度传感器的测量值。

为了达到电气规范中规定的测量精度，片内温度传感器必须使用 HFCLK 晶体振荡器作为时钟源。当温度测量完成后，片内温度传感器 TEMP 的模拟电路会关闭以降低功耗。

片内温度传感器 TEMP 仅支持单次测量，不支持连续测量，所以，每次测量时都需要通过触发 START 任务启动测量。也就是，触发一次 START 任务只能完成一次测量，如果需要再次测量，就必须再次触发 START 任务。

片内温度传感器电气规格如下表所示：

表 18-1: 片内温度传感器电气规范

RTC	描述	最小	典型值	最大值	单位
t_{TEMP}	温度测量所需时间	—	36	—	μs
$T_{TEMP,RANGE}$	温度范围	-40	—	85	°C
$T_{TEMP,ACC}$	温度传感器精度	-5	—	+5	°C
$T_{TEMP,RES}$	温度传感器分辨率	—	0.25	—	°C

3. TEMP 寄存器

片内温度传感器 TEMP 的寄存器基址是 0x4000C000，包含的寄存器如下表所示。

表 18-2: SAADC 基址

外设名称	基址
TEMP	0x4000C000

表 18-3: PPI 相关寄存器

序号	寄存器名	偏移地址	功能描述
任务寄存器			
1	START	0x000	启动温度测量。
2	STOP	0x004	停止温度测量。
事件寄存器			
1	DATARDY	0x100	温度测量完成，测量数据就绪。
通用寄存器			
1	INTENSET	0x304	使能中断
2	INTENCLR	0x308	关闭中断
3	TEMP	0x508	温度测量值，步进 0.25°C。
5	A0	0x520	第 1 段线性函数斜率
6	A1	0x524	第 2 段线性函数斜率
7	A2	0x528	第 3 段线性函数斜率
8	A3	0x52C	第 4 段线性函数斜率
9	A4	0x530	第 5 段线性函数斜率
10	A5	0x534	第 6 段线性函数斜率
11	B0	0x540	第 1 段线性函数 y 轴截距
12	B1	0x544	第 2 段线性函数 y 轴截距
13	B2	0x548	第 3 段线性函数 y 轴截距
14	B3	0x54C	第 4 段线性函数 y 轴截距
15	B4	0x550	第 5 段线性函数 y 轴截距
16	B5	0x554	第 6 段线性函数 y 轴截距
17	T0	0x560	第 1 段线性函数端点
18	T1	0x564	第 2 段线性函数端点
19	T2	0x568	第 3 段线性函数端点
20	T3	0x56C	第 4 段线性函数端点
21	T4	0x570	第 5 段线性函数端点

■ INTENSET: 中断使能寄存器

INTENSET 寄存器用于使能中断。位的值写为 1 时使能对应的中断，写入 0 无效。注意 INTENSET 寄存器只能用来使能中断。

表 18-4: INTENSET 寄存器

位	Field	RW	复位值	描述
位 0	DATARDY	读/写	0	使能 DATARDY 事件产生中断。 写, 1: 使能。 读, 0: 已禁止。 读, 1: 已使能。

■ INTENCLR: 中断禁止寄存器

INTENCLR 寄存器用于禁止中断。位的值写为 1 时禁止对应的中断，写入 0 无效。注意 INTENCLR 寄存器只能用来禁止中断。

表 18-5: INTENCLR 寄存器

位	Field	RW	复位值	描述
位 0	DATARDY	读/写	0	禁止 DATARDY 事件产生中断。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。

■ TEMP: 温度测量值寄存器

TEMP 寄存器保存温度测量结果，测量完成后，可从该寄存器读出测量值，需要注意的是：因为片内温度传感器的测量分辨率是 0.25°C，所以测量结果要乘以 0.25（即除以 4）才是实际的温度值。

表 18-6: TEMP 寄存器

位	Field	RW	复位值	描述
位 0~位 31	TEMP	读/写	0	温度值, 步进 0.25°C。

■ An 寄存器, n=0~5: 斜率寄存器

表 18-7: TEMP 寄存器

位	Field	RW	复位值	描述
位 11~位 0	An, n=0~5	读/写	0x00000320	第(n+1)段线性函数斜率。

4. 软件设计

4.1. 库函数的应用

片内温度传感器使用起来比较简单，除了 DATARDY 事件中断之外，没有其他参数需要配置，并且由于片内温度传感器测量时间很短，所以使用时一般都不会开启 DATARDY 事件中断，而是用查询的方式进行温度测量。

片内温度传感器使用查询的方式测量温度的步骤如下图所示，简单的几步即可读取温度值。首先初始化温度传感器，然后触发 START 任务启动温度传感器，之后通过查询 DATARDY 事件是否置位判断操作是否完成，DATARDY 事件置位后即温度测量完成并且测量值已保存到 TEMP 寄存器，这时，读取 TEMP 寄存器的值即可获取温度测量值。测量值读取之后，因为温度传感器的测量分辨率是 0.25，所以测量结果要乘以 0.25（即除以 4）才是实际的温度值。

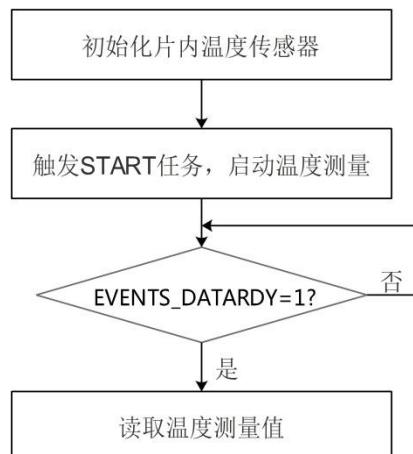


图 18-1: PPI 的应用步骤

1. 初始化片内温度传感器

片内温度传感器使用 `nrf_temp_init()` 函数完成初始化，函数中配置的是片内温度传感器的隐藏寄存器，该函数原型如下表所示。

表 18-8: `void nrf_temp_init()` 函数

函数原型	<code>static __INLINE void nrf_temp_init(void)</code>
函数功能	初始化片内温度传感器。注意：片内温度传感器在初始化的时候是没有参数要配置的，我们使用温度传感器时，只需要调用该函数初始化即可。。
参 数	无。
返回值	无。

■ 初始化 PPI 程序模块示例：

```
nrf_temp_init();
```

2. 启动温度测量

片内温度传感器通过触发 START 任务启动测量，触发 START 任务的代码是：

```
NRF_TEMP->TASKS_START = 1;
```

3. 等待测量完成

温度测量完成后，DATARDY 事件会置位，程序中通过查询 DATARDY 事件是否置位即判断温度测量是否完成。

4. 读取温度测量值

温度测量完成后，测量值通过 nrf_temp_read() 函数读出，读出的测量值乘以 4 即为温度值。nrf_temp_read() 函数原型如下表所示。

表 18-9: nrf_temp_read() 函数

函数原型	<code>static __INLINE int32_t nrf_temp_read(void)</code>
函数功能	读取温度测量值。
参数	无。
返回值	温度测量值，注意：因为片内温度传感器的测量分辨率是 0.25°C，所以测量结果要乘以 0.25（即除以 4）才是实际的温度值。

4.2. 温度测量实验

本实验在“实验 10-1：串口数据收发”的基础上修改。初始化温度传感器后，每 500 毫秒启动一次温度测量，测量完成后读取测量值，并将测量值计算成实际温度值后通过串口打印出温度值。

❖ 注：本节对应的实验源码是：“实验 18-1：片内温度传感器温度测量”。

4.2.1. 头文件引用和路径设置

片内温度传感器的驱动程序是以头文件的形式提供的，所以我们只需引用头文件和设置路径即可。

5. 需要引用的头文件

因为在“main.c”文件中使用了片内温度传感器程序模块，所以“main.c”文件需要引用下面的头文件。

```
#include "nrf_temp.h"
```

6. 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

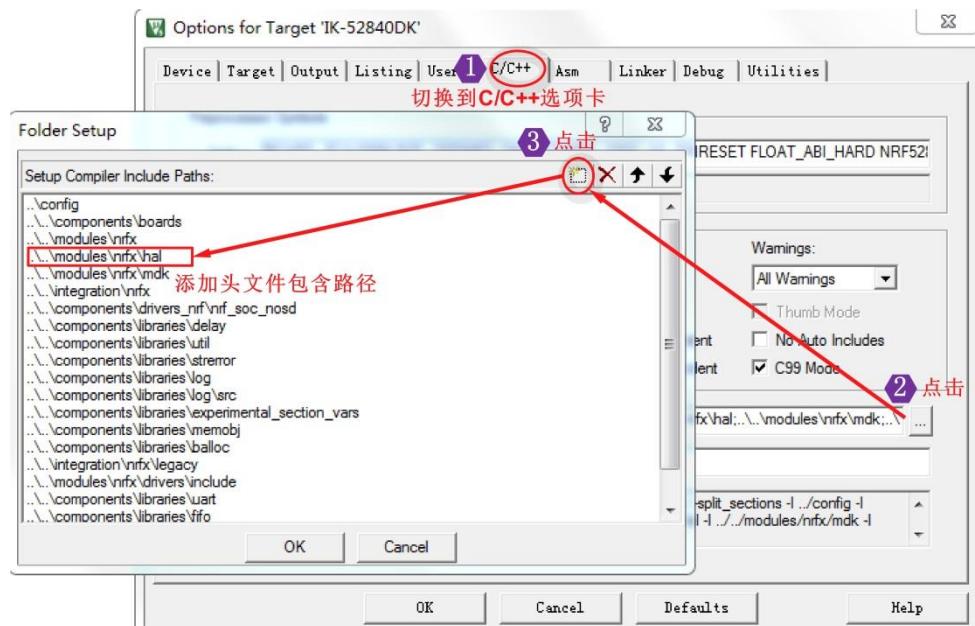


图 18-2: 添加头文件包含路径

片内温度传感器需要添加的头文件路径如下表:

表 18-10: 头文件包含路径

序号	路径
1	..\\modules\\nrfx\\hal

4.2.2. 代码编写

按照前文描述的步骤，我们使用查询的方式读取温度测量值，并将温度测量值计算为温度，代码清单如下。

代码清单：查询的方式测量温度

```

1. int main(void)
2. {
3.     int32_t volatile temp;
4.     // 初始化开发板上的 4 个 LED，即将驱动 LED 的 GPIO 配置为输出,
5.     bsp_board_init(BSP_INIT_LEDS);
6.
7.     uart_config(); // 初始化 uart
8.     // 初始化温度传感器
9.     nrf_temp_init();
10.    while(true)
11.    {
12.        // 触发片内温度传感器 Start 任务，启动温度测量
13.        NRF_TEMP->TASKS_START = 1;
14.
15.        // 等待测量完成

```

```

16.     while (NRF_TEMP->EVENTS_DATARDY == 0){}
17.     //清零事件
18.     NRF_TEMP->EVENTS_DATARDY = 0;
19.     //读取温度值
20.     temp = (nrf_temp_read() / 4);
21.     //温度传感器模块的模拟电路在检测完成后不会自动关闭，需要手动关闭以节省功耗
22.     NRF_TEMP->TASKS_STOP = 1; // 触发 Stop 任务关闭片内温度传感器
23.
24.     printf("temperature: %d\r\n", (int)temp); //串口打印温度值
25.     nrf_delay_ms(500); //延时 500ms，方便观察实验现象
26.     nrf_gpio_pin_toggle(LED_1); //指示灯 D1 状态翻转指示程序运行
27. }
28.

```

4.2.3. 硬件连接

本实验需要使用 P0.13 驱动 LED 指示灯 D1，P0.06 和 P0.08 作为串口通讯引脚，按照下图所示短接跳线帽。

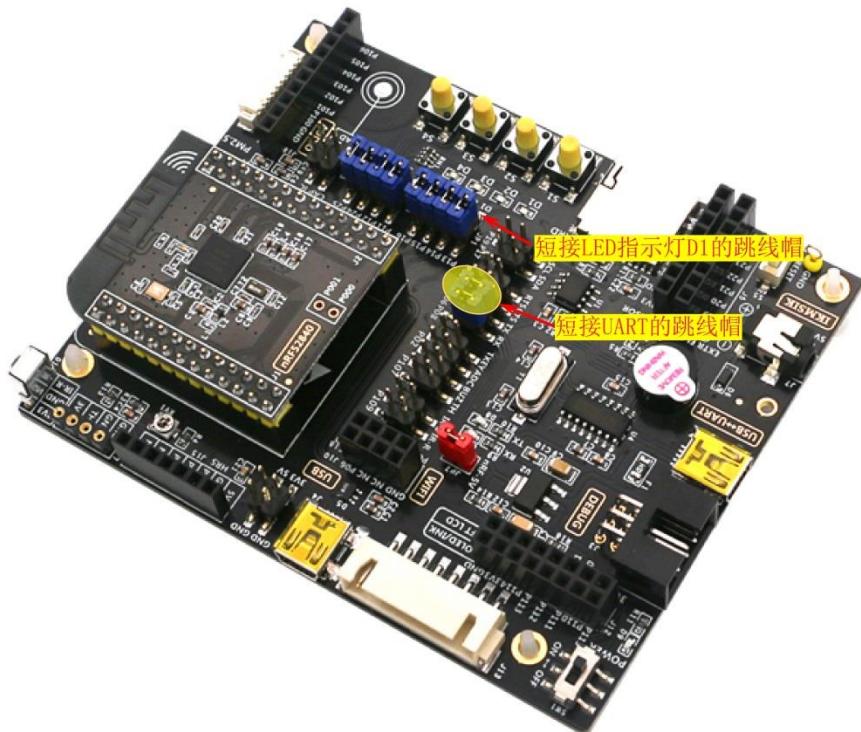


图 18-3：开发板跳线帽短接

4.2.4. 实验步骤

- 解压“…\3: 开发指南（上册）配套试验源码\”目录下的压缩文件“实验 18-1：片内温度传感器温度测量”，将解压后得到的文件夹“temp_sensor”拷贝到合适的目录，如“D\ nRF52840”。
- 启动 MDK5.23。
- 在 MDK5 中执行“Project→Open Project”打开“…\temp_sensor\project\mdk5”目录下

的工程“temp_sensor.uvproj”。

4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，每 500ms 测量一次温度，并将测量值计算为温度值后通过串口输出，用手指按着 nRF52840 芯片，可以观察到测量温度的变化。

```
temperature: 15
temperature: 15
temperature: 15
temperature: 15
temperature: 15
temperature: 15
temperature: 16
temperature: 16
temperature: 16
temperature: 17
temperature: 17
temperature: 17
temperature: 17
temperature: 17
temperature: 17
```

图 18-4：串口调试助手接收的温度值

第十九章：随机数发生器

1. 学习目的

1. 了解 nRF52840 的随机数发生器的原理。
2. 掌握基于库函数编程读取随机数的方法。

2. RNG 原理

nRF52840 片内集成一个随机数发生器 RNG(Random number generator)，采用热噪声生成的真正的非确定性数据流，并且无需种子值，其原理框图如下：

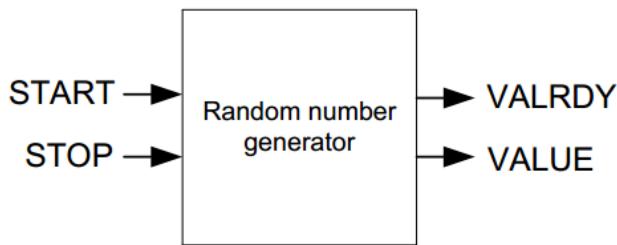


图 19-1：随机数发生器

触发 START 任务启动随机数发生器，触发 STOP 任务停止随机数发生器，随机数发生器启动后，新随机数连续生成并在准备就绪时写入 VALUE 寄存器以备读出，每个新的随机数写入到 VALUE 寄存器后会产生 VALRDY 事件。这意味着，在 VALRDY 事件产生之后，CPU 在下一个 VALRDY 事件产生之前若没有读出随机数，随机数将会被新的数值覆盖。

随机数发生器采用校正算法来移除数据流中趋于 1 或 0 的数据，然后经过校正的数据顺序进入 8 位寄存器以备读出。通过配置 CONFIG 寄存器可以使能校正算法，使能校正算法使能后会减慢随机数生成的速度，但是可以确保随机值的均匀分布。

RNG 产生一字节随机数所需的时间是不可预知的，并且一个字节和下一个字节之间的时间也是会变化的，当启用偏置校正时，尤其如此。

随机数发生器电气规范如下表所示：

表 19-1：RNG 电气规范

符号	描述	最小	典型值	最大值	单位
I_{RNG}	运行电流，CPU 睡眠。	—	500	—	μs
$T_{RNG,START}$	从触发 START 任务到开始生成随机数之间的时间。这是启动信号的一次延迟，在样本之间不适用。	—	128	—	μs
$T_{RNG,RAW}$	无校正时每个字节的运行时间，不能保证 0 和 1 的	—	30	—	μs

	均匀分布。				
T _{RNG,BC}	有校正时每个字节的运行时间，保证 0 和 1 的均匀分布，不能保证生成字节所需的时间。	—	120	—	μs

3. RNG 寄存器

随机数发生器 RNG 的寄存器基址是 0x4000D000，包含的寄存器如下表所示。

表 19-2: RNG 基址

外设名称	地址
RNG	0x4000D000

表 19-3: PPI 相关寄存器

序号	寄存器名	偏移地址	功能描述
任务寄存器			
1	TASKS_START	0x000	启动随机数发生器
2	TASKS_STOP	0x004	停止随机数发生器
事件寄存器			
1	EVENTS_VALRDY	0x100	数据就绪事件。新的随机数写入到 VALUE 后产生该事件。
快捷方式寄存器			
1	SHORTS	0x200	快捷功能寄存器
通用寄存器			
1	INTENSET	0x304	使能中断
2	INTENCLR	0x308	禁止中断
3	CONFIG	0x504	配置寄存器
4	VALUE	0x508	随机数输出寄存器

■ SHORTS: 快捷方式寄存器

表 19-4: SHORTS 寄存器

位	Field	RW	复位值	描述
位 0	VALRDY_STOP	读/写	0	VALRDY 事件和 STOP 任务之间的快捷方

		式 0: 禁止快捷方式。 1: 使能快捷方式。
--	--	-------------------------------

■ INTENSET: 中断使能寄存器

INTENSET 寄存器用于使能中断。位的值写为 1 时使能对应的中断，写入 0 无效。注意 INTENSET 寄存器只能用来使能中断。

表 19-5: INTENSET 寄存器

位	Field	RW	复位值	描述
位 0	VALRDY	读/写	0	使能 VALRDY 事件产生中断。 写, 1: 使能。 读, 0: 已禁止。 读, 1: 已使能。

■ INTENCLR: 中断禁止寄存器

INTENCLR 寄存器用于禁止中断。位的值写为 1 时禁止对应的中断，写入 0 无效。注意 INTENCLR 寄存器只能用来禁止中断。

表 19-6: INTENCLR 寄存器

位	Field	RW	复位值	描述
位 0	VALRDY	读/写	0	禁止 VALRDY 事件产生中断。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。

■ CONFIG: 配置寄存器

INTENCLR 寄存器用于禁止中断。位的值写为 1 时禁止对应的中断，写入 0 无效。注意 INTENCLR 寄存器只能用来禁止中断。

表 19-7: CONFIG 寄存器

位	Field	RW	复位值	描述
位 0	VALRDY	读/写	0	偏差修正。 1: 使能。 0: 禁止。

■ VALUE: 随机数值寄存器

VALUE 寄存器保存随机数发生器生成的随机数。

表 19-8: TEMP 寄存器

位	Field	RW	复位值	描述
位 7~位 0	VALUE	读/写	0	随机数发生器生成的随机数。

4. 软件设计

4.1. 库函数的应用

随机数发生器的应用步骤如下图所示，应用程序首先初始化 RNG，初始化函数中会配置 RNG 参数并启动 RNG，生成的随机数存储在驱动程序定义的缓存池中。应用程序需要读取随机数时通过查询缓存池获取已生成的随机数的数量，之后从缓存池中读取随机数数据。

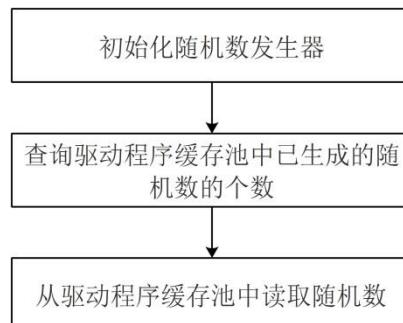


图 19-2: 随机数发生器的应用步骤

1. 初始化随机数发生器

随机数发生器使用 nrf_drv_rng_init() 函数完成初始化，该函数原型如下表所示。

表 19-9: void nrf_drv_rng_init() 函数

函数原型	ret_code_t nrf_drv_rng_init(nrf_drv_rng_config_t const * p_config)
函数功能	初始化随机数发生器。
参数	[in] p_config: 指向初始化配置结构体。
返回值	NRF_SUCCESS: 初始化成功。 NRF_ERROR_MODULE_ALREADY_INITIALIZED: RNG 程序模块已经初始化。

nrf_drv_rng_init() 函数中使用默认参数配置随机数发生器，配置项是：是否使用校正算法和中断优先级，另外，驱动程序中定义了用于存放生成的随机数的缓存池 m_rand_pool。配置项和缓存池的大小定义在工程配置向导 “sdk_config.h” 中，如我们需要修改这些参数时，无需更改 RNG 的驱动程序，在 “sdk_config.h” 中修改即可。

代码清单：RNG 配置

1. //1: 使能校正算法, 0: 禁止校正算法
2. #ifndef RNG_CONFIG_ERROR_CORRECTION
3. #define RNG_CONFIG_ERROR_CORRECTION 1

```

4. #endif
5.
6. //RNG 驱动程序内存池大小
7. #ifndef RNG_CONFIG_POOL_SIZE
8. #define RNG_CONFIG_POOL_SIZE 64
9. #endif
10.
11. //中断优先级，数值越大，优先级越低
12. // <i> Priorities 0,2 (nRF51) and 0,1,4,5 (nRF52) are reserved for SoftDevice
13. // <0=> 0 (highest)
14. // <1=> 1
15. // <2=> 2
16. // <3=> 3
17. // <4=> 4
18. // <5=> 5
19. // <6=> 6
20. // <7=> 7
21.
22. #ifndef RNG_CONFIG_IRQ_PRIORITY
23. #define RNG_CONFIG_IRQ_PRIORITY 7
24. #endif

```

RNG 驱动程序缓存池可以定义为溢出和不溢出，驱动程序中默认配置的是溢出。

- NRF_QUEUE_MODE_OVERFLOW: 缓存池存满后，新的随机数会覆盖之前存储的数据。
- NRF_QUEUE_MODE_NO_OVERFLOW: 缓存池存满后，不再接受新的随机数。

如果需要缓存池存满后不溢出，在“nrf_drv_rng.c”文件中将 NRF_QUEUE_DEF 的参数 NRF_QUEUE_MODE_OVERFLOW 改为 NRF_QUEUE_MODE_NO_OVERFLOW。

```
NRF_QUEUE_DEF(uint8_t, m_rand_pool,
              RNG_CONFIG_POOL_SIZE, NRF_QUEUE_MODE_NO_OVERFLOW);
```

2. 查询缓存池中生成的随机数数量

应用程序读取随机数的过程是先查询缓存池中随机数的数量，然后再读取，查询的函数是 nrf_drv_rng_bytes_available()，该函数通过输出参数返回缓存池中随机数的数量。

表 19-10: nrf_drv_rng_bytes_available()函数

函数原型	void nrf_drv_rng_bytes_available (uint8_t * p_bytes_available)
函数功能	获取当前可用的随机字节数。
参 数	[out] p_bytes_available: 当前可用的随机字节数。

返回值	无。
-----	----

3. 读取随机数

读取随机数是从缓存池中拷贝指定长度的数据到指定的缓存，读取随机数的函数是 nrf_drv_rng_rand ()，调用该函数时，需要提供一个缓存用来存放从 RNG 驱动程序缓存池中读取的随机数。

表 19-11: nrf_drv_rng_rand ()函数

函数原型	ret_code_t nrf_drv_rng_rand (uint8_t * p_buff,uint8_t length)
函数功能	初始化随机数发生器。
参数	[out] p_buff : 指向保存读取的随机数的缓存。 [out] length : 读取的随机数的字节数。
返回值	NRF_SUCCESS: 读取的随机数成功写入到 p_buff 指向的缓存。 NRF_ERROR_NOT_FOUND: 随机数成功写入到 p_buff 指向的缓存，因为没有足够的空间。

4.2. 随机数生成实验

本实验在“实验 10-1：串口数据收发”的基础上修改。初始化随机数发生器后，每 500 毫秒从缓存池中读取一次随机数，每次读取 16 个字节，读出的随机数通过串口输出。

◆ 注：本节对应的实验源码是：“实验 18-1：随机数发生器”。

4.2.1. 添加需要的文件

随机数发生器需要加入的文件如下表所示。

表 19-12: 随机数发生器需要加入的文件

文件名	SDK 中的目录	描述
nrf_drv_rng.c	..\integration\nrfx\legacy	旧版本 RNG 驱动文件。
nrfx_rng.c	..\modules\nrfx\drivers\src	新版本 RNG 驱动文件。
nrf_queue.c	..\components\libraries\queue	队列驱动文件。

4.2.2. 头文件引用和路径设置

1. 需要引用的头文件

因为在“main.c”文件中使用了随机数发生器程序模块，所以“main.c”文件需要引用下面的头文件。

```
#include "nrf_drv_rng.h"
```

2. 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

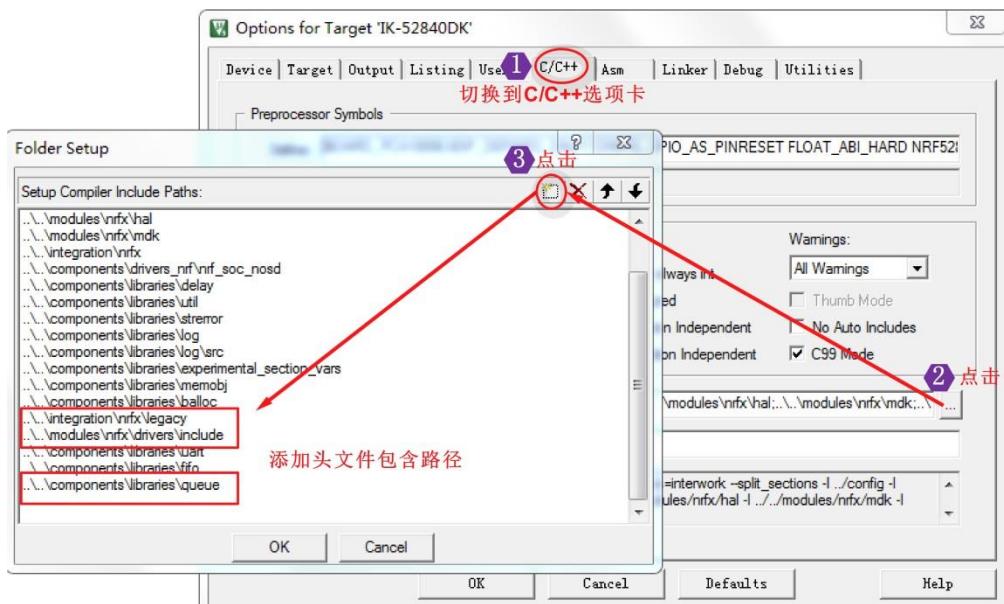


图 19-3: 添加头文件包含路径

随机数发生器需要添加的头文件路径如下表：

表 19-13: 头文件包含路径

序号	路径
1	..\..\modules\nrfx\drivers\include
2	..\..\integration\nrfx\legacy
3	..\..\components\libraries\queue

4.2.3. 工程配置

打开“ `sdk_config.h`”文件，加入 RNG 所需的配置，如下图所示，编辑内容时切换到“Text Editor”进行编辑（编辑的内容参考实验的源码，因此代码很长，此处不贴出代码，参见“ `sdk_config.h`”文件的(144~257 和 708~720)行），编辑完成后，切换到“Configuration Wizard”，即可观察到配置的具体项目：

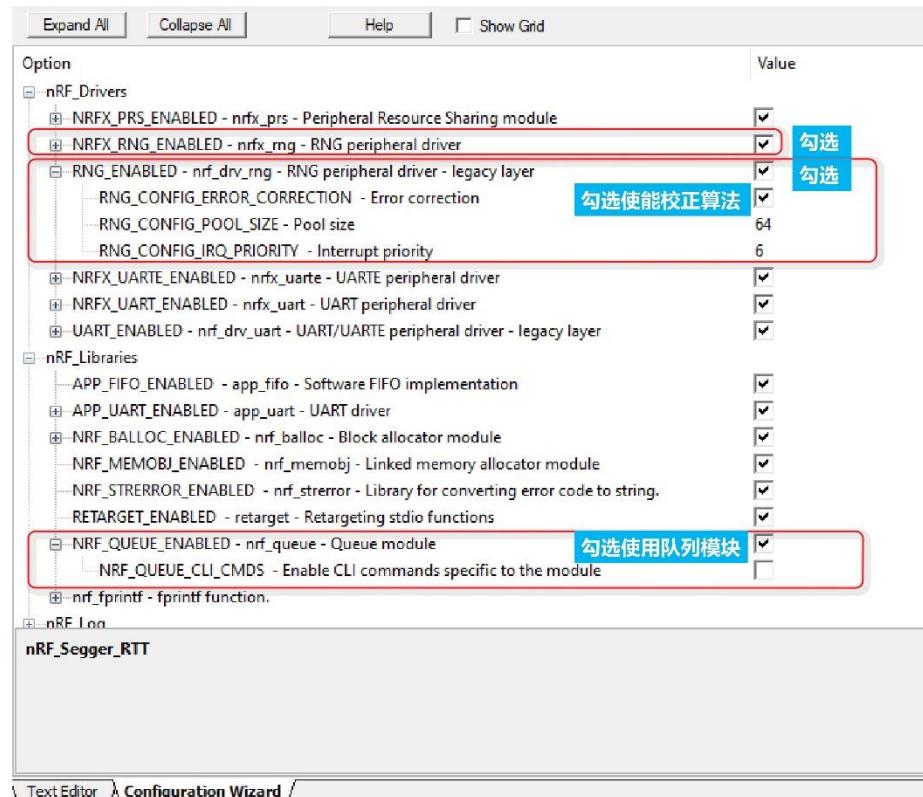


图 19-4：工程配置

4.2.4. 代码编写

按照前文描述的步骤，我们在“ `sdk_config.h`”文件中配置 RNG 驱动程序缓存池大小为 64，即 RNG 驱动程序可存储 64 个随机数。应用程序中定义一个大小为 16 个字节的缓存数组，用于存放读取的随机数，所以应用程序每次读取的随机数不能超过 16 个。应用程序中每 500ms 查询一次缓存池中可用的随机数数量，如可用的随机数超过 16 个，则读出 16 个，若可用的随机数不超过 16 个，则全部读出，并通过串口输出读出的随机数和缓存池可用的随机数数量。

代码清单：生成和读取随机数

```

1. #define RANDOM_BUFF_SIZE 16           //保存读出的随机数的缓存数组大小
2.
3. int main(void)
4. {
5.     uint32_t err_code;
6.     //保存缓存池中可用的随机数的数量
7.     uint8_t available;
8.     //初始化开发板上的 4 个 LED，即将驱动 LED 的 GPIO 配置为输出,
9.     bsp_board_init(BSP_INIT_LEDS);
10.    //串口初始化
11.    uart_config();
12.    //初始化 RNG
13.    err_code = nrf_drv_rng_init(NULL);

```

```
14.     APP_ERROR_CHECK(err_code);
15.     while(true)
16.     {
17.         uint8_t p_buff[RANDOM_BUFF_SIZE];
18.         //查询 RNG 驱动程序缓存池中可用的随机数数量
19.         nrf_drv_rng_bytes_available(&available);
20.
21.         printf("number of random numbers is: %d\r\n",available);
22.         //读取的长度取 RANDOM_BUFF_SIZE 和 available 中小的值。
23.         //缓存池中随机数不够 16 个时全部读出，超过 16 个时取出 16 个
24.         uint8_t length = MIN(RANDOM_BUFF_SIZE, available);
25.         //从缓存池中读出随机数，读出的随机数存储到 p_buff
26.         err_code = nrf_drv_rng_rand(p_buff, length);
27.         APP_ERROR_CHECK(err_code);
28.
29.         //串口输出读取的随机数
30.         printf("Random numbers:");
31.         for(uint8_t i = 0; i < length; i++)
32.         {
33.             printf(" %3d", (int)p_buff[i]);
34.         }
35.         printf("\r\n");           //回车换行
36.         nrf_delay_ms(500);       //延时，方便观察数据
37.         nrf_gpio_pin_toggle(LED_1); //指示灯 D1 指示程序运行
38.     }
39. }
```

4.2.5. 硬件连接

本实验需要使用 P0.13 驱动 LED 指示灯 D1，P0.06 和 P0.08 作为串口通讯引脚，按照下图所示短接跳线帽。

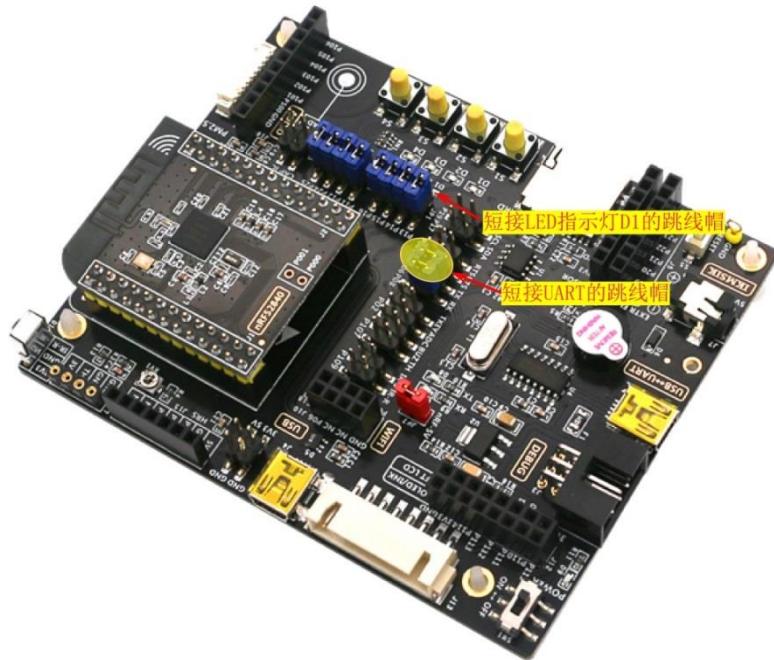


图 19-5：开发板跳线帽短接

4.2.6. 实验步骤

1. 解压“…\3: 教程（上册）配套试验源码\”目录下的压缩文件“实验 19-1：随机数发生器”，将解压后得到的文件夹“rng”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\rng\project\mdk5”目录下的工程“rng.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，每 500ms 从 RNG 驱动程序缓存池中读取一次随机数，并通过串口输出，串口调试助手中可以观察到读出的随机数。

```
Random numbers: 62 170 3 122 229 72 222 138 130 199 55 150 48 66 41 186
number of random numbers is: 64
Random numbers: 150 66 162 21 207 176 217 98 31 42 201 225 184 25 70 137
number of random numbers is: 64
Random numbers: 122 78 236 70 125 40 53 207 254 157 220 24 211 30 32 88
number of random numbers is: 64
Random numbers: 80 61 222 121 245 180 95 45 198 124 148 252 174 183 13 146
number of random numbers is: 64
Random numbers: 133 39 201 85 136 254 241 96 190 158 57 200 109 120 229 116
number of random numbers is: 64
Random numbers: 115 173 43 98 40 60 13 166 239 217 196 105 23 119 99 181
number of random numbers is: 64
Random numbers: 178 172 64 115 81 144 200 89 74 136 67 83 45 226 18 218
number of random numbers is: 64
Random numbers: 34 155 149 98 135 109 203 136 140 187 252 87 55 46 49 137
number of random numbers is: 64
Random numbers: 2 178 139 46 216 9 130 117 166 191 38 132 114 125 167 168
```

图 19-6：串口调试助手接收到的 RNG 数据

第二十章：实时计数器 RTC

1. 学习目的

1. 了解 nRF52840 的 RTC 的原理以及实时时钟和实时计数器的区别。
2. 掌握 RTC 的 TICK 事件产生的条件和应用。
3. 掌握 RTC 的 4 个通道的比较事件的应用。

2. RTC 原理

看到 RTC，我们首先想到的用来记录时间的实时时钟，但是要注意，nRF52840 的 RTC 并不是指实时时钟，我们不能通过 nRF52840 的 RTC 直接获取时间信息，nRF52840 的 RTC 指的是对振荡源脉冲进行计数的实时计数器，实时时钟和实时计数器区别如下。

- **Real Time Clock:** 实时时钟，具备日期寄存器(通过程序可从日期寄存器中读出年、月、日、时、分、秒的值，直接获取日期数据)，可用电池供电，即使主机电源断开，也能正常运行。
- **Real Time Counter:** 实时计数器。对振荡源(nRF52840 的 RTC 时钟源是 32.768KHz)的脉冲计数，每个脉冲计数值加 1，它没有日期寄存器，所以不能从实时计数器直接获取时间。

2.1. 原理框图和时钟源

nRF52840 片内集成了 3 个 RTC，如下表所示：

表 20-1: nRF52840 RTC

RTC	基址	CC 寄存器	描述
RTC0	0x4000B000	CC[0..2]	24 位实时计数器
RTC1	0x40011000	CC[0..3]	24 位实时计数器
RTC2	0x40024000	CC[0..3]	24 位实时计数器

3 个 RTC 为 24 位，使用 LFCLK 低频时钟，并带有 12 位分频器，可产生 TICK、比较和溢出事件。RTC 原理框图如下图所示。

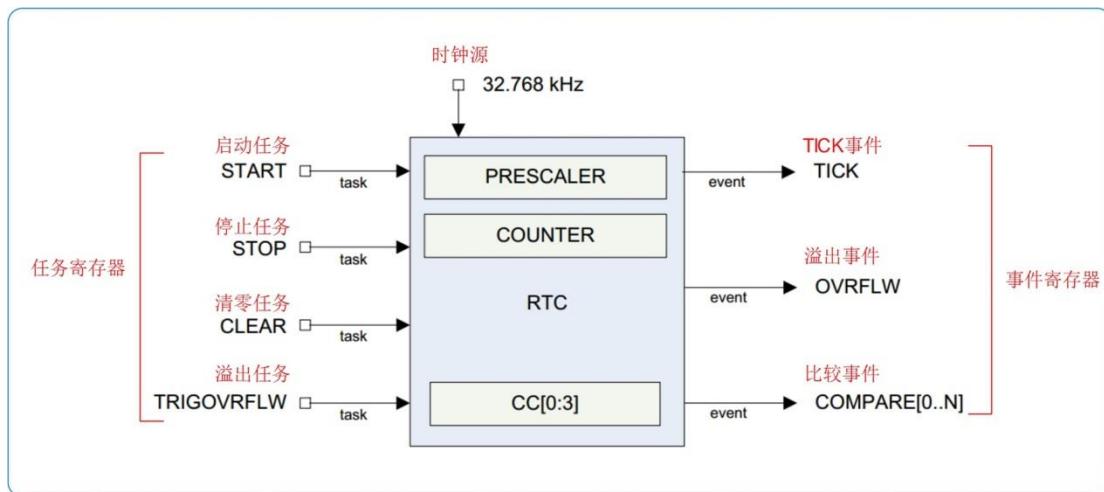


图 20-1: RTC 原理框图

■ 时钟源：

RTC 使用 LFCLK 作为时钟源，计数器(COUNTER)的分辨率为 30.517us，所以当 HFCLK 关闭和 PCLK16M 不可用时，RTC 仍可运行。

■ 计数器递增频率的计算

$$f_{RTC}[\text{KHz}] = \frac{32.768}{\text{PRESCALER} + 1}$$

由上式可以看出，设置递增频率也就是设置相应的 PRESCALER，如设置递增频率为 8Hz。那么，PRESCALER 值如下：

$$\text{PRESCALER} = \text{round(四舍五入)}(32.768 \text{ kHz} / 8 \text{ Hz}) - 1 = 4095$$

此时，递增周期是：125ms。

RTC 计数器分辨率和溢出时间如下表所示：

表 20-2: RTC 溢出分辨率

分频系数	计数器分辨率	溢出时间
0	30.517 μs	512 秒
2^8-1	7812.5 μs	131072 秒
$2^{12}-1$	125ms	582.542 小时

2.2. 计数器

计数器在每次计数时加 1，当我们把 PRESCALER(分频系数)设置为 0x00 时，计数器在 LFCLK(低频时钟，32.768KHz)的每个时钟周期加 1。

如果使能了 TICK 事件，那么计数器每次增加的时候会产生一次 TICK 事件，当然，如不需要 TICK 事件，也可以将其关闭。

◆ 这里需要注意的是：TICK 事件是在计数器增加的时候产生的，而不是在每个时钟周期

产生的。下面两个时序图可以很清楚地说明这一点。

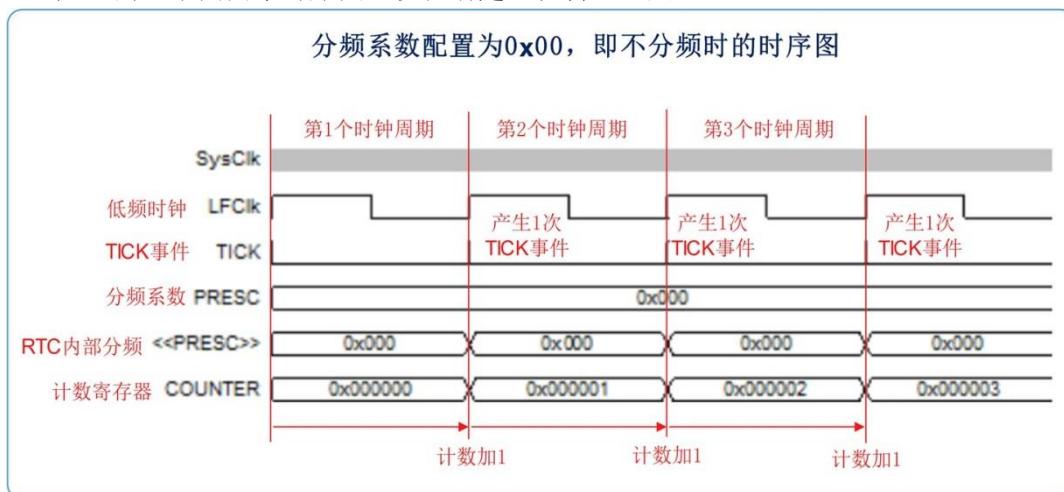


图 20-2: 分频系数为 0 时的时序图

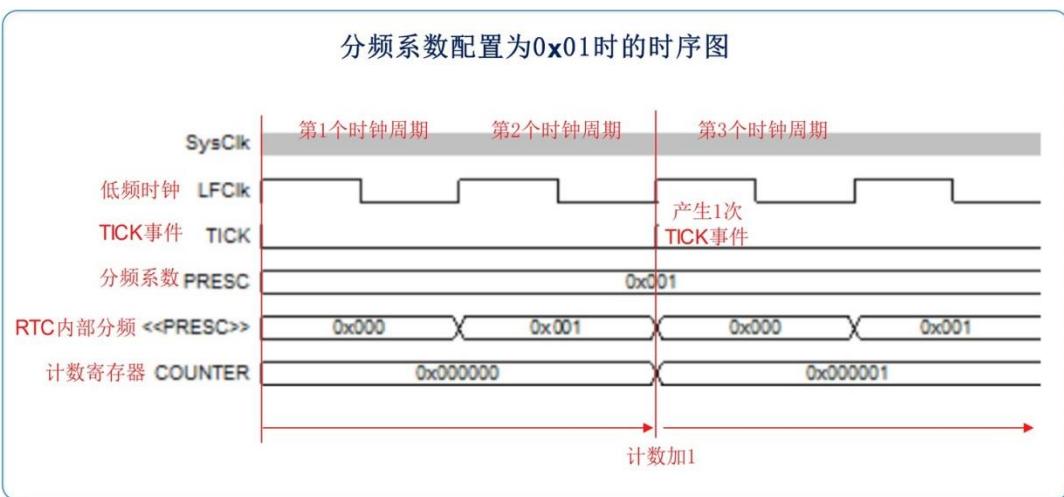


图 20-3: 分频系数为 1 时的时序图

2.3. RTC 溢出

nRF52840 的 RTC 可以通过触发任务来产生 OVRFLOW(溢出)事件：可以通过触发 TRIGOVRFLOW 任务将计数器的值设置为 0xFFFFF0，那么在 16 次计数后，计数器的值由 0xFFFFFFFF 变为 0x000000，这时会产生 OVRFLOW(溢出)事件。

❖ 注意：OVRFLOW(溢出)事件默认是禁止的。

2.4. TICK 事件

TICK 事件对于 nRF52840 来说非常重要，因为 BLE 协议栈是以 RTC0 的 TICK 事件来实现调度的。另外，SDK 中的 APP 定时器也是使用 RTC1 作为时基来实现的。

众所周知，RTOS 都需要一个时钟节拍(也称为时钟滴答或“心跳”)，通过时钟节拍划分时间片，从而实现调度。熟悉 ARM Cortex 的朋友都知道，ARM Cortex 有 SysTick，可以作为 RTOS 的“心跳”，但是 nRF52840 从低功耗的角度考虑，没有提供 SysTick，nRF52840 是通过 RTC 的 TICK 事件来实现时钟滴答的，这样就可以在低功耗的同时保持调度的运行。

- ◆ 注意：TICK 事件默认是禁止的。
- ◆ 特别注意：BLE 程序中，协议栈使用了 RTC0，APP 定时器使用了 RTC1，所以，BLE 程序中 RTC0 和 RTC1 是不能作为通用的外设使用的，RTC2 可以作为通用外设使用。

2.5. RTC 事件控制

RTC 的每个事件都是可以单独控制的，我们可以通过配置 EVTEN 寄存器来管理 RTC 事件：使能需要用到的事件、关闭用不到的事件，从而避免这些用不到的事件的产生、进一步降低 RTC 的电流消耗。如：程序中用不到 TICK 事件，那么就应该关闭掉该事件，否则，TICK 事件会频繁的产生，增加电流消耗，另外因为 TICK 事件会将系统从低功耗唤醒，减少了系统处于低功耗的时间，这也会导致电流消耗的增加。

2.6. RTC 比较事件

配置比较寄存器时，需要注意以下 RTC 比较事件的动作：

1. CC 寄存器值为 0 时触发 CLEAR 任务不会产生 COMPARE 事件

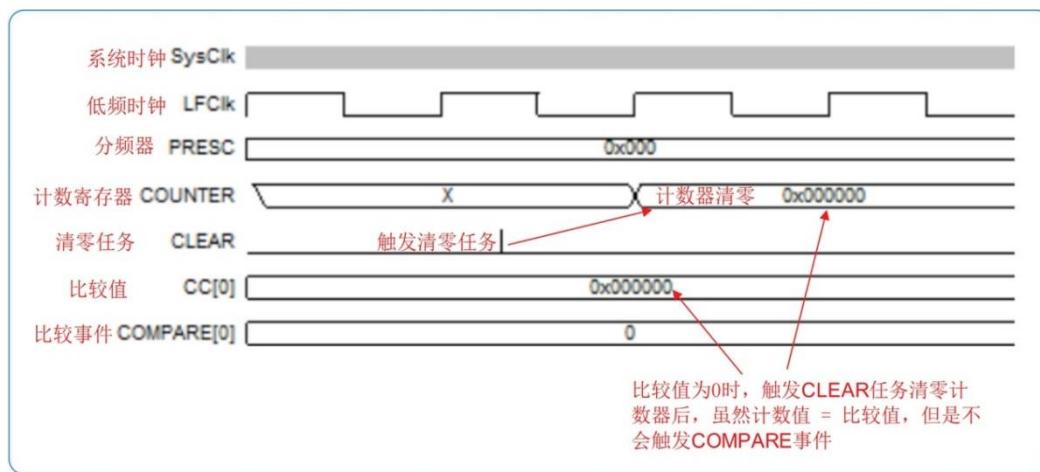


图 20-4: COMPARE 事件时序图-CLEAR 任务

2. 如果 CC 寄存器的值为 N 并且计数器 COUNTER 的值也是 N 时触发 START 任务不会产生 COMPARE 事件

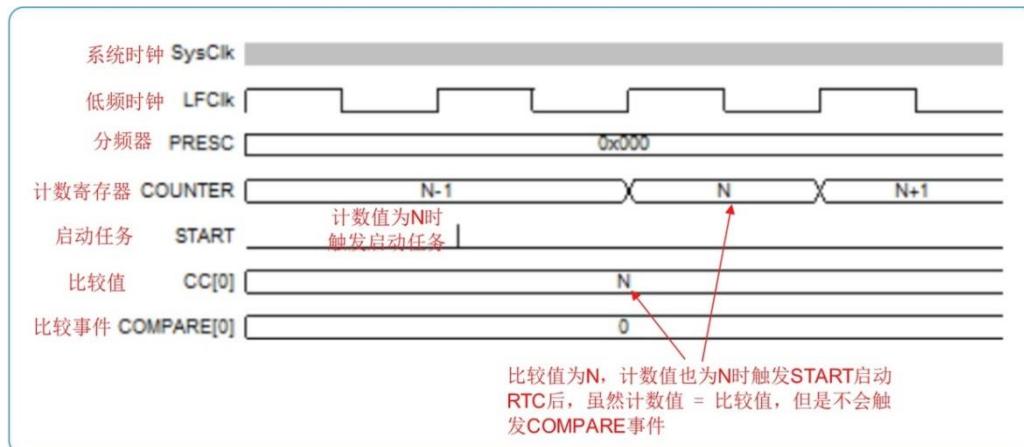


图 20-5: COMPARE 事件时序图-START 任务

3. CC 寄存器值为 N 时，当计数器的值由 N-1 递增为 N 时产生 COMPARE 事件

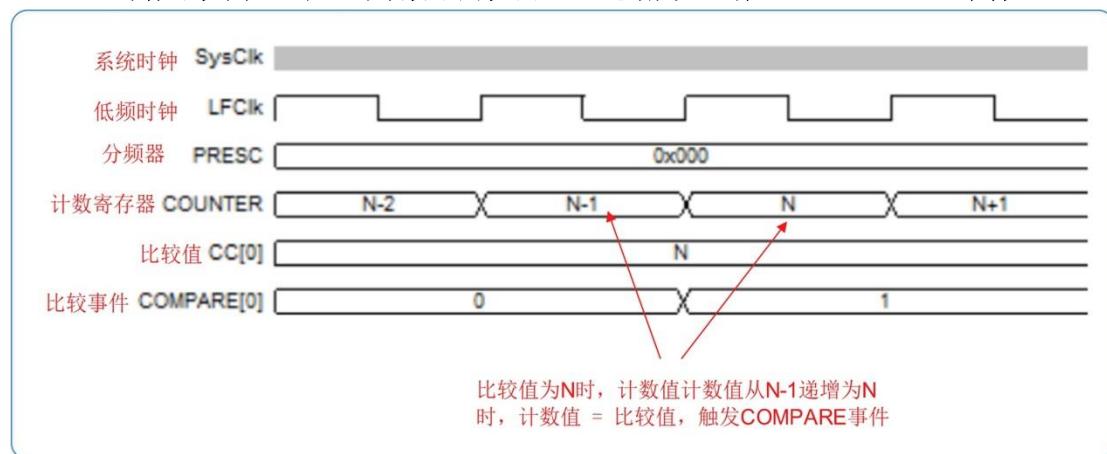


图 20-6: COMPARE 事件时序图- COMPARE

4. 如果计数器的值为 N，写入 N+2 到 CC 比较寄存器，可以确保当计数值递增到 N+2 时产生 COMPARE 事件

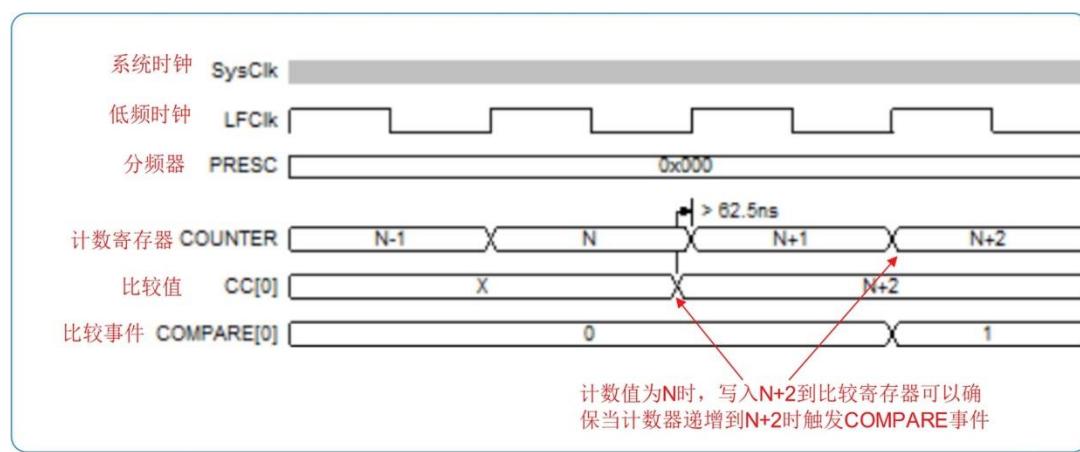


图 20-7: COMPARE 事件时序图- COMPARE_N+2

5. 如果计数器的值为 N，写入 N 或 N+1 到 CC 比较寄存器，可能不会产生 COMPARE 事件

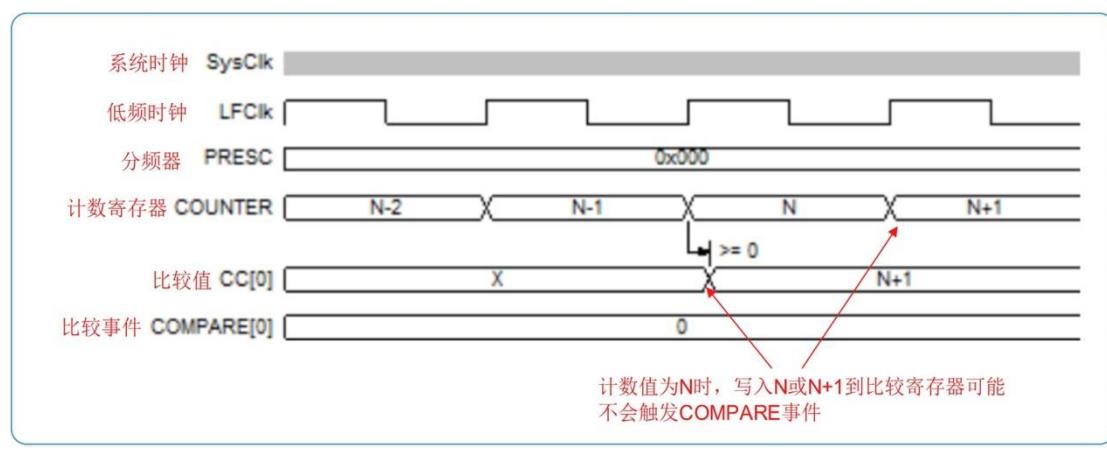


图 20-8: COMPARE 事件时序图- COMPARE_N+1

6. 如果计数器 COUNTER 的值为 N 并且当前 CC 寄存器的值是 N+1 或 N+2 时，写入新的比较值到 CC 寄存器，这时可能会以旧的比较值产生 COMPARE 事件，如果写入的新比较值大于 N+2，则不会出现这种情况。

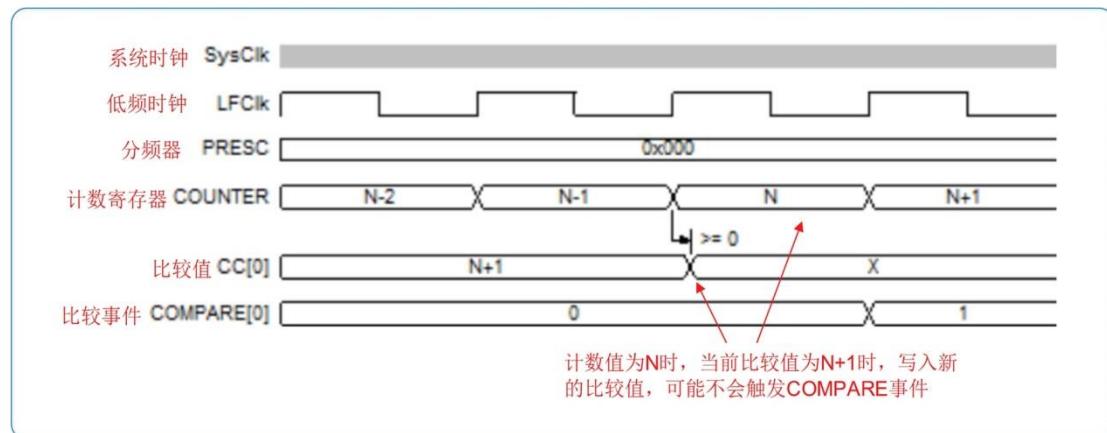


图 20-9: COMPARE 事件时序图- COMPARE_N-1

2.7. 读计数值(COUNTER 寄存器)

读计数器的值时，内部<<COUNTER>>（内部<<COUNTER>>是程序员不可见的）值会被采样，为了保证<<COUNTER>>能被安全可靠地采样(考虑到读的时候可能会发生 LFCLK 跳变)，CPU 和内核储存总线会通过拉低内核 PREADY 信号暂停 3 个时钟周期，另外读会占用额外两个时钟周期，所以读计数器的值寄存器总共会占用 5 个 PCLK16M 时钟周期，如下图所示：

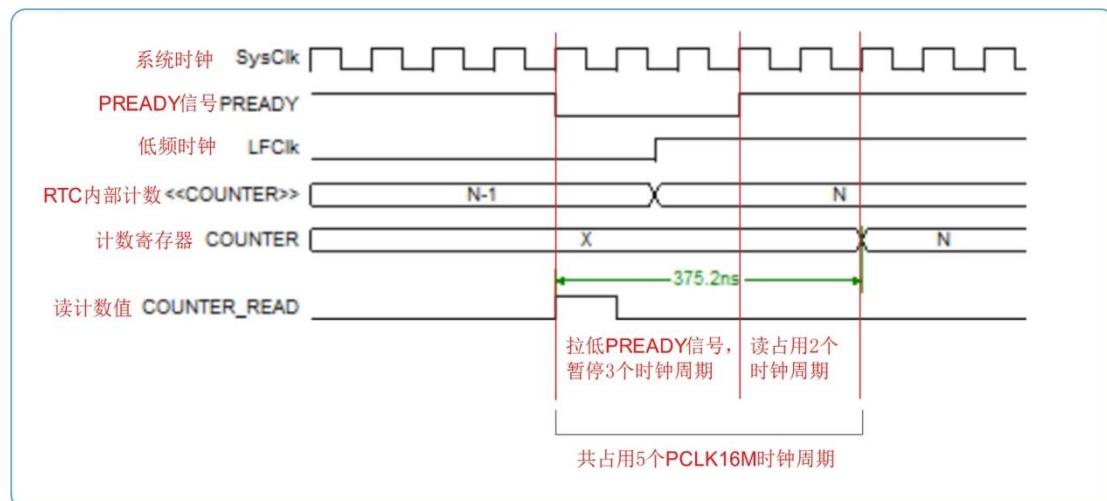


图 20-10: 读 COUNTER 寄存器时序图

3. RTC 寄存器

3 个 RTC 外设的寄存器基址如下表所示。

表 20-3: RNG 基址

外设名称	基址
------	----

RTC0	0x4000B000
RTC1	0x40011000
RTC2	0x40024000

每个 RTC 外设都包含如下表所示的寄存器。

表 20-4: PPI 相关寄存器

序号	寄存器名	偏移地址	功能描述
任务寄存器			
1	TASKS_START	0x000	启动 RTC 计数器。
2	TASKS_STOP	0x004	停止 RTC 计数器。
3	TASKS_CLEAR	0x008	清零 RTC 计数器。
4	TASKS_TRGOVRFLLW	0x00C	设置 COUNTER 的值为 0xFFFFF0。
事件寄存器			
1	EVENTS_TICK	0x100	计数器增量事件
2	EVENTS_OVRFLW	0x104	计数器溢出事件
3	EVENTS_COMPARE[0]	0x140	CC[0]比较匹配事件
4	EVENTS_COMPARE[1]	0x144	CC[1]比较匹配事件
5	EVENTS_COMPARE[2]	0x148	CC[2]比较匹配事件
6	EVENTS_COMPARE[3]	0x14C	CC[3]比较匹配事件
通用寄存器			
1	INTENSET	0x304	使能中断
2	INTENCLR	0x308	关闭中断
3	EVTEN	0x340	使能或关闭事件
4	EVTENSET	0x344	使能事件
5	EVTENCLR	0x348	关闭事件
6	COUNTER	0x504	当前计数值
7	PRESCALER	0x508	12 位分频系数，必须停止 RTC 后进行写操作。
8	CC[0]	0x540	比较寄存器 0
9	CC[1]	0x544	比较寄存器 1

10	CC[2]	0x548	比较寄存器 2
11	CC[3]	0x54C	比较寄存器 3

■ INTENSET：中断使能寄存器

INTENSET 寄存器用于使能中断。位的值写为 1 时使能对应的中断，写入 0 无效。注意 INTENSET 寄存器只能用来使能中断。

表 20-5: INTENSET 寄存器

位	Field	RW	复位值	描述
位 0	TICK	读/写	0	写入 1 时使能 TICK 事件产生中断，写入 0 无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。
位 1	OVRFLOW	读/写	0	写入 1 时使能 OVRFLOW 事件产生中断，写入 0 无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。
位 16	COMPARE0	读/写	0	写入 1 时使能 COMPARE0 事件产生中断，写入 0 无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。
位 17	COMPARE1	读/写	0	写入 1 时使能 COMPARE1 事件产生中断，写入 0 无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。
位 18	COMPARE2	读/写	0	写入 1 时使能 COMPARE2 事件产生中断，写入 0 无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。
位 19	COMPARE3	读/写	0	写入 1 时使能 COMPARE3 事件产生中断，写入 0 无效。

		写, 1: 使能。 读, 0: 已禁止。 读, 1: 已使能。
--	--	---------------------------------------

■ INTENCLR: 中断禁止寄存器

INTENCLR 寄存器用于禁止中断。位的值写为 1 时禁止对应的中断，写入 0 无效。注意 INTENCLR 寄存器只能用来禁止中断。

表 20-6: INTENCLR 寄存器

位	Field	RW	复位值	描述
位 0	TICK	读/写	0	写入 1 时禁止 TICK 事件产生中断，写入 0 无效。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。
位 1	OVRFLW	读/写	0	写入 1 时禁止 OVRFLW 事件产生中断，写入 0 无效。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。
位 16	COMPARE0	读/写	0	写入 1 时禁止 COMPARE0 事件产生中断，写入 0 无效。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。
位 17	COMPARE1	读/写	0	写入 1 时禁止 COMPARE1 事件产生中断，写入 0 无效。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。
位 18	COMPARE2	读/写	0	写入 1 时禁止 COMPARE2 事件产生中断，写入 0 无效。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。

位 19	COMPARE3	读/写 0	写入 1 时禁止 COMPARE3 事件产生中断，写入 0 无效。 写，1：禁止。 读，0：已禁止。 读，1：已使能。
------	----------	-------	--

■ EVTEN：事件使能/禁止寄存器

EVTEN 寄存器用于使能/禁止 RTC 的各个事件。位的值为 0 时禁止事件，位的值为 1 时使能事件。注意 EVTEN 寄存器既可以使能也可以禁止事件。

表 20-7: INTENSET 寄存器

位	Field	RW	复位值	描述
位 0	TICK	读/写 0	使能/禁止 TICK 事件。 0：禁止。 1：使能。	
位 1	OVRFLW	读/写 0	使能/禁止 OVRFLW 事件。 0：禁止。 1：使能。	
位 16	COMPARE0	读/写 0	使能/禁止 COMPARE0 事件。 0：禁止。 1：使能。	
位 17	COMPARE1	读/写 0	使能/禁止 COMPARE1 事件。 0：禁止。 1：使能。	
位 18	COMPARE2	读/写 0	使能/禁止 COMPARE2 事件。 0：禁止。 1：使能。	
位 19	COMPARE3	读/写 0	使能/禁止 COMPARE3 事件。 0：禁止。 1：使能。	

■ EVTENSET：事件使能寄存器

EVTENSET 寄存器用于使能 RTC 的各个事件。位的值写为 1 时使能事件，写入 0 无效。注意 EVTENSET 寄存器只能用来使能中断。

表 20-8: INTENSET 寄存器

位	Field	RW	复位值	描述
位 0	TICK	读/写	0	写入 1 时使能 TICK 事件，写入 0 无效。 写, 1: 使能。 读, 0: 已禁止。 读, 1: 已使能。
位 1	OVRFLW	读/写	0	写入 1 时使能 OVRFLW 事件，写入 0 无效。 写, 1: 使能。 读, 0: 已禁止。 读, 1: 已使能。
位 16	COMPARE0	读/写	0	写入 1 时使能 COMPARE0 事件，写入 0 无效。 写, 1: 使能。 读, 0: 已禁止。 读, 1: 已使能。
位 17	COMPARE1	读/写	0	写入 1 时使能 COMPARE1 事件，写入 0 无效。 写, 1: 使能。 读, 0: 已禁止。 读, 1: 已使能。
位 18	COMPARE2	读/写	0	写入 1 时使能 COMPARE2 事件，写入 0 无效。 写, 1: 使能。 读, 0: 已禁止。 读, 1: 已使能。
位 19	COMPARE3	读/写	0	写入 1 时使能 COMPARE3 事件，写入 0 无效。 写, 1: 使能。 读, 0: 已禁止。 读, 1: 已使能。

■ EVTENCLR: 事件使能寄存器

EVTENCLR 寄存器用于禁止 RTC 的各个事件。位的值写为 1 时禁止事件，写入 0 无效。

注意 EVTENCLR 寄存器只能用来禁止事件。

表 20-9: EVTENCLR 寄存器

位	Field	RW	复位值	描述
位 0	TICK	读/写	0	写入 1 时禁止 TICK 事件, 写入 0 无效。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。
位 1	OVRFLW	读/写	0	写入 1 时禁止 OVRFLW 事件, 写入 0 无效。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。
位 16	COMPARE0	读/写	0	写入 1 时禁止 COMPARE0 事件, 写入 0 无效。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。
位 17	COMPARE1	读/写	0	写入 1 时禁止 COMPARE1 事件, 写入 0 无效。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。
位 18	COMPARE2	读/写	0	写入 1 时禁止 COMPARE2 事件, 写入 0 无效。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。
位 19	COMPARE3	读/写	0	写入 1 时禁止 COMPARE3 事件, 写入 0 无效。 写, 1: 禁止。 读, 0: 已禁止。 读, 1: 已使能。

■ COUNTER 寄存器

COUNTER 寄存器保存 RTC 计数值，24 位。

表 20-10: COUNTER 寄存器

位	Field	RW	复位值	描述
位 23~位 0	COUNTER	只读	0	计数值。

■ PRESCALER: 分频系数配置寄存器

PRESCALER 寄存器用于配置分频系数，12 位。

表 20-11: PRESCALER 寄存器

位	Field	RW	复位值	描述
位 11~位 0	PRESCLER	读/写	0	分频系数。

■ CC[n]寄存器(n=0~3): 比较值配置寄存器。

CC[n](n=0~3)寄存器用于配置 RTC 通道的比较值，24 位。

表 20-12: CC[n](n=0~3)寄存器

位	Field	RW	复位值	描述
位 23~位 0	CC[n](n=0~3)	读/写	0	比较值。

4. 软件设计

4.1. 库函数的应用

RTC 的应用步骤如下图所示，因为 RTC 使用低频时钟 LFCLK 作为时钟源，所以我们需要配置低频时钟，时钟配置完成之后初始化驱动程序实例，驱动程序实例对应具体的硬件 RTC 外设 (RTC0~RTC2)。驱动程序实例初始化完成后，接着根据我们的需求配置需要使用的 RTC 事件，如我们要实现定时，可以使用 TICK 事件或比较事件，而溢出事件相对用的较少。事件配置后，当事件产生时，RTC 会进入应用程序初始化 RTC 驱动程序实例时注册的事件回调函数，应用程序在 RTC 事件回调函数中根据事件类型执行相应的功能代码。

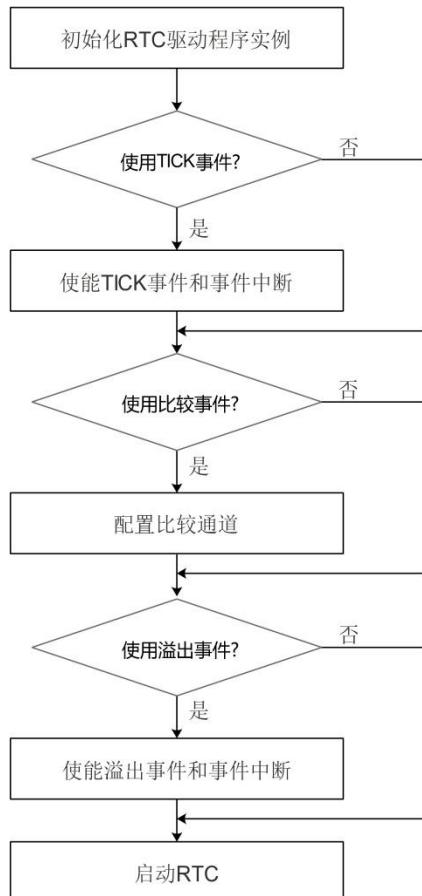


图 20-11：RTC 应用流程

4.1.1. 初始化 RTC 驱动程序实例

初始化 RTC 驱动程序实例之前，我们需要先定义 RTC 驱动程序实例，驱动程序实例数据结构如下，该结构体有 4 个成员变量，其中“p_reg”用于关联到一个具体的 RTC 外设（RTC0~RTC2），“instance_id”记录驱动程序实例在 RTC 驱动程序中的索引，因为驱动程序管理了 3 个 RTC 外设，所以需要通过索引来标志各个 RTC。“cc_channel_count”是该 RTC 拥有的比较通道数量。“irq”是 RTC 的中断号，RTC0、RTC1 和 RTC2 的中断号定义在“nrf52.h”头文件中。

代码清单：RTC 驱动程序实例数据结构

```

7. typedef struct
8. {
9.     NRF_RTC_Type * p_reg;           //指向 RTC 外设结构体
10.    IRQn_Type      irq;            //中断号
11.    uint8_t        instance_id;    //驱动程序实例索引
12.    uint8_t        cc_channel_count; //比较通道梳理
13. } nrfx_rtc_t;
  
```

驱动程序实例结构体 nrfx_rtc_t 描述了一个具体的 RTC 外设，当我们定义了 nrfx_rtc_t 类型的变量并对其赋值后，该变量就对应了一个具体的硬件 RTC 外设。

RTC 库函数中专门提供了一个用于初始化 RTC 驱动程序实例的宏 NRFX_RTC_INSTANCE，该宏的输入参数“id”的数值和 RTC 外设编号对应，如“id”为 0，该宏对应的就是 RTC0。

代码清单：RTC 驱动程序实例初始化宏

```

7. #define NRFX_RTC_INSTANCE(id) \
8. { \
9.     .p_reg      = NRFX_CONCAT_2(NRF_RTC, id), \
10.    .irq        = NRFX_CONCAT_3(RTC, id, _IRQn), \
11.    .instance_id = NRFX_CONCAT_3(NRFX_RTC, id, _INST_IDX), \
12.    .cc_channel_count = NRF_RTC_CC_CHANNEL_COUNT(id), \
13. }
```

初始化宏 NRFX_RTC_INSTANCE 中通过字符串连接的方式来区别各个 RTC 的参数，这么做的好处是只需要定义这一个初始化宏就可以通过“id”的值来实现对各个 RTC 驱动程序实例的配置。

下面是去除了字符串连接后 RTC0 应用程序实例的配置宏(即“id”值为 0)，这样看起来更清楚一些。

代码清单：“id”值为 0 时 RTC 驱动程序实例初始化宏

```

1. #define NRFX_RTC_INSTANCE(id) \
2. { \
3.     .p_reg      = NRF_RTC0, \
4.     .irq        = RTC0_IRQn, \
5.     .instance_id = NRFX_RTC0_INST_INDEX, \
6.     .cc_channel_count = RTC0_CC_NUM, \
7. }
```

参数说明如下：

- p_reg = NRF_RTC0 : 指向了 RTC0 的寄存器基址 0x4000B000。
- irq = RTC0_IRQn: RTC0 中断号，RTC0_IRQn = 11。
- instance_id = RTC0_INSTANCE_INDEX: RTC0 应用程序实例索引，RTC0_INSTANCE_INDEX = 0。
- cc_channel_count = RTC0_CC_NUM: RTC0 CC 通道数量，RTC0_CC_NUM = 3。

到这里，我们可以清楚地看到：声明的 RTC 应用程序实例被实例化后，驱动程序实例和硬件 RTC 关联在了一起。

- RTC 驱动程序实例定义示例：定义一个名称为 MY_RTC 的 RTC 驱动程序实例，该实例使用硬件 RTC0。

```
const nrfx_rtc_t MY_RTC = NRFX_RTC_INSTANCE(0);
```

- RTC 驱动程序实例定义示例：定义一个名称为 MY_RTC 的 RTC 驱动程序实例，该实例使用硬件 RTC1。

```
const nrfx_rtc_t MY_RTC= NRFX_RTC_INSTANCE(1);
```

定义了 RTC 驱动程序实例后，调用库函数 `nrfx_rtc_init()` 初始化 RTC，`nrfx_rtc_init()` 函数原型如下：

表 20-13: `nrfx_rtc_init()` 函数

函数原型	<code>nrfx_err_t nrfx_rtc_init(nrfx_rtc_t const *const p_instance, nrfx_rtc_config_t const * p_config, nrfx_rtc_handler_t handler)</code>
函数功能	RTC 初始化函数，初始化时注册事件句柄。初始化后，RTC 处于掉电模式。。
参数	<p>[in] <code>p_instance</code>: 指向 RTC 应用程序实例。</p> <p>[in] <code>p_config</code>: 指向 RTC 配置结构体。</p> <p>[in] <code>handler</code>: 用户提供的事件句柄，不能为 NULL。</p>
返回值	<p><code>NRF_SUCCESS</code>: 初始化成功。</p> <p><code>NRF_ERROR_INVALID_STATE</code>: 该 RTC 驱动程序实例已经初始化。</p>

`nrfx_rtc_init()` 函数有 3 个输入参数，其中 `p_instance` 指向我们定义的 RTC 驱动程序实例，`p_config` 指向 RTC 配置结构体，`handler` 指向 RTC 事件回调函数。所以在调用 `nrfx_rtc_init()` 函数之前，我们还需要提供 RTC 配置结构体和 RTC 事件回调函数。

6. 配置结构体

驱动程序提供了一个 RTC 配置结构体 `nrfx_rtc_config_t`，该结构体包含了 RTC 配置所用的参数，RTC 配置结构体声明如下：

代码清单：RTC 配置结构体

```
1. typedef struct  
2. {  
3.     uint16_t prescaler;           //分频系数  
4.     uint8_t  interrupt_priority; //中断优先级  
5.     uint8_t  tick_latency;       //中断抢占的最大时间(最大 7.7 ms)  
6.     bool    reliable;            //可靠模式标志  
7. } nrfx_rtc_config_t;
```

一般地，在定义 RTC 配置结构体时，会使用宏 `NRFX_RTC_DEFAULT_CONFIG` 初始化配置结构体，该宏定义了 RTC 的默认配置宏，代码清单如下。

代码清单：RTC 配置结构体初始化宏

```
1. #define NRFX_RTC_DEFAULT_CONFIG \\\  
2. { \\\
```

```

3.     .prescaler = RTC_FREQ_TO_PRESCALER(NRFX_RTC_DEFAULT_CONFIG_FREQUENCY), \
4.     .interrupt_priority = NRFX_RTC_DEFAULT_CONFIG_IRQ_PRIORITY, \
5.     .reliable          = NRFX_RTC_DEFAULT_CONFIG_RELIABLE, \
6.     .tick_latency       = NRFX_RTC_US_TO_TICKS(NRFX_RTC_MAXIMUM_LATENCY_US, \
7.                                         NRFX_RTC_DEFAULT_CONFIG_FREQUENCY), \
8.   }

```

RTC 配置结构体初始化宏中的参数定义在“ `sdk_config.h`”文件中，我们可以根据需要修改“ `sdk_config.h`”文件中的配置参数，下面是“ `sdk_config.h`”文件中 RTC 时钟的配置，对于其他参数的配置，“ `sdk_config.h`”文件中都有定义，此处不一一列出。

代码清单：“ `sdk_config.h`”文件中 RTC 时钟配置

```

1. // <o> RTC_DEFAULT_CONFIG_FREQUENCY - Frequency <16-32768>
2.
3. #ifndef RTC_DEFAULT_CONFIG_FREQUENCY
4. #define RTC_DEFAULT_CONFIG_FREQUENCY 32768
5. #endif

```

- RTC 配置结构体定义示例：定义一个名称为 `my_rtc_cfg` 的 RTC 配置结构体并初始化。

```
nrfx_rtc_config_t my_rtc_cfg= NRFX_RTC_DEFAULT_CONFIG;
```

7. RTC 事件回调函数

RTC 事件回调函数的编写格式如下，事件回调函数是在 RTC 初始化的时候注册给驱动程序的。当 RTC 事件产生后，事件回调函数被执行，在事件回调函数中我们可以判断产生了哪些事件，并添加需要执行的功能代码。

代码清单：RTC 事件回调函数

```

1. static void rtc_handler(nrf_drv_rtc_int_type_t int_type)
2. {
3.     //判断产生的事件是否是 TICK 事件
4.     if (int_type == NRFX_RTC_INT_COMPARE0)
5.     {
6.         /* 添加功能代码 */
7.     }
8.     /* 添加比较事件或溢出事件*/
9. }

```

RTC 可产生的事件有 6 个，即 `TICK` 事件，4 个比较事件（`CC[0]~CC[3]`）和溢出事件，这些事件定义在 `nrfx_rtc_int_type_t` 中。

代码清单：RTC 事件

```

1. typedef enum
2. {
3.     NRFX_RTC_INT_COMPARE0 = 0, //COMPARE0 比较事件
4.     NRFX_RTC_INT_COMPARE1 = 1, //COMPARE1 比较事件
5.     NRFX_RTC_INT_COMPARE2 = 2, //COMPARE2 比较事件
6.     NRFX_RTC_INT_COMPARE3 = 3, //COMPARE3 比较事件
7.     NRFX_RTC_INT_TICK      = 4, //TICK 事件
8.     NRFX_RTC_INT_OVERFLOW  = 5 //溢出事件
9. } nrfx_rtc_int_type_t;

```

4.1.2. 使能 TICK

如果我们的应用中需要使用 TICK 事件，那么就需要调用 `nrfx_rtc_tick_enable()` 函数使能 TICK 事件，该函数原型如下表所示。`nrfx_rtc_tick_enable()` 函数使能 TICK 事件时可选择是否使能 TICK 事件中断，如果我们需要 TICK 事件产生后，执行应用程序注册的 RTC 事件回调函数，“enable_irq” 应设置为“true”，即使能 TICK 事件中断。

表 20-14: `nrfx_rtc_tick_enable()` 函数

函数原型	<code>void nrfx_rtc_tick_enable (nrfx_rtc_t const *const p_instance, bool enable_irq)</code>
函数功能	使能 RTC tick 事件，可选择是否使能 tick 事件中断。
参 数	[in] <code>p_instance</code> : 指向 RTC 驱动程序实例。 [in] <code>enable_irq</code> : true 使能 tick 中断, false 禁止 tick 中断。
返回值	无。

4.1.3. 配置比较通道

每个 RTC 外设都拥有 4 个比较通道 CC[0]~ CC[3]，使用比较事件时需要设置具体比较通道的比较值，同时设置该通道比较事件是否产生中断。RTC 比较通道使用库函数 `nrfx_rtc_cc_set()` 配置，该函数有 4 个输入参数，其中 val 是通道的比较值，val 和分频系数一起决定了比较事件产生的间隔，计算方法见“2.1 节中计数器递增频率的计算”。输入参数“enable_irq”用于设置是否使能比较事件中断，“enable_irq”为“true”时，使能比较事件中断，比较事件产生后，执行 RTC 回调函数。

表 20-15: `nrfx_rtc_cc_set()` 函数

函数原型	<code>nrfx_err_t nrfx_rtc_cc_set(nrfx_rtc_t const *const p_instance, uint32_t channel, uint32_t val, bool enable_irq)</code>
-------------	--

函数功能	<p>设置 RTC 比较通道。如果没有初始化 RTC 驱动程序实例或者通道参数错误，函数会返回错误代码。如果 RTC 应用程序实例处于掉电状态，该函数会将应用程序实例设置为“上电”状态。</p> <p>在配置 RTC 时，驱动程序没有进入临界区，这意味着它可能会被抢占一段时间。当被抢占时，如果我们配置的比较值对应的时间较短，这就可能会出现配置的比较值落后于计数值。RTC 的驱动程序考虑到了这一点，RTC 驱动程序通过设置宏定义 <code>RTCn_CONFIG_RELIABLE = 1</code> 进入可靠的配置模式，所谓的可靠模式是指：如果出现待设置的比较值落后于计数值，函数返回 <code>NRF_ERROR_TIMEOUT</code> 通知应用程序。可靠配置模式是在下面两点成立的基础上实现的。</p> <ul style="list-style-type: none"> • 已知的最大抢占时间是 7.7ms (prescaler = 0, RTC 频率 32 kHz)。 • 应用程序应确保请求的比较绝对值不能超过 0xFFFFFFFF。
参 数	<p>[in] <code>p_instance</code>: 指向 RTC 驱动程序实例。</p> <p>[in] <code>channel</code>: RTC 比较通道号，RTC0 有 3 个比较通道，RTC1 和 RTC2 有 4 个比较通道。</p> <p>[in] <code>val</code>: 写入比较寄存器的数值(绝对值)。</p> <p>[in] <code>enable_irq</code>: 使能或禁止该比较通道的中断，True: 使能中断，False: 禁止中断。</p>
返回值	<p><code>NRF_SUCCESS</code>: 比较值设置成功。</p> <p><code>NRF_ERROR_TIMEOUT</code>: 设置的比较值落后于当前计数值时返回该错误代码。该错误代码仅在 <code>RTCn_CONFIG_RELIABLE = 1</code> 时有效。</p>

4.1.4. 启动/停止 RTC

RTC 配置完成后，调用 `nrfx_rtc_enable()` 函数启动 RTC，该函数会触发 START 任务启动 RTC，同时在驱动程序控制块中将该 RTC 设置为已初始化。停止 RTC 使用的库函数是 `nrfx_rtc_disable()` 函数，该函数会触发 STOP 任务停止 RTC，同时在驱动程序控制块中将该 RTC 设置为未初始化。

表 20-16: `nrfx_rtc_enable()` 函数

函数原型	<code>void nrfx_rtc_enable(nrfx_rtc_t const *const p_instance)</code>
函数功能	使能 RTC 驱动程序实例，即触发 START 任务启动该 RTC。
参 数	[in] <code>p_instance</code> : 指向 RTC 驱动程序实例。
返回值	无。

表 20-17: `nrfx_rtc_disable()` 函数

函数原型	<code>void nrfx_rtc_disable(nrfx_rtc_t const *const p_instance)</code>
函数功能	禁止 RTC 驱动程序实例，即触发 STOP 任务停止该 RTC。

参 数	[in] <code>p_instance</code> : 指向 RTC 驱动程序实例。
返回值	无。

4.2. TICK 事件定时实验

本实验在“实验 6-3: 流水灯(BSP 实现方式)”的基础上修改。配置 RTC0 的分频系数为 4095，根据计数器递增频率的计算公式可以计算出计数器的递增频率：

$$\text{递增频率} = 32768 / (4095 + 1) = 8\text{Hz}$$

即：计数器每 125ms 递增一次，由前文对 TICK 事件的描述，我们知道 TICK 事件是在 COUNTER 计数器每次递增时产生的(TICK 事件使能时)，所以 TICK 事件也是每 125ms 产生一次。每个 TICK 事件发生时在事件回调函数中将指示灯 D1 的状态取反。

❖ 注：本节对应的实验源码是：“实验 20-1：RTC TICK 事件实现定时”。

4.2.1. 添加需要的文件

随机数发生器需要加入的文件如下表所示。

表 20-18：随机数发生器需要加入的文件

文件名	SDK 中的目录	描述
<code>nrf_drv_clock.c</code>	<code>..\integration\nrfx\legacy</code>	旧版本 CLOCK 驱动文件。
<code>nrfx_clock.c</code>	<code>..\modules\nrfx\drivers\src</code>	新版本 CLOCK 驱动文件。
<code>nrfx_rtc.c</code>	<code>..\modules\nrfx\drivers\src</code>	RTC 驱动文件。

4.2.2. 头文件引用和路径设置

1. 需要引用的头文件

因为在“main.c”文件中使用了随机数发生器程序模块，所以“main.c”文件需要引用下面的头文件。

```
#include "nrf_drv_rtc.h"
#include "nrf_drv_clock.h"
```

2. 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

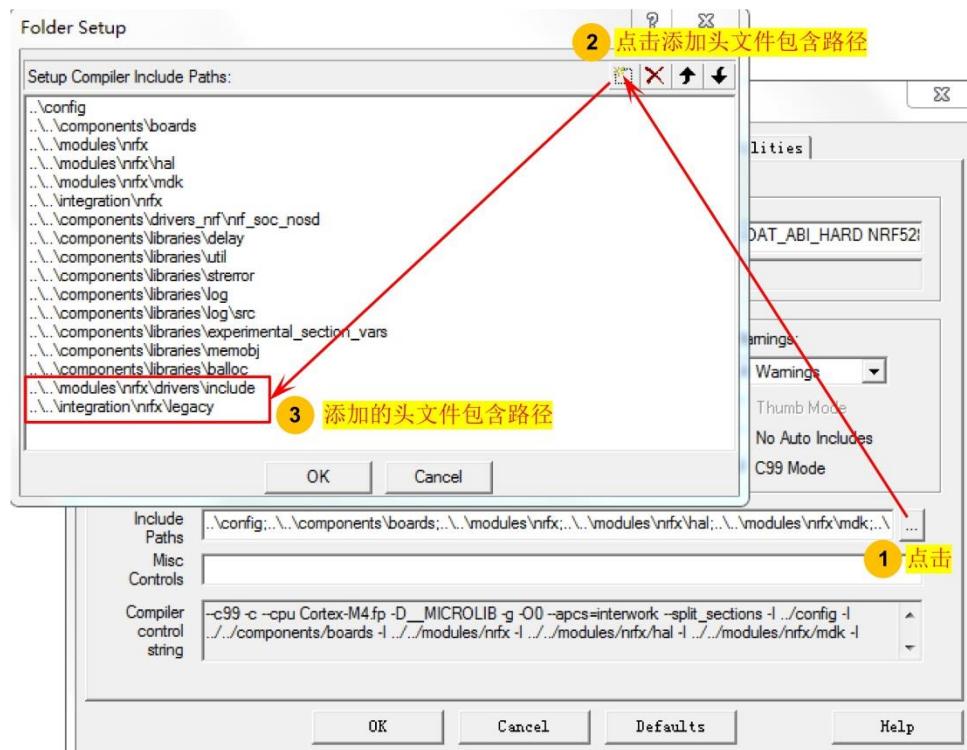


图 20-12：添加头文件包含路径

随机数发生器需要添加的头文件路径如下表：

表 20-19：头文件包含路径

序号	路径
1	..\\..\\integration\\nrfx\\legacy
2	..\\..\\modules\\nrfx\\drivers\\include

4.2.3. 工程配置

打开“sdk_config.h”文件，加入低频时钟 LFCLK 和 RTC 所需的配置，如下图所示，编辑内容时切换到“Text Editor”进行编辑（编辑的内容参考实验的源码，因此代码很长，此处不贴出代码，参见“sdk_config.h”文件的(52~355)行），编辑完成后，切换到“Configuration Wizard”，即可观察到配置的具体项目：

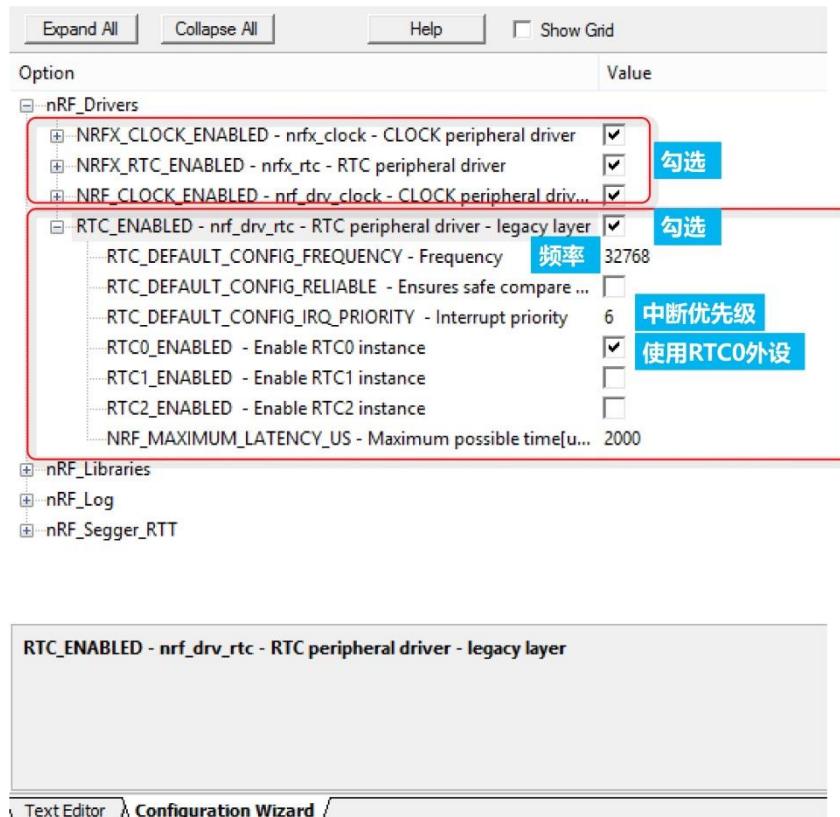


图 20-13: 工程配置

4.2.4. 代码编写

按照前文描述的步骤，我们使用 RTC 的时候，首先定义一个 RTC 驱动程序实例，本例中使用的是硬件 RTC0，程序清单如下。

代码清单：定义 RTC 驱动程序实例

```
1. //定义 RTC0 的驱动程序实例，0 对应 RTC0
2. const nrf_drv_rtc_t rtc = NRF_DRV_RTC_INSTANCE(0);
```

本例中使用 RTC 的 TICK 事件，并且应用程序需要在 RTC 事件回调函数中执行驱动 LED 指示灯，所以 RTC 初始化完成后，需要调用 nrfx_rtc_tick_enable() 函数使能 TICK 事件和 TICK 事件中断，代码清单如下。

代码清单：RTC 配置

```
40. static void rtc_config(void)
41. {
42.     uint32_t err_code;
43.
44.     //定义 RTC 初始化配置结构体，并使用默认参数初始化
45.     nrfx_rtc_config_t config = NRF_RTC_DEFAULT_CONFIG;
46.     //重写分频系数，分频系数设置为 4095 时的递增频率 = 32768/(4095+1) = 8Hz,
47.     //即每 125ms COUNTER 计数器递增一次。
48.     //TICK 事件是在 COUNTER 计数器递增时发生，所以 TICK 事件每 125ms 产生一次
```

```
49.     config.prescaler = 4095;
50.     //初始化 RTC 驱动程序实例的驱动，注册事件句柄
51.     err_code = nrfx_rtc_init(&rtc, &config, rtc_handler);
52.     APP_ERROR_CHECK(err_code);
53.
54.     //使能 TICK 事件
55.     nrfx_rtc_tick_enable(&rtc, true);
56.
57.     //启动 RTC
58.     nrfx_rtc_enable(&rtc);
59. }
```

RTC 初始化时需要注册事件回调函数，所以应用程序需要提供该回调函数，事件回调函数中驱动 LED 翻转状态，本例中 TICK 事件每 125ms 产生一次，即指示灯 D1 以 125ms 的间隔闪烁。

代码清单：RTC 配置

```
1. static void rtc_handler(nrfx_rtc_int_type_t int_type)
2. {
3.     //判断产生的事件是否是 TICK 事件，本例中设置的 TICK 事件每 125ms 产生一次
4.     if (int_type == NRFX_RTC_INT_TICK)
5.     {
6.         //翻转指示灯 D1 的状态
7.         nrf_gpio_pin_toggle(LED_1);
8.     }
9. }
```

最后，在主函数中调用 lfclk_config()函数配置低频时钟和 rtc_config()函数配置并启动 RTC。

代码清单：主函数

```
1. int main(void)
2. {
3.     //初始化开发板上的 4 个 LED，即将驱动 LED 的 GPIO 配置为输出，
4.     bsp_board_init(BSP_INIT_LEDS);
5.
6.     //初始化低频时钟
7.     lfclk_config();
8.     //RTC 配置
9.     rtc_config();
10.
11.    while (true)
12.    {
13.        //进入低功耗，等待事件唤醒
14.    }
15. }
```

```
14.     __SEV();  
15.     __WFE();  
16.     __WFE();  
17. }  
18. }
```

4.2.5. 硬件连接

本实验需要使用 P0.13 驱动 LED 指示灯 D1，按照下图所示短接跳线帽。

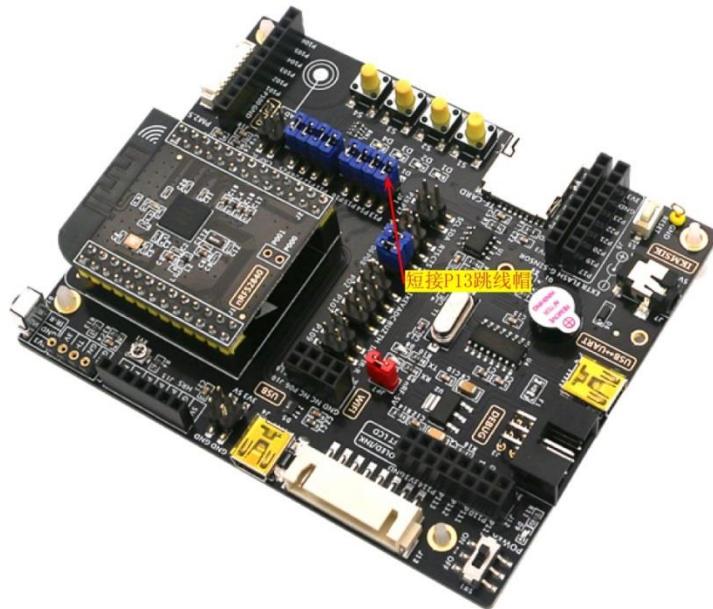


图 20-13：开发板跳线帽短接

4.2.6. 实验步骤

1. 解压“…\3: 实验源码\”目录下的压缩文件“实验 20-1: RTC TICK 事件”，将解压后得到的文件夹“rtc_tick”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\rtc_tick\project\mdk5”目录下的工程“rtc_tick.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nRF52840_qfaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，RTC0 每 125ms 产生一次 TICK 事件中断，事件回调函数中翻转指示灯 D1 的状态，可以观察到，D1 以 125ms 的间隔闪烁。

4.3. 比较事件定时实验

本实验在“实验 20-1: RTC TICK 事件”的基础上修改。RTC 部分的配置不变：RTC0

的分频系数同样设置为 4095，COUNTER 计数器的递增频率等于 8Hz。区别是不使用 TICK 事件，而是使用比较通道 0 来实现定时，本例中实现 2 秒的定时，所以需要将通道 0 的比较值设置为 $8 \times 2 = 16$ 。这样，每 2 秒会产生一次通道 0 比较匹配事件，在事件的回调函数中将指示灯 D1 的状态取反，同时清零计数器，重新计数。

◆ 注：本节对应的实验源码是：“实验 20-2：RTC 比较事件实现定时”。

4.3.1. 代码编写

本例和“RTC TICK 事件”实验的区别是：不使能 TICK 事件，而是使用通道 0 的比较匹配事件实现定时，所以需要调用 nrfx_rtc_cc_set() 函数配置通道 0，而不需要使能 TICK 事件。

代码清单： RTC0 配置

```
1. //计数器对应的时间，单位是秒，这里定义的是 2 秒
2. #define COMPARE_COUNTERTIME (2UL)
3.
4. static void rtc_config(void)
5. {
6.     uint32_t err_code;
7.
8.     //定义 RTC 初始化配置结构体，并使用默认参数初始化
9.     nrfx_rtc_config_t config = NRFX_RTC_DEFAULT_CONFIG;
10.    //重写分频系数，分频系数设置为 4095 时的递增频率 = 32768/(4095+1) = 8Hz,
11.    //即每 125ms 计数器递增一次。
12.    //TICK 事件是在 COUNTER 计数器递增时发生，所以 TICK 事件每 125ms 产生一次
13.    config.prescaler = 4095;
14.    //初始化 RTC 驱动程序实例的驱动，注册事件句柄
15.    err_code = nrfx_rtc_init(&rtc, &config, rtc_handler);
16.    APP_ERROR_CHECK(err_code);
17.
18.    //设置 RTC0 通道 0 的比较值：RTC0 分频系数设置的是 4095，所以计数器递增频率是 8,
19.    //即 1 秒递增 8 次，因此，这里 COMPARE_COUNTERTIME 对应的就是秒
20.    err_code = nrfx_rtc_cc_set(&rtc, 0, COMPARE_COUNTERTIME * 8, true);
21.    APP_ERROR_CHECK(err_code);
22.
23.    //启动 RTC
24.    nrfx_rtc_enable(&rtc);
25. }
```

RTC 事件回调函数中判断事件类型，如是通道 0 比较事件 NRFX_RTC_INT_COMPARE0，则翻转指示灯 D1 状态。

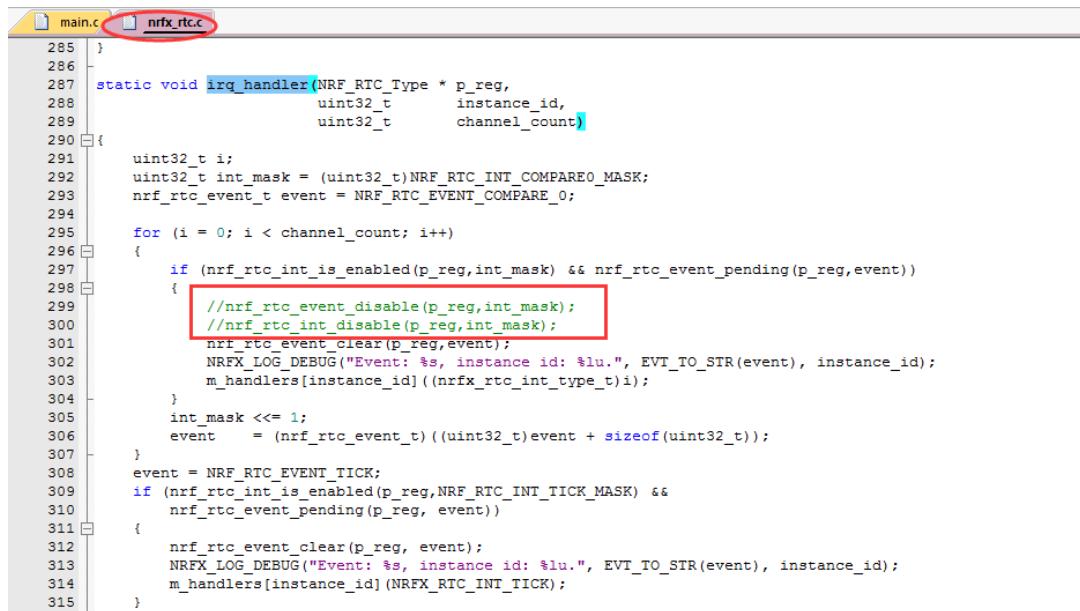
代码清单： RTC 事件回调函数中处理通道 0 比较事件

```

1. static void rtc_handler(nrfx_rtc_int_type_t int_type)
2. {
3.     //判断产生的事件是否是通道 0 比较事件
4.     if (int_type == NRF_RTC_INT_COMPARE0)
5.     {
6.         //翻转指示灯 D1 的状态
7.         nrf_gpio_pin_toggle(LED_1);
8.         //清零计数器
9.         nrfx_rtc_counter_clear(&rtc);
10.    }
11. }

```

RTC 产生事件后，会进入中断服务函数 `irq_handler()`，而中断服务函数中会禁止 RTC 比较事件和比较事件中断，也就是 RTC 比较事件产生后，因为 RTC 驱动程序中禁止了 RTC 比较事件和比较事件中断，所以比较事件只会执行一次事件回调函数，之后无法再次产生事件和事件中断，这显然无法实现重复定时的目的，因此，我们需要修改 RTC 驱动程序，即在 RTC 中断服务函数 `irq_handler()` 中屏蔽禁止比较事件和比较事件中断的代码，如下图所示。



```

main.c
nrfx_rtc.c

285 }
286
287 static void irq_handler(NRF_RTC_Type * p_reg,
288                         uint32_t           instance_id,
289                         uint32_t           channel_count)
290 {
291     uint32_t i;
292     uint32_t int_mask = (uint32_t)NRF_RTC_INT_COMPARE0_MASK;
293     nrf_rtc_event_t event = NRF_RTC_EVENT_COMPARE_0;
294
295     for (i = 0; i < channel_count; i++)
296     {
297         if (nrf_rtc_int_is_enabled(p_reg, int_mask) && nrf_rtc_event_pending(p_reg, event))
298         {
299             //nrf_rtc_event_disable(p_reg, int_mask);
300             //nrf_rtc_int_disable(p_reg, int_mask);
301             nrf_rtc_event_clear(p_reg, event);
302             NRF_LOG_DEBUG("Event: %s, instance id: %lu.", EVT_TO_STR(event), instance_id);
303             m_handlers[instance_id]((nrfx_rtc_int_type_t)i);
304         }
305         int_mask <= 1;
306         event    = (nrf_rtc_event_t)((uint32_t)event + sizeof(uint32_t));
307     }
308     event = NRF_RTC_EVENT_TICK;
309     if (nrf_rtc_int_is_enabled(p_reg, NRF_RTC_INT_TICK_MASK) &&
310         nrf_rtc_event_pending(p_reg, event))
311     {
312         nrf_rtc_event_clear(p_reg, event);
313         NRF_LOG_DEBUG("Event: %s, instance id: %lu.", EVT_TO_STR(event), instance_id);
314         m_handlers[instance_id](NRF_RTC_INT_TICK);
315     }
}

```

图 20-14：屏蔽禁止比较事件和比较事件中断的代码

4.3.2. 硬件连接

同“实验 17-1：RTC TICK 事件”。

4.3.3. 实验步骤

- 解压“…\3：开发指南（上册）配套试验源码\”目录下的压缩文件“实验 20-2：RTC

比较事件实现定时”，将解压后得到的文件夹“`rtc_compare`”拷贝到合适的目录，如“`D:\nRF52840`”。

2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“`…\rtc_compare\project\mdk5`”目录下的工程“`rtc_compare.uvproj`”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“`nRF52840_qfaa.hex`”位于工程目录下的“`Objects`”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，RTC0 每 2 秒产生一次通道 0 比较事件中断，事件回调函数中翻转指示灯 D1 的状态，可以观察到，D1 以 2 秒的间隔闪烁。

4.4. 溢出事件

本实验在“实验 20-1：RTC TICK 事件”的基础上修改。RTC0 的分频系数设置为 4095，COUNTER 计数器的递增频率等于 8Hz，即计数器每 125ms 递增一次。主循环检测按键 S1 按下后启动 RTC0，并触发 TRIGOVRFLLW(溢出)任务，触发后，RTC0 计数器的值被设置为：0xFFFFF0，16 次计数(计数器递增 16 次，即 $125\text{ms} \times 16 = 2000\text{ms}$)后，计数器的值由 0xFFFFF0 到 0x000000，这时会产生 OVRFLW(溢出)事件，在事件的回调函数中点亮指示灯 D1。

❖ 注：本节对应的实验源码是：“实验 20-3：RTC 溢出事件”。

4.4.1. 代码编写

本例和“RTC TICK 事件”实验的区别是：不使能 TICK 事件，RTC 配置函数 `rtc_config` 中增加使能 RTC0 溢出事件和事件中断的代码，同时去掉启动 RTC 的代码，即 RTC 在初始化配置的时候不启动。

代码清单： RTC0 配置

```
1. static void rtc_config(void)
2. {
3.     uint32_t err_code;
4.
5.     //定义 RTC 初始化配置结构体，并使用默认参数初始化
6.     nrfx_rtc_config_t config = NRFX_RTC_DEFAULT_CONFIG;
7.     //重写分频系数，分频系数设置为 4095 时的递增频率 = 32768/(4095+1) = 8Hz,
8.     //即每 125ms 计数器递增一次。
9.     config.prescaler = 4095;
10.    //初始化 RTC 驱动程序实例的驱动，注册事件句柄
11.    err_code = nrfx_rtc_init(&rtc, &config, rtc_handler);
12.    APP_ERROR_CHECK(err_code);
```

```
13.  
14.    //使能 RTC0 的溢出事件和事件中断  
15.    nrf_drv_rtc_overflow_enable(&rtc, true);  
16. }
```

RTC 事件回调函数中判断事件类型，如是通道 RTC00 溢出事件 NRF_DRV_RTC_INT_OVERFLOW，则翻转指示灯 D2 状态，同时停止 RTC0。

代码清单： RTC 事件回调函数中处理溢出事件

```
1. static void rtc_handler(nrfx_rtc_int_type_t int_type)  
2. {  
3.     //判断产生的事件是否是溢出事件  
4.     if (int_type == NRF_DRV_RTC_INT_OVERFLOW)  
5.     {  
6.         //翻转指示灯 D2 的状态  
7.         nrf_gpio_pin_clear(LED_2);  
8.         //停止 RTC:触发 RTC_STOP 任务  
9.         nrf_drv_rtc_disable(&rtc);  
10.    }  
11. }
```

主循环中查询按键 S1 状态，当按键 S1 按下后，点亮指示灯 D1 指示按键 S1 按下，熄灭指示灯 D2，之后等待按键 S1 释放，S1 释放后启动 RTC0 并触发 RTC0 溢出任务。16 次计数(计数器递增 16 次)即 $16 * 125\text{ms} = 2$ 秒后，产生 RTC0 溢出事件，指示灯 D2 被点亮。

代码清单： RTC 事件回调函数中处理溢出事件

```
1. int main(void)  
2. {  
3.     //初始化开发板上的 4 个 LED，即将驱动 LED 的 GPIO 配置为输出,  
4.     bsp_board_init(BSP_INIT_LEDS | BSP_INIT_BUTTONS);  
5.  
6.     //初始化低频时钟  
7.     lfcclk_config();  
8.     //RTC 配置  
9.     rtc_config();  
10.  
11.    while (true)  
12.    {  
13.        //检测按键 S1 是否按下  
14.        if (nrf_gpio_pin_read(BUTTON_1) == 0)  
15.        {  
16.            //点亮指示灯 D1，指示按键 S1 按下  
17.            nrf_gpio_pin_clear(LED_1);  
18.            //熄灭指示灯 D2
```

```
19.         nrf_gpio_pin_set(LED_2);
20.         while(nrf_gpio_pin_read(BUTTON_1) == 0); //等待按键释放
21.         //熄灭指示灯 D1
22.         nrf_gpio_pin_set(LED_1);
23.         //启动 RTC
24.         nrf_drv_rtc_enable(&rtc);
25.         //触发 RTC0 溢出事件,触发后, RTC0 的计数值被设置为 0xFFFFF0,
26.         //16 次计数后(16*125ms=2000ms), 产生溢出事件
27.         nrf_rtc_task_trigger(rtc.p_reg,NRF_RTC_TASK_TRIGGER_OVERFLOW);
28.     }
29. }
30. }
```

4.4.2. 硬件连接

本实验需要使用 P0.13 P0.14 驱动 LED 指示灯 D1 和 D2, P0.11 检测按键 S1 的状态, 需要用跳线帽短接这些引脚, 如下图所示。

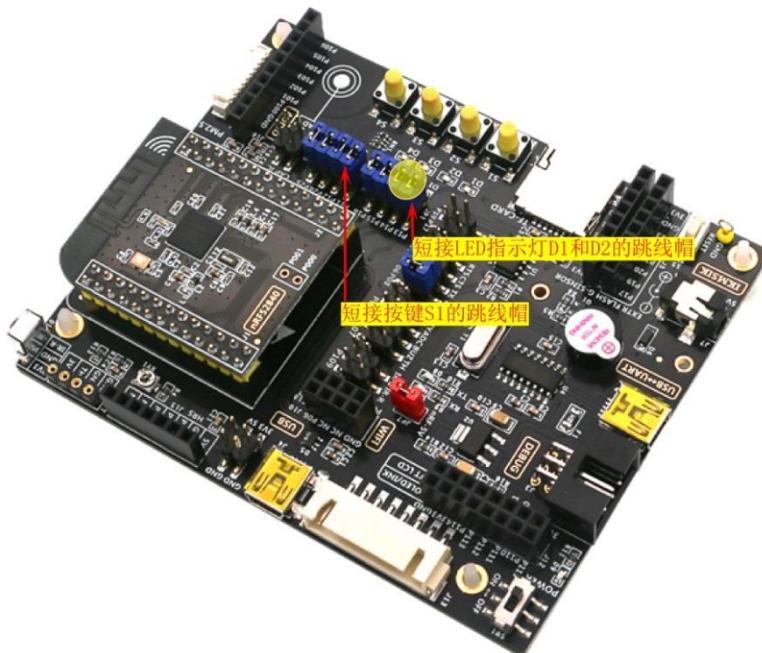


图 20-15：开发板跳线帽短接

4.4.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套试验源码\”目录下的压缩文件“实验 20-3: RTC 溢出事件”, 将解压后得到的文件夹“rtc_overflow”拷贝到合适的目录, 如“D\NRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project” 打开“…\rtc_overflow\project\mdk5” 目录下的工程“rtc_overflow.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏, 观察编译的结果, 如果有错误, 修改程序, 直到编译成功为止。编译后生成的 HEX 文件“nRF52840_qfaa.hex”位于工程目录

下的“Objects”文件夹中。

5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，按下 S1 按键然后松开，可以观察到约 2 秒后指示灯 D2 点亮，指示 RTC0 溢出事件的产生。

第二十一章：TWI(I2C)读写 EEPROM

1. 学习目的

1. 了解 I2C 总线的特点。
2. 掌握 I2C 地址的定义，对 I2C 地址要有深刻的了解，之后再看到 I2C 接口设备中描述的 7 位地址或 8 位地址，不会再有疑惑。
3. 了解 nRF52840 I2C 的特点以及库函数的编程方法。
4. 掌握通过 I2C 读写 eeprom AT24C02 以及需要注意的事项。

2. I2C 总线概述

2.1. 主要特征

nRF52840 片内集成了 TWI (Two-wire Serial Interface) 两线串行总线，TWI 完全兼容 I2C 总线，简单一点，可以直接认为 TWI 是 I2C 总线。

典型的 I2C 应用原理如下图所示，I2C 总线通讯仅需两根信号线，可以连接多个设备，从设备都有唯一的地址，主设备通过从设备的地址和不同的从设备通讯。

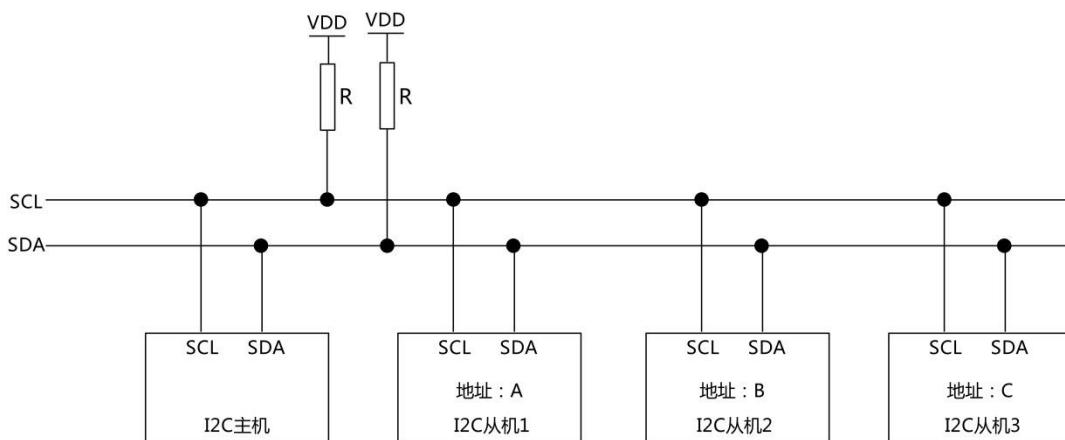


图 21-1：典型的 I2C 总线应用

- 1) I2C 总线硬件结构简单，仅需一根时钟线（SCL）、一根数据线（SDA）和两个上拉电阻即可实现通讯。I2C 总线的 SCL 和 SDA 均为开漏结构，开漏结构的电路只能输出“逻辑 0”，无法输出“逻辑 1”，因此 SCL 和 SDA 需要连接上拉电阻。上拉电阻的阻值影响传输速率，阻值越大，由于 RC 影响，会带来上升时间的增大，传输的速率慢，阻值小，传输的速率快，但是会增加电流的消耗，一般情况下，我们会选择 4.7K 左右的阻值，在从机数量少，信号线短的情况下，可以适当增加阻值，如使用 10K 的阻值。
- 2) I2C 总线中的从设备必须有自己的地址，并且该地址在其所处的 I2C 总线中唯一，主设备通过此唯一的地址即可和该从设备进行数据传输。

- 3) I2C 总线支持多主机，但是同一时刻只允许有一个主机。I2C 总线中存在多个主机时，为了避免冲突，I2C 总线通过总线仲裁决定由哪一个主机控制总线。
- 4) I2C 总线只能传输 8 位的数据，数据速率在标准模式下可达 100Kbit/s，快速模式下可达 400Kbit/s，高速模式下可达 3.4Mbit/s，nRF52840 的 I2C 总线不支持高速模式。
- 5) 同时连接到同一个 I2C 总线上的设备数量受总线最大电容（400pF）的限制。
- 6) I2C 总线电流消耗很低，抗干扰强，适合应用于低功耗的场合。

2.2. I2C 地址

I2C 总线中的设备必须要有唯一的地址，这意味着如果在总线中接入两个相同的设备，该设备必须有配置地址的功能，这也是我们经常用的 I2C 接口的设备会有几个引脚用来配置地址的原因。

对于 I2C 地址，我们经常看到有的 I2C 接口设备在规格书中描述的是 7 位地址，而有的 I2C 接口设备在规格书中描述的是 8 位地址，它们有什么区别？（I2C 也有 10 位地址，但用的较少，这里不做介绍，本章中的内容不涉及到 10 位地址）。

7 位地址和 8 位地址如下图所示，它们结构上是一样的，都是由 7 个地址位加一个用来表示读写的位组成，只是描述上有所区别。

- 规格书中描述 I2C 地址是 7 位地址的设备：给出的是 7 个地址位加 R/W 位，最低位（R/W 位）为 0 时表示为写地址，最低位为 1 时为读地址。如果把 0 和 1 分别带入 R/W 位，得到的地址就和 8 位地址一样了。
- 规格书中描述 I2C 地址是 8 位地址的设备：直接给出写地址和读地址，也就是最低位（R/W 位）为 0 时的地址和最低位为 1 时的地址。

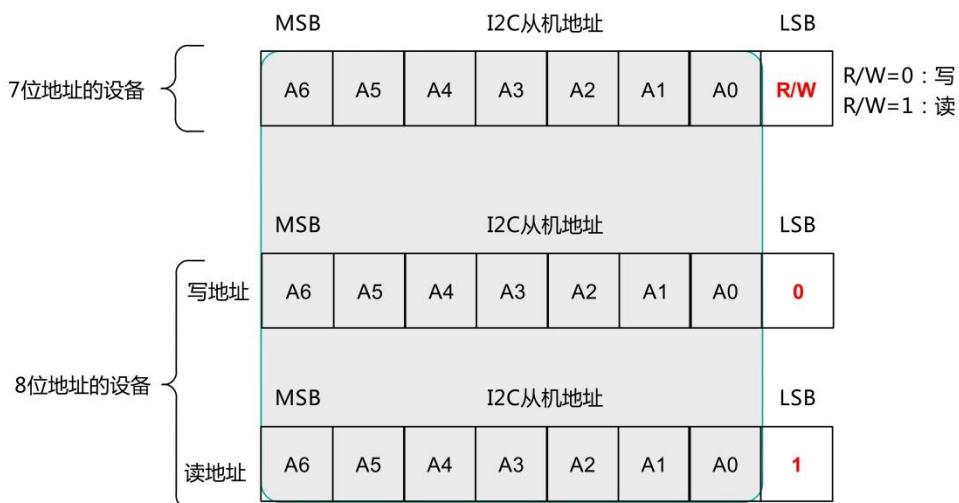


图 21-2:

由此可见，所谓的 7 位地址和 8 位地址实际上都是 7 位地址加上最低位的读写位，本质上是一样的，只是各个 I2C 接口设备的描述方式不一样。

I2C 保留了如下表所示的两组 I2C 地址，这些地址用于特殊用途。

表 21-1: 保留地址

从机地址	R/W 位	描述
------	-------	----

0000 000	0	广播呼叫地址。
0000 000	1	起始字节。
0000 001	X	CBUS 地址。
0000 010	X	保留给不同的总线格式。
0000 011	X	保留到将来使用。
0000 1XX	X	Hs 模式主机码。
1111 1XX	X	保留到将来使用。
1111 0XX	X	10 位从机寻址。

2.3. I2C 数据传输

1. 起始和停止条件 (START and STOP conditions)

所有的 I2C 事务都是以 START 开始、STOP 结束，起始和停止条件总是由主机产生，如下图所示，当 SCL 为高电平时，SDA 从高电平向低电平转换表示起始条件，当 SCL 是高电平时，SDA 由低电平向高电平转换表示停止条件。如果总线中存在多个主机，先将 SDA 拉低的主机获得总线控制权。

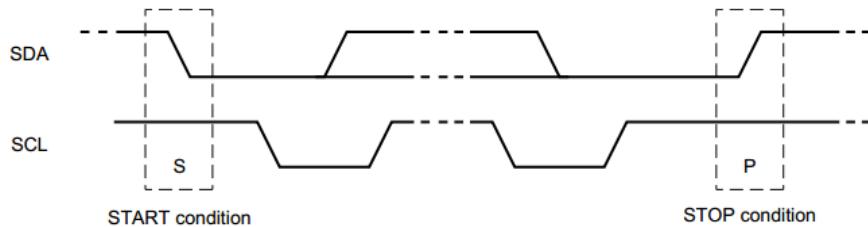
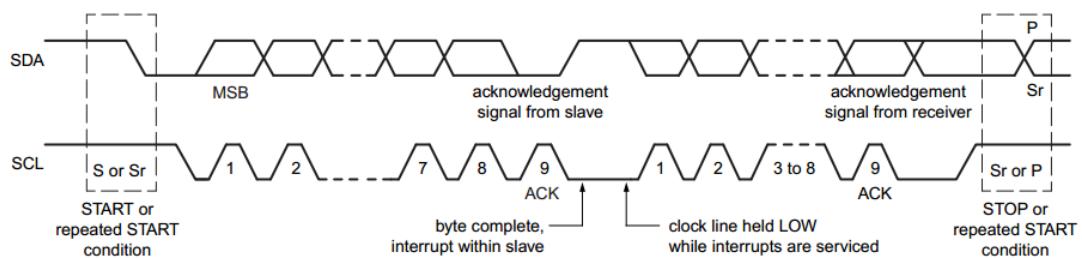


图 21-3: 起始和停止条件

2. 字节格式 (Byte format)

I2C 总线发送到 SDA 上的数据必须为 8 位，即一次传输一个字节，每次传输可以发送的字节数量不受限制。每个字节后必须跟一个响应位，首先传输的是数据的最高位 MSB，如果从机要完成一些其他功能后，例如一个内部中断服务程序才能接收或发送下一个完整的数据字节，那么从机可以将时钟线 SCL 保持为低电平强制主机进入等待状态，当从机准备好接收下一个字节数据并释放时钟线 SCL 后数据传输继续。



2.4. ACK 和 NACK

每个字节后会跟随一个 ACK 信号。接收者通过 ACK 位告知发送者已经成功接收一字节数据并准备好接收下一字节数据。所有的时钟脉冲包括 ACK 信号的时钟脉冲都是由主机

产生的。

- ACK 信号：发送者发送完 8 位数据后，在 ACK 时钟脉冲期间释放 SDA 线，接收者可以将 SDA 拉低并在时钟信号为高时保持低电平，这样就产生了 ACK 信号，从而使得主机知道从机已成功接收数据并且准备好了接收下一数据。
- NACK 信号：当 SDA 在第 9 个时钟脉冲的时候保持高电平，定义为 NACK 信号。这时，主机要么产生 STOP 条件来放弃这次传输，要么重复 START 条件来启动一个新的传输。

下面的 5 种情况会导致产生 NACK 信号：

- 1) 发送方寻址的接收方在总线上不存在，因此总线上没有设备应答。
- 2) 接收方正在处理一些实时的功能，尚未准备好与主机通信，因此接收方不能执行收发。
- 3) 在传输期间，接收方收到不能识别的数据或者命令。
- 4) 在传输期间，接收方无法接收更多的数据字节。
- 5) 主-接收器要通知从-发送器传输的结束。

2.5. 从机地址和 R/W 位

I2C 数据传输如下图所示，在起始条件(S)后，发送从机地址，从机地址是 7 位，从机地址后紧跟着的第 8 位是读写位 (R/W)，读写位为 0 表示写，读写位为 1 表示读。数据传输一般由主机产生的停止位 P 终止，但是，如果主机仍希望在总线上通讯，它可以产生重复起始条件 Sr 和寻址另一个从机而不是首先产生一个停止条件，在这种传输中可能有不同的读写格式结合。

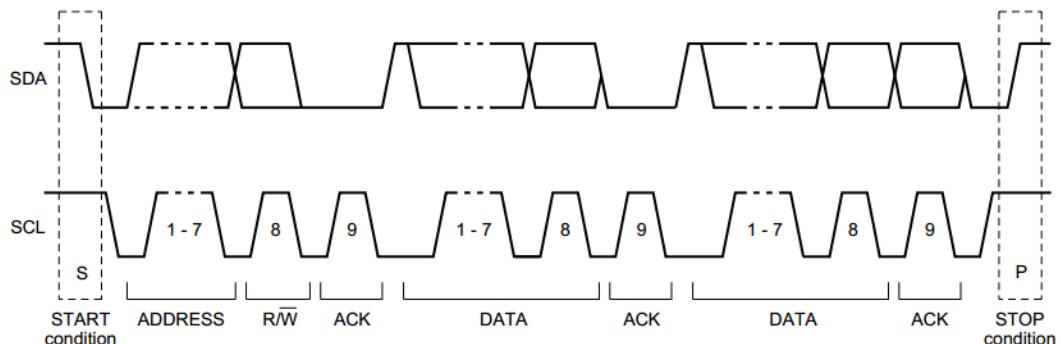


图 21-5: I2C 总线

可能的数据传输格式有：

1. 主机发送器发送到从机接收器，传输的方向不会改变，接收器应答每一个字节，如下图所示。

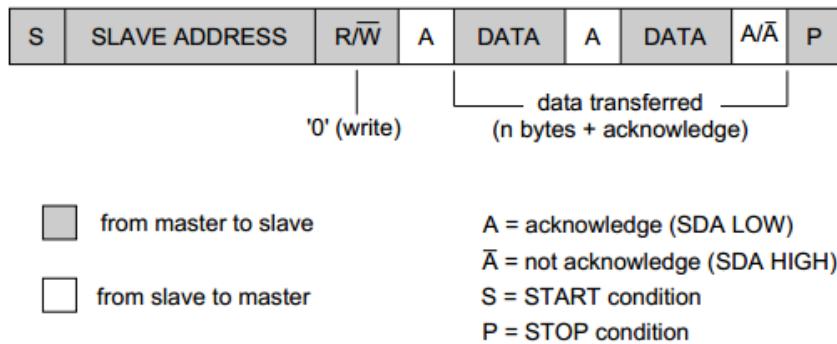


图 21-6: 主机发送器 7 位地址寻址从机接收器（传输方向不改变）

- 在第一个字节后，主机立即读从机，在第一次应答后，主机发送器变成主机接收器，从机接收器变成从机发送器。第一次应答仍由从机生成，主机生成后续应答。之前发送了一个非应答（A）的主机产生 STOP 条件。

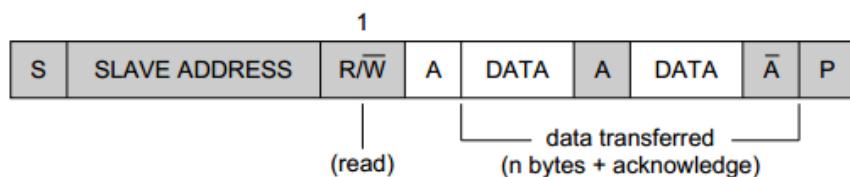


图 21-7: 主机发送第一个字节后立即读取从机

- 复合格式，如下图所示。传输改变方向的时候，起始条件和从机地址都会被重复，但 R/W 位取反。如果主接收器发送重复 START 条件，它会在重复 START 条件之前发送一个非应答（A）。

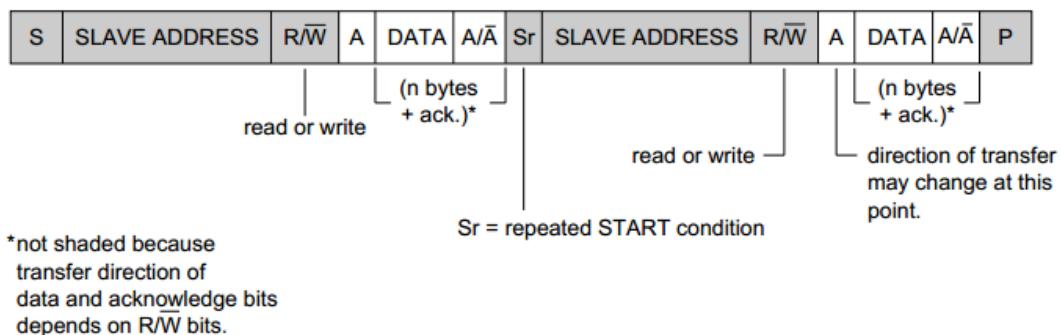


图 21-8: 复合格式

3. nRF52840 TWI 特点

nRF52840 片内集成的 TWI（两线串行总线）兼容 I2C 总线，带有 EasyDMA，可与连接到同一总线的多个从机设备通讯，主要特点如下：

- 兼容 I2C。
- 速率：100 kbps、250 kbps 或 400 kbps。
- 支持时钟延伸（非 I2C 兼容）。
- 带 EasyDMA。
- TWI 的 SCL 和 SDA 信号可以通过配置寄存器连接到任何一个 GPIO，这样可以灵活地

实现器件引脚排列，并有效利用电路板空间和信号路由。

nRF52840 的 TWI 的原理框图如下图所示，TWI 主机通过触发 STARTTX 或 STARTRX 任务启动 TWI 传输，通过触发 STOP 任务停止 TWI 传输。TWI 主机在挂起时无法停止，因此必须在 TWI 主机恢复后触发 STOP 任务停止 TWI。启动 TWI 主机后，在 TWI 主机停止之前，即在 LASTRX, LASTTX 或 STOPPED 事件之后，不应再次触发 STARTTX 任务或 STARTRX 任务。如果从机产生 NACK 输入，那么 TWI 主机将产生 ERROR 事件。

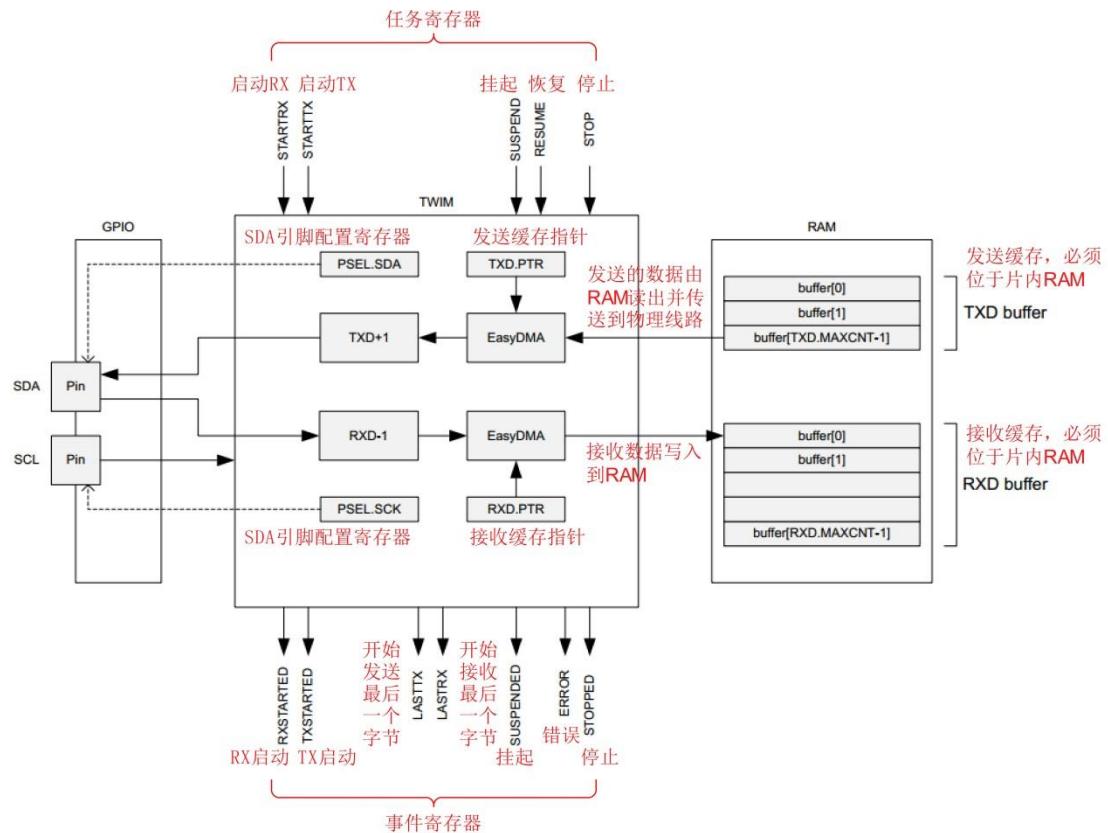


图 21-9: TWI 的原理框图

TWI 主机通过 EasyDMA 实现数据传输，因此 TWI 的接收和发送缓存必须位于数据 RAM 区域，如果 TXD.PTR 和 RXD.PTR 未指向数据 RAM 区域，则 EasyDMA 传输可能导致 HardFault 或 RAM 损坏。

.PTR 和 .MAXCNT 寄存器是双缓冲的，因此在收到 RXSTARTED/TXSTARTED 事件后，可以立即更新并准备下一个 RX/TX 传输。STOPPED 事件表示 EasyDMA 已完成访问 RAM 中的缓冲区。

4. TWI 寄存器

nRF52840 片内集成了 2 个 TWI 外设，如下表所示。

表 21-2: TWI 基址

外设名称	基址
TWI0	0x40003000
TWI1	0x40004000

表 21-3: SAADC 相关寄存器

序号	寄存器名	偏移地址	功能描述
任务寄存器			
1	TASKS_STARTRX	0x000	启动 TWI 接收序列。
2	TASKS_STARTTX	0x008	启动 TWI 发送序列。
3	TASKS_STOP	0x014	停止 TWI 传输。
4	TASKS_SUSPEND	0x01C	挂起 TWI 事务。
5	TASKS_RESUME	0x020	恢复 TWI 事务。
事件寄存器			
1	EVENTS_STOPPED	0x104	TWI 已停止。
2	EVENTS_ERROR	0x124	TWI 错误。
3	EVENTS_SUSPENDED	0x148	在触发 SUSPEND 任务后发送了最后一个字节, TWI 已挂起。
4	EVENTS_RXDREADY	0x14C	TWI 接收序列已启动。
5	EVENTS_TXSTARTED	0x150	TWI 发送序列已启动。
6	EVENTS_LASTRX	0x15C	字节边界, 开始接收最后一个字节。
7	EVENTS_LASTTX	0x160	字节边界, 开始接收最后一个字节。
快捷方式寄存器			
1	SHORTS	0x200	快捷方式
通用寄存器			
1	INTEN	0x300	中断使能/禁止。
2	INTENSET	0x304	使能中断。
3	INTENCLR	0x308	禁止中断。
4	ERRORSRC	0x4C4	错误源。
5	ENABLE	0x500	使能或禁用 TWI。
6	PSEL.SCL	0x508	SCL 引脚配置。
7	PSEL.SDA	0x50C	SDA 引脚配置。
8	RXD	0x518	接收。
9	TXD	0x51C	发送。

10	FREQUENCY	0x524	TWI 频率，精度取决于所选的 HFCLK 时钟源。
11	RXD.PTR	0x534	接收数据指针。
12	RXD.MAXCNT	0x538	接收缓存的最大字节数。
13	RXD.AMOUNT	0x53C	最后一次传送的字节数。
14	RXD.LIST	0x540	接收 EasyDMA 列表类型
15	TXD.PTR	0x544	发送数据指针。
16	TXD.MAXCNT	0x548	发送缓存的最大字节数。
17	TXD.AMOUNT	0x54C	最后一次传送的字节数。
18	TXD.LIST	0x550	发送 EasyDMA 列表类型。
19	ADDRESS	0x588	TWI 传输中使用的地址。

■ SHORTS: 快捷方式寄存器

表 21-4: SHORTS 寄存器

位	Field	RW	复位值	描述
位 7	LASTTX_STARTRX	读/写	0	LASTTX 事件和 STARTRX 任务之间的快捷方式。 0: 禁止快捷方式。 1: 使能快捷方式。
位 8	LASTTX_SUSPEND	读/写	0	LASTTX 事件和 SUSPEND 任务之间的快捷方式。 0: 禁止快捷方式。 1: 使能快捷方式。
位 9	LASTTX_STOP	读/写	0	LASTTX 事件和 STOP 任务之间的快捷方式。 0: 禁止快捷方式。 1: 使能快捷方式。
位 10	LASTRX_STARTTX	读/写	0	LASTRX 事件和 STARTRX 任务之间的快捷方式。 0: 禁止快捷方式。 1: 使能快捷方式。
位 11	LASTRX_SUSPEND	读/写	0	LASTRX 事件和 SUSPEND 任务之间的快捷方式。

			0: 禁止快捷方式。 1: 使能快捷方式。
位 12	LASTRX_STOP	读/写 0	LASTRX 事件和 STOP 任务之间的快捷方式。 0: 禁止快捷方式。 1: 使能快捷方式。

■ INTEN: 中断使能/禁止寄存器

INTEN 寄存器用于使能/禁止中断。位的值写为 1 时使能对应的中断，写入 0 时禁止对应的中断。

表 21-5: INTEN 寄存器

位	Field	RW	复位值	描述
位 1	STOPPED	读/写	0	使能或禁止 STOPPED 事件中断。 0: 使能。 1: 禁止。
位 9	ERROR	读/写	0	使能或禁止 ERROR 事件中断。 0: 使能。 1: 禁止。
位 18	SUSPENDED	读/写	0	使能或禁止 SUSPENDED 事件中断。 0: 使能。 1: 禁止。
位 19	RXSTARTED	读/写	0	使能或禁止 RXSTARTED 事件中断。 0: 使能。 1: 禁止。
位 20	TXSTARTED	读/写	0	使能或禁止 TXSTARTED 事件中断。 0: 使能。 1: 禁止。
位 23	LASTRX	读/写	0	使能或禁止 LASTRX 事件中断。 0: 使能。 1: 禁止。
位 24	LASTTX	读/写	0	使能或禁止 LASTTX 事件中断。 0: 使能。 1: 禁止。

■ INTENSET：中断使能寄存器

INTENSET 寄存器用于使能中断。位的值写为 1 时使能对应的中断，写入 0 无效。注意 INTENSET 寄存器只能用来使能中断。

表 21-6: INTENSET 寄存器

位	Field	RW	复位值	描述
位 1	STOPPED	读/写	0	写“1”使能 STOPPED 事件中断，写“0”无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。
位 9	ERROR	读/写	0	写“1”使能 ERROR 事件中断，写“0”无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。
位 18	SUSPENDED	读/写	0	写“1”使能 SUSPENDED 事件中断，写“0”无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。
位 19	RXSTARTED	读/写	0	写“1”使能 RXSTARTED 事件中断，写“0”无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。
位 20	TXSTARTED	读/写	0	写“1”使能 TXSTARTED 事件中断，写“0”无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。
位 23	LASTRX	读/写	0	写“1”使能 LASTRX 事件中断，写“0”无效。 写，1: 使能。 读，0: 已禁止。 读，1: 已使能。

位 24	LASTTX	读/写 0	写“1”使能 LASTTX 事件中断，写“0”无效。 写，1：使能。 读，0：已禁止。 读，1：已使能。
------	--------	-------	---

■ INTENCLR：中断禁止寄存器

INTENCLR 寄存器用于禁止中断。位的值写为 1 时禁止对应的中断，写入 0 无效。注意 INTENCLR 寄存器只能用来禁止中断。

表 21-7: INTENCLR 寄存器

位	Field	RW	复位值	描述
位 1	STOPPED	读/写 0	写“1”禁止 STOPPED 事件中断，写“0”无效。 写，1：禁止。 读，0：已禁止。 读，1：已使能。	
位 9	ERROR	读/写 0	写“1”禁止 ERROR 事件中断，写“0”无效。 写，1：禁止。 读，0：已禁止。 读，1：已使能。	
位 18	SUSPENDED	读/写 0	写“1”禁止 SUSPENDED 事件中断，写“0”无效。 写，1：禁止。 读，0：已禁止。 读，1：已使能。	
位 19	RXSTARTED	读/写 0	写“1”禁止 RXSTARTED 事件中断，写“0”无效。 写，1：禁止。 读，0：已禁止。 读，1：已使能。	
位 20	TXSTARTED	读/写 0	写“1”禁止 TXSTARTED 事件中断，写“0”无效。 写，1：禁止。 读，0：已禁止。	

				读, 1: 已使能。
位 23	LASTRX	读/写 0		写“1”禁止 LASTRX 事件中断, 写“0”无效。
				写, 1: 禁止。
				读, 0: 已禁止。
				读, 1: 已使能。
位 24	LASTTX	读/写 0		写“1”禁止 LASTTX 事件中断, 写“0”无效。
				写, 1: 禁止。
				读, 0: 已禁止。
				读, 1: 已使能。

■ ERRORSRC: TWI 错误源寄存器

ERRORSRC 寄存器用于指示 TWI 通讯过程中的错误源。

表 21-8: ERRORSRC 寄存器

位	Field	RW	复位值	描述
位 0	OVERRUN	读/写 0		溢出错误。在将前一个字节传输到 RXD 缓冲区之前接收到一个新字节。(之前的 数据丢失)。 读, 0: 无溢出错误发送。 读, 1: 发生了溢出错误。
位 1	ANACK	读/写 0		发送地址后接收到 NACK (写 1 清除)。 读, 0: 没有产生错误。 读, 1: 产生错误。
位 2	DNACK	读/写 0		发送一字节数据后接收到 NACK (写 1 清除)。 读, 0: 没有产生错误。 读, 1: 产生错误。

■ ENABLE: TWI 使能/禁止寄存器

ENABLE 寄存器是用于使能或禁用 TWI。

表 21-9: ENABLE 寄存器

位	Field	RW	复位值	描述
位 3~位 1	ENABLE	读/写 0		使能或禁止 TWI。 0: 禁止 TWI。

	5: 使能 TWI。
--	------------

■ PSEL.SCL: TWI SCL 连接引脚配置寄存器

PSEL.SCL 寄存器用于配置 TWI 的 SCL 连接的引脚, PORT 配置端口, PIN 配置引脚, 如配置 SCL 连接到引脚 P1.05, PORT 应设置为 1, PIN 应设置为 5。

表 21-10: PSEL.SCL 寄存器

位	Field	RW	复位值	描述
位 4~位 0	PIN	读/写	11111	引脚编号, 0~31。
位 5	PORT	读/写	1	端口号, 0 或 1。
位 31	CONNECT	读/写	1	1: 不连接。 0: 连接。

■ PSEL.SDA: TWI SDA 连接引脚配置寄存器

PSEL.SDA 寄存器用于配置 TWI 的 SDA 连接的引脚, PORT 配置端口, PIN 配置引脚, 如配置 SDA 连接到引脚 P1.05, PORT 应设置为 1, PIN 应设置为 5。

表 21-11: PSEL.SDA 寄存器

位	Field	RW	复位值	描述
位 4~位 0	PIN	读/写	11111	引脚编号, 0~31。
位 5	PORT	读/写	1	端口号, 0 或 1。
位 31	CONNECT	读/写	1	1: 不连接。 0: 连接。

■ FREQUENCY: TWI 频率配置寄存器

TWI 频率配置寄存器, 精度取决于所选的 HFCLK 时钟源。

表 21-12: FREQUENCY 寄存器

位	Field	RW	复位值	描述
位 31~位 0	FREQUENCY	读/写	0x04000000	TWI 主机时钟频率。 0x01980000: 100 kbps。 0x04000000: 200 kbps。 0x06400000: 400 kbps。

■ RXD.PTR: 接收数据指针寄存器

表 21-13: RXD.PTR 寄存器

位	Field	RW	复位值	描述
---	-------	----	-----	----

位 7~位 0	PTR	读/写 0	数据指针。
---------	-----	-------	-------

■ RXD.MAXCNT: 接收缓存的最大字节数配置寄存器

表 21-14: RXD.MAXCNT 寄存器

位	Field	RW	复位值	描述
位 15~位 0	MAXCNT	读/写 0	接收缓存的最大字节数。	

■ RXD.AMOUNT: 最后一次传送的字节数查询寄存器

表 21-15: RXD.AMOUNT 寄存器

位	Field	RW	复位值	描述
位 7~位 0	AMOUNT	只读 0	最后一次传送的字节数, 在 NACK 错误的情况下, 包括 NACK 的字节。	

■ RXD.LIST: 接收 EasyDMA 表类型配置寄存器

表 21-16: RXD.LIST 寄存器

位	Field	RW	复位值	描述
位 2~位 0	LIST	读/写 0	表类型。 0: 禁止 EasyDMA 表。 1: 使能 EasyDMA 表。	

■ TXD.PTR: 发送数据指针寄存器

表 21-17: TXD.PTR 寄存器

位	Field	RW	复位值	描述
位 7~位 0	PTR	读/写 0	数据指针。	

■ TXD.MAXCNT: 接收缓存的最大字节数配置寄存器

表 21-18: TXD.MAXCNT 寄存器

位	Field	RW	复位值	描述
位 7~位 0	MAXCNT	读/写 0	发送缓存的最大字节数。	

■ RXD.AMOUNT: 最后一次传送的字节数查询寄存器

表 21-19: RXD.AMOUNT 寄存器

位	Field	RW	复位值	描述
位 7~位 0	AMOUNT	只读 0	最后一次传送的字节数, 在 NACK 错误的	

情况下，包括 NACK 的字节。

■ TXD.LIST: 发送 EasyDMA 表类型配置寄存器

表 21-20: TXD.LIST 寄存器

位	Field	RW	复位值	描述
位 2~位 0	LIST	读/写	0	表类型。 0: 禁止 EasyDMA 表。 1: 使能 EasyDMA 表。

■ ADDRESS: 地址寄存器

表 21-21: ADDRESS 寄存器

位	Field	RW	复位值	描述
位 6~位 0	ADDRESS	读/写	0	TWI 传输中使用的地址。

5. 硬件电路设计

IK-52840DK 开发板上设计了 eeprom 存储器(AT24C02)，硬件电路如下图所示。AT24C02 通过 I2C 接口和 nRF52840 连接，I2C 接口为开漏输出，因此 SCL 和 SDA 线上需要增加上拉电阻，这里我们使用的上拉电阻阻值为 4.7K。用于 I2C 连接的 P0.17 和 P0.19 可以通过跳线连接或断开，当我们不使用 eeprom 存储器时可以拔掉跳线帽，将 P0.17 和 P0.19 用作其他用途。

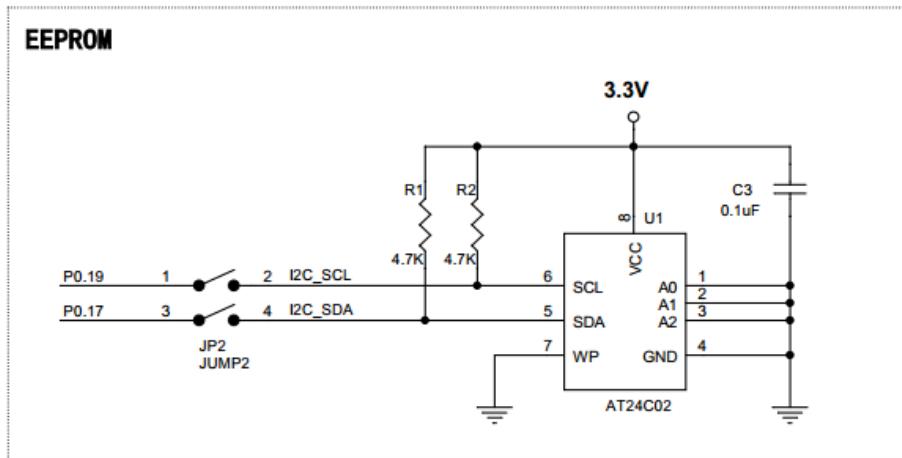


图 21-10: AT24C02 电路

AT24C02 的存储空间共有 2K 位 (256 个字节)，页面大小为 8 个字节，共 32 个页面。AT24C02 的地址的低 3 位可以通过引脚 A2 A1 A0 配置，本电路中引脚 A2 A1 A0 均连接到 GND，因此，地址的低 3 位均为 0。(AT24C02 的地址配置详见 6.2.4 代码编写章节)

eeprom 存储器 (AT24C02) 占用的 nRF52840 的引脚如下表：

表 21-22: TWI 连接 AT24C02 引脚分配

名称	引脚	说明
SCL	P0.19	独立 IO
SDA	P0.17	独立 IO

◆ 注：独立 IO 表示开发板没有其他的电路使用这个 IO。

6.软件设计

6.1. 库函数的应用

TWI 的应用步骤如下图所示，首先要定义 TWI 驱动程序实例，驱动程序实例对应具体的硬件 TWI 外设（TWI0 和 TWI1），驱动程序实例决定了我们使用的是 TWI0 还是 TWI1。接着初始化配置 TWI 连接的引脚和速率等参数，注册事件句柄（非堵塞模式下），初始化后使能 TWI，之后就可以使用 TWI 进行传输数据。



图 21-11: TWI 应用步骤

6.1.1. 定义 TWI 驱动程序实例

TWI 驱动程序实例使用 nrf_drv_twi_t 结构体定义，该结构体描述了具体的 TWI 外设，当我们定义了 nrf_drv_twi_t 类型的变量并对其赋值后，该变量就对应了一个具体的硬件 TWI 外设。

定义驱动程序实例代码如下，初始化宏 NRF_DRV_TWI_INSTANCE 的输入参数对应 TWI 外设的编号，即如果我们定义 TWI0 的驱动程序实例，TWI_INSTANCE_ID 的值设置为 0，定义 TWI1 的驱动程序实例，TWI_INSTANCE_ID 的值设置为 1。驱动程序实例定义后，我们即可通过该驱动程序实例访问对应的 TWI。

代码清单：定义 TWI 驱动程序实例

```

1. //TWI 驱动程序实例 ID, ID 和外设编号对应, 0:TWI0  1:TWI1
2. #define TWI_INSTANCE_ID      0
3. //定义 TWI 驱动程序实例
4. static const nrf_drv_twi_t 驱动实例名称 = NRF_DRV_TWI_INSTANCE(TWI_INSTANCE_ID);

```

6.1.2. 初始化 TWI

TWI 初始化的库函数是 nrf_drv_twi_init() 函数，该函数同时也配置了 TWI 是否使用阻塞模式。

- 应用程序提供事件句柄：TWI 工作于非阻塞模式。
- 应用程序不提供事件句柄（event_handler 设置为 NULL）：TWI 工作于阻塞模式。

表 21-23: nrf_drv_twi_init() 函数

函数原型	<pre>ret_code_t nrf_drv_twi_init (nrf_drv_twi_t const * p_instance, nrf_drv_twi_config_t const * p_config, nrf_drv_twi_evt_handler_t event_handler, void * p_context)</pre>
函数功能	初始化 TWI。
参数	<p>[in] <code>p_instance</code>: 指向驱动程序实例结构体。</p> <p>[in] <code>p_config</code>: 指向初始化配置结构体。</p> <p>[in] <code>event_handler</code>: 事件句柄，如果设置为 NULL，则使能 TWI 阻塞模式。</p> <p>[in] <code>p_context</code>: 指向传递给事件句柄的 Context。</p>
返回值	<p><code>NRF_SUCCESS</code>: 初始化成功。</p> <p><code>NRF_ERROR_INVALID_STATE</code>: 驱动处于无效的状态。</p> <p><code>NRF_ERROR_BUSY</code>: 使用了 ID 相同的其他外设。这种情况仅在 <code>PERIPHERAL_RESOURCE_SHARING_ENABLED</code> 设置为 1 而没有设置为 0 的时候出现。</p>

■ 初始化结构体

调用 nrf_drv_twi_init() 函数初始化 TWI 时需要提供初始化结构体变量作为函数的参数，初始化结构体包含了 TWI 初始化时所需要配置的参数，其声明如下：

代码清单：初始化配置结构体 nrf_drv_twi_config_t 声明

```
1. typedef struct
2. {
3.     uint32_t           scl;           //SCL 引脚
4.     uint32_t           sda;           //SDA 引脚
5.     nrf_drv_twi_frequency_t frequency; //TWI 频率
6.     uint8_t            interrupt_priority; //中断优先级
7.     //初始化期间，控制 TWI 的时钟线 SCL 发出 9 个时钟脉冲
8.     bool               clear_bus_init;
9.     bool               hold_bus_uninit; //反初始化后保持 PIN 上拉
10. } nrf_drv_twi_config_t;
```

对于所用 TWI 的 SCL、SDA 信号连接的引脚以及 TWI 主时钟频率都是通过初始化结构体配置的，下面的代码定义了名称为 twi_my_config 的 TWI 配置结构体变量并进行了初始化。

代码清单：定义并初始化 TWI 配置结构体

```

1. const nrf_drv_twi_config_t twi_my_config = {
2.     .scl          = TWI_SCL_M, // 定义 TWI SCL 引脚
3.     .sda          = TWI_SDA_M, // 定义 TWI SDA 引脚
4.     .frequency    = NRF_DRV_TWI_FREQ_400K, // TWI 速率
5.     .interrupt_priority = APP_IRQ_PRIORITY_HIGH, // TWI 优先级
6.     .clear_bus_init = false
7. };

```

■ 事件句柄

如果我们要 TWI 工作于非阻塞模式，就需要提供事件回调函数，并在初始化时将事件句柄作为 nrf_drv_twi_init() 函数的参数传递给函数，由初始化函数完成注册。TWI 事件回调函数的编写格式如下，我们编写 TWI 事件回调函数时需要遵循该格式。

代码清单：TWI 事件回调函数编写格式

```

1. void twi_handler(nrf_drv_twi_evt_t const * p_event, void * p_context)
2. {
3.     // 判断 TWI 事件类型
4.     switch (p_event->type)
5.     {
6.         // 传输完成事件
7.         case NRF_DRV_TWI_EVT_DONE:
8.             // 功能代码
9.             break;
10.        // 错误事件：发送地址后接收到 NACK
11.        case NRF_DRV_TWI_EVT_ADDRESS_NACK:
12.            // 功能代码
13.            break;
14.        // 错误事件：发送数据后接收到 NACK
15.        case NRF_DRV_TWI_EVT_DATA_NACK:
16.            // 功能代码
17.            break;
18.        default:
19.            break;
20.    }
21. }

```

TWI 驱动程序提供了如下所示的三个事件用来通过应用程序 TWI 总线的运行情况，我们编写 TWI 事件回调函数时可以根据需求处理各个事件。

代码清单：TWI 事件类型

```

1. typedef enum
2. {
3.     NRF_DRV_TWI_EVT_DONE,           //传输完成事件
4.     NRF_DRV_TWI_EVT_ADDRESS_NACK, //错误事件：发送数据后接收到 NACK
5.     NRF_DRV_TWI_EVT_DATA_NACK    //错误事件：发送数据后接收到 NACK
6. } nrf_drv_twi_evt_type_t;

```

6.1.3. 使能 TWI

TWI 初始化完成之后，还需要调用 `nrf_drv_twi_enable()` 函数使能该 TWI，使能后才可以进行数据传输，`nrf_drv_twi_enable()` 函数原型如下。

表 21-24: `nrf_drv_twi_enable()` 函数

函数原型	<code>_STATIC_INLINE void nrf_drv_twi_enable (nrf_drv_twi_t const * p_instance)</code>
函数功能	使能 TWI 实例。
参数	[in] <code>p_instance</code> : 指向 TWI 驱动程序实例结构体。
返回值	无。

6.1.4. 数据传输

TWI 驱动程序提供了两个单独的函数 `nrf_drv_twi_tx()` 函数和 `nrf_drv_twi_rx()` 函数，分别用于完成数据的发送和接收，函数原型如下表所示。使用 tx 和 rx 函数时，尤其要注意从机地址 address，函数接收的是 7 位地址，函数内部会自己加上读写位，因此，如果某个 I2C 接口设备提供的是 8 位地址，我们需要提取出 7 位地址赋值给 address，也就是去掉 8 位地址的最低位（R/W 位）。

■ TWI 发送函数原型

表 21-25: `nrf_drv_twi_tx()` 函数

函数原型	<code>_STATIC_INLINE ret_code_t nrf_drv_twi_tx (nrf_drv_twi_t const * p_instance, uint8_t address, uint8_t const * p_data, uint8_t length, bool no_stop)</code>
函数功能	向 TWI 从机发送数据，发生错误时将停止传输。如果传输正在进行，则

	该函数返回错误代码 NRF_ERROR_BUSY。
参 数	<p>[in] p_instance: 指向 TWI 驱动程序实例结构体。</p> <p>[in] address: 指定的从机地址（7 位 LSB）。</p> <p>[in] p_data: 指向传输数据缓存。</p> <p>[in] length: 发送的字节数。</p> <p>[in] no_stop: 如果置位，成功传输后总线上不会生成停止条件（允许在下一次传输中重复启动）。</p>
返回值	<p>NRF_SUCCESS: 发送成功。</p> <p>NRF_ERROR_BUSY: 驱动程序尚未准备好进行新的传输。</p> <p>NRF_ERROR_INTERNAL: 硬件检测到错误。</p> <p>NRF_ERROR_INVALID_ADDR: 使用了 EasyDMA，但是缓存地址没有位于 RAM 空间。</p> <p>NRF_ERROR_DRV_TWI_ERR_ANACK: 在轮询模式下发送地址后收到 NAC。</p> <p>NRF_ERROR_DRV_TWI_ERR_DNACK: 在轮询模式下发送数据后收到 NACK。</p>

■ TWI 接收函数原型

表 21-26: nrf_drv_twi_rx()函数

函数原型	<pre>__STATIC_INLINE ret_code_t nrf_drv_twi_rx (nrf_drv_twi_t const * p_instance, uint8_t address, uint8_t * p_data, uint8_t length)</pre>
函数功能	从 TWI 从机读取数据，发生错误时将停止传输。如果传输正在进行，则该函数返回错误代码 NRF_ERROR_BUSY。
参 数	<p>[in] p_instance: 指向 TWI 驱动程序实例结构体。</p> <p>[in] address: 指定的从机地址（7 位 LSB）。</p> <p>[in] p_data: 指向接收数据缓存。</p> <p>[in] length: 读取的字节数。</p>
返回值	<p>NRF_SUCCESS: 发送成功。</p> <p>NRF_ERROR_BUSY: 驱动程序尚未准备好进行新的传输。</p> <p>NRF_ERROR_INTERNAL: 硬件检测到错误。</p> <p>NRF_ERROR_DRV_TWI_ERR_OVERRUN: 接收数据未及时读取，被新</p>

	<p>接收的数据覆盖。</p> <p>NRF_ERROR_DRV_TWI_ERR_ANACK: 在轮询模式下发送地址后收到NACK。</p> <p>NRF_ERROR_DRV_TWI_ERR_DNACK: 在轮询模式下发送数据后收到NACK。</p>
--	---

6.2. eeprom 读写（TWI0 非阻塞）实验

本实验在“实验 10-1：串口数据收发”的基础上修改。通过 TWI 读写 eeprom 存储器 AT24C02，TWI 主频率使用 400K，本例中实现以下几个功能。

- 单个字节写入：向指定地址写入单个字节数据。
- 按页写：向指定页面连续写入不大于页面长度的数据。
- 批量写：向指定地址连续写入指定长度数据的功能，该功能实现了跨页写入。
- 批量读：从指定地址连续读取指定长度数据，并将读取的数据通过串口输出。

❖ 注：本节对应的实验源码是：“实验 21-1：TWI(I2C)读写 eeprom-TWI0 非阻塞模式”。

6.2.1. 添加需要的文件

TWI 需要加入的文件如下表所示。

表 21-27： I2C 需要加入的文件

文件名	SDK 中的目录	描述
nrf_drv_twi.c	...\\integration\\nrfx\\legacy	旧 TWI 驱动程序。
nrfx_twi.c	...\\modules\\nrfx\\drivers\\src	新 TWI 驱动程序
nrfx_twim.c	...\\modules\\nrfx\\drivers\\src	新 TWI 主机驱动程序

6.2.2. 头文件引用和路径设置

7. 需要引用的头文件

因为在“main.c”文件中使用了 TWI 和 AT24C02 的驱动程序，所以“main.c”文件需要引用下面的头文件。

```
#include "nrf_drv_twi.h"
#include "at24c02.h"
```

8. 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

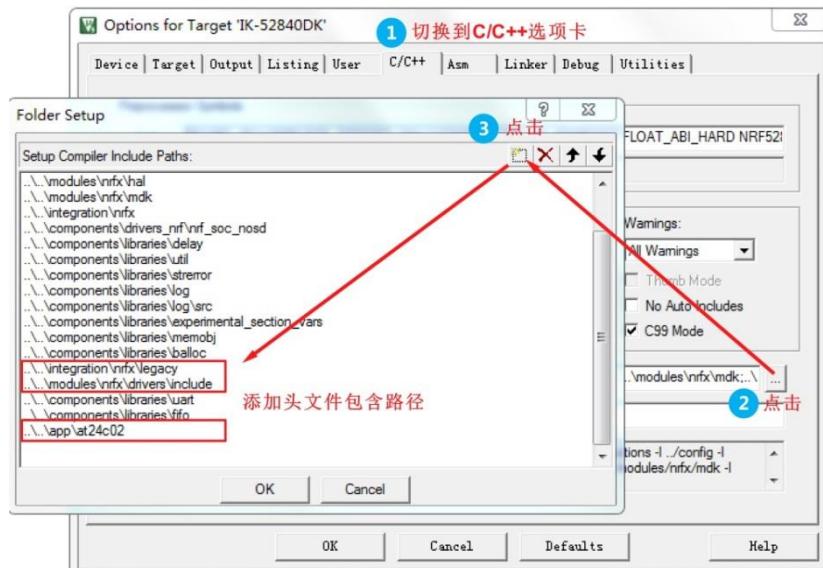


图 21-12：添加头文件包含路径

本例需要添加的头文件路径如下表：

表 21-28：头文件包含路径

序号	路径
1	..\\..\\modules\\nrfx\\drivers\\include
2	..\\..\\integration\\nrfx\\legacy
3	..\\..\\app\\at24c02

6.2.3. 工程配置

打开“sdk_config.h”文件，加入 TWI 需要的配置，如下图所示，编辑内容时切换到“Text Editor”进行编辑（编辑的内容参考实验的源码，因此代码很长，此处不贴出代码，参见“sdk_config.h”文件的（145~453）行），编辑完成后，切换到“Configuration Wizard”，即可观察到配置的具体项目：

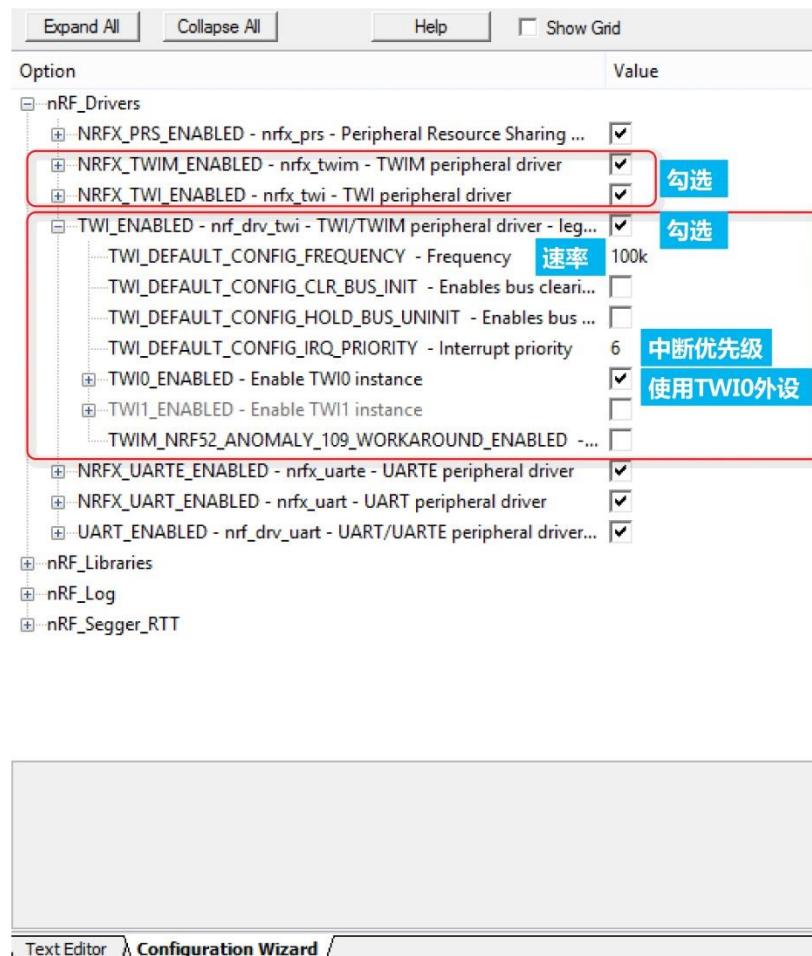


图 21-13: 工程配置

因为本例中使用的是 TWI0，因此需要勾选“TWI0_ENABLED”，另外，本例在 TWI 初始化函数重写了配置结构体的各个参数，所以这里配置的 TWI 主时钟为 100K 是不起作用的。

6.2.4. 代码编写

按照 TWI 的应用步骤，首先应定义和初始化 TWI 驱动程序实例，本例中使用 TWI0，因此 TWI 驱动程序的索引应为 0，程序清单如下。

代码清单：定义并初始化 TWI 驱动程序实例，使用 TWI0

```

1. //TWI 驱动程序实例 ID, ID 和外设编号对应, 0:TWI0  1:TWI1
2. #define TWI_INSTANCE_ID      0
3.
4. //定义 TWI 驱动程序实例，名称为 m_twi
5. static const nrf_drv_twi_t m_twi = NRF_DRV_TWI_INSTANCE(TWI_INSTANCE_ID);

```

接着初始化配置和使能 TWI，一般地，为了方便，我们会把初始化和使能封装到同一个函数里面，代码清单如下，在这里我们配置了 TWI 相关的参数和工作模式（阻塞模式和非阻塞模式）。

代码清单：初始化和使能 TWI 实例

```
1. #define TWI_SCL_M          19      //I2C SCL 引脚
2. #define TWI_SDA_M          17      //I2C SDA 引脚
3.
4. void twi_at24c02_init(void)
5. {
6.     ret_code_t err_code;
7.     //定义并初始化 TWI 配置结构体
8.     const nrf_drv_twi_config_t twi_24c02_config = {
9.         .scl          = TWI_SCL_M,  //定义 TWI SCL 引脚
10.        .sda         = TWI_SDA_M,  //定义 TWI SDA 引脚
11.        .frequency    = NRF_DRV_TWI_FREQ_400K, //TWI 速率
12.        .interrupt_priority = APP_IRQ_PRIORITY_HIGH, //TWI 优先级
13.        .clear_bus_init   = false//初始化期间不发送 9 个 SCL 时钟
14.    };
15.    //初始化 TWI
16.    err_code = nrf_drv_twi_init(&m_twi, &twi_24c02_config, twi_handler, NULL);
17.    //检查返回的错误代码
18.    APP_ERROR_CHECK(err_code);
19.    //使能 TWI
20.    nrf_drv_twi_enable(&m_twi);
21.}
```

本例中使用的是非阻塞工作模式，因此需要提供事件回调函数，同时定义一个变量 m_xfer_done 用来标志 TWI 传输是否完成。

代码清单：TWI 事件回调函数

```
1. //TWI 传输完成标志
2. static volatile bool m_xfer_done = false;
3.
4. //TWI 事件处理函数
5. void twi_handler(nrf_drv_twi_evt_t const * p_event, void * p_context)
6. {
7.     //判断 TWI 事件类型
8.     switch (p_event->type)
9.     {
10.         //传输完成事件
11.         case NRF_DRV_TWI_EVT_DONE:
12.             m_xfer_done = true; //置位传输完成标志
13.             break;
14.         default:
15.             break;
16.     }
17. }
```

■ AT24C02 地址的确定

查询 AT24C02 数据手册可知其地址高 4 位固定为“1010”，如下图所示，紧跟着的 3 个位由芯片的引脚 A2、A1 和 A0 的电平确定，最低位为读写位。开发板上 AT24C02 的硬件电路中将引脚 A2、A1 和 A0 连接到了 GND，因此 A2、A1 和 A0 均为 0，由此可提取出 7 位地址为 0x50，一般为了更直观，我们会将地址写作：(0xA0>>1)。

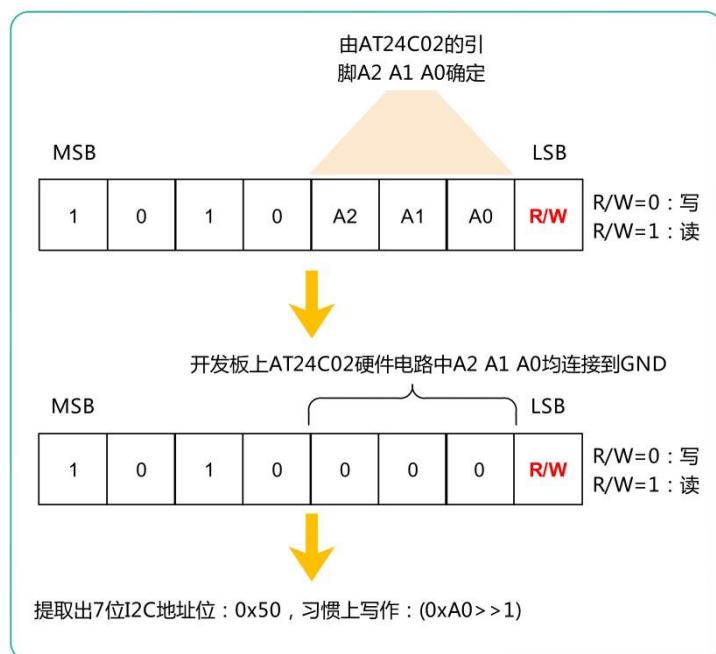


图 21-14: AT24C02 地址的确定

代码中，AT24C02 地址定义如下：

代码清单：定义 AT24C02 地址

```
1. #define AT24C02_ADDRESS      (0xA0>>1) //AT24C02 地址
```

■ 单字节写入：向指定地址写入单个字节数据

AT24C02 将单个字节写入到指定地址的时序如下图所示，首先产生起始条件，紧跟着发送 7 位地址 + 0（写），之后发送写入数据的地址和数据，最后产生停止条件。

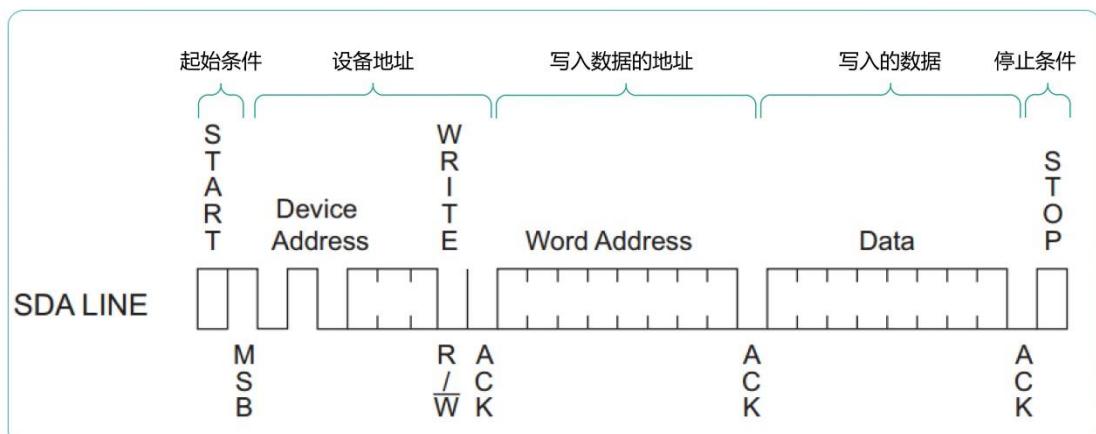


图 21-15: AT24C02 写入单个字节数据时序

根据时序，即可写出“写入单个字节数据”的函数，代码清单如下，这里面有几个重点需要关注，这些注意事项在按页写和批量写时同样需要注意。

- 因为时序要求写操作后产生停止条件，因此调用 nrf_drv_twi_tx() 函数时，参数“no_stop”必须为 false，即产生停止条件。
- 本例 TWI 的工作模式是非阻塞模式，是异步的，所以调用 nrf_drv_twi_tx() 函数后，要等待 TWI 传输完成，代码中通过查询 m_xfer_done 标志判断 TW 传输是否完成。
- TWI 发送完成后，仅表示 AT24C02 接收到了数据，并不表示 AT24C02 已经完成了数据的写入，AT24C02 的写入周期是 5ms，因此 TWI 传输完成后要延时 5ms 以确保 AT24C02 正确写入数据。

代码清单：向指定地址写入单个字节数据

```
1. ****
2. * 功 能 : 向 AT24C02 指定的地址写入一个字节数据
3. * 参 数 : WriteAddr[in]: 地址
4. *          :dat[in]: 写入的数据
5. * 返回值 : NRF_SUCCESS:写数据成功
6. ****
7. ret_code_t AT24C02_write_byte(uint8_t WriteAddr,uint8_t dat)
8. {
9.     ret_code_t err_code;
10.    uint8_t tx_buf[AT24C02_ADDRESS_LEN+1];
11.    //检查写入数据的地址是否合法
12.    if (WriteAddr > AT24C02_ENDADDR)
13.    {
14.        return NRF_ERROR_INVALID_ADDR;;
15.    }
16.    //准备写入的数据
17.    tx_buf[0] = WriteAddr;
18.    tx_buf[1] = dat;
19.    //TWI 传输完成标志设置为 false
20.    m_xfer_done = false;
21.    //写入数据
22.    err_code = nrf_drv_twi_tx(&m_twi, AT24C02_ADDRESS, tx_buf, AT24C02_ADDRESS_L
23.                                EN+1, false);
24.    APP_ERROR_CHECK(err_code);
25.    //等待 TWI 总线传输完成
26.    while (m_xfer_done == false);
27.    //延时确保 AT24C02 将接收的数据写入到 eeprom
28.    AT24C02_DELAY;
29.    return err_code;
30.}
```

■ 按页写入：向指定页面连续写入不大于页面长度的数据

AT24C02 按页写入的时序如下图所示，首先产生起始条件，紧跟着发送 7 位地址 + 0 (写)，之后发送 n 个数据的地址和数据，最后产生停止条件。

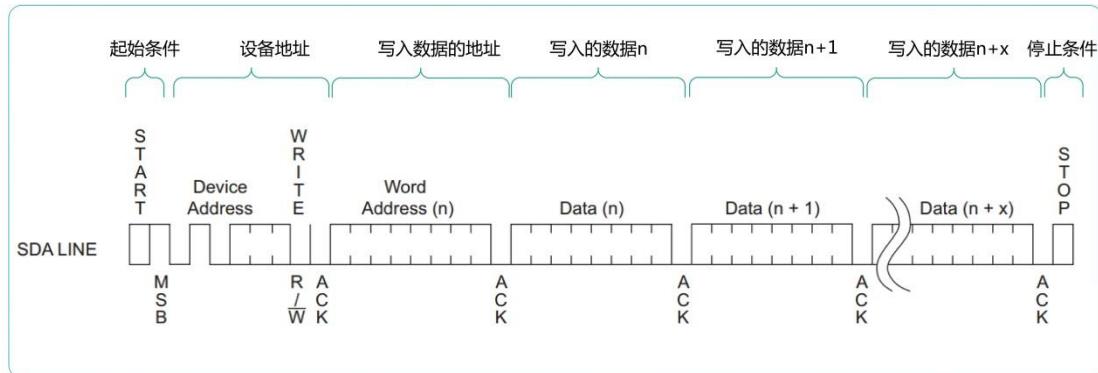


图 21-16: AT24C02 按页写入时序

按页写入除了遵循“单个字节写入”的注意事项外，还需要注意 AT24C02 按页写时的跨页问题，如果地址跨页，则写指针会返回到当前页的起始地址，这一点非常重要，接下来我们来分析按页写的几种情况。

AT24C02 的容量为 256 个字节，页面大小为 8 个字节，因此 AT24C02 被分为 32 个页面，第一个页面地址位 0~7，第二个页面地址位 8~15，以此类推。如下图所示，按页写时如果地址没有超过当前页面，写入正确。

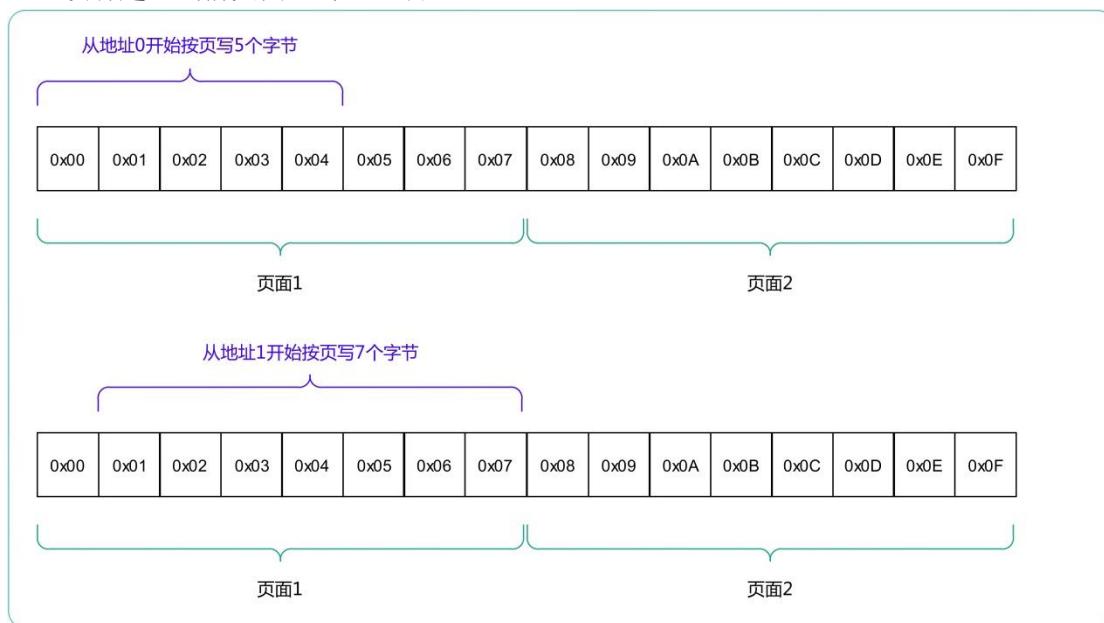


图 21-17: 地址在同一个页面

按页写时如果地址跨页，会出现如下图所示的情形：我们期望从地址 0x04 开始连续写入“ABCDEF”6 个数据，但是实际写时，因为写地址增加到 0x07 后自动复位到 0x00，所以实际写入的地址 0x04~0x07 写入“ABCD”4 个数据，地址 0x00~0x01 写入“EF”2 个数据。

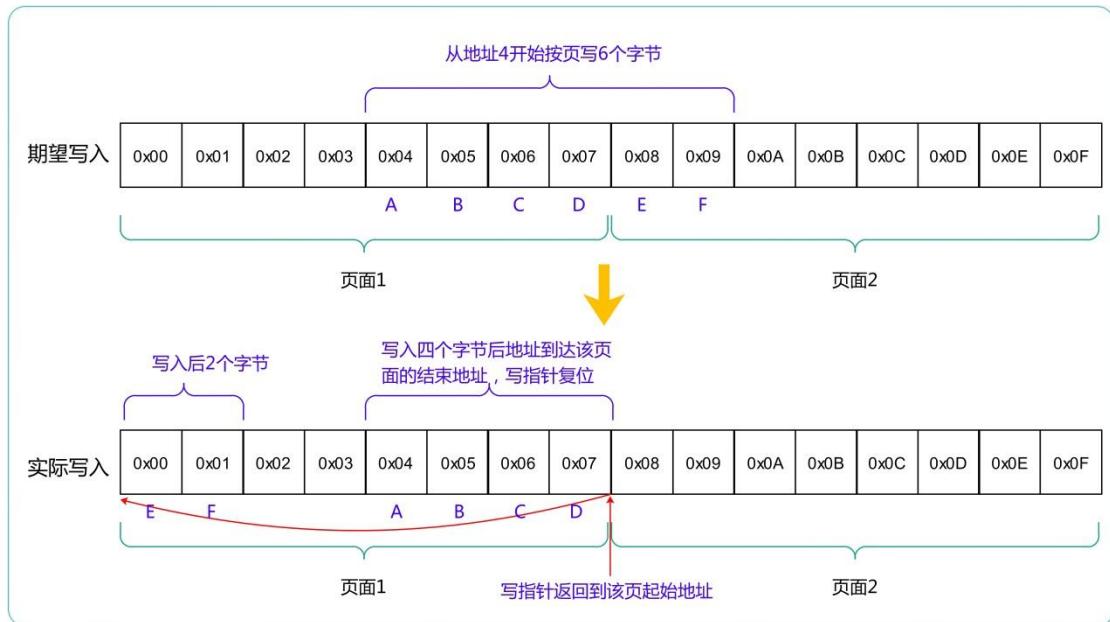


图 21-18：跨页后写指针复位

综上所述，我们编写按页写入函数，代码清单如下，按页写入函数可实现连续写，不需要每个字节都发起一次写流程，这会节省操作时间，此处我们编写的按页写入函数适合符合以下条件的情况下使用。

- 数据写入地址从页面起始地址开始。
- 数据写入长度不超过页面的大小，即不超过 8 个字节。

代码清单：按页写入

```

1. ****
2. * 功 能 : 向 AT24C02 指定的页面写入数据, 数据长度不能超过一个页面的大小 8 个字节。
3. * 参 数 : page[in]: 页面, 1~32
4. *          : pdata[in]: 指向待写入的数据缓存
5. *          : size[in]: 写入的数据长度, 不能超过 1 个页面的大小 8 个字节
6. * 返回值 : NRF_SUCCESS:写页面成功
7. ****
8. ret_code_t AT24C02_write_page(uint8_t page,uint8_t const * pdata,uint8_t size)
9. {
10.     ret_code_t err_code;
11.     uint8_t tx_buf[AT24C02_ADDRESS_LEN+AT24C02_PAGESIZE];
12.     //检查写入的数据长度是否合法, 写入长度不能超过页面的大小
13.     if (size > AT24C02_PAGESIZE)
14.     {
15.         return NRF_ERROR_INVALID_LENGTH;
16.     }
17.     //检查写入的页面是否合法, 页面定义为 1~32, 不在这个范围判定为无效
18.     if ((page == 0)|| (page > AT24C02_PAGENUM))
19.     {
20.         return NRF_ERROR_INVALID_ADDR;;

```

```
21.    }
22.    //准备写入的数据
23.    tx_buf[0] = (page-1)*AT24C02_PAGESIZE;
24.    memcpy(tx_buf+AT24C02_ADDRESS_LEN, pdata, size);
25.    //TWI 传输完成标志设置为 false
26.    m_xfer_done = false;
27.    //写入数据
28.    err_code = nrf_drv_twi_tx(&m_twi, AT24C02_ADDRESS, tx_buf, size+AT24C02_ADDR
29.                                ESS_LEN, false);
30.    //检查返回的错误代码
31.    APP_ERROR_CHECK(err_code);
32.    //等待 TWI 总线传输完成
33.    while (m_xfer_done == false);
34.    //延时确保 AT24C02 将接收的数据写入到 eeprom
35.    AT24C02_DELAY;
36.    return err_code;
37. }
```

■ 批量写入：向指定地址连续写入指定长度的数据

当写入的数据较多时，可以通过循环执行单个字节写入操作将数据写入到 eeprom，这种方法编写程序简单，但是缺点是每个字节都需要执行一次写操作流程（起始条件→7位地址 + 0（写）→写入数据的地址→数据→停止条件），会花费更多的时间。

更好的方法是使用按页写入的方式将数据写入到 eeprom，程序中将处于同一页面的数据通过按页写入的方式一次写入。这种方式的重点是处理地址跨页，程序中会按照写入的地址逐个取出待写入的数据，每次跨页前执行一次数据写入操作。

代码清单：向指定地址连续写入指定长度的数据

```
1. ****
2. * 功 能 : 向 AT24C02 指定的起始地址批量写入数据，函数内部实现了跨页写,
3. *          : 函数会检查 AT24C02 的空间是否能够容纳写入的数据。
4. * 参 数 : WriteAddr[in]: 写入数据的起始地址
5. *          : p_buf[in]: 指向待写入的数据缓存
6. *          : size[in]: 写入的数据长度
7. * 返回值 : NRF_SUCCESS:写数据成功
8. ****
9. ret_code_t AT24C02_write_buf(uint8_t WriteAddr,uint8_t *p_buf,uint8_t size)
10. {
11.     ret_code_t err_code;
12.     uint8_t current_addr = WriteAddr;
13.     uint8_t sendlen = 0;
14.     uint8_t tx_buf[AT24C02_ADDRESS_LEN+AT24C02_PAGESIZE];
15.     //保存写入数据的起始地址
16.     tx_buf[0] = WriteAddr;
```

```
17.  
18. //AT24C02 剩余空间不够存放需要写入的数据, 返回: 长度无效的错误代码  
19. if ((AT24C02_ENDADDR-WriteAddr) < size)  
20. {  
21.     return NRF_ERROR_INVALID_LENGTH;  
22. }  
23. //连续写入数据, 如果跨页, 重新启动写流程  
24. while(size--)  
25. {  
26.     //到达页面的起始地址  
27.     if((current_addr%AT24C02_PAGESIZE) == 0)  
28.     {  
29.         //到达页面起始地址并且发送长度不等于 0, 表示即将跨页  
30.         if(sendlen != 0)  
31.         {  
32.             //TWI 传输完成标志设置为 false  
33.             m_xfer_done = false;  
34.             //执行一次写入操作  
35.             err_code = nrf_drv_twi_tx(&m_twi, AT24C02_ADDRESS, tx_buf, sendlen+  
36.                                         AT24C02_ADDRESS_LEN, false);  
37.             APP_ERROR_CHECK(err_code);  
38.             //等待 TWI 总线传输完成  
39.             while (m_xfer_done == false);  
40.             //清零发送长度  
41.             sendlen = 0;  
42.             //保存写入数据的起始地址  
43.             tx_buf[0] = current_addr;  
44.             //延时确保 AT24C02 将接收的数据写入到 eeprom  
45.             AT24C02_DELAY;  
46.         }  
47.         //数据保存到发送缓存 tx_buf  
48.         tx_buf[AT24C02_ADDRESS_LEN+sendlen++] = *(p_buf++);  
49.     }  
50.     else  
51.     {  
52.         //数据保存到发送缓存 tx_buf  
53.         tx_buf[AT24C02_ADDRESS_LEN+sendlen++] = *(p_buf++);  
54.         //写入到最后的页面的数据读取完成  
55.         if(size==0)  
56.         {  
57.             //TWI 传输完成标志设置为 false  
58.             m_xfer_done = false;  
59.             //执行一次写入操作  
60.             err_code = nrf_drv_twi_tx(&m_twi, AT24C02_ADDRESS, tx_buf, sendlen
```

```

61.                     +AT24C02_ADDRESS_LEN, false);
62.             APP_ERROR_CHECK(err_code);
63.             //等待 TWI 总线传输完成
64.             while (m_xfer_done == false);
65.             //清零发送长度
66.             sendlen = 0;
67.             //延时确保 AT24C02 将接收的数据写入到 eeprom
68.             AT24C02_DELAY;
69.         }
70.     }
71.     //地址加 1
72.     current_addr++;
73. }
74. return err_code;
75.}

```

■ 批量读取：从指定地址连续读取指定长度数据

AT24C02 从指定地址连续读数据的时序如下图所示，连续读相对于写要简单，因为不用考虑跨页的问题，连续读时每读出一个字节数据地址指针会自动加 1。

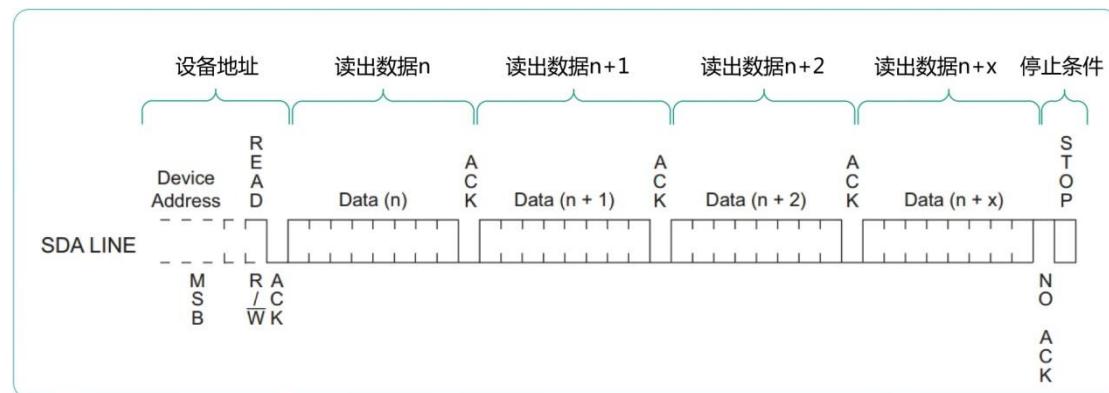


图 21-19: AT24C02 连续读时序

代码清单：从指定地址连续读取指定长度数据

```

1. ****
2. * 功 能 : 从指定的起始地址连续读出指定长度的数据
3. * 参 数 : WriteAddr[in]: 读出数据的起始地址
4. *       : p_buf[in]: 指向保存读出数据的缓存
5. *       : size[in]: 读出的数据长度
6. * 返回值 : NRF_SUCCESS:读数据成功
7. ****
8. ret_code_t AT24C02_read_buf(uint8_t ReadAddr,uint8_t *p_buf,uint8_t size)
9. {
10.     ret_code_t err_code;
11.     m_xfer_done = false;

```

```
12.     //读数据的长度已经超出了 AT24C02 的地址范围，返回：长度无效的错误代码
13.     if ((AT24C02_ENDADDR-ReadAddr) < size)
14.     {
15.         return NRF_ERROR_INVALID_LENGTH;
16.     }
17.     uint8_t tx_buf[1];
18.     tx_buf[0] = ReadAddr;
19.     //TWI 传输完成标志设置为 false
20.     m_xfer_done = false;
21.     //执行一次写操作，写入读取数据的起始地址
22.     err_code = nrf_drv_twi_tx(&m_twi, AT24C02_ADDRESS, tx_buf, 1, false);
23.     APP_ERROR_CHECK(err_code);
24.     //等待 TWI 传输完成
25.     while (m_xfer_done == false);
26.     //TWI 传输完成标志设置为 false
27.     m_xfer_done = false;
28.     //读出数据
29.     err_code = nrf_drv_twi_rx(&m_twi, AT24C02_ADDRESS, p_buf, size);
30.     APP_ERROR_CHECK(err_code);
31.     //等待 TWI 传输完成
32.     while (m_xfer_done == false){};
33.     return err_code;
34. }
```

编写好读写函数后，我们就可以通过读写函数对 eeprom 进行读和写，下面的程序分别使用单字节写入、页写入和批量写入函数，写入后通过读函数读取数据并将读取的数据通过串口输出。

代码清单：主函数

```
1. ****
2. * 描 述 : main 函数
3. * 入  参 : 无
4. * 返回值 : int 类型
5. ****
6. int main(void)
7. {
8.     uint8_t i;
9.     static uint8_t j = 0;
10.
11.    //设置 GPIO 输出电压为 3.3V
12.    gpio_output_voltage_setup_3v3();
13.    //配置用于驱动 LED 指示灯 D1 D2 D3 D4 的引脚脚，即配置 P0.13~P0.16 为输出，并将 LED 的初
14.    //始状态设置为熄灭，配置用于检测 4 个按键的 IO 位输入，并开启上拉电阻
15.    bsp_board_init(BSP_INIT_LEDS | BSP_INIT_BUTTONS);
```

```
16. //初始化串口
17. uart_config();
18. //初始化 TWI
19. twi_at24c02_init();
20.
21. while(true)
22. {
23.     //向指定地址写入单个字节数据:按动 S1 按键后向 AT24C02 的地址 0 写入 1 个字节数据(0x55)
24.     if(nrf_gpio_pin_read(BUTTON_1) == 0)
25.     {
26.         nrf_delay_ms(50); //延时消抖
27.         if(nrf_gpio_pin_read(BUTTON_1) == 0) //按键有效
28.         {
29.             //等待按键释放
30.             while(nrf_gpio_pin_read(BUTTON_1) == 0){}
31.             //向地址 0x00 写入一字节数据
32.             AT24C02_write_byte(1,0x55);
33.             printf("write 0x55 to AT24C02!\r\n");
34.             nrf_delay_ms(50);
35.             //从地址 0x00 读出一字节数据
36.             AT24C02_read_buf(1,i2c_rx_buf,1);
37.             printf("Read data from address 0: 0x%02x \r\n", i2c_rx_buf[0]);
38.         }
39.     }
40.     //向指定页面写入多个字节数据: 按动 S2 按键后向 AT24C02 的 page1 写入 8 个字节数据
41.     if(nrf_gpio_pin_read(BUTTON_2) == 0)
42.     {
43.         nrf_delay_ms(50); //延时消抖
44.         if(nrf_gpio_pin_read(BUTTON_2) == 0) //按键有效
45.         {
46.             //等待按键释放
47.             while(nrf_gpio_pin_read(BUTTON_2) == 0){}
48.             //初始化写入的数据
49.             for(i=0;i<8;i++)
50.             {
51.                 i2c_tx_buf[i] = 1+j++;
52.             }
53.             //向指定的页面写入数据, 数据长度不能超过页面的大小
54.             AT24C02_write_page(1,i2c_tx_buf,8);
55.             printf("write data to page 1!\r\n");
56.             //读出写入的数据
57.             AT24C02_read_buf(0,i2c_rx_buf,8);
58.             //串口打印出读取的数据
59.             printf("Read data from address 0:\r\n ");
```

```
60.         for(i=0;i<8;i++)printf("0x%02x ", i2c_rx_buf[i]);
61.         printf("\r\n");
62.     }
63. }
64. //向指定地址写入指定长度的数据：按动 S3 按键后从 AT24C02 的 0x07 地址开始连续写入 20 个
65. //字节数据
66. if(nrf_gpio_pin_read(BUTTON_3) == 0)
67. {
68.     nrf_delay_ms(50); //延时消抖
69.     if(nrf_gpio_pin_read(BUTTON_3) == 0) //按键有效
70.     {
71.         //等待按键释放
72.         while(nrf_gpio_pin_read(BUTTON_3) == 0){}
73.         //初始化写入的数据
74.         for(i=0;i<20;i++)
75.         {
76.             i2c_tx_buf[i] = 1+j++;
77.         }
78.         //向起始地址 0x07 连续写入 20 个字节数据
79.         AT24C02_write_buf(0x07,i2c_tx_buf,20);
80.         printf("Write 20 bytes of data starting at address 0x07!\r\n");
81.         //从起始地址 0x07 连续写入 20 个字节数据
82.         AT24C02_read_buf(0x07,i2c_rx_buf,20);
83.         //串口打印出读取的数据
84.         printf("Read 20 bytes of data starting at address 0x07!\r\n");
85.         for(i=0;i<20;i++)printf("0x%d ", i2c_rx_buf[i]);
86.         printf("\r\n");
87.     }
88. }
89. }
90. }
```

6.2.5. 硬件连接

本实验需要使用 LED 指示灯、按键、UART 和 AT24C02，按照下图所示短接跳线帽。

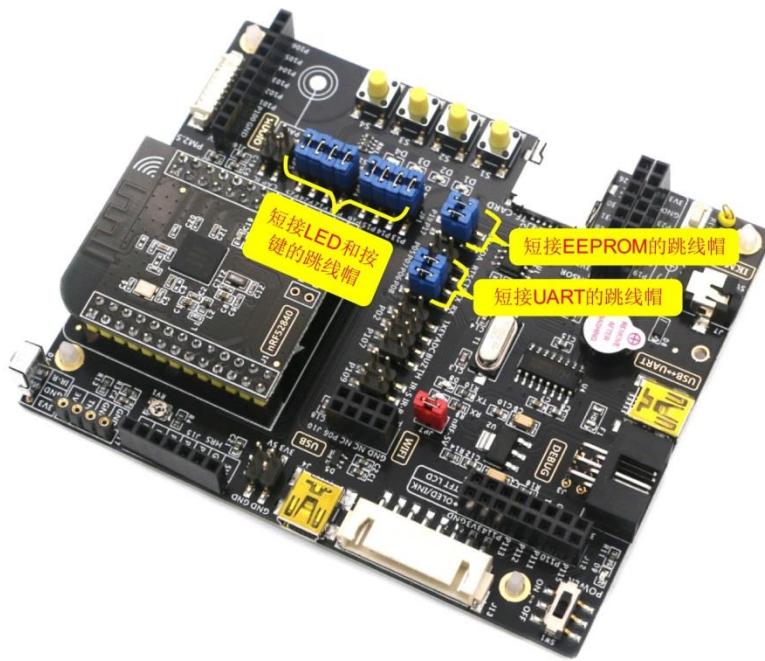


图 21-20：开发板跳线帽短接

6.2.6. 实验步骤

1. 解压“…\3: 开发指南(上册)配套试验源码\”目录下的压缩文件“实验 21-1: TWI(I2C)读写 eeprom-TWI0 非阻塞模式”，将解压后得到的文件夹“i2c_eeprom”拷贝到合适的目录，如“D\ nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\i2c_eeprom\project\mdk5”目录下的工程“i2c_eeprom.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，分别按下 S1、S2 和 S3 按键，执行读写 EEPROM。
 - 按下 S1 按键后松开按键：向 AT24C02 地址 0x00 写入 1 个字节数据（0x55），之后读出数据并通过串口输出。
 - 按下 S2 按键后松开按键：向 AT24C02 的第 1 页（地址 0x00~0x07）写入 8 个字节数据，之后读出数据并通过串口输出。
 - 按下 S3 按键后松开按键：向 AT24C02 批量写入数据，写入的起始地址为 0x07，连续写入 20 个字节数据，之后读出数据并通过串口输出。

6.3. eeprom 读写 (TWI0 阻塞) 实验

本实验在“实验 21-1: TWI(I2C)读写 eeprom-TWI0 非阻塞模式”的基础上修改。TWI 的模式设置为阻塞模式，程序的功能和“实验 21-1”一样。

❖ 注：本节对应的实验源码是：“实验 21-2：TWI(I2C)读写 eeprom-TWI0 阻塞模式”。

程序中需要修改的地方如下：

1. 调用 TWI 初始化函数 nrf_drv_twi_init() 初始化 TWI 时，事件句柄设置为 NULL，如下所示。

代码清单：初始化 TWI 为阻塞模式

- ```
1. //初始化 TWI
2. err_code = nrf_drv_twi_init(&m_twi, &twi_24c02_config, NULL, NULL);
3. //检查返回的错误代码
4. APP_ERROR_CHECK(err_code);
```
2. 删 除事件回调函数 twi\_handler(): 因为阻塞模式不需要事件句柄，因此应用程序无需提供 TWI 事件回调函数。
  3. 删 除掉读写函数中 TWI 传输完成标志清零的语句 “m\_xfer\_done == false;” 和等待 TWI 传输完成的语句 “while (m\_xfer\_done == false){ };”。  
修改后，编译并下载到开发板测试，实验步骤和“实验 21-1”一样。

#### ■ 思考题 1：如何使用 TWI1 读写 EEPROM？

提示：需要将 TWI 驱动程序实例的索引 TWI\_INSTANCE\_ID 的值修改为 1，“sdk\_config.h”

文件中关闭 TWI0，打开 TWI1，实验源码：

- 实验 21-3：TWI(I2C)读写 eeprom-TWI1 非阻塞模式。
- 实验 21-4：TWI(I2C)读写 eeprom-TWI1 阻塞模式。

## 第二十二章：SPI 总线及其应用

### 1. 学习目的

1. 了解 SPI 总线的结构、特点以及 4 种工作模式。
2. 掌握 nRF52840 的 SPI、SPIM 和 SPIS 的区别，尤其要注意 nRF52840 的 4 个 SPIM 外设（SPIM0~SPIM3）中 SPIM3 比较特殊，它的最大速度可以达到 32Mbps，并且支持硬件片选和 D/CX 输出。
3. 掌握通过 SPI 读、写和擦除 SPI Flash W25Q128FV 的方法以及注意事项。
4. 掌握 SPI 两种库的应用：
  - SPI 库：SPI、SPIM、SPIS 共用，缺点是一次 SPI 传输最大只能传输 255 个字节，如果我们一次 SPI 传输的数据需要超过 255 个字节，应用程序就需要增加相应的处理方式；优点是库函数的最大传输长度是按照 nRF52832 的 255 个字节进行的，因此应用程序在 nRF52840 和 nRF52832 之间移植很方便。
  - SPIM 库：SPIM 专用，一次 SPI 传输最大可传输 65535 个字节。

### 2. SPI 总线原理

SPI 是串行外设接口(Serial Peripheral Interface)的缩写，是一种高速、全双工、同步的通信总线。SPI 是 Motorola 公司推出的一种同步串行接口技术，SPI 由一个主设备和一个或多个从设备组成，在一次数据传输过程中，接口上只能有一个主机和一个从机能够通信。

SPI 总线的优点是操作简单、数据传输速率较高、全双工，缺点是只支持单个主机、没有指定的流控制，没有应答机制确认是否接收到数据。

#### 2.1. 接口信号定义

SPI 总线接口包括以下四种信号：

- 1) MOSI (Master Output, Slave Input): 主器件数据输出，从器件数据输入。
- 2) MISO (Master Input, Slave Output): 主器件数据输入，从器件数据输出。
- 3) SCK (Serial Clock): 有时也称为 SCLK，时钟信号，由主器件产生。
- 4) CS (Chip select): 有时也称为 SS，从器件使能信号，由主器件控制，实际使用时，经常用 GPIO 来代替。

SPI 总线支持连接多个从机，如下图所示，SPI 主机通过连接到从机的片选信号使能/禁止从机，并且同时只能使能一个从机，因此总线里面有多少个从机，就需要多少个片选信号。当 SPI 主机需要和总线中某个从机进行通讯时，主机会拉低对应的 CS 信号使能该从机，之后发起通讯，通讯完成后，拉高 CS 信号，解除总线的占用。

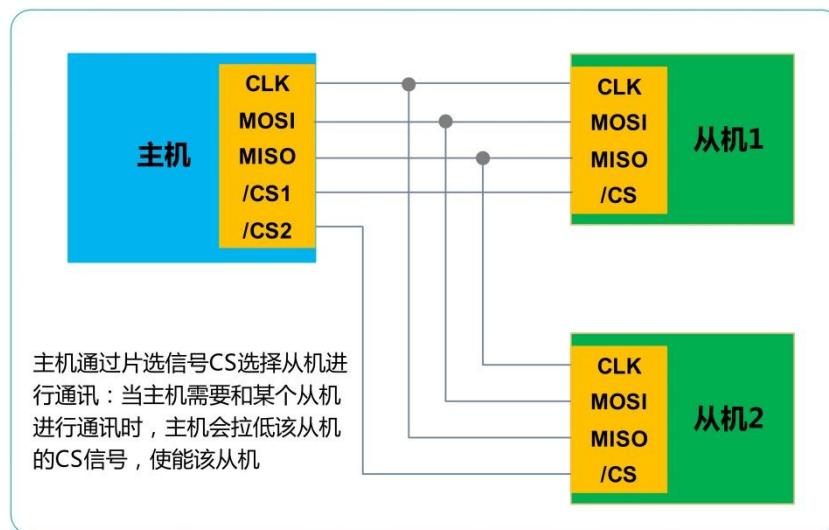


图 22-1: SPI 总线结构

对于 SPI 总线，我们还需要能深刻理解下面几个知识点。

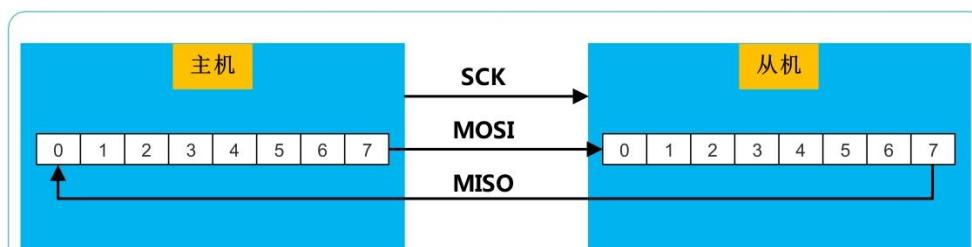
### 1. 硬件片选和软件片选的区别

所谓硬件片选指的是 SPI 本身具有片选信号，当我们通过 SPI 发送数据时，SPI 外设自动拉低 CS 信号使能从机，发送完成后自动拉高 CS 信号释放从机，这个过程是不需要软件操作的。而软件片选则是需要使用 GPIO 作为片选信号，SPI 在发送数据之前，需要先通过软件设置作为片选信号的 GPIO 输出低电平，发送完成之后再设置该 GPIO 输出高电平。

### 2. SPI 总线是回环结构

SPI 是一个环形总线结构，如下图所示，主设备和从设备构成一个环形。在时钟 SCK 的作用下，主设备发送一个位到从设备，因为是环形结构，所以从设备必定会同时传送一个位到主设备。同样，主设备向从设备发送一个字节，从设备也必定会同时传送一个字节到主设备。理解了环形结构，就很容易理解下面几点：

- SPI 是全双工，同步的通信总线。
- 主设备向从设备发送数据时，无论我们需不需要从设备返回数据，从设备都会返回数据。
- 主设备从从设备读取一个字节数据时，为什么需要写一个字节数据到从设备：因为是环形结构，不写数据过去，对方的数据就不会被移位过来。



在时钟 SCK 的作用下，主机每输出一个 bit 到 MOSI 信号线，从机会同时输出一个 bit 到 MISO 信号线，由此完成了全双工通讯。

图 22-2: SPI 数据传输示意图

### 3. SPI 主机和从机之间连接时信号不需要交叉

SPI 主机和从机连接时，MOSI 和 MISO 信号是不需要交叉连接的，因为 MOSI 本身就已经表示了主机输出、从机输入，MISO 表示主机输入、从机输出，因此不能交叉连接。

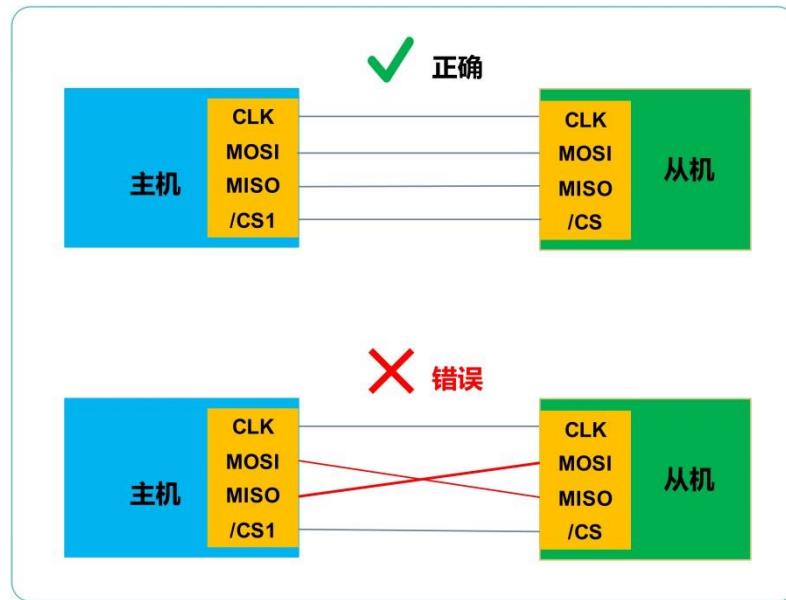


图 22-3: MOSI 和 MISO 不能交叉连接

## 2.2. SPI 的 4 种工作模式

SPI 总线共有 4 种工作模式：模式 0~模式 3，这 4 种工作模式是由时钟相位和时钟极性确定的。

1. 时钟相位 CPOL (Clock polarity): SPI 总线空闲时，时钟信号 SCLK 的电平称为时钟极性，有以下两种模式：
  - CPOL=0: SPI 总线空闲时，时钟信号为低电平。
  - CPOL=1: SPI 总线空闲时，时钟信号为高电平。
2. 时钟极性 CPHA (Clock phase): SPI 在时钟信号 SCLK 第几个边沿开始采样数据，有以下两种模式：
  - CPHA=0: 在第 1 个时钟边沿进行数据采样。
  - CPHA=1: 在第 2 个时钟边沿进行数据采样。

时钟极性 CPOL 时钟相位 CPHA 各有 2 种模式，它们两两组合就形成了 SPI 的 4 种工作模式，如下表所示。

表 22-1: SPI 总线的 4 种工作模式

| 模式   | 描述             |
|------|----------------|
| 模式 0 | CPOL=0, CPHA=0 |
| 模式 1 | CPOL=0, CPHA=1 |
| 模式 2 | CPOL=1, CPHA=0 |
| 模式 3 | CPOL=1, CPHA=1 |

SPI 的 4 种模式中，最常用的是模式 0 和模式 3。正是由于 SPI 有 4 种工作模式，因此当我们使用 SPI 总线时，需要去查询 SPI 总线中主机设备（如 nRF52840）和从机设备（如

SPI Flash) 的数据手册, 确定它们支持什么模式, 从而选择适合的工作模式。

SPI 的 4 种模式的时序图如下。

### ■ 时钟相位 CPHA=0 时的时序

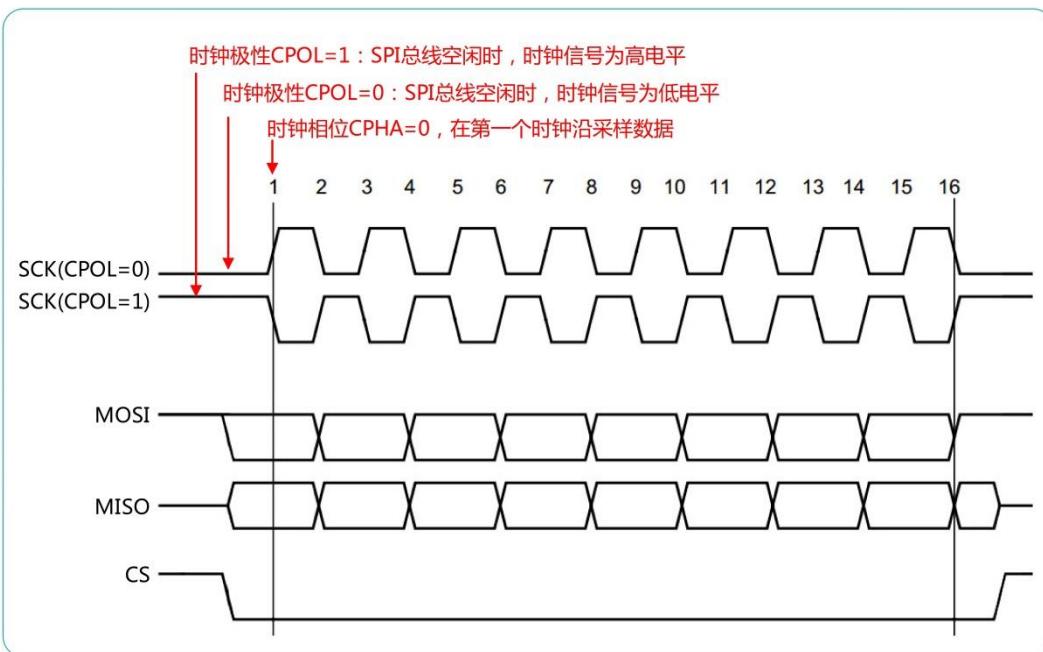


图 22-4: CPHA=0 时 SPI 时序

### ■ 时钟相位 CPHA=1 时的时序

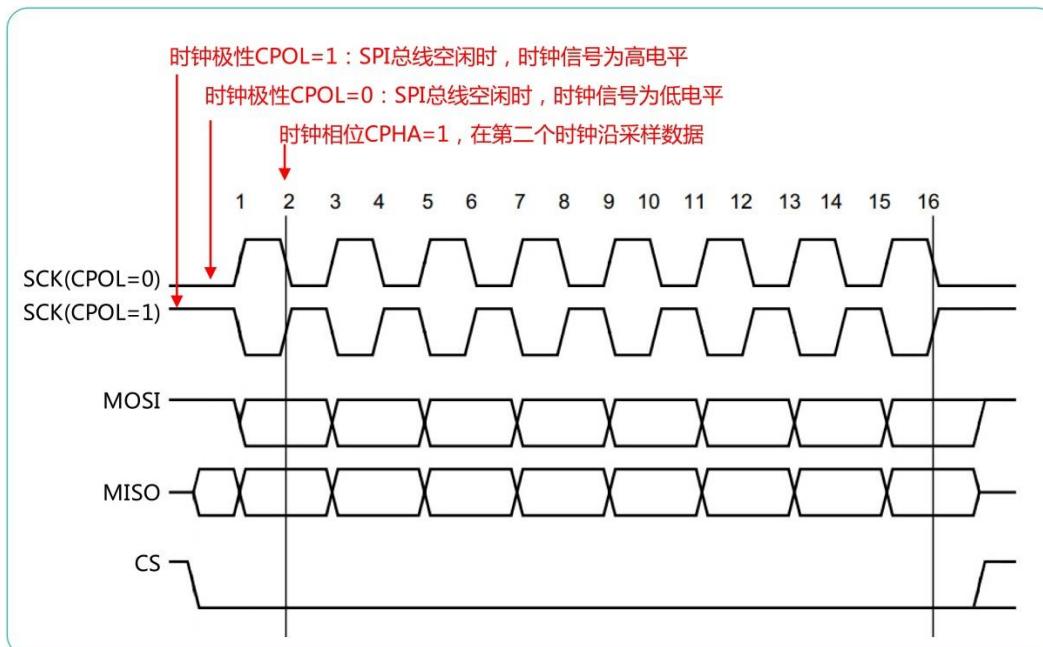


图 22-5: CPHA=1 时 SPI 时序

### 3. nRF52840 的 SPI

学习 nRF52840 的 SPI 前，我们首先要清楚 nRF52840 芯片的 SPI、SPIM 和 SPIS 的概念，nRF52840 片内集成的 SPI 分为三种，称为 SPI、SPIM 和 SPIS，它们的区别如下：

- SPI：不含 EasyDMA 的 SPI 主机，不推荐使用。
- SPIM：含 EasyDMA 的 SPI 主机。
- SPIS：含 EasyDMA 的 SPI 从机。

SPI、SPIM 和 SPIS 的寄存器是共用相同内存的，也就是说，SPI、SPIM 和 SPIS，我们同时只能用一个，如我们使用了 SPIM0，那么就不能同时使用 SPI0 和 SPISO。

本章的讲解是针对 SPIM 的，即带 EasyDMA 的 SPI 主机。

SPIM（SPI master with EasyDMA），表示具有 EasyDMA 的 SPI 主机，其特点如下：

- 片内集成 4 个 SPI 主机。
- SPIM 没有专用的片选引脚，连接多个从机时需要使用 IO 作为片选引脚区分从机。
- 可以通过 EasyDMA 从 RAM 存取数据。
- SPIM 支持 4 种 SPI 模式：模式 0~模式 3。
- SPI 的引脚可以通过配置相应的引脚配置寄存器映射到任意 IO。
- 可选的 D / CX 输出线，用于区分命令和数据字节。

#### 3.1. SPI 主机事务序列

如下图所示，SPI 主机通过触发 START 任务启动 SPI 事务，当发送器发送完 TXD.MAXCNT 寄存器中指定的 TXD 缓存区中的所有字节时，将产生 ENDTX 事件。接收器接收数据后填充到 RXD 缓存区，当填入到缓存区的数据字节数达到 RXD.MAXCNT 寄存器中指定的数值时将产生 ENDRX 事件。

SPI 主机执行 START 任务后，当 ENDRX 和 ENDTX 都产生时，SPI 主机将产生 END 事件。

SPI 主机可以在传输时通过触发 STOP 任务停止 SPI，SPI 主机将在停止前完成当前字节的发送/接收，SPI 主机停止后会产生 STOPPED 事件。

如果 SPI 主机停止时尚未产生 ENDTX 事件，则 SPI 主机将会产生 ENDTX 事件，即使 TXD.MAXCNT 寄存器中指定的 TXD 缓存中的所有字节均未传输。

如果在 SPI 主机停止时尚未产生 ENDRX 事件，则 SPI 主机同样会产生 ENDRX 事件，即使 RX 缓存区未满。

SPI 主机事务可以使用 SUSPEND 和 RESUME 任务挂起和恢复。当触发 SUSPEND 任务时，SPI 主机将在挂起之前完成发送和接收当前正在进行的字节。

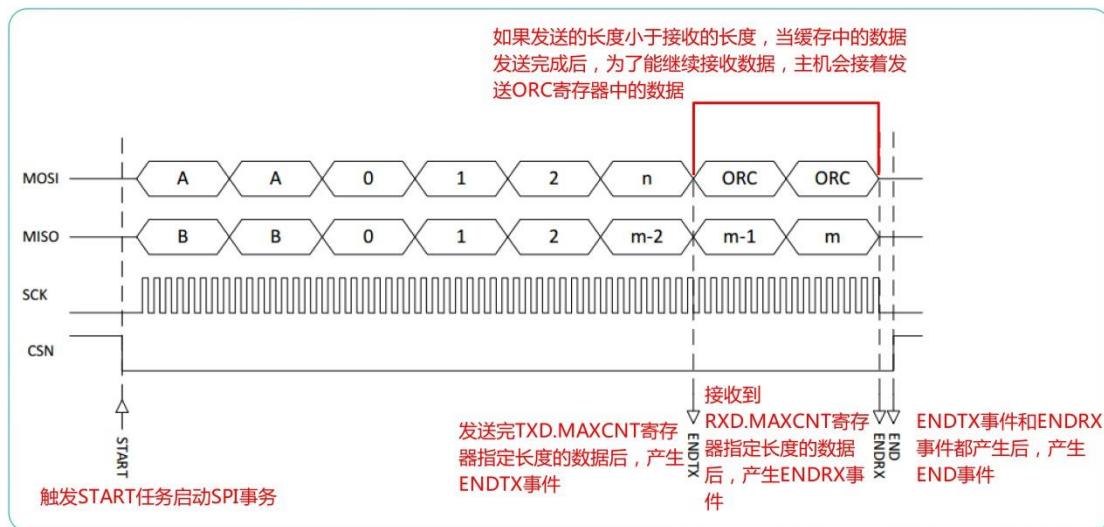


图 22-6: SPI 主机事务序列

### 3.2. D/CX 功能

某些 SPI 从设备（例如显示驱动器）需要来自 SPI 主设备的附加信号来区分命令和数据字节。对于显示驱动程序，该信号线通常称为 D/CX。

SPIM 支持这种 D/CX 输出线，在命令字节传输期间 D/CX 线设置为低，在数据字节传输期间设置为高，从而区分命令和数据。

D/CX 信号连接的引脚编号使用 PSELDCX 寄存器配置，SPI 传输的数据字节前的命令字节数通过 DCXCNT 寄存器配置，但在 SPI 的传输期间，不允许配置 DCXCNT 寄存器。从下面的时序图中可以清晰地看到 D/CX 信号的作用（DCXCNT 寄存器配置为 1，即前一个字节是命令）。

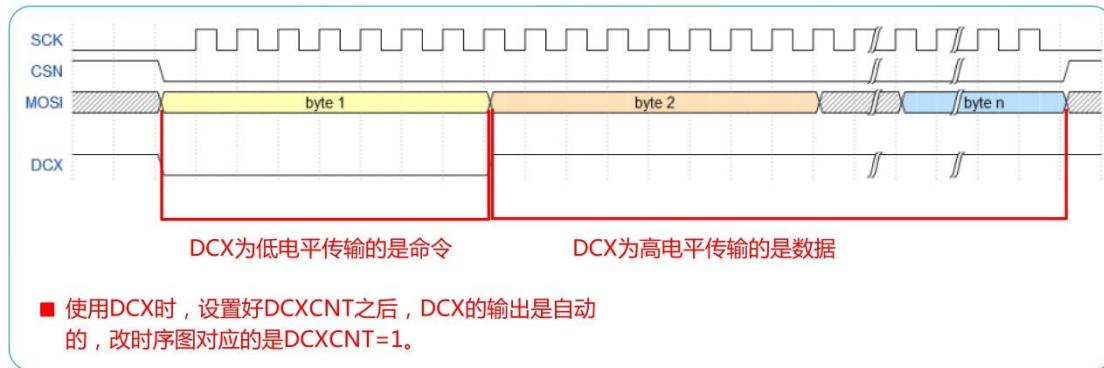


图 22-6: DCXCNT 配置为 1 时的时序图

### 3.3. EasyDMA

SPI 主机使用 EasyDMA 从 RAM 读取数据，而无需 CPU 参与。如果 RXD.PTR 和 TXD.PTR 没有指向数据 RAM 区域，EasyDMA 传输可能会导致 HardFault 或 RAM 数据损坏。

如果多个 AHB 总线主设备同时尝试访问同一个 AHB 从设备，则可能会发生 AHB 总线拥塞。SPIM 就是这样的 AHB 主设备，当发生拥塞时，SPIM 可能必须等待对 RAM 的访问才能被授予。如果 SPIM 必须等待总线访问，则事务将暂时停止，一旦 SPIM 被授权访问总线，事务恢复。注意，某些 SPIM 实例不支持此停止机制，请参阅寄存器章节的 SPIM 实例

表以获取有关各个实例中支持的功能的信息。

## ■ EasyDMA 数组列表

EasyDMA 数组列表可以由数据结构 array list\_类型表示，如代码示例所示。

```
#define BUFFER_SIZE 4

typedef struct ArrayList
{
 uint8_t buffer[BUFFER_SIZE];
} ArrayList_type;

ArrayList_type MyArrayList[3];

//replace 'Channel' below by the specific data channel you want to use,
// for instance 'NRF_SPIM->RXD', 'NRF_TWIM->RXD', etc.
Channel.MAXCNT = BUFFER_SIZE;
Channel.PTR = &MyArrayList;
```

该数据结构仅包括大小等于 Channel.maxcnt 的缓冲区，EasyDMA 将使用 Channel.maxcnt 寄存器确定缓冲区何时填满。使用时将' Channel '替换为要使用的特定数据通道，例如 'nrf\_spim->rx'、'nrf\_spim->tx'、'nrf\_twim->rx'等。

Channel.MAXCNT 寄存器的值不能大于缓冲区的实际大小。如果 Channel.MAXCNT 大于缓冲区的大小，则 EasyDMA 通道可能会溢出。

EasyDMA 数组列表不提供显式指定列表中下一个条目所在位置的机制，相反，它假定列表被组织为一个线性数组，其中条目在 RAM 中一个接一个地存放，如下图所示。

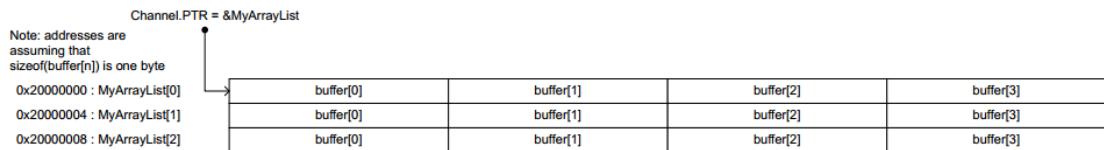


图 22-1: EasyDMA 数组列表

## 3.4. 引脚配置

SPIM 的 SCK、CSN、DCX、MOSI 和 MISO 信号可以自由映射，通过设置 PSEL.SCK、PSEL.CSN、PSEL.DCX、PSEL.MOSI 和 PSEL.MISO 寄存器可以将它们映射到任意的物理引脚，如下表所示。为确保 SPIM 中的正确行为，在使用 SPIM 前必须要正确配置 SPIM 的引脚，但是需要注意的是不能同时将多个 SPIM 外设的信号映射到同一个引脚，如不能将 SPIM0 和 SPIM1 的 MISO 同时映射到 P0.05，否则，可能会导致不可预测的行为。

表 22-2: SPIM 引脚配置

| SPIM 信号 | SPIM 引脚                    | 方向  |
|---------|----------------------------|-----|
| SCK     | 由 PSEL.SCK 寄存器设置信号映射的物理引脚。 | 输出。 |
| CSN     | 由 PSEL.CSN 寄存器设置信号映射的物理引脚。 | 输出。 |
| DCX     | 由 PSEL.DCX 寄存器设置信号映射的物理引脚。 | 输出。 |

|             |                             |     |
|-------------|-----------------------------|-----|
| <b>MOSI</b> | 由 PSEL.MOSI 寄存器设置信号映射的物理引脚。 | 输出。 |
| <b>MISO</b> | 由 PSEL.MISO 寄存器设置信号映射的物理引脚。 | 输入。 |

注意，某些 SPIM 实例不支持自动控制 CSN，也就是不具备我们前文所说的硬件片选，请参阅寄存器章节的 SPIM 实例表以获取有关各个实例中支持的功能的信息。

### 3.5. 低功耗

当系统处于低功耗且不需要 SPIM 外设时，为了尽可能降低功耗，应先停止 SPIM，然后禁用 SPIM 外设来实现最低功耗。

如果 SPIM 已经停止，则不需要触发 STOP 任务停止 SPIM，但是如 SPIM 正在执行发送，则应等待直到收到 STOPPED 事件作为响应，然后再通过 ENABLE 寄存器禁用外设。

## 4. SPIM 寄存器

nRF52840 片内集成了 4 个 SPIM 外设，如下表所示。注意 SPIM3 是比较特殊的，它最支持的最大数据速率是 32Mbps，同时支持硬件片选和 D/CX 输出。

表 22-3: SPIM 基址

| 外设名称  | 基址         | 不支持的配置                                                                                      |
|-------|------------|---------------------------------------------------------------------------------------------|
| SPIM0 | 0x40003000 | 不支持: >8Mbps 数据速率, CSNPOL 寄存器, DCX 功能, IFTIMING.x 寄存器, 硬件 CSN 控制 (PSEL.CSN), AHB 总线拥塞时的停止机制。 |
| SPIM1 | 0x40004000 | 不支持: >8Mbps 数据速率, CSNPOL 寄存器, DCX 功能, IFTIMING.x 寄存器, 硬件 CSN 控制 (PSEL.CSN), AHB 总线拥塞时的停止机制。 |
| SPIM2 | 0x40023000 | 不支持: >8Mbps 数据速率, CSNPOL 寄存器, DCX 功能, IFTIMING.x 寄存器, 硬件 CSN 控制 (PSEL.CSN), AHB 总线拥塞时的停止机制。 |
| SPIM3 | 0x4002F000 | ——                                                                                          |

表 22-4: SPIM 相关寄存器

| 序号           | 寄存器名          | 偏移地址  | 功能描述       |
|--------------|---------------|-------|------------|
| <b>任务寄存器</b> |               |       |            |
| <b>1</b>     | TASKS_START   | 0x010 | 启动 SPI 传输。 |
| <b>2</b>     | TASKS_STOP    | 0x014 | 停止 SPI 传输。 |
| <b>3</b>     | TASKS_SUSPEND | 0x01C | 挂起 SPI 传输。 |
| <b>4</b>     | TASKS_RESUME  | 0x020 | 恢复 SPI 传输。 |
| <b>事件寄存器</b> |               |       |            |

|          |                |       |                    |
|----------|----------------|-------|--------------------|
| <b>1</b> | EVENTS_STOPPED | 0x104 | SPI 传输停止。          |
| <b>2</b> | EVENTS_ENDRX   | 0x110 | 到达 RXD 缓存区结束。      |
| <b>3</b> | EVENTS_END     | 0x118 | 到达 RXD 和 TXD 缓存区结束 |
| <b>4</b> | EVENTS_ENDTX   | 0x120 | 到达 TXD 缓存区结束。      |
| <b>5</b> | EVENTS_STARTED | 0x14C | SPI 传输已启动。         |

**快捷方式寄存器**

|          |        |       |          |
|----------|--------|-------|----------|
| <b>1</b> | SHORTS | 0x200 | 快捷功能寄存器。 |
|----------|--------|-------|----------|

**通用寄存器**

|           |            |       |                                                                             |
|-----------|------------|-------|-----------------------------------------------------------------------------|
| <b>1</b>  | INTENSET   | 0x304 | 使能中断。                                                                       |
| <b>2</b>  | INTENCLR   | 0x308 | 禁止中断。                                                                       |
| <b>3</b>  | STALLSTAT  | 0x400 | EasyDMA RAM 访问的停顿状态，每当出现停顿时，该寄存器中的字段由硬件设置为 STALL，并且可以由 CPU 清除（设置为 NOSTALL）。 |
| <b>4</b>  | ENABLE     | 0x500 | 使能 SPIM。                                                                    |
| <b>5</b>  | PSEL.SCK   | 0x508 | SPIM SCK 信号引脚设置。                                                            |
| <b>6</b>  | PSEL.MOSI  | 0x50C | SPIM MOSI 信号引脚设置。                                                           |
| <b>7</b>  | PSEL.MISO  | 0x510 | SPIM MISO 信号引脚设置。                                                           |
| <b>8</b>  | PSEL.CSN   | 0x514 | SPIM CSN 信号引脚设置。                                                            |
| <b>9</b>  | FREQUENCY  | 0x524 | SPI 时钟频率。精度取决于 HFCLK 所选的时钟源。                                                |
| <b>10</b> | RXD.PTR    | 0x534 | 接收数据指针。                                                                     |
| <b>11</b> | RXD.MAXCNT | 0x538 | 接收缓存的最大字节数。                                                                 |
| <b>12</b> | RXD.AMOUNT | 0x53C | 最后一次传送的字节数。                                                                 |
| <b>13</b> | RXD.LIST   | 0x540 | 接收 EasyDMA 列表类型。                                                            |
| <b>14</b> | TXD.PTR    | 0x544 | 发送数据指针。                                                                     |
| <b>15</b> | TXD.MAXCNT | 0x548 | 发送缓存的最大字节数。                                                                 |
| <b>16</b> | TXD.AMOUNT | 0x54C | 最后一次传送的字节数。                                                                 |
| <b>17</b> | TXD.LIST   | 0x550 | 发送 EasyDMA 列表类型。                                                            |
| <b>18</b> | CONFIG     | 0x554 | 配置寄存器。                                                                      |

|           |                  |       |                                                                                      |
|-----------|------------------|-------|--------------------------------------------------------------------------------------|
| <b>19</b> | IFTIMING.RXDELAY | 0x560 | MISO 上输入串行数据的采样延迟。                                                                   |
| <b>20</b> | IFTIMING.CSNDUR  | 0x564 | CSN 的边沿与 SCK 的边沿之间的最小持续时间以及 CSN 必须在 SPI 事务之间保持高电平最小持续时间。单位是 64 MHz 时钟周期 (15.625 ns)。 |
| <b>21</b> | CSNPOL           | 0x568 | SPIM 传输时 CSN 的极性。                                                                    |
| <b>22</b> | PSELDCX          | 0x56C | DCX 信号引脚设置。                                                                          |
| <b>23</b> | DCXCNT           | 0x570 | DCXCNT 指定 SPIM 传输时数据字节之前的命令字节数。                                                      |
| <b>24</b> | ORC              | 0x5C0 | Over-read 字符。                                                                        |

## ■ SHORTS: 快捷方式寄存器

表 22-5: SHORTS 寄存器

| 位    | Field     | RW  | 复位值 | 描述                                                   |
|------|-----------|-----|-----|------------------------------------------------------|
| 位 17 | END_START | 读/写 | 0   | END 事件和 START 任务之间的快捷方式。<br>0: 禁止快捷方式。<br>1: 使能快捷方式。 |

## ■ INTENSET: 中断使能寄存器

INTENSET 寄存器用于使能中断。位的值写为 1 时使能对应的中断，写入 0 无效。注意 INTENSET 寄存器只能用来使能中断。

表 22-6: INTENSET 寄存器

| 位   | Field   | RW  | 复位值 | 描述                                                                   |
|-----|---------|-----|-----|----------------------------------------------------------------------|
| 位 1 | STOPPED | 读/写 | 0   | 写“1”使能 STOPPED 事件中断，写“0”无效。<br>写, 1: 使能。<br>读, 0: 已禁止。<br>读, 1: 已使能。 |
| 位 4 | ENDRX   | 读/写 | 0   | 写“1”使能 ENDRX 事件中断，写“0”无效。<br>写, 1: 使能。<br>读, 0: 已禁止。<br>读, 1: 已使能。   |
| 位 6 | END     | 读/写 | 0   | 写“1”使能 END 事件中断，写“0”无效。<br>写, 1: 使能。                                 |

|      |         |       |                                                                       |
|------|---------|-------|-----------------------------------------------------------------------|
|      |         |       | 读, 0: 已禁止。<br>读, 1: 已使能。                                              |
| 位 8  | ENDTX   | 读/写 0 | 写“1”使能 ENDTX 事件中断, 写“0”无效。<br>写, 1: 使能。<br>读, 0: 已禁止。<br>读, 1: 已使能。   |
| 位 19 | STARTED | 读/写 0 | 写“1”使能 STARTED 事件中断, 写“0”无效。<br>写, 1: 使能。<br>读, 0: 已禁止。<br>读, 1: 已使能。 |

### ■ INTENCLR: 中断禁止寄存器

INTENCLR 寄存器用于禁止中断。位的值写为 1 时禁止对应的中断，写入 0 无效。注意 INTENCLR 寄存器只能用来禁止中断。

表 22-7: INTENCLR 寄存器

| 位   | Field   | RW    | 复位值                                                                   | 描述 |
|-----|---------|-------|-----------------------------------------------------------------------|----|
| 位 1 | STOPPED | 读/写 0 | 写“1”禁止 STOPPED 事件中断, 写“0”无效。<br>写, 1: 禁止。<br>读, 0: 已禁止。<br>读, 1: 已使能。 |    |
| 位 4 | ENDRX   | 读/写 0 | 写“1”禁止 ENDRX 事件中断, 写“0”无效。<br>写, 1: 禁止。<br>读, 0: 已禁止。<br>读, 1: 已使能。   |    |
| 位 6 | END     | 读/写 0 | 写“1”禁止 END 事件中断, 写“0”无效。<br>写, 1: 禁止。<br>读, 0: 已禁止。<br>读, 1: 已使能。     |    |
| 位 8 | ENDTX   | 读/写 0 | 写“1”禁止 ENDTX 事件中断, 写“0”无效。                                            |    |

|      |         |     |   |                              |
|------|---------|-----|---|------------------------------|
| 位 19 | STARTED | 读/写 | 0 | 写, 1: 禁止。                    |
|      |         |     |   | 读, 0: 已禁止。                   |
|      |         |     |   | 读, 1: 已使能。                   |
|      |         |     |   | 写“1”禁止 STARTED 事件中断, 写“0”无效。 |
|      |         |     |   | 写, 1: 禁止。                    |
|      |         |     |   | 读, 0: 已禁止。                   |
|      |         |     |   | 读, 1: 已使能。                   |

### ■ STALLSTAT: EasyDMA RAM 访问的停顿状态寄存器

EasyDMA RAM 访问的停顿状态, 每当出现停顿时, 该寄存器中的字段由硬件设置为 STALL, 并且可以由 CPU 清除 (设置为 NOSTALL)。

表 22-8: ENABLE 寄存器

| 位   | Field | RW  | 复位值 | 描述                    |
|-----|-------|-----|-----|-----------------------|
| 位 0 | TX    | 读/写 | 0   | EasyDMA RAM 读取的停顿状态。  |
|     |       |     |     | 0: 没有停顿。<br>1: 发生了停顿。 |
| 位 1 | RX    | 读/写 | 0   | EasyDMA RAM 写入的停顿状态。  |
|     |       |     |     | 0: 没有停顿。<br>1: 发生了停顿。 |

### ■ ENABLE: SPIM 使能/禁止寄存器

ENABLE 寄存器是用于使能或禁用 SPIM。

表 22-8: ENABLE 寄存器

| 位       | Field  | RW  | 复位值 | 描述                                        |
|---------|--------|-----|-----|-------------------------------------------|
| 位 3~位 0 | ENABLE | 读/写 | 0   | 使能或禁止 SPIM。<br>0: 禁止 SPIM。<br>7: 使能 SPIM。 |

### ■ PSEL.SCK: SPIM 时钟信号 SCK 连接引脚配置寄存器

PSEL.SCK 寄存器用于配置 SPIM 的 SCK 连接的引脚, 其中 PIN 用于配置 SCK 连接的引脚编号, PORT 用于设置端口号, CONNECT 用于配置 SCK 和引脚是否连接。

表 22-9: PSEL.SCK 寄存器

| 位       | Field | RW  | 复位值   | 描述          |
|---------|-------|-----|-------|-------------|
| 位 4~位 0 | PIN   | 读/写 | 11111 | 引脚编号, 0~31。 |

|      |         |     |   |                             |
|------|---------|-----|---|-----------------------------|
| 位 5  | PORT    | 读/写 | 1 | 端口号<br>0: 端口 0。<br>1: 端口 1。 |
| 位 31 | CONNECT | 读/写 | 1 | 1: 不连接。<br>0: 连接。           |

### ■ PSEL.MOSI: SPIM 的 MOSI 信号连接引脚配置寄存器

PSEL.MOSI 寄存器用于配置 SPIM 的 MOSI 连接的引脚，其中 PIN 用于配置 MOSI 连接的引脚编号，PORT 用于设置端口号，CONNECT 用于配置 MOSI 和引脚是否连接。

表 22-10: PSEL.MOSI 寄存器

| 位       | Field   | RW  | 复位值   | 描述                          |
|---------|---------|-----|-------|-----------------------------|
| 位 4~位 0 | PIN     | 读/写 | 11111 | 引脚编号, 0~31。                 |
| 位 5     | PORT    | 读/写 | 1     | 端口号<br>0: 端口 0。<br>1: 端口 1。 |
| 位 31    | CONNECT | 读/写 | 1     | 1: 不连接。<br>0: 连接。           |

### ■ PSEL.MISO: SPIM 时钟信号 MISO 连接引脚配置寄存器

PSEL.MISO 寄存器用于配置 SPIM 的 MISO 连接的引脚，其中 PIN 用于配置 MISO 连接的引脚编号，PORT 用于设置端口号，CONNECT 用于配置 MISO 和引脚是否连接。

表 22-11: PSEL.MISO 寄存器

| 位       | Field   | RW  | 复位值   | 描述                          |
|---------|---------|-----|-------|-----------------------------|
| 位 4~位 0 | PIN     | 读/写 | 11111 | 引脚编号, 0~31。                 |
| 位 5     | PORT    | 读/写 | 1     | 端口号<br>0: 端口 0。<br>1: 端口 1。 |
| 位 31    | CONNECT | 读/写 | 1     | 1: 不连接。<br>0: 连接。           |

### ■ PSEL.CSN: SPIM 片选信号 CSN 连接引脚配置寄存器

PSEL.CSN 寄存器用于配置 SPIM 的 CSN 连接的引脚，其中 PIN 用于配置 CSN 连接的引脚编号，PORT 用于设置端口号，CONNECT 用于配置 CSN 和引脚是否连接。

表 22-11: PSEL.CSN 寄存器

| 位       | Field   | RW  | 复位值   | 描述                          |
|---------|---------|-----|-------|-----------------------------|
| 位 4~位 0 | PIN     | 读/写 | 11111 | 引脚编号, 0~31。                 |
| 位 5     | PORT    | 读/写 | 1     | 端口号<br>0: 端口 0。<br>1: 端口 1。 |
| 位 31    | CONNECT | 读/写 | 1     | 1: 不连接。<br>0: 连接。           |

### ■ FREQUENCY: SPI 主机数据速率寄存器

SPI 主机数据速率配置寄存器, 精度取决于所选的 HFCLK 时钟源。

表 22-12: FREQUENCY 寄存器

| 位        | Field     | RW  | 复位值        | 描述                                                                                                                                                                                                                                       |
|----------|-----------|-----|------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 位 31~位 0 | FREQUENCY | 读/写 | 0x04000000 | SPI 主机数据速率。<br>0x02000000: 125 kbps。<br>0x04000000: 250 kbps。<br>0x08000000: 500 kbps。<br>0x10000000: 1M kbps。<br>0x20000000: 2M kbps。<br>0x40000000: 4M kbps。<br>0x80000000: 8M kbps。<br>0x0A000000: 16M kbps。<br>0x14000000: 32M kbps。 |

### ■ RXD.PTR: 接收数据指针寄存器

表 22-13: RXD.PTR 寄存器

| 位        | Field | RW  | 复位值 | 描述    |
|----------|-------|-----|-----|-------|
| 位 31~位 0 | PTR   | 读/写 | 0   | 数据指针。 |

### ■ RXD.MAXCNT: 接收缓存的最大字节数配置寄存器

表 22-14: RXD.MAXCNT 寄存器

| 位        | Field  | RW  | 复位值 | 描述          |
|----------|--------|-----|-----|-------------|
| 位 15~位 0 | MAXCNT | 读/写 | 0   | 接收缓存的最大字节数。 |

### ■ RXD.AMOUNT: 最后一次传送的字节数查询寄存器

表 22-15: RXD.AMOUNT 寄存器

| 位        | Field  | RW | 复位值 | 描述                                    |
|----------|--------|----|-----|---------------------------------------|
| 位 15~位 0 | AMOUNT | 只读 | 0   | 最后一次传送的字节数，在 NACK 错误的情况下，包括 NACK 的字节。 |

### ■ RXD.LIST: 接收 EasyDMA 表类型配置寄存器

表 22-16: RXD.LIST 寄存器

| 位       | Field | RW  | 复位值 | 描述                                           |
|---------|-------|-----|-----|----------------------------------------------|
| 位 2~位 0 | LIST  | 读/写 | 0   | 表类型。<br>0: 禁止 EasyDMA 表。<br>1: 使能 EasyDMA 表。 |

### ■ TXD.PTR: 发送数据指针寄存器

表 22-17: TXD.PTR 寄存器

| 位        | Field | RW  | 复位值 | 描述    |
|----------|-------|-----|-----|-------|
| 位 31~位 0 | PTR   | 读/写 | 0   | 数据指针。 |

### ■ TXD.MAXCNT: 接收缓存的最大字节数配置寄存器

表 22-18: TXD.MAXCNT 寄存器

| 位        | Field  | RW  | 复位值 | 描述          |
|----------|--------|-----|-----|-------------|
| 位 15~位 0 | MAXCNT | 读/写 | 0   | 发送缓存的最大字节数。 |

### ■ RXD.AMOUNT: 最后一次传送的字节数查询寄存器

表 22-19: RXD.AMOUNT 寄存器

| 位        | Field  | RW | 复位值 | 描述                                    |
|----------|--------|----|-----|---------------------------------------|
| 位 15~位 0 | AMOUNT | 只读 | 0   | 最后一次传送的字节数，在 NACK 错误的情况下，包括 NACK 的字节。 |

### ■ TXD.LIST: 发送 EasyDMA 表类型配置寄存器

表 22-20: TXD.LIST 寄存器

| 位       | Field | RW  | 复位值 | 描述                                           |
|---------|-------|-----|-----|----------------------------------------------|
| 位 2~位 0 | LIST  | 读/写 | 0   | 表类型。<br>0: 禁止 EasyDMA 表。<br>1: 使能 EasyDMA 表。 |

### ■ CONFIG: SPIM 配置寄存器

表 22-21: CONFIG 寄存器

| 位   | Field | RW  | 复位值 | 描述                                                             |
|-----|-------|-----|-----|----------------------------------------------------------------|
| 位 0 | ORDER | 读/写 | 0   | 位序。<br>0: 高位在前。<br>1: 低位在前。                                    |
| 位 1 | CPHA  | 读/写 | 0   | 时钟相位。<br>0: 在时钟前沿采样，在尾随时移位串行数据边缘。<br>1: 在时钟的后沿采样，在前导上移位串行数据边缘。 |
| 位 2 | CPOL  | 读/写 | 0   | 时钟极性。<br>0: SPI 总线空闲时，时钟信号为低电平。<br>1: SPI 总线空闲时，时钟信号为高电平。      |

### ■ IFTIMING.RXDELAY: MISO 采样延迟配置寄存器

表 22-21: IFTIMING.RXDELAY 寄存器

| 位       | Field   | RW  | 复位值 | 描述                                                                                                                                                                              |
|---------|---------|-----|-----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 位 2~位 0 | RXDELAY | 读/写 | 0   | MISO 上输入串行数据的采样延迟。该值指定从 SCK 的采样边沿(CONFIG.CPHA = 0 的前沿, CONFIG.CPHA = 1 的后沿) 延迟到输入串行数据采样的 64 MHz 时钟周期(15.625 ns)的数量。例如, 当 RXDELAY = 0 且 CONFIG.CPHA = 0 时在 SCK 的上升沿对输入的串行数据进行采样。 |

### ■ IFTIMING.CSNDUR: MISO 采样延迟配置寄存器

表 22-21: IFTIMING.CSNDUR 寄存器

| 位       | Field  | RW  | 复位值 | 描述                                                                                   |
|---------|--------|-----|-----|--------------------------------------------------------------------------------------|
| 位 7~位 0 | CSNDUR | 读/写 | 0   | CSN 的边沿与 SCK 的边沿之间的最小持续时间以及 CSN 必须在 SPI 事务之间保持高电平最小持续时间。单位是 64 MHz 时钟周期 (15.625 ns)。 |

### ■ PSELDCX: DCX 信号连接引脚配置寄存器

PSELDCX 寄存器用于配置 SPIM 的 D/CX 连接的引脚, 其中 PIN 用于配置 D/CX 连接的引脚编号, PORT 用于设置端口号, CONNECT 用于配置 D/CX 和引脚是否连接。

表 22-11: PSELDCX 寄存器

| 位       | Field   | RW  | 复位值   | 描述                          |
|---------|---------|-----|-------|-----------------------------|
| 位 4~位 0 | PIN     | 读/写 | 11111 | 引脚编号, 0~31。                 |
| 位 5     | PORT    | 读/写 | 1     | 端口号<br>0: 端口 0。<br>1: 端口 1。 |
| 位 31    | CONNECT | 读/写 | 1     | 1: 不连接。<br>0: 连接。           |

### ■ DCXCNT: DCX 配置寄存器

表 22-21: DCXCNT 寄存器

| 位       | Field  | RW  | 复位值 | 描述                                                                           |
|---------|--------|-----|-----|------------------------------------------------------------------------------|
| 位 3~位 0 | DCXCNT | 读/写 | 0   | 该寄存器指定数据字节之前的命令字节数。PSEL.DCX 线在命令字节传输期间为低电平，在数据字节传输期间为高电平。值 0xF 表示所有字节都是命令字节。 |

### ■ ORC: Over-read 字符设置寄存器

ORC 寄存器用于设置当接收的数据长度大于发送的数据长度时，继续接收数据时发送的字符。

表 22-22: TXD.LIST 寄存器

| 位       | Field | RW  | 复位值 | 描述            |
|---------|-------|-----|-----|---------------|
| 位 7~位 0 | ORC   | 读/写 | 0   | Over-read 字符。 |

## 5. 软件设计-SPI 库

SPI 库是 SPI、SPIM、SPIS 共用，SPI 库的最大传输长度是按照 nRF52832 的 255 个字节进行的，因此使用 SPI 库的应用程序在 nRF52840 和 nRF52832 之间移植很方便。但是 nRF52840 的 SPI 是支持一次最大传输 65535 个字节的，显然使用 SPI 库无法充分发挥 nRF52840 的性能，不过对于一次 SPI 传输不大于 255 个字节的应用，如果考虑移植的问题，SPI 库会有优势一些。

### 5.1. 库函数的应用

SPI 的库函数支持 SPI 和 SPIM，也就是库函数里面可以配置是否使用 EasyDMA (sdk\_config.h 中配置)，不使用 EasyDMA 即为 SPI，使用 EasyDMA 即为 SPIM，具体配置方法在下文中会有描述。

SPI 的应用步骤如下图所示，首先要定义 SPI 驱动程序实例，驱动程序实例对应具体的

硬件 SPI 外设，接着配置是否使用 EasyDMA，之后初始化配置 SPI 连接的物理引脚和速率等参数，注册事件句柄（非堵塞模式下），配置完成后即可使用 SPI 进行传输数据。



图 22-7: SPI 应用步骤

### 5.1.1. 定义 SPI 驱动程序实例

SPI 驱动程序实例使用 nrf\_drv\_spi\_t 结构体定义，该结构体描述了具体的 SPI 外设，当我们定义了 nrf\_drv\_spi\_t 类型的变量并对其赋值后，该变量就对应了一个具体的硬件 SPI 外设，具体对应的是 nRF52840 的 SPI 外设还是 SPIM 外设，取决于我们是否配置使用 EasyDMA。

定义 SPI 驱动程序实例代码如下，初始化宏 NRF\_DRV\_SPI\_INSTANCE 的输入参数对应 SPI 外设的编号。

#### 代码清单：定义 SPI 驱动程序实例

```
1. //SPI 驱动程序实例 ID, ID 和外设编号对应, 0:SPIM0 1:SPIM1 2:SPIM2
2. #define SPI_INSTANCE 0
3. //定义 SPI 驱动程序实例
4. static const nrf_drv_spi_t 驱动实例名称= NRF_DRV_SPI_INSTANCE(SPI_INSTANCE);
```

### 5.1.2. 配置是否使用 EasyDMA

SPI 是否使用 EasyDMA 是在 sdk\_config.h 文件中配置的，如下图所示。需要注意的是，如果自己的工程里面没有 SPI 的配置项目，需要手动从例子程序的 sdk\_config.h 文件中将 SPI 这部分配置拷贝到自己工程的 sdk\_config.h 文件里面。

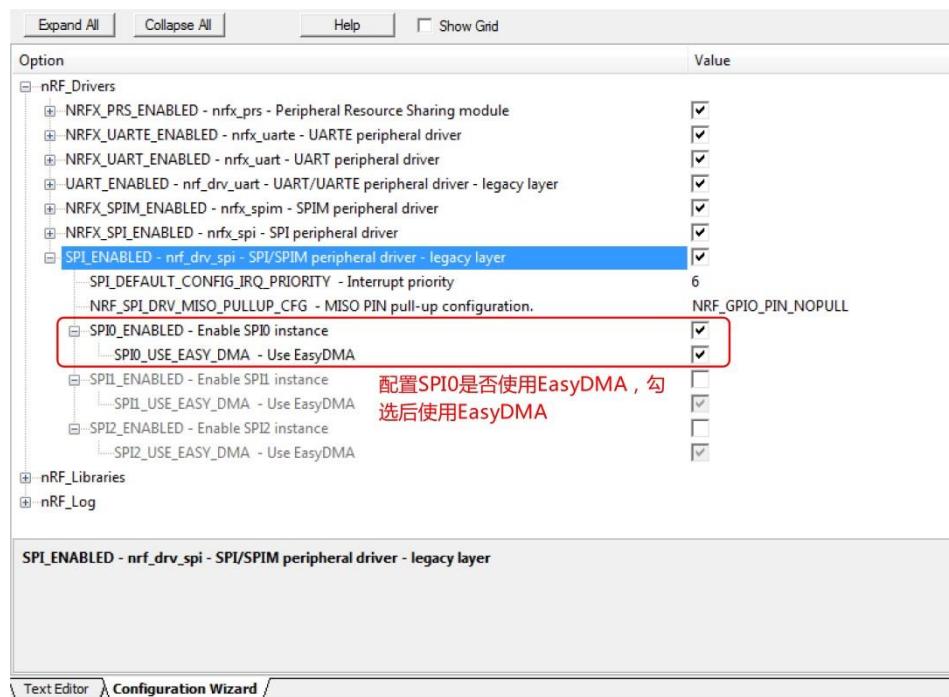


图 22-8：配置是否使用 EasyDMA

### 5.1.3. 初始化 SPI

SPI 初始化的库函数是 `nrf_drv_spi_init()` 函数，该函数初始化 SPI 后会在函数内部使能 SPI，函数原型如下。

表 22-23: `nrf_drv_spi_init()` 函数

|        |                                                                                                                                                                                                                                                               |
|--------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 函数原型   | <pre>ret_code_t nrf_drv_spi_init(nrf_drv_spi_t const *const p_instance,                             nrf_drv_spi_config_t const * p_config,                             nrf_drv_spi_evt_handler_t handler,                             void * p_context)</pre> |
| 函数功能   | 初始化 SPI 主机驱动程序实例，该函数配置并且使能指定的 SPI 外设。                                                                                                                                                                                                                         |
| 参    数 | <p>[in] <code>p_instance</code>: 指向驱动程序实例结构体。</p> <p>[in] <code>p_config</code>: 指向初始化配置结构体。</p> <p>[in] <code>handler</code>: 指向事件句柄，如果为 NULL，SPI 主机工作于阻塞模式。</p> <p>[in] <code>p_context</code>: 传递给事件句柄的 Context。</p>                                       |
| 返回值    | <p><code>NRF_SUCCESS</code>: 初始化成功。</p> <p><code>NRF_ERROR_INVALID_STATE</code>: 驱动程序实例已经初始化。</p> <p><code>NRF_ERROR_BUSY</code>: 其它具有相同 ID 的外设已经在使用了，只有当 <code>PERIPHERAL_RESOURCE_SHARING_ENABLED</code> 设置为非 0 时才可能 会出现这种情况。</p>                           |

## ■ 初始化结构体

调用 nrf\_drv\_spi\_init() 函数初始化 SPI 时需要提供初始化结构体变量作为函数的参数，初始化结构体包含了 SPI 初始化时所需要配置的参数，其声明如下：

#### 代码清单：SPI 初始化配置结构体 nrf\_drv\_spi\_config\_t 声明

```
1. typedef struct
2. {
3. //SPI SCK 信号连接的引脚
4. uint8_t sck_pin;
5. //SPI MOSI 信号连接的引脚(可选)，如果不需要 MOSI，设置为 NRF_DRV_SPI_PIN_NOT_USED
6. uint8_t mosi_pin;
7. //SPI MISO 信号连接的引脚(可选)，如果不需要 MISO，设置为 NRF_DRV_SPI_PIN_NOT_USED
8. uint8_t miso_pin;
9. //SPI CS 信号连接的引脚(可选)，如果不需要 CS，设置为 NRF_DRV_SPI_PIN_NOT_USED
10. uint8_t ss_pin;
11. //中断优先级
12. uint8_t irq_priority;
13. //ORC 寄存器内容
14. uint8_t orc;
15. //SPI 速率
16. nrf_drv_spi_frequency_t frequency;
17. //SPI 模式
18. nrf_drv_spi_mode_t mode;
19. //SPI 传输位序
20. nrf_drv_spi_bit_order_t bit_order;
21. } nrf_drv_spi_config_t;
```

由 nrf\_drv\_spi\_config\_t 声明可以看到，SPI 初始化时可以配置的 SPI 参数如下：

- SPI 信号连接的物理引脚：可以设置连接到任意引脚，但是两个 SPI 信号不能连接到同一个引脚，如 MISO 和 MOSI 不能连接到同一个引脚。不同 SPI 外设的信号也不能连接到同一个引脚，如 SPI0 的 MISO 和 SPI1 的 MISO 不能连接到同一个引脚。另外，SPI 外设本身是没有片选信号（CS）的，这里设置了 CS 连接的引脚表示 CS 由 SPI 驱动程序管理，如果设置 CS 为 “NRF\_DRV\_SPI\_PIN\_NOT\_USED”，那么应用程序就需要自己管理 CS。
- SPI 速率：SPI 速率定义如下，最快速率 8 Mbps。

#### 代码清单：SPI 速率定义

```
1. typedef enum
2. {
3. NRF_DRV_SPI_FREQ_125K = NRF_SPI_FREQ_125K, //125 kbps.
4. NRF_DRV_SPI_FREQ_250K = NRF_SPI_FREQ_250K, //250 kbps.
5. NRF_DRV_SPI_FREQ_500K = NRF_SPI_FREQ_500K, //500 kbps.
6. NRF_DRV_SPI_FREQ_1M = NRF_SPI_FREQ_1M, //1 Mbps.
7. NRF_DRV_SPI_FREQ_2M = NRF_SPI_FREQ_2M, //2 Mbps.
```

```

8. NRF_DRV_SPI_FREQ_4M = NRF_SPI_FREQ_4M, //4 Mbps.
9. NRF_DRV_SPI_FREQ_8M = NRF_SPI_FREQ_8M //8 Mbps.
10. } nrf_drv_spi_frequency_t;

```

- SPI 模式: SPI 可配置为模式 0~模式 3, 模式定义如下。

#### 代码清单: SPI 模式定义

```

1. typedef enum
2. {
3. NRF_DRV_SPI_MODE_0 = NRF_SPI_MODE_0, //模式 0
4. NRF_DRV_SPI_MODE_1 = NRF_SPI_MODE_1, //模式 1
5. NRF_DRV_SPI_MODE_2 = NRF_SPI_MODE_2, //模式 2
6. NRF_DRV_SPI_MODE_3 = NRF_SPI_MODE_3 //模式 3
7. } nrf_drv_spi_mode_t;

```

- SPI 位序: SPI 的位序可配置为 MSB 或者 LSB, 大多数情况下, 用的都是 MSB, 位序的定义如下:

#### 代码清单: SPI 位序定义

```

1. typedef enum
2. {
3. NRF_DRV_SPI_BIT_ORDER_MSB_FIRST = NRF_SPI_BIT_ORDER_MSB_FIRST, //MSB
4. NRF_DRV_SPI_BIT_ORDER_LSB_FIRST = NRF_SPI_BIT_ORDER_LSB_FIRST //LSB
5. } nrf_drv_spi_bit_order_t;

```

- ORC 内容: ORC 寄存器内容一般设置为 0xFF, 当发送长度小于接收数据长度时, 驱动程序继续发送 ORC 寄存器的内容以读取剩余的数据。
- 中断优先级: 可设置的中断优先级位 0~7, BLE 工程中, 优先级 0、1、4、5 保留给了 SoftDevice 使用, SPI 不能使用这些中断优先级。

#### ■ 事件句柄

应用程序是否提供事件句柄决定了 SPI 工作与阻塞模式还是非阻塞模式。

- 应用程序提供事件句柄: SPI 工作于非阻塞模式。
- 应用程序不提供事件句柄 (event\_handler 设置为 NULL): SPI 工作于阻塞模式。

如果我们需要 SPI 工作于非阻塞模式, 就需要提供事件回调函数, 并在初始化时将事件句柄作为 nrf\_drv\_spi\_init() 函数的参数传递给函数, 由初始化函数完成注册。SPI 事件回调函数的编写格式如下, 我们编写 SPI 事件回调函数时需要遵循该格式。

#### 代码清单: SPI 事件回调函数编写格式

```

1. void spi_event_handler(nrf_drv_spi_evt_t const * p_event,
2. void * p_context)
3. {
4. //功能代码
5. }

```

### 5.1.4. 数据传输

SPI 驱动程序提供的用于数据传输的函数是 nrf\_drv\_spi\_transfer(), 函数原型如下表所示, SPI 是全双工回环总线, 一个函数即可完成数据收发。

表 22-24: nrf\_drv\_spi\_transfer() 函数

|      |                                                                                                                                                                                                                                                                                                                    |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 函数原型 | <pre>_STATIC_INLINE ret_code_t nrf_drv_spi_transfer(<br/>    (p_instance,<br/>     p_tx_buffer,<br/>     tx_buffer_length,<br/>     p_rx_buffer,<br/>     rx_buffer_length))</pre>                                                                                                                                 |
| 函数功能 | <p>启动 SPI 数据传输。如果调用 nrf_drv_spi_init 函数初始化 SPI 时, 注册了事件句柄, 调用该函数后会立即返回, 当 SPI 传输完成后会通过事件句柄通知应用程序。如果没有注册事件句柄, 该函数工作于阻塞模式, 即该函数会一直等待数据传输完成才会返回。</p> <p>外设使用 EasyDMA (如 SPIM) 时, 传输数据的缓存需要放置于数据 RAM 区域, 如果缓存没有放置到数据 RAM 区域并且使用了 SPIM, 该函数会返回错误代码: NRF_ERROR_INVALID_ADDR。</p>                                         |
| 参 数  | <p>[in] <code>p_instance</code>: 指向驱动程序实例结构体。</p> <p>[in] <code>p_tx_buffer</code>: 指向发送缓存, 如果没有数据发送, 设为 NULL。</p> <p>[in] <code>tx_buffer_length</code>: 发送缓存长度, 最大 255 个字节。</p> <p>[in] <code>p_rx_buffer</code>: 指向接收缓存, 如果不需要接收数据, 设为 NULL。</p> <p>[in] <code>rx_buffer_length</code>: 接收缓存长度, 最大 255 个字节。</p> |
| 返回值  | <p>NRF_SUCCESS: 操作成功。</p> <p>NRF_ERROR_BUSY: 之前启动的传输还没有完成。</p> <p>NRF_ERROR_INVALID_ADDR: 提供的缓存没有位于数据 RAM 区域。</p>                                                                                                                                                                                                  |

#### ■ 发送超过 255 个字节数据

使用 nrf\_drv\_spi\_transfer() 函数时需要注意该函数发送长度 `tx_buffer_length` 和接收长度 `rx_buffer_length` 都是 `uint8_t` 类型变量, 因此一次最多只能发送/接收 255 个字节 (因为 SPIM 的 EasyDMA 的最大收发长度是 255 个字节)。

但是在实际应用中, 如读/写外部 Flash, 一次发送 255 个字节显然很不方便, 如果我们希望一次传输能够收/发更多的数据, 处理方式如下:

- CS 不能交由 SPI 驱动程序管理, 应交由应用程序管理, 应用程序在传输之前拉低 CS 使能从机设备, 直到所有数据传输完成后拉高 CS, 释放从机设备。
- 传输的数据分次通过通过 EasyDMA 传输, 如下图所示。

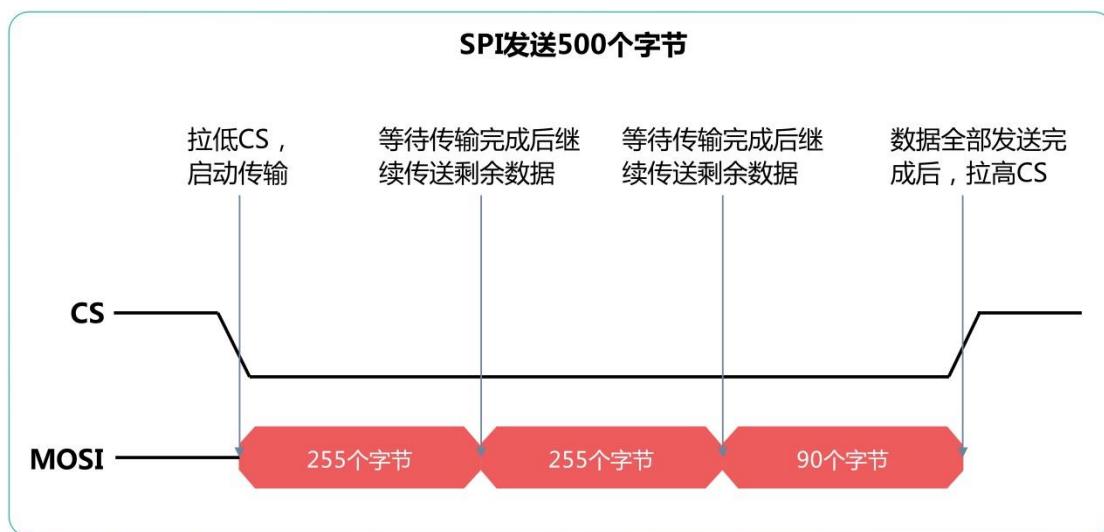


图 22-9: SPI 传输 500 个字节

## 5.2. W25Q128FV 读写（SPIM0 非阻塞）实验

本实验在“实验 10-1：串口数据收发”的基础上修改。通过 SPI 读写 SPI 接口的存储器 W25Q128FV。

W25Q128FV 是华邦公司（Winbond）推出的串行 NOR Flash 系列存储器中的一员，该系列还有 W25Q80/16/32/64 等，W25Q128FV 名称的意义如下：

- W：华邦公司（Winbond）。
- 25Q：SpiFlash 串行 NOR Flash，具有 4KB 扇区，双/四路 I/O。
- 128F：128M-bit。
- V：工作电压范围为（2.7~3.6）V。

W25Q128FV 的容量为 128Mb 共 16MB（注意大写的 B 表示字节，小写的 b 表示位），W25Q128FV 将 16M 的容量分为 256 个块（Block），每个块大小为 64KB，每个块又分为 16 个扇区（Sector），每个扇区 4K 个字节。W25Q128FV 的最小擦除单位为一个扇区，也就是每次至少擦除 4K 字节。

本例中，SPI 速率使用最快速率 8Mbps、SPI 模式使用模式 0、使用 EasyDMA 传输数据，SPI 连接 W25Q128FV 所用的引脚如下表所示。

表 22-25: SPI 连接 W25Q128FV 引脚分配

| 名称   | 引脚    | 说明                             |
|------|-------|--------------------------------|
| CS   | P0.23 | SPI 片选信号，连接到 W25Q128FV 的 CS。   |
| CLK  | P0.22 | SPI 时钟信号，连接到 W25Q128FV 的 CLK。  |
| MISO | P0.21 | SPI 主入从出，连接到 W25Q128FV 的 MISO。 |
| MOSI | P0.20 | SPI 主出从入，连接到 W25Q128FV 的 MOSI。 |

本例中实现的主要功能如下：

- 读取 W25Q128FV 芯片 ID：通过读芯片 ID 可以判断芯片类型以及判断 W25Q128FV 是

否正确接入。

- 扇区擦除：擦除一个指定的扇区。
- 全片擦除：擦除整个芯片。
- 页编程：向指定页面连续写入不超过页面地址范围的数据。
- 批量编程：向指定地址连续写入指定长度数据的功能，该功能实现了跨页写入。
- 批量读：从指定地址连续读取指定长度数据，并将读取的数据通过串口输出。

❖ 注：本节对应的实验源码是：“实验 22-1：SPIM0 读写 W25Q128FV-非阻塞模式”。

### 5.2.1. 添加需要的文件

本例需要添加 SPI 驱动文件和 W25Q128FV 驱动文件，需要加入的文件如下表所示。

表 20-26：SPI 需要加入的文件

| 文件名           | SDK 中的目录                        | 描述                    |
|---------------|---------------------------------|-----------------------|
| nrf_drv_spi.c | ..\\integration\\nrfx\\legacy   | 旧版本 SPI 驱动文件。         |
| nrfx_spi.c    | ..\\modules\\nrfx\\drivers\\src | 新版本 SPI 驱动文件。         |
| nrfx_spim.c   | ..\\modules\\nrfx\\drivers\\src | SPI 主机驱动文件。           |
| w25q128.c     | ..\\app\\w25q128                | 编写的 W25Q128FV 驱动程序文件。 |

### 5.2.2. 头文件引用和路径设置

#### 1. 需要引用的头文件

因为在“main.c”文件中使用了 SPI 程序模块和我们编写的 W25Q128FV 驱动程序，所以“main.c”文件需要引用下面的头文件。

```
#include "nrf_drv_spi.h"
#include "w25q128.h"
```

#### 2. 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

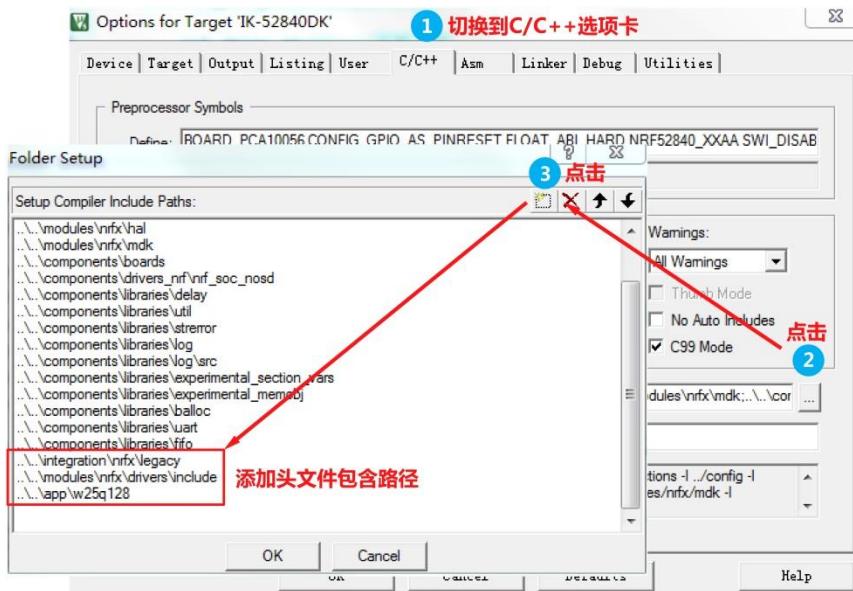


图 22-10：添加头文件包含路径

本例需要包含 SPI 驱动头文件路径和 W25Q128FV 驱动头文件路径，需要添加的头文件路径如下表：

表 22-27： 头文件包含路径

| 序号 | 路径                                 |
|----|------------------------------------|
| 1  | ..\..\integration\nrfx\legacy      |
| 2  | ..\..\modules\nrfx\drivers\include |
| 3  | ..\..\app\w25q128                  |

### 5.2.3. 工程配置

打开“sdk\_config.h”文件，加入SPI需要的配置，如下图所示，编辑内容时切换到“Text Editor”进行编辑（编辑的内容参考实验的源码，因此代码很长，此处不贴出代码，参见“sdk\_config.h”文件的（144~439）行），编辑完成后，切换到“Configuration Wizard”，即可观察到配置的具体项目：

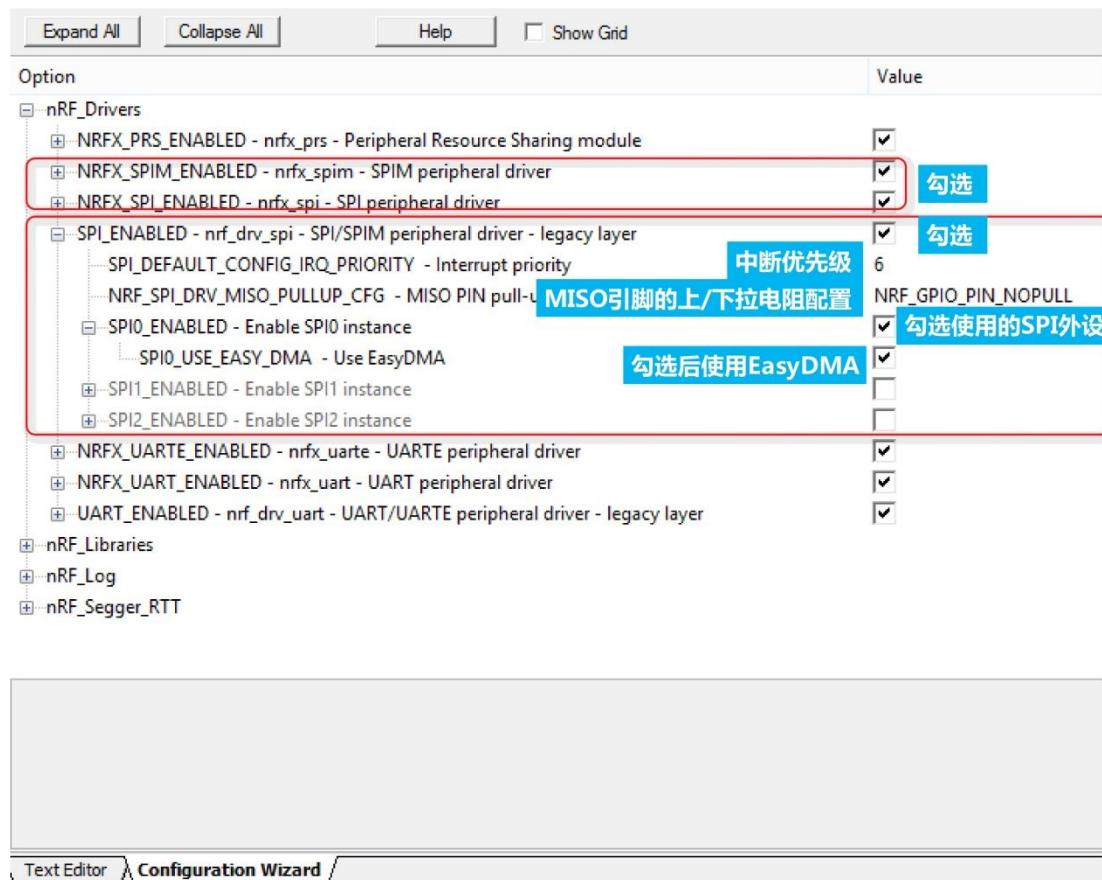


图 22-11：工程配置

这里要注意一下，正如我们前文所说，Nordic 的 SPI 库支持 SPI 和 SPIM，同时因为新/旧版本的关系，工程配置中有 3 个 SPI 的项目都需要勾选开启。

### 5.2.4. 代码编写

按照 TWI 的应用步骤，首先应定义和初始化 TWI 驱动程序实例，本例中使用 TWI0，因此 TWI 驱动程序的索引应为 0，程序清单如下。

## 代码清单：定义并初始化 SPI 驱动程序实例，使用 SPI0

```

1. //SPI 驱动程序实例 ID, ID 和外设编号对应, 0:SPI0 1:SPI1 2:SPI2
2. #define SPI_INSTANCE 0
3. //定义名称为 spi 的 SPI 驱动程序实例
4. static const nrf_drv_spi_t spi = NRF_DRV_SPI_INSTANCE(SPI_INSTANCE);

```

接着初始化配置 SPI，代码清单如下，一般地，为了方便，我们会在定义初始化配置结构体时用默认参数配置宏初始化配置结构体，之后根据需要去重写要修改的参数。如本例的初始化代码中，我们定义配置结构体 spi\_config 时用初始化宏 NRF\_DRV\_SPI\_DEFAULT\_CONFIG 初始化配置结构体，之后重写配置结构体 spi\_config 中 SPI 引脚的定义。

## 代码清单：初始化 SPI 实例

```

1. void SPI_Flash_Init(void)
2. {
3. //配置用于 SPI 片选的引脚为输出
4. nrf_gpio_cfg_output(SPI_SS_PIN);
5. //拉高 CS
6. SPIFlash_CS_HIGH;
7. //使用默认配置参数初始化 SPI 配置结构体
8. nrf_drv_spi_config_t spi_config = NRF_DRV_SPI_DEFAULT_CONFIG;
9. //重写 SPI 信号连接的引脚配置
10. spi_config.ss_pin = NRF_DRV_SPI_PIN_NOT_USED;
11. spi_config.miso_pin = SPI_MISO_PIN;
12. spi_config.mosi_pin = SPI_MOSI_PIN;
13. spi_config.sck_pin = SPI_SCK_PIN;
14. //初始化 SPI
15. APP_ERROR_CHECK(nrf_drv_spi_init(&spi, &spi_config, spi_event_handler, NULL));
16. }

```

SPI 默认参数配置宏定义如下，如果我们需要修改某个参数配置，可以修改初始化配置宏，更好的方式是和在 SPI\_Flash\_Init() 函数中重写需要修改的参数。

## 代码清单： SPI 默认参数配置宏

```

1. #define NRF_DRV_SPI_DEFAULT_CONFIG \
2. { \
3. .sck_pin = NRF_DRV_SPI_PIN_NOT_USED, \
4. .mosi_pin = NRF_DRV_SPI_PIN_NOT_USED, \
5. .miso_pin = NRF_DRV_SPI_PIN_NOT_USED, \
6. .ss_pin = NRF_DRV_SPI_PIN_NOT_USED, \
7. .irq_priority = SPI_DEFAULT_CONFIG_IRQ_PRIORITY, \
8. .orc = 0xFF, \

```

```

9. .frequency = NRF_DRV_SPI_FREQ_8M,
10. .mode = NRF_DRV_SPI_MODE_0,
11. .bit_order = NRF_DRV_SPI_BIT_ORDER_MSB_FIRST,
12. }

```

结合 SPI 默认参数初始化宏和 SPI\_Flash\_Init() 函数，我们可以看到，本例中我们配置的 SPI 参数如下：

- SCK 引脚：SPI\_SCK\_PIN，该宏的值为 22，即 SCK 连接到 P0.22。
- MOSI 引脚：SPI\_MOSI\_PIN，该宏的值为 20，即 SCK 连接到 P0.20。
- MISO 引脚：SPI\_MSIO\_PIN，该宏的值为 21，即 SCK 连接到 P0.21。
- CS 引脚：NRF\_DRV\_SPI\_PIN\_NOT\_USED，即 SPI 驱动程序不使用 CS 信号，CS 信号由应用程序管理，这么做的目的是为了发送数据时突破 255 个字节的限制。
- 中断优先级：SPI\_DEFAULT\_CONFIG\_IRQ\_PRIORITY，该宏的值为 6。
- ORC 寄存器内容：0xFF。
- SPI 速率：8Mbps。
- SPI 模式：模式 0。
- SPI 传输位序：MSB，即高位在前。
- SPI 工作于非阻塞模式，因为注册了事件回调函数。

本例中使用的是非阻塞工作模式，因此需要提供事件回调函数，同时定义一个变量 spi\_xfer\_done 用来标志 SPI 传输是否完成，应用程序传输数据后通过查询该标志即可判断传输是否完成。

#### 代码清单：SPI 事件回调函数

```

1. void spi_event_handler(nrf_drv_spi_evt_t const * p_event,
2. void * p_context)
3. {
4. //设置 SPI 传输完成
5. spi_xfer_done = true;
6. }

```

#### ■ 读取 W25Q128FV 芯片 ID

W25Q128FV 芯片的 ID 固定为 0xEF17，我们可以通过读取 ID 判断 W25Q128FV 是否在线，读取 ID 的时序如下图所示。读取 ID 执行的操作如下。

- 拉低片选信号 CS，使能 W25Q128FV。
- 发送扇区擦除命令 0x90。
- 发送 24 位地址（读取 ID 时固定为 0x000000），高地址在前。
- 发送 2 个字节数据（0xFF）读取 ID，读取的 ID 是接收数组的最后 2 个字节。
- 拉高片选信号 CS，释放 W25Q128FV。

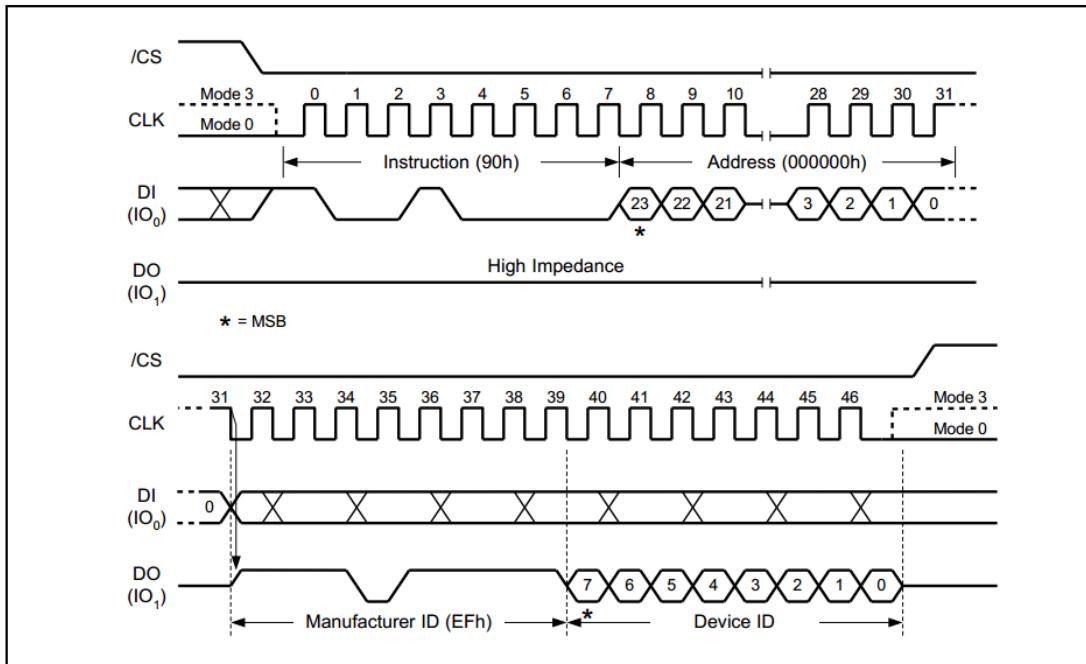


图 22-12: 读取 ID 时序

读取 ID 的函数代码清单如下：

#### 代码清单：读取 W25Q128FV 的 ID

```

1. ****
2. ** 描 述: 读取 W25Q128 芯片 ID
3. ** 参 数: 无
4. ** 返回值: 16 位 ID, W25Q128 芯片 ID 为: 0xEF17
5. ****
6. uint16_t SpiFlash_ReadID(void)
7. {
8. uint16_t dat = 0;
9. //准备数据, 读取 ID 需要发送 6 个字节数据: 90 + 000000 + FFFF
10. spi_tx_buf[0] = SPIFlash_ReadID; //读取 ID 的命令字 0x90
11. spi_tx_buf[1] = 0x00; //地址 16~23 位
12. spi_tx_buf[2] = 0x00; //地址 8~15 位
13. spi_tx_buf[3] = 0x00; //地址 0~7 位
14. spi_tx_buf[4] = 0xFF; //发送 2 个 0xFF 是为了读取 16 位的 ID
15. spi_tx_buf[5] = 0xFF;
16. //传输完成标志设置为 false
17. spi_xfer_done = false;
18. //拉低 CS, 使能 W25Q128FV
19. SPIFlash_CS_LOW;
20. //启动数据传输
21. APP_ERROR_CHECK(nrf_drv_spi_transfer(&spi, spi_tx_buf, 6, spi_rx_buf, 6));
22. //等待传输完成
23. while(!spi_xfer_done);

```

```

24. //拉高 CS, 释放 W25Q128FV
25. SPIFlash_CS_HIGH;
26. //接收数组最后两个字节才是读取的 ID
27. dat|=spi_rx_buf[4]<<8;
28. dat|=spi_rx_buf[5];
29. return dat;
30. }

```

### ■ 擦除扇区：擦除指定的扇区

#### ❖ Flash 为什么要擦除？

因为 Flash 的编程原理都是只能将各个 bit 由 1 写为 0，而不能将 0 写为 1，因此在 Flash 编程之前，为了保证写入的正确性，必须将对应的扇区擦除，擦除操作会将该扇区的内容全部恢复为 0xFF，这样执行写入操作就可以正确执行了。

W25Q128FV 支持扇区擦除、块擦除和全片擦除，W25Q128FV 的最小擦除单位为一个扇区，也就是每次至少擦除 4K 字节。我们在操作 Flash 的时候，要特别注意 Flash 编程时间和擦除时间，尤其是擦除时间，因为这些操作通常用时较长，程序中如果处理不好的话，可能会导致程序运行堵塞。W25Q128FV 编程时间和擦除时间如下表所示。

表 22-28: W25Q128FV 编程和擦除时间

| 描述                       | 符号               | 规格  |           |      | 单位 |
|--------------------------|------------------|-----|-----------|------|----|
|                          |                  | 最小值 | 典型值       | 最大值  |    |
| 字节编程时间 (第一个字节) (注 1)     | t <sub>BP1</sub> | —   | 30        | 50   | 微秒 |
| 另外的字节编程时间 (第一个字节后) (注 1) | t <sub>BP2</sub> | —   | 2.5       | 12   | 微秒 |
| 页编程时间                    | t <sub>PP</sub>  | —   | 0.7       | 3    | 毫秒 |
| 块擦除时间(4KB)               | t <sub>SE</sub>  | —   | 100<br>45 | 400  | 毫秒 |
| 块擦除时间(32KB)              | t <sub>BE1</sub> | —   | 120       | 1600 | 毫秒 |
| 块擦除时间(64KB)              | t <sub>BE2</sub> | —   | 150       | 2000 | 毫秒 |
| 全片擦除时间                   | t <sub>CE</sub>  | —   | 40        | 200  | 秒  |

注 1: 同一个页面内多个字节编程的时间为: t<sub>BPN</sub> = t<sub>BP1</sub>+ t<sub>BP2</sub> \* N (max), N=编程的字节数。

W25Q128FV 扇区擦除时序如下图所示，扇区擦除执行的操作如下，注意擦除扇区之前必须先发送“写使能”命令开启 W25Q128FV 的写使能。

- 拉低片选信号 CS，使能 W25Q128FV。
- 发送扇区擦除命令 0x20。
- 发送扇区 24 位地址，高地址在前。
- 拉高片选信号 CS，释放 W25Q128FV。

命令发送完成并不表示 W25Q128FV 已经执行完成扇区擦除操作，因此命令发送完后还需要通过查询状态寄存器的 BUSY 位来判断擦除操作是否完成，当 BUSY 位的值为 0 时表

示操作完成，W25Q128FV 就绪。

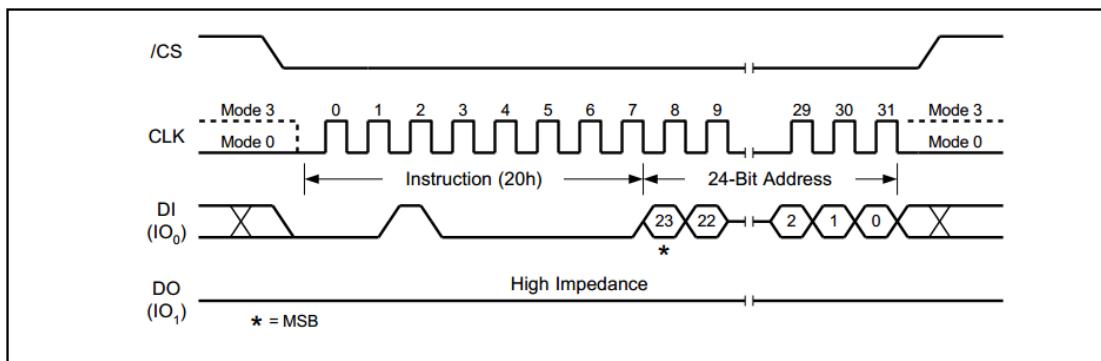


图 22-13：擦除扇区时序

根据 W25Q128FV 扇区擦除时序，编写代码如下：

### 代码清单：扇区擦除

```

1. ****
2. ** 描 述: 擦除扇区, W25Q128FV 最小的擦除单位是扇区
3. ** 参 数: [in]SecAddr: 扇区地址
4. ** 返回值: 无
5. ****
6. void SPIFlash_Erase_Sector(uint32_t SecAddr)
7. {
8. //发送写使能命令
9. SpiFlash_Write_Enable();
10. //扇区擦除命令
11. spi_tx_buf[0] = SPIFlash_SecErase;
12. //24 位地址
13. spi_tx_buf[1] = (uint8_t)((SecAddr&0x00ff0000)>>16);
14. spi_tx_buf[2] = (uint8_t)((SecAddr&0x0000ff00)>>8);
15. spi_tx_buf[3] = (uint8_t)SecAddr;
16. //传输完成标志设置为 false
17. spi_xfer_done = false;
18. //拉低 CS, 使能 W25Q128FV
19. SPIFlash_CS_LOW;
20. //启动数据传输: 发送长度 4 个字节, 读取长度 0 字节
21. APP_ERROR_CHECK(nrf_drv_spi_transfer(&spi, spi_tx_buf, SPIFLASH_CMD_LENGTH,
22. spi_tx_buf, 0));
23. //等待 SPI 传输完成
24. while(!spi_xfer_done);
25. //拉高 CS, 释放 W25Q128FV
26. SPIFlash_CS_HIGH;
27. //等待 W25Q128FV 完成操作
28. SpiFlash_Wait_Busy();
29. }
```

## ■ 全片擦除：擦除整个芯片

W25Q128FV 全片擦除时序如下图所示，全片擦除执行的操作如下。注意全片擦除之前必须先发送“写使能”命令开启 W25Q128FV 的写使能。

- 拉低片选信号 CS，使能 W25Q128FV。
- 发送扇区擦除命令 0XC7。
- 拉高片选信号 CS，释放 W25Q128FV。

全片擦除花费时间较长，典型时间是 40 秒，命令发送完后需要查询状态寄存器的 BUSY 位，直到 BUSY 位的值为 0（操作完成，W25Q128FV 就绪）才可以执行其它操作。

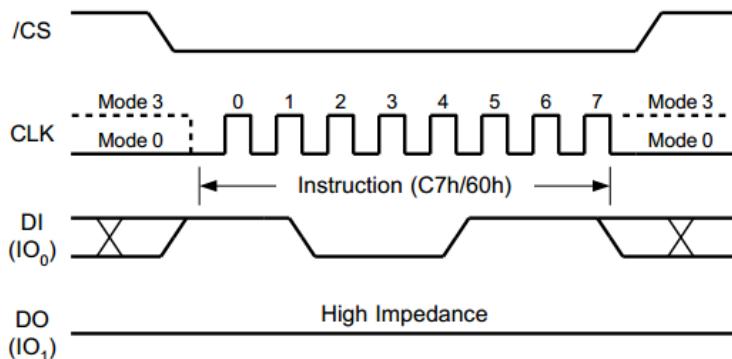


图 22-14：全片擦除时序

根据 W25Q128FV 全片擦除时序，代码清单如下：

### 代码清单：全片擦除

```

1. ****
2. ** 描 述: 全片擦除 W25Q128FV, 全片擦除所需的时间典型值为: 40 秒
3. ** 参 数: 无
4. ** 返回值: 无
5. ****
6. void SPIFlash_Erase_Chip(void)
7. {
8. //发送写使能命令
9. SpiFlash_Write_Enable();
10. //全片擦除命令
11. spi_tx_buf[0] = SPIFlash_ChipErase;
12. //传输完成标志设置为 false
13. spi_xfer_done = false;
14. //拉低 CS, 使能 W25Q128FV
15. SPIFlash_CS_LOW;
16. //启动数据传输: 发送长度 1 个字节, 读取长度 0 字节
17. APP_ERROR_CHECK(nrf_drv_spi_transfer(&spi, spi_tx_buf, 1, spi_tx_buf, 0));
18. while(!spi_xfer_done);
19. //拉高 CS, 释放 W25Q128FV
20. SPIFlash_CS_HIGH;
21. //等待 W25Q128FV 完成操作

```

```

22. SpiFlash_Wait_Busy();
23. }

```

### ■ 按页写：向指定页面连续写入不超过页面地址范围的数据

W25Q128FV 扇区擦除时序如下图所示，扇区擦除执行的操作如下，注意擦除扇区之前必须先发送“写使能”命令开启 W25Q128FV 的写使能。

- 拉低片选信号 CS，使能 W25Q128FV。
- 发送页编程命令 0x02。
- 发送 24 位地址，高地址在前。
- 发送写入到 Flash 的数据，注意最大写入的长度不能超过该地址所处页面的剩余空间。
- 拉高片选信号 CS，释放 W25Q128FV。

页编程时，数据传输完成并不表示 W25Q128FV 已经将接收的数据写入到自身的 Flash 内，因此数据传输完后还需要通过查询状态寄存器的 BUSY 位来判断编程是否完成，当 BUSY 位的值为 0 时表示编程完成，W25Q128FV 就绪。

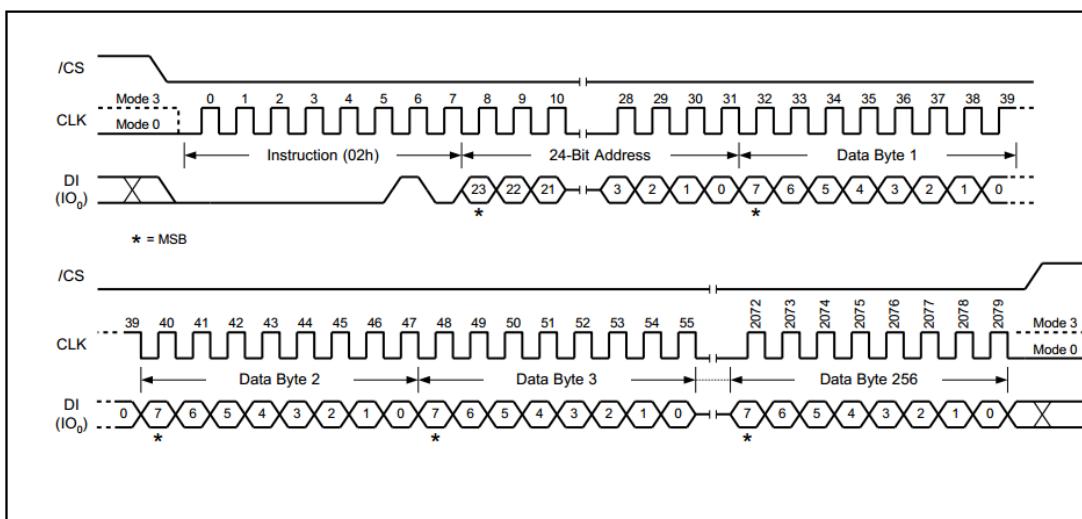


图 22-15: 页编程时序

根据 W25Q128FV 页编程时序，代码清单如下：

#### 代码清单：页编程

```

1. uint8_t SpiFlash_Write_Page(uint8_t *pBuffer, uint32_t WriteAddr, uint32_t size)
2. {
3. //检查写入的数据长度是否合法，写入长度不能超过页面的大小
4. if (size > (SPIFlash_PAGE_SIZE - (WriteAddr%SPIFlash_PAGE_SIZE)))
5. {
6. return NRF_ERROR_INVALID_LENGTH;
7. }
8. if (size == 0) return NRF_ERROR_INVALID_LENGTH;
9.
10. //发送写使能命令
11. SpiFlash_Write_Enable();

```

```

12. //页编程命令
13. spi_tx_buf[0] = SPIFlash_PageProgram;
14. //24位地址，高地址在前
15. spi_tx_buf[1] = (uint8_t)((WriteAddr&0x00ff0000)>>16);
16. spi_tx_buf[2] = (uint8_t)((WriteAddr&0x0000ff00)>>8);
17. spi_tx_buf[3] = (uint8_t)WriteAddr;
18. spi_tx_buf[4] = *pBuffer;
19.
20. //拉低 CS，使能 W25Q128FV
21. SPIFlash_CS_LOW;
22. //传输完成标志设置为 false
23. spi_xfer_done = false;
24. //启动数据传输
25. APP_ERROR_CHECK(nrf_drv_spi_transfer(&spi, spi_tx_buf, SPIFLASH_CMD_LENGTH+1,
26. spi_rx_buf, 0));
27. //等待 SPI 传输完成
28. while(!spi_xfer_done);
29. //传输完成标志设置为 false
30. spi_xfer_done = false;
31. //启动数据传输
32. APP_ERROR_CHECK(nrf_drv_spi_transfer(&spi, pBuffer+1, size-1, spi_rx_buf, 0))
33. ;
34. //等待 SPI 传输完成
35. while(!spi_xfer_done);
36. //拉高 CS，释放 W25Q128FV
37. SPIFlash_CS_HIGH;
38. //等待 W25Q128FV 完成操作
39. SpiFlash_Wait_Busy();
40. return NRF_SUCCESS;
41. }

```

### ■ 批量编程：向指定地址连续写入指定长度数据的功能，该功能实现了跨页写入。

批量编程是在页编程的基础上，将编程的数据拆分进行多次页编程，从而实现跨页编程，也就是实现从任意地址开始写入任意长度的数据，当然，地址范围和编程的数据长度不能超过 W25Q128FV 的自身的限制。

#### 代码清单：批量编程

```

1. uint8_t SpiFlash_Write_Buf(uint8_t *pBuffer, uint32_t WriteAddr, uint32_t size)
2. {
3. uint32_t PageByteRemain = 0;
4. //计算起始地址所处页面的剩余空间
5. PageByteRemain = SPIFlash_PAGE_SIZE - WriteAddr%SPIFlash_PAGE_SIZE;
6. //如果编程的数据长度不大于页面的剩余空间，编程数据长度等于 size
7. if(size <= PageByteRemain)

```

```
8. {
9. PageByteRemain = size;
10. }
11. //分次编程，直到所有的数据编程完成
12. while(true)
13. {
14. //编程 PageByteRemain 个字节
15. SpiFlash_Write_Page(pBuffer,WriteAddr,PageByteRemain);
16. //如果编程完成，退出循环
17. if(size == PageByteRemain)
18. {
19. break;
20. }
21. else
22. {
23. //计算编程取数据的缓存地址
24. pBuffer += PageByteRemain;
25. //计算编程地址
26. WriteAddr += PageByteRemain;
27. //数据长度减去 PageByteRemain
28. size -= PageByteRemain;
29. //计算下次编程的数据长度
30. if(size > SPIFlash_PAGE_SIZE)
31. {
32. PageByteRemain = SPIFlash_PAGE_SIZE;
33. }
34. else
35. {
36. PageByteRemain = size;
37. }
38. }
39. }
40. return NRF_SUCCESS;
41. }
```

#### ■ 批量读：从指定地址连续读取指定长度数据，并将读取的数据通过串口输出

W25Q128FV 本身支持连续读任意长度数据，甚至可以通过读命令一次将整个 Flash 内容读取出来，但是正如前文所述，nRF52840 的 SPIM 带的 EasyDMA 最大数据收发长度是 255 个字节，也就是一次最多只能读取 255 个字节，因此批量读取的时候同样需要分次读取，代码清单如下。

#### 代码清单：批量读取

```
1. uint8_t SpiFlash_Read(uint8_t *pBuffer,uint32_t ReadAddr,uint32_t size)
2. {
```

```
3. uint8_t read_size;
4. spi_tx_buf[0] = SPIFlash_ReadData;
5. //24位地址，高地址在前
6. spi_tx_buf[1] = (uint8_t)((ReadAddr&0x00ff0000)>>16);
7. spi_tx_buf[2] = (uint8_t)((ReadAddr&0x0000ff00)>>8);
8. spi_tx_buf[3] = (uint8_t)ReadAddr;
9.
10. //拉低 CS，使能 W25Q128FV
11. SPIFlash_CS_LOW;
12. //传输完成标志设置为 false
13. spi_xfer_done = false;
14. //启动数据传输
15. APP_ERROR_CHECK(nrf_drv_spi_transfer(&spi, spi_tx_buf, SPIFLASH_CMD_LENGTH,
16. spi_rx_buf, 0));
17. //等待 SPI 传输完成
18. while(!spi_xfer_done);
19. //开始读取数据
20. while(size!=0)
21. {
22. if(size<=SPI_TXRX_MAX_LEN)
23. {
24. read_size = size;
25. size = 0;
26. }
27. else
28. {
29. read_size = SPI_TXRX_MAX_LEN;
30. size -= SPI_TXRX_MAX_LEN;
31. }
32. //传输完成标志设置为 false
33. spi_xfer_done = false;
34. //启动数据传输
35. APP_ERROR_CHECK(nrf_drv_spi_transfer(&spi, spi_tx_buf, 0, pBuffer, read_s
36. ize));
37. //等待 SPI 传输完成
38. while(!spi_xfer_done);
39. pBuffer += read_size;
40. }
41. //拉高 CS，释放 W25Q128FV
42. SPIFlash_CS_HIGH;
43. return NRF_SUCCESS;
44. }
```

### 5.2.5. 硬件连接

本实验需要使用 LED 指示灯、按键、UART 和外接 W25Q128FV 模块，按照下图所示短接跳线帽和接入 W25Q128FV 模块。

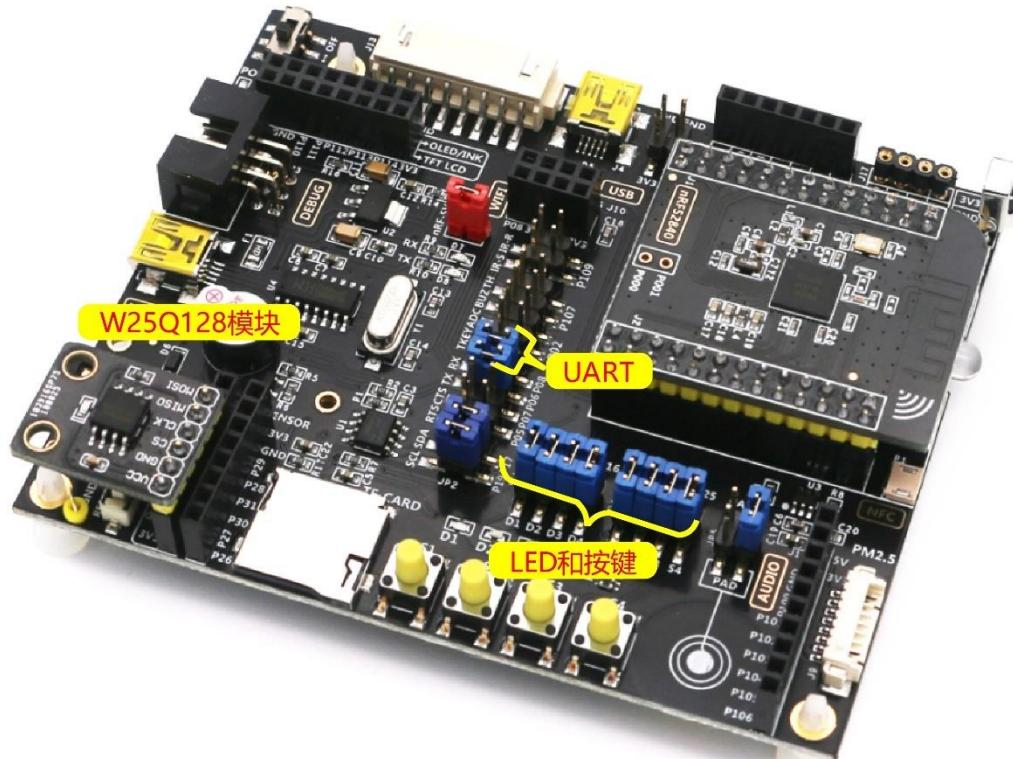


图 22-16：开发板跳线帽短接和 W25Q128FV 模块安装

### 5.2.6. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 22-1: SPI 读写 W25Q128FV-非阻塞模式”，将解压后得到的文件夹“spi\_w25q128”拷贝到合适的目录，如“D\ nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\spi\_w25q128\project\mdk5”目录下的工程“spi\_w25q128.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nRF52840\_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 计算机上打开串口调试助手，注意勾选“HEX 显示”，程序运行后，分别按下 S1、S2 和 S3 按键，执行读写 W25Q128FV。
- 按下 S1 按键后松开按键：按页编程，向扇区 0 的第一页写入 256 个字节，之后读出数据并通过串口输出。

- 按下 S2 按键后松开按键：批量写入和读出，向起始地址 100 连续写入 500 个字节数据，之后读出数据并通过串口输出。
- 按下 S4 按键后松开按键：全片擦除 W25Q128FV，全片擦除所需的时间典型值为：40 秒。

### 5.3. W25Q128FV 读写（SPIM0 阻塞）实验

本实验在“实验 22-1：SPI 读写 W25Q128FV-非阻塞模式”的基础上修改。SPI 的模式设置为阻塞模式，程序的功能和“实验 22-1”一样。

❖ 注：本节对应的实验源码是：“实验 22-1：SPI0 读写 W25Q128FV-阻塞模式”。

程序中需要修改的地方如下：

1. 调用 SPI 初始化函数 nrf\_drv\_spi\_init() 初始化 SPI 时，事件句柄设置为 NULL，如下所示。

#### 代码清单：初始化 SPI 为阻塞模式

1. //初始化 SPI
2. APP\_ERROR\_CHECK(nrf\_drv\_spi\_init(&spi, &spi\_config, **NULL**, **NULL**));
2. 删除事件回调函数 spi\_event\_handler()：因为阻塞模式不需要事件句柄，因此应用程序无需提供 SPI 事件回调函数。
3. 删除掉读写函数中 SPI 传输完成标志清零的语句 “spi\_xfer\_done == false;” 和等待 SPI 传输完成的语句 “while (spi\_xfer\_done == false){ };”。

修改后，编译并下载到开发板测试，实验步骤和“实验 22-1”一样。

#### ■ 思考题 1：如何使用 SPIM1 和 SPIM2 读写 W25Q128FV？

提示：使用 SPIM1 时需要将 SPI 驱动程序实例的索引 SPI\_INSTANCE 的值修改为 1，使用 SPIM2 时需要将 SPI 驱动程序实例的索引 SPI\_INSTANCE 的值修改为 2，“sdk\_config.h”文件中开启对应的 SPIM 外设，实验源码：

- 实验 22-3：SPIM1 读写 W25Q128FV-非阻塞模式
- 实验 22-4：SPIM2 读写 W25Q128FV-非阻塞模式

## 6. 软件设计-SPIM 库

### 6.1. 库函数的应用

SPIM 库是 SPIM 专用，对于 nRF52840 来说 SPIM0~SPIM2 可以使用 SPI 库，也可以使用 SPIM 库，而 SPIM3 只能使用 SPIM3 库。

SPIM 库的应用步骤和 SPI 类似，如下图所示主要区别是 SPIM 没有在“sdk\_config.h”中配置是否使用 EasyDMA 的步骤，因为 SPIM 只能使用 EasyDMA 传输数据。

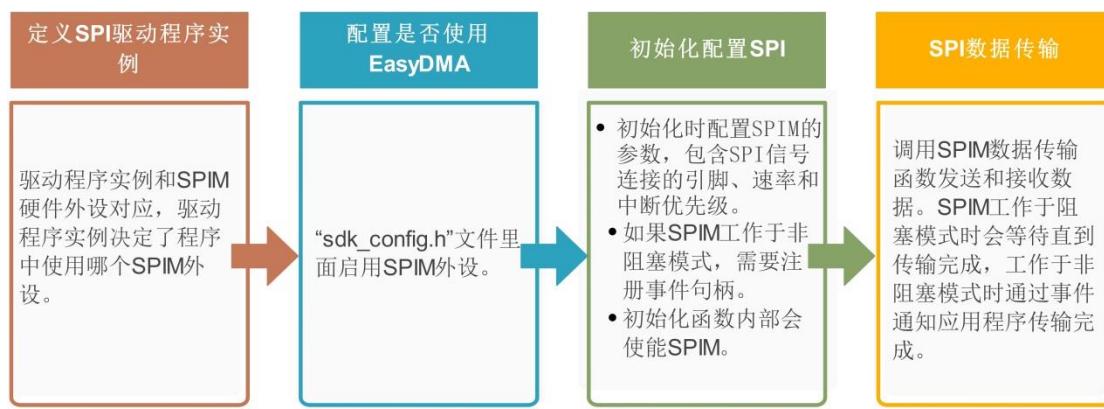


图 22-17: SPIM 库应用步骤

### 6.1.1. 定义 SPIM 驱动程序实例

SPIM 驱动程序实例使用 nrfx\_spim\_t 结构体定义，该结构体描述了具体的 SPIM 外设，当我们定义了 nrfx\_spim\_t 类型的变量并对其赋值后，该变量就对应了一个具体的硬件 SPIM 外设。

定义 SPIM 驱动程序实例代码如下，初始化宏 NRFX\_SPIM\_INSTANCE 的输入参数对应 SPI 外设的编号。

#### 代码清单：定义 SPIM 驱动程序实例

```

1. //SPIM 驱动程序实例 ID, ID 和外设编号对应, 0:SPIM0 1:SPIM1 2:SPIM2 3:SPIM3
2. #define SPI_INSTANCE 0
3. //定义 SPI 驱动程序实例
4. static const nrfx_spim_t 驱动实例名称= NRFX_SPIM_INSTANCE(SPI_INSTANCE);

```

### 6.1.2. 启用 SPIM 实例

定义了 SPIM 驱动程序实例后，还需要在 sdk\_config.h 启用该 SPIM 实例，如下图所示。需要注意的是，如果自己的工程里面没有 SPI 的配置项目，需要手动从例子程序的 sdk\_config.h 文件中将 SPI 这部分配置拷贝到自己工程的 sdk\_config.h 文件里面。

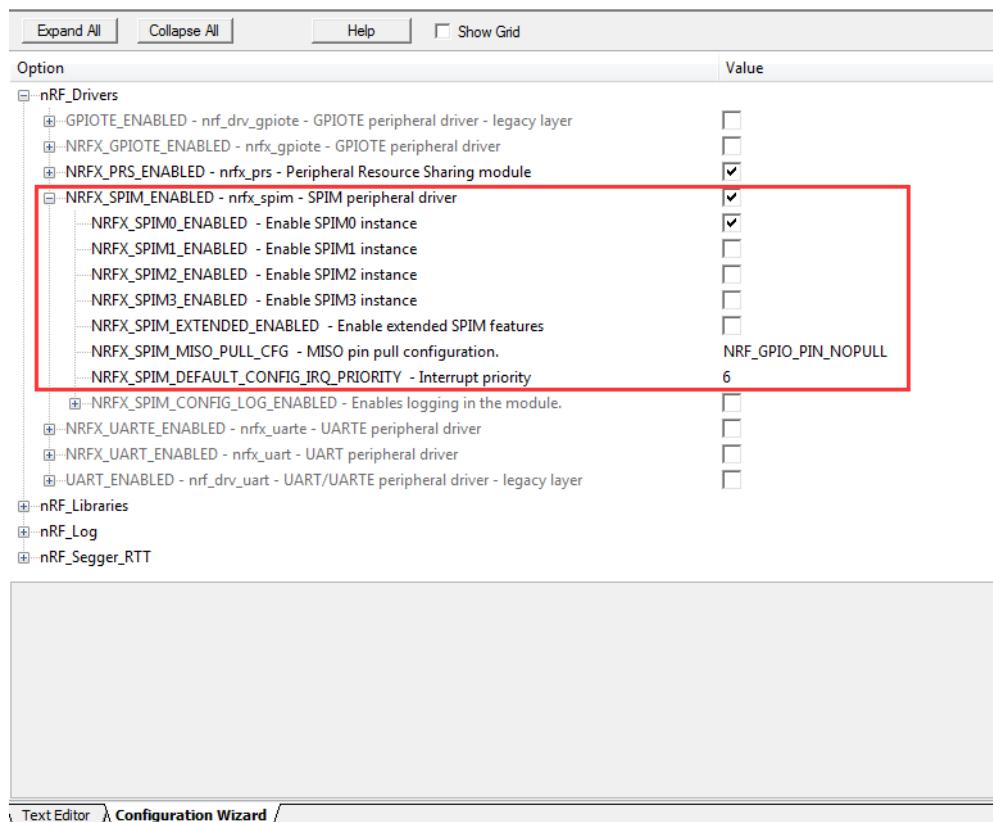


图 22-18: sdk\_config.h 启用该 SPIM 实例

从上图中可以看到，sdk\_config.h 文件里面主要配置项目如下：

- 启用 SPIM 实例，根据定义的 SPIM 驱动程序实例启用对应的实例。
- NRFX\_SPIM\_EXTENDED\_ENABLED：只有 SPIM3 才可以勾选。
- NRFX\_SPIM\_MISO\_PULL\_CFG：配置 SPIM 的 MISO 引脚的上拉电阻，这个配置项目在 SD 卡应用中需要特别注意，因为 SD 使用 SPI 访问时，MISO 必须上拉。
- 配置中断优先级。

### 6.1.3. 初始化 SPIM

SPIM 初始化的库函数是 nrfx\_spim\_init() 函数，该函数初始化 SPIM 后会在函数内部使能 SPIM，函数原型如下。

表 22-29: nrfx\_spim\_init() 函数

|      |                                                                                                                                                                              |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 函数原型 | nrfx_err_t nrfx_spim_init<br>(<br>nrfx_spim_t const * const p_instance,<br>nrfx_spim_config_t const * p_config,<br>nrfx_spim_evt_handler_t handler,<br>void * p_context<br>) |
| 函数功能 | 初始化 SPIM 驱动程序实例，该函数配置并且使能指定的 SPIM 外设。                                                                                                                                        |
| 参 数  | [in] p_instance：指向驱动程序实例结构体。                                                                                                                                                 |

|     |                                                                                                                                                                                 |
|-----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|     | <p>[in] <b>p_config</b>: 指向初始化配置结构体。</p> <p>[in] <b>handler</b>: 指向事件句柄，如果为 NULL，SPIM 工作于阻塞模式。</p> <p>[in] <b>p_context</b>: 传递给事件句柄的 Context。</p>                              |
| 返回值 | <p>NRF_SUCCESS: 初始化成功。</p> <p>NRF_ERROR_INVALID_STATE: 驱动程序实例已经初始化。</p> <p>NRF_ERROR_BUSY: 其它具有相同 ID 的外设已经在使用了，只有当 PERIPHERAL_RESOURCE_SHARING_ENABLED 设置为非 0 时才可能 会出现这种情况。</p> |

## ■ 初始化结构体

调用 nrfx\_spim\_init() 函数初始化 SPIM 时需要提供初始化结构体变量作为函数的参数，初始化结构体包含了 SPIM 初始化时所需要配置的参数，其声明如下：

代码清单：SPIM 初始化配置结构体 nrfx\_spim\_config\_t 声明

```

1. typedef struct
2. {
3. uint8_t sck_pin; //SPIM SCK 信号连接的引脚
4. //SPIM MOSI 信号连接的引脚(可选)，如果不需 MOSI，设置为 NRFX_SPI_MOSI_PIN_NOT_USED
5. uint8_t mosi_pin;
6. //SPIM MISO 信号连接的引脚(可选)，如果不需 MISO，设置为 NRFX_SPI_MISO_PIN_NOT_USED
7. uint8_t miso_pin;
8. //SPIM SS 信号连接的引脚(可选)，如果不需 SS，设置为 NRFX_SPI_SS_PIN_NOT_USED
9. uint8_t ss_pin;
10. bool ss_active_high; //SS 在 SPIM 传输时的极性
11. uint8_t irq_priority; //中断优先级
12. uint8_t orc; //ORC 寄存器内容
13. nrf_spim_frequency_t frequency; //SPIM 速率
14. nrf_spim_mode_t mode; //SPIM 模式
15. nrf_spim_bit_order_t bit_order; //SPIM 传输位序
16. //下面的配置项目必须 NRFX_SPI_EXTENDED_ENABLED 才能使用，且只有 SPI3 才可以使用
17. #if NRFX_CHECK(NRFX_SPI_EXTENDED_ENABLED) || defined(__NRFX_DOXYGEN__)
18. //SPIM DCX 信号连接的引脚(可选)
19. uint8_t dcx_pin;
20. //MISO 上输入串行数据的采样延迟。该值指定从 SCK 的采样边沿 (CONFIG.CPHA = 0 的前沿,
21. //CONFIG.CPHA = 1 的后沿) 延迟到输入串行数据采样的 64 MHz 时钟周期(15.625 ns)的数量
22. uint8_t rx_delay;
23. //是否使用硬件片选
24. bool use_hw_ss;
25. //CSN 的边沿与 SCK 的边沿之间的最小持续时间以及 CSN 必须在 SPI 事务之间保持高电平最小
26. //持续时间。单位是 64 MHz 时钟周期 (15.625 ns)
27. uint8_t ss_duration;
28. #endif

```

```
29. } nrfx_spim_config_t;
```

由 nrfx\_spim\_config\_t 声明可以看到，SPI 初始化时可以配置的 SPIM 参数如下：

- SPIM 信号连接的物理引脚：可以设置连接到任意引脚，但是两个 SPIM 信号不能连接到同一个引脚，如 MISO 和 MOSI 不能连接到同一个引脚。不同 SPI 外设的信号也不能连接到同一个引脚，如 SPIM0 的 MISO 和 SPIM1 的 MISO 不能连接到同一个引脚。另外仅有 SPIM3 具有硬件片选，SPIM0~SPIM2 外设本身是没有片选信号（CS）的，SPIM0~SPIM2 设置了 CS 连接的引脚表示 CS 由 SPI 驱动程序管理。如果设置 CS 为“NRF\_DRV\_SPI\_PIN\_NOT\_USED”，那么应用程序就需要自己管理 CS。
- SPI 速率：SPI 速率定义如下，最快速率 SPIM0~SPIM2 最大数据速率 8 Mbps，SPIM3 最大数据速率 32Mbps。

#### 代码清单：SPIM 数据速率定义

```
1. typedef enum
2. {
3. NRF_SPIM_FREQ_125K = SPIM_FREQUENCY_FREQUENCY_K125, //数据速率 125 kbps
4. NRF_SPIM_FREQ_250K = SPIM_FREQUENCY_FREQUENCY_K250, //数据速率 250 kbps.
5. NRF_SPIM_FREQ_500K = SPIM_FREQUENCY_FREQUENCY_K500, //数据速率 500 kbps.
6. NRF_SPIM_FREQ_1M = SPIM_FREQUENCY_FREQUENCY_M1, //数据速率 1 Mbps.
7. NRF_SPIM_FREQ_2M = SPIM_FREQUENCY_FREQUENCY_M2, //数据速率 2 Mbps.
8. NRF_SPIM_FREQ_4M = SPIM_FREQUENCY_FREQUENCY_M4, //数据速率 4 Mbps.
9. NRF_SPIM_FREQ_8M = (int)SPIM_FREQUENCY_FREQUENCY_M8, //数据速率 8 Mbps.
10. //以下的数据速率仅 SPIM3 可以使用
11. #if defined(SPIM_FREQUENCY_FREQUENCY_M16) || defined(__NRFX_DOXYGEN__)
12. NRF_SPIM_FREQ_16M = SPIM_FREQUENCY_FREQUENCY_M16, //数据速率 16 Mbps.
13. #endif
14. #if defined(SPIM_FREQUENCY_FREQUENCY_M32) || defined(__NRFX_DOXYGEN__)
15. NRF_SPIM_FREQ_32M = SPIM_FREQUENCY_FREQUENCY_M32 //数据速率 32 Mbps.
16. #endif
17. } nrf_spim_frequency_t;
```

- SPIM 模式：SPIM 可配置为模式 0~模式 3，模式定义如下。

#### 代码清单：SPIM 模式定义

```
1. typedef enum
2. {
3. NRF_SPIM_MODE_0, //模式 0: 总线空闲时 SCK 为低电平,第 1 个时钟沿采样
4. NRF_SPIM_MODE_1, //模式 1: 总线空闲时 SCK 为低电平,第 2 个时钟沿采样
5. NRF_SPIM_MODE_2, //模式 2: 总线空闲时 SCK 为高电平,第 1 个时钟沿采样
6. NRF_SPIM_MODE_3 //模式 3: 总线空闲时 SCK 为高电平,第 2 个时钟沿采样
7. } nrf_spim_mode_t;
```

- SPIM 位序：SPIM 的位序可配置为 MSB 或者 LSB，大多数情况下，用的都是 MSB，

位序的定义如下：

### 代码清单：SPIM 位序定义

```
1. typedef enum
2. {
3. NRF_SPIM_BIT_ORDER_MSB_FIRST = SPIM_CONFIG_ORDER_MsbFirst, //MSB
4. NRF_SPIM_BIT_ORDER_LSB_FIRST = SPIM_CONFIG_ORDER_LsbFirst //LSB
5. } nrf_spim_bit_order_t;
```

- ORC 内容：ORC 寄存器内容一般设置为 0xFF，当发送长度小于接收数据长度时，驱动程序继续发送 ORC 寄存器的内容以读取剩余的数据。
- 中断优先级：可设置的中断优先级位 0~7，BLE 工程中，优先级 0、1、4、5 保留给了 SoftDevice 使用，SPIM 不能使用这些中断优先级。

### ■ 事件句柄

应用程序是否提供事件句柄决定了 SPIM 工作与阻塞模式还是非阻塞模式。

- 应用程序提供事件句柄：SPIM 工作于非阻塞模式。
- 应用程序不提供事件句柄（handler 设置为 NULL）：SPI 工作于阻塞模式。

如果我们需要 SPI 工作于非阻塞模式，就需要提供事件回调函数，并在初始化时将事件句柄作为 nrfx\_spim\_init() 函数的参数传递给函数，由初始化函数完成注册。SPIM 事件回调函数的编写格式如下，我们编写 SPIM 事件回调函数时需要遵循该格式。

### 代码清单：SPI 事件回调函数编写格式

```
1. void spi_event_handler(nrfx_spim_evt_t const * p_event,
2. void * p_context)
3. {
4. //功能代码
5. }
```

#### 6.1.4. 数据传输

SPIM 驱动程序提供的用于数据传输的函数是 nrfx\_spim\_xfer()，函数原型如下表所示，SPI 是全双工回环总线，一个函数即可完成数据收发。

表 22-30: nrf\_drv\_spi\_transfer() 函数

|      |                                                                                                                                             |
|------|---------------------------------------------------------------------------------------------------------------------------------------------|
| 函数原型 | nrfx_err_t nrfx_spim_xfer<br>(<br>nrfx_spim_t const *const p_instance,<br>nrfx_spim_xfer_desc_t const * p_xfer_desc,<br>uint32_t flags<br>) |
| 函数功能 | 启动 SPIM 数据传输。使用 flags 参数提供了其他选项<br>NRFX_SPIM_FLAG_TX_POSTINC 和 NRFX_SPIM_FLAG_RX_POSTINC：缓存区                                                |

|     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|     | <p>地址后递增，仅 SPIM 支持。</p> <p><b>NRFX_SPIM_FLAG_HOLD_XFER:</b> SPIM 驱动不启动传输，当传输由 PPI 在外部触发时，使用此标志，仅 SPIM 支持。</p> <p><b>NRFX_SPIM_FLAG_NO_XFER_EVT_HANDLER:</b> 传输完成后没有用户事件处理程序，这也意味着在传输结束时没有中断，仅 SPIM 支持。如果使用了 <b>NRFX_SPIM_FLAG_NO_XFER_EVT_HANDLER</b>，则驱动程序不会将 SPIM 实例设置为忙状态，因此必须确保在 SPIM 未处于活动状态时设置下一次传输。<br/> <b>nrfx_spim_end_event_get()</b> 函数可用于检测传输结束，该选项可与 <b>NRFX_SPIM_FLAG_REPEAT_XFER</b> 一起使用，以准备一系列 SPI 传输而不会中断。</p> <p><b>NRFX_SPIM_FLAG_REPEAT_XFER:</b> 准备重复传输，该标志使得我们可以建立多个通过外部触发的传输（如通过 PPI 触发）。一个示例是具有 <b>NRFX_SPIM_FLAG_RX_POSTINC</b>, <b>NRFX_SPIM_FLAG_NO_XFER_EVT_HANDLER</b> 和 <b>NRFX_SPIM_FLAG_REPEAT_XFER</b> 多个选项的 TXRX 传输的例子。在设置传输之后，通过 PPI 即可触发一组传输，这些传输将读取例如外部部件的相同寄存器并将其放入 RAM 缓冲区而没有任何中断。<br/> <b>nrfx_spim_end_event_get()</b> 函数可用于检测传输结束，统计传输次数。如果使用了 <b>NRFX_SPIM_FLAG_REPEAT_XFER</b>，则驱动程序不会将 SPIM 实例设置为忙状态，因此必须确保在 SPIM 未处于活动状态时设置下一次传输。</p> |
| 参 数 | <p>[in] <b>p_instance</b>: 指向驱动程序实例结构体。</p> <p>[in] <b>p_xfer_desc</b>: 指向 SPIM 传输描述符。</p> <p>[in] <b>flags</b>: 传输选项 (0=使用默认设置)。</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| 返回值 | <p><b>NRF_SUCCESS</b>: 操作成功。</p> <p><b>NRF_ERROR_BUSY</b>: 驱动未准备好传输新的数据。</p> <p><b>NRFX_ERROR_NOT_SUPPORTED</b>: 驱动程序不支持提供的参数。</p> <p><b>NRF_ERROR_INVALID_ADDR</b>: 提供的缓存没有位于数据 RAM 区域。</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |

使用 SPIM 库传输数据时，操作方式和 SPI 库略有差别，SPIM 库传输数据时需要先定义一个 `nrfx_spim_xfer_desc_t` 类型的 SPIM 传输描述符，该描述符中设置收/发缓存和收/发长度，之后以该描述符为参数调用 `nrfx_spim_xfer()` 函数发起传输。SPIM 传输描述符的声明如下：

#### 代码清单：SPI 事件回调函数编写格式

```

1. typedef struct
2. {
3. uint8_t const * p_tx_buffer; //指向发送缓存
4. size_t tx_length; //发送的数据长度
5. uint8_t * p_rx_buffer; //指向接收缓存

```

```

6. size_t rx_length; //接收的数据长度
7. } nrfx_spim_xfer_desc_t;

```

## 6.2. W25Q128FV 读写（SPIM3 非阻塞）实验

本实验在“实验 10-1：串口数据收发”的基础上修改。通过 SPIM3 读写 SPI 接口的存储器 W25Q128FV，本例中设置的 SPIM3 传输速率是 32Mbps。

◇ 注：本节对应的实验源码是：“实验 22-5：SPIM3（32M）读写 W25Q128FV-非阻塞模式”。

### 6.2.1. 添加需要的文件

本例需要添加 SPIM 驱动文件和 W25Q128FV 驱动文件，需要加入的文件如下表所示。

表 20-31：SPIM 需要加入的文件

| 文件名         | SDK 中的目录                    | 描述                    |
|-------------|-----------------------------|-----------------------|
| nrfx_spim.c | ..\modules\nrfx\drivers\src | SPI 主机驱动文件。           |
| w25q128.c   | ..\app\w25q128              | 编写的 W25Q128FV 驱动程序文件。 |

### 6.2.2. 头文件引用和路径设置

#### 1. 需要引用的头文件

因为在“w25q128.c”文件中使用了 SPIM 程序模块和我们编写的 W25Q128FV 驱动程序，所以“w25q128.c”文件中“main.c”文件需要引用下面的头文件。

```
#include "nrfx_spim.h"
```

```
#include "w25q128.h"
```

“main.c”文件中使用了 W25Q128FV 驱动程序，因此需要引用下面的头文件。

```
#include "w25q128.h"
```

#### 2. 需要添加的头文件包含路径

MDK 中点击魔术棒，打开工程配置窗口，按照下图所示添加头文件包含路径。

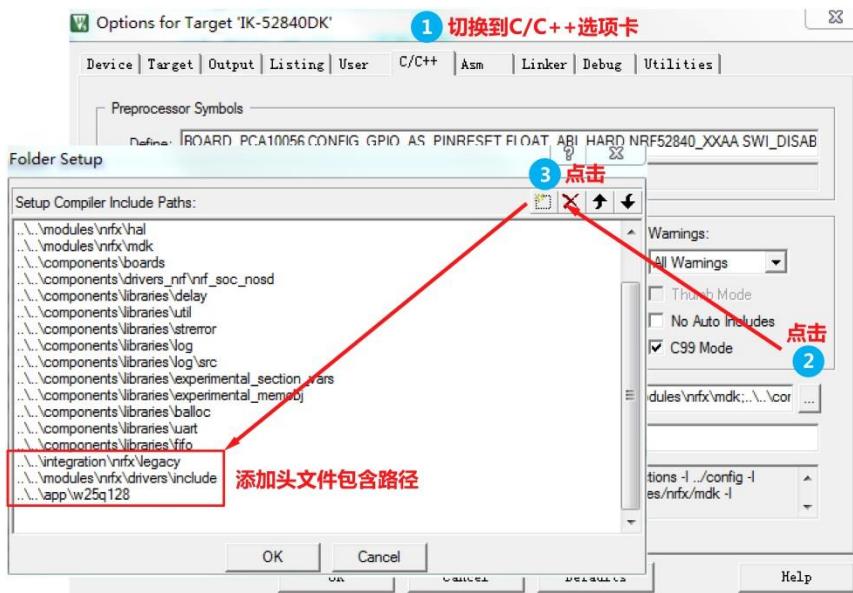


图 22-19：添加头文件包含路径

本例需要包含 SPI 驱动头文件路径和 W25Q128FV 驱动头文件路径，需要添加的头文件路径如下表：

表 22-32：头文件包含路径

| 序号 | 路径                                 |
|----|------------------------------------|
| 1  | ..\..\modules\nrfx\drivers\include |
| 2  | ..\..\app\w25q128                  |

### 6.2.3. 工程配置

打开“sdk\_config.h”文件，加入 SPIM 需要的配置，如下图所示，编辑内容时切换到“Text Editor”进行编辑（编辑的内容参考实验的源码，因此代码很长，此处不贴出代码，参见“sdk\_config.h”文件的（144~260）行），编辑完成后，切换到“Configuration Wizard”，即可观察到配置的具体项目：

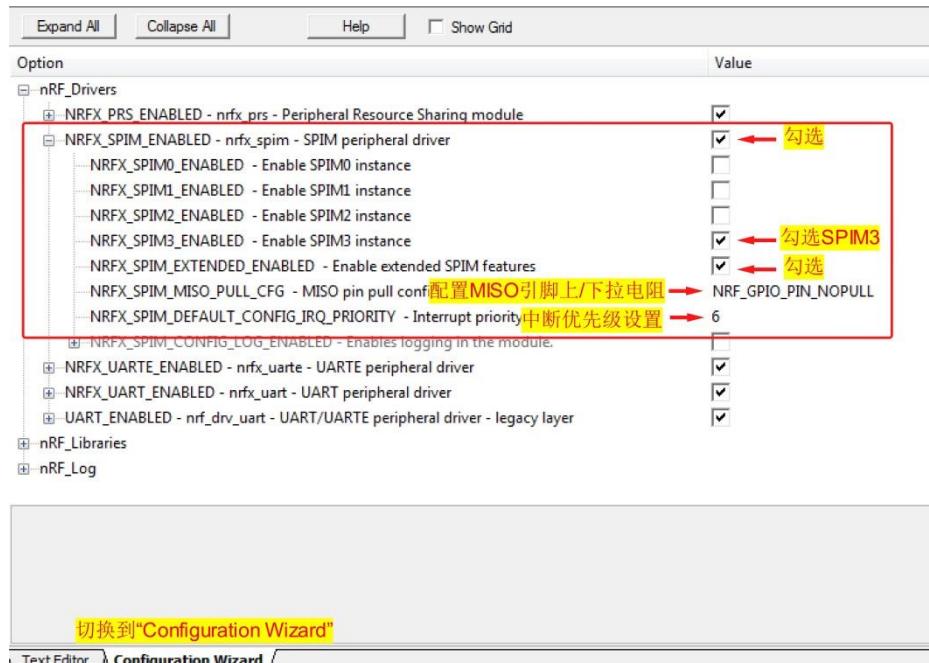


图 22-20：工程配置

这里要注意一下，SPIM 库和 SPI 库的配置是不一样的，使用 SPIM 库只需要配置如上图所示的 SPIM 这一个项目就可以了。

### 6.2.4. 代码编写

使用 SPIM 库编写的 W25Q128FV 驱动代码和 SPI 库类似，这里不再贴出代码，读者可以阅读“试验 22-5”的代码，代码里面都有详细的注释。

### 6.2.5. 实验步骤

- 解压“…\3：开发指南（上册）配套实验源码\”目录下的压缩文件“实验 22-5：SPIM3（32M）读写 W25Q128FV-非阻塞模式”，将解压后得到的文件夹“spim3\_w25q128”拷贝到合适的目录，如“D\NRF52840”。
- 启动 MDK5.23。

3. 在 MDK5 中执行 “Project→Open Project” 打开 “…\spim3\_w25q128\project\mdk5” 目录下的工程 “spim3\_w25q128.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件 “nRF52840\_qiaa.hex” 位于工程目录下的 “Objects” 文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 计算机上打开串口调试助手，注意勾选 “HEX 显示”，程序运行后，分别按下 S1、S2 和 S3 按键，执行读写 W25Q128FV。
  - 按下 S1 按键后松开按键：按页编程，向扇区 0 的第一页写入 256 个字节，之后读出数据并通过串口输出。
  - 按下 S2 按键后松开按键：批量写入和读出，向起始地址 100 连续写入 500 个字节数据，之后读出数据并通过串口输出。
  - 按下 S4 按键后松开按键：全片擦除 W25Q128FV，全片擦除所需的时间典型值为：40 秒。

## 第二十三章：PWM 脉冲宽度调制

### 1. 学习目的

- 掌握 PWM 的基本概念。
- 了解 nRF52840 的 PWM 的特点、PWM 周期和占空比的计算。
- 掌握 PWM 库的应用步骤、以及 4 种装载模式的运用。
- 掌握 PWM 序列控制以及单个序列和复合序列的播放。

### 2. PWM 原理

PWM (全称是 Pulse Width Modulation)脉冲宽度调制是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术，广泛应用在从测量、通信到功率控制与变换的许多领域中。

所谓的脉冲宽度调制，也就是输出占空比可变的脉冲波形。下图是一个 PWM 控制模拟电路（电压是模拟量）的示例，当开关直接接通时，电灯供电电压为 9V，当使用 PWM 脉冲控制开关的接通和闭合时，PWM 占空比为 50% 的情况下，可等效为 4.5V 电池给电灯供电，电灯的亮度会有所下降。同理，占空比是 10%，等效于 0.9V 电池给电灯供电，亮度会很暗。由此，我们可以看到，通过改变 PWM 信号的频率和占空比可以控制电灯的状态和亮度。

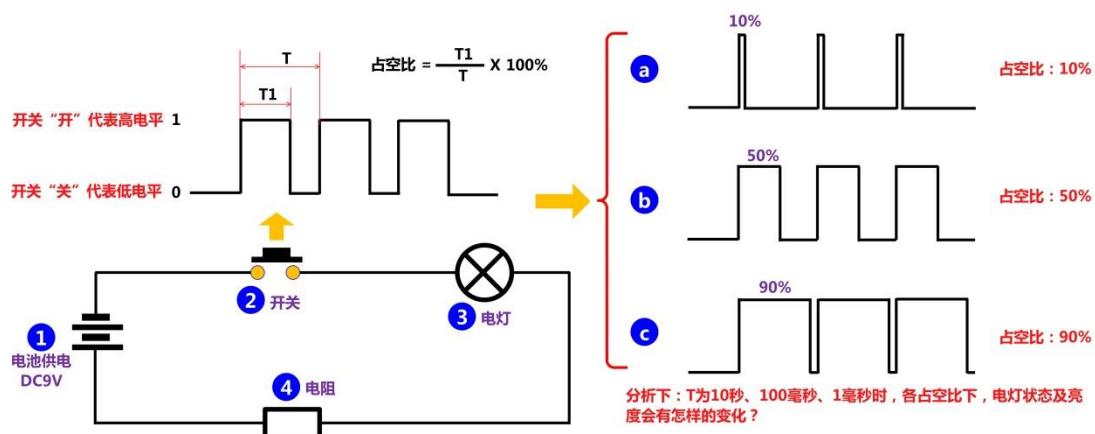


图 23-1：PWM 控制模拟电路示意图

#### ■ PWM 的重要概念

##### 1) 占空比

一个 PWM 周期中，高电平保持的时间与该 PWM 的周期的时间之比，如一个 PWM 的周期是 10ms，高电平的时间是 1ms，那么占空比就是 10%。

##### 2) 边沿对齐、中心对齐、单斜率和双斜率

- 单斜率：PWM 计数器从 0 开始计数到最大值，达到最大值后清零重新从 0 开始计数，如此反复。单斜率和 PWM 比较值实现的是边沿对齐。

- 双斜率：PWM 计数器从 0 开始计数到最大值，再从最大值计数到 0，如此反复。双斜率和 PWM 比较值实现的是中心对齐。

### 3) 分辨率

占空比最小能达的数值，如 8 位的 PWM，理论的分辨率就是 1:255(单斜率)，16 位的 PWM 理论就是 1:65535(单斜率)。

### ■ PWM 信号控制模拟电路的优点

PWM 信号是数字信号，处理器到被控系统信号都是数字形式的，可将噪声影响降到最低，噪声只有在强到足以将逻辑 1 改变为逻辑 0 或将逻辑 0 改变为逻辑 1 时，也才能对数字信号产生影响。PWM 对噪声抵抗能力的增强，因此，从模拟信号转向 PWM 可以极大地延长通信距离。

### ■ PWM 的应用领域

- 1) 电机驱动：PWM 被称为“开关驱动装置”，PWM 信号的高低电平可控制电机是否通电，电机通电，就会加速；电机断电，就会减速甚至停止。只要 PWM 信号频率达到一定值，改变 PWM 占空比可实现对电机速度的控制。
- 2) 开关电源：PWM 开关型稳压电路是在控制电路输出频率不变的情况下，通过电压反馈调整其占空比，从而达到稳定输出电压的目的。
- 3) 电池充电：在镍氢电池智能充电器中采用的脉宽 PWM 法，它是把每一脉冲宽度均相等的脉冲列作为 PWM 波形，通过改变脉冲列的周期可以调频，改变脉冲的宽度或占空比可以调压，采用适当控制方法即可使电压与频率协调变化。实现通过调整 PWM 的周期、PWM 的占空比而达到控制充电电流的目的。
- 4) D/A 转换：PWM 高频输出后加 RC 滤波电路，通过改变 PWM 信号的占空比实现输出不同电压值的目的。
- 5) 逆变电路：目前中小功率的逆变电路几乎都采用 PWM 技术，逆变电路是 PWM 控制技术极为重要的应用场合。

## 3. nRF52840 的 PWM

nRF52840 片内集成了 4 个 PWM 外设，每个 PWM 外设具有 4 个通道，每个通道均可在 GPIO 上生成脉冲宽度调制信号。PWM 模块实现了一个向上或向下计数器，带有四个 PWM 通道，用于驱动分配的 GPIO。nRF52840 的 PWM 模块主要特点如下：

- 可编程 PWM 频率。
- 最多四个 PWM 通道，每个通道具有单独的极性和占空比值。
- 支持边沿对齐和中心对齐。
- RAM 中可定义多个占空比序列。
- 通过 EasyDMA 直接从存储器中自主且无干扰地更新占空比值（无 CPU 参与）。
- 每个 PWM 周期可改变极性、周期宽度和占空比。
- 支持双序列，两个序列可以重复或连接到循环中。

### 3.1. 波形计数器

波形计数器负责产生一定占空比的脉冲，该占空比取决于比较值，并且其频率取决于 COUNTERTOP 寄存器的值。

PWM 有一个通用的 15 位计数器，带有四个比较通道，四个比较通道共用 COUNTERTOP，因此，四个通道将共享相同的 PWM 周期（PWM 频率），每个通道的比较值可以单独设置，因此，各个通道拥有单独的占空比和极性（极性由从 RAM 读取的值设置）。

计数器有向上计数和上/下计数两种计数模式，计数模式通过 MODE 寄存器设置。计数器最大值由 COUNTERTOP 寄存器控制，该寄存器值与 PWM\_CLK 所选 PRESCALER 一起将产生给定的 PWM 周期。如果 COUNTERTOP 的值小于设置的比较值将导致不产生 PWM 边沿状态。4 个比较寄存器都是 PWM 内部的，他们只能通过下一节中介绍的解码器进行配置，而不能通过寄存器直接写入数值。COUNTERTOP 寄存器可以随时安全地写入数据。

#### 1. 向上计数模式

向上计数会产生边沿对齐的 PWM 信号，下图给出了向上计数模式下通道 0 在极性位等于 0（PWM 周期内第一个边沿为上升沿）和 1（PWM 周期内第一个边沿为下降沿）时的信号，对于通道 1~通道 3，情况和通道 0 是一样的。

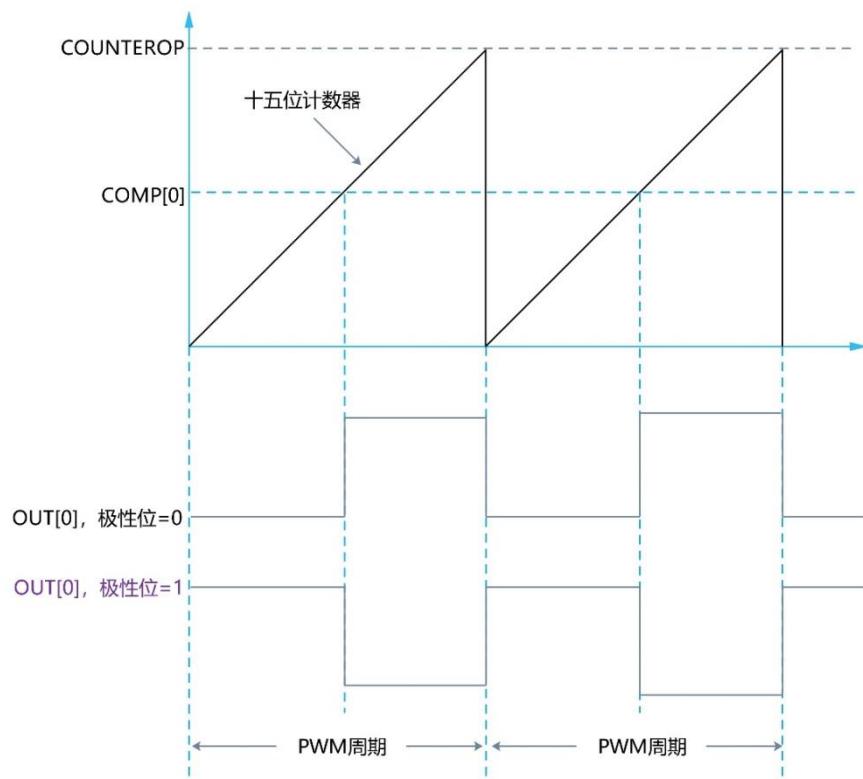


图 23-2: 向上计数

由上图可以看到，计数器从 0 开始计数，当计数值达到 COUNTERTOP 时，计数器自动复位为零，OUT [n] 将反转。

- **PWM 周期的确定:** PWM 时钟频率 PWM\_CLK 确定后，COUNTERTOP 的值确定了 PWM 周期，如 PWM\_CLK 设置为 1MHz，COUNTERTOP 的值设置为 10000，那么 PWM 周

期为  $10000/1M = 10ms$ 。即 PWM 周期计算公式为  $T_{PWM(Up)} = T_{PWM\_CLK} * COUNTERTOP$ 。

- 步进宽度:  $T_{PWM\_CLK}$ 。
- 极性: 确定 PWM 周期内第一个边沿是上升沿还是下降沿。PWM 的计数器是 15 位的, 从 RAM 中装载的数据是 16 位的, 其中的最高位即为极性位。极性位=0: 第一个边沿是上升沿, 这种情况下, 如果比较值为 0, 则 OUT[n]保持低电平; 极性位=1: 第一个边沿是下降沿, 这种情况下, 如果比较值为 0, 则 OUT[n]保持高电平。
- 占空比: 极性位=0 时, 占空比=  $(COUNTERTOP-COMP[n]) / COUNTERTOP$ ; 极性位=1 时, 占空比= $COMP[n]/ COUNTERTOP$ 。

## 2. 上/下计数模式

上/下计数会产生中心对齐的 PWM 信号, 下图给出了上/下计数模式下通道 0 在极性位等于 0 (PWM 周期内第一个边沿为上升沿) 和 1 (PWM 周期内第一个边沿为下降沿) 时的信号, 对于通道 1~通道 3, 情况和通道 0 是一样的。

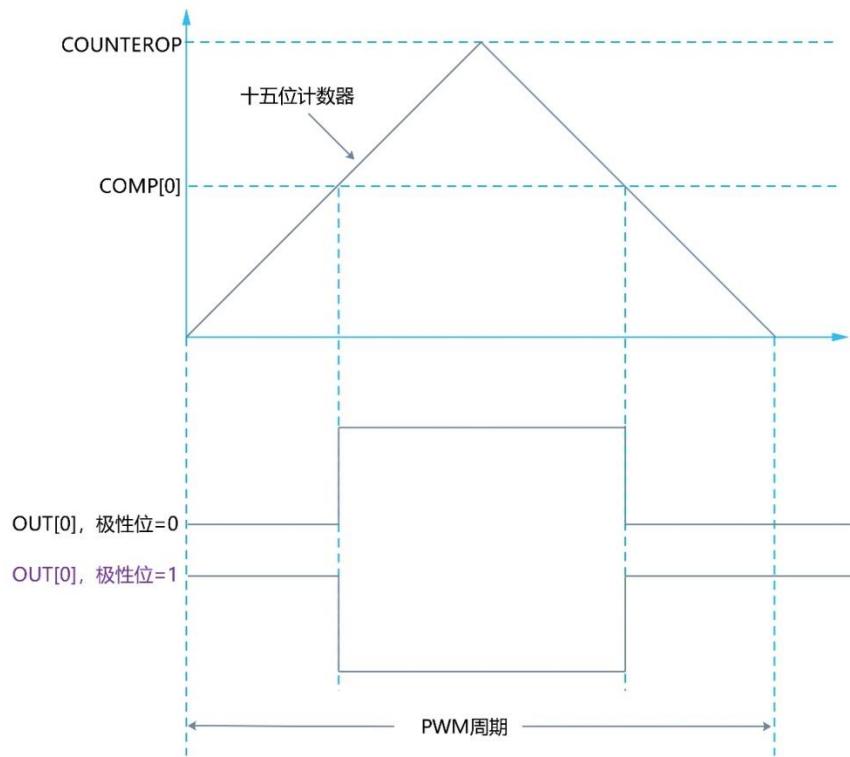


图 23-3: 上/下计数

由上图可以看到, 计数器从 0 开始计数, 当计数值达到 COUNTERTOP 时, 计数器开始递减到零, 并且当第二次达到比较值时反转 OUT [n]。

- PWM 周期的确定: 相对于向上计数模式, PWM 周期增加了一倍, PWM 周期计算公式为  $T_{PWM(Up And Down)} = T_{PWM\_CLK} * COUNTERTOP * 2$ 。
- 步进宽度:  $T_{PWM\_CLK} * 2$ 。
- 极性: 和向上计数模式一样。
- 占空比: 占空比和向上计数模式一样, 极性位=0 时, 占空比= $(COUNTERTOP-COMP[n]) / COUNTERTOP$ ; 极性位=1 时, 占空比= $COMP[n]/ COUNTERTOP$ 。

### 3.2. EasyDMA 解码器

PWM 通道的比较和极性寄存器是程序员不可见的，PWM 参数（通道的比较值和极性）是不能直接写入到该寄存器的，PWM 参数必须存储在 RAM 中，由 EasyDMA 获取，之后由解码器（DECODER）根据设定的装载模式来更新波形计数器的内部比较寄存器。

#### 3.2.1. PWM 序列的组织

PWM 参数被组织成包至少含一个半字（16 位）的序列，其最高有效位[15]表示 OUT [n] 的极性，而位[14:0]是 15 位比较值，如下表所示。

表 23-1: PWM 参数结构

| 位        | Field    | RW  | 复位值 | 描述                                           |
|----------|----------|-----|-----|----------------------------------------------|
| 位 14~为 0 | COMPARE  | 读/写 | 0   | 占空比设置，加载到内部比较寄存器的值。                          |
| 位 15     | POLARITY | 读/写 | 0   | 0: PWM 周期内第一个边沿是上升沿。<br>1: PWM 周期内第一个边沿是下降沿。 |

#### 3.2.2. 装载模式

装载模式给出了如何解释 RAM 中存储的 PWM 序列并将其加载到 PWM 内部比较寄存器中，装载模式由 DECODER 寄存器控制，DECODER 寄存器的 LOAD 字段定义了装载模式，共有如下图所示的 4 种装载模式。

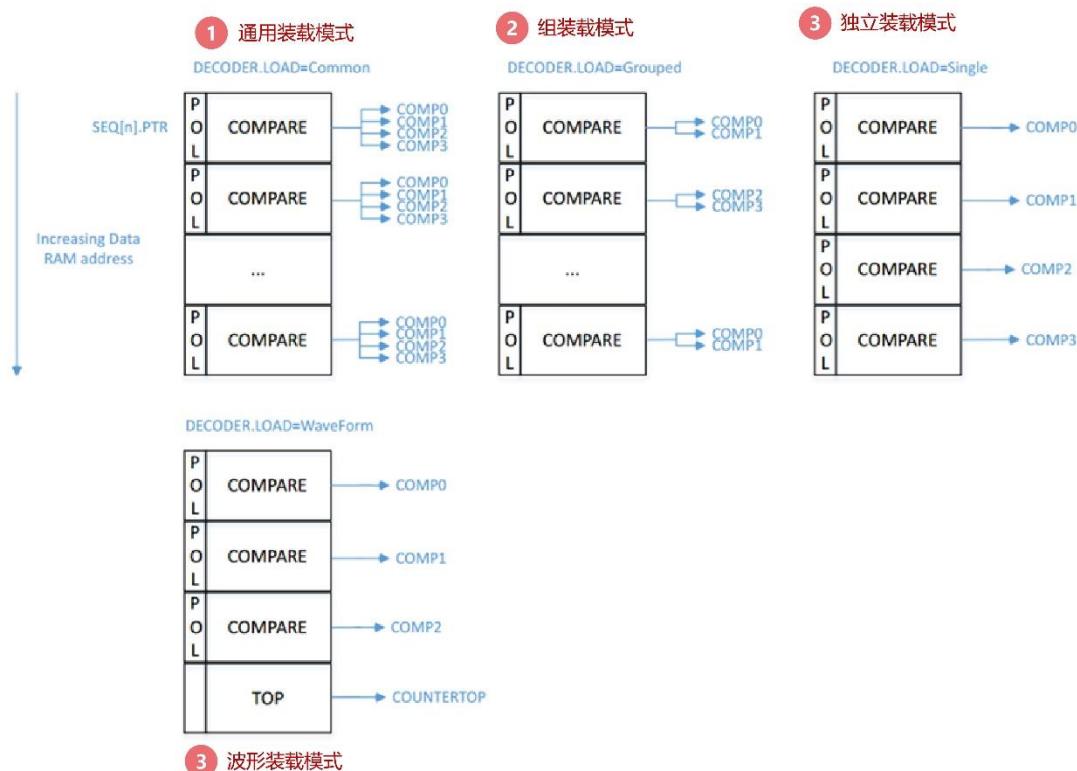


图 23-4: 解码器内存访问模式

四种装载模式执行过程如下。

1. 通用装载模式 (Common): 序列中的每个 PWM 参数会同时装载到 PWM 模块的 4 个通道 (通道[0..3])。
2. 组装载模式 (Grouped): 序列中的第 1 个 PWM 参数装载到 PWM 模块的通道 0 和通道 1, 第 2 个 PWM 参数装载到 PWM 模块的通道 2 和通道 3, 以此类推。
3. 独立装载模式 (Individual): 序列中的第 1 个 PWM 参数装载到 PWM 模块的通道 0, 第 2 个装载到通道 1, 第 3 个装载到通道 2, 第 4 个装载到通道 3, 以此类推。
4. 波形装载模式 (WaveForm): 序列中的第 1 个 PWM 参数装载到 PWM 模块的通道 0, 第 2 个装载到通道 1, 第 3 个装载到通道 2, 第 4 个装载到 COUNTERTOP 寄存器, 以此类推。

这 4 种装载模式中, 前 3 种模式最多可以使用 4 个 PWM 通道, COUNTERTOP 寄存器的值需要单独写。而波形装载模式比较特殊, 他最多只能用 4 个通道中的 3 个 (通道 0~2), 这种模式下从 RAM 中读取的 PWM 参数序列, 前 3 个会装载到通道 0~2 的比较寄存器, 而第 4 个参数会作为 COUNTERTOP 寄存器的值装载到 COUNTERTOP 寄存器。由此可见, 波形装载模式可以改变每个 PWM 的周期, 这种工作模式对于应用中的任意波形生成非常有用, 例如 LED 照明。

设置了 PWM 序列装载模式后, 还需要设置 PWM 序列中周期的推进方式, 这是由 DECODER 寄存器的 MODE 字段定义的。

- DECODER.MODE=RefreshCount: SEQ[n].REFRESH 寄存器决定了 PWM 序列中的值何时装载到内部比较器。如 SEQ [n] .REFRESH = K (n = 0 或 1), 当前 PWM 周期会执行 K 次, 在第(K+1)个 PWM 周期装载下一个值。如 SEQ [n] .REFRESH = 0 (n = 0 或 1), 则每个周期都会更新。
- DECODER. MODE= NextStep: 每触发一次 TASKS\_NEXTSTEP 任务, 装载 PWM 序列中的一个值, 这种方式下, 寄存器 SEQ [n] .REFRESH 和 SEQ [n] .ENDDELAY 被忽略。

### 3.2.3. PWM 序列在 RAM 中的存储

序列必须存储在片内 RAM 中, SEQ[n].PTR 是用于从 RAM 获取 PWM 参数的指针, SEQ[n].PTR 保存的是 PWM 序列在 RAM 中的起始地址。如果 SEQ[n].PTR 没有指向 RAM 区域, EasyDMA 传输可能会导致硬件错误或 RAM 数据被破坏。

SEQ[n].PTR 设置为 PWM 序列在 RAM 中的起始地址后, 必须将 SEQ[n].CNT 寄存器设置为序列中 16 位半字的数量。另外, 注意到装载模式不同, 占用的 RAM 大小也有所差异, 如通用装载模式 4 个通道需要一个半字, 而独立模式每个通道就需要一个半字。

如果在触发 SEQSTART[n]任务时 PWM 发生器未运行, 则任务将从 RAM 加载第一个值, 然后启动 PWM 发生器。一旦 EasyDMA 从 RAM 读取第一个 PWM 参数并且波形计数器已开始执行他, 就会生成 SEQSTARTED[n]事件。

### 3.2.4. 序列播放控制

通过配置相关寄存器, 可以控制序列播放时序列中周期的重复次数、序列后插入延时以及序列的重放次数。

#### 1. 序列中周期的重复次数

寄存器 DECODER. MODE=RefreshCount 时，SEQ[n].REFRESH 寄存器决定了 PWM 序列中每个周期的重复次数。如下图所示，PWM 序列包含了 2 个周期（为了直观，两个周期占空比不一样），当 SEQ [0] .REFRESH = 0 时，先播放 PWM 周期 1，接着播放周期 2。当 SEQ [0] .REFRESH = 1 时，先播放 PWM 周期 1，接着重放周期 1 一次，之后播放周期 2，接着重放周期 2 一次。

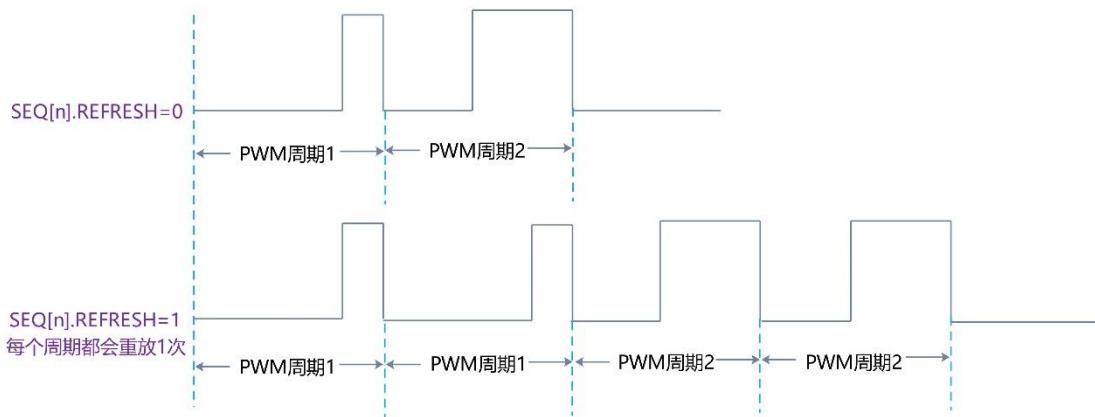


图 23-5: 序列中周期的重复次数

## 2. 序列后插入延时

通过设置寄存器 SEQ[n].ENDDELAY (n=0..1) 的值可以在序列后插入延时，延时单位是最后一个播放的 PWM 周期。如下图所示，PWM 序列包含了 2 个周期，当 SEQ[0].ENDDELAY = 1 时，序列后会接着播放一次序列中的最后一个周期。

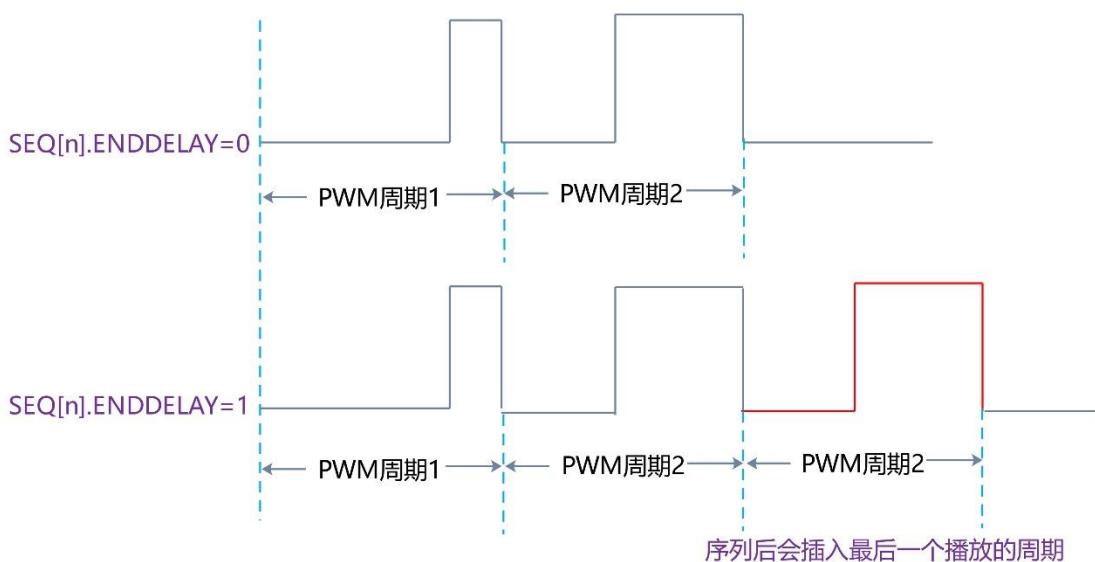


图 23-6: 序列中周期的重复次数

## 3. 序列的回放次数

通过设置 LOOP 寄存器，可以控制序列回放的次数，当 LOOP.CNT = 0 时，序列播放一次，当 LOOP.CNT = 1 时，序列播放一次后再回放一次。注意当 LOOP.CNT = 0 时，PWM 序列播放完成后，会继续以序列中最后一个周期的参数播放。

下图展示了一个简单序列在 LOOP.CNT = 0 时的播放，序列 0 中包含 4 个 PWM 周期，

当加载序列中的最后一个周期并开始执行后,将生成一个 SEQEND [n]事件,由于 LOOP.CNT =0, PWM 会一直继续播放最后一个周期直到停止 PWM。

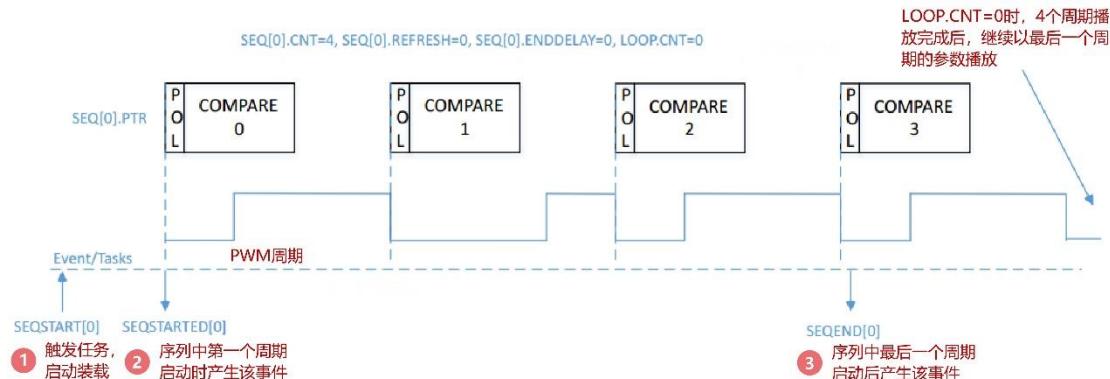


图 23-7: `LOOP.CNT = 0` 时的序列播放

### 3.2.5. 复合序列

PWM 支持双序列, 寄存器 `SEQ[0].PTR` ( $n=0$  或  $1$ ) 指向序列  $0$  或  $1$  在 RAM 中的起始地址, 寄存器 `SEQ[n].CNT` ( $n=0$  或  $1$ ) 保存序列  $0$  或  $1$  中的 PWM 周期数量。

两个序列可以串接组成复合序列, 序列的串接是隐式的, 当配置了两个序列后, 如果以 `SEQSTART [0]` 任务开始, 会首先播放 `SEQ [0]`, 之后播放 `SEQ [1]`。如果以 `SEQSTART [1]` 任务开始, 则会首先播放 `SEQ [1]`, 之后播放 `SEQ [0]`。

复合序列同样可以通过寄存器 `LOOP.CNT` 设置回放次数, 注意复合序列回放一次是串接的两个序列都回放一次。复合序列在回放了寄存器 `LOOP.CNT` 指定的次数后, 会产生 `LOOPS DONE` 事件通知 CPU。

### 3.3. 限制

如果 PWM 周期短于 EasyDMA 从 RAM 检索和更新内部比较寄存器所需的时间, 则重複上一个比较值。这是为了确保即使在很短的脉宽调制周期内也能无故障运行。

### 3.4. 引脚配置

通过配置 `PSEL.OUT [n]` ( $n = 0..3$ ) 寄存器, 每个 PWM 外设的 4 个通道均可映射到任一物理引脚, 如果 `PSEL.OUT [n].CONNECT` 设置为 `Disconnected`, 则关联 PWM 模块信号不会连接到任何物理引脚。

只要 PWM 模块使能且 PWM 发生器激活 (波形计数器启动), 就会使用 `PSEL.OUT [n]` 寄存器及其配置, 这些配置只有在设备处于 `System ON` 模式时才会被保留。

为确保 PWM 模块中的正确行为, 必须在启用 PWM 模块之前以下列方式在 GPIO 外设中配置所使用的引脚。

表 23-2: PWM 引脚配置

| PWM 信号              | PWM 引脚                                      | 方向 | 输出值 | 说明                                   |
|---------------------|---------------------------------------------|----|-----|--------------------------------------|
| <code>OUT[n]</code> | <code>PSEL.OUT [n]</code> ( $n = 0..3$ ) 定义 | 输出 | 0   | 空闲状态由 GPIO 的 <code>OUT</code> 寄存器定义。 |

为了确保正确驱动 PWM 模块使用的引脚，引脚在空闲时的状态应该是预知的，PWM 引脚的空闲状态由 GPIO 模块中的 OUT 寄存器定义，通过设置 GPIO 模块中的 OUT 寄存器即可确定空闲状态下的引脚状态。

注意：不能在同一时刻将多个 PWM 外设的信号映射到同一个物理引脚，如不能将 PWM0 和 PWM1 的通道 0 的信号同时映射到 P0.05，否则可能会导致不可预知的行为。

## 4. PWM 寄存器

nRF52840 片内集成了 4 个 PWM 外设，如下表所示。

表 23-3: nRF52840 的 PWM 外设

| 外设名称 | 基址         |
|------|------------|
| PWM0 | 0x4001C000 |
| PWM1 | 0x40021000 |
| PWM2 | 0x40022000 |
| PWM3 | 0x4002D000 |

表 23-4: SPIM 相关寄存器

| 序号           | 寄存器名                     | 偏移地址  | 功能描述                                                                                              |
|--------------|--------------------------|-------|---------------------------------------------------------------------------------------------------|
| <b>任务寄存器</b> |                          |       |                                                                                                   |
| 1            | TASKS_STOP               | 0x004 | 在当前 PWM 周期结束时停止所有通道上的 PWM 脉冲生成，并停止 PWM 序列回放。                                                      |
| 2            | TASKS_SEQSTART[0]        | 0x008 | 从序列 0 加载所有启用通道上的第一个 PWM 值，并以 SEQ [0] REFRESH 和/或 DECODER.MODE 中定义的速率开始播放该序列。如 PWM 未启动，该任务会启动 PWM。 |
| 3            | TASKS_SEQSTART[1]        | 0x00C | 从序列 1 加载所有启用通道上的第一个 PWM 值，并以 SEQ [1] REFRESH 和/或 DECODER.MODE 中定义的速率开始播放该序列。如 PWM 未启动，该任务会启动 PWM。 |
| 4            | TASKS_NEXTSTEP           | 0x010 | 如果 DECODER.MODE =NextStep，则在所有已启用通道上按当前序列中的一个值逐步执行。如 PWM 未启动，该任务不会启动 PWM。                         |
| <b>事件寄存器</b> |                          |       |                                                                                                   |
| 1            | EVENTS_STOPPED           | 0x104 | PWM 停止事件，响应 STOP 任务，当不再生成 PWM 脉冲时发出。                                                              |
| 2            | EVENTS_SEQSTARTE<br>D[0] | 0x108 | 第一个 PWM 周期从序列 0 开始。                                                                               |
| 3            | EVENTS_SEQSTARTE<br>D[1] | 0x10C | 第一个 PWM 周期从序列 1 开始。                                                                               |

|          |                         |       |                                      |
|----------|-------------------------|-------|--------------------------------------|
| <b>4</b> | EVENTS_SEQEND[0]        | 0x110 | 在每个序列 0 结束时发出，当 RAM 的最后一个值应用于波形计数器时。 |
| <b>5</b> | EVENTS_SEQEND[1]        | 0x114 | 在每个序列 1 结束时发出，当 RAM 的最后一个值应用于波形计数器时。 |
| <b>6</b> | EVENTS_PWMPERIOD<br>END | 0x118 | 在每个 PWM 周期结束时发出。                     |
| <b>7</b> | EVENTS_LOOPSDONE        | 0x11C | 串联序列已经播放了 LOOP.CNT 中定义的次数。           |

**快捷方式寄存器**

|          |        |       |          |
|----------|--------|-------|----------|
| <b>1</b> | SHORTS | 0x200 | 快捷功能寄存器。 |
|----------|--------|-------|----------|

**通用寄存器**

|           |                 |       |                          |
|-----------|-----------------|-------|--------------------------|
| <b>1</b>  | INTEN           | 0x304 | 使能或禁止中断。                 |
| <b>2</b>  | INTENSET        | 0x304 | 使能中断。                    |
| <b>3</b>  | INTENCLR        | 0x308 | 禁止中断。                    |
| <b>4</b>  | ENABLE          | 0x500 | 使能 PWM。                  |
| <b>5</b>  | MODE            | 0x504 | 波形计数器模式。                 |
| <b>6</b>  | COUNTERTOP      | 0x508 | 脉冲发生器计数器计数的值。            |
| <b>7</b>  | PRESCALER       | 0x50C | 预分频系数，配置 PWM 频率 PWM_CLK。 |
| <b>8</b>  | DECODER         | 0x510 | 译码器的配置。                  |
| <b>9</b>  | LOOP            | 0x514 | 循环播放次数。                  |
| <b>10</b> | SEQ[0].PTR      | 0x520 | 序列在 RAM 中的起始地址。          |
| <b>11</b> | SEQ[0].CNT      | 0x524 | 序列中的值（占空比）数。             |
| <b>12</b> | SEQ[0].REFRESH  | 0x528 | 样本加载到比较寄存器之间的附加 PWM 周期数。 |
| <b>13</b> | SEQ[0].ENDDELAY | 0x52C | 序列后插入的延时时间。              |
| <b>14</b> | SEQ[1].PTR      | 0x540 | 序列在 RAM 中的起始地址。          |
| <b>15</b> | SEQ[1].CNT      | 0x544 | 序列中的值（占空比）数。             |
| <b>16</b> | SEQ[1].REFRESH  | 0x548 | 样本加载到比较寄存器之间的附加 PWM 周期数。 |
| <b>17</b> | SEQ[1].ENDDELAY | 0x54C | 序列后插入的延时时间。              |
| <b>18</b> | PSEL.OUT[0]     | 0x560 | PWM 通道 0 输出引脚配置。         |
| <b>19</b> | PSEL.OUT[1]     | 0x564 | PWM 通道 1 输出引脚配置。         |

|    |             |       |                  |
|----|-------------|-------|------------------|
| 20 | PSEL.OUT[2] | 0x568 | PWM 通道 2 输出引脚配置。 |
| 21 | PSEL.OUT[3] | 0x56C | PWM 通道 3 输出引脚配置。 |

### ■ SHORTS: 快捷方式寄存器

用于配置本地事件和任务之间的快捷方式。

表 23-5: SHORTS 寄存器

| 位   | Field               | RW    | 复位值 | 描述                                                               |
|-----|---------------------|-------|-----|------------------------------------------------------------------|
| 位 0 | SEQEND0_STOP        | 读 / 写 | 0   | SEQEND[0]事件和 STOP 任务之间的快捷方式。<br>0: 禁止快捷方式。<br>1: 使能快捷方式。         |
| 位 1 | SEQEND1_STOP        | 读 / 写 | 0   | SEQEND[1]事件和 STOP 任务之间的快捷方式。<br>0: 禁止快捷方式。<br>1: 使能快捷方式。         |
| 位 2 | LOOPSDONE_SEQSTART0 | 读 / 写 | 0   | LOOPSDONE 事件和 SEQSTART[0] 任务之间的快捷方式。<br>0: 禁止快捷方式。<br>1: 使能快捷方式。 |
| 位 3 | LOOPSDONE_SEQSTART1 | 读 / 写 | 0   | LOOPSDONE 事件和 SEQSTART[1] 任务之间的快捷方式。<br>0: 禁止快捷方式。<br>1: 使能快捷方式。 |
| 位 4 | LOOPSDONE_STOP      | 读 / 写 | 0   | LOOPSDONE 事件和 STOP 任务之间的快捷方式。<br>0: 禁止快捷方式。<br>1: 使能快捷方式。        |

### ■ INTEN: 中断使能/禁止寄存器

INTEN 寄存器既可以使能中断也可以禁止中断。位的值写为 1 时使能对应的中断，写入 0 时禁止对应的中断。

表 23-6: INTEN 寄存器

| 位   | Field   | RW  | 复位值 | 描述                               |
|-----|---------|-----|-----|----------------------------------|
| 位 1 | STOPPED | 读/写 | 0   | 使能或禁止 STOPPED 事件中断。<br>写, 0: 禁止。 |

|     |               |       |                           |                        |
|-----|---------------|-------|---------------------------|------------------------|
|     |               |       |                           | 写, 1: 使能。              |
| 位 2 | SEQSTARTED[0] | 读/写 0 | 使能或禁止 SEQSTARTED[0]事件中断。  | 写, 0: 禁止。<br>写, 1: 使能。 |
| 位 3 | SEQSTARTED[1] | 读/写 0 | 使能或禁止 SEQSTARTED[1]事件中断。  | 写, 0: 禁止。<br>写, 1: 使能。 |
| 位 4 | SEQEND[0]     | 读/写 0 | 使能或禁止 SEQEND [0]事件中断。     | 写, 0: 禁止。<br>写, 1: 使能。 |
| 位 5 | SEQEND[1]     | 读/写 0 | 使能或禁止 SEQEND [1]事件中断。     | 写, 0: 禁止。<br>写, 1: 使能。 |
| 位 6 | PWMPPERIODEND | 读/写 0 | 使能或禁止 PWMPPERIODEND 事件中断。 | 写, 0: 禁止。<br>写, 1: 使能。 |
| 位 7 | LOOPSDONE     | 读/写 0 | 使能或禁止 LOOPSDONE 事件中断。     | 写, 0: 禁止。<br>写, 1: 使能。 |

### ■ INTENSET：中断使能寄存器

INTENSET 寄存器用于使能中断。位的值写为 1 时使能对应的中断，写入 0 无效。注意 INTENSET 寄存器只能用来使能中断。

表 23-7: INTENSET 寄存器

| 位   | Field         | RW    | 复位值                                                                  | 描述 |
|-----|---------------|-------|----------------------------------------------------------------------|----|
| 位 1 | STOPPED       | 读/写 0 | 写“1”使能 STOPPED 事件中断，写“0”无效。<br>写, 1: 使能。<br>读, 0: 已禁止。<br>读, 1: 已使能。 |    |
| 位 2 | SEQSTARTED[0] | 读/写 0 | 写“1”使能 SEQSTARTED[0]事件中断，                                            |    |

|     |               |       |                                                                                 |
|-----|---------------|-------|---------------------------------------------------------------------------------|
|     |               |       | 写“0”无效。<br>写, 1: 使能。<br>读, 0: 已禁止。<br>读, 1: 已使能。                                |
| 位 3 | SEQSTARTED[1] | 读/写 0 | 写“1”使能 SEQSTARTED[1]事件中断,<br>写“0”无效。<br>写, 1: 使能。<br>读, 0: 已禁止。<br>读, 1: 已使能。   |
| 位 4 | SEQEND[0]     | 读/写 0 | 写“1”使能 SEQEND [0]事件中断, 写<br>“0”无效。<br>写, 1: 使能。<br>读, 0: 已禁止。<br>读, 1: 已使能。     |
| 位 5 | SEQEND[1]     | 读/写 0 | 写“1”使能 SEQEND [1]事件中断, 写<br>“0”无效。<br>写, 1: 使能。<br>读, 0: 已禁止。<br>读, 1: 已使能。     |
| 位 6 | PWMPPERIODEND | 读/写 0 | 写“1”使能 PWMPPERIODEND 事件中<br>断, 写“0”无效。<br>写, 1: 使能。<br>读, 0: 已禁止。<br>读, 1: 已使能。 |
| 位 7 | LOOPSDONE     | 读/写 0 | 写“1”使能 LOOPSDONE 事件中断,<br>写“0”无效。<br>写, 1: 使能。<br>读, 0: 已禁止。<br>读, 1: 已使能。      |

### ■ INTENCLR: 中断禁止寄存器

INTENCLR 寄存器用于禁止中断。位的值写为 1 时禁止对应的中断，写入 0 无效。注意 INTENCLR 寄存器只能用来禁止中断。

表 23-8: INTENSET 寄存器

| 位   | Field         | RW  | 复位值 | 描述                                                                         |
|-----|---------------|-----|-----|----------------------------------------------------------------------------|
| 位 1 | STOPPED       | 读/写 | 0   | 写“1”禁止 STOPPED 事件中断，写“0”无效。<br>写, 1: 禁止。<br>读, 0: 已禁止。<br>读, 1: 已使能。       |
| 位 2 | SEQSTARTED[0] | 读/写 | 0   | 写“1”禁止 SEQSTARTED[0]事件中断，写“0”无效。<br>写, 1: 禁止。<br>读, 0: 已禁止。<br>读, 1: 已使能。  |
| 位 3 | SEQSTARTED[1] | 读/写 | 0   | 写“1”禁止 SEQSTARTED[1]事件中断，写“0”无效。<br>写, 1: 禁止。<br>读, 0: 已禁止。<br>读, 1: 已使能。  |
| 位 4 | SEQEND[0]     | 读/写 | 0   | 写“1”禁止 SEQEND [0]事件中断，写“0”无效。<br>写, 1: 禁止。<br>读, 0: 已禁止。<br>读, 1: 已使能。     |
| 位 5 | SEQEND[1]     | 读/写 | 0   | 写“1”禁止 SEQEND [1]事件中断，写“0”无效。<br>写, 1: 禁止。<br>读, 0: 已禁止。<br>读, 1: 已使能。     |
| 位 6 | PWMPPERIODEND | 读/写 | 0   | 写“1”禁止 PWMPPERIODEND 事件中断，写“0”无效。<br>写, 1: 禁止。<br>读, 0: 已禁止。<br>读, 1: 已使能。 |
| 位 7 | LOOPSDONE     | 读/写 | 0   | 写“1”禁止 LOOPSDONE 事件中断，写“0”无效。<br>写, 1: 禁止。                                 |

|  |  |                          |
|--|--|--------------------------|
|  |  | 读, 0: 已禁止。<br>读, 1: 已使能。 |
|--|--|--------------------------|

### ■ ENABLE: PWM 使能/禁止寄存器

ENABLE 寄存器用于使能或禁用 PWM 模块。

表 23-9: ENABLE 寄存器

| 位   | Field  | RW  | 复位值 | 描述                                              |
|-----|--------|-----|-----|-------------------------------------------------|
| 位 0 | ENABLE | 读/写 | 0   | 使能或禁止 PWM 模块。<br>0: 禁止 PWM 模块。<br>1: 使能 PWM 模块。 |

### ■ MODE: 运行模式选择寄存器

MODE 寄存器用于选择 PWM 的运行模式。

表 23-10: MODE 寄存器

| 位   | Field  | RW  | 复位值 | 描述                                                 |
|-----|--------|-----|-----|----------------------------------------------------|
| 位 0 | UPDOWN | 读/写 | 0   | 选择 PWM 的运行模式。<br>0: 向上计数, 边沿对齐。<br>1: 上/下计数, 中心对齐。 |

### ■ COUNTERTOP: 计数器最大计数值设置寄存器

COUNTERTOP 寄存器用于脉冲发生器计数器的最大计数值。

表 23-11: COUNTERTOP 寄存器

| 位        | Field  | RW  | 复位值        | 描述                                                                             |
|----------|--------|-----|------------|--------------------------------------------------------------------------------|
| 位 0~位 15 | UPDOWN | 读/写 | 0x000003FF | [3..32767]: 脉冲发生器计数器计数的最大值。当 DECODER.MODE = WaveForm 并且仅使用来自 RAM 的值时, 将忽略该寄存器。 |

### ■ PRESCALER: 预分频系数设置寄存器

PRESCALER 寄存器用于设置预分频系数, 从而设置 PWM 频率 PWM\_CLK。

表 23-12: COUNTERTOP 寄存器

| 位       | Field    | RW  | 复位值 | 描述                                                                            |
|---------|----------|-----|-----|-------------------------------------------------------------------------------|
| 位 2~位 0 | PRESALER | 读/写 | 0   | 0: 1 分频 (16 MHz)。<br>1: 2 分频 (8 MHz)。<br>2: 4 分频 (4 MHz)。<br>3: 8 分频 (2 MHz)。 |

|  |  |                      |
|--|--|----------------------|
|  |  | 4: 16 分频 (1 MHz)。    |
|  |  | 5: 32 分频 (500 KHz)。  |
|  |  | 6: 64 分频 (250 KHz)。  |
|  |  | 7: 128 分频 (125 KHz)。 |

### ■ DECODER: 解码器配置寄存器

DECODER 寄存器用于配置 PWM 装载模式和 PWM 周期推进方式。

表 23-13: DECODER 寄存器

| 位       | Field | RW  | 复位值 | 描述                                                                         |
|---------|-------|-----|-----|----------------------------------------------------------------------------|
| 位 1~位 0 | LOAD  | 读/写 | 0   | 从 RAM 读取的序列如何传送到比较寄存器。<br>0: 通用模式。<br>1: 组模式。<br>2: 独立模式。<br>3: 波形模式。      |
| 位 8     | MODE  | 读/写 | 0   | 0: SEQ [n] .REFRESH 用于决定何时装载新的值到比较寄存器。<br>1: 每触发一次 TASKS_NEXTSTEP 任务，装载新值。 |

### ■ LOOP: 循环回放次数配置寄存器

LOOP 寄存器用于配置循环回放次数。

表 23-14: LOOP 寄存器

| 位        | Field | RW  | 复位值 | 描述                               |
|----------|-------|-----|-----|----------------------------------|
| 位 0~位 15 | CNT   | 读/写 | 0   | 循环的回放次数。<br>0 表示循环禁用 (在序列结束时停止)。 |

### ■ SEQ[n].PTR (n=0..1): 序列起始地址配置寄存器

该寄存器用于设置序列在 RAM 中的起始地址。

表 23-15: SEQ[n].PTR (n=0..1) 寄存器

| 位        | Field | RW  | 复位值 | 描述              |
|----------|-------|-----|-----|-----------------|
| 位 0~位 31 | PTR   | 读/写 | 0   | 序列在 RAM 中的起始地址。 |

### ■ SEQ[n].CNT (n=0..1): 序列数值数量配置寄存器

该寄存器用于设置序列中的数值（占空比）数量。

表 23-16: SEQ[n].CNT (n=0..1)寄存器

| 位         | Field | RW  | 复位值 | 描述                                          |
|-----------|-------|-----|-----|---------------------------------------------|
| 位 0~ 位 31 | PTR   | 读/写 | 0   | 序列中的数值（占空比）数量。<br>0 表示序列被禁用，并且不能启动，因为序列是空的。 |

### ■ SEQ[n].REFRESH (n=0..1): 附加周期配置寄存器

该寄存器用于设置 PWM 附加周期数量。

表 23-17: SEQ[n].REFRESH (n=0..1)寄存器

| 位         | Field | RW  | 复位值 | 描述                                                                       |
|-----------|-------|-----|-----|--------------------------------------------------------------------------|
| 位 0~ 位 23 | CNT   | 读/写 | 0   | 样本加载到比较寄存器之间的附加 PWM 周期数。每次加载 REFRESH.CNT+1 个 PWM 周期。<br>0 表示更新每个 PWM 周期。 |

### ■ SEQ[n].ENDDELAY (n=0..1): 延时时间配置寄存器

该寄存器用于设置 PWM 序列后插入的延时时间。

表 23-18: SEQ[n].ENDDELAY (n=0..1)寄存器

| 位         | Field | RW  | 复位值 | 描述                            |
|-----------|-------|-----|-----|-------------------------------|
| 位 0~ 位 23 | PTR   | 读/写 | 0   | PWM 序列后插入的延时时间（时间单位为 PWM 周期）。 |

### ■ PSEL.OUT[n] (n=0..3): PWM 通道输出引脚配置寄存器

该寄存器用于配置 PWM 通道的输出引脚。

表 23-19: PSEL.OUT[n] (n=0..3)寄存器

| 位       | Field   | RW  | 复位值   | 描述                                   |
|---------|---------|-----|-------|--------------------------------------|
| 位 4~位 0 | PIN     | 读/写 | 11111 | 引脚编号。                                |
| 位 5     | PORT    | 读/写 | 1     | 端口号。                                 |
| 位 31    | CONNECT | 读/写 | 1     | 1: PWM 通道和引脚连接断开。<br>0: PWM 通道和引脚连接。 |

## 5. 软件设计

### 5.1. 库函数应用

PWM 库的应用步骤如下图所示，包含添加文件在内共 6 个步骤。



图 23-8: PWM 应用步骤

#### 5.1.1. 添加文件和配置工程

##### 1. 需要添加到工程的文件

表 23-20: PWM 需要添加到工程的文件

| 文件名        | SDK 中的目录                    | 描述        |
|------------|-----------------------------|-----------|
| nrfx_pwm.c | ..\modules\nrfx\drivers\src | PWM 驱动文件。 |

##### 2. 需要引用的头文件

使用 PWM 库函数的文件中需要引用 PWM 头文件，代码如下。

```
#include " nrfx_pwm.h"
```

##### 3. 需要添加的头文件包含路劲

表 23-21: 头文件包含路径

| 序号 | 路径                                 |
|----|------------------------------------|
| 1  | ..\..\modules\nrfx\drivers\include |

#### 5.1.2. 定义 PWM 驱动程序实例

PWM 驱动程序实例使用 nrfx\_pwm\_t 结构体定义，该结构体描述了具体的 PWM 外设，当我们定义了 nrfx\_pwm\_t 类型的变量并对其赋值后，该变量就对应了一个具体的硬件 PWM 外设。

定义 PWM 驱动程序实例代码如下，初始化宏 PWM\_INSTANCE 的输入参数对应 SPI 外设的编号。

**代码清单：定义 PWM 驱动程序实例**

```

1. //PWM 驱动程序实例 ID, ID 和外设编号对应, 0:PWM0 1:PWM1 2:PWM2 3:PWM3
2. #define PWM_INSTANCE 0
3.
4. //定义名称为 m_pwm0 的 PWM 驱动程序实例, 参数为 0 表示该实例对应的 PWM 外设为 PWM0
5. static nrfx_pwm_t m_pwm0 = NRFX_PWM_INSTANCE(PWM_INSTANCE);

```

### 5.1.3. sdk\_config.h 中启用 PWM 实例

打开“sdk\_config.h”文件，切换到“Text Editor”，将 PWM 配置的代码拷贝到该文件中（可以从本章的例子中的“sdk\_config.h”文件中拷贝，(492~668)行）。之后，切换到“Configuration Wizard”，勾选需要使用的 PWM 外设，这和下一节中的“定义 PWM 驱动程序实例”是对应的，如我们需要使用 PWM0 外设，那么就需要实例化 PWM0 的驱动程序实例，并且在“sdk\_config.h”文件中勾选“NRFX\_PWM0\_ENABLED – Enable PWM0 Instance”，如下图所示。

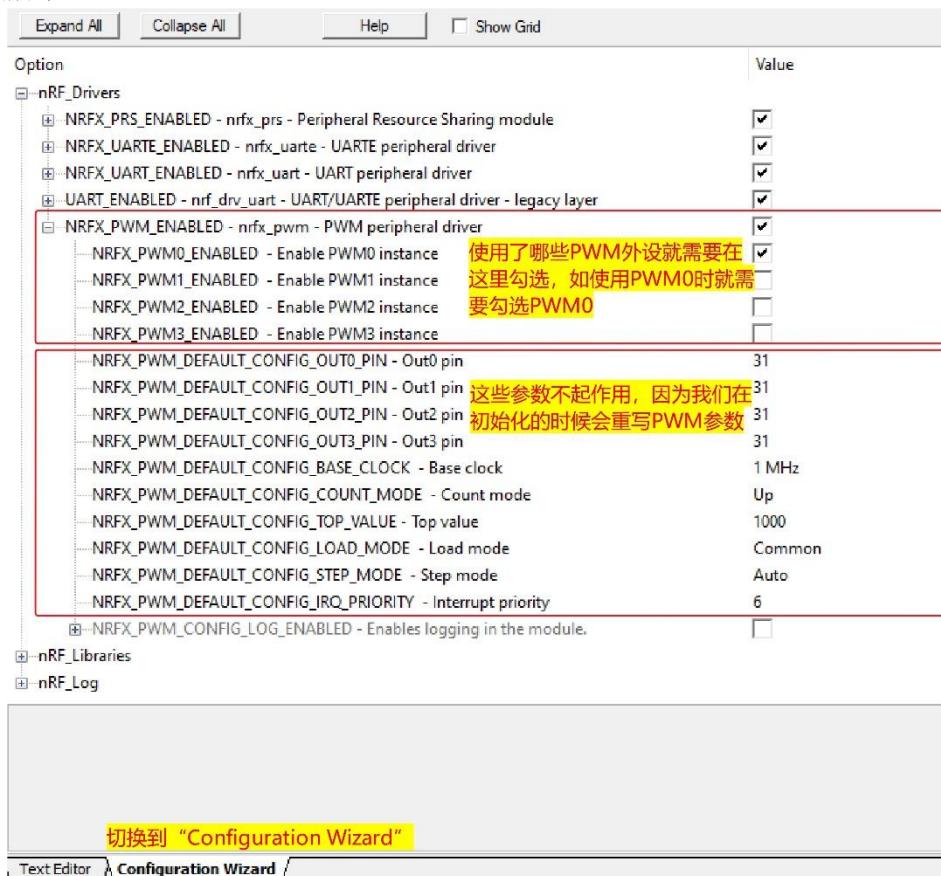


图 23-9：勾选 PWM 实例

### 5.1.4. 定义 PWM 序列

PWM 有 4 种装载模式，PWM 库为每种装载模式声明了对应的结构体用于定义 PWM 序列。由于 PWM 序列是通过 EasyDMA 从 RAM 装载到 PWM 外设的寄存器，因此，PWM 序列必须位于 RAM 中，即 PWM 序列必须定义为 static 类型。

#### 1. 通用装载模式 (Common)

通用装载模式使用“nrf\_pwm\_values\_common\_t”定义序列，由于通用装载模式下序列

中的每个 PWM 周期会同时装载到 PWM 模块的 4 个通道（通道[0..3]），而 PWM 周期参数是 16 位的（1 个极性位+15 个比较值），因此，通用装载模式下的 PWM 序列是 uint16\_t 类型，为了直观，PWM 库中使用“nrf\_pwm\_values\_common\_t”对 uint16\_t 重命名，代码清单如下。

#### 代码清单：通用装载模式 PWM 序列声明

```
1. //用于定义通用装载模式 (NRF_PWM_LOAD_COMMON) 用的 PWM 序列
2. typedef uint16_t nrf_pwm_values_common_t;
```

通用装载模式的 PWM 序列中可以包含一个或多个 PWM 周期参数，当序列中包含多个周期参数时，序列定义为数组形式即可。

#### ■ 示例 1：序列中只有一个 PWM 周期

#### 代码清单：通用装载模式 PWM 序列定义-序列中只包含 1 个 PWM 周期参数

```
1. //定义 PWM 序列 (通用装载模式)，序列只包含一个 PWM 周期
2. static nrf_pwm_values_common_t seq0_values;
```

#### ■ 示例 2：序列中包含 10 个 PWM 周期

#### 代码清单：通用装载模式 PWM 序列定义-序列中包含 10 个 PWM 周期参数

```
1. //定义 PWM 序列 (通用装载模式)，序列只包含一个 PWM 周期
2. static nrf_pwm_values_common_t seq0_values[10];
```

### 2. 组装载模式 (Grouped)

组装载模式使用“nrf\_pwm\_values\_grouped\_t”定义序列，序列中的每个变量包含 2 个周期参数，代码清单如下，“group\_0”通道 0 和 1 的 PWM 周期参数，“group\_1”通道 2 和 3 的 PWM 周期参数。

#### 代码清单：组装载模式 PWM 序列声明

```
1. //用于定义组装载模式 (NRF_PWM_LOAD_GROUPED) 用的 PWM 序列
2. typedef struct {
3. uint16_t group_0; //通道 0 和 1 的 PWM 周期参数
4. uint16_t group_1; //通道 2 和 3 的 PWM 周期参数
5. } nrf_pwm_values_grouped_t;
```

组装载模式的 PWM 序列中可以包含一个或多个描述 PWM 周期参数的变量，当序列中包含多个变量时，序列定义为数组形式即可。

#### ■ 示例 1：序列中只有一个描述 PWM 周期的变量

#### 代码清单：组装载模式 PWM 序列定义-序列中只包含 1 个描述 PWM 周期的变量

```
1. //定义 PWM 序列 (通用装载模式)，序列只包含一个 PWM 周期
2. static nrf_pwm_values_grouped_t seq0_values;
```

**■ 示例 2:** 序列中包含 10 个描述 PWM 周期的变量**代码清单：组装载模式 PWM 序列定义-序列中包含 10 个描述 PWM 周期参数的变量**

```
1. //定义 PWM 序列（通用装载模式），序列只包含一个 PWM 周期
2. static nrf_pwm_values_grouped_t seq0_values[10];
```

**3. 独立装载模式（Individual）**

独立装载模式使用“nrf\_pwm\_values\_individual\_t”定义序列，序列中的每个变量包含 4 个周期参数，分别用于通道 0、1、2 和 3，代码清单如下。

**代码清单：独立载模式 PWM 序列声明**

```
1. //用于定义独立装载模式（NRF_PWM_LOAD_INDIVIDUAL）用的 PWM 序列
2. typedef struct
3. {
4. uint16_t channel_0; //通道 0 的 PWM 周期参数
5. uint16_t channel_1; //通道 1 的 PWM 周期参数
6. uint16_t channel_2; //通道 2 的 PWM 周期参数
7. uint16_t channel_3; //通道 3 的 PWM 周期参数
8. } nrf_pwm_values_individual_t;
```

独立装载模式的 PWM 序列中也可以包含一个或多个描述 PWM 周期参数的变量，当序列中包含多个变量时，序列定义为数组形式即可。

**■ 示例 1:** 序列中只有一个描述 PWM 周期的变量**代码清单：组装载模式 PWM 序列定义-序列中只包含 1 个描述 PWM 周期的变量**

```
1. //定义 PWM 序列（通用装载模式），序列只包含一个 PWM 周期
2. static nrf_pwm_values_individual_t seq0_values;
```

**■ 示例 2:** 序列中包含 10 个描述 PWM 周期的变量**代码清单：组装载模式 PWM 序列定义-序列中包含 10 个描述 PWM 周期参数的变量**

```
1. //定义 PWM 序列（通用装载模式），序列只包含一个 PWM 周期
2. static nrf_pwm_values_individual_t seq0_values[10];
```

**4. 波形装载模式（WaveForm）**

波形装载模式使用“nrf\_pwm\_values\_wave\_form\_t”定义序列，序列中的每个变量包含 3 个周期参数（分别用于通道 0、1、2）和一个写到 COUNTERTOP 寄存器的数值，代码清单如下。这里也可以看出：波形装载模式最多只能用 4 个通道中的 3 个（通道 0~2），序列中的前 3 个会装载到通道 0~2 的比较寄存器，而第 4 个参数会作为 COUNTERTOP 寄存器的值装载到 COUNTERTOP 寄存器。

**代码清单：独立装载模式 PWM 序列声明**

```

1. //用于定义波形装载模式（NRF_PWM_LOAD_WAVE_FORM）用的 PWM 序列
2. typedef struct {
3. uint16_t channel_0; //通道 0 的 PWM 周期参数
4. uint16_t channel_1; //通道 1 的 PWM 周期参数
5. uint16_t channel_2; //通道 2 的 PWM 周期参数
6. uint16_t counter_top; //写到 COUNTERTOP 寄存器的数值
7. } nrf_pwm_values_wave_form_t;

```

波形装载模式的 PWM 序列中同样也可以包含一个或多个描述 PWM 周期参数的变量，当序列中包含多个变量时，序列定义为数组形式即可。

■ **示例 1：**序列中只有一个描述 PWM 周期的变量

**代码清单：波形装载模式 PWM 序列定义-序列中只包含 1 个描述 PWM 周期的变量**

```

1. //定义 PWM 序列（通用装载模式），序列只包含一个 PWM 周期
2. static nrf_pwm_values_wave_form_t seq0_values;

```

■ **示例 2：**序列中包含 10 个描述 PWM 周期的变量

**代码清单：波形装载模式 PWM 序列定义-序列中包含 10 个描述 PWM 周期参数的变量**

```

1. //定义 PWM 序列（通用装载模式），序列只包含一个 PWM 周期
2. static nrf_pwm_values_wave_form_t seq0_values[10];

```

### 5.1.5. 初始化 PWM

PWM 初始化的库函数是 nrfx\_pwm\_init()，该函数函数原型如下表所示。

表 23-22: nrfx\_pwm\_init()函数

|      |                                                                                                                                                                   |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 函数原型 | nrfx_err_t nrfx_pwm_init(<br>nrfx_pwm_t const *const p_instance,<br>nrfx_pwm_config_t const * p_config,<br>nrfx_pwm_handler_t      handler<br>)                   |
| 函数功能 | 初始化 PWM。                                                                                                                                                          |
| 参数   | [in] <b>p_instance</b> : 指向 PWM 驱动程序实例结构体。<br>[in] <b>p_config</b> : 指向 PWM 初始化配置结构体。<br>[in] <b>handler</b> : 指向用户提供的事件句柄，如果为 <b>NULL</b> ，则不会执行事件通知，并禁用 PWM 中断。 |
| 返回值  | <b>NRF_SUCCESS</b> : 初始化成功。<br><b>NRF_ERROR_INVALID_STATE</b> : 驱动程序实例已经初始化。                                                                                      |

调用 nrfx\_pwm\_init()函数初始化 PWM 时，除了 PWM 驱动程序实例外，还需要提供 PWM 初始化配置结构体和事件句柄（PWM 事件处理函数）作为函数的参数。

## 1. PWM 初始化配置结构体

初始化结构体包含了 SPIM 初始化时所需要配置的参数，其声明如下：

### 代码清单：PWM 初始化配置结构体声明

```

1. typedef struct
2. {
3. //PWM 通道映射的物理引脚,如果不使用某个通道,将其设置为 NRF_PWM_PIN_NOT_USED 即可
4. uint8_t output_pins[NRF_PWM_CHANNEL_COUNT];
5. uint8_t irq_priority; //中断优先级,一般设置为 7
6. nrf_pwm_clk_t base_clock; //PWM 时钟频率
7. nrf_pwm_mode_t count_mode; //计数模式: 可配置为向上计数或者上/下计数
8. uint16_t top_value; //写入到 COUNTERTOP 寄存器的值, 计数器最大值。
9. nrf_pwm_dec_load_t load_mode; //装载模式
10. nrf_pwm_dec_step_t step_mode; //序列中的周期推进方式
11. } nrfx_pwm_config_t;

```

PWM 初始化配置结构体中需要重点理解的几个参数如下：

#### 1) PWM 通道映射的物理引脚

`output_pins` 数组是一个 `uint8_t` 类型的数组，每个通道映射的引脚用 8 位表示，如下表所示。

表 23-23: PWM 引脚映射

| 位 7                  | 位 5~位 0 |
|----------------------|---------|
| =0: 设置空闲状态时该引脚输出低电平。 |         |
| =1: 设置空闲状态时该引脚输出高电平。 | 引脚号     |

#### 2) `base_clock`、`count_mode`、`top_value`

`base_clock`、`count_mode`、`top_value` 和 PWM 序列中的周期参数一起决定了 PWM 的周期和占空比。

如 `base_clock = 1MHz`, `count_mode = 向上计数`, `top_value = 10000`, PWM 序列中的一个周期参数为 1000 (极性位 = 0, PWM 周期内第一个边沿是上升沿), 可计算出 PWM 周期和占空比如下：

- PWM 周期 =  $\text{top\_value}/\text{base\_clock} = 10\text{ms}$ 。
- 占空比 =  $(10000-1000)/10000 = 90\%$ 。

#### 3) 装载模式

装载模式可设置为 4 种模式中的一种，对应的定义如下。

### 代码清单：4 种装载模式

```

1. typedef enum
2. {
3. NRF_PWM_LOAD_COMMON = PWM_DECODER_LOAD_Common, //通用装载模式
4. NRF_PWM_LOAD_GROUPED = PWM_DECODER_LOAD_Grouped, //组装载模式

```

```

5. NRF_PWM_LOAD_INDIVIDUAL = PWM_DECODER_LOAD_Individual, //独立装载模式
6. NRF_PWM_LOAD_WAVE_FORM = PWM_DECODER_LOAD_WaveForm //波形装载模式
7. } nrf_pwm_dec_load_t;

```

#### 4) 序列中的周期推进方式

这个参数设置的是 DECODER 寄存器的 MODE 字段（详见 3.3.2.装载模式），程序中的定义如下。

#### 代码清单：周期推进方式

```

1. typedef enum
2. {
3. // MODE 字段设置为 0: 自动装载序列中的下一个值 (SEQ[n].REFRESH 寄存器值为 0) 或重复当
4. // 前 PWM 周期 “SEQ[n].REFRESH” 次后装载序列中的下一个值。
5. NRF_PWM_STEP_AUTO = PWM_DECODER_MODE_RefreshCount,
6. //MODE 字段设置为 1: 每触发一次 TASKS_NEXTSTEP 任务，装载 PWM 序列中的一个值
7. NRF_PWM_STEP_TRIGGERED = PWM_DECODER_MODE_NextStep
8. } nrf_pwm_dec_step_t;

```

#### 2. 事件句柄

应用程序是否提供事件句柄决定了 PWM 是否向应用程序提交事件和产生中断。

- 应用程序不提供事件句柄 (handler 设置为 NULL)

PWM 播放由硬件自动执行，PWM 中断关闭，即 PWM 不会产生中断，CPU 在 PWM 播放的过程中仍可保持在睡眠模式（浅睡）。

- 应用程序提供的事件句柄

PWM 驱动程序向应用程序提交的事件有以下 4 种，如果我们需要关注某个事件，在事件处理函数里面处理该事件即可。

#### 代码清单：PWM 事件

```

1. typedef enum
2. {
3. NRFX_PWM_EVT_FINISHED, //序列播放结束
4. NRFX_PWM_EVT_END_SEQ0, //已播放到序列 0 的末端，此时可安全地修改序列 0 的数据
5. NRFX_PWM_EVT_END_SEQ1, //已播放到序列 1 的末端，此时可安全地修改序列 1 的数据
6. NRFX_PWM_EVT_STOPPED, //PWM 外设已经停止
7. } nrfx_pwm_evt_type_t;

```

如果应用程序需要获取 PWM 驱动程序的事件，就需要提供事件处理函数，并在初始化时将事件句柄作为 nrft\_pwm\_init() 函数的参数传递给函数，由初始化函数完成注册。PWM 事件处理函数的编写格式如下，我们编写 PWM 事件处理函数时需要遵循该格式。

#### 代码清单：PWM 事件处理函数编写格式

```
1. static void pwm_handler(nrft_pwm_evt_type_t event_type)
```

```

2. {
3. if (event_type == NRF_DRV_PWM_EVT_FINISHED) //序列播放结束事件
4. {
5. //添加功能代码
6. }
7. //其他事件
8. }

```

### 5.1.6. 启动 PWM 播放

PWM 库提供了 2 个库函数用于播放 PWM 序列，nrfx\_pwm\_simple\_playback()函数用于播放单个序列，nrfx\_pwm\_complex\_playback()函数用于播放复合序列（双序列），他们的函数原型如下。

表 23-24: nrfx\_pwm\_simple\_playback()函数

|      |                                                                                                                                                                                                                                |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 函数原型 | <pre>uint32_t nrfx_pwm_simple_playback (     nrfx_pwm_t const *const p_instance,     nrf_pwm_sequence_t const * p_sequence,     uint16_t           playback_count,     uint32_t           flags )</pre>                        |
| 函数功能 | 该函数用于启动单个 PWM 序列的播放。                                                                                                                                                                                                           |
| 参数   | <p>[in] <code>p_instance</code>: 指向 PWM 驱动程序实例结构体。</p> <p>[in] <code>p_sequence</code>: 指向 PWM 播放序列。</p> <p>[in] <code>playback_count</code>: 序列回放次数（不能设置为 0）。</p> <p>[in] <code>flags</code>: 附加选项，传递任何播放标记组合，或 0 表示默认设置。</p> |
| 返回值  | 如果使用 <code>NRFX_PWM_FLAG_START_VIA_TASK</code> 标志，返回触发开始播放的任务的地址，否则返回 0。                                                                                                                                                       |

表 23-25: nrfx\_pwm\_complex\_playback()函数

|      |                                                                                                                                                                                                                                                         |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 函数原型 | <pre>uint32_t nrfx_pwm_complex_playback (     nrfx_pwm_t const *const p_instance,     nrf_pwm_sequence_t const * p_sequence_0,     nrf_pwm_sequence_t const * p_sequence_1,     uint16_t           playback_count,     uint32_t           flags )</pre> |
| 函数功能 | 该函数用于启动双 PWM 序列的播放。                                                                                                                                                                                                                                     |
| 参数   | [in] <code>p_instance</code> : 指向 PWM 驱动程序实例结构体。                                                                                                                                                                                                        |

|     |                                                                                                                                                                                                                                                     |
|-----|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|     | <p>[in] <code>p_sequence_0</code>: 指向要播放的第 1 个 PWM 播放序列。</p> <p>[in] <code>p_sequence_1</code>: 指向要播放的第 2 个 PWM 播放序列。</p> <p>[in] <code>playback_count</code>: 序列回放次数 (不能设置为 0)。</p> <p>[in] <code>flags</code>: 附加选项, 传递 PWM 播放标记组合, 0 表示默认设置。</p> |
| 返回值 | 如果使用 <code>NRFX_PWM_FLAG_START_VIA_TASK</code> 标志, 返回触发开始播放的任务的地址, 否则返回 0。                                                                                                                                                                          |

在调用播放函数时, 需要先定义播放序列, 所谓的播放序列是包含了 PWM 序列的起始地址、大小和序列播放控制描述的结构体, 该结构体声明如下。

#### 代码清单：播放序列声明

```

1. typedef struct
2. {
3. nrf_pwm_values_t values; //指向 PWM 序列
4. uint16_t length; //PWM 序列中包含的周期个数
5. //序列中周期重复次数, 如周期推进方式为: NRF_PWM_STEP_TRIGGERED, 该参数被忽略
6. uint32_t repeats;
7. //序列后插入的延时(周期数), 如周期推进方式为:NRF_PWM_STEP_TRIGGERED, 该参数被忽略
8. uint32_t end_delay;
9. } nrf_pwm_sequence_t;
```

播放函数的参数 “`flags`” 用于提供播放的附件选项, 如 “`flags`” 设置为 “`NRFX_PWM_FLAG_STOP`”, 序列播放完成后, PWM 外设停止, “`flags`” 可设置的值如下:

#### 代码清单：“`flags`”定义

```

1. typedef enum
2. {
3. //播放完成, 停止 PWM 外设。
4. //注意: 当从 RAM 加载最后一个序列的最后一个值时触发 STOP 任务, 并且外设在当前 PWM 周期
5. //结束时停止。对于配置了周期重复的序列, 这可能导致最后一个值的重复次数小于设置的值
6. NRFX_PWM_FLAG_STOP = 0x01,
7. //播放完成后, 从头开始播放。
8. //如果与 NRFX_PWM_FLAG_STOP 一起使用, 则忽略该标志。
9. //注意: 通过 PWM 外设中配置的快捷方式完成播放重启。
10. NRFX_PWM_FLAG_LOOP = 0x02,
11. //加载序列 0 的最后一个值时调用事件处理函数
12. NRFX_PWM_FLAG_SIGNAL_END_SEQ0 = 0x04,
13. //加载序列 1 的最后一个值时调用事件处理函数
14. NRFX_PWM_FLAG_SIGNAL_END_SEQ1 = 0x08,
15. //禁止播放完成事件 (默认启用)
16. NRFX_PWM_FLAG_NO_EVT_FINISHED = 0x10,
```

```

17. //播放函数不直接启动播放，调用播放函数后返回任务地址，由其他程序触发该任务播放。
18. NRFX_PWM_FLAG_START_VIA_TASK = 0x80,
19. } nrfx_pwm_flag_t;

```

## 5.2. 通用装载模式实现呼吸灯实验

本实验在“实验 10-1：串口数据收发”的基础上修改。本例中驱动指示灯 D1、D2 和 D3 实现呼吸灯，程序运行后可以连续地观察到指示灯光强度的增加和减少，本例的功能需求如下表所示。

表 23-26：功能需求表

| 使用的 PWM 外设 | PWM0                                                                   |
|------------|------------------------------------------------------------------------|
| 引脚配置       | 通道 0 映射到 P0.13 驱动指示灯 D1，空闲状态输出高电平。                                     |
|            | 通道 0 映射到 P0.14 驱动指示灯 D2，空闲状态输出高电平。                                     |
|            | 通道 0 映射到 P0.15 驱动指示灯 D3，空闲状态输出高电平。                                     |
|            | 未使用，设置为 NRFX_PWM_PIN_NOT_USED。                                         |
| PWM 周期     | PWM 时钟频率：1MHz，最大计数值：10000，周期 = 10ms。                                   |
| PWM 序列     | 单序列（序列 0），序列中包含 200 个 PWM 周期，周期步进值 100，占空比按照步进值逐步增加，达到 100% 后再逐步减小到 0。 |
| 序列播放控制     | 周期不重复、不插入延时、序列不回放。                                                     |
| PWM 事件     | 应用程序不关注 PWM 事件，即 PWM 初始化时不注册事件处理函数。                                    |

❖ 注：本节对应的实验源码是：“实验 23-1：PWM 呼吸灯-通用装载模式”。

### 5.2.1. 代码编写

按照 PWM 的应用步骤，编写代码。

#### 1. 定义 PWM 实例和配置工程

本例中使用 PWM0，因此 PWM 驱动程序的索引应为 0，程序清单如下。

#### 代码清单：定义 PWM 驱动程序实例，使用 PWM0

```

1. //PWM 驱动程序实例 ID, ID 和外设编号对应, 0:PWM0 1:PWM1 2:PWM2 3:PWM3
2. #define PWM_INSTANCE 0
3.
4. //定义名称为 m_pwm0 的 PWM 驱动程序实例，参数为 0 表示该实例对应的 PWM 外设为 PWM0
5. static nrfx_pwm_t m_pwm0 = NRFX_PWM_INSTANCE(PWM_INSTANCE);

```

之后，在“sdk\_config.h”文件中勾选定义的 PWM 实例，本例使用的是 PWM0，因此在“sdk\_config.h”文件中勾选“NRFX\_PWM0\_ENABLED – Enable PWM0 Instance”。

#### 2. 定义 PWM 序列

按照功能需求表定义 PWM 序列，并对序列赋值，代码清单如下。

#### 代码清单：定义 PWM 序列和初始化序列中的值

```

1. //定义 PWM 序列（通用装载模式），该序列必须位于 RAM 中，因此要定义为 static 类型的
2. uint16_t step = m_top / m_step;
3. static nrf_pwm_values_common_t seq0_values[m_step*2];
4. uint16_t value = 0;
5.
6. //设置 PWM 序列中各个周期的值：逐步修改占空比，实现呼吸灯效果
7. for (uint8_t i = 0; i < m_step; ++i)
8. {
9. value += step;
10. seq0_values[i] = value;
11. seq0_values[m_step+i] = m_top - value;
12. }

```

### 3. 初始化 PWM

同样，按照功能需求表定义 PWM 初始化配置结构体，这里注意定义 PWM 通道映射的引脚时，与“NRFX\_PWM\_PIN\_INVERTED”相或表示空闲状态时该引脚输出高电平（因为指示灯是低电平驱动，常态时要保证指示灯熄灭，因此 3 个通道映射的引脚均与“NRFX\_PWM\_PIN\_INVERTED”相或）。

本例中不关注 PWM 驱动的事件，无需注册事件处理函数，调用 `nrfx_pwm_init()` 函数初始化 PWM 时，参数“handler”设置为 NULL，代码清单如下。

#### 代码清单：初始化 PWM

```

1. //定义 PWM 初始化配置结构体并初始化参数
2. nrfx_pwm_config_t const config0 =
3. {
4. .output_pins =
5. {
6. //通道 0 映射到 P0.13（驱动开发板上的指示灯 D1），空闲状态输出高电平
7. BSP_LED_0 | NRFX_PWM_PIN_INVERTED,
8. //通道 1 映射到 P0.14（驱动开发板上的指示灯 D2），空闲状态输出高电平
9. BSP_LED_1 | NRFX_PWM_PIN_INVERTED,
10. //通道 2 映射到 P0.15（驱动开发板上的指示灯 D3），空闲状态输出高电平
11. BSP_LED_2 | NRFX_PWM_PIN_INVERTED,
12. //通道 3 不使用
13. NRFX_PWM_PIN_NOT_USED
14. },
15. .irq_priority = APP_IRQ_PRIORITY_LOWEST,//中断优先级
16. .base_clock = NRF_PWM_CLK_1MHz, //PWM 时钟频率设置为 1MHz
17. .count_mode = NRF_PWM_MODE_UP, //向上计数模式
18. .top_value = m_top, //计数最大值为 10000
19. .load_mode = NRF_PWM_LOAD_COMMON, //通用装载模式
20. .step_mode = NRF_PWM_STEP_AUTO //序列中的周期自动推进
21. };

```

```
22. //初始化 PWM
23. APP_ERROR_CHECK(nrfx_pwm_init(&m_pwm0, &config0, NULL));
```

#### 4. 启动 PWM 播放

定义播放序列、设置序列播放控制后，因为本例中只有一个序列，因此调用 nrfx\_pwm\_simple\_playback() 函数启动 PWM 序列播放。

nrfx\_pwm\_simple\_playback() 函数的参数 “flags” 设置为 NRFX\_PWM\_FLAG\_LOOP，序列播放完成后会自动触发任务重新播放，如果设置为 NRFX\_PWM\_FLAG\_STOP，则序列播放完成后 PWM 停止。

##### 代码清单：启动 PWM 序列播放

```
1. static void pwm_play(void)
2. {
3. //定义 PWM 播放序列，播放序列包含了 PWM 序列的起始地址、大小和序列播放控制描述
4. nrf_pwm_sequence_t const seq0 =
5. {
6. .values.p_common = seq0_values,//指向 PWM 序列
7. .length = NRF_PWM_VALUES_LENGTH(seq0_values),//PWM 序列中包含的周期个数
8. .repeats = 0, //序列中周期重复次数为 0
9. .end_delay = 0 //序列后不插入延时
10. };
11. //启动 PWM 序列播放
12. (void)nrfx_pwm_simple_playback(&m_pwm0, &seq0, 1,
13. NRFX_PWM_FLAG_LOOP);
14. }
```

##### 5.2.2. 硬件连接

本章的 PWM 实验会使用到 P0.13、P0.14、P0.15 和 P0.16 驱动 LED 指示灯 D1 D2 D3 D4（本实验只使用了指示灯 D1 D2 D3），这里我们将 4 个指示灯的跳线均短接上，以方便后续实验。



图 23-10：开发板跳线帽短接

### 5.2.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 23-1: PWM 呼吸灯-通用装载模式”，将解压后得到的文件夹“pwm\_common”拷贝到合适的目录，如“D\ nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\pwm\_common\project\mdk5”目录下的工程“pwm\_common.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nRF52840\_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，PWM 序列中的每一个 PWM 周期参数会同时装载到 3 个 PWM 通道，因此可观察到指示灯 D1、D2、D3 的呼吸灯效果完全一样。

## 5.3. 组装载模式实现呼吸灯实验

本实验在“实验 23-1: PWM 呼吸灯-通用装载模式”的基础上修改。本例和“实验 23-1”的区别是：

- 1) PWM0 的 4 个通道均使用，分别用于驱动指示灯 D1、D2、D3 和 D4。
- 2) 使用组装载模式。

❖ 注：本节对应的实验源码是：“实验 23-2: PWM 呼吸灯-组装载模式”。

### 5.3.1. 代码编写

代码编写部分只讲解和“实验 23-1”有区别的地方，以节省篇幅。

## 1. 定义 PWM 序列

PWM 序列定义为“nrf\_pwm\_values\_grouped\_t”类型，注意组装载模式序列中的每个变量均包含 2 个 PWM 周期参数，因此初始化赋值的时候 2 个周期参数都要赋值，代码清单如下。

### 代码清单：定义 PWM 序列和初始化序列中的值

```
1. //定义 PWM 序列（组装载模式），该序列必须位于 RAM 中，因此要定义为 static 类型的
2. uint16_t step = m_top / m_step;
3. static nrf_pwm_values_grouped_t seq0_values[m_step*2];
4. uint16_t value = 0;
5.
6. //设置 PWM 序列中各个周期的值：逐步修改占空比，实现呼吸灯效果
7. for (uint8_t i = 0; i < m_step; ++i)
8. {
9. value += step;
10. seq0_values[i].group_0 = value;
11. seq0_values[i].group_1 = value;
12. seq0_values[m_step+i].group_0 = m_top - value;
13. seq0_values[m_step+i].group_1 = m_top - value;
14. }
```

## 2. 初始化 PWM

初始化时将装载模式设置为组装载模式，即“.load\_mode=NRF\_PWM\_LOAD\_GROUPED”。

PWM0 的 4 个通道映射到 P0.13、P0.14、P0.15 和 P0.16，分别用于驱动指示灯 D1、D2、D3 和 D4，代码清单如下。

### 代码清单：初始化 PWM

```
24. .output_pins =
25. {
26. //通道 0 映射到 P0.13（驱动开发板上的指示灯 D1），空闲状态输出高电平
27. BSP_LED_0 | NRFX_PWM_PIN_INVERTED,
28. //通道 1 映射到 P0.14（驱动开发板上的指示灯 D2），空闲状态输出高电平
29. BSP_LED_1 | NRFX_PWM_PIN_INVERTED,
30. //通道 2 映射到 P0.15（驱动开发板上的指示灯 D3），空闲状态输出高电平
31. BSP_LED_2 | NRFX_PWM_PIN_INVERTED,
32. //通道 3 映射到 P0.15（驱动开发板上的指示灯 D4），空闲状态输出高电平
33. BSP_LED_3 | NRFX_PWM_PIN_INVERTED
34. }
```

### 5.3.2. 硬件连接

同“实验 23-1”。

### 5.3.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 23-2: PWM 呼吸灯-组装载模式”，将解压后得到的文件夹“pwm\_grouped”拷贝到合适的目录，如“D\nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\pwm\_grouped\project\mdk5”目录下的工程“pwm\_grouped.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nRF52840\_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，PWM 序列中的每一个 PWM 周期参数的“group\_0”会装载到通道 0 和通道 1，“group\_1”会装载到通道 2 和通道 3，因为“group\_0”和“group\_1”在初始化时赋值一样，因此可观察到指示灯 D1、D2、D3 和 D4 的呼吸灯效果完全一样。这里我们也可以看到：本例实现的效果和“实验 23-1”一样，但是本例 PWM 序列占用的 RAM 是“实验 23-1”的 2 倍。

## 5.4. 独立装载模式实现呼吸灯实验

本实验在“实验 23-1: PWM 呼吸灯-通用装载模式”的基础上修改。本例和“实验 23-1”的区别是：

- 1) PWM0 的 4 个通道均使用，分别用于驱动指示灯 D1、D2、D3 和 D4。
- 2) 使用独立装载模式。
- 3) 应用程序关注 PWM 驱动的 NRF\_DRV\_PWM\_EVT\_FINISHED 事件，在该事件中修改占空比。

◆ 注：本节对应的实验源码是：“实验 23-3: PWM 呼吸灯-独立装载模式”。

### 5.4.1. 代码编写

代码编写部分只讲解和“实验 23-1”有区别的地方，以节省篇幅。

#### 1. 定义 PWM 序列

PWM 序列定义为“nrf\_pwm\_values\_individual\_t”类型，本例中 PWM 周期的占空比在 PWM 事件处理函数中修改，因此定义的序列中只包含了一个变量，该变量包含 4 个 PWM 周期参数，分别用于通道 0~3，代码清单如下。

代码清单：定义 PWM 序列和呼吸灯用到的参数

```
1. //最大计数值
2. static uint16_t const m_top = 10000;
3. //步进值
4. static uint16_t const m_step = 200;
```

```
5. //记录4个呼吸灯的8个阶段，每个呼吸灯2个阶段：占空比按照步进值递增到最大，再按照步进值
6. //递减到最小
7. static uint8_t m_phase;
8.
9. //定义PWM序列（独立装载模式），该序列必须位于RAM中，因此要定义为static类型的
10. static nrf_pwm_values_individual_t seq0_values;
```

## 2. 初始化 PWM

初始化时将装载模式设置为独立装载模式，即“.load\_mode= NRF\_PWM\_LOAD\_INDIVIDUAL”。

PWM 初始化时注册事件处理函数 pwm0\_handler(), 该函数中关注“NRFX\_PWM\_EVT\_FINISHED”事件，在该事件中修改占空比，事件处理函数代码清单如下。

PWM0 的 4 个通道映射到 P0.13、P0.14、P0.15 和 P0.16，分别用于驱动指示灯 D1、D2、D3 和 D4，代码清单如下。

### 代码清单：PWM 事件处理参数

```
1. static void pwm0_handler(nrfx_pwm_evt_type_t event_type)
2. {
3. //序列播放完成事件
4. if (event_type == NRFX_PWM_EVT_FINISHED)
5. {
6. uint8_t channel = m_phase >> 1;
7. bool down = m_phase & 1;
8. bool next_phase = false;
9. //获取当前通道的 PWM 周期值
10. uint16_t * p_channels = (uint16_t *)&seq0_values;
11. uint16_t value = p_channels[channel];
12.
13. if (down) //down 置位，占空比按照步进值递减
14. {
15. value -= m_step;
16. if (value == 0)
17. {
18. next_phase = true;
19. }
20. }
21. else //否则，占空比按照步进值递增
22. {
23. value += m_step;
24. if (value >= m_top)
25. {
26. next_phase = true;
27. }
28. }
29. }
30. }
```

```
28. }
29. //修改通道周期参数
30. p_channels[channel] = value;
31. //执行呼吸灯的下一个阶段
32. if (next_phase)
33. {
34. //每个呼吸灯 2 个阶段，因此要乘以 2
35. if (++m_phase >= 2 * NRF_PWM_CHANNEL_COUNT)
36. {
37. m_phase = 0; //若 4 个通道全部执行完成，重新开始
38. }
39. }
40. }
41. }
```

#### 5.4.2. 硬件连接

同“实验 23-1”。

#### 5.4.3. 实验步骤

1. 解压“…\3：开发指南（上册）配套实验源码\”目录下的压缩文件“实验 23-3：PWM 呼吸灯-组装载模式”，将解压后得到的文件夹“pwm\_individual”拷贝到合适的目录，如“D\ nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\pwm\_individual\project\mdk5”目录下的工程“pwm\_individual.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nRF52840\_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，可以观察到 4 个指示灯轮流实现呼吸灯效果。

### 5.5. 波形装载模式实现呼吸灯实验

本实验在“实验 23-1：PWM 呼吸灯-通用装载模式”的基础上修改。本例和“实验 23-1”的区别是：

- 1) 使用 PWM0 的 3 个通道（通道 0~2）用于驱动指示灯 D1、D2、D3（波形装载模式最多只能使用 3 个通道），通道 0 占空比设置为 90%，通道 1 占空比设置为 70%，通道 2 占空比设置为 50%。由于指示灯是低电平驱动，因此，3 个指示灯的亮度会不一样，D1 亮度最弱，D3 最亮。
- 2) 使用波形装载模式。

◆ 注：本节对应的实验源码是：“实验 23-4：PWM 驱动指示灯-波形装载模式”。

### 5.5.1. 代码编写

代码编写部分只讲解和“实验 23-1”有区别的地方，以节省篇幅。

#### 1. 定义 PWM 序列

PWM 序列定义为“nrf\_pwm\_values\_wave\_form\_t”类型，注意波形装载模式下最多只能使用 3 个通道，第 4 个 PWM 参数会装载到 COUNTERTOP 寄存器作为最大计数值，代码清单如下。

#### 代码清单：定义 PWM 序列和初始化序列中的值

```
1. //定义 PWM 序列，该序列必须位于 RAM 中，因此要定义为 static 类型的
2. static nrf_pwm_values_wave_form_t seq0_values[] =
3. {
4. 1000, //通道 1，占空比 90%
5. 3000, //通道 2，占空比 70%
6. 5000, //通道 3，占空比 50%
7. 10000 //最大比较直
8. };
```

#### 2. 初始化 PWM

初始化时将装载模式设置为波形装载模式，即“.load\_mode= NRF\_PWM\_LOAD\_WAVE\_FORM”。

PWM0 的 3 个通道映射到 P0.13 P0.14 和 P0.15，分别用于驱动指示灯 D1、D2 和 D3 代码清单如下。

#### 代码清单：初始化 PWM

```
1. static void pwm_common_init(void)
2. {
3. //定义 PWM 初始化配置结构体并初始化参数
4. nrfx_pwm_config_t const config0 =
5. {
6. .output_pins =
7. {
8. //通道 0 映射到 P0.13（驱动开发板上的指示灯 D1），空闲状态输出高电平
9. BSP_LED_0 | NRFX_PWM_PIN_INVERTED,
10. //通道 1 映射到 P0.14（驱动开发板上的指示灯 D2），空闲状态输出高电平
11. BSP_LED_1 | NRFX_PWM_PIN_INVERTED,
12. //通道 2 映射到 P0.15（驱动开发板上的指示灯 D3），空闲状态输出高电平
13. BSP_LED_2 | NRFX_PWM_PIN_INVERTED,
14. NRFX_PWM_PIN_NOT_USED //因为使用了波形装载模式，通道 3 不使用
15. },
16. .irq_priority = APP_IRQ_PRIORITY_LOWEST,//中断优先级
17. .base_clock = NRF_PWM_CLK_1MHz, //PWM 时钟频率设置为 1MHz
```

```

18. .count_mode = NRF_PWM_MODE_UP, //向上计数模式
19. .top_value = 0, //使用波形装载模式时，该值被忽略
20. .load_mode = NRF_PWM_LOAD_WAVE_FORM, //波形装载模式
21. .step_mode = NRF_PWM_STEP_AUTO //序列中的周期自动推进
22. };
23. //初始化 PWM
24. APP_ERROR_CHECK(nrfx_pwm_init(&m_pwm0, &config0, NULL));
25. }

```

### 5.5.2. 硬件连接

同“实验 23-1”。

### 5.5.3. 实验步骤

- 解压“…\3：开发指南（上册）配套实验源码\”目录下的压缩文件“实验 23-4：PWM 驱动指示灯-波形装载模式”，将解压后得到的文件夹“pwm\_waveform”拷贝到合适的目录，如“D\ nRF52840”。
- 启动 MDK5.23。
- 在 MDK5 中执行“Project→Open Project”打开“…\pwm\_waveform\project\mdk5”目录下的工程“pwm\_waveform.uvproj”。
- 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nRF52840\_qiaa.hex”位于工程目录下的“Objects”文件夹中。
- 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
- 程序运行后，可以观察到：由于 3 个通道的占空比不一样，因此 3 个指示灯的亮度不一样，通道 0 的占空比最高，D1 亮度最弱，D3 最亮。

## 5.6. 复合序列驱动振动马达实验

本实验在“实验 23-1：PWM 呼吸灯-通用装载模式”的基础上修改。本例中使用 P1.07 驱动振动马达，定义 2 个具有不同占空比的序列，以达到控制振动马达强度的目的，本例的功能需求如下表所示。

表 23-27：功能需求表

| 使用的 PWM 外设 | PWM0                                              |
|------------|---------------------------------------------------|
| 引脚配置       | 通道 0 映射到 P1.03 驱动振动马达，因为振动马达是高电平驱动，因此引脚空闲状态输出低电平。 |
|            | 不使用。                                              |
|            | 不使用。                                              |
|            | 不使用。                                              |
| PWM 周期     | PWM 时钟频率：1MHz，最大计数值：10000，周期 = 10ms。              |
| PWM 序列     | 定义 2 个序列，序列 0 占空比为 30%，序列 1 的占空比为 90%，两个          |

|        |                                                                             |
|--------|-----------------------------------------------------------------------------|
|        | 序列串接播放时，因为占空比不一样，马达振动强度不一样。                                                 |
| 序列播放控制 | 序列 0 包含一个 PWM 周期参数，周期重复 200 次、不插入延时；序列 1 包含 2 个周期参数，周期重复 100 次、不插入延时。序列不回放。 |
| PWM 事件 | 应用程序不关注 PWM 事件，即 PWM 初始化时不注册事件处理函数。                                         |

❖ 注：本节对应的实验源码是：“实验 23-5：PWM 驱动振动马达-复合序列”。

### 5.6.1. 代码编写

#### 1. 定义 PWM 序列

PWM 序列定义如下，这里需要注意的是：因为先播放序列 0，后播放序列 1，所以如果序列 1 中只包含一个周期参数的话，周期是不能重复的，因此序列 1 包含了 2 个周期参数。  
**代码清单：定义 PWM 序列**

```
1. //定义 PWM 序列，该序列必须位于 RAM 中，因此要定义为 static 类型
2. static nrf_pwm_values_common_t seq0_values[] = {7000}; //序列 0，占空比 30%
3. static nrf_pwm_values_common_t seq1_values[]; //序列 1，占空比 90%
```

#### 2. 初始化 PWM

这里注意定义 PWM 通道映射的引脚时，由于振动马达高电平有效，因此空闲状态时要输出低电平，即定义引脚时不能与“NRFX\_PWM\_PIN\_INVERTED”相或，代码清单如下。

---

#### 代码清单：初始化 PWM

---

```
1. static void pwm_common_init(void)
2. {
3. //定义 PWM 初始化配置结构体并初始化参数
4. nrfx_pwm_config_t const config0 =
5. {
6. .output_pins =
7. {
8. //通道 0 映射到 P1.07 用于驱动振动马达模块，由于振动马达高电平有效，因此空闲状态
9. //输出低电平
10. MOTOR_CONTROL_PIN,
11. NRFX_PWM_PIN_NOT_USED, //通道 1 不使用
12. NRFX_PWM_PIN_NOT_USED, //通道 2 不使用
13. NRFX_PWM_PIN_NOT_USED //通道 3 不使用
14. },
15. .irq_priority = APP_IRQ_PRIORITY_LOWEST, //中断优先级
16. .base_clock = NRF_PWM_CLK_1MHz, //PWM 时钟频率设置为 1MHz
17. .count_mode = NRF_PWM_MODE_UP, //向上计数模式
18. .top_value = 10000, //最大计数值 10000
19. .load_mode = NRF_PWM_LOAD_COMMON, //波形装载模式
20. .step_mode = NRF_PWM_STEP_AUTO //序列中的周期自动推进
21. };
}
```

```

22. //初始化 PWM
23. APP_ERROR_CHECK(nrfx_pwm_init(&m_pwm0, &config0, NULL));
24. }

```

### 3. 启动 PWM 播放

复合序列使用 nrfx\_pwm\_complex\_playback()函数播放，本例使用了 2 个序列，相应地，要定义 2 个播放序列，代码清单如下。

#### 代码清单：启动 PWM 播放

```

1. static void pwm_play(void)
2. {
3. //定义 PWM 播放序列，播放序列包含了 PWM 序列的起始地址、大小和序列播放控制描述
4. nrf_pwm_sequence_t const seq0 =
5. {
6. .values.p_common = seq0_values,//指向 PWM 序列 0
7. //PWM 序列中包含的周期个数
8. .length = NRF_PWM_VALUES_LENGTH(seq0_values),
9. .repeats = 200, //序列中周期重复次数为 200
10. .end_delay = 0 //序列后不插入延时
11. };
12. nrf_pwm_sequence_t const seq1 =
13. {
14. .values.p_common = seq1_values,//指向 PWM 序列 1
15. //PWM 序列中包含的周期个数
16. .length = NRF_PWM_VALUES_LENGTH(seq1_values),
17. .repeats = 100, //序列中周期重复次数为 100
18. .end_delay = 0 //序列后不插入延时
19. };
20. //启动 PWM 序列播放，flags 设置为 NRFX_PWM_FLAG_LOOP：序列播放完成后，自动触发任
21. //务重新播放如改为 NRFX_PWM_FLAG_STOP，则播放结束后，PWM 停止
22. (void)nrfx_pwm_complex_playback(&m_pwm0, &seq0, &seq1, 1,
23. NRFX_PWM_FLAG_LOOP);
24. }

```

#### 5.6.2. 硬件连接

本实验需要连接振动马达模块，模块和开发板之间的引脚连接如下表所示。

表 23-28：振动马达模块和开发板连接

| 振动马达模块 | 开发板         |
|--------|-------------|
| GND    | JP8 的 GND。  |
| VCC    | JP8 的 3V3。  |
| IN     | JP3 的 P107。 |

振动马达模块和开发板的连接图如下：

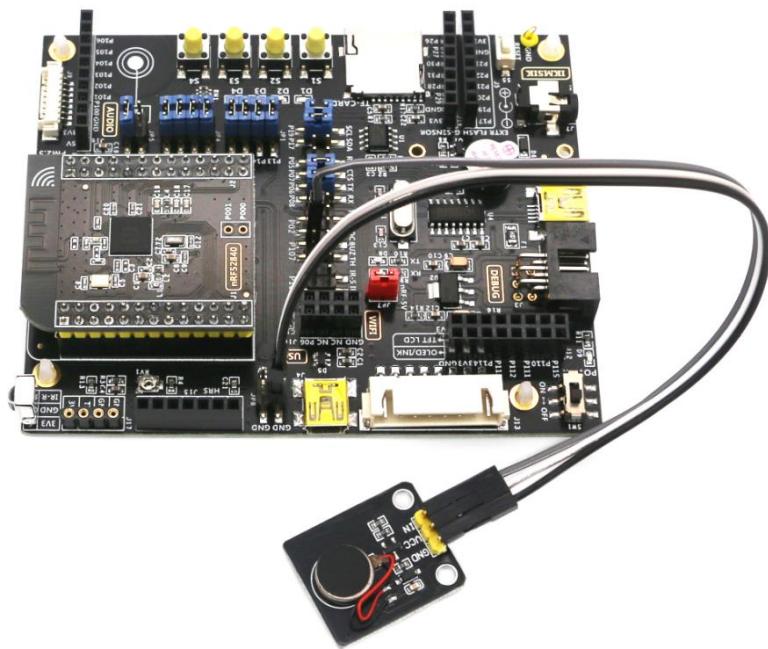


图 23-11：振动马达模块和开发板图

### 5.6.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 24-5: PWM 驱动振动马达-复合序列”，将解压后得到的文件夹“pwm\_complex”拷贝到合适的目录，如“D\ nRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\pwm\_common\project\mdk5”目录下的工程“pwm\_common.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nRF52840\_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载到 nRF52840 进行仿真。
6. 程序运行后，可观察到振动马达开始振动，振动强度 2 秒弱（序列 0 播放时）、2 秒强（序列 1 播放时）。

#### ■ 本章思考题：如何同时使用多个 PWM 外设？

提示：使用多个 PWM 外设时主要步骤如下：

- 1) 定义多个 PWM 驱动程序实例，并在“sdk\_config.h”文件的 PWM 配置项中勾选对应的实例。
- 2) 为每个实例定义 PWM 序列和播放序列。
- 3) 分别对每个实例进行初始化和启动播放。

实验源码参见：实验 23-6：同时使用 4 个 PWM 外设。

## 第二十五章：DHT11 数字温湿度传感器

### 1. 学习目的

1. 了解 DHT11 数字温湿度传感器的基本原理及其数据格式。
2. 掌握 nRF52840 与 DHT11 单总线通信的程序设计，通讯步骤，数据校验等。

### 2. 硬件电路设计

#### 2.1. 湿敏元件简介

湿敏元件是最简单的湿度传感器。湿敏元件比较常见的是电阻式湿敏元件和电容式湿敏元件。除此之外，还有电解质离子型湿敏元件、重量型湿敏元件(利用感湿膜重量的变化来改变振荡频率)、光强型湿敏元件、声表面波湿敏元件等。下面就电阻式湿敏元件和电容式湿敏元件进行介绍。

电阻式湿敏元件一般指的是湿敏电阻，湿敏电阻的特点是在基片上覆盖一层用感湿材料制成的膜，当空气中的水蒸气吸附在感湿膜上时，元件的电阻率和电阻值都发生变化，利用这一特性即可测量湿度。湿敏电阻的种类很多，例如金属氧化物湿敏电阻、硅湿敏电阻、陶瓷湿敏电阻等。

湿敏电阻的优点是灵敏度高，主要缺点是线性度和产品的互换性差。

电容式湿敏元件一般指的就是湿敏电容，湿敏电容一般是用高分子薄膜电容制成的，常用的高分子材料有聚苯乙烯、聚酰亚胺、醋酸醋酸纤维等。当环境湿度发生改变时，湿敏电容的介电常数发生变化，使其电容量也发生变化，其电容变化量与相对湿度成正比。

湿敏电容的主要优点是灵敏度高、产品互换性好、响应速度快、湿度的滞后量小、便于制造、容易实现小型化和集成化，其精度一般比湿敏电阻要低一些。

湿敏元件的线性度及抗污染性差，在检测环境湿度时，湿敏元件要长期暴露在待测环境中，很容易被污染而影响其测量精度及长期稳定性。

温度的概念大家比较了解，下面重点介绍下与湿度有关的知识点。湿度，一般在气象学中指的是空气湿度，它是空气中水蒸气的含量。空气中液态或固态的水不算在湿度中，不含水蒸气的空气被称为干空气。下面介绍与湿度有关的 3 个概念。

- 相对湿度 (Relative Humidity) 是指气体中 (通常为空气中) 所含水蒸气量 (水蒸气压) 与其空气相同情况下饱和水蒸气量 (饱和水蒸气压) 的百分比，一般用百分数表示。通常所说的湿度都是指相对湿度，比如说某个房间的湿度为 60%，即指该房间的相对湿度是 60%。
- 绝对湿度是指一定体积的空气中含有的水蒸气的质量，一般其单位是克/立方米。绝对湿度的最大限度是饱和状态下的最高湿度。绝对湿度只有与温度一起才有意义，因为空气中能够含有的湿度的量随温度而变化，在不同的温度中绝对湿度也不同，因为随着温度的变化空气的体积也要发生变化。但绝对湿度越靠近最高湿度，它随温度的变化就越

小。

- 饱和湿度是表示在一定温度下，单位容积空气中所能容纳的水蒸气量的最大限度。如果超过这个限度，多余的水蒸气就会凝结，变成水滴，此时的空气湿度便称为饱和湿度。它们之间的关系是：相对湿度 = 绝对湿度 / 饱和湿度 × 100% 。

## 2.2. DHT11 数字温湿度传感器

DHT11 数字温湿度传感器是一款含有已校准数字信号输出的温湿度复合传感器，也是一款电容式湿度传感器。它应用专用的数字模块采集技术和温湿度传感技术，确保产品具有极高的可靠性与卓越的长期稳定性。传感器包括一个电容式感湿元件和一个 NTC 测温元件，并与一个高性能 8 位单片机相连接。

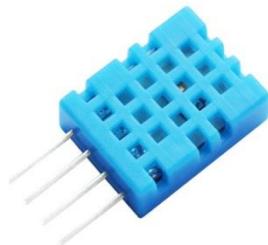


图 25-1：DHT11 数字温湿度传感器实物图

### 1. DHT11 数字温湿度传感器的规格参数

DHT11 数字温湿度传感器的规格参数如下表所示。

表 25-1：DHT11 数字温湿度传感器规格参数

| DHT11 管脚号 | 功能描述                        |
|-----------|-----------------------------|
| 工作电压      | 3.3V~5.5V。                  |
| 外形尺寸      | 23.2(L)mm×12.5(W)mm。        |
| 测量范围      | 温度： -20~+60°C 湿度： 5~95%RH。  |
| 精度        | 温度： ±2°C。 湿度： ±5%RH (25°C)。 |
| 分 辨 率     | 温度： 0.1°C。 湿度： 1%RH。        |
| 衰 减 值     | 温度： <0.1°C/年。 湿度： <1%RH/年。  |
| 输出信号      | 单总线数字信号。                    |
| 外壳材料      | ABS 塑料。                     |
| 重量        | 1g。                         |
| 引脚数       | 4 个。                        |

### 2. DHT11 数字温湿度传感器的管脚定义

DHT11 数字温湿度传感器的引脚定义如下。

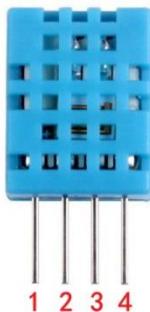


图 25-2: DHT11 数字温湿度传感器管脚号

4 个引脚的功能如下表所示。

表 25-2: DHT11 数字温湿度传感器引脚定义

| DHT11 管脚号 | 管脚名  | 功能描述             |
|-----------|------|------------------|
| 1         | VCC  | 供电, (3.3~5.5) V。 |
| 2         | DATA | 串行数据, 单总线。       |
| 3         | NC   | 空脚。              |
| 4         | GND  | 接地, 电源负极。        |

### 2.3. DHT11 接口电路

IK-52840DK 开发板上设计了 DHT11 数字温湿度传感器接口，电路如下图所示，这里需要注意的是 DHT11 的 DATA 引脚需要上拉，建议连接线长度短于 5m 时用 4.7K 上拉电阻，大于 5m 时根据实际情况降低上拉电阻的阻值。

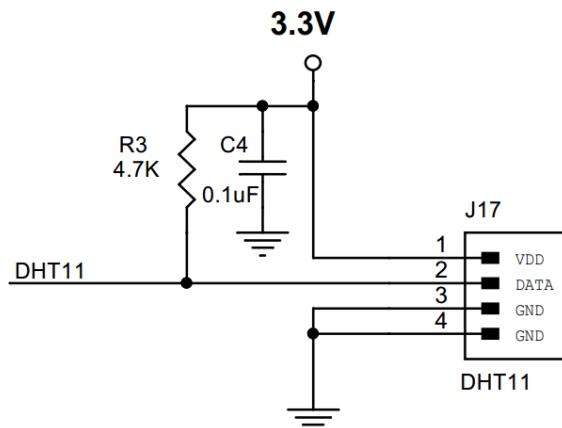


图 25-3: DHT11 接口电路

DHT11 占用的 nRF52840 的引脚如下表：

表 25-3: DHT11 引脚分配

| DHT11    | 引脚    | 说明     |
|----------|-------|--------|
| DHT11 数据 | P1.07 | 和蜂鸣器共用 |

### 3. 软件设计

DHT11 数字温湿度传感器采用简化的单总线通信。单总线即只有一根数据线，系统中的数据交换、控制均由单总线完成。设备（主机或从机）通过一个漏极开路或三态端口连至该数据线，以允许设备在不发送数据时能够释放总线，而让其它设备使用总线；单总线通常要求外接一个约  $4.7\text{k}\Omega$  的上拉电阻，这样，当总线闲置时，其状态为高电平。由于它们是主从结构，只有主机呼叫从机时，从机才能应答，因此主机访问器件都必须严格遵循单总线序列，如果出现序列混乱，器件将不响应主机。

MCU 作为单总线通信的主机与 DHT11 从机之间通信的步骤如下图所示，共需 4 个步骤。

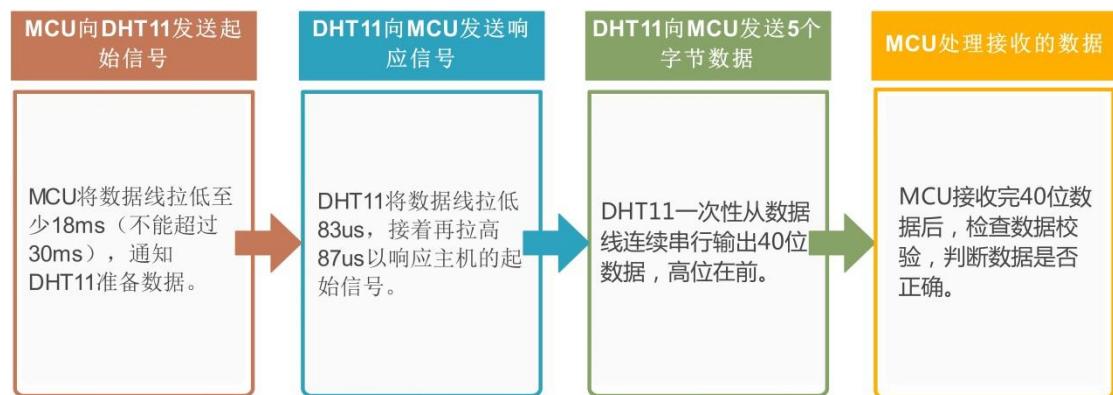


图 25-4: MCU 与 DHT11 单总线通信步骤

#### 1. MCU 向 DHT11 发送起始信号

DHT11 上电后要等待 1S 以越过不稳定状态，在此期间不能发送任何指令。MCU 的 I/O 此时配置为输入，因为 DHT11 的 DATA 数据线接了上拉电阻，因此保持高电平。

MCU 从 DHT11 读取温湿度数据时，首先发送起始信号，即 MCU 配置 I/O 为输出，同时输出低电平，且低电平保持时间不能小于 18ms（最大不得超过 30ms），然后 MCU 的 I/O 设置为输入状态，由于上拉电阻，微处理器的 I/O 即 DHT11 的 DATA 数据线也随之变高，等待 DHT11 作出回答信号。

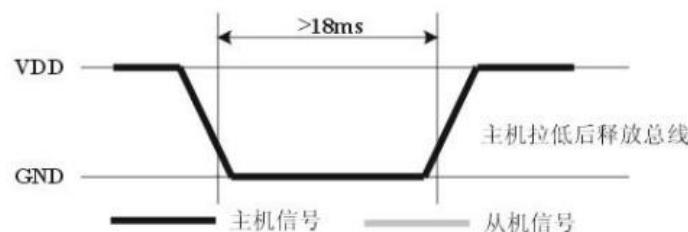


图 25-5: 起始信号时序

#### 2. DHT11 向 MCU 发送响应信号

DHT11 的 DATA 引脚检测到外部信号有低电平时，等待外部信号低电平结束，延迟后 DHT11 的 DATA 引脚处于输出状态，输出 83 微秒的低电平作为应答信号，紧接着输出 87 微秒的高电平通知外设准备接收数据，MCU 的 I/O 此时处于输入状态，检测到 I/O 有低电平（DHT11 回应信号）后，等待 87 微秒的高电平后的数据接收。

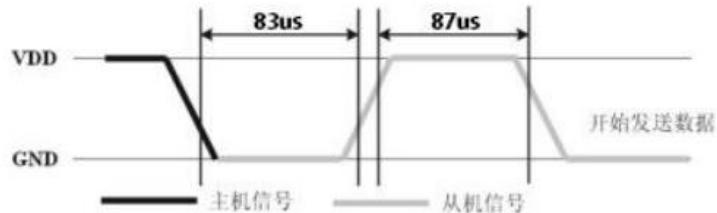


图 25-6: DHT11 响应信号时序

### 3. DHT11 向 MCU 发送 5 个字节数据

由 DHT11 的 DATA 引脚输出 40 位数据, MCU 根据 I/O 电平的变化接收 40 位数据, “0” 和 “1”的格式定义如下:

- 位数据 “0”: 54us 的低电平 + (23~27) us 的高电平;
- 位数据 “1”: 54us 的低电平 + (68~74) us 的高电平。

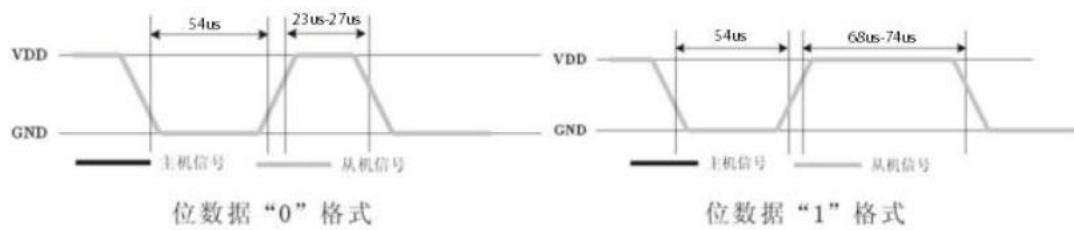


图 25-7: “0” 和 “1”的时序

40 位数据包含了温湿度和校验, 数据格式为: 8bit 湿度整数数据+8bit 湿度小数数据+8bit 温度整数数据+8bit 温度小数数据+8bit 校验位, 其中湿度小数部分为 0。

结束信号: DHT11 的 DATA 引脚输出 40 位数据后, 继续输出低电平 54 微秒后转为输入状态, 由于上拉电阻随之变为高电平。但 DHT11 内部重测环境温湿度数据, 并记录数据, 等待外部信号的到来。

### 4. MCU 处理接收的数据

MCU 接收完数据之后, 需要检查数据的校验, 以判断接收的数据是否正确, 检查方式为: 8bit 校验位=8bit 湿度整数数据+8bit 湿度小数数据+8bit 温度整数数据+8bit 温度小数数据。

MCU 解析温度数据时, 需要注意 DHT11 正/负温度的表示方式: 当温度低于 0°C 时温度数据的低 8 位的最高位置为 1, 否则为 0。

#### ■ 正温示例

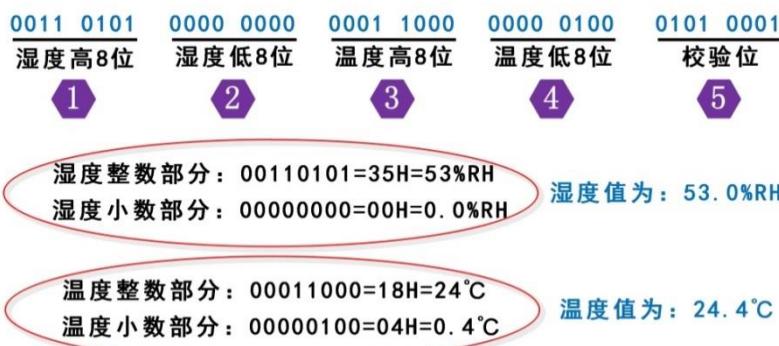
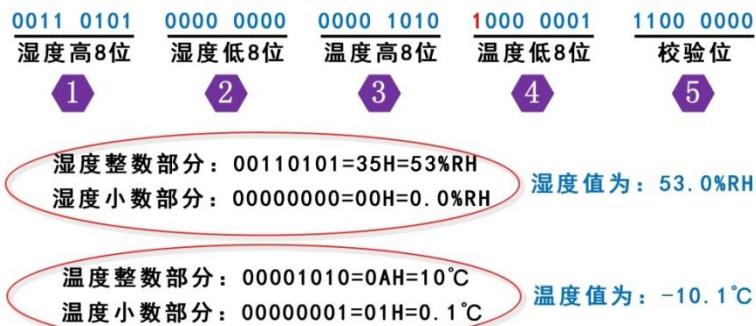


图 25-8: 正温度数据

## ■ 负温示例



注：温度低8位的最高位为1时表示温度值为负温。

图 25-9: 负温度数据

### 3.1. DHT11 温湿度读取实验

本实验在“实验 10-1：串口数据收发（查询方式）”的基础上修改，每 2.5 秒从 DHT11 读取一次温湿度数据，正确读取后通过串口输出温湿度数据。

❖ 注：本节对应的实验源码是：“实验 25-1：DHT11 温湿度读取”。

#### 3.1.1. 代码编写

按照前文描述的步骤，编写读取 DHT 数据的代码。MCU 首先向 DHT11 发送起始信号、之后等待 DHT11 向 MCU 发送响应信号，接收到响应信号后 MCU 继续接收 DHT11 向 MCU 发送的 40 位数据。注意数据读取完成后，需要检查数据的校验。

##### 代码清单：读取温湿度数据

```

1. ****
2. * 描述：一次完整的数据传输为 40bit，高位在前
3. * 入参：8bit 湿度整数 + 8bit 湿度小数 + 8bit 温度整数 + 8bit 温度小数+8bit 校验
4. * 返回值：DHT11_SUCCESS: 成功, DHT11_NACK: 无应答, DHT11_DATA_ERR: 数据校验错误
5. ****
6. uint32_t Read_DHT11(DHT11_Data_t *DHT11_Data)
7. {
8. //配置引脚为输出
9. PIN_DATA_OUT
10. //引脚输出低电平
11. PIN_DATA_CLEAR;
12. //低电平时间不能小于 18ms，不能超过 30ms，这里延时 19ms
13. nrf_delay_ms(19);
14. //引脚拉高
15. PIN_DATA_SET;
16. //配置引脚为输入
17. PIN_DATA_IN;

```

```

18. //延时 30us
19. nrf_delay_us(30);
20.
21. //DHT11 有低电平响应信号，执行数据读取，否则返回错误代码：DHT11_NACK
22. if(nrf_gpio_pin_read(DATA_PIN) == 0)
23. {
24. //DHT11 发出 83us 低电平应答信号。主机等待低电平结束
25. if(waitfor_state(1)!=DHT11_SUCCESS) return DHT11_NACK;
26. //DHT11 发出 87us 高电平信号通知主机准备接收数据。主机等待高电平结束
27. if(waitfor_state(0)!=DHT11_SUCCESS) return DHT11_NACK;
28. //数据格式：8bit 湿度整数数据+8bit 湿度小数数据+8bit 温度整数数据+8bit 温度
29. //小数数据+8bit 校验位，因此这里要连续读出 5 个字节
30. DHT11_Data->h_int= Read_Byte();
31. DHT11_Data->h_deci= Read_Byte();
32. DHT11_Data->t_int= Read_Byte();
33. DHT11_Data->t_deci= Read_Byte();
34. DHT11_Data->check_sum= Read_Byte();
35.
36. PIN_DATA_OUT; //读取结束，引脚改为输出模式
37. PIN_DATA_SET; //主机拉高
38.
39. //校验：校验和 = 8bit 湿度整数数据+8bit 湿度小数数据+8bit 温度整数数据+8bit
40. //温度小数数据+8bit 校验位
41. if(DHT11_Data->check_sum == DHT11_Data->h_int + DHT11_Data->h_deci +
42. DHT11_Data->t_int+ DHT11_Data->t_deci)
43. return DHT11_SUCCESS;
44. else
45. return DHT11_DATA_ERR;//校验错误，返回错误代码 DHT11_DATA_ERR
46. }
47. else //DHT11 无响应，返回错误代码 DHT11_NACK
48. {
49. return DHT11_NACK;
50. }
51.}

```

上面代码中的 31~34 行，MCU 通过调用 Read\_Byte() 函数分 5 次接收 DHT11 的 40 位数据（40 位共 5 个字节），这样更便于程序的处理。Read\_Byte() 函数代码清单如下。

#### 代码清单：接收一个字节数据

```

1. ****
2. * 描述：从 DHT11 读取一个字节，MSB 在前
3. * 入参：无
4. * 返回值：接收的 1 字节数据
5. ****

```

```

6. static uint8_t Read_Byte(void)
7. {
8. uint8_t i, temp=0;
9. //循环读取 8 个位
10. for(i=0;i<8;i++)
11. {
12. //每 bit 以 50us 低电平标置开始。等待引脚状态变为高电平，100us 无变化认为超时，
13. //退出函数
14. if(waitfor_state(1)!=DHT11_SUCCESS) return DHT11_NACK;
15.
16. //DHT11 以 23~27us 的高电平表示 0，以 68~74us 的高电平表示 1
17. nrf_delay_us(40);
18. //40us 后仍为高电平表示数据 1
19. if(nrf_gpio_pin_read(DATA_PIN)==1)
20. {
21. //等待数据 1 的高电平结束，低电平到来
22. if(waitfor_state(0)!=DHT11_SUCCESS) return DHT11_NACK;
23. //保存数据位“1”，MSB 在前
24. temp|=(uint8_t)(0x01<<(7-i));
25. }
26. else//40us 后为低电平表示数据 0
27. {
28. //保存数据位“0”，MSB 在前
29. temp&=(uint8_t)~(0x01<<(7-i));
30. }
31. }
32. return temp;
33. }

```

MCU 在等待 DHT11 信号时，为了防止堵塞，我们在等待中加入超时时间，如果达到超时时间后，MCU 等待的信号没有出现，则认为出现错误，程序退出等待，同时返回错误代码，这样就可以避免由于 DHT11 故障导致出现堵塞的问题，同时通过错误代码通知了应用程序故障原因。MCU 等待 DHT11 信号的代码清单如下。

### 代码清单：等待 DHT11 信号

```

1. ****
2. * 描述：等待连接 DHT11 的引脚状态变化，为了防止 DHT11 损坏出现堵塞，设置超时时间为
3. * 100us
4. * 入参：[in]pin_state, 0=低电平，1=高电平
5. * 返回值：DHT11_SUCCESS：成功，DHT11_NACK：无应答
6. ****
7. static uint32_t waitfor_state(bool pin_state)
8. {
9. //超时时间 100us，100us 内引脚状态无变化，认为超时

```

```
10. uint8_t delay_us = 100;
11. do
12. {
13. //读取引脚状态, 如引脚状态变化, 函数运行结束, 返回 DHT11_SUCCESS
14. if(nrf_gpio_pin_read(DATA_PIN)==pin_state)
15. {
16. return DHT11_SUCCESS;
17. }
18. nrf_delay_us(1);
19. delay_us--;
20. }while(delay_us);
21. //返回错误代码: DHT11_NACK
22. return DHT11_NACK;
23. }
```

最后，在主函数中使用 Read\_DHT11() 函数每隔 2.5 秒读一次温湿度数据，并通过串口输出数据。

#### 代码清单：等待 DHT11 信号

```
1. int main(void)
2. {
3. //设置 GPIO 输出电压为 3.3V
4. gpio_output_voltage_setup_3v3();
5. //初始化开发板上的 4 个 LED, 即将驱动 LED 的 GPIO 配置为输出,
6. bsp_board_init(BSP_INIT_LEDS);
7. //初始化串口
8. uart_config();
9. nrf_delay_ms(1000);
10. while(true)
11. {
12. //从 DHT11 读取温湿度数据
13. uint32_t err_code = Read_DHT11(&DHT11_Data);
14. //判断函数返回值, 读取成功, 串口输出温湿度数据
15. if(err_code==DHT11_SUCCESS)
16. {
17. printf("Temperature: %d.%d °C Humidity: %d.%d %RH\r\n", DHT11_Data
18. .t_int,DHT11_Data.t_deci,DHT11_Data.h_int,DHT11_Data.h_deci);
19. }
20. //数据校验错误
21. else if(err_code==DHT11_DATA_ERR)
22. {
23. printf("Data Checksum ERROR!\r\n");
24. }
25. //DHT11 无应答
```

```

26. else if(err_code==DHT11_NACK)
27. {
28. printf("No ack ERROR!\r\n");
29. }
30. else
31. {
32. }
33. //每次读取温湿度数据之间的间隔不能小于 2 秒
34. nrf_delay_ms(2500);
35. //指示灯 D1 状态翻转，指示程序的运行
36. nrf_gpio_pin_toggle(LED_1);
37. }
38.

```

### 3.1.2. 硬件连接

本实验需要使用 P0.13 驱动 LED 指示灯 D1, P0.06 和 P0.08 作为串口通讯引脚, P1.07 连接 DHT11 数字温湿度传感器, 按照下图所示安装 DHT11 和短接跳线帽。

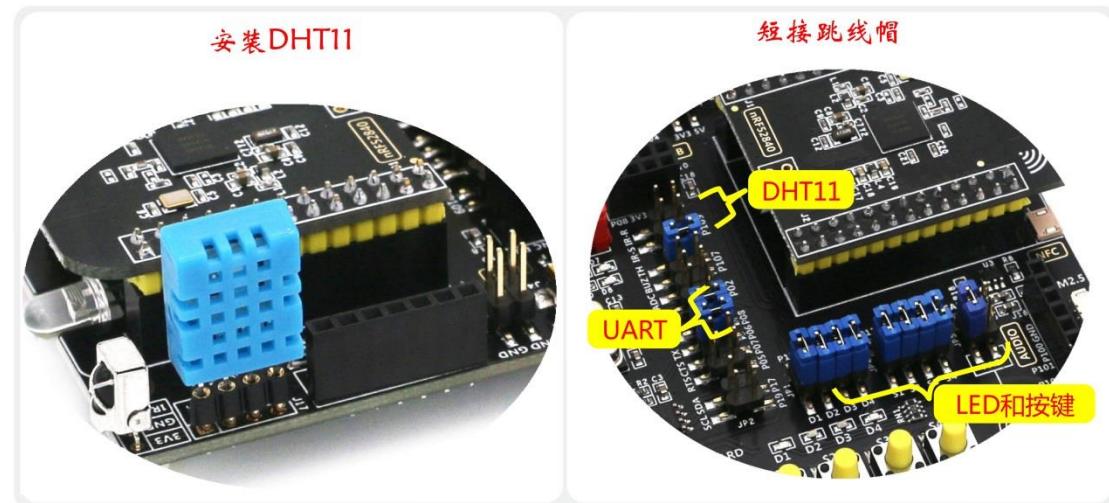


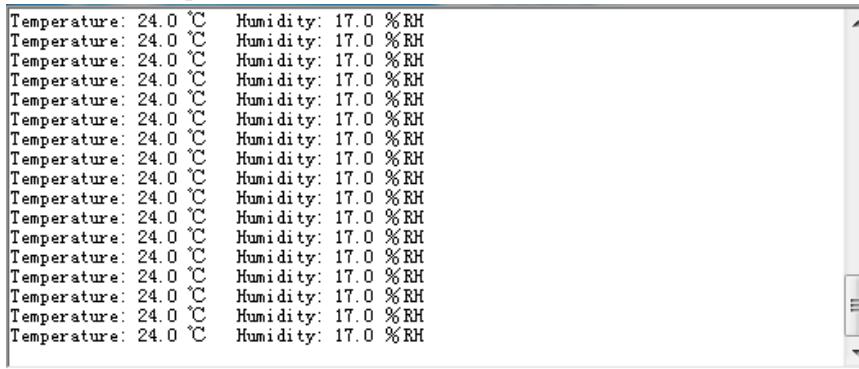
图 25-10: DHT11 安装和跳线帽短接

### 3.1.3. 实验步骤

1. 解压“…\3: 开发指南（上册）配套实验源码\”目录下的压缩文件“实验 25-1: DHT11 温湿度读取”，将解压后得到的文件夹“DHT11”拷贝到合适的目录，如“D\NRF52840”。
2. 启动 MDK5.23。
3. 在 MDK5 中执行“Project→Open Project”打开“…\DHT11\project\mdk5”目录下的工程“dht11.uvproj”。
4. 点击编译按钮编译工程。注意查看编译输出栏，观察编译的结果，如果有错误，修改程序，直到编译成功为止。编译后生成的 HEX 文件“nrf52840\_qiaa.hex”位于工程目录下的“Objects”文件夹中。
5. 点击下载按钮下载程序。如果需要对程序进行仿真，点击 Debug 按钮即可将程序下载

到 nRF52840 进行仿真。

6. 程序运行后，开发板每 2.5 秒读取一次 DHT11 的温湿度数据，每次读取时取反一次 D1 指示灯的状态指示程序的运行。读取的数据通过串口输出，电脑上打开串口调试助手(波特率设置为 115200bps)，即可观察到读取的温湿度数据，如下图所示。



The screenshot shows a terminal window with a list of 20 identical entries, each consisting of two lines of text. The first line is 'Temperature: 24.0 °C' and the second line is 'Humidity: 17.0 %RH'. This indicates that the device is consistently reading the same values from the DHT11 sensor.

```
Temperature: 24.0 °C Humidity: 17.0 %RH
```

图 25-11：读取的温湿度数据

## 参考文献

1. Cortex-M4 Technical Reference Manual, ARM
2. nRF5\_SDK\_16.0.0\_98a08e2, Nordic Semiconductor.
3. nRF5\_SDK\_16.0.0\_offline\_doc, Nordic Semiconductor。
4. nRF52840\_PS\_v1.0, Nordic Semiconductor。
5. nRF5x\_Command\_Line\_Tools\_v1.3, Nordic Semiconductor。
6. I2C-bus specification and user manual, NXP。
7. AT24C02 数据手册, Microchip。
8. dht11-v1\_3 说明书（详细版）, 奥松电子。
9. W25Q128FV 数据手册, WINBOND。