ING916XX 系列芯片外设开发者手册

Ingchips Technology Co., Ltd.

目录

表格

插图

第一章　版本历史

版本	信息	日期
0.1	初始版本	2022-xx-xx

第二章　概览

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

ING916XX 系列芯片支持蓝牙 5.3 规范，内置高性能 32bit RISC MCU（支持 DSP 和 FPU）、Flash、低功耗 PMU，以及丰富的外设、高性能低功耗 BLE RF 收发机。BLE 发射功率。

本文介绍 SoC 外设及其开发方法。每个章节介绍一种外设，各种外设与芯片数据手册之外设一一对应，基于 API 的兼容性、避免误解等因素，存在以下例外：

- PINCTRL 对应于数据手册之 IOMUX
- PCAP 对应于数据手册之 PCM
- SYSCTRL 是一个 “虚拟” 外设，负责管理各种 SoC 功能，组合了几种相关的硬件模块

SDK 外设驱动的源代码开放，其中包含很多常数，而且几乎没有注释 —— 这是有意为之，开发者只需要关注头文件，而不要尝试修改源代码。

2.1　缩略语及术语

表 2.1: 缩略语

缩略语	说明
ADC	模数转换器（Analog-to-Digital Converter）
DMA	直接存储器访问（Direct Memory Access）
EFUSE	电编程熔丝（Electronic Fuses）
FIFO	先进先出队列（First In First Out）
FOTA	固件空中升级（Firmware Over-The-Air）
GPIO	通用输入输出（General-Purpose Input/Output）
I2C	集成电路间总线（Inter-Integrated Circuit）

缩略语	说明
I2S	集成电路音频总线（Inter-IC Sound）
IR	红外线（Infrared）
PCAP	脉冲捕捉（Pulse CAPture）
PDM	脉冲密度调制（Pulse Density Modulation）
PTE	外设触发引擎（Peripheral Trigger Engine）
PWM	脉宽调制信号（Pulse Width Modulation）
QDEC	正交解码器（Quadrature Decoder）
RTC	实时时钟（Real-time Clock）
SPI	串行外设接口（Serial Peripheral Interface）
UART	通用异步收发器（Universal Asynchronous Receiver/Transmitter）
USB	通用串行总线（Universal Serial Bus）

2.2 参考文档

1. Bluetooth SIG[1]
2. ING916XX 系列芯片数据手册

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

第三章　ADC 简介

ADC 全称 Analog-to-Digital Converter，即模数转换器。

其主要作用是通过 PIN 测量电压，并将采集到的电压模拟信号转换成数字信号。

3.1　功能描述

3.1.1　特点

- 最多 12 个单端输入通道或 4 个差分输入通道
- 14 位分辨率
- 电压输入范围（0~3.6V）
- 支持 APB 总线
- 采样频率可编程
- 支持单一转换模式
- 支持回路转换模式，每个通道均可启用或禁用

3.1.2　ADC 模式

- 校准模式（calibration）：用于校准 ADC 采样精度。分为单端模式校准和差分模式校准，两种模式校准互相独立
- 转换模式（conversion）：用于正常工作状态下的模数转换

根据 ADC 输入模式完成对应的模式校准，之后在转换模式下进行正常模数转换。

3.1.3 ADC 输入模式

- 单端输入（single end）：使用单个输入引脚，采用 ADC 内部的参考电压
- 差分输入（differential）：使用一组输入引脚分别作为参考电压

一般来说，差分输入有利于避免共模干扰的影响，结果相对准确。

3.1.4 ADC 转换模式

- 单次转换（single）：单次转换在使能转换通道，采样和转换后，ADC 将停止，数据将被拉入 `FIFO`
- 连续转换（continuous）：连续转换在逐个使能通道，采样并转换后，经过 `loop-delay` 时间后循环进行操作，ADC 不会停止直到手动关闭

3.1.5 ADC 通道

ADC 共 12 个 channel，即 ch0-ch11。

其中 ch0-ch7 是通用通道，ch8-ch11 是内部通道，可以用来采集内部信号。

具体通道的输入连接引脚如下：

表 3.1: ADC 输入连接引脚

通道	连接引脚
ch0	GPIO7
ch1	GPIO8
ch2	GPIO9
ch3	GPIO10
ch4	GPIO11
ch5	GPIO12
ch6	GPIO13
ch7	GPIO14
ch8	diag_hv_soc
ch9	VREF12_ADC_IN
ch10	VCC

通道	连接引脚
ch11	V18_FLASH

ch0-ch7 可以配置成为 4 对差分输入通道，配置后将对应差分的通道 0-3。

注意：配置 ADC 输入模式为差分模式下则只有 ch0-ch3 以及 ch8-ch11。

3.1.6 PGA

ADC 的 PGA 默认是开启的，目前使用时不支持关闭。

ADC 的 PGA 有 8 个增益值，其大小为 `2^n(n=0-7)`，但是目前增益值 n=6,7 暂不支持配置。

对于 PGA 增益值配置，目前 ch0-ch7 可以配置的增益值为 `2^n(n=0-5)`

ch8-ch11 增益值暂不支持配置，其增益值固定为 2.

3.2 使用方法

3.2.1 输入电压范围

不同 ADC 输入模式下的输入电压要求如下：

- 单端模式：VIN [VREFP/2-VREFP/PGA_GAIN, VREFP/2+VREFP/PGA_GAIN]
- 差分模式：VINP-VINN [-VREFP/PGA_GAIN, VREFP/PGA_GAIN]，且 VINP，VINN>=0

其中 PGA_GAIN 是 PGA 的增益值

注意：请务必保证输入电压满足以上的范围要求，过大、过小的输入电压可能会使 ADC 不能正常工作甚至造成损坏芯片的严重后果。

3.2.2 采样率

采样率和时钟、loop-delay 大小有关，其计算关系如下：

SAMPLERATE=ADC_CLK/(loop-delay+16)

3.2.3 方法概述

方法概述为：时钟配置，ADC 校准，ADC 参数配置和中断（DMA 中断）注册。

3.2.3.1 时钟配置

当前 ADC 所用时钟源为 `clock slow` 经过分频得到的 ADC 工作时钟

当前 ADC 工作时钟可以配置为 1M、2M、4M 和 6M。高于 6M 的时钟暂不支持配置

注意：由于 ADC 工作时钟只可以取以上特定值，ADC 时钟必须经过 `clock slow` 分频。故在同时使用和 ADC 同时钟源模块时，如 IR，可能出现时钟配置冲突的现象。开发者在实际使用时需要格外注意，具体请参考 916 时钟树（图 3.1）。

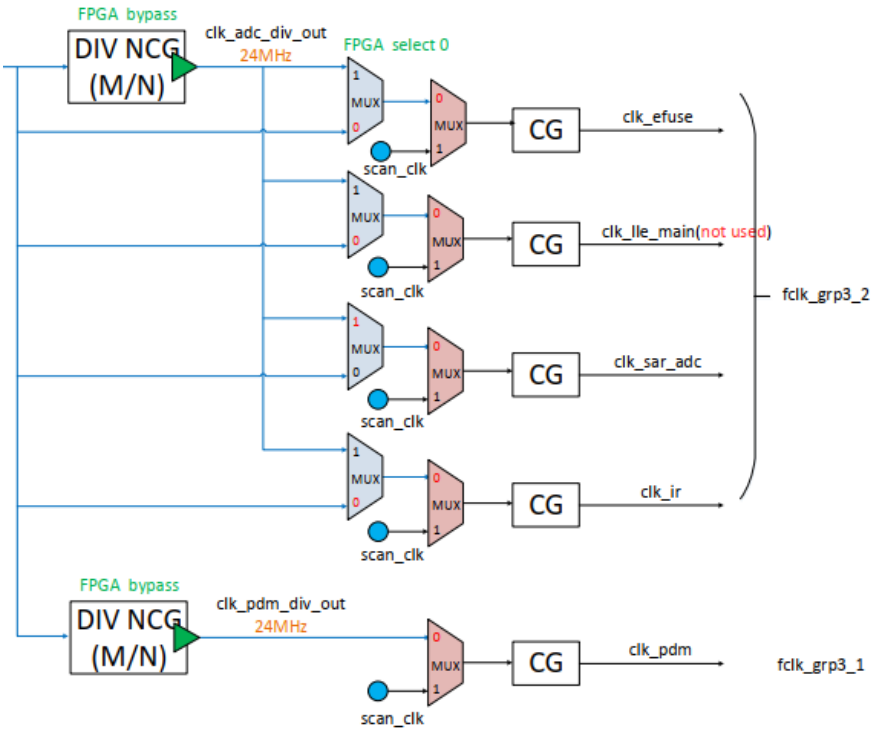


图 3.1: 916 时钟树 ADC 模块部分截图

3.2.3.2 ADC 校准

ADC 校准包含精度校准和内部参考电压校准

ADC 精度校准接口：ADC_Calibration

内部参考电压校准接口：ADC_VrefCalibration

在进行 ADC 转换之前需要进行 ADC 精度校准，如果需要计算采集电压则需要进行内部参考电压校准

ADC 精度校准需要明确 ADC 输入模式（单端/差分），两种模式需要分别进行精度校准

3.2.3.3 ADC 参数配置

ADC 参数配置接口：ADC_ConvCfg

涉及参数有：ADC 转换模式、ADC 输入模式、PGA 增益值、PGA 开关（目前不支持关闭）、采样通道、data 触发中断数、data 触发 DMA 搬运数和 `loop-delay`

data 触发中断数和 data 触发 DMA 搬运数决定了搬运 ADC 数据的方式。前者用触发中断的方式，后者用触发 DMA 搬运的方式

对于中断/DMA 搬运方式选择上，建议如下：

- 一般在小数据量情况下，如定时采集温度，建议采用触发中断并 CPU 读数的方式
- 一般大数据量采样，如模拟麦克风采样，建议采用 DMA 搬运方式或者乒乓搬运，可以大大提高数据搬运处理效率

3.2.3.4 ADC 数据处理

ADC 数据处理的推荐方法为：

1. 调用 ADC_PopFifoData（或 DMA 搬运 buff）读取 FIFO 中的 ADC 原始数据

2. 调用 ADC_GetDataChannel 得到原始数据中的数据所属通道（如需要）

3. 调用 ADC_GetData 得到原始数据中的 ADC 数据

4. 调用 ADC_GetVol 通过 ADC 数据计算得到其对应的电压值（如需要）

也可以通过调用 ADC_ReadChannelData 接口直接得到指定通道的 ADC 数据。但这样会丢弃其他通道数据，请谨慎使用。其可以作为辅助接口使用，非主要方式。

3.2.4 注意点

- 一般来说配置 ADC 时钟频率越高，其工作效率越好。较为推荐使用 4M、6M 工作时钟，性能稳定

- ADC 时钟选择需要根据实际设备采样率选择，如模拟 mic，应参考 mic 操作手册来选择对应 ADC 时钟频率
- 差分输入电压需要控制范围为 0~3.6V
- 如果需要计算采集电压建议周期性地进行参考电压校准
- 目前暂不支持用户自己输入参考电压值，只能用内部电压值作为参考电压
- PGA 增益值的选择需要结合实际设备，可以参考具体设备的使用手册或者测试电压范围计算得到。注意不能超过参考电压阈值
- data 触发中断数和 data 触发 DMA 搬运数应该一个为 0，一个非 0,。如果两值都非 0 则优先选择触发中断的方式
- 通过 ADC 数据计算得到的电压值会被限制在 0 到正参考电压之间

3.3 编程指南

3.3.1 驱动接口

此处列举几个较为常用的接口：

- ADC_ConvCfg：ADC 转换标准配置接口
- ADC_Calibration：ADC 校准标准配置接口
- ADC_VrefCalibration：内部参考电压校准接口
- ADC_Reset：ADC 复位接口
- ADC_Start：ADC 使能接口
- ADC_AdcClose：ADC 关闭接口
- ADC_GetFifoEmpty：读取 FIFO 是否为空接口
- ADC_PopFifoData：读取 FIFO 原始数据接口
- ADC_GetDataChannel：读取原始数据中通道号接口
- ADC_GetData：读取原始数据中 ADC 数据接口

- ADC_ReadChannelData：读取特定通道 ADC 数据接口
- ADC_GetVol：读取 ADC 数据对应电压值接口

还有部分接口不推荐直接使用，在此不进行介绍，详见对应驱动程序头文件声明。

3.3.2 代码示例

3.3.2.1 触发中断搬运

下面展示用 ADC 中断进行数据搬运的基本用法：

以下 ADC 设置参数仅供参考，具体参数请结合实际需要进行配置。

```
#define ADC_CHANNEL    ADC_CH_0
#define ADC_CLK_MHZ    6

static uint32_t ADC_cb_isr(void *user_data)
{
    uint32_t data = ADC_PopFifoData();
    SADC_channelId channel = ADC_GetDataChannel(data);
    if (channel == ADC_CHANNEL) {
        uint16_t sample = ADC_GetData(data);
        // do something with 'sample'
    }
    return 0;
}

void test(void)
{
    SYSCTRL_ClearClkGate(SYSCTRL_ITEM_APB_ADC);
    SYSCTRL_SetAdcClkDiv(24 / ADC_CLK_MHZ);
    SYSCTRL_ReleaseBlock(SYSCTRL_ITEM_APB_ADC);
    ADC_Calibration(SINGLE_END_MODE);
    ADC_ConvCfg(SINGLE_MODE, PGA_GAIN_4, 1, ADC_CHANNEL, 1, 0, SINGLE_END_MODE, 0);
    platform_set_irq_callback(PLATFORM_CB_IRQ_ADC, ADC_cb_isr, 0);
```

```
    ADC_Start(1);
}
```

以上代码展示了配置 ADC 时钟，逻辑复位、ADC 校准、ADC 转换配置，并在触发的 ADC 中断程序里获取到最终的 sample 数值。

当然，由于使用单一 ADC 通道，可以直接获取特定通道的 ADC 数据，代码如下：

```
static uint32_t ADC_cb_isr(void *user_data)
{
    uint16_t sample = ADC_ReadChannelData(ADC_CHANNEL);
    // do something with 'sample'
    return 0;
}
```

在只有单个通道数据时或者只需要单一通道数据时可以采用以上方式，多通道采样有丢数据的风险。

3.3.2.2 连续中断搬运

以上代码展示的是单次搬运情况，如果是多次搬运，建议采用以下两种读数方案：

1. 读数并结合调用 ADC_GetFifoEmpty 接口判断 FIFO 状态，读数直到 FIFO 为空为止

2. 每次读取的数据量等于配置的 data 触发中断数

方案 1 代码示例：

```
static uint32_t ADC_cb_isr(void *user_data)
{
    while (!ADC_GetFifoEmpty()) {
        uint16_t sample = ADC_ReadChannelData(ADC_CHANNEL);
        // do something with 'sample'
    }
    return 0;
}
```

方案 2 代码示例：

```
#define INT_TRIGGER_NUM    8
static uint32_t ADC_cb_isr(void *user_data)
{
    uint8_t i = 0;
    while (i < INT_TRIGGER_NUM) {
        uint16_t sample = ADC_ReadChannelData(ADC_CHANNEL);
        // do something with 'sample'
        i++;
    }
    return 0;
}
```

CPU 资源方面，方案 2 节省了每次读数调用接口的开销，建议优选

实际效果方面以上两种方案均可，开发者可以自行选用。

3.3.2.3 获取电压值

获取电压值需要在上面例子中加入参考电压校准，并调用接口计算电压值，如下：

```
#define ADC_CHANNEL    ADC_CH_0
#define ADC_CLK_MHZ    6

static uint32_t ADC_cb_isr(void *user_data)
{
    uint16_t sample = ADC_ReadChannelData(ADC_CHANNEL);
    float voltage = ADC_GetVol(sample);
    return 0;
}

void test(void)
{
```

```
    SYSCTRL_ClearClkGate(SYSCTRL_ITEM_APB_ADC);
    SYSCTRL_SetAdcClkDiv(24 / ADC_CLK_MHZ);
    SYSCTRL_ReleaseBlock(SYSCTRL_ITEM_APB_ADC);
    ADC_Calibration(SINGLE_END_MODE);
    ADC_VrefCalibration();      // calibrate the referenced voltage
    ADC_ConvCfg(SINGLE_MODE, PGA_GAIN_4, 1, ADC_CHANNEL, 1, 0, SINGLE_END_MODE, 0);
    platform_set_irq_callback(PLATFORM_CB_IRQ_ADC, ADC_cb_isr, 0);
    ADC_Start(1);
}
```

voltage 即为最终计算得到的采样电压值。

3.3.2.4 ADC & DMA 乒乓搬运

ADC 结合 DMA 乒乓搬运数据的方式是一种推荐的标准用法，其优点主要是节省 CPU 资源，提高处理数据的效率。

下面展示 DMA 乒乓搬运 ADC 数据并转换成电压的实例：

```
#include "pingpong.h"
#define ADC_CHANNEL    ADC_CH_0
#define DMA_CHANNEL    0
#define ADC_CLK_MHZ    6
static DMA_PingPong_t PingPong;

static uint32_t DMA_cb_isr(void *user_data)
{
    uint32_t state = DMA_GetChannelIntState(DMA_CHANNEL);
    DMA_ClearChannelIntState(DMA_CHANNEL, state);

    uint32_t *buff = DMA_PingPongIntProc(&PingPong, DMA_CHANNEL);
    uint32_t tranSize = DMA_PingPongGetTransSize(&PingPong);
    for (uint32_t i = 0; i < tranSize; ++i) {
        if (ADC_GetDataChannel(buff[i]) != ADC_CHANNEL) continue;
```

```
        uint16_t sample = ADC_GetData(buff[i]);
        float voltage = ADC_GetVol(sample);
    }
    return 0;
}
void test(void)
{
    SYSCTRL_ClearClkGateMulti((1 << SYSCTRL_ITEM_APB_DMA));

    SYSCTRL_ClearClkGate(SYSCTRL_ITEM_APB_ADC);
    SYSCTRL_SetAdcClkDiv(24 / ADC_CLK_MHZ);
    SYSCTRL_ReleaseBlock(SYSCTRL_ITEM_APB_ADC);
    ADC_Calibration(DIFFERENTAIL_MODE);
    ADC_VrefCalibration();
    ADC_ConvCfg(CONTINUES_MODE, PGA_GAIN_4, 1, ADC_CHANNEL, 0, 8, DIFFERENTAIL_MODE,

    SYSCTRL_SelectUsedDmaItems(1 << 9);
    DMA_PingPongSetup(&PingPong, SYSCTRL_DMA_ADC, 80, 8);
    platform_set_irq_callback(PLATFORM_CB_IRQ_DMA, DMA_cb_isr, 0);
    DMA_PingPongEnable(&PingPong, DMA_CHANNEL);

    ADC_Start(1);
}
```

第四章　DMA 简介

DMA 全称 direct memory access，即直接存储器访问。

其主要作用是不占用 CPU 大量资源，在 AMBA AHB 总线上的设备之间以硬件方式高速有效地传输数据。

4.1　功能描述

4.1.1　特点

- 符合 AMBA、AHB 和 APB4 标准
- 最多 8 个 DMA 通道
- 最多 16 个硬件握手请求/确认配对
- 支持 8/16/32/64 位宽的数据传输
- 支持 24-64 位地址宽度
- 支持成链传输数据

4.1.2　搬运方式

- 单次数据块搬运：DMA 使用单个通道，一次使能将数据从 SRC 到 DST 位置搬运一次
- 成串多数据块搬运：DMA 使用单个通道，一次使能按照 DMA 链表信息依次将数据从 SRC 到 DST 位置搬运多次或循环搬运。

其根本区别是有无注册有效的 DMA 链表。

4.1.3 搬运类型

- memory 到 memory 搬运
- memory 到 peripheral 搬运
- peripheral 到 memory 搬运
- peripheral 到 peripheral 搬运

4.1.4 中断类型

- IntErr：错误中断表示 DMA 传输发生了错误而触发中断，主要包括总线错误、地址没对齐和传输数据宽度没对齐等。
- IntAbt：终止传输中断会在终止 DMA 通道传输时产生。
- IntTC：TC 中断会在没有产生 IntErr 和 IntAbt 的情况下完成一次传输时产生。

4.1.5 数据地址类型

- Increment address
- Decrement address
- Fixed address

如果 Increment 则 DMA 从地址由小到大搬运数据，相反的 Decrement 则由大到小搬运。fixed 地址适用于外设 FIFO 的寄存器搬运数据。

4.1.6 数据方式

- normal mode
- handshake mode

一般外设寄存器选择用握手方式。

选择握手方式要和外设协商好 SrcBurstSize，当前支持 2^n(n = 0-7) 大小的 SrcBurstSize。

4.1.7 数据位宽

DMA 传输要求传输两端的数据类型一致，支持数据类型有：

- Byte transfer
- Half-word transfer
- Word transfer
- Double word transfer

覆盖所有常见数据类型。

4.2 使用方法

4.2.1 方法概述

首先确认数据搬运需求是单次搬运还是成串搬运，搬运类型 memory 和 peripheral 的关系。

4.2.1.1 单次搬运

1. 注册 DMA 中断

2. 定义一个 DMA_Descriptor 变量用来配置 DMA 通道寄存器

3. 根据数据搬运类型选择 DMA 寄存器配置接口，正确调用驱动接口配置 DMA 寄存器

4. 使能 DMA 通道开始搬运

4.2.1.2 成串搬运

1. 注册一个或多个 DMA 中断

2. 定义多个 DMA_Descriptor 变量用来配置 DMA 通道寄存器

3. 根据数据搬运类型选择 DMA 寄存器配置接口，正确调用驱动接口配置 DMA 寄存器

4. 将多个 DMA_Descriptor 变量首尾相连成串，类似链表

5. 使能 DMA 通道开始搬运

4.2.2 注意点

- 定义 DMA_Descriptor 变量需要 8 字节对齐，否则 DMA 搬运不成功
- 成串搬运如果配置多个 DMA 中断则需要在每个中断里使能 DMA，直到最后一次搬运完成
- 对于从外设搬运需要确认外设是否支持 DMA
- 建议从外设搬运选择握手方式，并与外设正确协商 burstSize
- burstSize 尽量取较大值，有利于减少 DMA 中断次数提高单次中断处理效率。但 burstSize 太大可能最后一次不能搬运丢弃较多数据
- 建议设置从外设搬运总数据量为 burstSize 的整数倍或采用乒乓搬运的方式
- 在 DMA 从外设搬运的情况下，正确的操作顺序是先配置并使能好 DMA，再使能外设开始产生数据

4.3 编程指南

4.3.1 驱动接口

- DMA_PrepareMem2Mem：memory 到 memory 搬运标准 DMA 寄存器配置接口
- DMA_PreparePeripheral2Mem：Peripheral 到 memory 搬运标准 DMA 寄存器配置接口
- DMA_PrepareMem2Peripheral：memory 到 Peripheral 搬运标准 DMA 寄存器配置接口
- DMA_PreparePeripheral2Peripheral：Peripheral 到 Peripheral 搬运标准 DMA 寄存器配置接口
- DMA_Reset：DMA 复位接口
- DMA_GetChannelIntState：DMA 通道中断状态获取接口
- DMA_ClearChannelIntState：DMA 通道清中断接口
- DMA_EnableChannel：DMA 通道使能接口
- DMA_AbortChannel：DMA 通道终止接口

4.3.2 代码示例

4.3.2.1 单次搬运

下面以 memory 到 memory 单次搬运展示 DMA 的基本用法：

```
#define CHANNEL_ID  0
char src[] = "hello world!";
char dst[20];
DMA_Descriptor test __attribute__((aligned (8)));
static uint32_t DMA_cb_isr(void *user_data)
{
    uint32_t state = DMA_GetChannelIntState(CHANNEL_ID);
    DMA_ClearChannelIntState(CHANNEL_ID, state);

    printf("dst = %s\n", dst);
    return 0;
}

void DMA_Test(void)
{
    DMA_Reset();
    platform_set_irq_callback(PLATFORM_CB_IRQ_DMA, DMA_cb_isr, 0);
    DMA_PrepareMem2Mem(&test[0],
                       dst,
                       src, strlen(src),
                       DMA_ADDRESS_INC, DMA_ADDRESS_INC, 0);
    DMA_EnableChannel(CHANNEL_ID, &test);
}
```

最终会在 DMA 中断程序里面将搬运到 dst 中的 “hello world!” 字符串打印出来。

4.3.2.2 成串搬运

下面以 memory 到 memory 两块数据搬运拼接字符串展示 DMA 成串搬运的基本用法：

```c
#define CHANNEL_ID  0
char src[] = "hello world!";
char src1[] = "I am a chinese.";
char dst[100];
DMA_Descriptor test[2] __attribute__((aligned (8)));
static uint32_t DMA_cb_isr(void *user_data)
{
    uint32_t state = DMA_GetChannelIntState(CHANNEL_ID);
    DMA_ClearChannelIntState(CHANNEL_ID, state);

    printf("dst = %s\n", dst);
    return 0;
}

void DMA_Test(void)
{
    DMA_Reset();
    platform_set_irq_callback(PLATFORM_CB_IRQ_DMA, DMA_cb_isr, 0);
      test[0].Next = &test[1];    // make a DMA link chain
      test[1].Next = NULL;
    DMA_PrepareMem2Mem(&test[0],
                       dst,
                       src, strlen(src),
                       DMA_ADDRESS_INC, DMA_ADDRESS_INC, 0);
    DMA_PrepareMem2Mem(&test[1],
                       dst + strlen(src),
                       src1, sizeof(src1),
                       DMA_ADDRESS_INC, DMA_ADDRESS_INC, 0);
    DMA_EnableChannel(CHANNEL_ID, &test[0]);
}
```

最终将会打印出 “hello world!I am a chinese.” 字符串。

4.3.2.3 DMA 乒乓搬运

DMA 乒乓搬运是一种 DMA 搬运的特殊用法，其主要应用场景是将外设 FIFO 中数据循环搬运到 memory 中并处理。

可实现 “搬运” 和 “数据处理” 分离，从而大大提高程序处理数据的效率。

对于大量且连续的数据搬运，如音频，我们推荐选用 DMA 乒乓搬运的方式。

4.3.2.3.1 DMA 乒乓搬运接口 在最新 SDK 中我们已将 DMA 乒乓搬运封装成标准接口，方便开发者调用，提高开发效率。

使用时请添加 `pingpong.c` 文件，并包含 `pingpong.h` 文件。

- DMA_PingPongSetup：DMA 乒乓搬运建立接口
- DMA_PingPongIntProc：DMA 乒乓搬运标准中断处理接口
- DMA_PingPongGetTransSize：获取 DMA 乒乓搬运数据量接口
- DMA_PingPongEnable：DMA 乒乓搬运使能接口
- DMA_PingPongDisable：DMA 乒乓搬运去使能接口

4.3.2.3.2 DMA 乒乓搬运示例 下面将以最常见的 DMA 乒乓搬运 I2s 数据为例展示 DMA 乒乓搬运的用法。

I2s 的相关配置不在本文的介绍范围内，默认 I2s 已经配置好，DMA 和 I2s 协商 burstSize=8。

```
#include "pingpong.h"
#define CHANNEL_ID  0
DMA_PingPong_t PingPong;

static uint32_t DMA_cb_isr(void *user_data)
{
    uint32_t state = DMA_GetChannelIntState(CHANNEL_ID);
    DMA_ClearChannelIntState(CHANNEL_ID, state);

    // call 'DMA_PingPongIntProc' to get the pointer of data-buff.
```

```
    uint32_t *rr = DMA_PingPongIntProc(&PingPong, CHANNEL_ID);
    uint32_t i = 0;
    // call 'DMA_PingPongGetTransSize' to kwon how much data in data-buff.
    uint32_t transSize = DMA_PingPongGetTransSize(&PingPong);
    while (i < transSize) {
        // do something with data 'rr[i]'
        i++;
    }

    return 0;
}

void DMA_Test(void)
{
    // call 'DMA_PingPongSetup' to setup ping-pong DMA.
    DMA_PingPongSetup(&PingPong, SYSCTRL_DMA_I2S_RX, 100, 8);
    platform_set_irq_callback(PLATFORM_CB_IRQ_DMA, DMA_cb_isr, 0);

    // call 'DMA_PingPongEnable' to start ping-pong DMA transmission.
    DMA_PingPongEnable(&PingPong, CHANNEL_ID);
    I2S_ClearRxFIFO(APB_I2S);
    I2S_DMAEnable(APB_I2S, 1, 1);
    I2S_Enable(APB_I2S, 0, 1);   // Enable I2s finally
}
```

停止 DMA 乒乓搬运可以调用以下接口：

```
void Stop(void)
{
    // call 'DMA_PingPongEnable' to disable ping-pong DMA transmission.
    DMA_PingPongDisable(&PingPong, CHANNEL_ID);
    I2S_Enable(APB_I2S, 0, 0);
    I2S_DMAEnable(APB_I2S, 0, 0);
}
```

第五章　eFuses(electronic fuses)

5.1　功能概述

Efuse 是一种片内一次性可编程存储器，可以在断电后保持数据，并且编程后无法被再次修改。ING916 系列提供 128bit EFUSE，支持按 bit 编程或者按 Word 编程，bit 的默认值是 0，通过编程可以写成 1

5.2　使用说明

5.2.1　模块初始化

```
void setup_peripherals_efuse_module(void)
{
    // 打开 clock，并且 reset 模块
    SYSCTRL_ClearClkGateMulti( (1 << SYSCTRL_ITEM_APB_EFUSE));
    SYSCTRL_ResetBlock(SYSCTRL_ITEM_APB_EFUSE);
    SYSCTRL_ReleaseBlock(SYSCTRL_ITEM_APB_EFUSE);
}
```

5.2.2　按 bit 编程

```
void peripherals_efuse_write_bit(void)
{
    // 提供需要编程的 bit 位置,0 到 127 之间
    EFUSE_UnLock(APB_EFUSE, EFUSE_UNLOCK_FLAG);
    EFUSE_WriteEfuseDataBitToOne(APB_EFUSE, pos);//pos is bit position from 0 to 127

    //写操作完成
    //如果要读取，需要 reset 模块
    SYSCTRL_ResetBlock(SYSCTRL_ITEM_APB_EFUSE);
    SYSCTRL_ReleaseBlock(SYSCTRL_ITEM_APB_EFUSE);

    //设置读取 flag
    EFUSE_SetRdFlag(APB_EFUSE);
    //等待数据读取标志
    while(!EFUSE_GetDataValidFlag(APB_EFUSE));
    //读取数据
    platform_printf("word 0: 0x%x \n",EFUSE_GetEfuseData(APB_EFUSE, 0));//bit 0  - 31
    platform_printf("word 1: 0x%x \n",EFUSE_GetEfuseData(APB_EFUSE, 1));//bit 32 - 63
    platform_printf("word 2: 0x%x \n",EFUSE_GetEfuseData(APB_EFUSE, 2));//bit 64 - 95
    platform_printf("word 3: 0x%x \n",EFUSE_GetEfuseData(APB_EFUSE, 3));//bit 96 - 127
}
```

5.2.3 按 word 编程

```
void peripherals_efuse_write_word(void)
{
    // 提供需要编程的 word 位置,0 到 3 之间, 每个 word 32bit, 一共 128bit
    // EFUSE_PROGRAMWORDCNT_0 代表 word 0
    // data 是要写入的 32bit 数据
    EFUSE_WriteEfuseDataWord(APB_EFUSE, EFUSE_PROGRAMWORDCNT_0, data);

    //写操作完成
```

```
    //如果要读取，需要 reset 模块
    SYSCTRL_ResetBlock(SYSCTRL_ITEM_APB_EFUSE);
    SYSCTRL_ReleaseBlock(SYSCTRL_ITEM_APB_EFUSE);

    //设置读取 flag
    EFUSE_SetRdFlag(APB_EFUSE);
    //等待数据读取标志
    while(!EFUSE_GetDataValidFlag(APB_EFUSE));
    //读取数据
    platform_printf("word 0: 0x%x \n",EFUSE_GetEfuseData(APB_EFUSE, 0));//bit 0  - 31
    platform_printf("word 1: 0x%x \n",EFUSE_GetEfuseData(APB_EFUSE, 1));//bit 32 - 63
    platform_printf("word 2: 0x%x \n",EFUSE_GetEfuseData(APB_EFUSE, 2));//bit 64 - 95
    platform_printf("word 3: 0x%x \n",EFUSE_GetEfuseData(APB_EFUSE, 3));//bit 96 - 12
}
```

第六章　通用输入输出（GPIO）

6.1　功能概述

GPIO 模块常用于驱动 LED 或者其它指示器，控制片外设备，感知数字信号输入，检测信号边沿，或者从低功耗状态唤醒系统。ING916XX 系列芯片内部支持最多 42 个 GPIO，通过PINCTRL 可将 GPIO n 引出到芯片 IO 管脚 n。

特性：

- 每个 GPIO 都可单独配置为输入或输出
- 每个 GPIO 都可作为中断请求，中断触发方式支持边沿触发（上升、下降单沿触发，或者双沿触发）和电平触发（高电平或低电平）
- 硬件去抖

在硬件上存在 **两个** GPIO 模块，每个模块包含 21 个 GPIO，相应地定义了两个 `SYSCTRL_Item`：

```
typedef enum
{
    SYSCTRL_ITEM_APB_GPIO0     ,
    SYSCTRL_ITEM_APB_GPIO1     ,
    // ...
} SYSCTRL_Item;
```

注意按照所使用的 GPIO 管脚打开对应的 GPIO 模块。

6.2 使用说明

6.2.1 设置 IO 方向

在使用 GPIO 之前先按需要配置 IO 方向：

- 需要用于输出信号时：配置为输出
- 需要用于读取信号时：配置为输入
- 需要用于生产中断请求时：配置为输入
- 需要高阻态时：配置为高阻态

使用 `GIO_SetDirection` 配置 GPIO 的方向。GPIO 支持四种方向：

```
typedef enum
{
    GIO_DIR_INPUT,  // 输入
    GIO_DIR_OUTPUT, // 输出
    GIO_DIR_BOTH,   // 同时支持输入、输出
    GIO_DIR_NONE    // 高阻态
} GIO_Direction_t;
```

如无必要，不要使用 `GIO_DIR_BOTH`。

6.2.2 读取输入

使用 `GIO_ReadValue` 读取某个 GPIO 当前输入的电平信号，例如读取 GPIO 0 的输入：

```
uint8_t value = GIO_ReadValue(GIO_GPIO_0);
```

使用 `GIO_ReadAll` 可以同时读取所有 GPIO 当前输入的电平信号。其返回值的第 n 比特（第 0 比特为最低比特）对应 GPIO n 的输入；如果 GPIO n 当前不支持输入，那么第 n 比特为 0：

```
uint64_t GIO_ReadAll(void);
```

6.2.3 设置输出

- 设置单个输出

  使用 GIO_WriteValue 设置某个 GPIO 输出的电平信号，例如使 GPIO 0 输出高电平（1）：

```
GIO_WriteValue(GIO_GPIO_0, 1);
```

- 同时设置所有输出

  通过 GIO_WriteAll 可同时设置所有 GPIO 输出的电平信号：

```
void GIO_WriteAll(const uint64_t value);
```

- 将若干输出置为高电平

  通过 GIO_SetBits 可同时将若干 GPIO 输出置为高电平：

```
void GIO_SetBits(const uint64_t index_mask);
```

  比如要将 GPIO 0、5 置为高电平，那么 index_mask 为 (1 << 0) | (1 << 5)。

- 将若干输出置为低电平

  通过 GIO_ClearBits 可同时将若干 GPIO 输出置为低电平：

```
void GIO_ClearBits(const uint64_t index_mask);
```

  index_mask 的使用与 GIO_SetBits 相同。

6.2.4 配置中断请求

使用 GIO_ConfigIntSource 配置 GPIO 生成中断请求。

```
void GIO_ConfigIntSource(
  const GIO_Index_t io_index,      // GPIO 编号
  const uint8_t enable,            // 使能的边沿或者电平类型组合
  const GIO_IntTriggerType_t type // 触发类型
  );
```

其中的 enable 为以下两个值的组合（0 表示禁止产生中断请求）：

```
typedef enum
{
    ...LOGIC_LOW_OR_FALLING_EDGE = ..., // 低电平或者下降沿
    ...LOGIC_HIGH_OR_RISING_EDGE = ...  // 高电平或者上升沿
} GIO_IntTriggerEnable_t;
```

触发类型有两种：

```
typedef enum
{
    GIO_INT_EDGE,    // 边沿触发
    GIO_INT_LOGIC    // 电平触发
} GIO_IntTriggerType_t;
```

- 例如将 GPIO 0 配置为上升沿触发中断

```
GIO_ConfigIntSource(GIO_GPIO_0,
  ...LOGIC_HIGH_OR_RISING_EDGE,
  GIO_INT_EDGE);
```

- 例如将 GPIO 0 配置为双沿触发中断

```
GIO_ConfigIntSource(GIO_GPIO_0,
  ...LOGIC_HIGH_OR_RISING_EDGE | ..._HIGH_OR_RISING_EDGE,
  GIO_INT_EDGE);
```

- 例如将 GPIO 0 配置为高电平触发

```
GIO_ConfigIntSource(GIO_GPIO_0,
  ...LOGIC_HIGH_OR_RISING_EDGE,
  GIO_INT_LOGIC);
```

6.2.5 处理中断状态

用 `GIO_GetIntStatus` 获取某个 GPIO 上的中断触发状态，返回非 0 值表示该 GPIO 上产生了中断请求；用 `GIO_GetAllIntStatus` 一次性获取所有 GPIO 的中断触发状态，第 n 比特（第 0 比特为最低比特）对应 GPIO n 上的中断触发状态。

GPIO 产生中断后，需要消除中断状态方可再次触发。用 `GIO_ClearIntStatus` 消除某个 GPIO 上中断状态，用 `GIO_ClearAllIntStatus` 一次性清除所有 GPIO 上可能存在的中断触发状态。

6.2.6 输入去抖

使用 `GIO_DebounceCtrl` 配置输入去抖参数，每个 GPIO 硬件模块使用单独的参数：

```
void GIO_DebounceCtrl(
  uint8_t group_mask,     // 比特 0 为 1 时配置模块 0
                          // 比特 1 为 1 时配置模块 1
  uint8_t clk_pre_scale,
  GIO_DbClk_t clk         // 防抖时钟选择
  );
```

所谓去抖就是过滤掉长度小于 `(clk_pre_scale + 1)` 个防抖时钟周期的“毛刺”。

防抖时钟共有 2 种：

```
typedef enum
{
    GIO_DB_CLK_32K,      // 使用 32k 时钟
    GIO_DB_CLK_PCLK,     // 使用快速 PCLK
} GIO_DbClk_t;
```

快速 PCLK 的具体频率参考SYSCTRL。

通过 `GIO_DebounceEn` 为单个 GPIO 使能去抖。例如要在 GPIO 0 上启用硬件去抖，忽略宽度小于 $5/32768 \approx 0.15(ms)$ 的“毛刺”：

```
GIO_DebounceCtrl(1, 4, GIO_DB_CLK_32K);
GIO_DebounceEn(GIO_GPIO_0, 1);
```

6.2.7 低功耗保持状态

所有 GPIO 可以在芯片进入低功耗状态后保持状态。根据功能的不同，存在两种类型的 GPIO，表 6.1 列举了所有的 A 型 GPIO，其它 GPIO 为 B 型。

表 6.1: A 型 GPIO

GPIO
0
5
6
21
22
23
36
37

- 对于 A 型 GPIO

使用 GIO_EnableRetentionGroupA 使能或禁用 A 型 GPIO 的低功耗状态保持功能。使能状态保持功能时，IOMUX 与之相关的所有配置都被锁存，即使处于各种低功耗状态下。使能后，对这些 GPIO 的配置再做修改无法生效。只有禁用保持功能后，才会生效。使能后，低功耗状态下这些 GPIO 不掉电。

```
void GIO_EnableRetentionGroupA(uint8_t enable);
```

- 对于 B 型 GPIO

  使用 GIO_EnableRetentionGroupB 使能或禁用 B 型 GPIO 的低功耗状态保持功能。使能状态保持功能时，与之相关的配置（以及输出值 —— 对于 IO 方向为输出的 GPIO）都被锁存，即使处于各种低功耗状态下。使能后，对这些 GPIO 的配置再做修改无法生效。只有禁用保持功能后，才会生效。

```
void GIO_EnableRetentionGroupB(uint8_t enable);
```

  使用 GIO_EnableHighZGroupB 使能或禁用 B 型 GPIO 的低功耗高阻功能。使能该功能后，IO 方向为输出的 B 型 GPIO 处于高阻状态，对这些 GPIO 的配置再做修改无法生效。只有禁用保持功能后，才会生效。

```
void GIO_EnableHighZGroupB(uint8_t enable);
```

这些功能只支持对所有 GPIO 同时使能或禁用，不能对单个 GPIO 分别控制。

6.2.8 睡眠唤醒源

一部分 GPIO 支持作为低功耗状态的唤醒源：出现指定的电平信号时，将系统从低功耗状态下唤醒。对于深度睡眠（DEEP Sleep），这些 GPIO （$\{0..17, 21..25, 29..37\}$）可作为唤醒源，其中包含所有的 A 型 GPIO 和部分 B 型 GPIO。对于更深度的睡眠（DEEPER Sleep），所有的 A 型 GPIO 可作为唤醒源。

- 深度睡眠唤醒源

  使用 GIO_EnableDeepSleepWakeupSource 使能（或停用）某个 GPIO 的唤醒功能。其中，io_index 应该为支持该功能的 GPIO 的编号；level 为触发电平，1 为高电平唤醒，0 为低电平唤醒；触发电平与上下拉应该相互配合：高电平唤醒时，使用下拉；低电平唤醒时，使用上拉。对于 A 型 GPIO，忽略 pull 参数，其上下拉由 PINCTRL_Pull 控制。

  对于 B 型 GPIO，这里的上下拉配置仅用于低功耗状态。对于这种 GPIO，建议正常状态时的上下拉与这里的配置保持一致，以免由于配置切换产生“毛刺”。

```
int GIO_EnableDeepSleepWakeupSource(
  GIO_Index_t io_index,      // GPIO 编号
  uint8_t enable,            // 使能 (1)/禁用 (0)
  uint8_t level,             // 触发唤醒的电平
  pinctrl_pull_mode_t pull   // 上下拉配置
);
```

  任意一个唤醒源检测到唤醒电平就会将系统从低功耗状态唤醒。

- 更深度睡眠唤醒源

  使用 GIO_EnableDeeperSleepWakeupSourceGroupA 使能（或停用）A 型 GPIO 的更深度睡眠唤醒功能。其中，level 为触发电平，1 为高电平唤醒，0 为低电平唤醒使能后，所有 IO 方向为输入的 A 型 GPIO 都将作为唤醒源。任意一个唤醒源检测到唤醒电平就会将系统从低功耗状态唤醒。

```
void GIO_EnableDeeperSleepWakeupSourceGroupA(
  uint8_t enable,            // 使能 (1)/禁用 (0)
  uint8_t level              // 触发唤醒的电平
);
```

第七章　I2C 功能概述

- 两个 I2C 模块
- 支持 Master/Slave 模式
- 支持 7bit/10bit 地址
- 支持速率调整
- 支持 DMA

7.1　I2C 使用说明

针对 916 系列，IIC master 有两种使用方式可以选择：

- 方法 1：以 blocking 的方式操作 I2C（读写操作完成后 API 才会返回），针对 I2C Master 读取外设的场景。
- 方法 2：使用 I2C 中断操作 I2C，需要在中断中操作读写的数据。

IIC Slave 则需要使用方法 2，以中断方式操作。

7.1.1　方法 1（blocking）

- 参考：`\ING_SDK\sdk\src\BSP\iic.c`
- 包含 API：

```
/**
 * @brief Init an I2C peripheral
 *
 * @param[in] port              I2C peripheral ID
 */
void i2c_init(const i2c_port_t port);

/**
 * @brief Write data to an I2C slave
 *
 * @param[in] port              I2C peripheral ID
 * @param[in] addr              address of the slave
 * @param[in] byte_data         data to be written
 * @param[in] length            data length
 * @return                      0 if success else non-0 (e.g. time out)
 */
int i2c_write(const i2c_port_t port, uint8_t addr, const uint8_t *byte_data, int16_t leng

/**
 * @brief Read data from an I2C slave
 *
 * @param[in] port              I2C peripheral ID
 * @param[in] addr              address of the slave
 * @param[in] write_data        data to be written before reading
 * @param[in] write_len         data length to be written before reading
 * @param[in] byte_data         data to be read
 * @param[in] length            data length to be read
 * @return                      0 if success else non-0 (e.g. time out)
 */
int i2c_read(const i2c_port_t port, uint8_t addr, const uint8_t *write_data, int16_t writ
             uint8_t *byte_data, int16_t length);
```

- 使用方法：
  - 初始化 IIC 模块

```
setup_peripherals_i2c_pin();
i2c_init(I2C_PORT_0);
```

setup_peripherals_i2c_pin() 的使用请参考该文档中配置 pin 的描述 (不需要配置 IIC 中断)。

– 写数据

```
i2c_write(I2C_PORT_0, ADDRESS, write_data, DATA_CNT);
```

当读操作完成后 API 才会返回，为了避免长时间等待 ACK 等意外情况，使用 I2C_HW_TIME_OUT 来控制 blocking 的时间。

– 读数据

```
i2c_read(I2C_PORT_0, ADDRESS, write_data, DATA_CNT, read_data, DATA_CNT);
```

如果 write_data 不为空，则会首先执行写操作，然后再执行读操作。

7.1.2 方法 2（Interrupt）

IIC slave 以及 IIC Master 方法 2 的使用请参考场景 5。

以下场景中均以 I2C0 为例，如果需要 I2C1 则可以根据情况修改

7.2 场景 1：Master 读 Slave，不使用 DMA

其中 I2C 配置为 Master 读操作，Slave 收到地址后，将数据返回给 Master，CPU 操作读写，没有使用 DMA。配置之前需要决定使用的 GPIO，请参考 datasheet 了解可以使用的 GPIO

```
#define I2C_SCL          GIO_GPIO_10
#define I2C_SDA          GIO_GPIO_11
```

7.2.1 I2C Master 配置

// 测试数据，每次传输 10 个字节（fifo 深度是 8 字节），每个传输单元必须是 1 字节

```
#define DATA_CNT (10)
uint8_t read_data[DATA_CNT] = {0,};
uint8_t read_data_cnt = 0;
```

7.2.1.1 配置 Pin

将 GPIO 映射成 I2C 引脚

```
void setup_peripherals_i2c_pin(void)
{
  // 打开 clock，注意此处使用的是 I2C0
  // GPIO 的 clock 请根据需要打开 GPIO0 或者 GPIO1
  SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_I2C0)
                              | (1 << SYSCTRL_ITEM_APB_SysCtrl)
                              | (1 << SYSCTRL_ITEM_APB_PinCtrl)
                              | (1 << SYSCTRL_ITEM_APB_GPIO1)
                              | (1 << SYSCTRL_ITEM_APB_GPIO0));
  // 使用默认内部上拉，实际使用中建议使用外部上拉 - 响应速度更快，可以支持更高的时钟
  // 如果使用外部上拉，则不需要 pull
  PINCTRL_Pull(I2C_SCL,PINCTRL_PULL_UP);
  PINCTRL_Pull(I2C_SDA,PINCTRL_PULL_UP);

  // 将 GPIO 映射成 I2C 引脚
  PINCTRL_SelI2cIn(I2C_PORT_0,I2C_SCL,I2C_SDA);
  PINCTRL_SetPadMux(I2C_SCL,IO_SOURCE_I2C0_SCL_OUT);
```

```
  PINCTRL_SetPadMux(I2C_SDA,IO_SOURCE_I2C0_SDA_OUT);

  // 打开 I2C 中断
  platform_set_irq_callback(PLATFORM_CB_IRQ_I2C0, peripherals_i2c_isr, NULL);

}
```

7.2.1.2 初始化 I2C 模块

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
  // 配置为 Master, 7bit 地址
  I2C_Config(APB_I2C0,I2C_ROLE_MASTER,I2C_ADDRESSING_MODE_07BIT,ADDRESS);
  // 配置时钟，可选
  I2C_ConfigClkFrequency(APB_I2C0,I2C_CLOCKFREQUENY_STANDARD);

  I2C_Enable(APB_I2C0,1);
  // 打开传输结束中断和 fifo 满中断
  I2C_IntEnable(APB_I2C0,(1<<I2C_INT_CMPL)|(1 << I2C_INT_FIFO_FULL));

}
```

7.2.1.3 I2C 中断实现

```
static uint32_t peripherals_i2c_isr(void *user_data)
{
  uint8_t i;
  uint32_t status = I2C_GetIntState(APB_I2C0);
```

```
  // FIFO 满之后，触发中断，此时需要将所有数据都读出来
  if(status & (1 << I2C_STATUS_FIFO_FULL))
  {
    for(; read_data_cnt < DATA_CNT; read_data_cnt++)
    {
      if(I2C_FifoEmpty(APB_I2C0)){break;}
      read_data[read_data_cnt] = I2C_DataRead(APB_I2C0);
    }
  }

  //传输结束后，触发中断，读取完剩下的数据
  if(status & (1 << I2C_STATUS_CMPL))
  {
    for(; read_data_cnt < DATA_CNT; read_data_cnt++)
    {
      read_data[read_data_cnt] = I2C_DataRead(APB_I2C0);
    }

    I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_CMPL));

  }

  return 0;
}
```

7.2.1.4 I2C master 触发传输

```
void peripheral_i2c_send_data(void)
{
  // 设置方向，Master 读取
  I2C_CtrlUpdateDirection(APB_I2C0,I2C_TRANSACTION_SLAVE2MASTER);
  // 设置每次传输的大小
```

```
  I2C_CtrlUpdateDataCnt(APB_I2C0, DATA_CNT);
  // 触发传输
  I2C_CommandWrite(APB_I2C0, I2C_COMMAND_ISSUE_DATA_TRANSACTION);

}
```

7.2.1.5 使用流程

- 设置 GPIO，setup_peripherals_i2c_pin()
- 初始化 I2C，setup_peripherals_i2c_module()
- 在需要时候触发 I2C 读取，peripheral_i2c_send_data()
- 检查中断状态

7.2.2 I2C Slave 配置

// 测试数据，每次传输 10 个字节（fifo 深度是 8 字节），每个传输单元必须是 1 字节

```
#define DATA_CNT (10)
uint8_t write_data[DATA_CNT] = {0,};
uint8_t write_data_cnt = 0;
```

7.2.2.1 配置 Pin

将 GPIO 映射成 I2C 引脚

```
void setup_peripherals_i2c_pin(void)
{
  // 打开 clock，注意此处使用的是 I2C0
  // GPIO 的 clock 请根据需要打开 GPIO0 或者 GPIO1
  SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_I2C0)
                              | (1 << SYSCTRL_ITEM_APB_SysCtrl)
                              | (1 << SYSCTRL_ITEM_APB_PinCtrl)
```

```
                                    | (1 << SYSCTRL_ITEM_APB_GPIO1)
                                    | (1 << SYSCTRL_ITEM_APB_GPIO0));
  // 使用默认内部上拉，实际使用中建议使用外部上拉 - 响应速度更快，可以支持更高的时钟
  // 如果使用外部上拉，则不需要 pull
  PINCTRL_Pull(I2C_SCL,PINCTRL_PULL_UP);
  PINCTRL_Pull(I2C_SDA,PINCTRL_PULL_UP);

  // 将 GPIO 映射成 I2C 引脚
  PINCTRL_SelI2cIn(I2C_PORT_0,I2C_SCL,I2C_SDA);
  PINCTRL_SetPadMux(I2C_SCL,IO_SOURCE_I2C0_SCL_OUT);
  PINCTRL_SetPadMux(I2C_SDA,IO_SOURCE_I2C0_SDA_OUT);

  // 打开 I2C 中断
  platform_set_irq_callback(PLATFORM_CB_IRQ_I2C0, peripherals_i2c_isr, NULL);

}
```

7.2.2.2 初始化 I2C 模块

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
  // 配置为 Slave, 7bit 地址
  I2C_Config(APB_I2C0,I2C_ROLE_SLAVE,I2C_ADDRESSING_MODE_07BIT,ADDRESS);

  I2C_Enable(APB_I2C0,1);
  // 打开传输结束中断和地址触发中断
  I2C_IntEnable(APB_I2C0,(1<<I2C_INT_ADDR_HIT)|(1<<I2C_INT_CMPL));

}
```

7.2.2.3 I2C 中断实现以及发送数据

```
static uint32_t peripherals_i2c_isr(void *user_data)
{
  uint8_t i;
  static uint8_t dir = 2;
  uint32_t status = I2C_GetIntState(APB_I2C0);

  // Slave 收到匹配的地址，触发中断
  if(status & (1 << I2C_STATUS_ADDRHIT))
  {
    // 判断是读操作还是写操作
    dir = I2C_GetTransactionDir(APB_I2C0);
    if(dir == I2C_TRANSACTION_MASTER2SLAVE)
    {
        I2C_IntEnable(APB_I2C0,(1 << I2C_INT_FIFO_FULL));
    }
    else if(dir == I2C_TRANSACTION_SLAVE2MASTER)
    {
        I2C_IntEnable(APB_I2C0,(1 << I2C_INT_FIFO_EMPTY));
    }

    I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_ADDRHIT));

  }

  // 如果是读操作，则会触发 empty 中断，此时需要填写需要发送的数据，直到 FIFO 满
  if(status & (1 << I2C_STATUS_FIFO_EMPTY))
  {
    // push data until fifo is full
    for(; write_data_cnt < DATA_CNT; write_data_cnt++)
    {
      if(I2C_FifoFull(APB_I2C0)){break;}
      I2C_DataWrite(APB_I2C0,write_data[write_data_cnt]);
```

```
    }
  }

  // 传输结束，清理打开的中断
  if(status & (1 << I2C_STATUS_CMPL))
  {

    I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_CMPL));

    // prepare for next
    if(dir == I2C_TRANSACTION_MASTER2SLAVE)
    {
        I2C_IntDisable(APB_I2C0,(1 << I2C_INT_FIFO_FULL));
    }
    else if(dir == I2C_TRANSACTION_SLAVE2MASTER)
    {
        I2C_IntDisable(APB_I2C0,(1 << I2C_INT_FIFO_EMPTY));
    }

  }

  return 0;
}
```

7.2.2.4 使用流程

- 设置 GPIO，setup_peripherals_i2c_pin()
- 初始化 I2C，setup_peripherals_i2c_module()
- 检查中断状态，在中断中发送数据，I2C_STATUS_CMPL 中断代表传输结束

7.3 场景 2：Master 写 Slave，不使用 DMA

其中 I2C 配置为 Master 写操作，Slave 收到地址后，将从 Master 读取数据，CPU 操作读写，没有使用 DMA。配置之前需要决定使用的 GPIO，请参考 datasheet 了解可以使用的 GPIO

```
#define I2C_SCL          GIO_GPIO_10
#define I2C_SDA          GIO_GPIO_11
```

7.3.1 I2C Master 配置

// 测试数据，每次传输 10 个字节（fifo 深度是 8 字节），每个传输单元必须是 1 字节

```
#define DATA_CNT (10)
uint8_t write_data[DATA_CNT] = {0,};
uint8_t write_data_cnt = 0;
```

7.3.1.1 配置 Pin

将 GPIO 映射成 I2C 引脚

```
void setup_peripherals_i2c_pin(void)
{
  // 打开 clock，注意此处使用的是 I2C0
  // GPIO 的 clock 请根据需要打开 GPIO0 或者 GPIO1
  SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_I2C0)
                              | (1 << SYSCTRL_ITEM_APB_SysCtrl)
                              | (1 << SYSCTRL_ITEM_APB_PinCtrl)
                              | (1 << SYSCTRL_ITEM_APB_GPIO1)
                              | (1 << SYSCTRL_ITEM_APB_GPIO0));
  // 使用默认内部上拉，实际使用中建议使用外部上拉 - 响应速度更快，可以支持更高的时钟
  // 如果使用外部上拉，则不需要 pull
  PINCTRL_Pull(I2C_SCL,PINCTRL_PULL_UP);
```

```
  PINCTRL_Pull(I2C_SDA,PINCTRL_PULL_UP);

  // 将 GPIO 映射成 I2C 引脚
  PINCTRL_SelI2cIn(I2C_PORT_0,I2C_SCL,I2C_SDA);
  PINCTRL_SetPadMux(I2C_SCL,IO_SOURCE_I2C0_SCL_OUT);
  PINCTRL_SetPadMux(I2C_SDA,IO_SOURCE_I2C0_SDA_OUT);

  // 打开 I2C 中断
  platform_set_irq_callback(PLATFORM_CB_IRQ_I2C0, peripherals_i2c_isr, NULL);

}
```

7.3.1.2 初始化 I2C 模块

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
  // 配置为 Master, 7bit 地址
  I2C_Config(APB_I2C0,I2C_ROLE_MASTER,I2C_ADDRESSING_MODE_07BIT,ADDRESS);
  // 配置时钟, 可选
  I2C_ConfigClkFrequency(APB_I2C0,I2C_CLOCKFREQUENY_STANDARD);

  I2C_Enable(APB_I2C0,1);
  // 打开传输结束中断和 fifo 空中断
  I2C_IntEnable(APB_I2C0,(1<<I2C_INT_CMPL)|(1 << I2C_INT_FIFO_EMPTY));

}
```

7.3.1.3 I2C 中断实现

```
static uint32_t peripherals_i2c_isr(void *user_data)
{
  uint8_t i;
  uint32_t status = I2C_GetIntState(APB_I2C0);

  // 传输结束，触发中断
  if(status & (1 << I2C_STATUS_CMPL))
  {
    I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_CMPL));
  }

  // empty FIFO 中断，填写需要发送的数据，数据全部发送之后，关掉 I2C_INT_FIFO_EMPTY
  if(status & (1 << I2C_STATUS_FIFO_EMPTY))
  {
    // push data until fifo is full
    for(; write_data_cnt < DATA_CNT; write_data_cnt++)
    {
      if(I2C_FifoFull(APB_I2C0)){break;}
      I2C_DataWrite(APB_I2C0,write_data[write_data_cnt]);
    }

    // if its the last, disable empty int
    if(write_data_cnt == DATA_CNT)
    {
      I2C_IntDisable(APB_I2C0,(1 << I2C_INT_FIFO_EMPTY));
    }

  }

  return 0;
}
```

7.3.1.4 I2C master 触发传输

```
void peripheral_i2c_send_data(void)
{
  // 设置方向，Master 写
  I2C_CtrlUpdateDirection(APB_I2C0,I2C_TRANSACTION_MASTER2SLAVE);
  // 设置每次传输的大小
  I2C_CtrlUpdateDataCnt(APB_I2C0, DATA_CNT);
  // 触发传输
  I2C_CommandWrite(APB_I2C0, I2C_COMMAND_ISSUE_DATA_TRANSACTION);

}
```

7.3.1.5 使用流程

- 设置 GPIO，setup_peripherals_i2c_pin()
- 初始化 I2C，setup_peripherals_i2c_module()
- 在需要时候发送 I2C 数据，peripheral_i2c_send_data()
- 检查中断状态

7.3.2 I2C Slave 配置

// 测试数据，每次传输 10 个字节（fifo 深度是 8 字节），每个传输单元必须是 1 字节

```
#define DATA_CNT (10)
uint8_t read_data[DATA_CNT] = {0,};
uint8_t read_data_cnt = 0;
```

7.3.2.1 配置 Pin

将 GPIO 映射成 I2C 引脚

```
void setup_peripherals_i2c_pin(void)
{
  // 打开 clock，注意此处使用的是 I2C0
  // GPIO 的 clock 请根据需要打开 GPIO0 或者 GPIO1
  SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_I2C0)
                              | (1 << SYSCTRL_ITEM_APB_SysCtrl)
                              | (1 << SYSCTRL_ITEM_APB_PinCtrl)
                              | (1 << SYSCTRL_ITEM_APB_GPIO1)
                              | (1 << SYSCTRL_ITEM_APB_GPIO0));
  // 使用默认内部上拉，实际使用中建议使用外部上拉 - 响应速度更快，可以支持更高的时钟
  // 如果使用外部上拉，则不需要 pull
  PINCTRL_Pull(I2C_SCL,PINCTRL_PULL_UP);
  PINCTRL_Pull(I2C_SDA,PINCTRL_PULL_UP);

  // 将 GPIO 映射成 I2C 引脚
  PINCTRL_SelI2cIn(I2C_PORT_0,I2C_SCL,I2C_SDA);
  PINCTRL_SetPadMux(I2C_SCL,IO_SOURCE_I2C0_SCL_OUT);
  PINCTRL_SetPadMux(I2C_SDA,IO_SOURCE_I2C0_SDA_OUT);

  // 打开 I2C 中断
  platform_set_irq_callback(PLATFORM_CB_IRQ_I2C0, peripherals_i2c_isr, NULL);

}
```

7.3.2.2 初始化 I2C 模块

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
  // 配置为 Slave, 7bit 地址
  I2C_Config(APB_I2C0,I2C_ROLE_SLAVE,I2C_ADDRESSING_MODE_07BIT,ADDRESS);
```

```
  I2C_Enable(APB_I2C0,1);
  // 打开传输结束中断和地址触发中断
  I2C_IntEnable(APB_I2C0,(1<<I2C_INT_ADDR_HIT)|(1<<I2C_INT_CMPL));

}
```

7.3.2.3 I2C 中断实现以及发送数据

```
static uint32_t peripherals_i2c_isr(void *user_data)
{
  uint8_t i;
  static uint8_t dir = 2;
  uint32_t status = I2C_GetIntState(APB_I2C0);

  // Slave 收到匹配的地址，触发中断
  if(status & (1 << I2C_STATUS_ADDRHIT))
  {
    // 判断是读操作还是写操作
    dir = I2C_GetTransactionDir(APB_I2C0);
    if(dir == I2C_TRANSACTION_MASTER2SLAVE)
    {
        I2C_IntEnable(APB_I2C0,(1 << I2C_INT_FIFO_FULL));
    }
    else if(dir == I2C_TRANSACTION_SLAVE2MASTER)
    {
        I2C_IntEnable(APB_I2C0,(1 << I2C_INT_FIFO_EMPTY));
    }

    I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_ADDRHIT));

  }
```

```
// 等待 FIFO FULL，读取 FIFO 直到 FIFO 变空
if(status & (1 << I2C_STATUS_FIFO_FULL))
{
  for(; read_data_cnt < DATA_CNT; read_data_cnt++)
  {
    if(I2C_FifoEmpty(APB_I2C0)){break;}
    read_data[read_data_cnt] = I2C_DataRead(APB_I2C0);
  }
}

// 传输结束，读取 FIFO 中剩余的数据，清除相关中断
if(status & (1 << I2C_STATUS_CMPL))
{
  for(;read_data_cnt < DATA_CNT; read_data_cnt++)
  {
    read_data[read_data_cnt] = I2C_DataRead(APB_I2C0);
  }

  I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_CMPL));

  // prepare for next
  if(dir == I2C_TRANSACTION_MASTER2SLAVE)
  {
      I2C_IntDisable(APB_I2C0,(1 << I2C_INT_FIFO_FULL));
  }
  else if(dir == I2C_TRANSACTION_SLAVE2MASTER)
  {
      I2C_IntDisable(APB_I2C0,(1 << I2C_INT_FIFO_EMPTY));
  }

}


return 0;
```

```
}
```

7.3.2.4 使用流程

- 设置 GPIO，setup_peripherals_i2c_pin()
- 初始化 I2C，setup_peripherals_i2c_module()
- 检查中断状态，I2C_STATUS_CMPL 中断代表传输结束

7.4 场景 3：Master 读 Slave，使用 DMA

其中 I2C 配置为 Master 读操作，Slave 收到地址后，将数据返回给 Master，DMA 操作读写。配置之前需要决定使用的 GPIO，请参考 datasheet 了解可以使用的 GPIO

```
#define I2C_SCL         GIO_GPIO_10
#define I2C_SDA         GIO_GPIO_11
```

7.4.1 I2C Master 配置

// 测试数据，每次传输 23 个字节（fifo 深度是 8 字节），每个传输单元必须是 1 字节

```
#define DATA_CNT (23)
uint8_t read_data[DATA_CNT] = {0,};
uint8_t read_data_cnt = 0;
```

7.4.1.1 配置 Pin

将 GPIO 映射成 I2C 引脚

```
void setup_peripherals_i2c_pin(void)
{
  // 打开 clock，注意此处使用的是 I2C0
  // GPIO 的 clock 请根据需要打开 GPIO0 或者 GPIO1
  SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_I2C0)
                              | (1 << SYSCTRL_ITEM_APB_SysCtrl)
                              | (1 << SYSCTRL_ITEM_APB_PinCtrl)
                              | (1 << SYSCTRL_ITEM_APB_GPIO1)
                              | (1 << SYSCTRL_ITEM_APB_GPIO0));
  // 使用默认内部上拉，实际使用中建议使用外部上拉 - 响应速度更快，可以支持更高的时钟
  // 如果使用外部上拉，则不需要 pull
  PINCTRL_Pull(I2C_SCL,PINCTRL_PULL_UP);
  PINCTRL_Pull(I2C_SDA,PINCTRL_PULL_UP);

  // 将 GPIO 映射成 I2C 引脚
  PINCTRL_SelI2cIn(I2C_PORT_0,I2C_SCL,I2C_SDA);
  PINCTRL_SetPadMux(I2C_SCL,IO_SOURCE_I2C0_SCL_OUT);
  PINCTRL_SetPadMux(I2C_SDA,IO_SOURCE_I2C0_SDA_OUT);

  // 打开 I2C 中断
  platform_set_irq_callback(PLATFORM_CB_IRQ_I2C0, peripherals_i2c_isr, NULL);

}
```

7.4.1.2 初始化 I2C 模块

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
  // 配置为 Master, 7bit 地址
  I2C_Config(APB_I2C0,I2C_ROLE_MASTER,I2C_ADDRESSING_MODE_07BIT,ADDRESS);
  // 配置时钟，可选
```

```
  I2C_ConfigClkFrequency(APB_I2C0,I2C_CLOCKFREQUENY_STANDARD);

  I2C_Enable(APB_I2C0,1);
  // 打开传输结束中断
  I2C_IntEnable(APB_I2C0,(1<<I2C_INT_CMPL));

}
```

7.4.1.3 初始化 DMA 模块

```
static void setup_peripherals_dma_module(void)
{
    SYSCTRL_ClearClkGateMulti(1 << SYSCTRL_ClkGate_APB_DMA);
    DMA_Reset(1);
    DMA_Reset(0);
}
```

7.4.1.4 I2C 中断实现

```
static uint32_t peripherals_i2c_isr(void *user_data)
{
  uint8_t i;
  uint32_t status = I2C_GetIntState(APB_I2C0);

  if(status & (1 << I2C_STATUS_CMPL))
  {
    I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_CMPL));
  }

  return 0;
}
```

7.4.1.5 I2C master DMA 设置

```
// 注意此处是 I2C0
void peripherals_i2c_rxfifo_to_dma(int channel_id, void *dst, int size)
{
  DMA_Descriptor descriptor __attribute__((aligned (8)));

  descriptor.Next = (DMA_Descriptor *)0;
  DMA_PreparePeripheral2Mem(&descriptor,dst,SYSCTRL_DMA_I2C0,size,DMA_ADDRESS_INC,0);

  DMA_EnableChannel(channel_id, &descriptor);
}
```

7.4.1.6 I2C master 触发传输

```
void peripheral_i2c_send_data(void)
{
  // I2C DMA 功能打开
  I2C_DmaEnable(APB_I2C0,1);
  // 设置传输方向，Master 读
  I2C_CtrlUpdateDirection(APB_I2C0,I2C_TRANSACTION_SLAVE2MASTER);
  // 设置需要传输的数据大小
  I2C_CtrlUpdateDataCnt(APB_I2C0, DATA_CNT);

  // 配置 DMA
  #define I2C_DMA_RX_CHANNEL   (0)//DMA channel 0
  peripherals_i2c_rxfifo_to_dma(I2C_DMA_RX_CHANNEL, read_data, sizeof(read_data));

  // 触发传输
  I2C_CommandWrite(APB_I2C0, I2C_COMMAND_ISSUE_DATA_TRANSACTION);

}
```

7.4.1.7 使用流程

- 设置 GPIO，setup_peripherals_i2c_pin()
- 初始化 I2C，setup_peripherals_i2c_module()
- 初始化 DMA，setup_peripherals_dma_module()
- 在需要时候触发 I2C 读取，peripheral_i2c_send_data()
- 检查中断状态

7.4.2 I2C Slave 配置

// 测试数据，每次传输 23 个字节（fifo 深度是 8 字节），每个传输单元必须是 1 字节

```
#define DATA_CNT (23)
uint8_t write_data[DATA_CNT] = {0,};
uint8_t write_data_cnt = 0;
```

7.4.2.1 配置 Pin

将 GPIO 映射成 I2C 引脚

```
void setup_peripherals_i2c_pin(void)
{
  // 打开 clock，注意此处使用的是 I2C0
  // GPIO 的 clock 请根据需要打开 GPIO0 或者 GPIO1
  SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_I2C0)
                              | (1 << SYSCTRL_ITEM_APB_SysCtrl)
                              | (1 << SYSCTRL_ITEM_APB_PinCtrl)
                              | (1 << SYSCTRL_ITEM_APB_GPIO1)
                              | (1 << SYSCTRL_ITEM_APB_GPIO0));
  // 使用默认内部上拉，实际使用中建议使用外部上拉 - 响应速度更快，可以支持更高的时钟
  // 如果使用外部上拉，则不需要 pull
  PINCTRL_Pull(I2C_SCL,PINCTRL_PULL_UP);
  PINCTRL_Pull(I2C_SDA,PINCTRL_PULL_UP);
```

```
  // 将 GPIO 映射成 I2C 引脚
  PINCTRL_SelI2cIn(I2C_PORT_0,I2C_SCL,I2C_SDA);
  PINCTRL_SetPadMux(I2C_SCL,IO_SOURCE_I2C0_SCL_OUT);
  PINCTRL_SetPadMux(I2C_SDA,IO_SOURCE_I2C0_SDA_OUT);

  // 打开 I2C 中断
  platform_set_irq_callback(PLATFORM_CB_IRQ_I2C0, peripherals_i2c_isr, NULL);

}
```

7.4.2.2 初始化 I2C 模块

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
  // 配置为 Slave, 7bit 地址
  I2C_Config(APB_I2C0,I2C_ROLE_SLAVE,I2C_ADDRESSING_MODE_07BIT,ADDRESS);

  I2C_Enable(APB_I2C0,1);
  // 打开传输结束中断和地址触发中断
  I2C_IntEnable(APB_I2C0,(1<<I2C_INT_ADDR_HIT)|(1<<I2C_INT_CMPL));

}
```

7.4.2.3 初始化 DMA 模块

```
static void setup_peripherals_dma_module(void)
{
    SYSCTRL_ClearClkGateMulti(1 << SYSCTRL_ClkGate_APB_DMA);
    DMA_Reset(1);
```

```
    DMA_Reset(0);
}
```

7.4.2.4 I2C 中断实现以及发送数据

```
static uint32_t peripherals_i2c_isr(void *user_data)
{
  uint8_t i;
  static uint8_t dir = 2;
  uint32_t status = I2C_GetIntState(APB_I2C0);

  // Slave 收到了匹配的地址，触发中断，判断方向
  if(status & (1 << I2C_STATUS_ADDRHIT))
  {
    dir = I2C_GetTransactionDir(APB_I2C0);
    if(dir == I2C_TRANSACTION_MASTER2SLAVE)
    {

    }
    else if(dir == I2C_TRANSACTION_SLAVE2MASTER)
    {
      // 设置 DMA，发送数据
      peripherals_i2c_write_data_dma_setup();
    }
    I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_ADDRHIT));
  }

  // 传输结束，关闭 DMA
  if(status & (1 << I2C_STATUS_CMPL))
  {
    I2C_DmaEnable(APB_I2C0,0);
    I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_CMPL));
```

```
  }

  return 0;
}
```

7.4.2.5 I2C Slave 发送数据 DMA 设置

```
// 此处配置的是 I2C0
void peripherals_i2c_dma_to_txfifo(int channel_id, void *src, int size)
{
  DMA_Descriptor descriptor __attribute__((aligned (8)));

  descriptor.Next = (DMA_Descriptor *)0;
  DMA_PrepareMem2Peripheral(&descriptor,SYSCTRL_DMA_I2C0,src,size,DMA_ADDRESS_INC,0);

  DMA_EnableChannel(channel_id, &descriptor);
}

void peripherals_i2c_write_data_dma_setup(void)
{
  // 设置 DMA
  #define I2C_DMA_TX_CHANNEL   (0)//DMA channel 0
  peripherals_i2c_dma_to_txfifo(I2C_DMA_TX_CHANNEL, write_data, sizeof(write_data));
  // 更新需要传输的数据
  I2C_CtrlUpdateDataCnt(APB_I2C0, DATA_CNT);
  // 打开 I2C DMA
  I2C_DmaEnable(APB_I2C0,1);
}
```

7.4.2.6 使用流程

- 设置 GPIO，setup_peripherals_i2c_pin()

- 初始化 I2C，setup_peripherals_i2c_module()
- 初始化 DMA，setup_peripherals_dma_module()
- 检查中断状态，在中断中设置 DMA 发送数据，I2C_STATUS_CMPL 中断代表传输结束

7.5 场景 4：Master 写 Slave，使用 DMA

其中 I2C 配置为 Master 写操作，Slave 收到地址后，读取 Master 发送的数据，DMA 操作读写。配置之前需要决定使用的 GPIO，请参考 datasheet 了解可以使用的 GPIO

```
#define I2C_SCL         GIO_GPIO_10
#define I2C_SDA         GIO_GPIO_11
```

7.5.1 I2C Master 配置

// 测试数据，每次传输 23 个字节（fifo 深度是 8 字节），每个传输单元必须是 1 字节

```
#define DATA_CNT (23)
uint8_t write_data[DATA_CNT] = {0,};
uint8_t write_data_cnt = 0;
```

7.5.1.1 配置 Pin

将 GPIO 映射成 I2C 引脚

```
void setup_peripherals_i2c_pin(void)
{
  // 打开 clock，注意此处使用的是 I2C0
  // GPIO 的 clock 请根据需要打开 GPIO0 或者 GPIO1
  SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_I2C0)
                              | (1 << SYSCTRL_ITEM_APB_SysCtrl)
                              | (1 << SYSCTRL_ITEM_APB_PinCtrl)
```

```
                                | (1 << SYSCTRL_ITEM_APB_GPIO1)
                                | (1 << SYSCTRL_ITEM_APB_GPIO0));
  // 使用默认内部上拉，实际使用中建议使用外部上拉 - 响应速度更快，可以支持更高的时钟
  // 如果使用外部上拉，则不需要 pull
  PINCTRL_Pull(I2C_SCL,PINCTRL_PULL_UP);
  PINCTRL_Pull(I2C_SDA,PINCTRL_PULL_UP);

  // 将 GPIO 映射成 I2C 引脚
  PINCTRL_SelI2cIn(I2C_PORT_0,I2C_SCL,I2C_SDA);
  PINCTRL_SetPadMux(I2C_SCL,IO_SOURCE_I2C0_SCL_OUT);
  PINCTRL_SetPadMux(I2C_SDA,IO_SOURCE_I2C0_SDA_OUT);

  // 打开 I2C 中断
  platform_set_irq_callback(PLATFORM_CB_IRQ_I2C0, peripherals_i2c_isr, NULL);

}
```

7.5.1.2 初始化 I2C 模块

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
  // 配置为 Master, 7bit 地址
  I2C_Config(APB_I2C0,I2C_ROLE_MASTER,I2C_ADDRESSING_MODE_07BIT,ADDRESS);
  // 配置时钟，可选
  I2C_ConfigClkFrequency(APB_I2C0,I2C_CLOCKFREQUENY_STANDARD);

  I2C_Enable(APB_I2C0,1);
  // 打开传输结束中断
  I2C_IntEnable(APB_I2C0,(1<<I2C_INT_CMPL));

}
```

7.5.1.3 初始化 DMA 模块

```
static void setup_peripherals_dma_module(void)
{
    SYSCTRL_ClearClkGateMulti(1 << SYSCTRL_ClkGate_APB_DMA);
    DMA_Reset(1);
    DMA_Reset(0);
}
```

7.5.1.4 I2C 中断实现

```
static uint32_t peripherals_i2c_isr(void *user_data)
{
  uint8_t i;
  uint32_t status = I2C_GetIntState(APB_I2C0);

  if(status & (1 << I2C_STATUS_CMPL))
  {
    I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_CMPL));
  }

  return 0;
}
```

7.5.1.5 I2C master DMA 设置

```
// 注意此处是 I2C0
void peripherals_i2c_dma_to_txfifo(int channel_id, void *src, int size)
{
```

```
    DMA_Descriptor descriptor __attribute__((aligned (8)));

    descriptor.Next = (DMA_Descriptor *)0;
    DMA_PrepareMem2Peripheral(&descriptor,SYSCTRL_DMA_I2C0,src,size,DMA_ADDRESS_INC,0

    DMA_EnableChannel(channel_id, &descriptor);
}
```

7.5.1.6 I2C master 触发传输

```
void peripheral_i2c_send_data(void)
{
  // I2C DMA 功能打开
  I2C_DmaEnable(APB_I2C0,1);
  // 设置传输方向，Master 读
  I2C_CtrlUpdateDirection(APB_I2C0,I2C_TRANSACTION_MASTER2SLAVE);
  // 设置需要传输的数据大小
  I2C_CtrlUpdateDataCnt(APB_I2C0, DATA_CNT);

  // 配置 DMA
  #define I2C_DMA_TX_CHANNEL   (0)//DMA channel 0
  peripherals_i2c_dma_to_txfifo(I2C_DMA_TX_CHANNEL, write_data, sizeof(write_data));

  // 触发传输
  I2C_CommandWrite(APB_I2C0, I2C_COMMAND_ISSUE_DATA_TRANSACTION);

}
```

7.5.1.7 使用流程

- 设置 GPIO，setup_peripherals_i2c_pin()
- 初始化 I2C，setup_peripherals_i2c_module()

- 初始化 DMA，setup_peripherals_dma_module()
- 在需要时候触发 I2C 读取，peripheral_i2c_send_data()
- 检查中断状态

7.5.2 I2C Slave 配置

// 测试数据，每次传输 23 个字节（fifo 深度是 8 字节），每个传输单元必须是 1 字节

```
#define DATA_CNT (23)
uint8_t read_data[DATA_CNT] = {0,};
uint8_t read_data_cnt = 0;
```

7.5.2.1 配置 Pin

将 GPIO 映射成 I2C 引脚

```
void setup_peripherals_i2c_pin(void)
{
  // 打开 clock，注意此处使用的是 I2C0
  // GPIO 的 clock 请根据需要打开 GPIO0 或者 GPIO1
  SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_I2C0)
                              | (1 << SYSCTRL_ITEM_APB_SysCtrl)
                              | (1 << SYSCTRL_ITEM_APB_PinCtrl)
                              | (1 << SYSCTRL_ITEM_APB_GPIO1)
                              | (1 << SYSCTRL_ITEM_APB_GPIO0));
  // 使用默认内部上拉，实际使用中建议使用外部上拉 - 响应速度更快，可以支持更高的时钟
  // 如果使用外部上拉，则不需要 pull
  PINCTRL_Pull(I2C_SCL,PINCTRL_PULL_UP);
  PINCTRL_Pull(I2C_SDA,PINCTRL_PULL_UP);

  // 将 GPIO 映射成 I2C 引脚
  PINCTRL_SelI2cIn(I2C_PORT_0,I2C_SCL,I2C_SDA);
  PINCTRL_SetPadMux(I2C_SCL,IO_SOURCE_I2C0_SCL_OUT);
```

```
  PINCTRL_SetPadMux(I2C_SDA,IO_SOURCE_I2C0_SDA_OUT);

  // 打开 I2C 中断
  platform_set_irq_callback(PLATFORM_CB_IRQ_I2C0, peripherals_i2c_isr, NULL);

}
```

7.5.2.2 初始化 I2C 模块

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
  // 配置为 Slave, 7bit 地址
  I2C_Config(APB_I2C0,I2C_ROLE_SLAVE,I2C_ADDRESSING_MODE_07BIT,ADDRESS);

  I2C_Enable(APB_I2C0,1);
  // 打开传输结束中断和地址触发中断
  I2C_IntEnable(APB_I2C0,(1<<I2C_INT_ADDR_HIT)|(1<<I2C_INT_CMPL));

}
```

7.5.2.3 初始化 DMA 模块

```
static void setup_peripherals_dma_module(void)
{
    SYSCTRL_ClearClkGateMulti(1 << SYSCTRL_ClkGate_APB_DMA);
    DMA_Reset(1);
    DMA_Reset(0);
}
```

7.5.2.4 I2C 中断实现以及发送数据

```
static uint32_t peripherals_i2c_isr(void *user_data)
{
  uint8_t i;
  static uint8_t dir = 2;
  uint32_t status = I2C_GetIntState(APB_I2C0);

  // Slave 收到了匹配的地址，检查传输方向
  if(status & (1 << I2C_STATUS_ADDRHIT))
  {
    dir = I2C_GetTransactionDir(APB_I2C0);
    if(dir == I2C_TRANSACTION_MASTER2SLAVE)
    {
      // 设置 DMA 读取数据
      peripherals_i2c_read_data_dma_setup();
    }
    else if(dir == I2C_TRANSACTION_SLAVE2MASTER)
    {

    }
    I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_ADDRHIT));
  }

  // 传输结束，关闭 DMA
  if(status & (1 << I2C_STATUS_CMPL))
  {
    I2C_DmaEnable(APB_I2C0,0);
    I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_CMPL));
  }

  return 0;
}
```

7.5.2.5 I2C Slave 发送数据 DMA 设置

```
// 此处配置的是 I2C0
void peripherals_i2c_rxfifo_to_dma(int channel_id, void *dst, int size)
{
    DMA_Descriptor descriptor __attribute__((aligned (8)));

    descriptor.Next = (DMA_Descriptor *)0;
    DMA_PreparePeripheral2Mem(&descriptor,dst,SYSCTRL_DMA_I2C0,size,DMA_ADDRESS_INC,0

    DMA_EnableChannel(channel_id, &descriptor);
}

void peripherals_i2c_read_data_dma_setup(void)
{
  // 设置 DMA
  #define I2C_DMA_RX_CHANNEL   (0)//DMA channel 0
  peripherals_i2c_rxfifo_to_dma(I2C_DMA_RX_CHANNEL, read_data, sizeof(read_data));
  // 更新需要传输的数据
  I2C_CtrlUpdateDataCnt(APB_I2C0, DATA_CNT);
  // 打开 I2C DMA
  I2C_DmaEnable(APB_I2C0,1);
}
```

7.5.2.6 使用流程

- 设置 GPIO，setup_peripherals_i2c_pin()
- 初始化 I2C，setup_peripherals_i2c_module()
- 初始化 DMA，setup_peripherals_dma_module()
- 检查中断状态，在中断中设置 DMA 读取数据，I2C_STATUS_CMPL 中断代表传输结束

7.6 场景 5：Master 写 Slave + Master 读 Slave

其中 I2C 操作为，首先执行写操作然后再执行读操作，CPU 操作读写，没有使用 DMA。配置之前需要决定使用的 GPIO，请参考 datasheet 了解可以使用的 GPIO

```
#define I2C_SCL          GIO_GPIO_10
#define I2C_SDA          GIO_GPIO_11
```

7.6.1 I2C Master 配置

// 测试数据，每次传输 8 个字节（fifo 深度是 8 字节），每个传输单元必须是 1 字节

```
#define DATA_CNT (8)
uint8_t write_data[DATA_CNT] = {0,};
uint8_t write_data_cnt = 0;
uint8_t read_data[DATA_CNT] = {0,};
uint8_t read_data_cnt = 0;
```

7.6.1.1 配置 Pin

IO 的配置请参考场景 1。

7.6.1.2 初始化 I2C 模块

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
  // 配置为 Master, 7bit 地址
  I2C_Config(APB_I2C0,I2C_ROLE_MASTER,I2C_ADDRESSING_MODE_07BIT,ADDRESS);
  // 配置时钟, 可选
  I2C_ConfigClkFrequency(APB_I2C0,I2C_CLOCKFREQUENY_STANDARD);
```

```
  I2C_Enable(APB_I2C0,1);
  // 打开传输结束中断和 addr hit 中断
  // 对于 master 来说，如果有 slave 响应了地址，则会有 I2C_INT_ADDR_HIT 中断
  I2C_IntEnable(APB_I2C0,(1<<I2C_INT_CMPL)|(1<<I2C_INT_ADDR_HIT));

}
```

7.6.1.3 I2C 中断实现

首先定义一个变量，如果为 1，代表是 master 的写操作，否则为 master 的读操作。

```
uint8_t master_write_flag = 0;
```

```
static uint32_t peripherals_i2c_isr(void *user_data)
{
  uint8_t i;
  uint32_t status = I2C_GetIntState(APB_I2C0);

  // 传输结束中断，代表 DATA_CNT 个字节接收或者发射完成
  if(status & (1 << I2C_STATUS_CMPL))
  {
    // master 写 DATA_CNT 个字节成功
    if(master_write_flag)
    {
      platform_printf("wr cmp %d\n", I2C_CtrlGetDataCnt(APB_I2C0));
      I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_CMPL));
    }
    else// master 读 DATA_CNT 个字节成功
    {
      // 将剩余的 fifo 中的数据读出来
      for(; read_data_cnt < DATA_CNT; read_data_cnt++)
```

```
    {
      read_data[read_data_cnt] = I2C_DataRead(APB_I2C0);
    }

    I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_CMPL));

    // debug trace
    platform_printf("rd cmp %d ",read_data_cnt);
    for(i=0;i<DATA_CNT;i++){platform_printf(" 0x%x -", read_data[i]);};
    printf("\n");

    // prepare for next
    read_data_cnt = 0;
  }
}

// 有 slave 响应了 master 的地址
if(status & (1 << I2C_STATUS_ADDRHIT))
{
  // 打开相应中断，主要是用来填写或者读取 fifo
  if(master_write_flag)
  {
    I2C_IntDisable(APB_I2C0,(1 << I2C_INT_FIFO_FULL));
    I2C_IntEnable(APB_I2C0,(1<<I2C_INT_CMPL)|(1 << I2C_INT_FIFO_EMPTY));
  }
  else
  {
    I2C_IntDisable(APB_I2C0,(1 << I2C_INT_FIFO_EMPTY));
    I2C_IntEnable(APB_I2C0,(1<<I2C_INT_CMPL)|(1 << I2C_INT_FIFO_FULL));
  }
  I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_ADDRHIT));
  platform_printf("addr hit\n");
}
```

```
// 该中断在 master_write_flag==1 打开，代表 fifo 为空，需要填充待发送的数据
if(status & (1 << I2C_STATUS_FIFO_EMPTY))
{
  if(master_write_flag)
  {
    // push data until fifo is full
    for(; write_data_cnt < DATA_CNT; write_data_cnt++)
    {
      //platform_printf("ept: %d \n",write_data_cnt);
      if(I2C_FifoFull(APB_I2C0)){break;}
      I2C_DataWrite(APB_I2C0,write_data[write_data_cnt]);
    }

    // if its the last, disable empty int
    if(write_data_cnt == DATA_CNT)
    {
      I2C_IntDisable(APB_I2C0,(1 << I2C_INT_FIFO_EMPTY));
    }

  }
}

// 该中断在 master_write_flag==0 打开，FIFO 满之后，触发中断，此时需要将所有数据都读
if(status & (1 << I2C_STATUS_FIFO_FULL))
{
  if(!master_write_flag)
  {
    for(; read_data_cnt < DATA_CNT; read_data_cnt++)
    {
      if(I2C_FifoEmpty(APB_I2C0)){break;}
      read_data[read_data_cnt] = I2C_DataRead(APB_I2C0);
    }
    platform_printf("rd full %d \n", read_data_cnt);
  }
```

```
  }

  return 0;
}
```

7.6.1.4 I2C master 写传输

```
void peripheral_i2c_write_data(void)
{
  master_write_flag = 1;
  // 设置方向，Master 写
  I2C_CtrlUpdateDirection(APB_I2C0,I2C_TRANSACTION_MASTER2SLAVE);
  // 设置每次传输的大小
  I2C_CtrlUpdateDataCnt(APB_I2C0, DATA_CNT);
  // 触发传输
  I2C_CommandWrite(APB_I2C0, I2C_COMMAND_ISSUE_DATA_TRANSACTION);

}
```

7.6.1.5 I2C master 读传输

```
void peripheral_i2c_read_data(void)
{
  master_write_flag = 0;
  I2C_CtrlUpdateDirection(APB_I2C0,I2C_TRANSACTION_SLAVE2MASTER);
  I2C_CtrlUpdateDataCnt(APB_I2C0, DATA_CNT);
  I2C_CommandWrite(APB_I2C0, I2C_COMMAND_ISSUE_DATA_TRANSACTION);
}
```

7.6.1.6 使用流程

- 设置 GPIO，setup_peripherals_i2c_pin()
- 初始化 I2C，setup_peripherals_i2c_module()
- 在需要时候触发 I2C 写数据，peripheral_i2c_write_data()，I2C_STATUS_CMPL 代表结束
- 当写结束后，可以触发 I2C 读取，peripheral_i2c_read_data()，I2C_STATUS_CMPL 代表结束
- 检查中断状态

7.6.2 I2C Slave 配置

7.6.2.1 配置 Pin

IO 的配置请参考场景 1。

7.6.2.2 初始化 I2C 模块

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
  // 配置为 Slave, 7bit 地址
  I2C_Config(APB_I2C0,I2C_ROLE_SLAVE,I2C_ADDRESSING_MODE_07BIT,ADDRESS);

  I2C_Enable(APB_I2C0,1);
  // 打开传输结束中断和地址触发中断
  I2C_IntEnable(APB_I2C0,(1<<I2C_INT_ADDR_HIT)|(1<<I2C_INT_CMPL));

}
```

7.6.2.3 I2C 中断实现以及发送数据

```
static uint32_t peripherals_i2c_isr(void *user_data)
{
  uint8_t i;
  static uint8_t dir = 2;
  uint32_t status = I2C_GetIntState(APB_I2C0);

  // Slave 收到匹配的地址，触发中断
  if(status & (1 << I2C_STATUS_ADDRHIT))
  {
    // 判断是读操作还是写操作
    dir = I2C_GetTransactionDir(APB_I2C0);
    if(dir == I2C_TRANSACTION_MASTER2SLAVE)
    {
        master_write_flag = 1;
        I2C_IntEnable(APB_I2C0,(1 << I2C_INT_FIFO_FULL));
        platform_printf("addr wr 0x%x\n", APB_I2C0->IntEn);
    }
    else if(dir == I2C_TRANSACTION_SLAVE2MASTER)
    {
        master_write_flag = 0;
        I2C_IntEnable(APB_I2C0,(1 << I2C_INT_FIFO_EMPTY));

        //示例：  slave 此时应将要发送的数据准备好
        write_data_cnt = 0;
        memset(write_data, 0x44, sizeof(write_data));
        platform_printf("addr rd \n");
    }

    I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_ADDRHIT));

  }

  // 如果是读操作，则会触发 empty 中断，此时需要填写需要发送的数据，直到 FIFO 满
  if(status & (1 << I2C_STATUS_FIFO_EMPTY))
```

```
{
  // master read
  if(!master_write_flag)
  {
    // push data until fifo is full
    for(; write_data_cnt < DATA_CNT; write_data_cnt++)
    {
      if(I2C_FifoFull(APB_I2C0)){break;}
      I2C_DataWrite(APB_I2C0,write_data[write_data_cnt]);
    }
    platform_printf("rd empty %d \n", write_data_cnt);
  }
}

// 如果是写操作，FIFO 满之后会触发该中断，slave 应读取 fifo 数据
if(status & (1 << I2C_STATUS_FIFO_FULL))
{
  // master write
  if(master_write_flag)
  {
    for(; read_data_cnt < DATA_CNT; read_data_cnt++)
    {
      if(I2C_FifoEmpty(APB_I2C0)){break;}
      read_data[read_data_cnt] = I2C_DataRead(APB_I2C0);
    }
    platform_printf("wr full %d \n", read_data_cnt);
  }
}

// 传输结束，清理打开的中断
if(status & (1 << I2C_STATUS_CMPL))
{
  // master write，此时应读取 fifo 中剩余的数据
  if(master_write_flag)
```

```
{
  for(;read_data_cnt < DATA_CNT; read_data_cnt++)
  {
    read_data[read_data_cnt] = I2C_DataRead(APB_I2C0);
  }

  I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_CMPL));

  // prepare for next
  if(dir == I2C_TRANSACTION_MASTER2SLAVE)
  {
      I2C_IntDisable(APB_I2C0,(1 << I2C_INT_FIFO_FULL));
  }
  else if(dir == I2C_TRANSACTION_SLAVE2MASTER)
  {
      I2C_IntDisable(APB_I2C0,(1 << I2C_INT_FIFO_EMPTY));
  }

  // debug trace
  platform_printf("wr cmp %d ",read_data_cnt);
  for(i=0;i<DATA_CNT;i++){platform_printf(" 0x%x -", read_data[i]);};
  printf("\n");

  read_data_cnt = 0;
}
else//master 写操作，清理中断
{
  I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_CMPL));

  // prepare for next
  if(dir == I2C_TRANSACTION_MASTER2SLAVE)
  {
      I2C_IntDisable(APB_I2C0,(1 << I2C_INT_FIFO_FULL));
  }
```

```
      else if(dir == I2C_TRANSACTION_SLAVE2MASTER)
      {
          I2C_IntDisable(APB_I2C0,(1 << I2C_INT_FIFO_EMPTY));
      }

      write_data_cnt = 0;
      platform_printf("rd cmp \n ");
    }

  }

  return 0;
}
```

7.6.2.4 使用流程

- 设置 GPIO，setup_peripherals_i2c_pin()
- 初始化 I2C，setup_peripherals_i2c_module()
- 检查中断状态，在中断中发送数据，I2C_STATUS_CMPL 中断代表传输结束
- 如果是读操作，slave 应该在 master_write_flag=0 之后准备好数据写到 fifo

7.7 I2C 时钟配置

I2C 时钟配置使用 API：

```
/**
 * @brief Set clk frequency for controller.
 * @param[in] I2C_BASE              base address
 * @param[in] option                see I2C_ClockFrequenyOptions
 */
void I2C_ConfigClkFrequency(I2C_TypeDef *I2C_BASE, I2C_ClockFrequenyOptions option);
```

其中 option 中定义了几个可选项

```
typedef enum
{
    I2C_CLOCKFREQUENY_NULL,
    I2C_CLOCKFREQUENY_STANDARD,//up to 100kbit/s
    I2C_CLOCKFREQUENY_FASTMODE,//up to 400kbit/s
    I2C_CLOCKFREQUENY_FASTMODE_PLUS,//up to 1Mbit/s
    I2C_CLOCKFREQUENY_MANUAL
} I2C_ClockFrequenyOptions;
```

如果选择 MANUAL, 需要手动配置相关寄存器来生成需要的时钟

```
I2C_BASE->TPM : multiply factor, width 5bit, 所有 setup 中的时间参数都会被乘以（TPM+1）
I2C_BASE->Setup: 使用 I2C_ConfigSCLTiming() 配置该寄存器
  scl_hi: 高电平持续时间, width 9bit, default 0x10
  scl_ratio: 低电平持续时间因子, width 1bit, default 1
  hddat: SCL 拉低后 SDA 的保持时间, width 5bit, default 5
  sp: 可以被过滤的脉冲毛刺宽度, width 3bit, default 1
  sudat: 释放 SCL 之前的数据建立时间, width 5bit, default 5
```

- 高电平持续时间计算：SCL high period = (2 * pclk) + (2 + sp + scl_hi) * pclk * (TPM + 1) 其中 pclk 为 I2C 模块的系统时钟，默认为 24M

如果 sp = 1, pclk = 42ns, TPM = 3, scl_hi = 150：

SCL high period = (2 * 42) + (2 + 1 + 150) * 42 * (3 + 1) = 25788ns

- 低电平持续时间计算：SCL low period = (2 * pclk) + (2 + sp + scl_hi * (scl_ratio+1)) * pclk * (TPM + 1)

如果 sp = 1, pclk = 42ns, TPM = 3, scl_hi = 150, scl_ratio = 0：

SCL low period = (2 * 42) + (2 + 1 + 150 * 1) * 42 * (3 + 1) = 25788ns

- 毛刺抑制宽度：spike suppression width = sp * pclk * (TPM + 1)

如果 sp = 1, pclk = 42ns, TPM = 3:

spike suppression width = 1 * 42 * (3 + 1) = 168ns

- SCL 之前的数据建立时间：setup time = (2 * pclk) + (2 + sp + sudat) * pclk * (TPM + 1)

如果 sp = 1, pclk = 42ns, TPM = 3, sudat = 5:

setup time = (2 * 42) + (2 + 1 + 5) * 42 * (3 + 1) = 1428ns

协议对 SCL 之前的数据建立时间要求为：

standard mode: 最小 250ns - fast mode: 最小 100ns - fast mode plus: 最小 50ns

- SCL 拉低后 SDA 的保持时间 hold time = (2 * pclk) + (2 + sp + hddat) * pclk * (TPM + 1)

如果 sp = 1, pclk = 42ns, TPM = 3, hddat = 5: hold time = (2 * 42) + (2 + 1 + 5) * 42 * (3 + 1) = 1428ns

协议对 SCL 拉低后 SDA 的保持时间要求为：standard mode: 最小 300ns - fast mode: 最小 300ns - fast mode plus: 最小 0ns

第八章　I2s 简介

I2S（inter-IC sound）总线是数字音频专用总线。它有四个引脚，两个数据引脚 (DOUT 和 DIN)，一个位率时钟引脚 (BCLK) 和一个左右通道选择引脚 (LRCLK)。

另外，通过 ING91682A 的 MCLK 输出，它可用于给外部 DAC/ADC 芯片提供时钟（可选）。

8.1　功能描述

8.1.1　特点

- 遵从 I2S 协议标准，支持 12S 标准模式和左对齐模式
- 支持 PCM(脉冲编码调制) 时序
- 可编程的主从模式
- 可配置的 LRCLK 和 BCLK 极性
- 可配置数据位宽
- 独立发送和接收 FIFO
- TX 和 RX 的 FIFO 深度分别为 16*32bit
- 支持立体声和单声道模式
- 可配置的采样频率
- TX 和 RX 分别支持 DMA 搬运

8.1.2 I2s 角色

在 I2S 总线上，提供时钟和通道选择信号的器件是 MASTER，另一方则为 SLAVER。

MASTER 和 SLAVE 都可以进行数据收发。

8.1.3 I2s 工作模式

I2S 有两种工作模式：一种是立体声音频模式，另外一种是话音模式。

8.1.4 串行数据

串行数据是以高位（MSB）在前，低位（LSB）在后的方式进行传送的。

如果音频 codec 发送的位数多于 I2S 控制器的接收位数，I2S 控制器会将低位多余的位数忽略掉；

如果音频 codec 发送的位数小于 I2S 控制器接收位数，I2S 控制器将后面的位补零。

8.1.5 时钟分频

916 芯片可选用系统 24MHz 时钟或者 PLL 作为 I2s 时钟源。

位率时钟（BCLK）可以通过对功能时钟进行分频得到；

通道选择时钟（LRCLK）即音频数据的采样频率可以通过对 BCLK 进行分频得到。

音频 Codec 中对采样频率 LRCLK 要求精度比较高，我们在计算分频时应该首先根据不同的采样频率计算得到对应的 MCLK 和 BCLK。

8.1.5.1 时钟分频计算

计算示例：

假设当前 codec 采用 16K 采样频率，mic 要求一帧 64 位（参考具体的 mic 使用手册）。

有以下关系：

- f_bclk = clk/(2*b_div)
- f_lrclk = f_bclk/(2*lr_div)

其中 clk 为 codec 时钟，f_bclk、f_lrclk 分别为 BCLK 和 LRCLK，b_div、lr_div 分别为 BCLK 和 LRCLK 的分频系数。

BCLK 和 LRCLK 之间的关系是可变的，但是 BCLK 必须大于等于 LRCLK 的 48 倍。即 lr_div>=24。

支持 lr_div = 32，DATA_LEN = 32 位的配置，其他情况下 lr_div - DATA_LEN > 3。

通过 f_lrclk = 16000，lr_div = 32 计算出 f_bclk = 1.024MHz。也就是 clk = 2.048*b_div。

clk 通过时钟源分频得到必定是整数，b_div 也同样是整数，通过计算得知在 384MHz 内只有当 b_div = 125 时 clk = 256MHz 为整数。

故需要将 PLL 时钟配置为 256MHz，b_div = 125 可以得到 16K 采样率。

8.1.6 i2s 存储器

采用两个深度为 16，宽度为 32bit 的 FIFO 分别存储接收、发送的音频数据。

有如下规则：

- 音频数据位宽为 16bit 时，每 32bit 存储两个音频数据，高 16bit 存储左声道数据，低 16bit 存储右声道数据。
- 音频数据位宽大于 16bit 时，每 32bit 存储一个音频数据，低地址存储左声道数据，高地址存储右声道数据。

8.2 使用方法

8.2.1 方法概述

I2s 使用方法总结为：时钟配置，I2s 配置（包括采样率）和数据处理。

数据发送：

1. I2s 引脚 GPIO 配置

2. 配置外部 codec 芯片，使其处于工作模式

3. 写相应配置寄存器

4. 将数据写入 TX_MEM

5. 使能 I2s

6. 等待中断产生

7. 读取状态寄存器，将数据写入 TX_MEM

8. 传输完毕，关闭 I2S

数据接收：

1. I2s 引脚 GPIO 配置

2. 配置外部 codec 芯片，使其处于工作模式

3. 写相应配置寄存器

4. 使能 I2s

5. 等待中断产生

6. 读取状态寄存器，读取 RX_MEM 中数据

7. 传输完毕，关闭 I2S

I2S 控制器操作流程图如下：

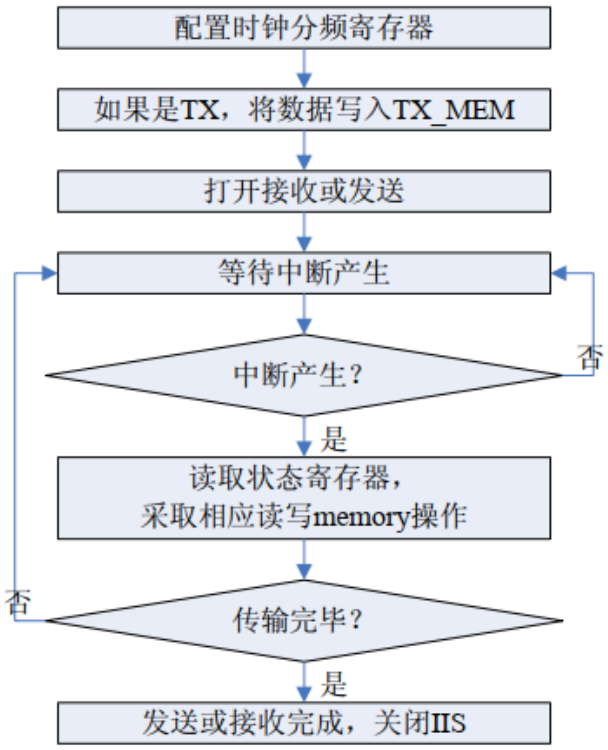


图 8.1: I2S 控制器操作流程图

如果需要用到 DMA 搬运则需要在使能 I2s 之前配置 DMA 并使能。

8.2.2 注意点

- I2s 时钟源可以选择晶振 24M 时钟和 PLL 时钟，要注意是选择哪一个时钟源
- I2s 数据可能会进行采样，需要注意具体的数据结构以及对应的数据处理，如是否需要数据移位等
- 当前 I2s 支持的发送/接收数据位宽为 16-32bit，需要查阅 mic 文档或其他使用手册来确定数据位宽，否则不能正常工作
- 配置 DMA 要在使能 I2s 之前完成，使能 I2s 一定是最后一步
- 建议采用 DMA 乒乓搬运的方式来传输 I2s 数据

8.3 编程指南

8.3.1 驱动接口

- I2S_ConfigClk：I2s 时钟配置接口
- I2S_Config：I2s 配置接口
- I2S_ConfigIRQ：I2s 中断配置接口
- I2S_DMAEnable：I2s DMA 使能接口
- I2S_Enable：I2s 使能接口
- I2S_PopRxFIFO、I2S_PushTxFIFO：I2s FIFO 读写接口
- I2S_ClearRxFIFO、I2S_ClearTxFIFO：I2s 清 FIFO 接口
- I2S_GetIntState、I2S_ClearIntState：I2s 获取中断、清中断接口
- I2S_GetRxFIFOCount、I2S_GetTxFIFOCount：I2s 获取 FIFO 数据数量接口
- I2S_DataFromPDM：I2s 获取 PDM 数据接口

8.3.2 代码示例

下面将通过实际代码展示 I2s 的基本配置及使用代码。

8.3.2.1 I2s 配置

```
#define I2S_PIN_BCLK        21
#define I2S_PIN_IN          22
#define I2S_PIN_LRCLK       35
void I2sSetup(void)
{
    // pinctrl & GPIO mux
    PINCTRL_SetPadMux(I2S_PIN_BCLK, IO_SOURCE_I2S_BCLK_OUT);
    PINCTRL_SetPadMux(I2S_PIN_IN, IO_SOURCE_I2S_DATA_IN);
    PINCTRL_SelI2sIn(IO_NOT_A_PIN, IO_NOT_A_PIN, I2S_PIN_IN);
    PINCTRL_SetPadMux(I2S_PIN_LRCLK, IO_SOURCE_I2S_LRCLK_OUT);
    PINCTRL_Pull(IO_SOURCE_I2S_DATA_IN, PINCTRL_PULL_DOWN);

    // CLK & Register
    SYSCTRL_ConfigPLLClk(6, 128, 2); // sorce clk PLL = 256MHz
    SYSCTRL_SelectI2sClk(SYSCTRL_CLK_PLL_DIV_1 + 4); // I2s_Clk = 51.2MHz
    I2S_ConfigClk(APB_I2S, 25, 32); // F_bclk = 1.024MHz, F_lrclk = 16K
    I2S_ConfigIRQ(APB_I2S, 0, 1, 0, 10);
    I2S_DMAEnable(APB_I2S, 0, 0);
    I2S_Config(APB_I2S, I2S_ROLE_MASTER, I2S_MODE_STANDARD, 0, 0, 0, 1, 24);

    // I2s interrupt
    platform_set_irq_callback(PLATFORM_CB_IRQ_I2S, cb_isr, 0);
}
```

8.3.2.2 I2s 使能

I2s 使能分 3 种情况：I2s 发送、I2s 接收、使用 DMA 搬运

接收：

```
void I2sStart(void)
{
    I2S_ClearRxFIFO(APB_I2S);
    I2S_Enable(APB_I2S, 0, 1);
}
```

发送：

```
uint8_t  sendSize = 10;
uint32_t sendData[10];
void I2sStart(void)
{
    int i;
    I2S_ClearTxFIFO(APB_I2S);
    // push data into TX_FIFO first
    for (i = 0; i < sendSize; i++) {
        I2S_PushTxFIFO(APB_I2S, sendData[i]);
    }
    I2S_Enable(APB_I2S, 0, 1);
}
```

使用 DMA（接收）：

```
#define CHANNEL_ID  0
DMA_Descriptor test __attribute__((aligned (8)));
void I2sStart(uint32_t data)
{
   DMA_EnableChannel(CHANNEL_ID, &test);
   I2S_ClearRxFIFO(APB_I2S);
   I2S_DMAEnable(APB_I2S, 1, 1);
   I2S_Enable(APB_I2S, 0, 1);
}
```

无论哪种情况都必须最后一步使能 I2s，否则 I2s 工作异常。

8.3.2.3 I2s 中断

接收：

```
uint32_t cb_isr(void *user_data)
{
    uint32_t state = I2S_GetIntState(APB_I2S);
    I2S_ClearIntState(APB_I2S, state);

    int i = I2S_GetRxFIFOCount(APB_I2S);

    while (i) {
        uint32_t data = I2S_PopRxFIFO(APB_I2S);
        i--;
        // do something with data
    }

    return 0;
}
```

发送：

```
uint8_t  sendSize = 10;
uint32_t sendData[10];
uint32_t cb_isr(void *user_data)
{
    uint32_t state = I2S_GetIntState(APB_I2S);
    I2S_ClearIntState(APB_I2S, state);

    int i;
    for (i = 0; i < sendSize; i++) {
        I2S_PushTxFIFO(APB_I2S, sendData[i]);
    }
    return 0;
}
```

8.3.2.4 I2s & DMA 乒乓搬运

下面以经典的 DMA 乒乓搬运 I2s 接收数据为例展示 I2s 实际使用方法。

这里我们采用 16K 采样率，单个数据帧固定 64 位，和 DMA 协商握手、burstSize=8、一次搬运 80 个数据。

```
#include "pingpong.h"
#define I2S_PIN_BCLK        21
#define I2S_PIN_IN          22
#define I2S_PIN_LRCLK       35
#define CHANNEL_ID  0
DMA_PingPong_t PingPong;

static uint32_t DMA_cb_isr(void *user_data)
{
    uint32_t state = DMA_GetChannelIntState(CHANNEL_ID);
    DMA_ClearChannelIntState(CHANNEL_ID, state);

    uint32_t *rr = DMA_PingPongIntProc(&PingPong, CHANNEL_ID);
    uint32_t i = 0;
    uint32_t transSize = DMA_PingPongGetTransSize(&PingPong);
    while (i < transSize) {
        // do something with data 'rr[i]'
        i++;
    }

    return 0;
}
void I2sSetup(void)
{
    // pinctrl & GPIO mux
```

```
    PINCTRL_SetPadMux(I2S_PIN_BCLK, IO_SOURCE_I2S_BCLK_OUT);
    PINCTRL_SetPadMux(I2S_PIN_IN, IO_SOURCE_I2S_DATA_IN);
    PINCTRL_SelI2sIn(IO_NOT_A_PIN, IO_NOT_A_PIN, I2S_PIN_IN);
    PINCTRL_SetPadMux(I2S_PIN_LRCLK, IO_SOURCE_I2S_LRCLK_OUT);
    PINCTRL_Pull(IO_SOURCE_I2S_DATA_IN, PINCTRL_PULL_DOWN);

    // CLK & Register
    SYSCTRL_ConfigPLLClk(6, 128, 2); // sorce clk PLL = 256MHz
    SYSCTRL_SelectI2sClk(SYSCTRL_CLK_PLL_DIV_1 + 4); // I2s_Clk = 51.2MHz
    I2S_ConfigClk(APB_I2S, 25, 32); // F_bclk = 1.024MHz, F_lrclk = 16K
    I2S_ConfigIRQ(APB_I2S, 0, 1, 0, 8);
    I2S_DMAEnable(APB_I2S, 0, 0);
    I2S_Config(APB_I2S, I2S_ROLE_MASTER, I2S_MODE_STANDARD, 0, 0, 0, 1, 24);

    // setup DMA
    DMA_PingPongSetup(&PingPong, SYSCTRL_DMA_I2S_RX, 100, 8);
    platform_set_irq_callback(PLATFORM_CB_IRQ_DMA, DMA_cb_isr, 0);

    // start working
    DMA_PingPongEnable(&PingPong, CHANNEL_ID);
    I2S_ClearRxFIFO(APB_I2S);
    I2S_DMAEnable(APB_I2S, 1, 1);
    I2S_Enable(APB_I2S, 0, 1);
}
```

DMA（乒乓搬运）的具体用法请参见本手册 DMA 一节。

更加系统化的 I2s 代码请参考 SDK 中 voice_remote_ctrl 例程。

第九章　IR 红外

9.1　功能概述

- 支持红外发射 & 接收
- 时序可调整，支持多种编码

9.2　使用说明

9.2.1　参数 (不同编码的时间参数)

```
//32KHz 调制时钟
#define FCLK            32000

//NEC 载波频率 38KHz
#define NEC_WAVE_FREQ   38000

// RC5 载波频率 36KHz
#define RC5_WAVE_FREQ   36000

// 计算载波频率发生器数据，由此产生如 38KHz NEC 的载波，NEC 与 TC9012 通用 38KHz 载波
#define IR_WAVE_NEC_TC9012_FREQ (OSC_CLK_FREQ/NEC_WAVE_FREQ)
#define IR_WAVE_RC5_FREQ (OSC_CLK_FREQ/RC5_WAVE_FREQ)

//通过 FCLK 产生各种协议每 bit 位调制周期最小单位，如 NEC 协议中，560us 为最小调制周期
#define NEC_UINT    (FCLK*560/1000000+1)
```

```
#define TC9012_UINT (FCLK*260/1000000+1)
#define RC5_UINT    (FCLK*889/1000000+1)

//不必要的参数
#define INESSENTIAL     0

typedef struct
{
    uint16_t timer1;//发送模式下，表示引导码低电平时间：如 NEC 为 16*UNIT = 9ms，接收模式
    uint16_t timer2;//发送模式下，表示重复码低电平时间：如 NEC 为 4*UNIT = 2.25ms，接收模
    uint16_t timer3;//发送模式下，表示引导码高电平时间：如 NEC 为 8*UNIT = 4.5ms，接收模式
    uint16_t timer4;//发送模式下，表示重复码高电平时间：如 NEC 为 UNIT = 560us，接收模式下
    uint16_t timer5;//接收时接收超时定时器，发射不必关注
    uint16_t btimer1;//逻辑 0 的 bit 时长：如 NEC = 2*UINT = 1.12ms.
    uint16_t btimer2;//逻辑 1 的 bit 时长：如 NEC = 4*UINT = 2.25ms.
    uint16_t bit_cycle;//发射模式下：bit 调制周期最小单位，如 NEC 为 560us，接收模式下为 0
    uint16_t carry_low;//载波低电平时长，与 carry_high 组合形成占空比可调的载波波形，如 NE
    uint16_t carry_high;//载波高电平时长，与 carry_low 组合形成占空比可调的载波波形，如 NE
}Ir_mode_param_t;

//由于发送和接收初始化不同参数 定义结构体表示接收发送初始化参数。
typedef struct{
    Ir_mode_param_t param_tx;
    Ir_mode_param_t param_rx;
}Ir_type_param_t;

//定义初始化数据，初始化函数体根据不同协议自动从此表适配参数
const static Ir_type_param_t t_ir_type_param_table[] =
{
    {//NEC param
        {  //TX
            16*NEC_UINT-1,4*NEC_UINT-1,8*NEC_UINT-1,1*NEC_UINT-1,INESSENTIAL,
            2*NEC_UINT-1, 4*NEC_UINT-1,1*NEC_UINT-1,IR_WAVE_NEC_TC9012_FREQ*2/3,IR_WAVE_N
        {   //RX
```

```
            14*NEC_UINT-1,18*NEC_UINT-1,22*NEC_UINT-1,26*NEC_UINT-1,0xfff,
            INESSENTIAL,2*NEC_UINT-1,0x7f,INESSENTIAL,INESSENTIAL},
    },
    {//TC9012 param
        {   //TX
            16*TC9012_UINT-1,8*TC9012_UINT-1,16*TC9012_UINT-1,2*TC9012_UINT-1,INESSEN
            4*TC9012_UINT-1,6*TC9012_UINT-1,2*TC9012_UINT-1,IR_WAVE_NEC_TC9012_FREQ*2
        {    //RX
            28*TC9012_UINT-1,36*TC9012_UINT-1,44*TC9012_UINT-1,32*TC9012_UINT-1,0xff
            4*TC9012_UINT-1,0x7f,INESSENTIAL,INESSENTIAL},
    },
    {//RC5 param
        {   //TX
            2*RC5_UINT-1,2*RC5_UINT-1,INESSENTIAL,INESSENTIAL,INESSENTIAL,2*RC5_UINT-
            INESSENTIAL,1*RC5_UINT,IR_WAVE_RC5_FREQ*2/3,IR_WAVE_RC5_FREQ*1/3},

        {   //RX
            1*RC5_UINT-2,1*RC5_UINT,3*RC5_UINT-1,5*RC5_UINT-1,INESSENTIAL,1*RC5_UINT-
            1*RC5_UINT-1,2*RC5_UINT-1,INESSENTIAL,INESSENTIAL},
    }
};
```

9.2.2 红外发射接收

9.2.2.1 配置 pin

```
#define IR_DOUT GIO_GPIO_10
#define IR_DIN  GIO_GPIO_11

void setup_peripherals_ir_module(void)
{
    // 打开时钟
```

```
    // 大于等于 GPIO 18 则使用 SYSCTRL_ClkGate_APB_GPIO1, 否则是 SYSCTRL_ClkGate_APB_GPIO
    SYSCTRL_ClearClkGateMulti(   (1 << SYSCTRL_ClkGate_APB_GPIO0)
                               | (1 << SYSCTRL_ClkGate_APB_GPIO1)
                               | (1 << SYSCTRL_ClkGate_APB_IR)
                               | (1 << SYSCTRL_ClkGate_APB_PinCtrl));

    PINCTRL_SelIrIn(IR_DIN);
    PINCTRL_SetPadMux(IR_DOUT,IO_SOURCE_IR_DATA_OUT);
    platform_set_irq_callback(PLATFORM_CB_IRQ_IR_INT, IRQHandler_IR_INT, NULL);
}
```

9.2.2.2 配置模块

```
static void user_ir_device_init(IR_IrMode_e mode,IR_TxRxMode_e tx_rx_mode)
{
    IR_CtrlSetIrMode(APB_IR,mode);
    IR_CtrlSetTxRxMode(APB_IR,tx_rx_mode);
    IR_CtrlSetIrIntEn(APB_IR);

    if(IR_TXRX_MODE_TX_MODE == tx_rx_mode)
    {
        IR_TxConfigIrTxPol(APB_IR);
        IR_TxConfigCarrierCntClr(APB_IR);
        IR_TxConfigIrIntEn(APB_IR);
        IR_CarryConfigSetIrCarryLow(APB_IR,t_ir_type_param_table[mode].param_tx.carry_low
        IR_CarryConfigSetIrCarryHigh(APB_IR,t_ir_type_param_table[mode].param_tx.carry_hi
        IR_TimeSetIrTime1(APB_IR,t_ir_type_param_table[mode].param_tx.timer1);
        IR_TimeSetIrTime2(APB_IR,t_ir_type_param_table[mode].param_tx.timer2);
        IR_TimeSetIrTime3(APB_IR,t_ir_type_param_table[mode].param_tx.timer3 );
        IR_TimeSetIrTime4(APB_IR,t_ir_type_param_table[mode].param_tx.timer4);
        IR_CtrlIrSetBitTime1(APB_IR,t_ir_type_param_table[mode].param_tx.btimer1);
        IR_CtrlIrSetBitTime2(APB_IR,t_ir_type_param_table[mode].param_tx.btimer2);
```

```
        IR_CtrlIrSetIrBitCycle(APB_IR,t_ir_type_param_table[mode].param_tx.bit_cycle)
    }
    else{
        IR_CtrlSetIrEndDetectEn(APB_IR);//end code detect en
        IR_CtrlIrUsercodeVerify(APB_IR);
        IR_CtrlIrDatacodeVerify(APB_IR);
        IR_TimeSetIrTime1(APB_IR,t_ir_type_param_table[mode].param_rx.timer1);
        IR_TimeSetIrTime2(APB_IR,t_ir_type_param_table[mode].param_rx.timer2);
        IR_TimeSetIrTime3(APB_IR,t_ir_type_param_table[mode].param_rx.timer3);
        IR_TimeSetIrTime4(APB_IR,t_ir_type_param_table[mode].param_rx.timer4);
        IR_TimeSetIrTime5(APB_IR,t_ir_type_param_table[mode].param_rx.timer5);
        IR_CtrlIrSetBitTime1(APB_IR,t_ir_type_param_table[mode].param_rx.btimer1);
        IR_CtrlIrSetBitTime2(APB_IR,t_ir_type_param_table[mode].param_rx.btimer2);
        IR_CtrlIrSetIrBitCycle(APB_IR,t_ir_type_param_table[mode].param_rx.bit_cycle)
    }
    IR_CtrlEnable(APB_IR);
}
```

9.2.2.3 IR 中断 (以及 IR 接收)

```
static uint32_t IRQHandler_IR_INT(void *user_data)
{
    if(IR_FsmGetIrTransmitOk(APB_IR))
        ;//platform_printf("int ir send ok\n");
    if(IR_FsmGetIrTxRepeat(APB_IR))
        ;//platform_printf("int ir repeat ok\n");
    if(IR_FsmGetIrReceivedOk(APB_IR))
    {
        uint32_t value = IR_RxCodeGetIrRxUsercode(APB_IR) <<16 | IR_RxCodeGetIrRxData
        platform_printf("Received:0x%x\n",value);
    }
```

```
    if(IR_FsmGetIrRepeat(APB_IR))
        ;
    IR_FsmClearIrInt(APB_IR);
    return 0;
}
```

9.2.2.4 IR 发射

```
static int ir_transmit_fun(uint16_t addr,uint16_t data) //ir hard transmit data
{
    IR_TxCodeSetIrTxUsercode(APB_IR,addr);
    IR_TxCodeSetIrTxDatacode(APB_IR,data);
    IR_CleanIrTxRepeatMode(APB_IR);//must clearn the repeat mode reg
    IR_TxConfigTxStart(APB_IR);
    //while(!IR_FsmGetIrTransmitOk(APB_IR));
    return 0;
}

static int ir_transmit_repeat(void) //ir hard transmit repeat
{
    IR_CtrlIrTxRepeatMode(APB_IR);
    IR_TxConfigTxStart(APB_IR);
 //   while(!IR_FsmGetIrTransmitOk(APB_IR));
    return 0;
}
```

第十章　PDM 简介

PDM 全称 pulse density modulation，即脉冲密度调制。

PDM 模块处理来自外部音频前端 (如数字麦克风) 的脉冲密度调制信号的输入。该模块生成 PDM 时钟，支持双通道数据输入。

10.1　功能描述

10.1.1　特点

- 支持双通道，数据输入相同
- 16kHz 输出采样率，24 位采样
- HW 抽取过滤器
- 时钟和输出采样率之间的可选比为 64 或 80
- 支持 DMA 和 I2S 的样本缓冲

10.1.2　PDM & PCM

PDM 和 PCM 同为用数字信号表示模拟信号的音频数据调制方法，其主要区别是：

- PDM 不像 PCM 等间隔采样
- PDM 只有 1 位非 0 即 1 的输出，而 PCM 采样结果可以是 Nbit
- PDM 使用远高于 PCM 采样率的时钟频率进行采样
- PDM 逻辑相对 PCM 复杂
- PDM 只需要 2 根信号线，即时钟线和数据线；PCM 需要 4 根线

10.2 使用方法

10.2.1 方法概述

PDM 使用方法分为 PDM 结合 I2s 使用和 PDM 数据直接 DMA 搬运两种。

PDM 结合 I2s：

1. PDM 引脚 GPIO 配置（时钟、数据）

2. 外部时钟配置，使其处于工作模式

3. 写相应配置寄存器

4. 配置 I2s 数据源为 PDM，配置 I2s 时钟、寄存器、中断

5. 使能 PDM，使能 I2s

6. 等待 I2s 中断产生

7. 读取状态寄存器，读取 RX_MEM 中数据

8. 传输完毕，关闭 PDM 和 I2S

PDM 数据 DMA 搬运：

1. PDM 引脚 GPIO 配置（时钟、数据）

2. 外部时钟配置，使其处于工作模式

3. 写相应配置寄存器

4. 配置 DMA 寄存器、中断

5. 使能 DMA，使能 PDM

6. 等待 DMA 中断产生

7. 传输完毕，关闭 PDM 和 DMA

其中 I2s 和 DMA 相关配置不在本节介绍内容范围内，可参考本手册对应章节。

10.2.2 注意点

- I2s 时钟源可以选择晶振 24M 时钟和 PLL 时钟，要注意是选择哪一个时钟源
- 建议选择晶振 24M 作为时钟源，这样可以获得较好的准确度和防抖动效果
- 需要查阅数字麦克风数据手册了解其时钟要求，并正确配置 PDM 时钟频率

- 结合 I2s 使用时要先使能 PDM 最后开启 I2s
- 结合 DMA 使用时要先使能 DMA 最后开启 PDM
- 建议采用 DMA 乒乓搬运的方式

10.3 编程指南

10.3.1 驱动接口

- PDM_Config：PDM 配置接口
- PDM_Start：PDM 使能接口
- PDM_DmaEnable：PDM 使能 DMA 接口

10.3.2 代码示例

下面以 PDM 结合 I2s 使用和 PDM 数据直接 DMA 搬运两种方式来展示 PDM 的具体使用方法。已知现有 mic 使用的时钟频率为 3M，I2s 采样率 16K。

10.3.2.1 PDM 结合 I2s：

```
#define PDM_PIN_MCLK        28
#define PDM_PIN_IN          29
static uint32_t I2s_isr(void *user_data)
{
    uint32_t state = I2S_GetIntState(APB_I2S);
    I2S_ClearIntState(APB_I2S, state);

    int i = I2S_GetRxFIFOCount(APB_I2S);
    while (i) {
        // do something with these data
        i--;
```

```
    }
}

void audio_input_setup(void)
{
    // GPIO & Pin Ctrl
    PINCTRL_SetPadMux(PDM_PIN_MCLK, IO_SOURCE_PDM_DMIC_MCLK);
    PINCTRL_SetPadMux(PDM_PIN_IN, IO_SOURCE_PDM_DMIC_IN);
    PINCTRL_SelPdmIn(PDM_PIN_IN);

    // PDM clock configuration, 3M
    SYSCTRL_SetPdmClkDiv(8, 1);
    SYSCTRL_SelectTypeAClk(SYSCTRL_ITEM_APB_PDM, SYSCTRL_CLK_ADC_DIV);

    // PDM register configuration
    PDM_Config(APB_PDM, 0, 1, 1, 1, 0);

    // I2s configuration, bclk=2.4M, samplerate=16K, data from PDM
    I2S_DataFromPDM(1);
    I2S_ConfigClk(APB_I2S, 5, 75);
    I2S_ConfigIRQ(APB_I2S, 0, 1, 0, 10);
    I2S_DMAEnable(APB_I2S, 0, 0);
    I2S_Config(APB_I2S, I2S_ROLE_MASTER, I2S_MODE_STANDARD, 0, 1, 0, 1, 24);

    // I2s interruption
    platform_set_irq_callback(PLATFORM_CB_IRQ_I2S, I2s_isr, 0);

    // enable I2s and PDM
    I2S_ClearRxFIFO(APB_I2S);
    PDM_Start(APB_PDM, 1);
    I2S_Enable(APB_I2S, 0, 1);
}
```

上面示例涉及到的关于 I2s 配置参考手册的 I2s 章节：

10.3.2.2 PDM 数据 DMA 搬运

```
#define PDM_PIN_MCLK        28
#define PDM_PIN_IN          29
#define CHANNEL_ID  0
uint32_t buff[80];
DMA_Descriptor test __attribute__((aligned (8)));
static uint32_t DMA_cb_isr(void *user_data)
{
    uint32_t state = DMA_GetChannelIntState(CHANNEL_ID);
    DMA_ClearChannelIntState(CHANNEL_ID, state);

    DMA_EnableChannel(CHANNEL_ID, &test);

    // do something with data in buff
}

void DMA_SetUp(void)
{
    DMA_Reset();
    platform_set_irq_callback(PLATFORM_CB_IRQ_DMA, DMA_cb_isr, 0);
    test.Next = NULL;
    DMA_PreparePeripheral2RAM(&test,
                              buff,
                              SYSCTRL_DMA_PDM,
                              80,
                              DMA_ADDRESS_INC,
                              0 | 1 << 24);
    DMA_EnableChannel(CHANNEL_ID, &test);
}
void audio_input_setup(void)
{
    //GPIO & Pin Ctrl
    PINCTRL_SetPadMux(PDM_PIN_MCLK, IO_SOURCE_PDM_DMIC_MCLK);
```

```
    PINCTRL_SetPadMux(PDM_PIN_IN, IO_SOURCE_PDM_DMIC_IN);
    PINCTRL_SelPdmIn(PDM_PIN_IN);

    // PDM clock configuration, 3M
    SYSCTRL_SetAdcClkDiv(8, 1);
    SYSCTRL_SelectTypeAClk(SYSCTRL_ITEM_APB_PDM, SYSCTRL_CLK_ADC_DIV);

    // PDM register configuration
    PDM_Config(APB_PDM, 0, 1, 1, 0, 0);
    PDM_DmaEnable(APB_PDM, 1, 0);

    // DMA setup
    DMA_SetUp();

    // enable DMA and PDM
    PDM_DmaEnable(APB_PDM, 1, 1);
    PDM_Start(APB_PDM, 1);
}
```

建议采用 DMA 乒乓搬运的方法进行数据搬运，具体讲解参考手册 DMA 章节。

第十一章　管脚管理（PINCTRL）

11.1　功能概述

PINCTRL 模块管理芯片所有 IO 管脚的功能，包括外设 IO 的映射，上拉、下拉选择，输入模式控制，输出驱动能力设置等。

每个 IO 管脚都可以配置为数字或模拟模式，当配置为数字模式时，特性如下：

- 每个 IO 管脚可以映射多种不同功能的外设
- 每个 IO 管脚都支持上拉或下拉
- 每个 IO 管脚都支持施密特触发输入方式
- 每个 IO 管脚支持四种输出驱动能力

鉴于片内外设丰富、IO 管脚多，进行管脚全映射并不现实，为此，PINCTRL 尽量保证灵活性的前提下做了一定取舍、优化。部分常用外设的输入、输出功能管脚可与 $\{0..17, 21, 22, 31, 34, 35\}$ 这 23 个常用 IO 之间任意连接（全映射），这部分常用外设功能管脚总结于表 11.1。表 11.2 列出了其它外设功能管脚支持映射到哪些 IO 管脚上。除此以外，所有 IO 管脚都可以配置为 GPIO 或者 DEBUG 模式。GPIO 模式的输入、输出方向由 `GIO_SetDirection` 控制。DEBUG 模式为保留功能，具体功能暂不开放。

表 11.1: 支持与常用 IO 全映射的常用功能管脚

外设	功能管脚
I2C	I2C0_SCL_I, I2C0_SCL_O, I2C0_SDA_I, I2C0_SDA_O, I2C1_SCL_I, I2C1_SCL_O, I2C1_SDA_I, I2C1_SDA_O
I2S	I2S_BCLK_I, I2S_BCLK_O, I2S_DIN, I2S_DOUT, I2S_LRCLK_I, I2S_LRCLK_O
IR	IR_DATIN, IR_DATOUT, IR_WAKEUP

外设	功能管脚
PCAP	PCAP0_IN, PCAP1_IN, PCAP2_IN, PCAP3_IN, PCAP4_IN, PCAP5_IN
PDM	PDM_DMIC_IN, PDM_DMIC_MCLK
QDEC	QDEC_INDEX, QDEC_PHASEA, QDEC_PHASEB
SPI0	SPI0_CLK_IN, SPI0_CLK_OUT, SPI0_CSN_IN, SPI0_CSN_OUT, SPI0_HOLD_IN, SPI0_HOLD_OUT, SPI0_MISO_IN, SPI0_MISO_OUT, SPI0_MOSI_IN, SPI0_MOSI_OUT, SPI0_WP_IN, SPI0_WP_OUT
SPI	SPI1_CLK_IN, SPI1_CLK_OUT, SPI1_CSN_IN, SPI1_CSN_OUT, SPI1_HOLD_IN, SPI1_HOLD_OUT, SPI1_MISO_IN, SPI1_MISO_OUT, SPI1_MOSI_IN, SPI1_MOSI_OUT, SPI1_WP_IN, SPI1_WP_OUT
SWD	SWDO, SW_TCK, SW_TMS
UART0	UART0_CTS, UART0_RTS, UART0_RXD, UART0_TXD
UART1	UART1_CTS, UART1_RTS, UART1_RXD, UART1_TXD

表 11.2: 其它外设功能管脚的映射关系

外设功能管脚	可连接到的 IO 管脚
KEY_IN_COL_0	0, 23
KEY_IN_COL_1	1, 24
KEY_IN_COL_2	2, 25
KEY_IN_COL_3	3, 29
KEY_IN_COL_4	4, 30
KEY_IN_COL_5	5, 31
KEY_IN_COL_6	6, 32
KEY_IN_COL_7	7, 33
KEY_IN_COL_8	8, 34
KEY_IN_COL_9	9, 35
KEY_IN_COL_10	10, 36
KEY_IN_COL_11	11, 37
KEY_IN_COL_12	12, 38
KEY_IN_COL_13	13, 39
KEY_IN_COL_14	14, 40
KEY_IN_COL_15	15, 41
KEY_IN_COL_16	16

外设功能管脚	可连接到的 IO 管脚
KEY_IN_COL_17	17
KEY_IN_COL_18	21
KEY_IN_COL_19	22
KEY_OUT_ROW_0	0, 23
KEY_OUT_ROW_1	1, 24
KEY_OUT_ROW_2	2, 25
KEY_OUT_ROW_3	3, 29
KEY_OUT_ROW_4	4, 30
KEY_OUT_ROW_5	5, 31
KEY_OUT_ROW_6	6, 32
KEY_OUT_ROW_7	7, 33
KEY_OUT_ROW_8	8, 34
KEY_OUT_ROW_9	9, 35
KEY_OUT_ROW_10	10, 36
KEY_OUT_ROW_11	11, 37
KEY_OUT_ROW_12	12, 38
KEY_OUT_ROW_13	13, 39
KEY_OUT_ROW_14	14, 40
KEY_OUT_ROW_15	15, 41
KEY_OUT_ROW_16	16
KEY_OUT_ROW_17	17
KEY_OUT_ROW_18	21
KEY_OUT_ROW_19	22
ANT_SW0	0, 3, 6, 9, 12, 15, 21, 34
ANT_SW1	1, 4, 7, 10, 13, 16, 22, 35
ANT_SW2	2, 5, 8, 11, 14, 17, 31
ANT_SW3	0, 3, 6, 9, 12, 15, 21, 34
ANT_SW4	1, 4, 7, 10, 13, 16, 22, 35
ANT_SW5	2, 5, 8, 11, 14, 17, 31
ANT_SW6	0, 3, 6, 9, 12, 15, 21, 34
ANT_SW7	1, 4, 7, 10, 13, 16, 22, 35
PA_RXEN	11, 12, 13, 14, 15, 16, 17, 34, 35
PA_TXEN	4, 5, 6, 7, 8, 9, 10, 34, 35
TIMER0_PWM_0A	0, 2, 4, 6, 8, 10, 12, 14, 16, 21, 31, 35

外设功能管脚	可连接到的 IO 管脚
TIMER0_PWM_0B	1, 3, 5, 7, 9, 11, 13, 15, 17, 22, 34
TIMER0_PWM_1A	0, 2, 4, 6, 8, 10, 12, 14, 16, 21, 31, 35
TIMER0_PWM_1B	1, 3, 5, 7, 9, 11, 13, 15, 17, 22, 34
TIMER1_PWM_0A	0, 2, 4, 6, 8, 10, 12, 14, 16, 21, 31, 35
TIMER1_PWM_0B	1, 3, 5, 7, 9, 11, 13, 15, 17, 22, 34
TIMER1_PWM_1A	0, 2, 4, 6, 8, 10, 12, 14, 16, 21, 31, 35
TIMER1_PWM_1B	1, 3, 5, 7, 9, 11, 13, 15, 17, 22, 34
TIMER2_PWM_0A	0, 2, 4, 6, 8, 10, 12, 14, 16, 21, 31, 35
TIMER2_PWM_0B	1, 3, 5, 7, 9, 11, 13, 15, 17, 22, 34
TIMER2_PWM_1A	0, 2, 4, 6, 8, 10, 12, 14, 16, 21, 31, 35
TIMER2_PWM_1B	1, 3, 5, 7, 9, 11, 13, 15, 17, 22, 34
PWM_0A	0, 2, 4, 6, 8, 10, 12, 14, 16, 21, 31, 35
PWM_0B	1, 3, 5, 7, 9, 11, 13, 15, 17, 22, 34
PWM_1A	0, 2, 4, 6, 8, 10, 12, 14, 16, 21, 31, 35
PWM_1B	1, 3, 5, 7, 9, 11, 13, 15, 17, 22, 34
PWM_2A	0, 2, 4, 6, 8, 10, 12, 14, 16, 21, 31, 35
PWM_2B	1, 3, 5, 7, 9, 11, 13, 15, 17, 22, 34
QDEC_EXT_IN_CLK	3, 9
QDEC_TIMER_EXT_IN1_A	1, 7
QDEC_TIMER_EXT_IN2_A	2, 8
QDEC_TIMER_EXT_IN2_B	5, 11
QDEC_TIMER_EXT_OUT0_A	0, 6
QDEC_TIMER_EXT_OUT1_A	1, 7
QDEC_TIMER_EXT_OUT2_A	2, 8
QDEC_TIMER_EXT_OUT0_B	3, 9
QDEC_TIMER_EXT_OUT1_B	4, 10
QDEC_TIMER_EXT_OUT2_B	5, 11
SPI0_CLK_IN	19
SPI0_CLK_OUT	19
SPI0_CSN_IN	18
SPI0_CSN_OUT	18
SPI0_HOLD_IN	20
SPI0_HOLD_OUT	20
SPI0_MISO_IN	27

外设功能管脚	可连接到的 IO 管脚
SPI0_MISO_OUT	27
SPI0_MOSI_IN	28
SPI0_MOSI_OUT	28
SPI0_WP_IN	26
SPI0_WP_OUT	26
SPI2AHB_CS	16
SPI2AHB_DI	17
SPI2AHB_DO	17
SPI2AHB_SCLK	15

11.2 使用说明

11.2.1 为外设配置 IO 管脚

1. 将外设输出连接到 IO 管脚

   通过 `PINCTRL_SetPadMux` 将外设输出连接到 IO 管脚。注意按照表 11.1 和表 11.2 确认硬件是否支持。对于不支持的配置，显然无法生效，函数将返回非 0 值。

```
int PINCTRL_SetPadMux(
  const uint8_t io_pin_index, // IO 序号 (0 .. IO_PIN_NUMBER - 1)
  const io_source_t source    // IO 源
);
```

   例如将 IO 管脚 10 配置为 GPIO 模式：

```
PINCTRL_SetPadMux(10, IO_SOURCE_GPIO);
```

2. 将 IO 管脚连接到外设的输入

   对于有些外设的输入同样通过 `PINCTRL_SetPadMux` 配置。对于另一些输入，PINCTRL 为不同的外设分别提供了 API 用以配置输入。比如 UART 的数据输入 RXD 和用于硬件流控的 CTS，需要通过 `PINCTRL_SelUartIn` 配置：

```
int PINCTRL_SelUartIn(
  uart_port_t port,     // UART 序号
  uint8_t io_pin_rxd,   // 连接到 RXD 输入的 IO 管脚
  uint8_t io_pin_cts);  // 连接到 CTS 输入的 IO 管脚
```

对于不需要配置的输入，可在对应的参数上填入值 `IO_NOT_A_PIN`。

表 11.3 罗列了为各外设提供的输入配置函数。

表 11.3: 各外设的输入配置函数

外设	配置函数
KeyScan	`PINCTRL_SelKeyScanColIn`
I2C	`PINCTRL_SelI2cIn`
I2S	`PINCTRL_SelI2sIn`
IR	`PINCTRL_SelIrIn`
PDM	`PINCTRL_SelPdmIn`
PCAP	`PINCTRL_SelPCAPIn`
QDEC	`PINCTRL_SelQDECIn`
SWD	`PINCTRL_SelSwIn`
SPI	`PINCTRL_SelSpiIn`
UART	`PINCTRL_SelUartIn`
USB	`PINCTRL_SelUSB`

11.2.2 配置上拉、下拉

IO 管脚的上拉、下拉模式通过 `PINCTRL_Pull` 配置：

```
void PINCTRL_Pull(
  const uint8_t io_pin_index,      // IO 管脚序号
  const pinctrl_pull_mode_t mode   // 模式
  );
```

表 11.4 列出了各管脚默认的上下拉配置。

表 11.4: 管脚上下拉默认配置

管脚	默认配置
1	上拉
2	上拉
3	上拉
4	上拉
15	下拉
16	下拉
17	下拉
其它	禁用上下拉

11.2.3 配置驱动能力

通过 `PINCTRL_SetDriveStrength` 配置 IO 管脚的驱动能力：

```
void PINCTRL_SetDriveStrength(
  const uint8_t io_pin_index,
  const pinctrl_drive_strength_t strength);
```

默认驱动能力共分 4 档，分别为 $2mA$、$4mA$、$8mA$、$12mA$。除了 IO1 驱动能力默认为 $12mA$ 之外，其它管脚驱动能力默认 $8mA$。

11.2.4 配置天线切换控制管脚

支持最多 8 个管脚用于天线切换控制，相应地，天线切换模板（switching pattern）内每个数字包含 8 个比特，取值范围为 0..255。这 8 个比特可依次通过 `IO_SOURCE_ANT_SW0`、……、`IO_SOURCE_ANT_SW7` 选择。例如，查表 11.2 可知管脚 0 能够映射为 ANT_SW0，即比特 0。通过下面这行代码就可将管脚 0 映射为 ANT_SW0：

```
PINCTRL_SetPadMux(0, IO_SOURCE_ANT_SW0);
```

通过函数 `PINCTRL_EnableAntSelPins` 可批量配置用于天线切换控制的管脚：

```
int PINCTRL_EnableAntSelPins(
  int count,                // 数目
  const uint8_t *io_pins);  // 管脚数组
```

管脚数组 io_pins 里的第 n 个元素代表第 n 个比特所要映射的管脚。如果不需要为某个比特配置管脚，则在 io_pins 的对应位置填入 IO_NOT_A_PIN。比如，只选用第 0、第 2 等两个比特用作控制，分别映射到管脚 0 和 5：

```
const uint8_t io_pins[] = {0, IO_NOT_A_PIN, 5};
PINCTRL_EnableAntSelPins(sizeof(io_pins), io_pins);
```

11.2.5 配置模拟模式

通过以下 3 步可将一个管脚配置为模拟模式：

1. 配置为 GPIO 模式；
2. 禁用上下拉；
3. 将 GPIO 配置为高阻态。

函数 PINCTRL_EnableAnalog 封装了以上步骤：

```
void PINCTRL_EnableAnalog(const uint8_t io_index);
```

模拟模式适用于以下几种外设。

- USB

  函数 PINCTRL_SelUSB() 内部封装了 PINCTRL_EnableAnalog，开发者不需要再为 USB 管脚调用该函数配置模拟模式。

- ADC

  开发者需要调用该函数使能某管脚的 ADC 输入功能。支持 ADC 输入功能的管脚如表 11.5 所示。

表 11.5: 支持 ADC 输入的管脚

管脚	单端模式	差分模式
7	AIN 0	AIN 0 P
8	AIN 1	AIN 0 N
9	AIN 2	AIN 1 P
10	AIN 3	AIN 1 N
11	AIN 4	AIN 2 P
12	AIN 5	AIN 2 N
13	AIN 6	AIN 3 P
14	AIN 7	AIN 3 N

第十二章　PTE 简介

PTE 全称 Peripheral trigger engine，即外设触发引擎。

其主要作用是使外围设备可以通过其他外围设备或事件独立于 CPU 进行自主交互。PTE 允许外围设备之间可以精确触发。

12.1　功能描述

12.1.1　特点

- 支持 APB 总线触发
- 支持 4 通道 PTE
- 支持复用触发源或复用触发地址
- 支持产生 CPU 中断

12.1.2　PTE 原理图

12.1.3　功能

PTE 具有不同外设之间的可编程内部通道，可以从 src 外设触发 dst 外设。PTE 可以不依赖 CPU 而通过硬件的方式触发任务，因此任务可以在同步 DFF 所占用的周期内启动。

src 外设通过 pte_in_mask 配置，dst 外设通过 pte_out_mask 配置。在 SOC 中集成了 4 个 PTE 通道，每个通道可以通过通道使能信号来启用/禁用。

当 DFF 为高时 PTE 中断将挂起。在清除 PTE 中断之前，src 外设中断必须被清除，否则另一个启动脉冲将发送到 dst 外设，这可能会产生未知的错误。

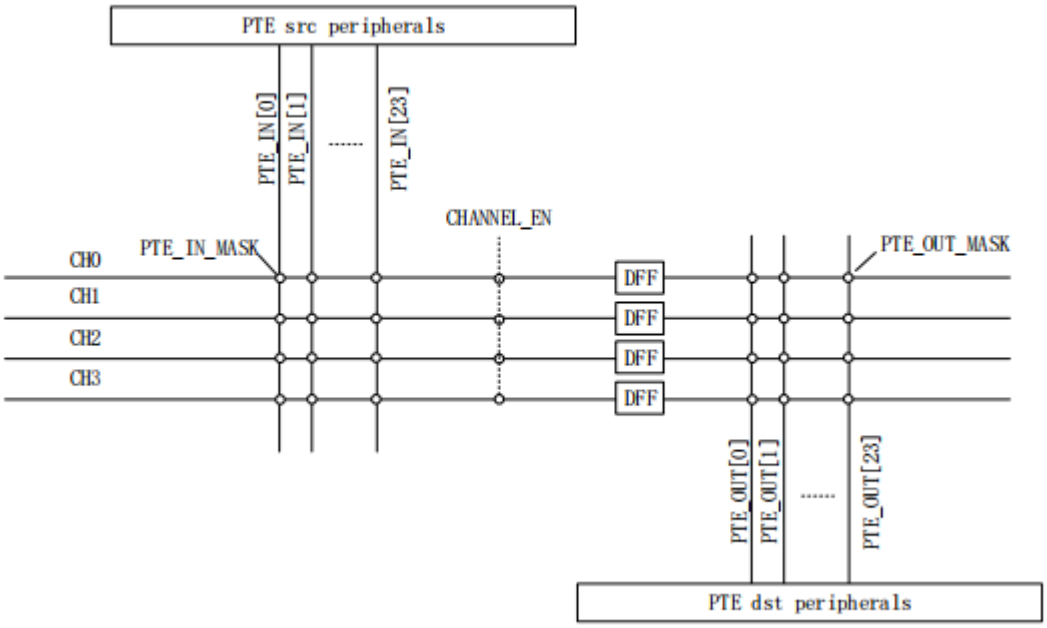


图 12.1: PTE 原理图

12.2 使用方法

12.2.1 方法概述

PTE 使用方法总结为：建议不使用 PTE 中断，在 dst 外设中断里清 PTE 中断（或关闭 PTE 通道）。

1. 配置触发外设和被触发外设以及相应中断（被触发外设中断一定要有）

2. 配置要使用的 PTE 通道寄存器以及中断（建议不使用 PTE 中断）

3. 使能触发外设，等待 PTE 中断（如定义）和被触发外设来中断

4. 在 PTE 中断中清 PTE mask（如定义）

5. 在被触发外设中断中清 PTE 中断（如定义），如果只触发一次则直接关闭 PTE 通道

12.2.2 注意点

- 不清 src 中断会循环通过 PTE 触发 dst 外设，使程序陷入死循环
- 不清 PTE 中断会循环触发 dst 外设，使程序陷入死循环

- PTE 中断优先级低容易被打断，在极端情况下如果 dst 外设来中断非常快会出问题（一般不会）
- 使用 PTE 中断会更多占用 CPU 资源并增加触发过程操作复杂度、增加出错风险，中断处理程序完全可以在 src 和 dst 中断中完成，所以强烈建议不要使用 PTE 中断

12.3 编程指南

12.3.1 src&dst 外设

当前 PTE 支持的 src 外设定义在 SYSCTRL_PTE_SRC_INT 中：

```
typedef enum
{
    SYSCTRL_PTE_I2C0_INT       = 0,
    SYSCTRL_PTE_I2C1_INT       = 1,
    SYSCTRL_PTE_SARADC_INT     = 2,
    SYSCTRL_PTE_I2S_INT        = 3,
    SYSCTRL_PTE_DMA_INT        = 4,
    SYSCTRL_PTE_IR_INT         = 5,
    SYSCTRL_PTE_KEYSCANNER_INT = 6,
    SYSCTRL_PTE_PWMC0_INT      = 7,
    SYSCTRL_PTE_PWMC1_INT      = 8,
    SYSCTRL_PTE_PWMC2_INT      = 9,
    SYSCTRL_PTE_TIMER0_INT     = 10,
    SYSCTRL_PTE_TIMER1_INT     = 11,
    SYSCTRL_PTE_TIMER2_INT     = 12,
    SYSCTRL_PTE_GPIO0_INT      = 13,
    SYSCTRL_PTE_GPIO1_INT      = 14,
    SYSCTRL_PTE_UART0_INT      = 15,
    SYSCTRL_PTE_UART1_INT      = 16,
    SYSCTRL_PTE_SPI0_INT       = 17,
    SYSCTRL_PTE_SPI1_INT       = 18,
    SYSCTRL_PTE_SPIFLASH       = 19,
```

```
    SYSCTRL_PTE_RCT_CNT         = 20,
    SYSCTRL_PTE_IR_WAKEUP       = 21,
    SYSCTRL_PTE_USB_INT         = 22,
    SYSCTRL_PTE_QDEC_INT        = 23,

    SYSCTRL_PTE_SRC_INT_MAX     = 24,
} SYSCTRL_PTE_SRC_INT;
```

dst 外设定义在 SYSCTRL_PTE_DST_EN 中：

```
typedef enum
{
    SYSCTRL_PTE_I2C0_EN         = 0,
    SYSCTRL_PTE_I2C1_EN         = 1,
    SYSCTRL_PTE_SARADC_EN       = 2,
    SYSCTRL_PTE_I2S_TX_EN       = 3,
    SYSCTRL_PTE_I2S_RX_EN       = 4,
    SYSCTRL_PTE_IR_EN           = 5,
    SYSCTRL_PTE_KEYSCANNER_EN   = 6,
    SYSCTRL_PTE_PWMC0_EN        = 7,
    SYSCTRL_PTE_PWMC1_EN        = 8,
    SYSCTRL_PTE_PWMC2_EN        = 9,
    SYSCTRL_PTE_TIMER0_CH0_EN   = 10,
    SYSCTRL_PTE_TIMER0_CH1_EN   = 11,
    SYSCTRL_PTE_TIMER1_CH0_EN   = 12,
    SYSCTRL_PTE_TIMER1_CH1_EN   = 13,
    SYSCTRL_PTE_TIMER2_CH0_EN   = 14,
    SYSCTRL_PTE_TIMER2_CH1_EN   = 15,

    SYSCTRL_PTE_DST_EN_MAX      = 16,
} SYSCTRL_PTE_DST_EN;
```

通过 PTE 连接的 src 外设和 dst 外设需要在已注册枚举中选取。

12.3.2 驱动接口

- PTE_ConnectPeripheral：PTE 外设连接接口
- PTE_EnableChennel：PTE 通道使能接口
- PTE_ChennelClose：PTE 通道关闭接口
- PTE_IrqProcess：PTE 标准中断程序接口
- PTE_OutPeripheralContinueProcess：dst 外设中断标准 PTE 中继触发接口
- PTE_OutPeripheralEndProcess：dst 外设中断标准 PTE 结束接口

12.3.3 代码示例

下面以 Timer0 通过 PTE 通道 0 触发 Timer1 为例展示 PTE 的具体使用方法。

src 外设和 dst 外设配置方法不在本文档介绍范围内，我们默认 Timer0 和 Timer1 已经配置好并注册好中断。

```
uint32_t Timer0Isr(void *user_data)
{
    TMR_IntClr(APB_TMR0);
    return 0;
}

uint32_t Timer1Isr(void *user_data)
{
    TMR_IntClr(APB_TMR1);
    PTE_OutPeripheralContinueProcess(0);
    return 0;
}

// 仅供参考，不建议注册 PTE 中断
uint32_t PTE0Isr(void *user_data)
{
```

```
    PTE_IrqProcess(0);
    return 0;
}

void PTE_Test(void)
{
    PTE_ConnectPeripheral(SYSCTRL_PTE_CHENNEL_0,
                          SYSCTRL_PTE_TIMER0_INT,
                          SYSCTRL_PTE_TIMER1_CH0_EN);
    TMR_Enable(APB_TMR0);
}
```

上面示例会保留 PTE 通道 0 并等待下一次触发。如果想要触发之后直接关闭通道代码如下：

```
uint32_t Timer1Isr(void *user_data)
{
    TMR_IntClr(APB_TMR1);
    PTE_OutPeripheralEndProcess(0);
    return 0;
}
```

关闭通道会断开 Timer0 和 Timer1 的连接，再次触发需要重新建立连接。

第十三章　增强型脉宽调制发生器（PWM）

增强型脉宽调制发生器具有两大功能：生成脉宽调制信号（PWM），捕捉外部脉冲输入（PCAP）。增强型脉宽调制发生器具备 3 个通道，每个通道都可以单独配置为 PWM 或者 PCAP 模式。每个通道拥有独立的 FIFO。FIFO 里的每个存储单元为 2 个 20bit 数据。FIFO 深度为 4，即最多存储 4 个单元，共 $8 \times 20bit$ 数据。这里的 20bit 位宽是因为本硬件模块内部 PWM 使用的各计数器都是 20 比特。可根据 FIFO 内的数据量触发中断或者 DMA 传输。

说明：TIMER 也支持生成脉宽调制信号，但是可配置的参数较简单，不支持死区等。

PWM 特性：

- 最多支持 3 个 PWM 通道，每一个通道包含 A、B 两个输出
- 每个通道参数独立
- 支持死区
- 支持通过 DMA 更新 PWM 配置

PCAP 特性：

- 支持 3 个 PCAP 通道，每一个通道包含两个输入
- 支持捕捉上升沿、下降沿
- 支持通过 DMA 读取数据

13.1　PWM 工作模式

PWM 使用的时钟频率可配置，请参考SYSCTRL。

每个 PWM 通道支持以下多种工作模式：

```
typedef enum
{
    ..._UP_WITHOUT_DIED_ZONE          = ...,
    ..._UP_WITH_DIED_ZONE             = ...,
    ..._UPDOWN_WITHOUT_DIED_ZONE      = ...,
    ..._UPDOWN_WITH_DIED_ZONE         = ...,
    ..._SINGLE_WITHOUT_DIED_ZONE      = ...,
    ..._DMA                           = ...,
} PWM_WorkMode_t;
```

13.1.1 最简单的模式：UP_WITHOUT_DIED_ZONE

此模式需要配置两个门限：计数器回零门限 PERA_TH、高门限 HIGH_TH，HIGH_TH 必须小于 HIGH_TH。以伪代码描述 A、B 输出如下：

```
cnt = 0;
on_clock_rising_edge()
{
    cnt = cnt < PERA_TH ? cnt + 1 : 0;
    A = HIGH_TH <= cnt;
    B = !A;
}
```

13.1.2 UP_WITH_DIED_ZONE

与 UP_WITHOUT_DIED_ZONE 相比，此模式需要一个新的死区门限 DZONE_TH，DZONE_TH 必须小于 HIGH_TH。以伪代码描述 A、B 输出如下：

cnt = 0; on_clock_rising_edge() { cnt = cnt < PERA_TH ? cnt + 1 : 0; A = HIGH_TH + DZONE_TH <= cnt; B = DZONE_TH <= cnt < HIGH_TH); }

13.1.3 UPDOWN_WITHOUT_DIED_ZONE

此模式需要的门限参数与 UP_WITHOUT_DIED_ZONE 相同。以伪代码描述 A、B 输出如下：

```
cnt = 0;
on_clock_rising_edge()
{
    cnt = cnt < 2 * PERA_TH ? cnt + 1 : 0;
    A = PERA_TH - HIGH_TH <= cnt <= PERA_TH + HIGH_TH;
    B = !A;
}
```

13.1.4 UPDOWN_WITH_DIED_ZONE

与 UP_WITHOUT_DIED_ZONE 相比，此模式需要一个新的死区门限 DZONE_TH。以伪代码描述 A、B 输出如下：

```
cnt = 0;
on_clock_rising_edge()
{
    cnt = cnt < 2 * PERA_TH ? cnt + 1 : 0;
    A = PERA_TH - HIGH_TH + DZONE_TH <= cnt <= PERA_TH + HIGH_TH;
    B = (cnt < PERA_TH - HIGH_TH) || (cnt > PERA_TH + HIGH_TH + DZONE_TH);
}
```

13.1.5 SINGLE_WITHOUT_DIED_ZONE

此模式需要配置两个门限：计数器回零门限 PERA_TH、高门限 HIGH_TH，HIGH_TH 必须小于 HIGH_TH。此模式只产生一个脉冲，以伪代码描述 A、B 输出如下：

```
cnt = 0;
on_clock_rising_edge()
{
    cnt++;
    A = HIGH_TH <= cnt < PERA_TH;
```

```
    B = !A;
}
```

以上伪代码仅用于辅助描述硬件行为，与实际行为可以存在微小差异。

13.1.6 DMA 模式

此模式支持通过 DMA 实时更新门限。

13.1.7 输出控制

对于每个通道的每一路输出，另有 3 个参数控制最终的两路输出：掩膜、停机输出值、反相。最终的输出以伪代码描述如下：

```
output_control(v)
{
    if (掩膜 == 1) return A 路输出 0、B 路输出 1;
    if (本通道已停机) return 停机输出值;
    if (反相) v = !v;
    return v;
}
```

13.2 PCAP

PCAP 每个通道包含两路输入。PCAP 内部有一个单独的 32 比特计数器[1]，当检测到输入信号变化（包含上升沿和下降沿）时，PCAP 将计数器的值及边沿变化信息作为一个存储单元压入 FIFO：

[1]所有 6 路输入共有此计数器。

```
struct data0
{
    uint32_t cnt_high:12;
    uint32_t p_cap_0_p:1; // A 路出现上升沿
    uint32_t p_cap_0_n:1; // A 路出现下降沿
    uint32_t p_cap_1_p:1; // B 路出现上升沿
    uint32_t p_cap_1_n:1; // B 路出现下降沿
    uint32_t tag:4;
    uint32_t padding:12;
};
struct data1
{
    uint32_t cnt_low:20;
    uint32_t padding:12;
};
```

通过复位整个模块可以清零 PCAP 计数器。

13.3 PWM 使用说明

13.3.1 启动与停止

共有两个开关与 PWM 的启动和停止有关：使能（Enable）、停机控制（HaltCtrl）。只有当 `Enable` 为 1，`HaltCtrl` 为 0 时，PWM 才真正开始工作。

相关的 API 为：

```
// 使能 PWM 通道
void PWM_Enable(
    const uint8_t channel_index,     // 通道号
    const uint8_t enable             // 使能或禁用
    );
```

```
// PWM 通道停机控制
void PWM_HaltCtrlEnable(
    const uint8_t channel_index,    // 通道号
    const uint8_t enable            // 停机 (1) 或运转 (0)
    );
```

13.3.2 配置工作模式

```
void PWM_SetMode(
    const uint8_t channel_index,    // 通道号
    const PWM_WorkMode_t mode       // 模式
    );
```

13.3.3 配置门限

```
// 配置 PERA_TH
void PWM_SetPeraThreshold(
    const uint8_t channel_index,
    const uint32_t threshold);
```

```
// 配置 DZONE_TH
void PWM_SetDiedZoneThreshold(
    const uint8_t channel_index,
    const uint32_t threshold);
```

```
// 配置 HIGH_TH
void PWM_SetHighThreshold(
    const uint8_t channel_index,
    const uint8_t multi_duty_index, // 对于 ING916XX，此参数无效
    const uint32_t threshold);
```

各门限值最大支持 0xFFFFF，共 20 个比特。

13.3.4 输出控制

```
// 掩膜控制
void PWM_SetMask(
    const uint8_t channel_index,    // 通道号
    const uint8_t mask_a,           // A 路掩膜
    const uint8_t mask_b            // B 路掩膜
    );
```

```
// 配置停机输出值
void PWM_HaltCtrlCfg(
    const uint8_t channel_index,    // 通道号
    const uint8_t out_a,            // A 路停机输出值
    const uint8_t out_b             // B 路停机输出值
    );
```

```
// 反相
void PWM_SetInvertOutput(
    const uint8_t channel_index,    // 通道号
    const uint8_t inv_a,            // A 路是否反相
    const uint8_t inv_b             // B 路是否反相
    );
```

13.3.5 综合示例

下面的例子将 channel_index 通道配置成输出频率为 frequency、占空比为 (on_duty)% 的方波，涉及 3 个关键参数：

- 生成这种最简单的 PWM 信号需要的模式为 UP_WITHOUT_DIED_ZONE；
- PERA_TH 控制输出信号的频率，设置为 PWM_CLOCK_FREQ / frequency；
- HIGH_TH 控制信号的占空比，设置为 PERA_TH * (100 - on_duty) %

```
void PWM_SetupSimple(
    const uint8_t channel_index,
    const uint32_t frequency,
    const uint16_t on_duty)
{
    uint32_t pera = PWM_CLOCK_FREQ / frequency;
    uint32_t high = pera > 1000 ?
          pera / 100 * (100 - on_duty)
        : pera * (100 - on_duty) / 100;
    PWM_HaltCtrlEnable(channel_index, 1);
    PWM_Enable(channel_index, 0);
    PWM_SetPeraThreshold(channel_index, pera);
    PWM_SetHighThreshold(channel_index, 0, high);
    PWM_SetMode(channel_index, PWM_WORK_MODE_UP_WITHOUT_DIED_ZONE);
    PWM_SetMask(channel_index, 0, 0);
    PWM_Enable(channel_index, 1);
    PWM_HaltCtrlEnable(channel_index, 0);
}
```

13.3.6 使用 DMA 实时更新配置

使用 DMA 能够实时更新配置（相当于工作在 UP_WITHOUT_DIED_ZONE，但是每个循环使用不同的参数）：每当 PWM 计数器计完一圈回零时，自动使用来自 DMA 的数据更新配置。这些数据以 2 个 uint32_t 为一组，依次表示 HIGH_TH 和 PERA_TH。

```
void PWM_DmaEnable(
    const uint8_t channel_index, // 通道号
    uint8_t trig_cfg,            // DMA 请求触发门限
    uint8_t enable               // 使能
    );
```

当 PWM 内部 FIFO 数据少于 `trig_cfg`，PWM 请求 DMA 传输数据。PWM FIFO 深度为 4（指可以存储 4 组 PWM 配置），所以 `trig_cfg` 的取值范围为 1..4。

13.4 PCAP 使用说明

13.4.1 配置 PCAP 模式

要启用 PCAP 模式，需要 5 个步骤：

1. 关闭整个模块的时钟（参考SYSCTRL）
2. 使用 `PCAP_Enable` 使能 PCAP 模式

```
void PCAP_Enable(
    const uint8_t channel_index     // 通道号
);
```

3. 使用 `PCAP_EnableEvents` 选择要检测的事件

```
void PCAP_EnableEvents(
    const uint8_t channel_index,
    uint8_t events_on_0,
    uint8_t events_on_1);
```

`events` 为下面两个事件的组合：

```
enum PCAP_PULSE_EVENT
{
    PCAP_PULSE_RISING_EDGE  = 0x1,
    PCAP_PULSE_FALLING_EDGE = 0x2,
};
```

比如在通道 1 的 A 路输入上同时检测、上报上升沿和下降沿：

```
PCAP_EnableEvents(1,
    PCAP_PULSE_RISING_EDGE
    | PCAP_PULSE_FALLING_EDGE,
    ...);
```

4. 打开整个模块的时钟（参考SYSCTRL）

5. 配置 DMA 传输

   当 PCAP 通道 FIFO 内存储的数据多于 `trig_cfg`，请求 DMA 传输数据。`trig_cfg` 的取值范围为 0..4。

```
void PCAP_DmaEnable(
    const uint8_t channel_index, // 通道号
    uint8_t trig_cfg,            // DMA 请求触发门限
    uint8_t enable               // 使能
    );
```

6. 使能计数器

```
void PCAP_CounterEnable(
    uint8_t enable              // 使能 (1)/禁用 (0)
    );
```

13.4.2 读取计数器

```
uint32_t PCAP_ReadCounter(void);
```

第十四章　QDEC 简介

QDEC 全称 Quadrature Decoder，即正交解码器。

其作用是用来解码来自旋转编码器的脉冲序列，以提供外部设备运动的步长和方向。

14.1　功能描述

14.1.1　特点

- 可配置时钟
- 支持过滤器
- 支持 APB 总线
- 支持 DMA

14.1.2　正转和反转

QDEC 是通过采集到 phase_a、phase_b 相邻两次的数值变化来判断外设的运动方向。

顺时针采集数据如图所示：

逆时针采集数据如图所示：

在 QDEC 数据上，如果引脚配置和连接正确，顺时针转动则采集到的数据逐渐增大，逆时针则数据逐渐减小。

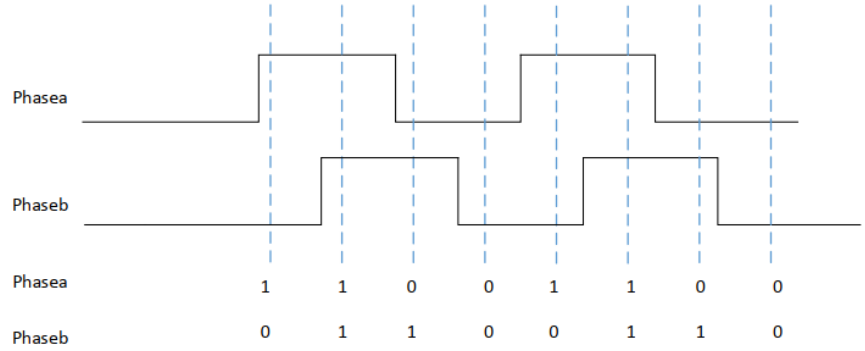


图 14.1: QDEC 顺时针采集数据

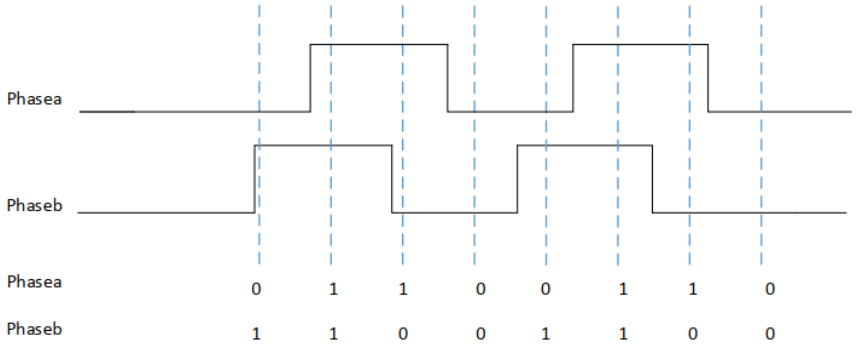


图 14.2: QDEC 逆时针采集数据

14.2 使用方法

14.2.1 方法概述

方法概述为：GPIO 选配，时钟配置，QDEC 参数配置以及数据处理。

14.2.1.1 GPIO 选择

驱动接口：PINCTRL_SelQDECIn

QDEC 的 GPIO 选择，请参考《ING91682X_BLE5.3_ 芯片数据手册》中的 “IO 引脚控制器 Pin Controller” 一节

对 phase_a、phase_b 选定要配置的 GPIO 口，并调用 PINCTRL_SelQDECIn 接口进行配置。

14.2.1.2 时钟配置

驱动接口：SYSCTRL_SelectQDECClk

当前 QDEC 可以选择使用的时钟源为 HCLK 时钟或者 sclk_slow 时钟

对所选用的时钟源还需要进行一次分频，分频系数范围为 1-1023，默认值为 2

这里需要特别注意的是：**请务必配置 pclk 时钟频率不大于 qdec 时钟源频率**

注意：如果配置 pclk 频率大于 qdec 时钟源频率，会出现 qdec 参数配置失败从而不能正常工作的现象。

为了方便开发者使用，可以直接调用下面提供的接口来配置 pclk 时钟符合上述要求：

```
static void QDEC_PclkCfg(void)
{
    uint32_t hclk = SYSCTRL_GetHClk();
    uint32_t qdecClk = SYSCTRL_GetClk(SYSCTRL_ITEM_APB_QDEC);
    uint8_t div = hclk / qdecClk;
    if (hclk % qdecClk)
        div++;
    if (!(div >> 4))
        SYSCTRL_SetPClkDiv(div);
}
```

开发者可以将以上代码拷贝到程序里，在配置 qdec 之前调用即可。

14.2.1.3 QDEC 参数配置

驱动接口：QDEC_EnableQdecDiv、QDEC_QdecCfg

共有 3 个参数需要配置：qdec_clk_div、filter 和 miss

其含义分别如下：

- **qdec_clk_div**：用于控制 qdec 结果上报频率。即多少个时钟周期上报一次采样结果
- **filter**：用于过滤 **filter× 时钟周期**时长以内的毛刺

- **miss**：用于控制 qdec 可以自动补偿的最大 miss 结果数。例如由于滚轮转动过快，导致两次采样中变换了不止一个结果，则此时会自动补偿最多 miss 个结果。

注意：对于 miss 值的配置需要格外注意，miss 值的设置主要考虑到可能由于转动速度过快导致有数据丢失的情况，但此补偿机制容易受设备信号质量影响。对于信号质量较差的设备，如毛刺较多，则不建议加入 miss，否则可能出现 **“采样数据跳变”** 和 **“换向迟钝”** 的问题。对于此类设备，此处建议采用较大工作时钟，不加入 miss 进行采样。测试证明此种方式也可以有较为出色的采样效果。

对于 miss 值配置此处建议先加入较小的 miss 值（如 miss=1）测试效果，如果有数据跳变或者换向迟钝则采用较大工作时钟（如 HCLK 时钟），不加 miss 进行采样。

开发者如果在调试 qdec 过程中出现数据跳变和换向迟钝的问题，建议进行以下几方面尝试：

1. 如选用 sclk_slow 作为时钟源，检查是否有配置 pclk 频率小于 qdec 工作频率

2. 改用较小工作时钟

3. 采样较小的 miss 值（如 miss=1）和较大的 filter 值

4. 采用较大工作时钟（如 HCLK 时钟），不加 miss 进行采样

如果偶尔有较小的数据跳变，如 5 以内，则属于正常情况。

14.2.2 注意点

- phase_a、phase_b 引脚配置注意区分正反，交换引脚则得到相反的转向
- 配置 pclk 时钟可能会影响其他外设，请参考 916 时钟树进行正确配置
- qdec 采样快慢和时钟正相关，和 qdec_clk_div 大小无关，qdec_clk_div 只控制对结果的上报频率
- qdec 上报结果会同时触发 qdec 中断或者 DMA_REQ，qdec_clk_div 设置过低会占用较多 CPU 资源
- filter 建议选择较大值，受毛刺影响较小，稳定性较好

14.3 编程指南

14.3.1 驱动接口

- QDEC_QdecCfg：qdec 标准配置接口
- QDEC_EnableQdecDiv：qdec_clk_div 设置使能接口
- QDEC_ChannelEnable：qdec 通道使能接口
- QDEC_GetData：qdec 获取数据接口
- QDEC_GetDirection：qdec 获取转向接口
- QDEC_Reset：qdec 复位接口

14.3.2 代码示例

下面一段代码展示了 qdec 全部配置并循环读数：

```
static void QDEC_PclkCfg(void)
{
    uint32_t hclk = SYSCTRL_GetHClk();
    uint32_t qdecClk = SYSCTRL_GetClk(SYSCTRL_ITEM_APB_QDEC);
    uint8_t div = hclk / qdecClk;
    if (hclk % qdecClk)
        div++;
    if (!(div >> 4))
        SYSCTRL_SetPClkDiv(div);
}
void test(void)
{
    // setup qdec
    SYSCTRL_ClearClkGate(SYSCTRL_ITEM_APB_PinCtrl |
                         SYSCTRL_ITEM_APB_QDEC);
    QDEC_PclkCfg();    // set pclk not bigger than sclk_slow
```

```
    PINCTRL_SelQDECIn(16, 17);    // set GPIO16=phase_a, GPIO17=phase_b
    SYSCTRL_ReleaseBlock(SYSCTRL_ITEM_APB_PinCtrl |
                         SYSCTRL_ITEM_APB_QDEC);

    SYSCTRL_SelectQDECClk(SYSCTRL_CLK_SLOW, 100);
    QDEC_EnableQdecDiv(QDEC_DIV_1024);
    QDEC_QdecCfg(50, 1);
    QDEC_ChannelEnable(1);

    // print qdec data and direction when rotate the mouse wheel manually
    uint16_t preData = 0;
    uint16_t data = 0;
    uint8_t dir;
    while(1) {
        data = QDEC_GetData();
        dir = QDEC_GetDirection();
        if (data != preData) {
            if (dir) {
                printf("data: %d, %s\n", data, "anticlockwise");
            } else {
                printf("data: %d, %s\n", data, "clockwise");
            }
        }
        preData = data;
    }
}
```

当手动转动鼠标滚轮时，会打印出收到的 qdec 数据和转向。

qdec 的详细使用请参考 HID mouse 例程。

第十五章　实时时钟（RTC）

15.1　功能描述

实时时钟是一个独立的定时器。RTC 模块拥有一组连续计数的计数器，在相应的软件配置下，可提供时钟日历的功能。修改计数器的值可以重新设置系统当前的时间和日期。

15.2　使用说明

15.2.1　RTC 使能

使用 `RTC_Enable` 使能 RTC。

```
void RTC_Enable(uint8_t enable);
```

15.2.2　获取当前日期

使用 `RTC_GetTime` 获取当前时间（包括时分秒）。

```
uint16_t RTC_GetTime(
  uint8_t *hour,
  uint8_t *minute,
  uint8_t *second
  );
```

15.2.3 修改日期

使用 RTC_ModifyTime 修改当前时间。

```
void RTC_ModifyTime(
  uint16_t day,
  uint8_t hour,
  uint8_t minute,
  uint8_t second
  );
```

15.2.4 配置闹钟

使用 RTC_ConfigAlarm

```
void RTC_ConfigAlarm(
  uint8_t hour,
  uint8_t minute,
  uint8_t second
  );
```

15.2.5 配置中断请求

使用 RTC_EnableIRQ 配置并使能 RTC 中断请求。

```
void RTC_EnableIRQ(uint32_t mask);
```

其中的 mask 为 RTC 中断类型，一共有六种：

```
typedef enum
{
    RTC_IRQ_ALARM = 0x04,
    RTC_IRQ_DAY = 0x08,
    RTC_IRQ_HOUR = 0x10,
    RTC_IRQ_MINUTE = 0x20,
    RTC_IRQ_SECOND = 0x40,
    RTC_IRQ_HALF_SECOND = 0x80,
} rtc_irq_t;
```

- 例如将 RTC 设置为 alarm 中断

```
RTC_EnableIRQ(RTC_IRQ_ALARM);
```

- 例如将 RTC 设置为 day 中断

```
RTC_EnableIRQ(RTC_IRQ_DAY);
```

- 例如将 RTC 设置为 hour 中断

```
RTC_EnableIRQ(RTC_IRQ_HOUR);
```

- 例如将 RTC 设置为 minute 中断

```
RTC_EnableIRQ(RTC_IRQ_MINUTE);
```

- 例如将 RTC 设置为 second 中断

```
RTC_EnableIRQ(RTC_IRQ_SECOND);
```

- 例如将 RTC 设置为 half-second 中断

```
RTC_EnableIRQ(RTC_IRQ_HALF_SECOND);
```

15.2.6 获取当前中断状态

使用 RTC_GetIntState 获取当前 RTC 的中断状态。

```
uint32_t RTC_GetIntState(void);
```

15.2.7 清除中断

使用 RTC_ClearIntState 清除当前 RTC 的中断状态。

```
void RTC_ClearIntState(uint32_t state);
```

15.2.8 处理中断状态

用 RTC_GetIntState 获取 RTC 上的中断触发，返回非 0 值表示 RTC 上产生了中断请求；RTC 产生中断后，需要消除中断状态方可再次触发。利用 RTC_ClearIntState 可清除 RTC 的中断触发状态。

第十六章　SPI 功能概述

- 两个 SPI 模块
- 支持 SPI 主 & 从模式
- 支持 Quad SPI，可以从外挂 Flash 执行代码
- 独立的 RX&TX FIFO，深度为 8 个 word
- 支持 DMA

16.1　SPI 使用说明

以下场景中均以 SPI1 为例，如果需要 SPI0 则可以根据情况修改

16.2　场景 1：只读只写不带 DMA

其中 SPI 主配置为只写模式，SPI 从配置为只读模式，CPU 操作读写，没有使用 DMA 配置之前需要决定使用的 GPIO，如果是普通模式，则不需要 SPI_MIC_WP 和 SPI_MIC_HOLD

```
#define SPI_MIC_CLK         GIO_GPIO_10
#define SPI_MIC_MOSI        GIO_GPIO_11
#define SPI_MIC_MISO        GIO_GPIO_12
#define SPI_MIC_CS          GIO_GPIO_13
#define SPI_MIC_WP          GIO_GPIO_14
#define SPI_MIC_HOLD        GIO_GPIO_15
```

16.2.1 SPI 主配置

16.2.1.1 接口配置

```
static void setup_peripherals_spi_pin(void)
{
    // 打开 SPI 模块时钟
    // 此处是以 SPI1 为例，如果是 SPI0 则需要更改
    // 根据使用的 GPIO，选择打开 GPIO0 或者 GPIO1 的时钟
    SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_SPI1)
                                | (1 << SYSCTRL_ITEM_APB_SysCtrl)
                                | (1 << SYSCTRL_ITEM_APB_PinCtrl)
                                | (1 << SYSCTRL_ITEM_APB_GPIO1)
                                | (1 << SYSCTRL_ITEM_APB_GPIO0));

    // 设置 IO MUX，将 GPIO 映射成 SPI 功能，不需要的 pin 可以使用 IO_NOT_A_PIN 替代
    PINCTRL_SelSpiIn(SPI_PORT_1, SPI_MIC_CLK, SPI_MIC_CS, SPI_MIC_HOLD, SPI_MIC_WP, SPI_M
    PINCTRL_SetPadMux(SPI_MIC_CLK, IO_SOURCE_SPI1_CLK_OUT);
    PINCTRL_SetPadMux(SPI_MIC_CS, IO_SOURCE_SPI1_CSN_OUT);
    PINCTRL_SetPadMux(SPI_MIC_MOSI, IO_SOURCE_SPI1_MOSI_OUT);

    // 设置 SPI 的中断
    platform_set_irq_callback(PLATFORM_CB_IRQ_APBSPI, peripherals_spi_isr, NULL);
```

16.2.1.2 SPI 模块初始化

常用设置项用注释标出，详细定义请参考 “peripheral_ssp.h”

```
// 示例，每次传输大小是 8 个 word
#define DATA_LEN (SPI_FIFO_DEPTH)
static void setup_peripherals_spi_module(void)
{
    apSSP_sDeviceControlBlock pParam;
```

```
    pParam.eSclkDiv = SPI_INTERFACETIMINGSCLKDIV_DEFAULT_2M;// SPI 时钟设置
    pParam.eSCLKPolarity = SPI_CPOL_SCLK_LOW_IN_IDLE_STATES;// SPI 模式设置
    pParam.eSCLKPhase = SPI_CPHA_ODD_SCLK_EDGES;// SPI 模式设置
    pParam.eLsbMsbOrder = SPI_LSB_MOST_SIGNIFICANT_BIT_FIRST;
    pParam.eDataSize = SPI_DATALEN_32_BITS;// SPI 每个传输单位的大小
    pParam.eMasterSlaveMode = SPI_SLVMODE_MASTER_MODE;// SPI 主模式
    pParam.eReadWriteMode = SPI_TRANSMODE_WRITE_ONLY;// SPI 只写
    pParam.eQuadMode = SPI_DUALQUAD_REGULAR_MODE;
    pParam.eWriteTransCnt = DATA_LEN;// SPI 每次传输多少个单位（每个单位的大小是 pPar
    pParam.eReadTransCnt = DATA_LEN;// SPI 每次接收多少个单位（每个单位的大小是 pPara
    pParam.eAddrEn = SPI_ADDREN_DISABLE;
    pParam.eCmdEn = SPI_CMDEN_DISABLE;
    pParam.RxThres = DATA_LEN/2;
    pParam.TxThres = DATA_LEN/2;
    pParam.SlaveDataOnly = SPI_SLVDATAONLY_ENABLE;
    pParam.eAddrLen = SPI_ADDRLEN_1_BYTE;
    pParam.eInterruptMask = (1 << bsSPI_INTREN_ENDINTEN);// 打开 SPI 中断（传输结束后

    apSSP_DeviceParametersSet(APB_SSP1, &pParam);
}
```

16.2.1.3 SPI 中断

```
// SPI ENDINT 中断触发标志传输结束，清除中断状态
static uint32_t peripherals_spi_isr(void *user_data)
{
  uint32_t stat = apSSP_GetIntRawStatus(APB_SSP1);

  if(stat & (1 << bsSPI_INTREN_ENDINTEN))
  {
    apSSP_ClearIntStatus(APB_SSP1, 1 << bsSPI_INTREN_ENDINTEN);
  }
```

```
}
```

16.2.1.4 SPI 发送数据

```
uint32_t write_data[DATA_LEN];//数据大小等于 pParam.eDataSize，数组大小等于 pParam.eWrite
void peripherals_spi_send_data(void)
{
  // 写入命令，触发 SPI 传输
  apSSP_WriteCmd(APB_SSP1, 0x00, 0x00);//trigger transfer

  // 填写数据到 TX FIFO，这个例子中 DATA_LEN 等于 FIFO 的深度（8），如果大于 8，可以分为多
  for(i = 0; i < DATA_LEN; i++)
  {
    apSSP_WriteFIFO(APB_SSP1, write_data[i]);
  }

  // 等待发送结束
  while(apSSP_GetSPIActiveStatus(APB_SSP1));
}
```

16.2.1.5 使用流程

- 设置 GPIO，setup_peripherals_spi_pin()
- 初始化 SPI，setup_peripherals_spi_module()
- 在需要时候发送 SPI 数据，peripherals_spi_send_data()
- 检查中断状态

16.2.2 SPI 从配置

16.2.2.1 接口配置

```
static void setup_peripherals_spi_pin(void)
{
    // 打开 SPI 模块时钟
    // 此处是以 SPI1 为例，如果是 SPI0 则需要更改
    // 根据使用的 GPIO，选择打开 GPIO0 或者 GPIO1 的时钟
    SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_SPI1)
                                | (1 << SYSCTRL_ITEM_APB_SysCtrl)
                                | (1 << SYSCTRL_ITEM_APB_PinCtrl)
                                | (1 << SYSCTRL_ITEM_APB_GPIO1)
                                | (1 << SYSCTRL_ITEM_APB_GPIO0));

    // 设置 IO MUX，将 GPIO 映射成 SPI 功能，不需要的 pin 可以使用 IO_NOT_A_PIN 替代
    PINCTRL_Pull(IO_SOURCE_SPI1_CLK_IN,PINCTRL_PULL_DOWN);
    PINCTRL_Pull(IO_SOURCE_SPI1_CSN_IN,PINCTRL_PULL_UP);// CS 需要默认上拉
    PINCTRL_SelSpiIn(SPI_PORT_1, SPI_MIC_CLK, SPI_MIC_CS, SPI_MIC_HOLD, SPI_MIC_WP, S
    PINCTRL_SetPadMux(SPI_MIC_CLK, IO_SOURCE_SPI1_CLK_OUT);
    PINCTRL_SetPadMux(SPI_MIC_MISO, IO_SOURCE_SPI1_MISO_OUT);

    // 设置 SPI 的中断
    platform_set_irq_callback(PLATFORM_CB_IRQ_APBSPI, peripherals_spi_isr, NULL);
```

16.2.2.2 SPI 模块初始化

常用设置项用注释标出，详细定义请参考 “peripheral_ssp.h”

```
// 示例，每次传输大小是 8 个 word
#define DATA_LEN (SPI_FIFO_DEPTH)
static void setup_peripherals_spi_module(void)
{
```

```
    apSSP_sDeviceControlBlock pParam;
    pParam.eSclkDiv = SPI_INTERFACETIMINGSCLKDIV_DEFAULT_2M;// SPI 时钟设置
    pParam.eSCLKPolarity = SPI_CPOL_SCLK_LOW_IN_IDLE_STATES;// SPI 模式设置
    pParam.eSCLKPhase = SPI_CPHA_ODD_SCLK_EDGES;// SPI 模式设置
    pParam.eLsbMsbOrder = SPI_LSB_MOST_SIGNIFICANT_BIT_FIRST;
    pParam.eDataSize = SPI_DATALEN_32_BITS;// SPI 每个传输单位的大小
    pParam.eMasterSlaveMode = SPI_SLVMODE_SLAVE_MODE;// SPI 从模式
    pParam.eReadWriteMode = SPI_TRANSMODE_READ_ONLY;// SPI 只读
    pParam.eQuadMode = SPI_DUALQUAD_REGULAR_MODE;
    pParam.eWriteTransCnt = DATA_LEN;// SPI 每次传输多少个单位（每个单位的大小是 pParam.e
    pParam.eReadTransCnt = DATA_LEN;// SPI 每次接收多少个单位（每个单位的大小是 pParam.eD
    pParam.eAddrEn = SPI_ADDREN_DISABLE;
    pParam.eCmdEn = SPI_CMDEN_DISABLE;
    pParam.RxThres = DATA_LEN/2;
    pParam.TxThres = DATA_LEN/2;
    pParam.SlaveDataOnly = SPI_SLVDATAONLY_ENABLE;
    pParam.eAddrLen = SPI_ADDRLEN_1_BYTE;
    pParam.eInterruptMask = (1 << bsSPI_INTREN_ENDINTEN);// 打开 SPI 中断（传输结束后触发

    apSSP_DeviceParametersSet(APB_SSP1, &pParam);
}
```

16.2.2.3 SPI 接收数据

```
uint32_t read_data[DATA_LEN];//数据大小等于 pParam.eDataSize，数组大小等于 pParam.eReadTr
static uint32_t peripherals_spi_isr(void *user_data)
{
  uint32_t stat = apSSP_GetIntRawStatus(APB_SSP1), i;
  if(stat & (1 << bsSPI_INTREN_ENDINTEN))
  {
    /* check if rx fifo still have some left data */
    // 检查当前 RX FIFO 中有效值的个数，根据个数读取 RX FIFO
```

```
    uint32_t num = apSSP_GetDataNumInRxFifo(APB_SSP1);
    for(i = 0; i < num; i++)
    {
      apSSP_ReadFIFO(APB_SSP1, &read_data[i]);
    }

    apSSP_ClearIntStatus(APB_SSP1, 1 << bsSPI_INTREN_ENDINTEN);

  }
}
```

16.2.2.4 使用流程

- 设置 GPIO，setup_peripherals_spi_pin()
- 初始化 SPI，setup_peripherals_spi_module()
- 观察 SPI 中断，中断触发代表当前传输结束

16.3 场景 2：只读只写并且使用 DMA

其中 SPI 主配置为只写模式，SPI 从配置为只读模式，同时使用 DMA 进行读写配置之前需要决定使用的 GPIO，如果是普通模式，则不需要 SPI_MIC_WP 和 SPI_MIC_HOLD

```
#define SPI_MIC_CLK         GIO_GPIO_10
#define SPI_MIC_MOSI        GIO_GPIO_11
#define SPI_MIC_MISO        GIO_GPIO_12
#define SPI_MIC_CS          GIO_GPIO_13
#define SPI_MIC_WP          GIO_GPIO_14
#define SPI_MIC_HOLD        GIO_GPIO_15
```

16.3.1 SPI 主配置

16.3.1.1 接口配置

```
static void setup_peripherals_spi_pin(void)
{
    // 打开 SPI 模块时钟
    // 此处是以 SPI1 为例，如果是 SPI0 则需要更改
    // 根据使用的 GPIO，选择打开 GPIO0 或者 GPIO1 的时钟
    SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_SPI1)
                                | (1 << SYSCTRL_ITEM_APB_SysCtrl)
                                | (1 << SYSCTRL_ITEM_APB_PinCtrl)
                                | (1 << SYSCTRL_ITEM_APB_GPIO1)
                                | (1 << SYSCTRL_ITEM_APB_GPIO0));

    // 设置 IO MUX，将 GPIO 映射成 SPI 功能，不需要的 pin 可以使用 IO_NOT_A_PIN 替代
    PINCTRL_SelSpiIn(SPI_PORT_1, SPI_MIC_CLK, SPI_MIC_CS, SPI_MIC_HOLD, SPI_MIC_WP, SPI_M
    PINCTRL_SetPadMux(SPI_MIC_CLK, IO_SOURCE_SPI1_CLK_OUT);
    PINCTRL_SetPadMux(SPI_MIC_CS, IO_SOURCE_SPI1_CSN_OUT);
    PINCTRL_SetPadMux(SPI_MIC_MOSI, IO_SOURCE_SPI1_MOSI_OUT);

    // 设置 SPI 的中断
    platform_set_irq_callback(PLATFORM_CB_IRQ_APBSPI, peripherals_spi_isr, NULL);
```

16.3.1.2 SPI 模块初始化

常用设置项用注释标出，详细定义请参考 “peripheral_ssp.h”

```
// 示例，每次传输大小是 8 个 word
#define DATA_LEN (SPI_FIFO_DEPTH)
static void setup_peripherals_spi_module(void)
{
    apSSP_sDeviceControlBlock pParam;
```

```
    pParam.eSclkDiv = SPI_INTERFACETIMINGSCLKDIV_DEFAULT_2M;// SPI 时钟设置
    pParam.eSCLKPolarity = SPI_CPOL_SCLK_LOW_IN_IDLE_STATES;// SPI 模式设置
    pParam.eSCLKPhase = SPI_CPHA_ODD_SCLK_EDGES;// SPI 模式设置
    pParam.eLsbMsbOrder = SPI_LSB_MOST_SIGNIFICANT_BIT_FIRST;
    pParam.eDataSize = SPI_DATALEN_32_BITS;// SPI 每个传输单位的大小
    pParam.eMasterSlaveMode = SPI_SLVMODE_MASTER_MODE;// SPI 主模式
    pParam.eReadWriteMode = SPI_TRANSMODE_WRITE_ONLY;// SPI 只写
    pParam.eQuadMode = SPI_DUALQUAD_REGULAR_MODE;
    pParam.eWriteTransCnt = DATA_LEN;// SPI 每次传输多少个单位（每个单位的大小是 pPar
    pParam.eReadTransCnt = DATA_LEN;// SPI 每次接收多少个单位（每个单位的大小是 pPara
    pParam.eAddrEn = SPI_ADDREN_DISABLE;
    pParam.eCmdEn = SPI_CMDEN_DISABLE;
    pParam.RxThres = DATA_LEN/2;
    pParam.TxThres = DATA_LEN/2;
    pParam.SlaveDataOnly = SPI_SLVDATAONLY_ENABLE;
    pParam.eAddrLen = SPI_ADDRLEN_1_BYTE;
    pParam.eInterruptMask = (1 << bsSPI_INTREN_ENDINTEN);// 打开 SPI 中断（传输结束后

    apSSP_DeviceParametersSet(APB_SSP1, &pParam);

}
```

16.3.1.3 SPI DMA 初始化

// 初始化 DMA 模块

```
static void setup_peripherals_dma_module(void)
{
    SYSCTRL_ClearClkGateMulti(1 << SYSCTRL_ClkGate_APB_DMA);
    DMA_Reset(1);
    DMA_Reset(0);
}
```

16.3.1.4 SPI DMA 设置

// 此处是以 SPI1 为例

```
void peripherals_spi_dma_to_txfifo(int channel_id, void *src, int size)
{
    DMA_Descriptor descriptor __attribute__((aligned (8)));

    descriptor.Next = (DMA_Descriptor *)0;
    DMA_PrepareMem2Peripheral(&descriptor,SYSCTRL_DMA_SPI1_TX,src,size,DMA_ADDRESS_INC,0)

    DMA_EnableChannel(channel_id, &descriptor);
}
```

16.3.1.5 SPI 中断

```
// SPI ENDINT 中断触发标志传输结束，清除中断状态
static uint32_t peripherals_spi_isr(void *user_data)
{
  uint32_t stat = apSSP_GetIntRawStatus(APB_SSP1);

  if(stat & (1 << bsSPI_INTREN_ENDINTEN))
  {
    apSSP_ClearIntStatus(APB_SSP1, 1 << bsSPI_INTREN_ENDINTEN);
  }
}
```

16.3.1.6 SPI 发送数据

```
uint32_t write_data[DATA_LEN];//数据大小等于 pParam.eDataSize，数组大小等于 pParam.eW

void peripherals_spi_send_data(void)
{
  // 首先需要打开 SPI 模块中的 DMA 功能
  apSSP_SetTxDmaEn(APB_SSP1,1);
  // 初始化中已经设置了 pParam.eWriteTransCnt，如果需要调整则可以调用这个 API
  apSSP_SetTransferControlWrTranCnt(APB_SSP1,DATA_LEN);

  // DMA 共有 8 个 channel
  #define SPI_DMA_TX_CHANNEL   (0)//DMA channel 0
  // 配置 DMA，指向需要发送的数据
  peripherals_spi_dma_to_txfifo(SPI_DMA_TX_CHANNEL, write_data, sizeof(write_data));

  // 写入命令，触发 SPI 传输
  apSSP_WriteCmd(APB_SSP1, 0x00, 0x00);//trigger transfer

  // 等待发送结束
  while(apSSP_GetSPIActiveStatus(APB_SSP1));

  // 关闭 SPI 模块中的 DMA 功能
  apSSP_SetTxDmaEn(APB_SSP1,0);
}
```

16.3.1.7 使用流程

- 设置 GPIO，setup_peripherals_spi_pin()
- 初始化 SPI，setup_peripherals_spi_module()
- 初始化 DMA，setup_peripherals_dma_module()
- 在需要时候发送 SPI 数据，peripherals_spi_send_data()
- 检查中断状态

16.3.2 SPI 从配置

16.3.2.1 接口配置

```
static void setup_peripherals_spi_pin(void)
{
    // 打开 SPI 模块时钟
    // 此处是以 SPI1 为例，如果是 SPI0 则需要更改
    // 根据使用的 GPIO，选择打开 GPIO0 或者 GPIO1 的时钟
    SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_SPI1)
                                | (1 << SYSCTRL_ITEM_APB_SysCtrl)
                                | (1 << SYSCTRL_ITEM_APB_PinCtrl)
                                | (1 << SYSCTRL_ITEM_APB_GPIO1)
                                | (1 << SYSCTRL_ITEM_APB_GPIO0));

    // 设置 IO MUX，将 GPIO 映射成 SPI 功能，不需要的 pin 可以使用 IO_NOT_A_PIN 替代
    PINCTRL_Pull(IO_SOURCE_SPI1_CLK_IN,PINCTRL_PULL_DOWN);
    PINCTRL_Pull(IO_SOURCE_SPI1_CSN_IN,PINCTRL_PULL_UP);// CS 需要默认上拉
    PINCTRL_SelSpiIn(SPI_PORT_1, SPI_MIC_CLK, SPI_MIC_CS, SPI_MIC_HOLD, SPI_MIC_WP, SPI_M
    PINCTRL_SetPadMux(SPI_MIC_CLK, IO_SOURCE_SPI1_CLK_OUT);
    PINCTRL_SetPadMux(SPI_MIC_MISO, IO_SOURCE_SPI1_MISO_OUT);

    // 设置 SPI 的中断
    platform_set_irq_callback(PLATFORM_CB_IRQ_APBSPI, peripherals_spi_isr, NULL);
```

16.3.2.2 SPI 模块初始化

常用设置项用注释标出，详细定义请参考 “peripheral_ssp.h”

```
// 示例，每次传输大小是 8 个 word
#define DATA_LEN (SPI_FIFO_DEPTH)
static void setup_peripherals_spi_module(void)
{
```

```
    apSSP_sDeviceControlBlock pParam;
    pParam.eSclkDiv = SPI_INTERFACETIMINGSCLKDIV_DEFAULT_2M;// SPI 时钟设置
    pParam.eSCLKPolarity = SPI_CPOL_SCLK_LOW_IN_IDLE_STATES;// SPI 模式设置
    pParam.eSCLKPhase = SPI_CPHA_ODD_SCLK_EDGES;// SPI 模式设置
    pParam.eLsbMsbOrder = SPI_LSB_MOST_SIGNIFICANT_BIT_FIRST;
    pParam.eDataSize = SPI_DATALEN_32_BITS;// SPI 每个传输单位的大小
    pParam.eMasterSlaveMode = SPI_SLVMODE_SLAVE_MODE;// SPI 从模式
    pParam.eReadWriteMode = SPI_TRANSMODE_READ_ONLY;// SPI 只读
    pParam.eQuadMode = SPI_DUALQUAD_REGULAR_MODE;
    pParam.eWriteTransCnt = DATA_LEN;// SPI 每次传输多少个单位（每个单位的大小是 pPar
    pParam.eReadTransCnt = DATA_LEN;// SPI 每次接收多少个单位（每个单位的大小是 pPara
    pParam.eAddrEn = SPI_ADDREN_DISABLE;
    pParam.eCmdEn = SPI_CMDEN_DISABLE;
    pParam.RxThres = 0;
    pParam.TxThres = 0;
    pParam.SlaveDataOnly = SPI_SLVDATAONLY_ENABLE;
    pParam.eAddrLen = SPI_ADDRLEN_1_BYTE;
    pParam.eInterruptMask = (1 << bsSPI_INTREN_ENDINTEN);// 打开 SPI 中断（传输结束后

    apSSP_DeviceParametersSet(APB_SSP1, &pParam);
}
```

16.3.2.3 SPI DMA 初始化

// 初始化 DMA 模块

```
static void setup_peripherals_dma_module(void)
{
    SYSCTRL_ClearClkGateMulti(1 << SYSCTRL_ClkGate_APB_DMA);
    DMA_Reset(1);
    DMA_Reset(0);
}
```

16.3.2.4 SPI DMA 设置

// 此处是以 SPI1 为例

```
void peripherals_spi_rxfifo_to_dma(int channel_id, void *dst, int size)
{
    DMA_Descriptor descriptor __attribute__((aligned (8)));

    descriptor.Next = (DMA_Descriptor *)0;
    DMA_PreparePeripheral2Mem(&descriptor,dst,SYSCTRL_DMA_SPI1_RX,size,DMA_ADDRESS_INC,0)

    DMA_EnableChannel(channel_id, &descriptor);
}
```

16.3.2.5 SPI 接收数据

```
uint32_t read_data[DATA_LEN];//数据大小等于 pParam.eDataSize，数组大小等于 pParam.eReadTr
void peripherals_spi_read_data(void)
{
    // 打开 SPI DMA 功能
    apSSP_SetRxDmaEn(APB_SSP1,1);
    // 功能等同于重新设置 pParam.eReadTransCnt，代表一次传输的单位个数
    apSSP_SetTransferControlRdTranCnt(APB_SSP1,DATA_LEN);

    #define SPI_DMA_RX_CHANNEL   (0)//DMA channel 0
    peripherals_spi_rxfifo_to_dma(SPI_DMA_RX_CHANNEL, read_data, sizeof(read_data));
}
```

16.3.2.6 SPI 中断

```
// SPI ENDINT 中断触发标志传输结束，清除中断状态
static uint32_t peripherals_spi_isr(void *user_data)
{
  uint32_t stat = apSSP_GetIntRawStatus(APB_SSP1);

  if(stat & (1 << bsSPI_INTREN_ENDINTEN))
  {
    peripherals_spi_read_data();
    apSSP_ClearIntStatus(APB_SSP1, 1 << bsSPI_INTREN_ENDINTEN);
  }
}
```

16.3.2.7 使用流程

- 设置 GPIO，setup_peripherals_spi_pin()
- 初始化 SPI，setup_peripherals_spi_module()
- 初始化 DMA，setup_peripherals_dma_module()
- 设置接收 DMA，peripherals_spi_read_data()
- 观察 SPI 中断，中断触发代表当前接收结束

16.4 场景 3：同时读写不带 DMA

其中 SPI 主从都配置为同时读写模式，CPU 操作读写，没有使用 DMA 配置之前需要决定使用的 GPIO，如果是普通模式，则不需要 SPI_MIC_WP 和 SPI_MIC_HOLD

```
#define SPI_MIC_CLK         GIO_GPIO_10
#define SPI_MIC_MOSI        GIO_GPIO_11
#define SPI_MIC_MISO        GIO_GPIO_12
#define SPI_MIC_CS          GIO_GPIO_13
#define SPI_MIC_WP          GIO_GPIO_14
#define SPI_MIC_HOLD        GIO_GPIO_15
```

16.4.1 SPI 主配置

16.4.1.1 接口配置

```
static void setup_peripherals_spi_pin(void)
{
    // 打开 SPI 模块时钟
    // 此处是以 SPI1 为例，如果是 SPI0 则需要更改
    // 根据使用的 GPIO，选择打开 GPIO0 或者 GPIO1 的时钟
    SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_SPI1)
                                | (1 << SYSCTRL_ITEM_APB_SysCtrl)
                                | (1 << SYSCTRL_ITEM_APB_PinCtrl)
                                | (1 << SYSCTRL_ITEM_APB_GPIO1)
                                | (1 << SYSCTRL_ITEM_APB_GPIO0));

    // 设置 IO MUX，将 GPIO 映射成 SPI 功能，不需要的 pin 可以使用 IO_NOT_A_PIN 替代
    PINCTRL_SelSpiIn(SPI_PORT_1, SPI_MIC_CLK, SPI_MIC_CS, SPI_MIC_HOLD, SPI_MIC_WP, SPI_M
    PINCTRL_SetPadMux(SPI_MIC_CLK, IO_SOURCE_SPI1_CLK_OUT);
    PINCTRL_SetPadMux(SPI_MIC_CS, IO_SOURCE_SPI1_CSN_OUT);
    PINCTRL_SetPadMux(SPI_MIC_MOSI, IO_SOURCE_SPI1_MOSI_OUT);

    // 设置 SPI 的中断
    platform_set_irq_callback(PLATFORM_CB_IRQ_APBSPI, peripherals_spi_isr, NULL);
```

16.4.1.2 SPI 模块初始化

常用设置项用注释标出，详细定义请参考 “peripheral_ssp.h”

```
// 示例，每次传输大小是 8 个 word
#define DATA_LEN (SPI_FIFO_DEPTH)
static void setup_peripherals_spi_module(void)
{
    apSSP_sDeviceControlBlock pParam;
```

```
    pParam.eSclkDiv = SPI_INTERFACETIMINGSCLKDIV_DEFAULT_2M;// SPI 时钟设置
    pParam.eSCLKPolarity = SPI_CPOL_SCLK_LOW_IN_IDLE_STATES;// SPI 模式设置
    pParam.eSCLKPhase = SPI_CPHA_ODD_SCLK_EDGES;// SPI 模式设置
    pParam.eLsbMsbOrder = SPI_LSB_MOST_SIGNIFICANT_BIT_FIRST;
    pParam.eDataSize = SPI_DATALEN_32_BITS;// SPI 每个传输单位的大小
    pParam.eMasterSlaveMode = SPI_SLVMODE_MASTER_MODE;// SPI 主模式
    pParam.eReadWriteMode = SPI_TRANSMODE_WRITE_READ_SAME_TIME;// SPI 同时读写
    pParam.eQuadMode = SPI_DUALQUAD_REGULAR_MODE;
    pParam.eWriteTransCnt = DATA_LEN;// SPI 每次传输多少个单位（每个单位的大小是 pPar
    pParam.eReadTransCnt = DATA_LEN;// SPI 每次接收多少个单位（每个单位的大小是 pPara
    pParam.eAddrEn = SPI_ADDREN_DISABLE;
    pParam.eCmdEn = SPI_CMDEN_DISABLE;
    pParam.RxThres = DATA_LEN/2;
    pParam.TxThres = DATA_LEN/2;
    pParam.SlaveDataOnly = SPI_SLVDATAONLY_ENABLE;
    pParam.eAddrLen = SPI_ADDRLEN_1_BYTE;
    pParam.eInterruptMask = (1 << bsSPI_INTREN_ENDINTEN);// 打开 SPI 中断（传输结束后

    apSSP_DeviceParametersSet(APB_SSP1, &pParam);
}
```

16.4.1.3 SPI 中断

```
// SPI ENDINT 中断触发标志传输结束，清除中断状态
static uint32_t peripherals_spi_isr(void *user_data)
{
  uint32_t stat = apSSP_GetIntRawStatus(APB_SSP1);

  if(stat & (1 << bsSPI_INTREN_ENDINTEN))
  {
    apSSP_ClearIntStatus(APB_SSP1, 1 << bsSPI_INTREN_ENDINTEN);
  }
```

```
}
```

16.4.1.4 SPI 发送数据

```
uint32_t write_data[DATA_LEN];//数据大小等于 pParam.eDataSize，数组大小等于 pParam.eWrite
void peripherals_spi_send_data(void)
{
  // 写入命令，触发 SPI 传输
  apSSP_WriteCmd(APB_SSP1, 0x00, 0x00);//trigger transfer

  // 填写数据到 TX FIFO，这个例子中 DATA_LEN 等于 FIFO 的深度（8），如果大于 8，可以分为多
  // 每次发送完 8 个单位，需要读取 RX FIFO 中的数据
  for(i = 0; i < DATA_LEN; i++)
  {
    apSSP_WriteFIFO(APB_SSP1, write_data[i]);
  }

  // 等待发送结束
  while(apSSP_GetSPIActiveStatus(APB_SSP1));

  // 读取当前 RX FIFO 中有效值的个数，然后从 RX FIFO 中读取返回值
  uint32_t num = apSSP_GetDataNumInRxFifo(APB_SSP1);
  for(i = 0; i < num; i++)
  {
    apSSP_ReadFIFO(APB_SSP1, &read_data[i]);
  }
}
```

16.4.1.5 使用流程

- 设置 GPIO，setup_peripherals_spi_pin()
- 初始化 SPI，setup_peripherals_spi_module()

- 在需要时候发送 SPI 数据，peripherals_spi_send_data()
- 检查中断状态

16.4.2 SPI 从配置

16.4.2.1 接口配置

```
static void setup_peripherals_spi_pin(void)
{
    // 打开 SPI 模块时钟
    // 此处是以 SPI1 为例，如果是 SPI0 则需要更改
    // 根据使用的 GPIO，选择打开 GPIO0 或者 GPIO1 的时钟
    SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_SPI1)
                                | (1 << SYSCTRL_ITEM_APB_SysCtrl)
                                | (1 << SYSCTRL_ITEM_APB_PinCtrl)
                                | (1 << SYSCTRL_ITEM_APB_GPIO1)
                                | (1 << SYSCTRL_ITEM_APB_GPIO0));

    // 设置 IO MUX，将 GPIO 映射成 SPI 功能，不需要的 pin 可以使用 IO_NOT_A_PIN 替代
    PINCTRL_Pull(IO_SOURCE_SPI1_CLK_IN,PINCTRL_PULL_DOWN);
    PINCTRL_Pull(IO_SOURCE_SPI1_CSN_IN,PINCTRL_PULL_UP);// CS 需要默认上拉
    PINCTRL_SelSpiIn(SPI_PORT_1, SPI_MIC_CLK, SPI_MIC_CS, SPI_MIC_HOLD, SPI_MIC_WP, S
    PINCTRL_SetPadMux(SPI_MIC_CLK, IO_SOURCE_SPI1_CLK_OUT);
    PINCTRL_SetPadMux(SPI_MIC_MISO, IO_SOURCE_SPI1_MISO_OUT);

    // 设置 SPI 的中断
    platform_set_irq_callback(PLATFORM_CB_IRQ_APBSPI, peripherals_spi_isr, NULL);
```

16.4.2.2 SPI 模块初始化

常用设置项用注释标出，详细定义请参考 “peripheral_ssp.h”

```
// 示例，每次传输大小是 8 个 word
#define DATA_LEN (SPI_FIFO_DEPTH)
static void setup_peripherals_spi_module(void)
{
    apSSP_sDeviceControlBlock pParam;
    pParam.eSclkDiv = SPI_INTERFACETIMINGSCLKDIV_DEFAULT_2M;// SPI 时钟设置
    pParam.eSCLKPolarity = SPI_CPOL_SCLK_LOW_IN_IDLE_STATES;// SPI 模式设置
    pParam.eSCLKPhase = SPI_CPHA_ODD_SCLK_EDGES;// SPI 模式设置
    pParam.eLsbMsbOrder = SPI_LSB_MOST_SIGNIFICANT_BIT_FIRST;
    pParam.eDataSize = SPI_DATALEN_32_BITS;// SPI 每个传输单位的大小
    pParam.eMasterSlaveMode = SPI_SLVMODE_SLAVE_MODE;// SPI 从模式
    pParam.eReadWriteMode = SPI_TRANSMODE_WRITE_READ_SAME_TIME;// SPI 同时读写
    pParam.eQuadMode = SPI_DUALQUAD_REGULAR_MODE;
    pParam.eWriteTransCnt = DATA_LEN;// SPI 每次传输多少个单位（每个单位的大小是 pParam.e
    pParam.eReadTransCnt = DATA_LEN;// SPI 每次接收多少个单位（每个单位的大小是 pParam.eD
    pParam.eAddrEn = SPI_ADDREN_DISABLE;
    pParam.eCmdEn = SPI_CMDEN_DISABLE;
    pParam.RxThres = DATA_LEN/2;
    pParam.TxThres = DATA_LEN/2;
    pParam.SlaveDataOnly = SPI_SLVDATAONLY_ENABLE;
    pParam.eAddrLen = SPI_ADDRLEN_1_BYTE;
    pParam.eInterruptMask = (1 << bsSPI_INTREN_ENDINTEN);// 打开 SPI 中断（传输结束后触发

    apSSP_DeviceParametersSet(APB_SSP1, &pParam);
}
```

16.4.2.3 SPI 接收数据

```
void peripherals_spi_push_data(void)
{
    for(i = 0; i < DATA_LEN; i++)
    {
```

```
      apSSP_WriteFIFO(APB_SSP1, write_data[i]);
    }
}

uint32_t read_data[DATA_LEN];//数据大小等于 pParam.eDataSize，数组大小等于 pParam.eRe
static uint32_t peripherals_spi_isr(void *user_data)
{
  uint32_t stat = apSSP_GetIntRawStatus(APB_SSP1), i;
  if(stat & (1 << bsSPI_INTREN_ENDINTEN))
  {
    /* check if rx fifo still have some left data */
    // 检查当前 RX FIFO 中有效值的个数，根据个数读取 RX FIFO
    uint32_t num = apSSP_GetDataNumInRxFifo(APB_SSP1);
    for(i = 0; i < num; i++)
    {
      apSSP_ReadFIFO(APB_SSP1, &read_data[i]);
    }

    // 根据需要填充下一次发送的 SPI 数据
    peripherals_spi_push_data();

    apSSP_ClearIntStatus(APB_SSP1, 1 << bsSPI_INTREN_ENDINTEN);

  }
}
```

16.4.2.4 使用流程

- 设置 GPIO，setup_peripherals_spi_pin()
- 初始化 SPI，setup_peripherals_spi_module()
- 根据需要填充 TX FIFO，peripherals_spi_push_data()
- 观察 SPI 中断，中断触发代表当前传输结束

16.5 场景 4：同时读写并且使用 DMA

其中 SPI 主从配置为同时读写模式，同时使用 DMA 进行读写配置之前需要决定使用的 GPIO，如果是普通模式，则不需要 SPI_MIC_WP 和 SPI_MIC_HOLD

```
#define SPI_MIC_CLK         GIO_GPIO_10
#define SPI_MIC_MOSI        GIO_GPIO_11
#define SPI_MIC_MISO        GIO_GPIO_12
#define SPI_MIC_CS          GIO_GPIO_13
#define SPI_MIC_WP          GIO_GPIO_14
#define SPI_MIC_HOLD        GIO_GPIO_15

// RX FIFO 和 TX FIFO 使用两个 DMA channel
#define SPI_DMA_TX_CHANNEL   (0)//DMA channel 0
#define SPI_DMA_RX_CHANNEL   (1)//DMA channel 1
```

16.5.1 SPI 主配置

16.5.1.1 接口配置

```
static void setup_peripherals_spi_pin(void)
{
    // 打开 SPI 模块时钟
    // 此处是以 SPI1 为例，如果是 SPI0 则需要更改
    // 根据使用的 GPIO，选择打开 GPIO0 或者 GPIO1 的时钟
    SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_SPI1)
                                | (1 << SYSCTRL_ITEM_APB_SysCtrl)
                                | (1 << SYSCTRL_ITEM_APB_PinCtrl)
                                | (1 << SYSCTRL_ITEM_APB_GPIO1)
                                | (1 << SYSCTRL_ITEM_APB_GPIO0));

    // 设置 IO MUX，将 GPIO 映射成 SPI 功能，不需要的 pin 可以使用 IO_NOT_A_PIN 替代
```

```
    PINCTRL_SelSpiIn(SPI_PORT_1, SPI_MIC_CLK, SPI_MIC_CS, SPI_MIC_HOLD, SPI_MIC_WP, S
    PINCTRL_SetPadMux(SPI_MIC_CLK, IO_SOURCE_SPI1_CLK_OUT);
    PINCTRL_SetPadMux(SPI_MIC_CS, IO_SOURCE_SPI1_CSN_OUT);
    PINCTRL_SetPadMux(SPI_MIC_MOSI, IO_SOURCE_SPI1_MOSI_OUT);

    // 设置 SPI 的中断
    platform_set_irq_callback(PLATFORM_CB_IRQ_APBSPI, peripherals_spi_isr, NULL);
```

16.5.1.2 SPI 模块初始化

常用设置项用注释标出，详细定义请参考 “peripheral_ssp.h”

```
// 示例，每次传输大小是 8 个 word
#define DATA_LEN (SPI_FIFO_DEPTH)
static void setup_peripherals_spi_module(void)
{
    apSSP_sDeviceControlBlock pParam;
    pParam.eSclkDiv = SPI_INTERFACETIMINGSCLKDIV_DEFAULT_2M;// SPI 时钟设置
    pParam.eSCLKPolarity = SPI_CPOL_SCLK_LOW_IN_IDLE_STATES;// SPI 模式设置
    pParam.eSCLKPhase = SPI_CPHA_ODD_SCLK_EDGES;// SPI 模式设置
    pParam.eLsbMsbOrder = SPI_LSB_MOST_SIGNIFICANT_BIT_FIRST;
    pParam.eDataSize = SPI_DATALEN_32_BITS;// SPI 每个传输单位的大小
    pParam.eMasterSlaveMode = SPI_SLVMODE_MASTER_MODE;// SPI 主模式
    pParam.eReadWriteMode = SPI_TRANSMODE_WRITE_READ_SAME_TIME;// SPI 同时读写
    pParam.eQuadMode = SPI_DUALQUAD_REGULAR_MODE;
    pParam.eWriteTransCnt = DATA_LEN;// SPI 每次传输多少个单位（每个单位的大小是 pPar
    pParam.eReadTransCnt = DATA_LEN;// SPI 每次接收多少个单位（每个单位的大小是 pPara
    pParam.eAddrEn = SPI_ADDREN_DISABLE;
    pParam.eCmdEn = SPI_CMDEN_DISABLE;
    pParam.RxThres = DATA_LEN/2;
    pParam.TxThres = DATA_LEN/2;
    pParam.SlaveDataOnly = SPI_SLVDATAONLY_ENABLE;
    pParam.eAddrLen = SPI_ADDRLEN_1_BYTE;
```

```
    pParam.eInterruptMask = (1 << bsSPI_INTREN_ENDINTEN);// 打开 SPI 中断（传输结束后触发

    apSSP_DeviceParametersSet(APB_SSP1, &pParam);

}
```

16.5.1.3 SPI DMA 初始化

// 初始化 DMA 模块

```
static void setup_peripherals_dma_module(void)
{
    SYSCTRL_ClearClkGateMulti(1 << SYSCTRL_ClkGate_APB_DMA);
    DMA_Reset(1);
    DMA_Reset(0);
}
```

16.5.1.4 SPI DMA 设置

// 此处是以 SPI1 为例

```
// 分别设置 RX FIFO 和 TX FIFO 的 DMA
void peripherals_spi_dma_to_txfifo(int channel_id, void *src, int size)
{
    DMA_Descriptor descriptor __attribute__((aligned (8)));

    descriptor.Next = (DMA_Descriptor *)0;
    DMA_PrepareMem2Peripheral(&descriptor,SYSCTRL_DMA_SPI1_TX,src,size,DMA_ADDRESS_INC,0)

    DMA_EnableChannel(channel_id, &descriptor);
}
```

```
void peripherals_spi_rxfifo_to_dma(int channel_id, void *dst, int size)
{
    DMA_Descriptor descriptor __attribute__((aligned (8)));

    descriptor.Next = (DMA_Descriptor *)0;
    DMA_PreparePeripheral2Mem(&descriptor,dst,SYSCTRL_DMA_SPI1_RX,size,DMA_ADDRESS_IN

    DMA_EnableChannel(channel_id, &descriptor);
}
```

16.5.1.5 SPI 中断

```
// SPI ENDINT 中断触发标志传输结束，清除中断状态
static uint32_t peripherals_spi_isr(void *user_data)
{
  uint32_t stat = apSSP_GetIntRawStatus(APB_SSP1);

  if(stat & (1 << bsSPI_INTREN_ENDINTEN))
  {
    apSSP_ClearIntStatus(APB_SSP1, 1 << bsSPI_INTREN_ENDINTEN);
  }
}
```

16.5.1.6 SPI 接收数据

```
uint32_t read_data[DATA_LEN] = {0,};//数据大小等于 pParam.eDataSize，数组大小等于 pPa

void peripherals_spi_read_data(void)
{
  // 首先需要打开 SPI 模块中的 DMA 功能
```

```
  apSSP_SetRxDmaEn(APB_SSP1,1);
  apSSP_SetTransferControlRdTranCnt(APB_SSP1,DATA_LEN);
  // 配置 DMA，指向存储接收数据的地址
  peripherals_spi_rxfifo_to_dma(SPI_DMA_RX_CHANNEL, read_data, sizeof(read_data));
}
```

16.5.1.7 SPI 发送数据

```
uint32_t write_data[DATA_LEN];//数据大小等于 pParam.eDataSize，数组大小等于 pParam.eWrite

void peripherals_spi_push_data(void)
{
  // 首先需要打开 SPI 模块中的 DMA 功能
  apSSP_SetTxDmaEn(APB_SSP1,1);
  apSSP_SetTransferControlWrTranCnt(APB_SSP1,DATA_LEN);
  // 配置 DMA，指向需要发送的数据
  peripherals_spi_dma_to_txfifo(SPI_DMA_TX_CHANNEL, write_data, sizeof(write_data));

}

void peripherals_spi_send_data(void)
{
  // 分别设置接收和发射的 DMA
  peripherals_spi_read_data();
  peripherals_spi_push_data();

  // 写入命令，触发 SPI 传输
  apSSP_WriteCmd(APB_SSP1, 0x00, 0x00);//trigger transfer

  // 等待发送结束
  while(apSSP_GetSPIActiveStatus(APB_SSP1));
```

```
  // 关闭 SPI 模块中的 DMA 功能
  apSSP_SetTxDmaEn(APB_SSP1,0);
}
```

16.5.1.8 使用流程

- 设置 GPIO，setup_peripherals_spi_pin()
- 初始化 SPI，setup_peripherals_spi_module()
- 初始化 DMA，setup_peripherals_dma_module()
- 在需要时候发送 SPI 数据，peripherals_spi_send_data()
- 检查中断状态

16.5.2 SPI 从配置

16.5.2.1 接口配置

```
static void setup_peripherals_spi_pin(void)
{
    // 打开 SPI 模块时钟
    // 此处是以 SPI1 为例，如果是 SPI0 则需要更改
    // 根据使用的 GPIO，选择打开 GPIO0 或者 GPIO1 的时钟
    SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_SPI1)
                                | (1 << SYSCTRL_ITEM_APB_SysCtrl)
                                | (1 << SYSCTRL_ITEM_APB_PinCtrl)
                                | (1 << SYSCTRL_ITEM_APB_GPIO1)
                                | (1 << SYSCTRL_ITEM_APB_GPIO0));

    // 设置 IO MUX，将 GPIO 映射成 SPI 功能，不需要的 pin 可以使用 IO_NOT_A_PIN 替代
    PINCTRL_Pull(IO_SOURCE_SPI1_CLK_IN,PINCTRL_PULL_DOWN);
    PINCTRL_Pull(IO_SOURCE_SPI1_CSN_IN,PINCTRL_PULL_UP);// CS 需要默认上拉
    PINCTRL_SelSpiIn(SPI_PORT_1, SPI_MIC_CLK, SPI_MIC_CS, SPI_MIC_HOLD, SPI_MIC_WP, S
    PINCTRL_SetPadMux(SPI_MIC_CLK, IO_SOURCE_SPI1_CLK_OUT);
```

```
    PINCTRL_SetPadMux(SPI_MIC_MISO, IO_SOURCE_SPI1_MISO_OUT);

    // 设置 SPI 的中断
    platform_set_irq_callback(PLATFORM_CB_IRQ_APBSPI, peripherals_spi_isr, NULL);
```

16.5.2.2 SPI 模块初始化

常用设置项用注释标出，详细定义请参考 “peripheral_ssp.h”

```
// 示例，每次传输大小是 8 个 word
#define DATA_LEN (SPI_FIFO_DEPTH)
static void setup_peripherals_spi_module(void)
{
    apSSP_sDeviceControlBlock pParam;
    pParam.eSclkDiv = SPI_INTERFACETIMINGSCLKDIV_DEFAULT_2M;// SPI 时钟设置
    pParam.eSCLKPolarity = SPI_CPOL_SCLK_LOW_IN_IDLE_STATES;// SPI 模式设置
    pParam.eSCLKPhase = SPI_CPHA_ODD_SCLK_EDGES;// SPI 模式设置
    pParam.eLsbMsbOrder = SPI_LSB_MOST_SIGNIFICANT_BIT_FIRST;
    pParam.eDataSize = SPI_DATALEN_32_BITS;// SPI 每个传输单位的大小
    pParam.eMasterSlaveMode = SPI_SLVMODE_SLAVE_MODE;// SPI 从模式
    pParam.eReadWriteMode = SPI_TRANSMODE_WRITE_READ_SAME_TIME;// SPI 同时读写
    pParam.eQuadMode = SPI_DUALQUAD_REGULAR_MODE;
    pParam.eWriteTransCnt = DATA_LEN;// SPI 每次传输多少个单位（每个单位的大小是 pParam.e
    pParam.eReadTransCnt = DATA_LEN;// SPI 每次接收多少个单位（每个单位的大小是 pParam.eD
    pParam.eAddrEn = SPI_ADDREN_DISABLE;
    pParam.eCmdEn = SPI_CMDEN_DISABLE;
    pParam.RxThres = 0;
    pParam.TxThres = 0;
    pParam.SlaveDataOnly = SPI_SLVDATAONLY_ENABLE;
    pParam.eAddrLen = SPI_ADDRLEN_1_BYTE;
    pParam.eInterruptMask = (1 << bsSPI_INTREN_ENDINTEN);// 打开 SPI 中断（传输结束后触发

    apSSP_DeviceParametersSet(APB_SSP1, &pParam);
```

```
}
```

16.5.2.3 SPI DMA 初始化

// 初始化 DMA 模块

```
static void setup_peripherals_dma_module(void)
{
    SYSCTRL_ClearClkGateMulti(1 << SYSCTRL_ClkGate_APB_DMA);
    DMA_Reset(1);
    DMA_Reset(0);
}
```

16.5.2.4 SPI DMA 设置

// 此处是以 SPI1 为例

```
// 分别设置 RX FIFO 和 TX FIFO 的 DMA
void peripherals_spi_dma_to_txfifo(int channel_id, void *src, int size)
{
    DMA_Descriptor descriptor __attribute__((aligned (8)));

    descriptor.Next = (DMA_Descriptor *)0;
    DMA_PrepareMem2Peripheral(&descriptor,SYSCTRL_DMA_SPI1_TX,src,size,DMA_ADDRESS_IN

    DMA_EnableChannel(channel_id, &descriptor);
}

void peripherals_spi_rxfifo_to_dma(int channel_id, void *dst, int size)
{
    DMA_Descriptor descriptor __attribute__((aligned (8)));
```

```
    descriptor.Next = (DMA_Descriptor *)0;
    DMA_PreparePeripheral2Mem(&descriptor,dst,SYSCTRL_DMA_SPI1_RX,size,DMA_ADDRESS_INC,0)

    DMA_EnableChannel(channel_id, &descriptor);
}
```

16.5.2.5 SPI 接收数据

```
uint32_t read_data[DATA_LEN] = {0,};//数据大小等于 pParam.eDataSize，数组大小等于 pParam.

void peripherals_spi_read_data(void)
{
  // 首先需要打开 SPI 模块中的 DMA 功能
  apSSP_SetRxDmaEn(APB_SSP1,1);
  apSSP_SetTransferControlRdTranCnt(APB_SSP1,DATA_LEN);
  // 配置 DMA，指向存储接收数据的地址
  peripherals_spi_rxfifo_to_dma(SPI_DMA_RX_CHANNEL, read_data, sizeof(read_data));
}
```

16.5.2.6 SPI 发送数据

```
uint32_t write_data[DATA_LEN];//数据大小等于 pParam.eDataSize，数组大小等于 pParam.eWrite

void peripherals_spi_push_data(void)
{
  // 首先需要打开 SPI 模块中的 DMA 功能
  apSSP_SetTxDmaEn(APB_SSP1,1);
  apSSP_SetTransferControlWrTranCnt(APB_SSP1,DATA_LEN);
  // 配置 DMA，指向需要发送的数据
  peripherals_spi_dma_to_txfifo(SPI_DMA_TX_CHANNEL, write_data, sizeof(write_data));
```

```
}
```

16.5.2.7 SPI 中断

```
// SPI ENDINT 中断触发标志传输结束，清除中断状态
static uint32_t peripherals_spi_isr(void *user_data)
{
  uint32_t stat = apSSP_GetIntRawStatus(APB_SSP1);

  if(stat & (1 << bsSPI_INTREN_ENDINTEN))
  {
    // 根据情况决定是否需要准备下一次的接收和发射
    peripherals_spi_read_data();
    peripherals_spi_push_data();
    apSSP_ClearIntStatus(APB_SSP1, 1 << bsSPI_INTREN_ENDINTEN);
  }
}
```

16.5.2.8 使用流程

- 设置 GPIO，setup_peripherals_spi_pin()
- 初始化 SPI，setup_peripherals_spi_module()
- 初始化 DMA，setup_peripherals_dma_module()
- 设置接收 DMA，peripherals_spi_read_data();
- 设置发射 DMA，peripherals_spi_push_data();
- 观察 SPI 中断，中断触发代表当前接收结束

16.6 SPI 使用其他配置

16.6.1 SPI clock 配置

16.6.1.1 默认配置

对于默认配置，spi 时钟的配置通过 “pParam.eSclkDiv” 来实现，计算公式为：“spi interface clock / (2 * (eSclkDiv + 1))”，其中默认配置下，spi interface clock 为 24M，因此可以得到不同时钟下的 eSclkDiv：

```
#define SPI_INTERFACETIMINGSCLKDIV_DEFAULT_6M    (1)
#define SPI_INTERFACETIMINGSCLKDIV_DEFAULT_4M    (2)
#define SPI_INTERFACETIMINGSCLKDIV_DEFAULT_3M    (3)
#define SPI_INTERFACETIMINGSCLKDIV_DEFAULT_2M4   (4)
#define SPI_INTERFACETIMINGSCLKDIV_DEFAULT_2M    (5)
```

详细请参考 “peripheral_ssp.h”

16.6.1.2 HCLK 配置

对于更高的时钟，则需要在配置 “pParam.eSclkDiv” 之前打开 HCLK 配置，打开方式如下：

```
    SYSCTRL_SelectHClk(SYSCTRL_CLK_PLL_DIV_1+3);
    SYSCTRL_SelectSpiClk(SPI_PORT_1,SYSCTRL_CLK_HCLK);
```

其中 PLL 为 384M，“SYSCTRL_CLK_PLL_DIV_1+3” 代表 4 分频（推荐配置），可以将 spi interface clock 提高到 384M/4 = 96M。在此基础上通过计算公式 “spi interface clock / (2 * (eSclkDiv + 1))” 得到不同时钟下的 eSclkDiv：

```
#define SPI_INTERFACETIMINGSCLKDIV_HCLK_24M    (1)
#define SPI_INTERFACETIMINGSCLKDIV_HCLK_12M    (3)
```

详细请参考 “peripheral_ssp.h” 其中 spi interface clock 的大小可以在配置后，通过 “SYSCTRL_GetClk()” 来获取

ATTENTION : 此处的 API 使用均以 SPI1（SPI_PORT_1）为例，具体使用请根据需要调整

16.6.2 QSPI 使用

QSPI 的使用需要在以上的配置的基础上，做一些额外修改 #### pin 配置 QSPI 用到了 CLK,CS,MOSI,MISO,HOLD,WP, 主从都需要配置为输入输出

```
    PINCTRL_SelSpiIn(SPI_PORT_1, SPI_MIC_CLK, SPI_MIC_CS, SPI_MIC_HOLD, SPI_MIC_WP, S
    PINCTRL_SetPadMux(SPI_MIC_CLK, IO_SOURCE_SPI1_CLK_OUT);
    PINCTRL_SetPadMux(SPI_MIC_CS, IO_SOURCE_SPI1_CSN_OUT);
    PINCTRL_SetPadMux(SPI_MIC_MOSI, IO_SOURCE_SPI1_MOSI_OUT);
    PINCTRL_SetPadMux(SPI_MIC_MISO, IO_SOURCE_SPI1_MISO_OUT);
    PINCTRL_SetPadMux(SPI_MIC_WP, IO_SOURCE_SPI1_WP_OUT);
    PINCTRL_SetPadMux(SPI_MIC_HOLD, IO_SOURCE_SPI1_HOLD_OUT);
```

16.6.2.1 pParam 配置

QSPI 的部分 pParam 参数需要修改为如下，同时 Addr 和 Cmd 需要打开

```
    pParam.eQuadMode = SPI_DUALQUAD_QUAD_IO_MODE;
    pParam.SlaveDataOnly = SPI_SLVDATAONLY_DISABLE;
    pParam.eAddrEn = SPI_ADDREN_ENABLE;
    pParam.eCmdEn = SPI_CMDEN_ENABLE;
```

此外，主还需要切换读写 Mode（读和写顺序执行，比如首先利用四线完成写操作，然后再读）

```
    pParam.eReadWriteMode = SPI_TRANSMODE_WRITE_READ;
```

或者（在 Write 和 Read 之间添加 dummy（默认为 8 个 clk cycle））

```
    pParam.eReadWriteMode = SPI_TRANSMODE_WRITE_DUMMY_READ;
```

从则添加

```
    pParam.eReadWriteMode = SPI_TRANSMODE_READ_WRITE;
```

或者（在 Read 和 Write 之间添加 dummy（默认为 8 个 clk cycle））

```
    pParam.eReadWriteMode = SPI_TRANSMODE_READ_DUMMY_WRITE;
```

第十七章　系统控制（SYSCTRL）

17.1　功能概述

SYSCTRL 负责管理、控制各种片上外设，主要功能有：

- 外设的复位
- 外设的时钟管理，包括时钟源、频率设置、门控等
- DMA 规划
- 其它功能

17.1.1　外设标识

SYSCTRL 为外设定义了几种不同的标识。最常见的一种标识为：

```
typedef enum
{
    SYSCTRL_ITEM_APB_GPIO0     ,
    SYSCTRL_ITEM_APB_GPIO1     ,
    // ...
    SYSCTRL_ITEM_NUMBER,
} SYSCTRL_Item;
```

这种标识用于外设的复位、时钟门控等。`SYSCTRL_ResetItem` 和 `SYSCTRL_ClkGateItem` 是 `SYSCTRL_Item` 的两个别名。

下面这种标识用于 DMA 规划：

```
typedef enum
{
    SYSCTRL_DMA_UART0_RX = 0,
    SYSCTRL_DMA_UART1_RX = 1,
    //...
} SYSCTRL_DMA;
```

17.1.2 时钟树

从源头看，共有 4 个时钟源：

1. 内部 32KiHz RC 时钟；
2. 外部 32768Hz 晶体；
3. 内部高速 RC 时钟（8M/16M/24M/32M/48M/64MHz 可调）；
4. 外部 24MHz 晶体。

从 4 个时钟源出发，得到两组时钟：

1. 32KiHz 时钟（*clk_32k*）

   32k 时钟有两个来源：内部 32KiHz RC 电路，外部 32768Hz 晶体。

2. 慢时钟

   慢时钟有两个来源：内部高速 RC 时钟，外部 24MHz 晶体。

   BLE 子系统中射频相关的部分固定使用外部 24MHz 晶体提供的时钟。

之后，

1. PLL 输出（*clk_pll*）

   clk_pll 的频率 f_{pll} 可配置，受 $loop$、div_{pre} 和 div_{output} 等 3 个参数控制：

$$f_{vco} = \frac{f_{in} \times loop}{div_{pre}}$$

$$f_{pll} = \frac{f_{vco}}{div_{output}}$$

   这里，f_{in} 即慢时钟。要求 $f_{vco} \in [60, 600]MHz$，$f_{in}/div_{pre} \in [2, 24]MHz$。

2. *sclk_fast* 与 *sclk_slow*

   clk_pll 经过门控后的时钟称为 *sclk_fast*，慢时钟经过门控后称为 *sclk_slow*。

3. *hclk*

   sclk_fast 经过分频后得到 *hclk*。下列外设（包括 CPU）固定使用这个时钟[1]：

   - DMA
   - 片内 Flash
   - QSPI
   - USB[2]
   - 其它内部模块如 AES、Cache 等

   hclk 经过分频后得到 *pclk*。*pclk* 主要用于硬件内部接口。

4. *sclk_slow* 的进一步分频

   sclk_slow 经过若干独立的分频器得到以下多种时钟：

   - *sclk_slow_pwm_div*：专供 PWM 选择使用
   - *sclk_slow_timer_div*：供 TIMER0、TIMER1、TIMER2 选择使用
   - *sclk_slow_ks_div*：专供 KeyScan 选择使用
   - *sclk_slow_adc_div*：供 EFUSE、ADC、IR 选择使用
   - *sclk_slow_pdm_div*：专供 PDM 选择使用

5. *sclk_fast* 的进一步分频：

   sclk_fast 经过若干独立的分频器得到以下多种时钟：

   - *sclk_fast_i2s_div*：专供 I2S 选择使用
   - *sclk_fast_qspi_div*：专供 SPI0 选择使用
   - *sclk_fast_flash_div*：专供片内 Flash 选择使用
   - *sclk_fast_usb_div*：专供 USB 使用

各硬件外设可配置的时钟源汇总如表 17.1。

[1] 每个外设可单独对 *hclk* 门控。

[2] 仅高速时钟。

表 17.1: 各硬件外设的时钟源

外设	时钟源
GPIO0、GPIO1	选择 *sclk_slow* 或者 *clk_32k*
TMR0、TMR1、TMR2	独立配置 *sclk_slow_timer_div* 或者 *clk_32k*
WDT	*clk_32k*
PWM	*sclk_slow_pwm_div* 或者 *clk_32k*
PDM	*sclk_slow_pdm_div*
QDEC	对 *hclk* 或者 *sclk_slow*
KeyScan	*sclk_slow_ks_div* 或者 *clk_32k*
IR、ADC、EFUSE	独立配置 *sclk_slow_adc_div* 或者 *sclk_slow*
DMA	*hclk*
SPI0	*sclk_fast_qspi_div* 或者 *sclk_slow*
I2S	*sclk_fast_i2s_div* 或者 *sclk_slow*
UART0、UART1、SPI1	独立配置 *hclk* 或者 *sclk_slow*
I2C0、I2C1	*pclk*

17.1.3 DMA 规划

由于DMA 支持的硬件握手信号只有 16 种，无法同时支持所有外设。因此需要事先确定将要的外设握手信号，并通过 `SYSCTRL_SelectUsedDmaItems` 接口声明。

一个外设可能具备一个以上的握手信号，需要注意区分。比如 UART0 有两个握手信号 UART0_RX 和 UART0_TX，分别用于触发 DMA 发送请求（通过 DMA 传输接收到的数据）和读取请求（向 DMA 请求新的待发送数据）。外设握手信号定义在 `SYSCTRL_DMA` 内：

```
typedef enum
{
    SYSCTRL_DMA_UART0_RX = 0,
    SYSCTRL_DMA_UART1_RX = 1,
    // ...
} SYSCTRL_DMA;
```

17.2 使用说明

17.2.1 外设复位

通过 SYSCTRL_ResetBlock 复位外设，通过 SYSCTRL_ReleaseBlock 释放复位。

```
void SYSCTRL_ResetBlock(SYSCTRL_ResetItem item);
void SYSCTRL_ReleaseBlock(SYSCTRL_ResetItem item);
```

17.2.2 时钟门控

通过 SYSCTRL_SetClkGate 设置门控（即关闭时钟），通过 SYSCTRL_ClearClkGate 消除门控（即恢复时钟）。

```
void SYSCTRL_SetClkGate(SYSCTRL_ClkGateItem item);
void SYSCTRL_ClearClkGate(SYSCTRL_ClkGateItem item);
```

SYSCTRL_SetClkGateMulti 和 SYSCTRL_ClearClkGateMulti 可以同时控制多个外设的门控。items 参数里的各个比特与 SYSCTRL_ClkGateItem 里的各个外设一一对应。

```
void SYSCTRL_SetClkGateMulti(uint32_t items);
void SYSCTRL_ClearClkGateMulti(uint32_t items);
```

17.2.3 时钟配置

举例如下。

1. *clk_pll* 与 *hclk*

   使用 SYSCTRL_ConfigPLLClk 配置 *clk_pll*：

```
int SYSCTRL_ConfigPLLClk(
uint32_t div_pre,
uint32_t loop,
uint32_t div_output);
```

例如，假设慢时钟配置为 24MHz，下面的代码将 *hclk* 配置为 220MHz 并读取到变量：

```
SYSCTRL_ConfigPLLClk(6, 110, 1);
SYSCTRL_SelectHClk(SYSCTRL_CLK_PLL_DIV_1 + 1);
uint32_t SystemCoreClock = SYSCTRL_GetHClk();
```

2. 为硬件 I2S 配置时钟

使用 `SYSCTRL_SelectI2sClk` 为 I2S 配置时钟：

```
void SYSCTRL_SelectI2sClk(SYSCTRL_ClkMode mode);
```

`SYSCTRL_ClkMode` 的定为为：

```
typedef enum
{
    SYSCTRL_CLK_SLOW,              // 使用 sclk_slow
    SYSCTRL_CLK_32k = ...,         // 使用 32KiHz 时钟
    SYSCTRL_CLK_HCLK,              // 使用 hclk
    SYSCTRL_CLK_ADC_DIV = ...,     // 使用 sclk_slow_adc_div
    SYSCTRL_CLK_PLL_DIV_1 = ...,   // 对 sclk_fast 分频
    SYSCTRL_CLK_SLOW_DIV_1 = ..., // 对 sclk_slow 分配
} SYSCTRL_ClkMode;
```

根据表 17.1 可知，I2S 可使用 _slk_slow：

```
SYSCTRL_SelectI2sClk(SYSCTRL_CLK_SLOW);
```

或者独占一个分频器，对 *sclk_fast* 分频得到 *sclk_fast_i2s_div*，比如使用 *sclk_fast* 的 5[3] 分频：

```
SYSCTRL_SelectI2sClk(SYSCTRL_CLK_PLL_DIV_1 + 4);
```

3. 读取时钟频率

   使用 SYSCTRL_GetClk 可获得指定外设的时钟频率：

```
uint32_t SYSCTRL_GetClk(SYSCTRL_Item item);
```

   比如，

```
// I2S 使用 PLL 的 5 分频
SYSCTRL_SelectI2sClk(SYSCTRL_CLK_PLL_DIV_1 + 4);
// freq = sclk_fast 的频率 / 5
uint32_t freq = SYSCTRL_GetClk(SYSCTRL_ITEM_APB_I2S);
```

4. 降低频率以节省功耗

   降低系统各时钟的频率可以显著降低动态功耗。相关函数有：

   - SYSCTRL_SelectHClk：选择 *hclk*
   - SYSCTRL_SelectFlashClk：选择内部 Flash 时钟
   - SYSCTRL_SelectSlowClk：选择慢时钟
   - SYSCTRL_EnablePLL：开关 PLL

5. 配置用于慢时钟的高速 RC 时钟

   通过 SYSCTRL_EnableSlowRC 可以使能并配置内部高速 RC 时钟的频率模式：

[3] $5 = 1 + 4$

```
void SYSCTRL_EnableSlowRC(
    uint8_t enable,              // 使能或禁用
    SYSCTRL_SlowRCClkMode mode   // 频率模式
);
```

频率模式为：

```
typedef enum
{
    SYSCTRL_SLOW_RC_8M = ...,
    SYSCTRL_SLOW_RC_16M = ...,
    SYSCTRL_SLOW_RC_24M = ...,
    SYSCTRL_SLOW_RC_32M = ...,
    SYSCTRL_SLOW_RC_48M = ...,
    SYSCTRL_SLOW_RC_64M = ...,
} SYSCTRL_SlowRCClkMode;
```

由于内部（芯片之间）、外部环境（温度）存在微小差异或变化，所以这个 RC 时钟的频率或存在一定误差，需要通过进行调谐以尽量接近标称值。通过 `SYSCTRL_AutoTuneSlowRC` 可进行自动调谐[4]：

```
uint32_t SYSCTRL_AutoTuneSlowRC(void);
```

这个函数返回的数据为调谐参数。如果认为有必要[5]，可以把此参数储存起来，后续系统重启后，通过 `SYSCTRL_TuneSlowRC` 直接写入参数调谐频率。

温馨提示：

- 关闭 PLL 时，务必先将 CPU、Flash 时钟切换至慢时钟；
- 修改慢时钟配置时，需要保证 PLL 的输出、CPU 时钟等在支持的频率内；
- 推荐使用 SDK 提供的工具生成时钟配置代码，规避错误配置。

[4] 以 24MHz 晶体为参考。

[5] 比如认为 `SYSCTRL_AutoTuneSlowRC` 耗时过长。

17.2.4 DMA 规划

使用 SYSCTRL_SelectUsedDmaItems 配置要使用的 DMA 握手信号：

```
int SYSCTRL_SelectUsedDmaItems(
  uint32_t items // 各比特与 SYSCTRL_DMA 一一对应
  );
```

使用 SYSCTRL_GetDmaId 可获取为某外设握手信号的 DMA 信号 ID，如果返回 -1，说明没有规划该外设握手信号[6]：

```
int SYSCTRL_GetDmaId(SYSCTRL_DMA item);
```

[6] SYSCTRL_SelectUsedDmaItems 的 items 参数里对应的比特为 0

第十八章　定时器（TIMER）

18.1　功能概述

定时器（TIMER）是由两个寄存器组成的，其中一个寄存器用来确定计数器的工作形式和功能，另外一个计时器是用来控制单片机的启动和停止的，同时也设置了一个

18.2　使用说明

18.2.1　设置工作模式

在使用 TIMER 之前，需要使用 `TMR_SetOpMode` 按需设置 TIMER 的工作模式。

```
void TMR_SetOpMode(
  TMR_TypeDef *pTMR,
  uint8_t ch_id,
  uint8_t op_mode,
  uint8_t clk_mode,
  uint8_t pwm_park_value
  );
```

TIMER 的工作模式有以下 6 种，分别三类：定时器、PWM、复用。* 定时器功能包括：32bit 单定时器、16bit 双定时器、8bit 四定时器 “‘c #define TMR_CTL_OP_MODE_32BIT_TIMER_x1 1 // one 32bit timer #define TMR_CTL_OP_MODE_16BIT_TIMER_x2 2 // dual 16bit timers #define TMR_CTL_OP_MODE_8BIT_TIMER_x4 3 // four 8bit timers

* PWM功能：16bit双计数器

```c
 #define TMR_CTL_OP_MODE_16BIT_PWM            4        // PWM with two 16bit counters
```

- 定时器 +PWM 复用功能：8bit 双计数器 +16bit 单定时器、8bit 双计数器 +8bit 双定时器

```
   #define TMR_CTL_OP_MODE_8BIT_PWM_16BIT_TIMER_x1    6           // MIXED: PWM with tw
   #define TMR_CTL_OP_MODE_8BIT_PWM_8BIT_TIMER_x2     7           // MIXED: PWM with tw
```

- 例如将 TIMER1 的通道 0 设置为 `TMR_CTL_OP_MODE_32BIT_TIMER_x1` 模式：

```
   TMR_SetOpMode(APB_TMR1, 0, TMR_CTL_OP_MODE_32BIT_TIMER_x1, TMR_CLK_MODE_APB, 0);
   TMR_SetReload(APB_TMR1, 0, TMR_GetClk(APB_TMR1, 0) / 10);
   TMR_Enable(APB_TMR1, 0, 0xf);
   TMR_IntEnable(APB_TMR1, 0, 0xf);
```

18.2.2 时钟频率

使用 `TMR_GetClk` 获取 TIMER 某个通道的时钟频率。

```
uint32_t TMR_GetClk(
  TMR_TypeDef *pTMR,
  uint8_t ch_id
  );
```

18.2.3 重载值

使用 `TMR_SetReload` 设置 TIMER 某个通道的重载值。

```
void TMR_SetReload(
  TMR_TypeDef *pTMR,
  uint8_t ch_id,
  uint32_t value
  );
```

在不同的 TIMER 模式中，value 的值分配如下表。Table:

TIMER 模式	bits[0:7]	bits[8:15]	bits[1
TMR_CTL_OP_MODE_32BIT_TIMER_x1	Timer 0	Timer 0	Tim
TMR_CTL_OP_MODE_16BIT_TIMER_x2	Timer 0	Timer 0	Tim
TMR_CTL_OP_MODE_8BIT_TIMER_x4	Timer 0	Timer 1	Tim
TMR_CTL_OP_MODE_16BIT_PWM	PWM low period	PWM low period	PWM hig
TMR_CTL_OP_MODE_8BIT_PWM_16BIT_TIMER_x1	Timer 0	Timer 0	PWM lo
TMR_CTL_OP_MODE_8BIT_PWM_8BIT_TIMER_x2	Timer 0	Timer 1	PWM lo

18.2.4 使能

使用 TMR_Enable TIMER 的某个通道。

```
void TMR_Enable(
  TMR_TypeDef *pTMR,
  uint8_t ch_id,
  uint8_t mask
  );
```

18.2.5 比较

使用 TMR_GetCMP 。

```
uint32_t TMR_GetCMP(
  TMR_TypeDef *pTMR,
  uint8_t ch_id
  );
```

18.2.6 配置中断请求

使用 TMR_IntEnable 配置并使能 TIMER 中断。

```
void TMR_IntEnable(
  TMR_TypeDef *pTMR,
  uint8_t ch_id,
  uint8_t mask
  );
```

18.2.7 中断清除

使用 TMR_IntClr 清除某个 TIMER 的中断状态。

```
void TMR_IntClr(
  TMR_TypeDef *pTMR,
  uint8_t ch_id,
  uint8_t mask
  );
```

18.2.8 获取中断

使用 TMR_IntHappened 获取某个 TIMER 的中断状态。

```
uint8_t TMR_IntHappened(
  TMR_TypeDef *pTMR,
  uint8_t ch_id
  );
```

第十九章　通用异步收发传输器（UART）

19.1　功能概述

UART 负责处理数据总线和串行口之间的串/并、并/串转换，并规定了相应的帧格式，通信双方只要采用相同的帧格式和波特率，就能在未共享时钟信号的情况下，仅用两根信号线（RX 和 TX）完成通信过程。

特性：

- 异步串行通信，可为全双工、半双工、单发送（TX）或单接收（RX）模式；
- 支持 5~8 位数据位的配置，波特率几百 bps 至几百 Kbbps；
- 可配置奇校验、偶校验或无校验位；可配置 1、1.5 或 2 位停止位；
- 将并行数据写入内存缓冲区，再通过 FIFO 逐位发送，接收时同理；
- 输出传输时，从低位到高位传输。

19.2　使用说明

19.2.1　设置波特率

使用 `apUART_BaudRateSet` 设置对应 UART 设备的波特率。

```
void apUART_BaudRateSet(
  UART_TypeDef* pBase,
  uint32_t ClockFrequency,
  uint32_t BaudRate
  );
```

19.2.2 获取波特率

使用 apUART_BaudRateGet 获取对应 UART 设备的波特率。

```
uint32_t apUART_BaudRateGet (
  UART_TypeDef* pBase,
  uint32_t ClockFrequency
  );
```

19.2.3 接收错误查询

使用 apUART_Check_Rece_ERROR 查询接收产生的错误。

```
uint8_t apUART_Check_Rece_ERROR(
  UART_TypeDef* pBase
  );
```

19.2.4 FIFO 轮询模式

在轮询模式下，CPU 通过检查线路状态寄存器中的位来检测事件：

- 使用 apUART_Check_Rece_ERROR 查询接收产生的错误字。

```
uint8_t apUART_Check_Rece_ERROR(
  UART_TypeDef* pBase
  );
```

- 用 apUART_Check_RXFIFO_EMPTY 查询 RXFIFO 是否为空。

```
uint8_t apUART_Check_RXFIFO_EMPTY(
  UART_TypeDef* pBase
  );
```

- 使用 apUART_Check_RXFIFO_FULL 查询 RXFIFO 是否已满。

```
uint8_t apUART_Check_RXFIFO_FULL(
  UART_TypeDef* pBase
  );
```

- 使用 apUART_Check_TXFIFO_EMPTY 查询 TXFIFO 是否为空。

```
uint8_t apUART_Check_TXFIFO_EMPTY(
  UART_TypeDef* pBase
  );
```

- 使用 apUART_Check_TXFIFO_FULL 查询 TXFIFO 是否已满。

```
uint8_t apUART_Check_TXFIFO_FULL(
  UART_TypeDef* pBase
  );
```

19.2.5 发送数据

使用 UART_SendData 发送 8bits 数据。

```
void UART_SendData(
  UART_TypeDef* pBase,
  uint8_t Data
  );
```

19.2.6 接收数据

使用 UART_ReceData 接收 8bits 数据。

```
uint8_t UART_ReceData(
  UART_TypeDef* pBase
  );
```

19.2.7 配置中断请求

使用 apUART_Enable_TRANSMIT_INT 使能发送中断状态。

```
void apUART_Enable_TRANSMIT_INT(
  UART_TypeDef* pBase
  );
```

使用 apUART_Disable_TRANSMIT_INT 禁用发送中断状态。

```
void apUART_Disable_TRANSMIT_INT(
  UART_TypeDef* pBase
  );
```

使用 apUART_Enable_RECEIVE_INT 使能接收中断状态。

```
void apUART_Enable_RECEIVE_INT(
  UART_TypeDef* pBase
  );
```

使用 apUART_Disable_RECEIVE_INT 禁用接收中断状态。

```
void apUART_Disable_RECEIVE_INT(
  UART_TypeDef* pBase
  );
```

19.2.8 处理中断状态

uint8_t apUART_Get_ITStatus(UART_TypeDef* pBase,uint8_t UART_IT); uint32_t apUART_Get_all_raw_i
pBase);

void apUART_Clr_RECEIVE_INT(UART_TypeDef* pBase); void apUART_Clr_TX_INT(UART_TypeDef*
pBase); void apUART_Clr_NonRx_INT(UART_TypeDef* pBase);

19.2.9 UART 初始化

两个设备使用 UART 通讯时，必须先约定好传输速率和一些数据位。

```
typedef struct UART_xStateStruct
{
    // Line Control Register, UARTLCR_H
  UART_eWLEN      word_length;   // WLEN
  UART_ePARITY    parity;        // PEN, EPS, SPS
  uint8_t         fifo_enable;   // FEN
  uint8_t         two_stop_bits; // STP2
  // Control Register, UARTCR
  uint8_t         receive_en;        // RXE
  uint8_t         transmit_en;       // TXE
  uint8_t         UART_en;           // UARTEN
  uint8_t         cts_en;            //CTSEN
  uint8_t         rts_en;            //RTSEN
  // Interrupt FIFO Level Select Register, UARTIFLS
  uint8_t         rxfifo_waterlevel; // RXIFLSEL
  uint8_t         txfifo_waterlevel; // TXIFLSEL
  //UART_eFIFO_WATERLEVEL    rxfifo_waterlevel; // RXIFLSEL
  //UART_eFIFO_WATERLEVEL    txfifo_watchlevel; // TXIFLSEL

  // UART Clock Frequency
  uint32_t        ClockFrequency;
  uint32_t        BaudRate;
```

```
} UART_sStateStruct;
```

定义函数 config_uart，

```
void config_uart(
  uint32_t freq,
  uint32_t baud
  );
```

在函数中，对 UART_sStateStruct 的各项参数初始化，并调用 apUART_Initialize 对 UART 进行初始化。

```
void config_uart(uint32_t freq, uint32_t baud)
{
    UART_sStateStruct config;

    config.word_length       = UART_WLEN_8_BITS;
    config.parity            = UART_PARITY_NOT_CHECK;
    config.fifo_enable       = 1;
    config.two_stop_bits     = 0;
    config.receive_en        = 1;
    config.transmit_en       = 1;
    config.UART_en           = 1;
    config.cts_en            = 0;
    config.rts_en            = 0;
    config.rxfifo_waterlevel = 1;
    config.txfifo_waterlevel = 1;
    config.ClockFrequency    = freq;
    config.BaudRate          = baud;

    apUART_Initialize(PRINT_PORT, &config, 0);
}
```

19.2.10 发送数据

使用 UART_SendData 发送数据。

```
void UART_SendData(
  UART_TypeDef* pBase,
  uint8_t Data
  );
```

19.2.11 接收数据

使用 UART_ReceData 接收数据。

```
uint8_t UART_ReceData(
  UART_TypeDef* pBase
  );
```

19.2.12 清空 FIFO

使用 uart_empty_fifo 清空 UART 的 FIFO。

```
static void uart_empty_fifo(
  UART_TypeDef* pBase
  );
```

19.2.13 使能 FIFO

使用 uart_enable_fifo 使能 UART 的 FIFO。

```
static void uart_enable_fifo(
  UART_TypeDef* pBase
  );
```

19.2.14 处理中断状态

用 apUART_Get_ITStatus 获取某个 UART 上的中断触发状态，返回非 0 值表示该 UART 上产生了中断请求；用 apUART_Get_all_raw_int_stat 一次性获取所有 UART 的中断触发状态，第 n 比特（第 0 比特为最低比特）对应 UART n 上的中断触发状态。

UART 产生中断后，需要消除中断状态方可再次触发。用 apUART_Clr_RECEIVE_INT 消除某个 UART 上接收中断的状态，用 apUART_Clr_TX_INT 消除某个 UART 上发送中断的状态。用 apUART_Clr_NonRx_INT 消除某个 UART 上除接收以外的中断状态。

19.2.15 DMA 传输模式使能

DMA 向 CPU 发出总线请求，CPU 将总线交给 DMA 之后，由 DMA 控制数据的收发工作。

使用 UART_DmaEnable 使能 DMA 工作模式。

```
void UART_DmaEnable(
  UART_TypeDef *pBase,
  uint8_t tx_enable,
  uint8_t rx_enable,
  uint8_t dma_on_err
  );
```

第二十章　Universal serial bus device（USB）

20.1　功能概述

- 支持 full-speed (12 Mbps) device 模式
- 集成 PHY Transceiver，内置上拉，软件可控
- Endpoints:
  - Endpoints 0: control endpoint
  - Endpoints 1-5: 可以配置为 in/out，以及 control/isochronous/bulk/interrupt
- 支持 USB suspend, resume, remote-wakeup
- 内置 DMA 方便数据传输

20.2　使用说明

20.2.1　USB 软件结构

- driver layer，USB 的底层处理，不建议用户修改。
  - 处理了大部分和应用场景无关的流程，提供了 USB_IrqHandler，调用 event handler。
  - 位置：`\ING\_SDK\sdk\src\FWlib\peripheral\_usb.c`
- bsp layer，处理场景相关的流程，需要用户提供 event handler，并实现 control 和 transfer 相关处理。
  - 位置：`\ING\_SDK\sdk\src\BSP\bsp\_usb_xxx.c`

20.2.2 USB Device 状态

- USB 的使用首先需要配置 USB CLK，USB IO 以及 PHY，并且初始化 USB 模块，此时 USB 为 “NONE” 状态，等待 USB 的 reset 中断（USB_IrqHandler）。
- reset 中断的触发代表 USB cable 已经连接，而且 host 已经检测到了 device，在 reset 中断中，USB 模块完成相关的 USB 初始化。并继续等待中断。
- enumeration 中断的触发代表 device 可以开始接收 SOF 以及 control 传输，device 需要配置并打开 endpoint 0，进入 “DEFAULT” 状态。
- out 中断的触发代表收到了 host 的 get descriptor，用户需要准备好相应的 descriptor，并配置相关的 in endpoint。
- out 中断中的 set address request 会将 device 的状态切换为 “ADDRESS”。
- out 中断中的 set configuration 会将 device 的状态切换为 “CONFIGURED”。此时 device 可以开始在配置的 endpoint 上传输数据。
- bus 上的 idle 会自动触发 suspend 中断（用户需要在初始化中使能 suspend 中断），此时切换为 “SUSPEND” 状态。
- idle 之后任何 bus 上的活动将会触发 resume 中断（用户需要在初始化中使能 resume 中断）, 用户也可以选择使用 remote wakeup 主动唤醒。
- 唤醒之后的 usb 将重新进入 “CONFIGURED” 状态，每 1m（full-speed）将会收到 1 个 SOF 中断（用户需要在初始化中使能 sof 中断）。

```
typedef enum
{
  USB_DEVICE_NONE,
  /* A USB device may be attached or detached from the USB */
  USB_DEVICE_ATTACHED,
  /*USB devices may obtain power from an external source  */
  USB_DEVICE_POWERED,
  /*  After the device has been powered, and reset is done */
  USB_DEVICE_DEFAULT,
  /*  All USB devices use the default address when initially powered or after the device
      USB device is assigned a unique address by the host after attachment or after reset
  USB_DEVICE_ADDRESS,
  /* Before a USB device function may be used, the device must be configured. */
  USB_DEVICE_CONFIGURED,
```

```
  /* In order to conserve power, USB devices automatically enter the Suspended state
     observed no bus traffic for a specified period */
  USB_DEVICE_SUSPENDED,
  USB_DEVICE_TEST_RESET_DONE
}USB_DEVICE_STATE_E;
```

20.2.3 设置 IO

对于 Ing91682,USB 的 DP/DM 固定在 GPIO16/17,IO 初始化细节请参考 `ING_SDK\sdk\src\BSP\bsp\_usb` 中的 `bsp_usb_init()`

```
// ATTENTION ! FIXED IO FOR USB on 916 series
#define USB_PIN_DP GIO_GPIO_16
#define USB_PIN_DM GIO_GPIO_17
```

20.2.4 设置 PHY

使用 `SYSCTRL_USBPhyConfig()` 初始化 PHY, 细节请参考 `ING_SDK\sdk\src\BSP\bsp\_usb.c` 中的 `bsp_usb_init()`

```
/**
 * @brief Config USB PHY functionality
 *
 * @param[in] enable            Enable(1)/Disable(0) usb phy module
 * @param[in] pull_sel          DP pull up(0x1)/DM pull up(0x2)/DP&DM pull down(0x3)
 */
void SYSCTRL_USBPhyConfig(uint8_t enable, uint8_t pull_sel);
```

20.2.5 USB 模块初始化

细节请参考 `ING_SDK\sdk\src\BSP\bsp\_usb.c` 中的 `bsp_usb_init()`。

- USB 模块首选需要打开 USB 中断并配置相应接口，其中 USB_IrqHandler 由 driver 提供不需要用户修改。

```
platform_set_irq_callback(PLATFORM_CB_IRQ_USB, USB_IrqHandler, NULL);
```

- 其次需要初始化 USB 模块以及相关状态信息，入参结构体中用户需要提供 event handler，其余为可选项

```
/**
 * @brief interface API. initilize usb module and related variables, must be called befor
 *
 * @param[in] device callback function with structure USB_INIT_CONFIG_T.
 *            When this function has been called your device is ready to be enumerated by
 * @param[out] null.
 */
extern USB_ERROR_TYPE_E USB_InitConfig(USB_INIT_CONFIG_T *config);
```

20.2.6 event handler

USB 的用户层调用通过 event handler 来实现，细节请参考 `ING_SDK\sdk\src\BSP\bsp\_usb.c` 中的 `bsp_usb_event_handler()`。

event handler 需要包含对以下 event 事件的处理：

- USB_EVENT_EP0_SETUP
  - 该 event 包含 EPO(control endpoint) 上的所有 request，包括读取/设置 descriptor，设置 address，set/clear feature 等 request，按照 USB 协议，device 需要支持所有协议中的标准 request。
  - descriptor 需要按照协议格式准备，并且放置在 4bytes 对齐的全局地址，并通过 USB_SendData() 发送给 host，在整个过程中，该全局地址和数据需要保持。（4bytes 对齐是内部 DMA 搬运的要求，否则可能出现错误）。
  - 对于没有 data stage 的 request，event handler 中不需要使用 USB_SendData()/USB_RecvData()’。

  - 对于不支持的 request，需要设置 status 为：

```
status = USB_ERROR_REQUEST_NOT_SUPPORT;
```

  - 根据返回的 status，driver 判断当前 request 是否支持，否则按照协议发送 stall 给 host。
  - 对于包含 data stage 的 out 传输，driver 将继续接收数据，数据将在 USB_EVENT_EP_DATA_TRA 的 ep0 通知用户。
  - setup/data/status stage 的切换将在 driver 内进行。

- USB_EVENT_EP_DATA_TRANSFER

  数据相关的处理，接收和发射数据。

  - 参数包含 ep number

```
uint8_t  ep;
```

  以及数据处理类型，分别代表发送和接收 transfer 的结束。

```
typedef enum
{
    /// Event send when a receive transfert is finish
    USB_CALLBACK_TYPE_RECEIVE_END,
    /// Event send when a transmit transfert is finish
    USB_CALLBACK_TYPE_TRANSMIT_END
} USB_CALLBACK_EP_TYPE_T;
```

- USB_EVENT_DEVICE_RESET

  USB reset 中断的 event，代表枚举的开始。

- USB_EVENT_DEVICE_SOF

  SOF 中断，每 1m（full-speed）将会收到 1 个 SOF 中断（用户需要在初始化中使能 sof 中断）。

- USB_EVENT_DEVICE_SUSPEND

  bus 进入 idle 状态后触发 suspend，此时总线上没有 USB 活动。driver 会关闭 phy clock。

- USB_EVENT_DEVICE_RESUME

  bus 上的任何 USB 活动将会触发 wakeup 中断。resume 后 driver 打开 phy clock，USB 恢复到正常状态。

20.2.6.1 USB_EVENT_EP0_SETUP 的实现

control 以及枚举相关的流程实现需要通过 USB_EVENT_EP0_SETUP event 来进行。

- 默认的 control endpoint 是 ep0, 所有 request 都会触发该 event
- 如果场景不支持某个 request，则需要设置 status == USB_ERROR_REQUEST_NOT_SUPPORT。driver 会据此发送 stall。
- 如果场景需要处理某个 request，则需要将 status 设置为非 USB_ERROR_REQUEST_NOT_SUPPORT 状态。
- 用户需要将所有 descriptor 保存在 4bytes 对齐的全局地址。以 device descriptor 为例

```
case USB_REQUEST_DEVICE_DESCRIPTOR_DEVICE:
{
  size = sizeof(USB_DEVICE_DESCRIPTOR_REAL_T);
  size = (setup->wLength < size) ? (setup->wLength) : size;

  status |= USB_SendData(0, (void*)&DeviceDescriptor, size, 0);
}
break;
```

  首先判断 size，确保数据没有超出 request 要求，然后使用 USB_SendData 发送 in transfer 数据。其中 DeviceDescriptor 为 device descriptor 地址。

- set address request 需要配置 device 地址，因此在 driver layer 实现。

20.2.6.2 SUSPEND 的处理

SUSPEND 状态下可以根据需求进行 power saving，默认配置只关闭了 phy clock，其余的 USB power/clock 处理需要根据场景在应用层中的 low power mode 中来实现。

20.2.6.3 remote wakeup

进入 suspend 的 device 可以选择主动唤醒，唤醒通过 bsp_usb_device_remote_wakeup() 连续发送 10ms 的 resume signal 来实现。

```
void bsp_usb_device_remote_wakeup(void)
{
  USB_DeviceSetRemoteWakeupBit(U_TRUE);
  platform_set_timer(internal_bsp_usb_device_remote_wakeup_stop,16);// setup timer fo
}
```

20.2.7 常用 driver API

20.2.7.1 send usb data

使用该 API 发送 USB 数据（包括 setup data 和应用数据），但需要在 set config（打开 endpoint）之后使用。

```
/**
 * @brief interface API. send usb device data packet.
 *
 * @param[in] ep, endpoint number, the highest bit represent direction, use USB_EP_DI
 * @param[in] buffer, global buffer to hold data of the packet, it must be a DWORD-al
 * @param[in] size. it should be a value smaller than 512*mps(eg, for EP0, mps is 64b
 * @param[in] flag. null
 * @param[out] return U_TRUE if successful, otherwise U_FALSE.
 */
extern USB_ERROR_TYPE_E USB_SendData(uint8_t ep, void* buffer, uint16_t size, uint32_
```

20.2.7.2 receive usb data

使用该 API 接收 USB 数据（包括 setup data 和应用数据），但需要在 set config（打开 endpoint）之后使用。

```
/**
 * @brief interface API. receive usb device data packet.
 *
 * @param[in] ep, endpoint number, the highest bit represent direction, use USB_EP_DIRECT
 * @param[in] buffer, global buffer to hold data of the packet, it must be a DWORD-aligne
 * @param[in] size. For OUT transfers, the Transfer Size field in the endpoint Transfer S
 *            of the maximum packet size of the endpoint(eg, EP0 is 64byte), adjusted to
 * @param[in] flag. null
 * @param[out] return U_TRUE if successful, otherwise U_FALSE.
 */
extern USB_ERROR_TYPE_E USB_RecvData(uint8_t ep, void* buffer, uint16_t size, uint32_t fl
```

20.2.7.3 enable/disable ep

正常处理中不需要使用该 API，特殊情况下可以根据需求打开关闭某个特定的 endpoint

```
/**
 * @brief interface APIs. use this pair for enable/disable certain ep.
 *
 * @param[in] ep number with USB_EP_DIRECTION_IN/OUT.
 * @param[out] null
 */
extern void USB_EnableEp(uint8_t ep, USB_EP_TYPE_T type);
extern void USB_DisableEp(uint8_t ep);
```

20.2.7.4 usb close

USB 的 disable 请使用 bsp layer 中的 bsp_usb_disable()

```
/**
 * @brief interface API. shutdown usb module and reset all status data.
 *
 * @param[in] null.
 * @param[out] null.
 */
extern void USB_Close(void);
```

20.2.7.5 usb stall

```
/**
 * @brief interface API. set ep stall pid for current transfer
 *
 * @param[in] ep num with direction.
 * @param[in] U_TRUE: stall, U_FALSE: set back to normal
 * @param[out] null.
 */
extern void USB_SetStallEp(uint8_t ep, uint8_t stall);
```

20.2.7.6 usb In endpoint NAK

```
/**
 * @brief interface API. use this api to set NAK on a specific IN ep
 *
 * @param[in] U_TRUE: enable NAK on required IN ep. U_FALSE: stop NAK
 * @param[in] ep: ep number with USB_EP_DIRECTION_IN/OUT.
 * @param[out] null.
 */
extern void USB_SetInEndpointNak(uint8_t ep, uint8_t enable);
```

20.2.8 example 0: WINUSB

WinUSB 是适用于 USB 设备的通用驱动程序，随附在自 Windows Vista 起的所有 Windows 版本中。对于某些通用串行总线 (USB) 设备（例如只有单个应用程序访问的设备），可以将 WinUSB (Winusb.sys) 安装在设备的内核模式堆栈中作为 USB 设备的函数驱动程序而不是实现驱动程序。如果已将设备定义为 WinUSB 设备，Windows 会自动加载 Winusb.sys。

- 参考：`\ING\_SDK\sdk\src\BSP\bsp_usb.c`

首先调用 `bsp_usb_init()` 初始化 USB 模块, 之后的 USB 活动则全部在 `bsp_usb_event_handler()` 中处理。

```
#define FEATURE_WCID_SUPPORT
```

- device 需要在 enumeration 阶段提供 WCID 标识和相关 descriptor。示例中的 descriptor 实现如下：

```
#define USB_WCID_DESCRIPTOR_INDEX_4 \
{ \

#define USB_WCID_DESCRIPTOR_INDEX_5 \
{ \
```

- 通过修改 `USB_STRING_PRODUCT` 来改变产品名称 iproduct

```
#define USB_STRING_PRODUCT {16,0x3,'w',0,'i',0,'n',0,'-',0,'d',0,'e',0,'v',0}
```

第一个值是整个数组的长度，第二个值不变，之后是 16bit unicode 字符串（每个符号占用两个字节），该示例中 iproduct 为’win-dev’。

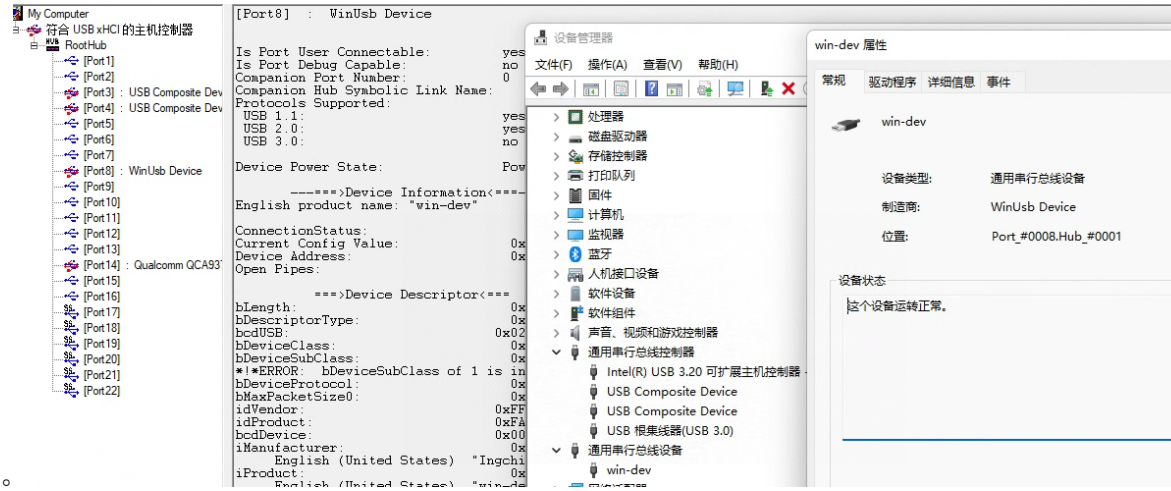
- 该示例中打开了两个 bulk endpoint，endpoint 1 为 input，endpoint 2 为 output, 最大包长为 64：

```
#define USB_EP_1_DESCRIPTOR \
{ \
  .size = sizeof(USB_EP_DESCRIPTOR_REAL_T), \
  .type = 5, \
  .ep = USB_EP_DIRECTION_IN(EP_IN), \
  .attributes = USB_EP_TYPE_BULK, \
  .mps = EP_X_MPS_BYTES, \
  .interval = 0 \
}

#define USB_EP_2_DESCRIPTOR \
{ \
  .size = sizeof(USB_EP_DESCRIPTOR_REAL_T), \
  .type = 5, \
  .ep = USB_EP_DIRECTION_OUT(EP_OUT), \
  .attributes = USB_EP_TYPE_BULK, \
  .mps = EP_X_MPS_BYTES, \
  .interval = 0 \
}
```

- 在 set configuration(USB_REQUEST_DEVICE_SET_CONFIGURATION) 之后，In/out endpoint 可以使用，通过 `USB_RecvData()` 配置 out endpoint 接收 host 发送的数据。在收到 host 的数据后（USB_CALLBACK_TYPE_RECEIVE_END），通过 `USB_SendData()` 将数据发送给 host（通过 in endpoint）。

- 在 WIN10 及以上的系统上，该设备会自动加载 winusb.sys 并枚举成 WinUsb Device, 设备



名称为“win-dev”。

- 通过 ing_usb.exe 可以对该设备进行一些简单数据测试 (ing_usb.exe VID:PID -w(write command) 2(transfer type(2==bulk)) 以及需要传输的数据（默认包长度为 endpoint 的 mps），数据会通过 out endpoint 发送给 USB Device 并通过 in endpoint 回环并打印出来)：

```
C:\Dropbox>ing_usb.exe FFF1:FA2F -w 2 0x1234 0x2345
Using libusb v1.0.25.11692

Opening device FFF1:FA2F...
       endpoint[0].address: 81
           max packet size: 0040
          polling interval: 00
       endpoint[1].address: 02
           max packet size: 0040
          polling interval: 00

Kernel driver attached for interface 0: -12

Claiming interface 0...

start transfer ...

  00000000  34 12 00 00 45 23 00 00 00 00 00 00 00 00 00 00  4...E#..........
  00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
type 2 send sucess,length: 64 bytes
 (0)  received 64 bytes

  00000000  34 12 00 00 45 23 00 00 00 00 00 00 00 00 00 00  4...E#..........
  00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  00000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

transfer end

Releasing interface 0...
Closing device...
```

- 使用 ing_usb.exe 可以读取 in endpoint 的数据（(ing_usb.exe VID:PID -r(read command) 2(transfer type(2==bulk))）, 但是 bsp layer 中需要做相应的修改（使用 in endpoint 发送数据给 host）。

20.2.9 example 1: HID composite

该示例实现了一个 mouse + keyboard 的复合设备，使用了两个独立的 interface，每个 interface 包含一个 In Endpoint。

- 参考：`\ING_SDK\sdk\src\BSP\bsp_usb_hid.c`

首先调用 `bsp_usb_init()` 初始化 USB 模块, 之后的 USB 活动则全部在 `bsp_usb_event_handler()` 中处理，report 的发送参考 `report data 发送`。

20.2.9.1 标准描述符

其标准描述符结构如下，configuration descriptor 之后分别是 interface descriptor, hid descriptor, endpoint descriptor。

```
typedef struct __attribute__((packed))
{
  USB_CONFIG_DESCRIPTOR_REAL_T config;
  USB_INTERFACE_DESCRIPTOR_REAL_T interface_kb;
  BSP_USB_HID_DESCRIPTOR_T hid_kb;
  USB_EP_DESCRIPTOR_REAL_T ep_kb[bNUM_EP_KB];
  USB_INTERFACE_DESCRIPTOR_REAL_T interface_mo;
  BSP_USB_HID_DESCRIPTOR_T hid_mo;
  USB_EP_DESCRIPTOR_REAL_T ep_mo[bNUM_EP_MO];
}BSP_USB_DESC_STRUCTURE_T;
```

上述描述符的示例在路径`\ING_SDK\sdk\src\BSP\bsp_usb_hid.h`，以 keyboard interface 为例：

```
#define USB_INTERFACE_DESCRIPTOR_KB \
{ \
  .size = sizeof(USB_INTERFACE_DESCRIPTOR_REAL_T), \
  .type = 4, \
  .interfaceIndex = 0x00, \
```

```
  .alternateSetting = 0x00, \
  .nbEp = bNUM_EP_KB,   \
  .usbClass = 0x03, \
  /* 0: no subclass, 1: boot interface */ \
  .usbSubClass = 0x00, \
  /* 0: none, 1: keyboard, 2: mouse */ \
  .usbProto = 0x00, \
  .iDescription = 0x00 \
}
```

其中可能需要根据场景修改的变量使用宏定义，其他则使用常量。请注意：该实现中没有打开boot function。

20.2.9.2 报告描述符

报告描述符的示例在路径`\ING_SDK\sdk\src\BSP\bsp_usb_hid.h`

- keyboard 的 report 描述符为：

```
#define USB_HID_KB_REPORT_DESCRIPTOR {  \
    0x05, 0x01, /* USAGE_PAGE (Generic Desktop)                  */  \
    0x09, 0x06, /* USAGE (Keyboard)                              */  \
    0xa1, 0x01, /* COLLECTION (Application)                       */  \
    0x05, 0x07, /*   USAGE_PAGE (Keyboard)                       */  \
    ...\ING_SDK\sdk\src\BSP\bsp_usb_hid.h
```

keyboard report descriptor 包含 8bit modifier input，8bit reserve, 5bit led output, 3bit reserve, 6bytes key usage id 。report 本地结构为：

```
#define KEY_TABLE_LEN (6)
typedef struct __attribute__((packed))
{
```

```
  uint8_t modifier;
  uint8_t reserved;
  uint8_t key_table[KEY_TABLE_LEN];
}BSP_KEYB_REPORT_s;
```

report 中的 usage id data 的实现分别在以下 enum 中：

```
BSP_KEYB_KEYB_USAGE_ID_e
BSP_KEYB_KEYB_MODIFIER_e
BSP_KEYB_KEYB_LED_e
```

- mouse 的 report 描述符为：

```
#define USB_HID_MOUSE_REPORT_DESCRIPTOR_SIZE (50)
#define USB_HID_MOUSE_REPORT_DESCRIPTOR { \
    0x05, 0x01, /* USAGE_PAGE (Generic Desktop)       */   \
    0x09, 0x02, /* USAGE (Mouse)                      */   \
    0xa1, 0x01, /* COLLECTION (Application)           */   \
    ...\ING_SDK\sdk\src\BSP\bsp_usb_hid.h
```

report 包含一个 3bit button(button 1 ~ button 3), 5bit reserve, 8bit x value, 8bit y value 结构为：

```
typedef struct __attribute__((packed))
{
  uint8_t button;/* 1 ~ 3 */
  int8_t pos_x;/* -127 ~ 127 */
  int8_t pos_y;/* -127 ~ 127 */
}BSP_MOUSE_REPORT_s;
```

20.2.9.3 standard/class request

ep0 的 request 处理在 event：USB_EVENT_EP0_SETUP。HID Class 相关的处理在 interface destination 下：USB_REQUEST_DESTINATION_INTERFACE。

以其中 keyboard report 描述符的获取为例：

- setup->wIndex 代表了 interface num，其中 0 为 keyboard interface（参考 BSP_USB_DESC_STRUCTURE_T）
- 使用 USB_SendData 发送 report 数据

```
case USB_REQUEST_DEVICE_GET_DESCRIPTOR:
{
  switch(((setup->wValue)>>8)&0xFF)
  {
    case USB_REQUEST_HID_CLASS_DESCRIPTOR_REPORT:
    {
      switch(setup->wIndex)
      {
        case 0:
        {
          size = sizeof(ReportKeybDescriptor);
          size = (setup->wLength < size) ? (setup->wLength) : size;

          status |= USB_SendData(0, (void*)&ReportKeybDescriptor, size, 0);
          KeybReport.pending = U_FALSE;
        }break;
        ...\ING_SDK\sdk\src\BSP\bsp_usb_hid.c
```

20.2.9.4 report data 发送

- keyboard key 的发送, 入参为一个键值以及其是否处于按下的状态，如果该键按下，则将其添加到 report 中并发送出去。

```
/**
 * @brief interface API. send keyboard key report
 *
 * @param[in] key: value comes from BSP_KEYB_KEYB_USAGE_ID_e
 * @param[in] press: 1: pressed, 0: released
 * @param[out] null.
 */
extern void bsp_usb_handle_hid_keyb_key_report(uint8_t key, uint8_t press);
```

- keyboard modifier 的发送，与 key 类似，区别是 modifier 是 bitmap data。

```
extern void bsp_usb_handle_hid_keyb_key_report(uint8_t key, uint8_t press);
/**
 * @brief interface API. send keyboard modifier report
 *
 * @param[in] modifier: value comes from BSP_KEYB_KEYB_MODIFIER_e
 * @param[in] press: 1: pressed, 0: released
 * @param[out] null.
 */
extern void bsp_usb_handle_hid_keyb_modifier_report(BSP_KEYB_KEYB_MODIFIER_e modifier
```

- keyboard led 的获取，该示例中没有 out endpoint，因此 led report 可能是用 ep0 的 set report 得到，参考：USB_REQUEST_HID_CLASS_REQUEST_SET_REPORT。
- mouse report 的发送, 入参分别为 x,y 的相对值和 button 的组合（按下为 1，释放为 0）。

```
/**
 * @brief interface API. send mouse report
 *
 * @param[in] x: 8bit int x axis value, relative,
 * @param[in] y: 8bit int y axis value, relative,
 * @param[in] btn: 8bit value, button 1 to 3,
```

```
 * @param[out] null.
 */
extern void bsp_usb_handle_hid_mouse_report(int8_t x, int8_t y, uint8_t btn);
```

第二十一章　看门狗（WATCHDOG）

21.1　功能概述

看门狗就是定期的查看芯片内部情况，一旦发生错误就向芯片内部发出重启信号。看门狗指令在程序的中断中拥有最高的优先级。防止程序跑飞，同时也能防止程序在线运行时出现死循环

21.2　使用说明

21.2.1　配置看门狗

使用 TMR_WatchDogEnable3 配置并启用看门狗。

```
void TMR_WatchDogEnable3(
  uint32_t int_timeout_ms,
  uint32_t reset_timeout_ms,
  uint8_t enable_int
  );
```

为了 ING916xx 和 ING918xx 统一接口，定义宏 TMR_WatchDogEnable 代替 TMR_WatchDogEnable3 。

```
#define TMR_WatchDogEnable(timeout) do {
  uint32_t TMR_CLK_FREQ = OSC_CLK_FREQ;
  uint32_t cnt = ((uint64_t)(timeout) * 1000 / OSC_CLK_FREQ);
  TMR_WatchDogEnable3(cnt, cnt, 0);
  } while (0)
```

例如：

21.2.2 重启看门狗

使用 TMR_WatchDogRestart 重启看门狗。

```
void TMR_WatchDogRestart(void);
```

21.2.3 清除中断

使用 TMR_WatchDogClearInt 清除看门狗的中断。

```
void TMR_WatchDogClearInt(void);
```

21.2.4 禁用看门狗

在使用 TMR_WatchDogEnable 启用或 TMR_WatchDogRestart 重启看门狗之后，需要使用 TMR_WatchDogDisable 对其禁用。

```
void TMR_WatchDogDisable(void);
```

第二十二章　内置 Flash（EFlash）

22.1　功能概述

芯片内置一定容量的 Flash，可编程擦写。擦除时以扇区（sector）为单位进行，写入时以 32bit 为单位。

22.2　使用说明

22.2.1　擦除并写入新数据

通过 `program_flash` 擦除并写入一段数据。

```
int program_flash(
    // 待写入的地址
    const uint32_t dest_addr,
    // 数据源的地址
    const uint8_t *buffer,
    // 数据长度（以字节为单位，必须是 4 的倍数）
    uint32_t size);
```

`dest_addr` 为统一编址后的地址，而非 Flash 内部从 0 开始的地址。`dest_addr` 必须对应于某个扇区的起始地址。数据源不可位于 Flash 内。

`program_flash` 将根据 `size` 自动擦除一个或多个扇区并写入数据。

本函数如果成功，则返回 0，否则返回非 0。

22.2.2 不擦除直接写入数据

通过 write_flash 不擦除直接写入数据。

```
int write_flash(
    // 待写入的地址
    const uint32_t dest_addr,
    // 数据源的地址
    const uint8_t *buffer,
    // 数据长度（以字节为单位，必须是 4 的倍数）
    uint32_t size);
```

dest_addr 为统一编址后的地址，必须 32bit 对齐。write_data 不擦除 Flash，而是直接写入。数据源不可位于 Flash 内。如果对应的 Flash 空间未被擦除，将无法写入。

本函数如果成功，则返回 0，否则返回非 0。

22.2.3 单独擦除

通过 erase_flash_sector 擦除一个指定的扇区。

```
int erase_flash_sector(
    // 待擦除的地址
    const uint32_t addr);
```

addr 必须对应于某个扇区的起始地址。本函数如果成功，则返回 0，否则返回非 0。

22.2.4 Flash 数据升级

通过 flash_do_update 可以升级 Flash 里的数据。这个函数可用于 FOTA 升级。

```
int flash_do_update(
    // 数据块数目
    const int block_num,
    // 每个数据块的信息
    const fota_update_block_t *blocks,
    // 用于缓存一个扇区的内存
    uint8_t *ram_buffer);
```

每个数据块的定为为：

```
typedef struct fota_update_block
{
    uint32_t src;
    uint32_t dest;
    uint32_t size;
} fota_update_block_t;
```

这个函数的行为大致如下：

```
flash_do_update() {
    for (block in blocks) {
        flash_copy(block.dest,
                   block.src,
                   block.size);
    }
}
```

如前所述，`program_flash` 的数据源不能位于 Flash，所以 `flash_copy` 需要把各扇区逐个读入 `ram_buffer`，然后使用 `program_flash` 擦除、写入。

这个函数如果成功，将自动重启系统，否则返回非 0。