



ING916XX 系列芯片外设开发者手册

桃芯科技（苏州）有限公司

官网: www.ingchips.com

邮箱: service@ingchips.com

电话: 010-85160285

地址: 北京市海淀区知春路 49 号紫金数 3 号楼 8 层 803

上海市浦东新区祥科路 58 号炬芯大厦 A 座 3 层 316

深圳市南山区科技园曙光大厦 1009

版权申明

本文档以及其所包含的内容为桃芯科技（苏州）有限公司所有，并受中国法律和其他可适用的国际公约的版权保护。未经桃芯科技的事先书面许可，不得在任何其他地方以任何形式（有形或无形的）以任何电子或其他方式复制、分发、传输、展示、出版或广播，不允许从内容的任何副本中更改或删除任何商标、版权或其他通知。违反者将对其违反行为所造成的任何以及全部损害承担责任，桃芯科技保留采取任何法律所允许范围内救济措施的权利。

目录

版本历史	xix
第一章 概览	1
1.1 缩略语及术语	1
1.2 参考文档	2
第二章 ADC 简介	3
2.1 功能描述	3
2.1.1 特点	3
2.1.2 ADC 模式	3
2.1.3 ADC 输入模式	4
2.1.4 ADC 转换模式	4
2.1.5 ADC 通道	4
2.1.6 PGA	5
2.1.7 输入电压范围	5
2.1.8 采样率	7
2.2 使用方法	7
2.2.1 时钟配置	7
2.2.2 ADC 精度初始化 & 校准	8
2.2.3 ADC 参数配置	9
2.2.4 ADC 数据处理	10

2.3	编程指南	10
2.3.1	驱动接口	10
2.3.2	代码示例	11
2.3.2.1	单次中断搬运	11
2.3.2.2	连续中断搬运	13
2.3.2.3	获取电压值 & 多通道采样	14
2.3.2.4	ADC & DMA 搬运	15
第三章	模拟比较器 (COMPARATOR)	17
3.1	功能概述	17
3.2	使用说明	18
3.2.1	模拟比较器模块初始化	18
3.2.2	获取模拟比较器结果	19
3.2.3	设置模拟比较器触发中断	19
3.2.4	设置模拟比较器睡眠唤醒	20
3.3	应用举例:	21
3.3.1	初始化模拟比较器模块	21
第四章	DMA 简介	25
4.1	功能描述	25
4.1.1	特点	25
4.1.2	搬运方式	25
4.1.3	搬运类型	26
4.1.4	中断类型	26
4.1.5	数据地址类型	26
4.1.6	数据方式	26
4.1.7	数据位宽	27
4.2	使用方法	27

4.2.1	方法概述	27
4.2.1.1	单次搬运	27
4.2.1.2	成串搬运	28
4.2.2	注意点	28
4.3	编程指南	28
4.3.1	驱动接口	28
4.3.2	代码示例	29
4.3.2.1	单次搬运	29
4.3.2.2	成串搬运	30
4.3.2.3	DMA 乒乓搬运	31
第五章	一次性可编程存储器 (eFuse)	35
5.1	功能概述	35
5.2	使用说明	35
5.2.1	模块初始化	35
5.2.2	按 bit 编程	35
5.2.3	按 word 编程	36
第六章	通用输入输出 (GPIO)	39
6.1	功能概述	39
6.2	使用说明	40
6.2.1	设置 IO 方向	40
6.2.2	读取输入	40
6.2.3	设置输出	41
6.2.4	配置中断请求	42
6.2.5	处理中断状态	43
6.2.6	输入去抖	43
6.2.7	低功耗保持状态	44
6.2.8	睡眠唤醒源	47

第七章 I2C 总线	49
7.1 功能概述	49
7.2 使用说明	49
7.2.1 方法 1 (blocking)	49
7.2.1.1 IO 配置	49
7.2.1.2 模块配置	50
7.2.2 方法 2 (Interrupt)	52
7.2.2.1 IO 配置	52
7.2.2.2 模块初始化	53
7.2.2.3 触发传输	54
7.2.2.4 中断配置	54
7.2.2.5 编程指南	55
7.2.3 时钟配置	83
第八章 I2S 简介	85
8.1 功能描述	85
8.1.1 特点	85
8.1.2 I2S 角色	86
8.1.3 I2S 工作模式	86
8.1.4 串行数据	86
8.1.5 时钟分频	86
8.1.5.1 时钟分频计算	86
8.1.6 I2S 存储器	87
8.2 使用方法	87
8.2.1 方法概述	87
8.2.2 注意点	89
8.3 编程指南	89
8.3.1 驱动接口	89

8.3.2	代码示例	89
8.3.2.1	I2S 配置	90
8.3.2.2	I2S 使能	90
8.3.2.3	I2S 中断	92
8.3.2.4	I2S & DMA 乒乓搬运	93
第九章 红外 (IR)		95
9.1	功能概述	95
9.2	使用说明	95
9.2.1	IO 配置	95
9.2.2	参数 (不同编码的时间参数)	96
9.2.3	红外发射接收	99
9.2.3.1	接收初始化	99
9.2.3.2	发射初始化	100
9.2.3.3	发射数据	101
9.2.3.4	中断实现 (以及红外接收)	102
第十章 硬件键盘扫描控制器 (KEYSCAN)		105
10.1	功能概述	105
10.2	使用说明	105
10.2.1	键盘矩阵的软件描述	106
10.2.2	KEYSCAN 模块初始化	107
10.2.3	获取扫描到的按键	108
10.3	应用举例	110
10.3.1	初始化 KEYSCAN 模块	110
10.3.2	中断数据处理	111
10.3.3	效果	115

第十一章 PDM 简介	117
11.1 功能描述	117
11.1.1 特点	117
11.1.2 PDM & PCM	117
11.2 使用方法	118
11.2.1 方法概述	118
11.2.2 注意点	118
11.3 编程指南	119
11.3.1 驱动接口	119
11.3.2 代码示例	119
11.3.2.1 PDM 结合 I2s:	119
11.3.2.2 PDM 数据 DMA 搬运	121
第十二章 管脚管理 (PINCTRL)	123
12.1 功能概述	123
12.2 使用说明	127
12.2.1 为外设配置 IO 管脚	127
12.2.2 配置上拉、下拉	128
12.2.3 配置驱动能力	129
12.2.4 配置天线切换控制管脚	129
12.2.5 配置模拟模式	130
第十三章 PTE 简介	133
13.1 功能描述	133
13.1.1 特点	133
13.1.2 PTE 原理图	133
13.1.3 功能	133
13.2 使用方法	134

13.2.1 方法概述	134
13.2.2 注意点	134
13.3 编程指南	135
13.3.1 src&dst 外设	135
13.3.2 驱动接口	136
13.3.3 代码示例	137
第十四章 增强型脉宽调制发生器 (PWM)	139
14.1 PWM 工作模式	139
14.1.1 最简单的模式: UP_WITHOUT_DIED_ZONE	140
14.1.2 UP_WITH_DIED_ZONE	140
14.1.3 UPDOWN_WITHOUT_DIED_ZONE	141
14.1.4 UPDOWN_WITH_DIED_ZONE	141
14.1.5 SINGLE_WITHOUT_DIED_ZONE	142
14.1.6 DMA 模式	142
14.1.7 输出控制	142
14.2 PCAP	143
14.3 PWM 使用说明	143
14.3.1 启动与停止	143
14.3.2 配置工作模式	144
14.3.3 配置门限	144
14.3.4 输出控制	145
14.3.5 综合示例	146
14.3.6 使用 DMA 实时更新配置	147
14.4 PCAP 使用说明	147
14.4.1 配置 PCAP 模式	147
14.4.2 读取计数器	149

第十五章 QDEC 简介	151
15.1 功能描述	151
15.1.1 特点	151
15.1.2 正转和反转	151
15.2 使用方法	152
15.2.1 方法概述	152
15.2.1.1 GPIO 选择	152
15.2.1.2 时钟配置	153
15.2.1.3 QDEC 参数配置	153
15.2.2 注意点	154
15.3 编程指南	155
15.3.1 驱动接口	155
15.3.2 代码示例	155
第十六章 实时时钟 (RTC)	157
16.1 功能描述	157
16.2 使用说明	157
16.2.1 RTC 使能	157
16.2.2 获取当前时间	157
16.2.3 修改时间	158
16.2.4 获取 RTC Counter 值	158
16.2.5 配置闹钟	159
16.2.6 配置中断请求	159
16.2.7 获取当前中断状态	160
16.2.8 清除中断	160
16.2.9 处理中断状态	161
16.2.10 睡眠唤醒源	161
16.2.11 数字调校	161

第十七章 串行外围设备接口 (SPI)	163
17.1 功能概述	163
17.2 使用说明	163
17.2.1 时钟以及 IO 配置	163
17.2.2 模块初始化	165
17.2.3 中断配置	166
17.2.4 编程指南	166
17.2.4.1 场景 1: 同时读写, 不使用 DMA	166
17.2.4.2 场景 2: 同时读写, 使用 DMA	170
17.2.5 其他配置	175
17.2.5.1 时钟配置 (pParam.eSclkDiv)	175
17.2.5.2 QSPI 使用	178
17.2.5.3 高速时钟和 IO 映射	183
第十八章 系统控制 (SYSCTRL)	185
18.1 功能概述	185
18.1.1 外设标识	185
18.1.2 时钟树	186
18.1.3 DMA 规划	188
18.2 使用说明	189
18.2.1 外设复位	189
18.2.2 时钟门控	189
18.2.3 时钟配置	189
18.2.4 DMA 规划	193
18.2.5 电源相关	193
18.2.5.1 内核电压	193
18.2.5.2 内置 Flash 电压	193
18.2.6 唤醒后的时钟配置	194

18.2.7 RAM 相关	195
第十九章 定时器 (TIMER)	199
19.1 功能概述	199
19.2 使用说明	199
19.2.1 设置 TIMER 工作模式	199
19.2.2 获取时钟频率	201
19.2.3 重载值	201
19.2.4 使能 TIMER	202
19.2.5 获取 TIMER 的比较值	203
19.2.6 获取 TIMER 的计数器值	203
19.2.7 计时器暂停	203
19.2.8 配置中断请求	203
19.2.9 处理中断状态	204
19.3 使用示例	205
19.3.1 使用计时器功能及暂停功能	205
19.3.2 使用 TIMER 的 PWM 功能	205
19.3.3 通道 0 产生 2 个周期性中断	206
19.3.4 产生 2 路对齐的 PWM 信号	206
第二十章 通用异步收发传输器 (UART)	209
20.1 功能概述	209
20.2 使用说明	209
20.2.1 设置波特率	209
20.2.2 获取波特率	210
20.2.3 UART 初始化	210
20.2.4 UART 轮询模式	212
20.2.5 UART 中断使能/禁用	213

20.2.6 处理中断状态	213
20.2.7 发送数据	214
20.2.8 接收数据	214
20.2.9 DMA 传输模式使能	214
20.3 示例代码	215
20.3.1 UART 接收变长字节数据	215
第二十一章 通用串行总线 (USB)	221
21.1 功能概述	221
21.2 使用说明	221
21.2.1 USB 软件结构	221
21.2.2 USB Device 状态	222
21.2.3 设置 IO	223
21.2.4 设置 PHY	223
21.2.5 USB 模块初始化	223
21.2.6 event handler	224
21.2.6.1 USB_EVENT_EP0_SETUP 的实现	226
21.2.6.2 SUSPEND 的处理	227
21.2.6.3 remote wakeup	227
21.2.7 常用 driver API	227
21.2.7.1 send usb data	227
21.2.7.2 receive usb data	228
21.2.7.3 enable/disable ep	228
21.2.7.4 usb close	228
21.2.7.5 usb stall	229
21.2.7.6 usb in endpoint nak	229
21.2.8 使用场景	230
21.2.8.1 example 0: WINUSB	230

21.2.8.2 example 1: HID composite	233
21.2.8.3 example 3: USB MSC	238
第二十二章 看门狗 (WATCHDOG)	239
22.1 功能概述	239
22.2 使用说明	239
22.2.1 配置看门狗	239
22.2.2 重启看门狗	242
22.2.3 清除中断	242
22.2.4 禁用看门狗	242
22.2.5 暂停看门狗	242
22.2.6 处理中断状态	243
第二十三章 内置 Flash (EFlash)	245
23.1 功能概述	245
23.2 使用说明	246
23.2.1 擦除并写入新数据	246
23.2.2 不擦除直接写入数据	246
23.2.3 单独擦除	247
23.2.4 Flash 数据升级	247

插图

2.1	916 时钟树 ADC 模块部分截图	8
8.1	I2S 控制器操作流程	88
13.1	PTE 原理图	134
15.1	QDEC 顺时针采集数据	152
15.2	QDEC 逆时针采集数据	152
22.1	WDT 阶段图	240

表格

1.1	缩略语	1
2.1	ADC 输入连接引脚	4
2.2	单端模式 PGA 电压范围表	6
2.3	差分模式 PGA 电压范围表	6
6.1	GPIO 的保持与唤醒功能	44
12.1	支持与常用 IO 全映射的常用功能管脚	123
12.2	其它外设功能管脚的映射关系	124
12.3	各外设的输入配置函数	128
12.4	管脚上下拉默认配置	129
12.5	支持 ADC 输入的管脚	131
18.1	各硬件外设的时钟源	188
18.2	默认参数对应的时钟频率	194
18.3	禁用时钟配置程序时重新唤醒后几个关键时钟的频率	195
18.4	可作为 SYS/SHARE RAM 的内存块	195
18.5	各软件包里的 SYS/SHARE RAM 配置	196
18.6	可用作高速缓存的内存块	197
23.1	部分芯片系列 Flash 电压与最高时钟频率	245

版本历史

版本	信息	日期
0.5	初始版本	2022-02-16
0.6	更新内容	2023-03-24
0.7	更新 ADC/DMA/IR/PDM/PTE/QDEC/USB/WATCHDOG	2023-04-19
0.8	更新 ADC/Comparator/GPIO/KeyScan/RTC	2023-06-08
0.9	更新 GPIO/RTC	2023-06-21

第一章 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。

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。

1.1 缩略语及术语

表 1.1: 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）
PLL	锁相环（Phase Locked Loop）
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）

1.2 参考文档

1. Bluetooth SIG¹
2. ING916XX 系列芯片数据手册²

¹<https://www.bluetooth.com/>

²<http://www.ingchips.com/product/70.html>

第二章 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。

2.1 功能描述

2.1.1 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~VBAT）
- 支持 APB 总线
- 采样频率可编程
- 支持单一转换模式和连续转换模式

2.1.2 ADC 模式

- 校准模式（calibration）：用于校准精度，分为单端模式校准和差分模式校准；
- 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。

2.1.3 ADC 输入模式

- 单端输入 (single-ended): 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。

2.1.4 ADC 转换模式

- 单次转换 (single): ADC 完成单次转换后, ADC 将停止, 数据将被拉入 FIFO;
- 连续转换 (continuous): ADC 经过 loop-delay 时间后循环进行转换, 直到手动关闭。

2.1.5 ADC 通道

ADC 共 12 个 channel, 即 ch0-ch11。

其中 ch0-ch8、ch10-ch11 为通用通道, ch9 为 1.2V 内部参考电压专用通道。

具体通道的输入连接引脚如下:

表 2.1: 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	GPIO35
ch9	VREF12_ADC_IN
ch10	GPIO30
ch11	GPIO31

通道输入模式配置规则：

1. 通用通道 ch0-ch8、ch10-ch11 均可配置成单端输入通道；
2. ch0-ch7 可以配置成 4 对差分输入通道，配置后对应差分通道 ch0-ch3，如差分通道 ch0 对应 ADC 通道 ch0 和 ch1，以此类推。

注意：配置 ADC 输入模式为差分模式下则只有 ch0-ch3 及 ch8-ch11，ch4-ch7 无意义。

以下是关于 ADC 通道对于开发者的几点使用建议：

1. 在使能通道前请先配置 ADC 输入模式；
2. 如需切换 ADC 输入模式，建议调用 `ADC_DisableAllChannels` 关闭之前模式下所有已使能通道，重新使能新的通道；
3. 建议一次只使用一个差分通道，如需同时使能两个差分通道可以选择 ch0 和 ch2；
4. 如需同时使用单端通道和差分通道建议进行方案优化。

对于同时使用单端通道和差分通道的情况，此处提供以下两种参考方案：

方案 1：单端、差分模式随配随用，即每次不同模式采样前需进行输入模式的切换；

方案 2：如只采用一路单端通道和一路差分通道，可以使能差分通道 ch0 和 ch2，其中一路的 VINN 接 0V 即可模拟单端输入。

2.1.6 PGA

ADC 的 PGA 默认开启，可调用 `ADC_PgaEnable(0)` 接口关闭 PGA。

ADC 的 PGA 有 6 个可配值，其大小为：

$$PGA_PARAM \in [0, 5]$$

目前 ch0-ch7 可以配置为上述任意 `PGA_PARAM`，ch8-ch11 的 `PGA_PARAM` 为固定值 1。

若 PGA 设置为关闭，单端模式时 `PGA_PARAM=1`，差分模式时 `PGA_PARAM=0`。（ch8-ch11 除外，仍为固定值）

注意：`PGA_PARAM` 是 PGA 的关键配置参数，`PGA_PARAM` 的选择请结合实际设备的电压范围计算获取，注意通过 PGA 放大后的电压极值不能超过参考电压阈值。

2.1.7 输入电压范围

不同 ADC 输入模式下的输入电压范围如下：

- 单端模式:

$$V_{IN} \in \left[\frac{V_{REFP}}{2} - \frac{V_{REFP}}{PGA_GAIN}, \frac{V_{REFP}}{2} + \frac{V_{REFP}}{PGA_GAIN} \right]$$

- 差分模式:

$$V_{INP} - V_{INN} \in \left[-\frac{V_{REFP}}{PGA_GAIN}, \frac{V_{REFP}}{PGA_GAIN} \right], V_{INP}, V_{INN} \geq 0$$

其中 PGA_GAIN 为:

$$PGA_GAIN = 2^{PGA_PARA}$$

注意: 请开发者确保输入电压满足以上范围要求, 过小的输入电压可能会使 ADC 无法正常工作, 而电压过大则可能会导致芯片损坏。

- 单端模式 PGA 电压范围如下表:

其中 Vmin、Vmax 指输入电压的最小值和最大值。

表 2.2: 单端模式 PGA 电压范围表

PGA_PARA	Vmin	Vmax	当 Vp=3.3 时 Vmin	当 Vp=3.3 时 Vmax
[000]	0	Vp	0	3.3
[001]	0	Vp	0	3.3
[010]	Vp/4	3Vp/4	0.825	2.475
[011]	3Vp/8	5Vp/8	1.2375	2.0625
[100]	7Vp/16	9Vp/16	1.44375	1.85625
[101]	15Vp/32	17Vp/32	1.546875	1.753125

- 差分模式 PGA 电压范围如下表:

其中 Vmin、Vmax 指两路输入电压差值 (即 VINP-VINN) 的最小值和最大值。

表 2.3: 差分模式 PGA 电压范围表

PGA_PARA	Vmin	Vmax	当 Vp=3.3 时 Vmin	当 Vp=3.3 时 Vmax
[000]	-Vp	Vp	-3.3	3.3
[001]	-Vp/2	Vp/2	-1.65	1.65
[010]	-Vp/4	Vp/4	-0.825	0.825
[011]	-Vp/8	Vp/8	-0.4125	0.4125

PGA_PARA	Vmin	Vmax	当 Vp=3.3 时 Vmin	当 Vp=3.3 时 Vmax
[100]	-Vp/16	Vp/16	-0.20625	0.20625
[101]	-Vp/32	Vp/32	-0.103125	0.103125

例 1. 输入电压在 1.3-1.7V 范围内波动，使用单端模式。则需要满足条件 $1.3 \geq V_{min}$ 且 $1.7 \leq V_{max}$ ，查表选择满足条件下最大 PGA_PARA=3。

例 2. 输入电压 VINP 在 1.1-1.7V 范围内波动，输入电压 VINN 在 1.0-1.2V 范围内波动，使用差分模式。计算得 $-0.1 \leq VINP - VINN \leq 0.7V$ ，需要满足条件 $-0.1 \geq V_{min}$ 且 $0.7 \leq V_{max}$ ，查表选择满足条件下最大 PGA_PARA=2。

选定 PGA_PARA 后，将其作为参数并调用接口（ADC_ConvCfg 或 ADC_PgaParaSet）完成配置。

差分模式下可以使 VINN=0V，这时输入的 VINP 相当于单端输入，但注意此时测量 VINP 的范围为 $0 - V_{BAT}/2$ 。如被测电压大于 $V_{BAT}/2$ 可考虑优化硬件方案，如分压等，或优化 ADC 采样策略。

2.1.8 采样率

采样率和时钟、loop-delay 大小以及使能通道个数有关，其计算关系如下：

当 loop-delay=0 时：

$$SAMPLERATE = \frac{ADC_CLK}{16 \times CH_NUM}$$

当 loop-delay>0 时：

$$SAMPLERATE = \frac{ADC_CLK}{loop_delay + 16 \times CH_NUM + 5}$$

2.2 使用方法

2.2.1 时钟配置

当前 ADC 所用时钟源为 clock_slow 或其经过分频得到的 ADC 工作时钟，建议选用后者。

当前 ADC 工作时钟可以配置范围为 500K-8M。

注意：由于 ADC 工作时钟须经过 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注意时钟规划，具体请参考 916 时钟树（图 2.1）。

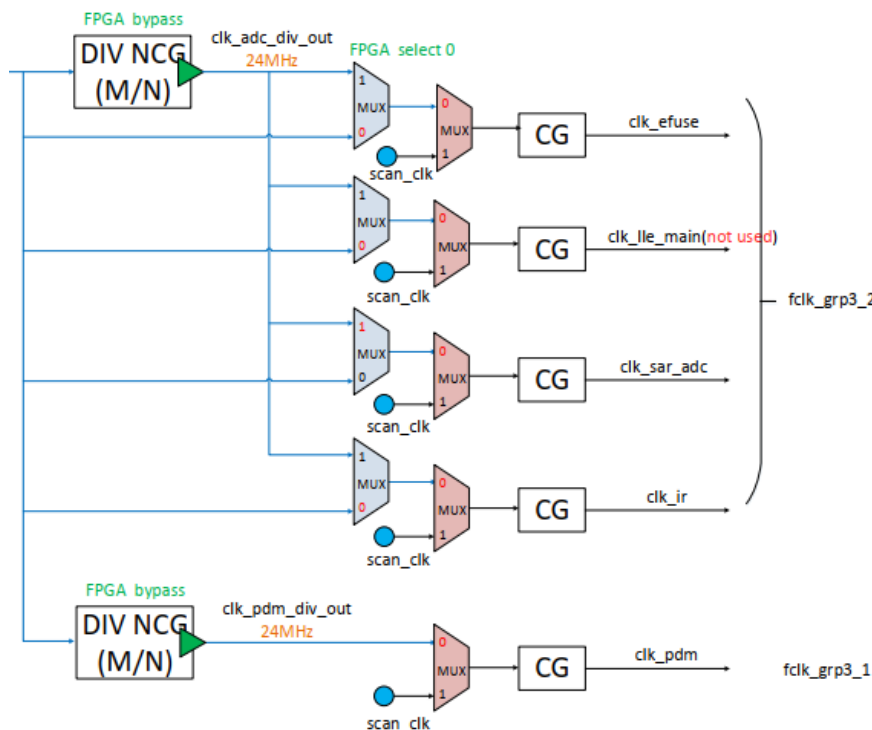


图 2.1: 916 时钟树 ADC 模块部分截图

2.2.2 ADC 精度初始化 & 校准

ADC 精度初始化接口: ADC_ftInit。

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。

有以下几点使用时的注意事项:

- ADC 精度初始化和 ADC 精度校准先后顺序不影响采样;
- 请先进行 ADC 精度初始化和 ADC 精度校准, 再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;
- ADC 精度初始化和精度校准只需要在初始化 ADC 模块时调用一次即可, 如调用 ADC_Reset 需重新进行 ADC 精度校准, 如调用 ADC_AdcClose 则需要重新完成整个初始化过程;
- 为提高参考电压可靠性, 建议周期性地进行内部参考电压校准, 如每次芯片唤醒后。但由于该校准过程有一定的耗时, 故不宜频繁调用;

- 每次进行内部参考电压校准后，需要重新配置 ADC 参数和使能通道；

2.2.3 ADC 参数配置

ADC 参数配置接口的函数声明如下：

```
void ADC_ConvCfg(SADC_adcCtrlMode ctrlMode,
                 SADC_pgaPara pgaPara,
                 uint8_t pgaEnable,
                 SADC_channelId ch,
                 uint8_t enNum,
                 uint8_t dmaEnNum,
                 SADC_adcInputMode inputMode,
                 uint32_t loopDelay);
```

涉及参数有：ADC 转换模式、PGA_PARA、PGA 使能开关、采样通道、data 触发中断数、data 触发 DMA 搬运数、ADC 输入模式和 loop-delay。

具体的参数取值范围请参考 ADC 头文件里对应的枚举定义或参数说明。

对于 pgaPara 的选取请参考本文档“PGA”和“输入电压范围”章节，或参考 peripheral_adc.h 文件中“ADC_PgaParaSet”的函数声明。

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。

注意：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 则默认选择触发中断的方式，DMA 配置不生效；

关于搬运方式的建议：

- 一般在小数据量情况下，如定时采集温度、电池电压，建议采用触发中断并 CPU 读数的方式。
- 一般大数据量连续采样，如模拟麦克风采样，建议采用 DMA 搬运方式（乒乓搬运），可以大大提高数据搬运处理效率。

注意：多次调用 ADC_ConvCfg 则以最后一次调用为准（除通道使能，不会自动关闭之前已使能的通道），如只需使能（关闭）ADC 通道可以通过 ADC_EnableChannel 接口完成。

2.2.4 ADC 数据处理

ADC 数据处理的推荐步骤为：

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。

注意：单端、差分模式得到的 ADC 数据范围均为 0-0x3fff，通过 ADC 数据计算得到的电压值会被限制在正/负参考电压范围内，且 ADC 数据在其范围内均匀分布。

另外对于单个数据的读取我们建议采用取若干数据求其平均值的方式，可以明显提高数据稳定性。

我们提供了方便开发者移植的求平均值程序，如有需求请参考 SDK 例程 peripheral_battery。

2.3 编程指南

2.3.1 驱动接口

初始化相关：

- ADC_ftInit: ADC 精度初始化
- ADC_Calibration: ADC 精度校准
- ADC_VrefCalibration: 内部参考电压校准

ADC 控制：

- ADC_Reset: ADC 复位
- ADC_Start: ADC 使能
- ADC_AdcClose: ADC 关闭

ADC 配置：

- ADC_ConvCfg: ADC 转换参数配置
- ADC_EnableChannel: 通道使能
- ADC_DisableAllChannels: 关闭所有通道

数据处理相关：

- ADC_GetFifoEmpty: 读取 FIFO 是否为空
- ADC_PopFifoData: 读取 FIFO 原始数据
- ADC_GetDataChannel: 读取原始数据中通道号
- ADC_GetData: 读取原始数据中 ADC 数据
- ADC_ReadChannelData: 读取特定通道 ADC 数据
- ADC_GetVol: 读取 ADC 数据对应电压值
- ADC_ClrFifo: 清空 FIFO

以上是 ADC 常用接口，还有部分接口不推荐直接使用，在此不进行罗列，详见头文件声明。

2.3.2 代码示例

下面展示 ADC 的基本用法：

（注：以下 ADC 参数设置仅供参考，具体参数请结合实际需要进行配置）

2.3.2.1 单次中断搬运

```
#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_Reset();
    ADC_ftInit();
    ADC_Calibration(SINGLE_END_MODE);
    ADC_ConvCfg(SINGLE_MODE, PGA_PARA_4, 1, ADC_CHANNEL, 1, 0, SINGLE_END_MODE, 0);
    platform_set_irq_callback(PLATFORM_CB_IRQ_SADC, ADC_cb_isr, 0);
    ADC_Start(1);
}
```

以上代码展示了 ADC 时钟配置、ADC 初始化 & 校准、ADC 转换参数配置，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;
}
```


在单通道采样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。

2.3.2.2 连续中断搬运

以上代码展示的是单次采样的数据搬运，如果是连续采样，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。

2.3.2.3 获取电压值 & 多通道采样

获取电压值需要对参考电压进行校准，并调用接口计算电压值。

下面我们使能 ch0、ch3、ch6 三个通道，并对 ch3 的采样值计算电压值：

```
#define ADC_CHANNEL_NUM      3
#define ADC_CHANNEL_0        ADC_CH_0
#define ADC_CHANNEL_3        ADC_CH_3
#define ADC_CHANNEL_6        ADC_CH_6
#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);
    uint16_t sample = ADC_GetData(data);
    if (channel == ADC_CHANNEL_0) {
        // do something with 'sample' of 'ADC_CHANNEL_0'
    } else if (channel == ADC_CHANNEL_3) {
        float voltage = ADC_GetVol(sample);
    } else if (channel == ADC_CHANNEL_6) {
        // do something with 'sample' of 'ADC_CHANNEL_6'
    }
    return 0;
}

void test(void)
{
    SYSCTRL_ClearClkGate(SYSCTRL_ITEM_APB_ADC);
    SYSCTRL_SetAdcClkDiv(24 / ADC_CLK_MHZ);
    SYSCTRL_ReleaseBlock(SYSCTRL_ITEM_APB_ADC);
    ADC_Reset();
    ADC_ftInit();
    ADC_Calibration(SINGLE_END_MODE);
    ADC_VrefCalibration();    // calibrate the referenced voltage
}
```

```

ADC_ConvCfg(SINGLE_MODE, PGA_PARA_4, 1, ADC_CHANNEL_0, ADC_CHANNEL_NUM, 0, SINGLE_END_MO
ADC_EnableChannel(ADC_CHANNEL_3, 1);    // call ADC_EnableChannel to enable more channel
ADC_EnableChannel(ADC_CHANNEL_6, 1);
platform_set_irq_callback(PLATFORM_CB_IRQ_SADC, ADC_cb_isr, 0);
ADC_Start(1);
}

```

voltage 即为最终计算得到的 ch3 采样电压值，ch0 和 ch6 采样值可执行其他操作。

2.3.2.4 ADC & DMA 搬运

对连续大量的数据推荐采用 ADC & DMA 搬运方式。

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 ADC_CLK_MHZ          6
#define SAMPLERATE           16000
#define ADC_CHANNEL_NUM      1
#define LOOP_DELAY(c, s, ch) (((c) * (1000000)) / (s)) - (((16) * (ch)) + (5))
#define DMA_CHANNEL          0

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_ClearClkGate(SYSCTRL_ITEM_APB_ADC);
    SYSCTRL_SetAdcClkDiv(24 / ADC_CLK_MHZ);
    SYSCTRL_ReleaseBlock(SYSCTRL_ITEM_APB_ADC);
    ADC_Reset();
    ADC_ftInit();
    ADC_Calibration(DIFFERENTAIL_MODE);
    ADC_VrefCalibration();
    ADC_ConvCfg(CONTINUES_MODE, PGA_PARA_4, 1, ADC_CHANNEL, 0, 8, DIFFERENTAIL_MODE,
        LOOP_DELAY(ADC_CLK_MHZ, SAMPLERATE, ADC_CHANNEL_NUM));

    SYSCTRL_ClearClkGateMulti((1 << SYSCTRL_ITEM_APB_DMA));
    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);
}
```

更多 ADC 程序请参考以下 SDK 例程:

peripheral_battery: 包括电压值采样, 可移植的多次采样求平均值程序

voice_remote_ctrl: ADC 模拟 mic 音频数据采集处理

第三章 模拟比较器 (COMPARATOR)

3.1 功能概述

模拟比较器比较两个输入电压，输出高电平或低电平。

特性：

- 支持轨到轨输入。
- 4 种功耗模式：超低功耗模式 (低速超低功耗模式模块功耗低至 400nA)、低功耗模式 (模块典型功耗为 5uA)、中等功耗模式 (模块典型功耗为 40uA)、高功耗模式 (模块典型功耗为 200uA)。
- 精确处理低至 20mV 电压。
- 支持 8 通道 VINP 和 6 通道 VINN，VINN 其中一个通道来自内部参考电压。
- VINN 支持 16 级阶梯。

3.2 使用说明

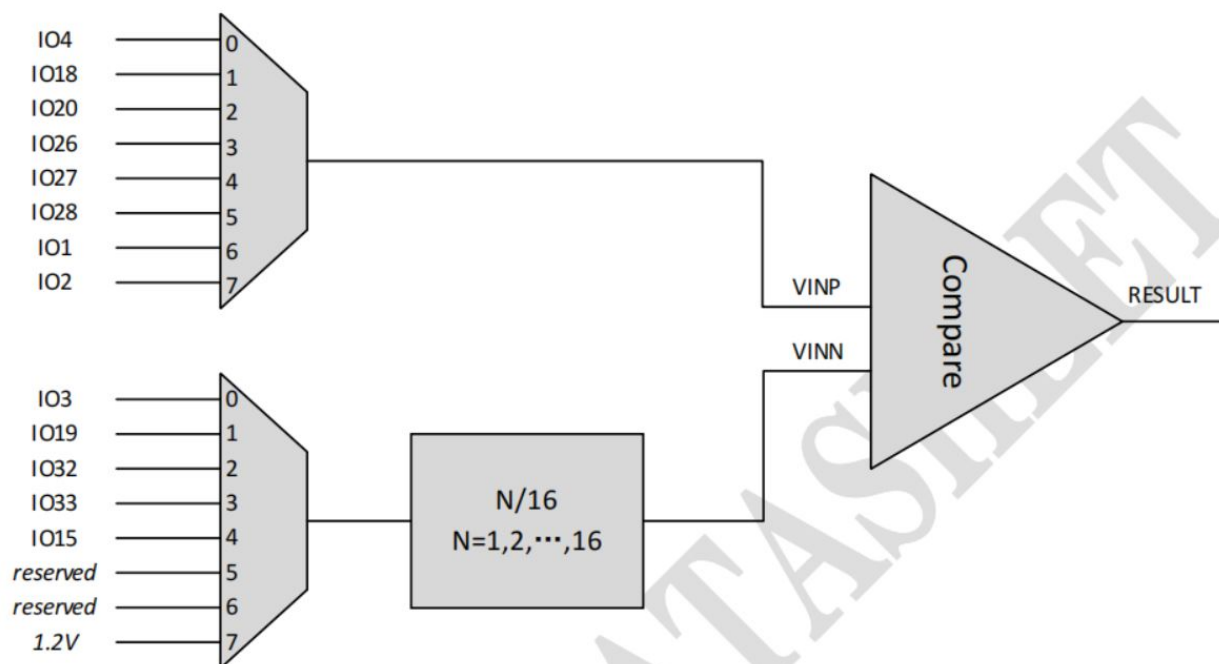


图 3-20 比较器结构

3.2.1 模拟比较器模块初始化

```
typedef struct {
    COMPARATOR_Vinp_t v_in_p;
    COMPARATOR_Vinn_t v_in_n;
    COMPARATOR_Enable_t en;
    COMPARATOR_HysteresisValtage_t hysteresis;
    COMPARATOR_VinnDivision_t vinn_division;
    COMPARATOR_WorkMode_t work_mode;
    COMPARATOR_OutputPolarity_t output_polarity;
} COMPARATOR_SetStateStruct;
```

```
/**
 * @brief Initialize comparator module
 *
 * @param[in] cmp_set          Initial parameter struct
 */
void COMPARATOR_Initialize(const COMPARATOR_SetStateStruct* cmp_set);
```

3.2.2 获取模拟比较器结果

模拟比较器模块输出结果的极性可以配置。

如果 output_polarity 配置为 0：如果 VINP 电压高于 VINN，比较器返回的结果是 1；如果 VINP 电压低于 VINN，比较器返回的结果是 0。

如果 output_polarity 配置为 1：如果 VINP 电压高于 VINN，比较器返回的结果是 0；如果 VINP 电压低于 VINN，比较器返回的结果是 1。

```
/**
 * @brief Get comparator result
 *
 * @return          comparator result
 */
uint8_t COMPARATOR_GetComparatorResult(void);
```

3.2.3 设置模拟比较器触发中断

模拟比较器模块有 2 个中断源，输出结果的上升沿中断和输出结果的下降沿中断，中断标志需要手动清除。

如果使能了模拟比较器模块的上升沿中断，当输出结果从 0 变为 1 时，会触发 POSEDGE 中断。

如果使能了模拟比较器模块的下降沿中断，当输出结果从 1 变为 0 时，会触发 NEGEDGE 中断。

上升沿或者下降沿触发中断可以单独配置，也可配成双沿触发。

```
typedef enum
{
    COMPARATOR_DISABLE_INT_MODE          = 0,
    COMPARATOR_NEGEDGE_INT_MODE           ,
    COMPARATOR_POSEDGE_INT_MODE           ,
    COMPARATOR_POSEDGE_AND_NEGEDGE_INT_MODE ,
} COMPARATOR_INTERRUPT_MODE_t;

/**
 * @brief Set comparator as interrupt source
 *
 * @param[in] int_mode          COMPARATOR_INTERRUPT_MODE_t
 * @return          0:OK, other:error
 */
int COMPARATOR_SetInterrupt(COMPARATOR_INTERRUPT_MODE_t int_mode);

/**
 * @brief Get comparator interrupt status and clear interrupt flag
 *
 * @return          interrupt status
 */
COMPARATOR_INTERRUPT_MODE_t COMPARATOR_GetIntStatusAndClear(void);
```

3.2.4 设置模拟比较器睡眠唤醒

模拟比较器模块可以作为唤醒源唤醒睡眠中的芯片。

根据模拟比较器模块输出结果的电平状态，可以配置成高电平或者低电平唤醒芯片；也可配置成边沿唤醒，上升沿、下降沿、双沿唤醒均可。

```
typedef enum
{
    COMPARATOR_HIGH_LEVEL_WAKEUP = 0,
    COMPARATOR_LOW_LEVEL_WAKEUP  ,
```



```

} COMPARATOR_LEVEL_WAKEUP_t;

typedef enum
{
    COMPARATOR_LEVEL_WAKEUP_MODE          = 0,
    COMPARATOR_POSEDGE_WAKEUP_MODE        ,
    COMPARATOR_NEGEDGE_WAKEUP_MODE        ,
    COMPARATOR_POSEDGE_AND_NEGEDGE_WAKEUP_MODE ,
} COMPARATOR_WAKEUP_MODE_t;

/**
 * @brief Set comparator as wakeup source from DEEP SLEEP mode
 *
 * @param[in] enable          0:disable, 1:enable
 * @param[in] level           COMPARATOR_LEVEL_WAKEUP_t
 * @param[in] wakeup_mode     COMPARATOR_WAKEUP_MODE_t
 * @return                   0:OK, other:error
 */
int COMPARATOR_SetDeepSleepWakeupSource(uint8_t enable,
                                         COMPARATOR_LEVEL_WAKEUP_t level,
                                         COMPARATOR_WAKEUP_MODE_t wakeup_mode);

```

3.3 应用举例：

3.3.1 初始化模拟比较器模块

模拟比较器模块的初始化配置如 `cmp_set`, GPIO18 作为 VINP, GPIO19 作为 VINN, 30mV 的迟滞, VINN 没有分压 (配置为 16/16), 超低功耗模式, 输出极性配置为 0。

例子中模拟比较器模块使能了上升沿中断, 使能了上升沿睡眠唤醒。

```

static uint32_t lpc_pos_cb_isr(void *user_data)
{

```

```

    COMPARATOR_INTERRUPT_MODE_t status = COMPARATOR_GetIntStatusAndClear();
    printf("pos isr status %d %d\r\n", status, COMPARATOR_GetComparatorResult());
    return 0;
}

static uint32_t lpc_neg_cb_isr(void *user_data)
{
    COMPARATOR_INTERRUPT_MODE_t status = COMPARATOR_GetIntStatusAndClear();
    printf("neg isr status %d %d\r\n", status, COMPARATOR_GetComparatorResult());
    return 0;
}

const static COMPARATOR_SetStateStruct cmp_set = {
    .v_in_p      = COMPARATOR_VINP1_GPIO18,
    .v_in_n      = COMPARATOR_VINN1_GPIO19,
    .en          = COMPARATOR_ENABLE,
    .hysteresis  = COMPARATOR_30mV_HYSTERRESIS,
    .vinn_division = COMPARATOR_16_16,
    .work_mode    = COMPARATOR_ULTRA_LOW_POWER,
    .output_polarity = COMPARATOR_OUT1_P_GREATER_N,
};

static void setup_peripherals_comparator(void)
{
    SYSCTRL_ClearClkGateMulti(1 << SYSCTRL_ITEM_APB_PinCtrl);
    SYSCTRL_ClearClkGateMulti(1 << SYSCTRL_ITEM_APB_GPIO0);
    SYSCTRL_ClearClkGateMulti(1 << SYSCTRL_ITEM_APB_GPIO1);
    SYSCTRL_ClearClkGateMulti(1 << SYSCTRL_ITEM_APB_LPC);

    COMPARATOR_Initialize(&cmp_set);

    platform_set_irq_callback(PLATFORM_CB_IRQ_LPC_POS, lpc_pos_cb_isr, 0);
    platform_set_irq_callback(PLATFORM_CB_IRQ_LPC_NEG, lpc_neg_cb_isr, 0);
    COMPARATOR_SetInterrupt(COMPARATOR_POSEDGE_INT_MODE);

    COMPARATOR_SetDeepSleepWakeupSource(1, COMPARATOR_HIGH_LEVEL_WAKEUP, COMPARATOR_POSEDGE_WAKEUP);
    return;
}

```

```
}
```


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

- 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

DMA 搬运前需要对数据源和数据目的地址的数据方式进行配置。

数据方式的选择有如下建议：

1. 从内存搬运数据选择 normal mode;
2. 从外设 FIFO 搬运数据选择 handshake mode, 同时要和外设协商好 BurstSize, 支持 2^n ($n = 0-7$) 大小的 BurstSize。

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, 以及搬运类型, 即 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：

```

#define CHANNEL_ID 0
char src[] = "hello world!";
char src1[] = "I am ING916.";
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 ING916.”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

更多程序开发者可以参考 voice_remote_ctrl 例程。

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 know 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 一次性可编程存储器 (eFuse)

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);
    //pos is bit position from 0 to 127
    EFUSE_WriteEfuseDataBitToOne(APB_EFUSE, pos);
    // 等待结束
    while(1 == EFUSE_GetStatusBusy(EFUSE_BASE));
    EFUSE_Lock(EFUSE_BASE);

    //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));
    //通过 EFUSE_GetEfuseData() 读取数据
}
```

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));
//通过 EFUSE_GetEfuseData() 读取数据
}
```


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 $(1 \ll 0) \mid (1 \ll 5)$ 。

- 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

```
GPIO_ConfigIntSource(GPIO_GPIO_0,
    ...LOGIC_HIGH_OR_RISING_EDGE | ..._HIGH_OR_RISING_EDGE,
    GPIO_INT_EDGE);
```

- 例如将 GPIO 0 配置为高电平触发

```
GPIO_ConfigIntSource(GPIO_GPIO_0,
    ...LOGIC_HIGH_OR_RISING_EDGE,
    GPIO_INT_LOGIC);
```

6.2.5 处理中断状态

用 `GPIO_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 `GPIO_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 `GPIO_ClearIntStatus` 消除某个 GPIO 上中断状态，用 `GPIO_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 `GPIO_DebounceCtrl` 配置输入去抖参数，每个 GPIO 硬件模块使用单独的参数：

```
void GPIO_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,
    GPIO_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
{
    GPIO_DB_CLK_32K,    // 使用 32k 时钟
    GPIO_DB_CLK_PCLK,   // 使用快速 PCLK
} GPIO_DbClk_t;
```

快速 PCLK 的具体频率参考SYSCTRL。

通过 GPIO_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”：

```
GPIO_DebounceCtrl(1, 4, GPIO_DB_CLK_32K);
GPIO_DebounceEn(GPIO_GPIO0_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，总结于表 6.1。

表 6.1: GPIO 的保持与唤醒功能

序号	分类	低功耗保持	DEEP 唤醒源	DEEPER 唤醒源
0	A	Y	Y	Y
1	B	Y	Y	
2	B	Y	Y	
3	B	Y	Y	
4	B	Y	Y	
5	A	Y	Y	Y
6	A	Y	Y	Y
7	B	Y	Y	
8	B	Y	Y	

序号	分类	低功耗保持	DEEP 唤醒源	DEEPER 唤醒源
9	B	Y	Y	
10	B	Y	Y	
11	B	Y	Y	
12	B	Y	Y	
13	B	Y	Y	
14	B	Y	Y	
15	B	Y	Y	
16	B	Y	Y	
17	B	Y	Y	
18	C			
19	C			
20	C			
21	A	Y	Y	Y
22	A	Y	Y	Y
23	A	Y	Y	Y
24	B	Y	Y	
25	B	Y	Y	
26	C			
27	C			
28	C			
29	B	Y	Y	
30	B	Y	Y	
31	B	Y	Y	
32	B	Y	Y	
33	B	Y	Y	
34	B	Y	Y	
35	B	Y	Y	
36	A	Y	Y	Y
37	A	Y	Y	Y
38	B	Y		
39	B	Y		
40	B	Y		
41	B	Y		

- 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（输出值 —— 对于 IO 方向为输出的 GPIO、上下拉）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。

说明：对于 IO 方向为输入的 GPIO，使能低功耗状态保持功能并进入低功耗状态后，确实可以保持其 IO 输入功能，但是并不能产生实际效果（产生中断或者唤醒系统）。

```
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);
```

这两个功能是互斥的，比如先后调用这两个函数，先使能保持再使能高阻，则只有高阻功能生效。

- 对于 C 型 GPIO

不支持低功耗保持。



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，包括所有的 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，参见表 6.1。

- 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；`mode` 为唤醒方式；对于 A 型 GPIO，忽略 `pull` 参数，其上下拉由 `PINCTRL_Pull` 控制。共支持以下 5 种唤醒方式：

- `GIO_WAKEUP_MODE_LOW_LEVEL`：通过低电平唤醒；
- `GIO_WAKEUP_MODE_HIGH_LEVEL`：通过高电平唤醒；
- `GIO_WAKEUP_MODE_RISING_EDGE`：通过上升沿唤醒；
- `GIO_WAKEUP_MODE_FALLING_EDGE`：通过下降沿唤醒；
- `GIO_WAKEUP_MODE_ANY_EDGE`：通过任意边沿唤醒，即上升沿或下降沿皆可。

当使用电平唤醒时，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。当使用边沿唤醒时，注意脉冲需要维持至少 $100\ \mu s$ 。

对于 B 型 GPIO，唤醒源与低功耗保持为两套独立的电路，因此：1）上下拉独立于 `PINCTRL_Pull`，而且一直生效——无论是否处于低功耗状态，所以，不要用 `PINCTRL_Pull` 配置相反的上下拉；2）使能或禁用 B 型 GPIO 的低功耗保持或者高阻功能不影响这里的唤醒源设置；3）不要将某 IO 同时设为输出和唤醒源，比如将某 IO 同时设为输出高电平、高电平唤醒，使能保持功能并进入低功耗时，这个 IO 上保持电路所输出的高电平将传输到唤醒源电路并触发唤醒。

```
int GIO_EnableDeepSleepWakeupSource(  
    GIO_Index_t io_index,      // GPIO 编号  
    uint8_t enable,           // 使能 (1)/禁用 (0)  
    uint8_t mode,             // 触发方式
```

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

第七章 I2C 总线

I2C(Inter — Integrated Circuit) 是一种通用的总线协议。它是一种只需要两个 IO 并且支持多主多从的双向两线制总线协议标准。

7.1 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

7.2 使用说明

I2C 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 读取外设的单一场景。
- 方法 2：使用I2C 中断操作 I2C，需要在中断中操作读写的的数据。

I2C Slave 则需要使用方法 2，以中断方式操作。

7.2.1 方法 1（blocking）

7.2.1.1 IO 配置

1. IO 选择，并非所有 IO 都可以映射成 I2C，请查看对应 datasheet 获取可用 IO。

2. 操作模块之前需要打开对应模块的时钟。使用 `SYSCTRL_ClearClkGateMulti()` 打开时钟。请查看下述代码示例，需要注意的是：

- `SYSCTRL_ITEM_APB_I2C0` 对应 I2C0，如果使用 I2C1 需要对应修改。

3. I2C IO 需要配置为默认上拉，芯片内置上拉可以通过 `PINCTRL_Pull()` 实现（已经包含在 `PINCTRL_SelI2cIn()` 中）。实际应用中建议在外部实现上拉（可以获得更快的响应速度和时钟）。

4. 将选定 IO 映射到 I2C 模块，两个 IO 均需要配置为双向（输入 + 输出）。请参考下述代码实现（`PINCTRL_SelI2cIn()` 中包含了输入 + 输出的配置）。

以下示例可以将指定 IO 映射成 I2C 引脚：

```
#define I2C_SCL          GPIO_GPIO_10
#define I2C_SDA          GPIO_GPIO_11

void setup_peripherals_i2c_pin(void)
{
    SYSCTRL_ClearClkGateMulti( (1 << SYSCTRL_ITEM_APB_I2C0)
                               | (1 << SYSCTRL_ITEM_APB_PinCtrl));
    PINCTRL_SelI2cIn(I2C_PORT_0, I2C_SCL, I2C_SDA);
}
```

7.2.1.2 模块配置

- 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 length);

/**
 * @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 write_len, uint8_t *byte_data, int16_t length);
```

- 使用方法:
 - 配置 IO。
 - 初始化 I2C 模块:

```
i2c_init(I2C_PORT_0);
```

– 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。

7.2.2 方法 2 (Interrupt)

I2C Slave 以及 I2C Master 方法 2 需要使用 Interrupt 方式。

7.2.2.1 IO 配置

1. IO 选择, 并非所有 IO 都可以映射成 I2C, 请查看对应 datasheet 获取可用 IO。
2. 操作模块之前需要打开对应模块的时钟。使用 SYSCTRL_ClearClkGateMulti() 打开时钟。请查看下述代码示例, 需要注意的是:
 - SYSCTRL_ITEM_APB_I2C0 对应 I2C0, 如果使用 I2C1 需要对应修改。
3. I2C IO 需要配置为默认上拉, 芯片内置上拉, 可以通过 PINCTRL_Pull() 实现 (已经包含在 PINCTRL_SelI2cIn() 中)。实际应用中建议在外部实现上拉 (可以获得更快的响应速度和时钟)。
4. 将选定 IO 映射到 I2C 模块, 两个 IO 均需要配置为双向 (输入 + 输出)。请参考下述代码实现 (PINCTRL_SelI2cIn() 中包含了输入 + 输出的配置)。

5. 如果需要使用中断，使用 `platform_set_irq_callback()` 配置应用中断。

以下示例可以将指定 IO 映射成 I2C 引脚，并配置了中断回调函数：

```
#define I2C_SCL          GPIO_GPIO_10
#define I2C_SDA          GPIO_GPIO_11

void setup_peripherals_i2c_pin(void)
{
    SYSCTRL_ClearClkGateMulti( (1 << SYSCTRL_ITEM_APB_I2C0)
                               | (1 << SYSCTRL_ITEM_APB_PinCtrl));

    PINCTRL_SelI2cIn(I2C_PORT_0, I2C_SCL, I2C_SDA);

    platform_set_irq_callback(PLATFORM_CB_IRQ_I2C0, peripherals_i2c_isr, NULL);
}
```

7.2.2.2 模块初始化

I2C 模块初始化需要通过以下 API 来实现：

1. 通过 `I2C_Config()` 选择 Master/Slave 角色，以及 I2C 地址。
2. 使用 `I2C_ConfigClkFrequency()` 更改时钟配置。
3. 根据使用场景打开相应的中断 `I2C_IntEnable()`：
 - `I2C_INT_CMPL`：该中断的触发代表传输结束。
 - `I2C_INT_FIFO_FULL`：代表 RX FIFO 中有数据。
 - `I2C_INT_FIFO_EMPTY`：TX FIFO 空，需要填充发送数据。
 - `I2C_INT_ADDR_HIT`：总线上检测到了匹配的地址。
4. 使能 I2C 模块 `I2C_Enable()`。

7.2.2.3 触发传输

1. 调用 `I2C_CtrlUpdateDirection()` 设置传输方向。
 - `I2C_TRANSACTION_SLAVE2MASTER`: Slave 发送数据, Master 读取数据。
 - `I2C_TRANSACTION_MASTER2SLAVE`: Master 发送数据, Slave 读取数据。
2. 通过 `I2C_CtrlUpdateDataCnt()` 设置该次传输的数据大小, 最大 8 个 bit (256 字节), 以字节为单位。
3. 使用 `I2C_CommandWrite()` 触发 I2C 传输:
 - `I2C_COMMAND_ISSUE_DATA_TRANSACTION`: Master 有效, 触发数据传输。
 - `I2C_COMMAND_RESPOND_ACK`: 在接收到的字节后发送一个 ACK。
 - `I2C_COMMAND_RESPOND_NACK`: 在接收到的字节后发送一个 NACK。
 - `I2C_COMMAND_CLEAR_FIFO`: 清空 FIFO。
 - `I2C_COMMAND_RESET`: reset I2C 模块。

7.2.2.4 中断配置

数据的读写需要在中断中进行。

1. 在中断触发后, 通过 `I2C_GetIntState()` 来读取中断状态。不同状态需要参考 `I2C_STATUS_xxx` 定义。
 - `I2C_STATUS_FIFO_FULL`: 读取数据, 并通过 `I2C_FifoEmpty()` 判断 FIFO 状态。
 - `I2C_STATUS_FIFO_EMPTY`: 填充数据, 并通过 `I2C_FifoFull()` 判断 FIFO 状态。
 - `I2C_STATUS_CMPL`: 判断 FIFO 中是否有剩余数据并读取。
 - `I2C_STATUS_ADDRHIT`: 地址匹配, 在该中断中通过 `I2C_GetTransactionDir()` 判断传输的方向。
 - `I2C_TRANSACTION_MASTER2SLAVE`: 代表 Master 发送, Slave 则需要读取数据。
 - `I2C_TRANSACTION_SLAVE2MASTER`: 代表 Slave 需要发送, Master 读取数据。
2. FIFO 相关中断不需要清除标志, 其余中断需要通过 `I2C_ClearIntState()` 来清除标志, 避免中断重复触发。

7.2.2.5 编程指南

7.2.2.5.1 场景 1: Master 只读, Slave 只写, 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 IO, 请参考 IO 配置。

7.2.2.5.1.1 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;
```

- 初始化 I2C 模块

此处配置为 Master, 7bit 地址, 并打开了传输结束中断和 FIFO FULL 中断。

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
    I2C_Config(APB_I2C0, I2C_ROLE_MASTER, I2C_ADDRESSING_MODE_07BIT, ADDRESS);
    I2C_ConfigClkFrequency(APB_I2C0, I2C_CLOCKFREQUENCY_STANDARD);
    I2C_Enable(APB_I2C0, 1);
    I2C_IntEnable(APB_I2C0, (1 << I2C_INT_CMPL) | (1 << I2C_INT_FIFO_FULL));
}
```

- Master 中断实现

中断中通过 I2C_STATUS_FIFO_FULL 来读取接收到的数据。当 I2C_STATUS_CMPL 触发时, 当前传输结束, 需要判断 FIFO 中有没有剩余数据。

```
static uint32_t peripherals_i2c_isr(void *user_data)
{
    uint8_t i;
```

```

uint32_t status = I2C_GetIntState(APB_I2C0);

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);
    }
}

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;
}

```

- Master 触发传输

首先需要设置传输方向，此处是 I2C_TRANSACTION_SLAVE2MASTER，即代表 Slave 发送数据，Master 读取数据。

```

void peripheral_i2c_send_data(void)
{
    I2C_CtrlUpdateDirection(APB_I2C0, I2C_TRANSACTION_SLAVE2MASTER);
    I2C_CtrlUpdateDataCnt(APB_I2C0, DATA_CNT);
    I2C_CommandWrite(APB_I2C0, I2C_COMMAND_ISSUE_DATA_TRANSACTION);
}

```

- 使用流程
 - 设置 IO, `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。

7.2.2.5.1.2 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;
```

- 初始化 I2C 模块

对于 Slave, 需要打开 `I2C_INT_ADDR_HIT`, 此中断的触发代表收到了匹配的地址。

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
    I2C_Config(APB_I2C0, I2C_ROLE_SLAVE, I2C_ADDRESSING_MODE_07BIT, ADDRESS);
    I2C_Enable(APB_I2C0, 1);
    I2C_IntEnable(APB_I2C0, (1<<I2C_INT_ADDR_HIT) | (1<<I2C_INT_CMPL));
}
```

- Slave 中断实现以及发送数据

1. 首先需要等待 `I2C_STATUS_ADDRHIT` 中断, 在该中断中通过 `I2C_GetTransactionDir()` 判断传输的方向。
 - `I2C_TRANSACTION_MASTER2SLAVE`: 代表 Slave 需要读取数据, 打开 `I2C_INT_FIFO_FULL` 中断。
 - `I2C_TRANSACTION_SLAVE2MASTER`: 代表 Slave 需要发送, 打开 `I2C_INT_FIFO_EMPTY`。

2. 如果是 Slave 写操作，则会触发 I2C_STATUS_FIFO_EMPTY 中断，此时填写需要发送的数据，直到 FIFO 满。
3. 等待 I2C_STATUS_CMPL 中断，该中断的触发代表传输结束，在中断中关闭打开的 FIFO 中断。

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

    if(status & (1 << I2C_STATUS_ADDRHIT))
    {
        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));
    }

    if(status & (1 << I2C_STATUS_FIFO_EMPTY))
    {
        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(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;
}

```

- 使用流程:

- 设置 IO, 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。

7.2.2.5.2 场景 2: Master 只写, Slave 只读, 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 IO, 请参考 IO 配置。

7.2.2.5.2.1 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;
```

- 初始化 I2C 模块

此处配置为 Master, 7bit 地址, 并打开了传输结束中断和 FIFO EMPTY 中断。I2C_INT_FIFO_EMPTY 中断用来发送数据。

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
    I2C_Config(APB_I2C0, I2C_ROLE_MASTER, I2C_ADDRESSING_MODE_07BIT, ADDRESS);
    I2C_ConfigClkFrequency(APB_I2C0, I2C_CLOCKFREQUENCY_STANDARD);
    I2C_Enable(APB_I2C0, 1);
    I2C_IntEnable(APB_I2C0, (1 << I2C_INT_CMPL) | (1 << I2C_INT_FIFO_EMPTY));
}
```

- Master 中断实现

1. 中断中通过 I2C_STATUS_FIFO_EMPTY 来发送数据。每次填充 FIFO 直到 FIFO 为满, 当填充完最后一个数据后, 需要关掉 I2C_INT_FIFO_EMPTY, 否则中断会继续触发。
2. 当 I2C_STATUS_CMPL 触发时, 当前传输结束。

```
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));
    }
}
```



```

    }

    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;
}

```

- Master 触发传输

首先需要设置传输方向，I2C_TRANSACTION_MASTER2SLAVE，即代表 Master 发送数据，Slave 读取数据。

```

void peripheral_i2c_send_data(void)
{
    I2C_CtrlUpdateDirection(APB_I2C0,I2C_TRANSACTION_MASTER2SLAVE);
    I2C_CtrlUpdateDataCnt(APB_I2C0, DATA_CNT);
    I2C_CommandWrite(APB_I2C0, I2C_COMMAND_ISSUE_DATA_TRANSACTION);
}

```

- 使用流程

- 设置 IO, 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。

7.2.2.5.2.2 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;
```

- 初始化 I2C 模块

对于 Slave, 需要打开 I2C_INT_ADDR_HIT, 此中断的触发代表收到了匹配的地址。

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
    I2C_Config(APB_I2C0, I2C_ROLE_SLAVE, I2C_ADDRESSING_MODE_07BIT, ADDRESS);
    I2C_Enable(APB_I2C0, 1);
    I2C_IntEnable(APB_I2C0, (1<<I2C_INT_ADDR_HIT) | (1<<I2C_INT_CMPL));
}
```

- Slave 中断实现以及接收数据

1. 首先需要等待 I2C_STATUS_ADDRHIT 中断, 在该中断中通过 I2C_GetTransactionDir() 判断传输的方向。
 - I2C_TRANSACTION_MASTER2SLAVE: 代表 Slave 需要读取数据, 打开 I2C_INT_FIFO_FULL 中断。
 - I2C_TRANSACTION_SLAVE2MASTER: 代表 Slave 需要发送, 打开 I2C_INT_FIFO_EMPTY。
2. I2C_STATUS_FIFO_FULL 的触发, 代表 FIFO 中有接收到的数据, 读取数据, 直到 FIFO 变空。

3. I2C_STATUS_CMPL 代表传输结束，检查 FIFO 中有没有剩余数据，并且关掉 FIFO 中断避免再次触发。

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

    if(status & (1 << I2C_STATUS_ADDRHIT))
    {
        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));
    }

    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);
        }
    }

    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;
}

```

- 使用流程
 - 设置 IO, 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。

7.2.2.5.3 场景 3: Master 只读, Slave 只写, 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 IO, 请参考 IO 配置。

7.2.2.5.3.1 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;
```

- 初始化 I2C 模块

此处配置为 Master, 7bit 地址, 并打开了传输结束中断, 由于使用 DMA 传输, 因此不需要打开 FIFO 相关中断。其余配置和场景 1 相同。

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
    I2C_Config(APB_I2C0, I2C_ROLE_MASTER, I2C_ADDRESSING_MODE_07BIT, ADDRESS);
    I2C_ConfigClkFrequency(APB_I2C0, I2C_CLOCKFREQUENCY_STANDARD);
    I2C_Enable(APB_I2C0, 1);
    I2C_IntEnable(APB_I2C0, (1 << I2C_INT_CMPL));
}
```

- 初始化 DMA 模块

使用前需要配置 DMA 模块。

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

- Master 中断实现

等待 I2C_STATUS_CMPL 中断的触发, 该中断代表传输结束。

```
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;
}
```

- Master DMA 设置

该 API 实现了 I2C0 RXFIFO 到 DMA 的配置，细节请参考 DMA 文档。

```
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);
}
```

- Master 触发传输

1. 打开 I2C 模块的 DMA 功能。
2. 设置传输方向 I2C_TRANSACTION_SLAVE2MASTER，与场景 1 相同。
3. 设置需要传输的数据大小。
4. 配置 DMA，并使用 I2C_COMMAND_ISSUE_DATA_TRANSACTION 触发 I2C 传输。

```
void peripheral_i2c_send_data(void)
{
    I2C_DmaEnable(APB_I2C0,1);
    I2C_CtrlUpdateDirection(APB_I2C0,I2C_TRANSACTION_SLAVE2MASTER);
    I2C_CtrlUpdateDataCnt(APB_I2C0, DATA_CNT);

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

    I2C_CommandWrite(APB_I2C0, I2C_COMMAND_ISSUE_DATA_TRANSACTION);
}
```

- 使用流程

- 设置 IO, 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。

7.2.2.5.3.2 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;
```

- 初始化 I2C 模块

对于 Slave, 需要打开 I2C_INT_ADDR_HIT, 此中断的触发代表收到了匹配的地址。

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
    I2C_Config(APB_I2C0,I2C_ROLE_SLAVE,I2C_ADDRESSING_MODE_07BIT,ADDRESS);
    I2C_Enable(APB_I2C0,1);
    I2C_IntEnable(APB_I2C0,(1<<I2C_INT_ADDR_HIT)|(1<<I2C_INT_CMPL));
}
```

- 初始化 DMA 模块

使用前需要配置 DMA 模块。

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

- Slave 中断实现以及发送数据

1. 首先需要等待 I2C_STATUS_ADDRHIT 中断,在该中断中通过 I2C_GetTransactionDir() 判断传输的方向。
 - I2C_TRANSACTION_SLAVE2MASTER: 代表 Slave 需要发送,设置 DMA 发送数据。
2. I2C_STATUS_CMPL 代表传输结束,关闭 I2C DMA 功能。

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



```

if(status & (1 << I2C_STATUS_ADDRHIT))
{
    dir = I2C_GetTransactionDir(APB_I2C0);
    if(dir == I2C_TRANSACTION_SLAVE2MASTER)
    {
        peripherals_i2c_write_data_dma_setup();
    }
    I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_ADDRHIT));
}

if(status & (1 << I2C_STATUS_CMPL))
{
    I2C_DmaEnable(APB_I2C0,0);
    I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_CMPL));
}

return 0;
}

```

- Slave 发送数据以及 DMA 设置

该 API 实现了 DMA 传输数据到 I2C0 FIFO 的配置，细节请参考 DMA 文档。

```

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);
}

```

设置 DMA 并打开 I2C DMA 功能。

```
void peripherals_i2c_write_data_dma_setup(void)
{
    #define I2C_DMA_TX_CHANNEL    (0)//DMA channel 0
    peripherals_i2c_dma_to_txfifo(I2C_DMA_TX_CHANNEL, write_data,
                                  sizeof(write_data));

    I2C_CtrlUpdateDataCnt(APB_I2C0, DATA_CNT);
    I2C_DmaEnable(APB_I2C0,1);
}
```

- 使用流程

- 设置 IO, 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。

7.2.2.5.4 场景 4: Master 只写, Slave 只读, 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 IO, 请参考 IO 配置。

7.2.2.5.4.1 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;
```

- 初始化 I2C 模块

请参考场景 3 中 Master 配置的初始化 I2C 模块。

- 初始化 DMA 模块

请参考场景 3 中 Master 配置的初始化 DMA 模块。

- I2C 中断实现

请参考场景 3 中 Master 配置的 Master 中断实现。

- I2C Master DMA 设置

该 API 实现了 DMA 传输数据到 I2C0 FIFO 的配置，细节请参考 DMA 文档。

```
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);
}
```

- I2C Master 触发传输

1. 打开 I2C DMA 功能。
2. 设置传输方向为 I2C_TRANSACTION_MASTER2SLAVE，和场景 2 相同。
3. 配置 DMA，并使用 I2C_COMMAND_ISSUE_DATA_TRANSACTION 触发 I2C 传输。

```
void peripheral_i2c_send_data(void)
{
    I2C_DmaEnable(APB_I2C0, 1);
    I2C_CtrlUpdateDirection(APB_I2C0, I2C_TRANSACTION_MASTER2SLAVE);
    I2C_CtrlUpdateDataCnt(APB_I2C0, DATA_CNT);

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

```
I2C_CommandWrite(APB_I2C0, I2C_COMMAND_ISSUE_DATA_TRANSACTION);  
  
}
```

- 使用流程

- 设置 IO, 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。

7.2.2.5.4.2 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;
```

- 初始化 I2C 模块

请参考场景 3 中 Slave 配置的初始化 I2C 模块。

- 初始化 DMA 模块

请参考场景 3 中 Slave 配置的初始化 DMA 模块。

- Slave 中断实现以及接收数据

1. 首先需要等待 I2C_STATUS_ADDRHIT 中断, 在该中断中通过 I2C_GetTransactionDir() 判断传输的方向。
 - I2C_TRANSACTION_MASTER2SLAVE: 代表 Slave 需要接收, 设置 DMA 接收数据。
2. I2C_STATUS_CMPL 代表传输结束, 关闭 I2C DMA 功能。

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

    if(status & (1 << I2C_STATUS_ADDRHIT))
    {
        dir = I2C_GetTransactionDir(APB_I2C0);
        if(dir == I2C_TRANSACTION_MASTER2SLAVE)
        {
            peripherals_i2c_read_data_dma_setup();
        }

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

    if(status & (1 << I2C_STATUS_CMPL))
    {
        I2C_DmaEnable(APB_I2C0, 0);
        I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_CMPL));
    }

    return 0;
}
```

- Slave 发送数据以及 DMA 设置

该 API 实现了 I2C0 RXFIFO 到 DMA 的配置，细节请参考 DMA 文档。

```
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);
}

```

设置 DMA 并打开 I2C DMA 功能。

```

void peripherals_i2c_read_data_dma_setup(void)
{
    #define I2C_DMA_RX_CHANNEL    (0)//DMA channel 0
    peripherals_i2c_rxfifo_to_dma(I2C_DMA_RX_CHANNEL, read_data,
                                  sizeof(read_data));

    I2C_CtrlUpdateDataCnt(APB_I2C0, DATA_CNT);
    I2C_DmaEnable(APB_I2C0,1);
}

```

- 使用流程

- 设置 IO, 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。

7.2.2.5.5 场景 5: Master/Slave 同时读写 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 IO, 请参考 IO 配置。

7.2.2.5.5.1 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;
```

- 初始化 I2C 模块

1. 配置为 Master, 7bit 地址。
2. 打开传输结束中断和 ADDR_HIT 中断。对于 Master 来说, 如果有 Slave 响应了该地址, 则会有 I2C_INT_ADDR_HIT 中断。

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
    I2C_Config(APB_I2C0, I2C_ROLE_MASTER, I2C_ADDRESSING_MODE_07BIT, ADDRESS);
    I2C_ConfigClkFrequency(APB_I2C0, I2C_CLOCKFREQUENCY_STANDARD);
    I2C_Enable(APB_I2C0, 1);
    I2C_IntEnable(APB_I2C0, (1<<I2C_INT_CMPL) | (1<<I2C_INT_ADDR_HIT));
}
```

- Master 中断实现

1. I2C_STATUS_ADDRHIT, 代表 Slave 响应了 Master 发送的地址:
 - 如果 Master 执行写操作, 需要打开 I2C_INT_FIFO_EMPTY, 用来发送数据。
 - 如果 Master 执行读操作, 需要打开 I2C_INT_FIFO_FULL, 用来接收数据。
2. 如果 Master 执行写操作, I2C_STATUS_FIFO_EMPTY 中断会出现, 此时需要填充待发送的数据。如果数据发送完成, 需要关闭 I2C_STATUS_FIFO_EMPTY 中断, 避免重复触发。
3. 如果 Master 执行读操作, I2C_STATUS_FIFO_FULL 中断会出现, 此时需要读取数据。
4. I2C_STATUS_CMPL, 代表传输结束:

- 如果 Master 执行写操作，代表写成功。
- 如果 Master 执行读操作，需要读取 FIFO 中的剩余数据。

```
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))
    {
        if(master_write_flag)
        {
            I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_CMPL));
        }
        else
        {
            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
            read_data_cnt = 0;
        }
    }

    if(status & (1 << I2C_STATUS_ADDRHIT))
    {
        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));
}

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++)
        {
            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));
        }
    }
}

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);
        }
    }
}

return 0;
}

```

- Master 写传输

首先需要设置传输方向，I2C_TRANSACTION_MASTER2SLAVE，即代表 Master 发送数据，Slave 读取数据。

```

void peripheral_i2c_write_data(void)
{
    master_write_flag = 1;

    I2C_CtrlUpdateDirection(APB_I2C0, I2C_TRANSACTION_MASTER2SLAVE);
    I2C_CtrlUpdateDataCnt(APB_I2C0, DATA_CNT);
    I2C_CommandWrite(APB_I2C0, I2C_COMMAND_ISSUE_DATA_TRANSACTION);
}

```

- Master 读传输

首先需要设置传输方向，此处是 I2C_TRANSACTION_SLAVE2MASTER，即代表 Slave 发送数据，Master 读取数据。

```

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);
}
```

- 使用流程

- 设置 IO, 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()。
- 当写结束后, 可以触发 I2C 读取, peripheral_i2c_read_data()。
- 检查中断状态。

7.2.2.5.5.2 Slave 配置

- 初始化 I2C 模块

1. 配置为 Slave, 7bit 地址。
2. 打开传输结束中断和 ADDR_HIT 中断。对于 Slave 来说, 如果有匹配的地址, 则会有 I2C_INT_ADDR_HIT 中断。

```
#define ADDRESS (0x71)
void setup_peripherals_i2c_module(void)
{
    I2C_Config(APB_I2C0,I2C_ROLE_SLAVE,I2C_ADDRESSING_MODE_07BIT,ADDRESS);
    I2C_Enable(APB_I2C0,1);
    I2C_IntEnable(APB_I2C0,(1<<I2C_INT_ADDR_HIT)|(1<<I2C_INT_CMPL));
}
```

- Slave 中断实现

1. I2C_STATUS_ADDRHIT, 在该中断中通过 I2C_GetTransactionDir() 判断传输的方向:
 - I2C_TRANSACTION_MASTER2SLAVE: 代表 Slave 需要读取数据, 打开 I2C_INT_FIFO_FULL 中断。

- I2C_TRANSACTION_SLAVE2MASTER: 代表 Slave 需要发送, 打开 I2C_INT_FIFO_EMPTY。
- 2. 如果 Slave 需要发送, I2C_STATUS_FIFO_EMPTY 中断会出现, 此时需要填充待发送的数据。
- 3. 如果 Slave 需要读取数据, I2C_STATUS_FIFO_FULL 中断会出现, 此时需要读取数据。
- 4. I2C_STATUS_CMPL, 代表传输结束:
 - 如果 Slave 执行写操作, 代表写成功, 关闭 I2C_STATUS_FIFO_EMPTY 中断。
 - 如果 Slave 执行读操作, 需要读取 FIFO 中的剩余数据, 关闭 I2C_STATUS_FIFO_FULL 中断。

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

    if(status & (1 << I2C_STATUS_ADDRHIT))
    {
        dir = I2C_GetTransactionDir(APB_I2C0);
        if(dir == I2C_TRANSACTION_MASTER2SLAVE)
        {
            master_write_flag = 1;
            I2C_IntEnable(APB_I2C0, (1 << I2C_INT_FIFO_FULL));
        }
        else if(dir == I2C_TRANSACTION_SLAVE2MASTER)
        {
            master_write_flag = 0;
            I2C_IntEnable(APB_I2C0, (1 << I2C_INT_FIFO_EMPTY));
            write_data_cnt = 0;
        }

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

```
}

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]);
        }
    }
}

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);
        }
    }
}

if(status & (1 << I2C_STATUS_CMPL))
{
    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));
read_data_cnt = 0;
}
else
{
    I2C_ClearIntState(APB_I2C0, (1 << I2C_STATUS_CMPL));
    write_data_cnt = 0;
}

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;
}
```

- 使用流程

- 设置 IO, 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。

7.2.3 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_ClockFrequencyOptions
 */
void I2C_ConfigClkFrequency(I2C_TypeDef *I2C_BASE, I2C_ClockFrequencyOptions option);
```

其中 option 中定义了几个可选项（I2C 时钟和系统时钟有关系，以下枚举可能需要根据实际时钟调整）：

```
typedef enum
{
    I2C_CLOCKFREQUENCY_NULL,
    I2C_CLOCKFREQUENCY_STANDARD, //up to 100kbit/s
    I2C_CLOCKFREQUENCY_FASTMODE, //up to 400kbit/s
    I2C_CLOCKFREQUENCY_FASTMODE_PLUS, //up to 1Mbit/s
    I2C_CLOCKFREQUENCY_MANUAL
} I2C_ClockFrequencyOptions;
```

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：乘数因子，位宽 5bit，所有 I2C_BASE->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，参数如下：
 - scl_hi: 高电平持续时间，位宽 9bit，默认 0x10。
 - scl_ratio: 低电平持续时间因子，位宽 1bit，默认 1。
 - hddat: SCL 拉低后 SDA 的保持时间，位宽 5bit，默认 5。
 - sp: 可以被过滤的脉冲毛刺宽度，位宽 3bit，默认 1。
 - sudat: 释放 SCL 之前的数据建立时间，位宽 5bit，默认 5。

每个参数和时钟的计算关系如下：

- 高电平持续时间计算：

$$SCL_{highperiod} = (2 \times pclk) + (2 + sp + sclhi) \times pclk \times (TPM + 1)$$

其中 $pclk$ 为 I2C 模块的系统时钟，默认为 24M，实际时钟可以从 `SYSCTRL_GetClk()` 获取。

如果 $sp = 1, pclk = 42ns, TPM = 3, sclhi = 150$ ，则：

$$SCL_{highperiod} = (2 \times 42) + (2 + 1 + 150) \times 42 \times (3 + 1) = 25788ns$$

- 低电平持续时间计算：

$$SCL_{lowperiod} = (2 \times pclk) + (2 + sp + sclhi \times (sclratio + 1)) \times pclk \times (TPM + 1)$$

如果 $sp = 1, pclk = 42ns, TPM = 3, sclhi = 150, sclratio = 0$ ，则：

$$SCL_{lowperiod} = (2 \times 42) + (2 + 1 + 150 \times 1) \times 42 \times (3 + 1) = 25788ns$$

- 毛刺抑制宽度： $spikesuppressionwidth = sp \times pclk \times (TPM + 1)$

如果 $sp = 1, pclk = 42ns, TPM = 3$ ，则：

$$spikesuppressionwidth = 1 \times 42 \times (3 + 1) = 168ns$$

- SCL 之前的数据建立时间：

$$setuptime = (2 \times pclk) + (2 + sp + sudat) \times pclk \times (TPM + 1)$$

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

$$setuptime = (2 \times 42) + (2 + 1 + 5) \times 42 \times (3 + 1) = 1428ns$$

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：

$$holdtime = (2 \times pclk) + (2 + sp + hddat) \times pclk \times (TPM + 1)$$

如果 $sp = 1, pclk = 42ns, TPM = 3, hddat = 5$ ，则：

$$holdtime = (2 \times 42) + (2 + 1 + 5) \times 42 \times (3 + 1) = 1428ns$$

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 I2S 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，另外一种为语音模式。

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：

- $f_{\text{bclk}} = \text{clk} / (2 * b_div)$
- $f_{\text{lrclk}} = f_{\text{bclk}} / (2 * lr_div)$

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 $lr_div \geq 24$ 。

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 $f_bclk = 1.024\text{MHz}$ 。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 $clk = 256\text{MHz}$ 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 控制器操作流程如下：

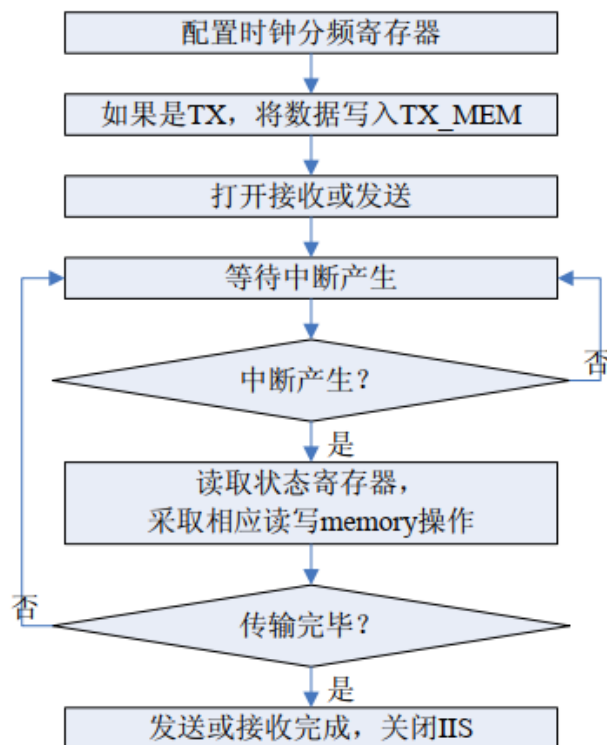


图 8.1: I2S 控制器操作流程

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); // source 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); // source 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 红外 (IR)

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 IO 配置

红外的发射和接收分别需要一个 IO，并非所有 IO 都可以映射成红外，请查看对应的 datasheet 获取可用的 IO。

1. 使用 `SYSCTRL_ClearClkGateMulti` 打开对应时钟，红外时钟为 `SYSCTRL_ClkGate_APB_IR`。
2. 红外输入（接收）需要使用 `PINCTRL_SelIrIn` 来配置 IO。
3. 红外输出（发射）需要使用 `PINCTRL_SetPadMux` 来配置 IO，使用方式请参考下述示例。
4. 使用 `platform_set_irq_callback` 配置红外中断。

以下示例可以将指定 IO 配置成红外，并打开红外时钟和中断：

```
#define IR_DOUT GPIO_GPIO_10
#define IR_DIN  GPIO_GPIO_11

void setup_peripherals_ir_module(void)
```

```
{  
    SYSCTRL_ClearClkGateMulti( (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 参数 (不同编码的时间参数)

红外模块支持 NEC, RC5, 不同编码方式需要提供对应序列中时间参数, 具体如下:

- timer1:
 - 发送模式下, 表示引导码低电平时间: 如 NEC 为 9ms。
 - 接收模式下与 timer2 组成检测引导码低电平窗口。
- timer2:
 - 发送模式下, 表示重复码低电平时间: 如 NEC 为 2.25ms。
 - 接收模式下与 timer1 组成检测引导码低窗口。
- timer3:
 - 发送模式下, 表示引导码高电平时间: 如 NEC 为 4.5ms。
 - 接收模式下与 timer4 组成检测引导码高电平 + 低电平窗口。
- timer4:
 - 发送模式下, 表示重复码高电平时间: 如 NEC 为 560us。
 - 接收模式下与 timer3 组成检测引导码高电平 + 低电平窗口。
- timer5:
 - 接收时接收超时定时器, 发射不必关注。
- btimer1:
 - 逻辑 0 的 bit 时长: 如 NEC 为 1.12ms。

- btimer2:
 - 逻辑 1 的 bit 时长：如 NEC 为 2.25ms。
- 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 bit 检测超时时间。
- 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 NEC 为 38KHz，30% 占空比载波则 $carry_{low} = (2/3)/times$ ， $carry_{high} = (1/3)/times$ 。
- 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。

结构体如下：

```
typedef struct
{
    uint16_t timer1;
    uint16_t timer2;
    uint16_t timer3;
    uint16_t timer4;
    uint16_t timer5;
    uint16_t btimer1;
    uint16_t btimer2;
    uint16_t bit_cycle;
    uint16_t carry_low;
    uint16_t carry_high;
}Ir_mode_param_t;
```

不同参数需要根据内部时钟来计算：

1. 载波参数根据 OSC_CLK_FREQ 得到：

NEC 的载波频率为 38KHz，RC5 为 36KHz，对应的载波参数为：

```
#define NEC_WAVE_FREQ 38000
#define RC5_WAVE_FREQ 36000

#define IR_WAVE_NEC_FREQ (OSC_CLK_FREQ/NEC_WAVE_FREQ)
#define IR_WAVE_TC9012_FREQ (OSC_CLK_FREQ/NEC_WAVE_FREQ)
#define IR_WAVE_RC5_FREQ (OSC_CLK_FREQ/RC5_WAVE_FREQ)
```

2. Bit 时长参数根据 32K 时钟计数得到:

以 NEC 为例，其逻辑”0”为 562.5 μ s 的有效脉冲加 562.5 μ s 的空闲间隔，总时长为 1.125ms。而逻辑”1”为 562.5 μ s 的有效脉冲加 1.6875ms 的空闲间隔，总时长为 2.25ms。因此 NEC 的逻辑 0 或者 1 的计数单位约为 560 μ s，以 32K 计数可以得到对应参数为:32000x560/1000000。

```
#define BASE_CLK 32000

#define NEC_UINT (BASE_CLK*560/1000000+1)
#define TC9012_UINT (BASE_CLK*560/1000000+1)
#define RC5_UINT (BASE_CLK*889/1000000+1)
```

根据以上参数可以得到不同编码下的所有时间参数:

```
typedef struct{
    Ir_mode_param_t param_tx;
    Ir_mode_param_t param_rx;
}Ir_type_param_t;

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, 0,
            2*NEC_UINT-1, 4*NEC_UINT-1, 1*NEC_UINT-1,
```

```

        IR_WAVE_NEC_FREQ*2/3,IR_WAVE_NEC_FREQ*1/3},
    { //RX
        14*NEC_UINT-1,18*NEC_UINT-1,22*NEC_UINT-1,26*NEC_UINT-1,0xff,
        0,2*NEC_UINT-1,0x7f,0,0},
    },
    { //TC9012 param
        { //TX
            8*TC9012_UINT-1,8*TC9012_UINT-1,8*TC9012_UINT-1,2*TC9012_UINT-1,0xff,
            2*TC9012_UINT-1,4*TC9012_UINT-1,1*TC9012_UINT-1,
            IR_WAVE_TC9012_FREQ*2/3,IR_WAVE_TC9012_FREQ*1/3},
        { //RX
            7*TC9012_UINT-1,9*TC9012_UINT-1,15*TC9012_UINT-1,17*TC9012_UINT-1,0xff,
            INESSENTIAL,2*TC9012_UINT-1,0x7f,INESSENTIAL,INESSENTIAL},
        },
    { //RC5 param
        { //TX
            2*RC5_UINT-1,2*RC5_UINT-1,0,0,0,2*RC5_UINT-1,
            0,1*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-1,0,1*RC5_UINT-3,
            1*RC5_UINT-1,2*RC5_UINT-1,0,0},
        }
    };

```

9.2.3 红外发射接收

9.2.3.1 接收初始化

接收功能的初始化需要配置：

- 编码方式 `mode`。
- 配置为接收模式 `IR_TXRX_MODE_RX_MODE`。
- 配置接收模式对应的时间参数 `Ir_mode_param_t`。

- 通过 `IR_CtrlEnable` 使能红外模块。

```
void setup_peripherals_ir_module_rx(IR_IrMode_e mode,
                                     const Ir_mode_param_t* param_p)
{
    IR_CtrlSetIrMode(APB_IR, mode == IR_IR_MODE_IR_9012 ? 0 : mode);
    IR_CtrlSetTxRxMode(APB_IR, IR_TXRX_MODE_RX_MODE);
    IR_CtrlSetIrIntEn(APB_IR);

    IR_CtrlSetIrEndDetectEn(APB_IR);
    IR_CtrlIrUserCodeVerify(APB_IR);
    IR_CtrlIrDatacodeVerify(APB_IR);
    IR_TimeSetIrTime1(APB_IR, param_p->timer1);
    IR_TimeSetIrTime2(APB_IR, param_p->timer2);
    IR_TimeSetIrTime3(APB_IR, param_p->timer3);
    IR_TimeSetIrTime4(APB_IR, param_p->timer4);
    IR_TimeSetIrTime5(APB_IR, param_p->timer5);
    IR_CtrlIrSetBitTime1(APB_IR, param_p->btimer1);
    IR_CtrlIrSetBitTime2(APB_IR, param_p->btimer2);
    IR_CtrlIrSetIrBitCycle(APB_IR, param_p->bit_cycle);

    IR_CtrlEnable(APB_IR);
}
```

9.2.3.2 发射初始化

发射功能的初始化需要配置：

- 编码方式 `mode`。
- 配置为发射模式 `IR_TXRX_MODE_TX_MODE`。
- 配置发射模式对应的时间参数 `Ir_mode_param_t`。
- 通过 `IR_CtrlEnable` 使能红外模块。


```
void setup_peripherals_ir_module_tx(IR_IrMode_e mode,
                                   const Ir_mode_param_t* param_p)
{
    IR_CtrlSetIrMode(APB_IR, mode == IR_IR_MODE_IR_9012 ? 0 : mode);
    IR_CtrlSetTxRxMode(APB_IR, IR_TXRX_MODE_TX_MODE);
    IR_CtrlSetIrIntEn(APB_IR);

    IR_TxConfigIrTxPol(APB_IR);
    IR_TxConfigCarrierCntClr(APB_IR);
    IR_TxConfigIrIntEn(APB_IR);
    IR_CarryConfigSetIrCarryLow(APB_IR, param_p->carry_low);
    IR_CarryConfigSetIrCarryHigh(APB_IR, param_p->carry_high);
    IR_TimeSetIrTime1(APB_IR, param_p->timer1);
    IR_TimeSetIrTime2(APB_IR, param_p->timer2);
    IR_TimeSetIrTime3(APB_IR, param_p->timer3 );
    IR_TimeSetIrTime4(APB_IR, param_p->timer4);
    IR_CtrlIrSetBitTime1(APB_IR, param_p->btimer1);
    IR_CtrlIrSetBitTime2(APB_IR, param_p->btimer2);
    IR_CtrlIrSetIrBitCycle(APB_IR, param_p->bit_cycle);

    IR_CtrlEnable(APB_IR);
}
```

9.2.3.3 发射数据

1. 使用 IR_TxCodeSetIrTxUsercode 填充 16bit 地址。
2. 使用 IR_TxCodeSetIrTxDatacode 填充 16bit 数据。
3. 关闭重复码功能。
4. 发送数据 IR_TxConfigTxStart。
5. 如果需要，可以通过 IR_FsmGetIrTransmitOk 等待直到发送完成，或者通过中断等待。

```
void ir_transmit_fun(uint16_t addr, uint16_t data)
{
```

```

IR_TxCodeSetIrTxUsercode(APB_IR,addr);
IR_TxCodeSetIrTxDatacode(APB_IR,data);
IR_CleanIrTxRepeatMode(APB_IR);
IR_TxConfigTxStart(APB_IR);
}

```

如果需要发送重复码，则需要打开重复码功能 IR_CtrlIrTxRepeatMode。

```

void ir_transmit_repeat(void)
{
    IR_CtrlIrTxRepeatMode(APB_IR);
    IR_TxConfigTxStart(APB_IR);
}

```

9.2.3.4 中断实现 (以及红外接收)

中断触发后，通过状态寄存器判断对应的操作：

- IR_FsmGetIrTransmitOk 代表发送成功。
- IR_FsmGetIrTxRepeat 代表重复码发送成功。
- IR_FsmGetIrReceivedOk 代表接收到了数据，通过以下 API 获取数据：
 - IR_RxCodeGetIrRxUsercode 获取地址。
 - IR_RxCodeGetIrRxDatacode 获取数据。
- 通过 IR_FsmClearIrInt 清除中断避免重复触发。

以下示例演示了从中断中接收数据：

```

static uint32_t IRQHandler_IR_INT(void *user_data)
{
    if(IR_FsmGetIrReceivedOk(APB_IR))
    {
        uint32_t value = IR_RxCodeGetIrRxUsercode(APB_IR)<<16 |

```

```
        IR_RxCodeGetIrRxDatacode(APB_IR);  
    }  
  
    IR_FsmClearIrInt(APB_IR);  
    return 0;  
}
```


第十章 硬件键盘扫描控制器（KEYSCAN）

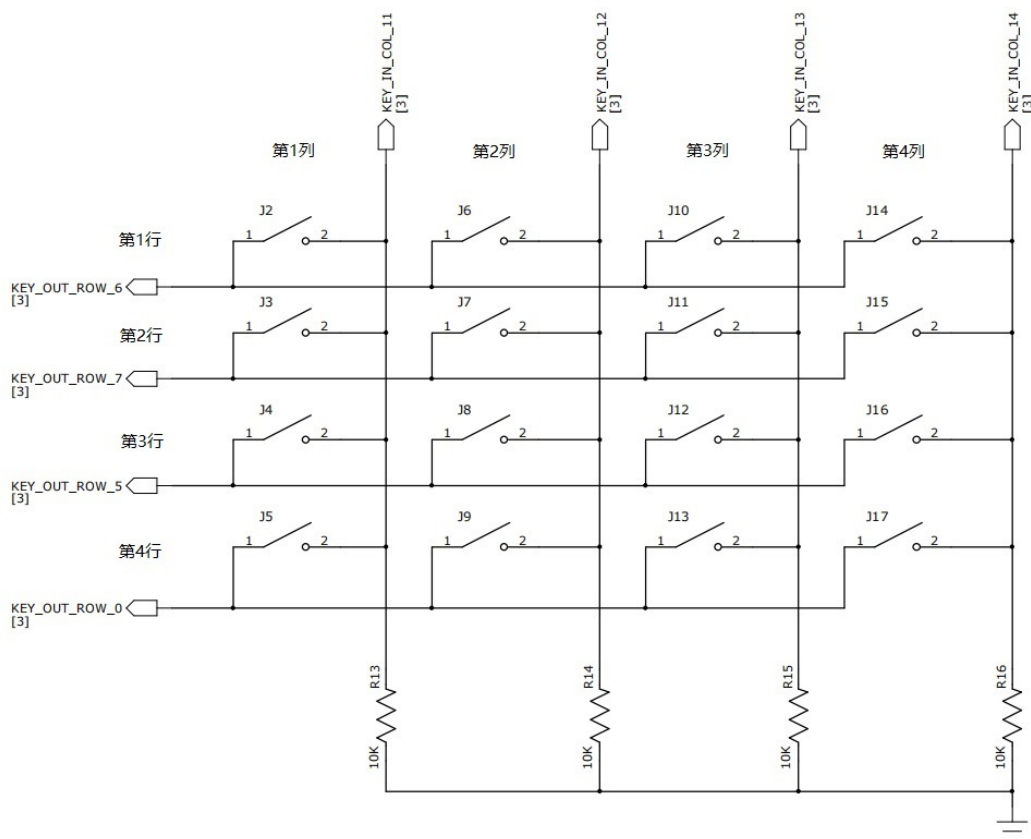
10.1 功能概述

特性：

- 可配置矩阵，最大支持 18 行 *18 列的键盘矩阵。
- 每个单独的行或者列可以设置启用或者禁用。
- 可配置时钟。
- 支持输入硬件去抖动，去抖时间可配置。
- 支持配置扫描间隔和释放时间，支持多按键同时按下。
- 支持中断和 DMA。

10.2 使用说明

以 4 行 *4 列的键盘矩阵为例：



10.2.1 键盘矩阵的软件描述

```
typedef struct {
    KEYSKAN_InColIndex_t in_col;
    GPIO_Index_t gpio;
} KEYSKAN_InColList;

typedef struct {
    KEYSKAN_OutRowIndex_t out_row;
    GPIO_Index_t gpio;
} KEYSKAN_OutRowList;
```

```
KEYSCAN_OutRowList key_out_row[] = {
    {KEY_OUT_ROW_6, GIO_GPIO_32}, // 第 1 行
    {KEY_OUT_ROW_7, GIO_GPIO_33}, // 第 2 行
    {KEY_OUT_ROW_5, GIO_GPIO_31}, // 第 3 行
    {KEY_OUT_ROW_0, GIO_GPIO_23}, // 第 4 行
};

#define key_out_row_num (sizeof(key_out_row) / sizeof(key_out_row[0]))

KEYSCAN_InColList key_in_col[] = {
    {KEY_IN_COL_11, GIO_GPIO_11}, // 第 1 列
    {KEY_IN_COL_12, GIO_GPIO_12}, // 第 2 列
    {KEY_IN_COL_13, GIO_GPIO_13}, // 第 3 列
    {KEY_IN_COL_14, GIO_GPIO_14}, // 第 4 列
};

#define key_in_col_num (sizeof(key_in_col) / sizeof(key_in_col[0]))
```

第 1 行按键接到了 GPIO32, 映射到 KEYSCAN 模块的 ROW6。第 1 列按键接到了 GPIO11, 映射到 KEYSCAN 模块的 COL11。以此类推 4 行 4 列的键盘阵列。

注意: KEYSCAN 的 ROW 和 COL 不是随意映射到 GPIO, 映射关系参考管脚管理 (PINC-TRL) 说明文档。

10.2.2 KEYSCAN 模块初始化

```
typedef struct {
    KEYSCAN_InColList *col;
    int col_num;

    KEYSCAN_OutRowList *row;
    int row_num;

    uint8_t fifo_num_trig_int;
    uint8_t dma_num_trig_int;
```

```

uint8_t dma_en;
uint8_t int_trig_en;
uint16_t release_time;
uint16_t scan_interval;
uint8_t debounce_counter;
} KEYSCAN_SetStateStruct;

```

```

/**
 * @brief Initialize keyscan module
 *
 * @param[in] keyscan_set      Initial parameter struct
 * @return          0 if success else non-0
 */
int KEYSCAN_Initialize(const KEYSCAN_SetStateStruct* keyscan_set);

/**
 * @brief Initialize mapping table of keyboard array row and col
 *
 * @param[in]  keyscan_set      Initial parameter struct
 * @param[out] ctx              keyboard array mapping table
 */
void KEYSCAN_InitKeyScanToIdx(const KEYSCAN_SetStateStruct* keyscan_set,
                              KEYSCAN_Ctx *ctx);

```

10.2.3 获取扫描到的按键

KEYSCAN 模块使能扫描后会按照行和列的配置开始扫描。模块有 FIFO 缓存扫描数据。每次扫描循环结束，FIFO 中压入 1 个 0x400 标志完成一次扫描。

可以配置 FIFO 中数据个数触发中断或者 DMA 触发中断：


```
void KEYSCAN_SetFifoNumTrigInt(uint32_t trig_num);
void KEYSCAN_SetDmaNumTrigInt(uint32_t trig_num);
```

获取 FIFO 是否为空的状态和数据:

```
/**
 * @brief Check keyscan FIFO empty or not
 *
 * @return 0: FIFO have data; 1: empty
 */
uint8_t KEYSCAN_GetIntStateFifoEmptyRaw(void);

/**
 * @brief GET keyscan FIFO data
 *
 * @return 0~4 bits: col; 5~9 bits: row; 10 bit: scan cycle end flag
 */
uint16_t KEYSCAN_GetKeyData(void);
```

按键 FIFO 原始数据的 0~4 位是按下按键所在的 KEYSCAN 模块中的 col, 5~9 位是 row, 注意这个值并不是键盘矩阵中的行和列, 可以用下面接口将原始数据解析为键盘矩阵中的行和列:

```
/**
 * @brief Transfer keyscan FIFO raw data to keyboard array row and col
 *
 * To use this helper function, `ctx` must be initialized with `KEYSCAN_InitKeyScanToIdx`.
 *
 * @param[in] ctx keyboard array mapping table
 * @param[in] key_data keyscan FIFO raw data
 * @param[out] row pressed key's 0-based row index in keyboard array
 * @param[out] col pressed key's 0-based col index in keyboard array
 * @return 0: scan cycle end data;
 * 1: find key pressed, *row and *col are key positions in keyboard array
```

```

*/
uint8_t KEYSKAN_KeyDataToRowColIdx(const KEYSKAN_Ctx *ctx, uint32_t key_data,
                                   uint8_t *row, uint8_t *col);

```

10.3 应用举例

10.3.1 初始化 KEYSKAN 模块

```

KEYSCAN_OutRowList key_out_row[] = {
    {KEY_OUT_ROW_6, GPIO_GPIO_32}, // 第 1 行
    {KEY_OUT_ROW_7, GPIO_GPIO_33}, // 第 2 行
    {KEY_OUT_ROW_5, GPIO_GPIO_31}, // 第 3 行
    {KEY_OUT_ROW_0, GPIO_GPIO_23}, // 第 4 行
};

#define key_out_row_num (sizeof(key_out_row) / sizeof(key_out_row[0]))

KEYSCAN_InColList key_in_col[] = {
    {KEY_IN_COL_11, GPIO_GPIO_11}, // 第 1 列
    {KEY_IN_COL_12, GPIO_GPIO_12}, // 第 2 列
    {KEY_IN_COL_13, GPIO_GPIO_13}, // 第 3 列
    {KEY_IN_COL_14, GPIO_GPIO_14}, // 第 4 列
};

#define key_in_col_num (sizeof(key_in_col) / sizeof(key_in_col[0]))

static KEYSKAN_Ctx key_ctx = {0};

static KEYSKAN_SetStateStruct keyscan_set = {
    .col                = key_in_col,
    .col_num            = key_in_col_num,
    .row                = key_out_row,
    .row_num            = key_out_row_num,
    .fifo_num_trig_int = 1,
};

```

```

        .release_time      = 110,
        .scan_interval    = 0xFFFF,
        .debounce_counter  = 50,
        .dma_num_trig_int  = 0x10,
        .dma_en            = 0,
        .int_trig_en       = 1,
    };

static uint32_t keyscan_cb_isr(void *user_data);
static void setup_peripherals_keyscan(void)
{
    SYSCTRL_ClearClkGateMulti(1 << SYSCTRL_ITEM_APB_KeyScan);
    SYSCTRL_ClearClkGateMulti(1 << SYSCTRL_ITEM_APB_PinCtrl);

    KEYSKAN_Initialize(&keyscan_set);
    KEYSKAN_InitKeyScanToIdx(&keyscan_set, &key_ctx);

    platform_set_irq_callback(PLATFORM_CB_IRQ_KEYSCAN, keyscan_cb_isr, 0);

    return;
}

```

10.3.2 中断数据处理

```

static uint8_t key_state_buf[2][key_out_row_num][key_in_col_num] = {0};
static uint8_t key_state_last_index = 0;
static uint8_t key_state_now_index = 1;
static void printf_key_state(void)
{
    int row, col;
    for (row = 0; row < key_out_row_num; row++) {
        for (col = 0; col < key_in_col_num; col++) {

```

```

        if (key_state_buf[key_state_now_index][row][col] !=
            key_state_buf[key_state_last_index][row][col]) {
            printf("row%u col%u %s\r\n",
                row + 1, col + 1,
                key_state_buf[key_state_now_index][row][col] == 0 ?
                    "release" : "press");
        }
    }
}

if (key_state_now_index == 0) {
    key_state_now_index = 1;
    key_state_last_index = 0;
} else {
    key_state_now_index = 0;
    key_state_last_index = 1;
}

for (row = 0; row < key_out_row_num; row++) {
    for (col = 0; col < key_in_col_num; col++) {
        key_state_buf[key_state_now_index][row][col] = 0;
    }
}

return;
}

static void key_state_clear(void)
{
    int row, col;
    for (row = 0; row < key_out_row_num; row++) {
        for (col = 0; col < key_in_col_num; col++) {
            key_state_buf[0][row][col] = 0;
            key_state_buf[1][row][col] = 0;
        }
    }
}

```

```

    }
}

static uint32_t keyscan_cb_isr(void *user_data)
{
    uint32_t key_data;
    uint8_t key_scan_row;
    uint8_t key_scan_col;
    uint8_t row = 0;
    uint8_t col = 0;
    static uint8_t have_key_pressed = 0;
    static uint8_t no_key_pressed_cnt = 0;

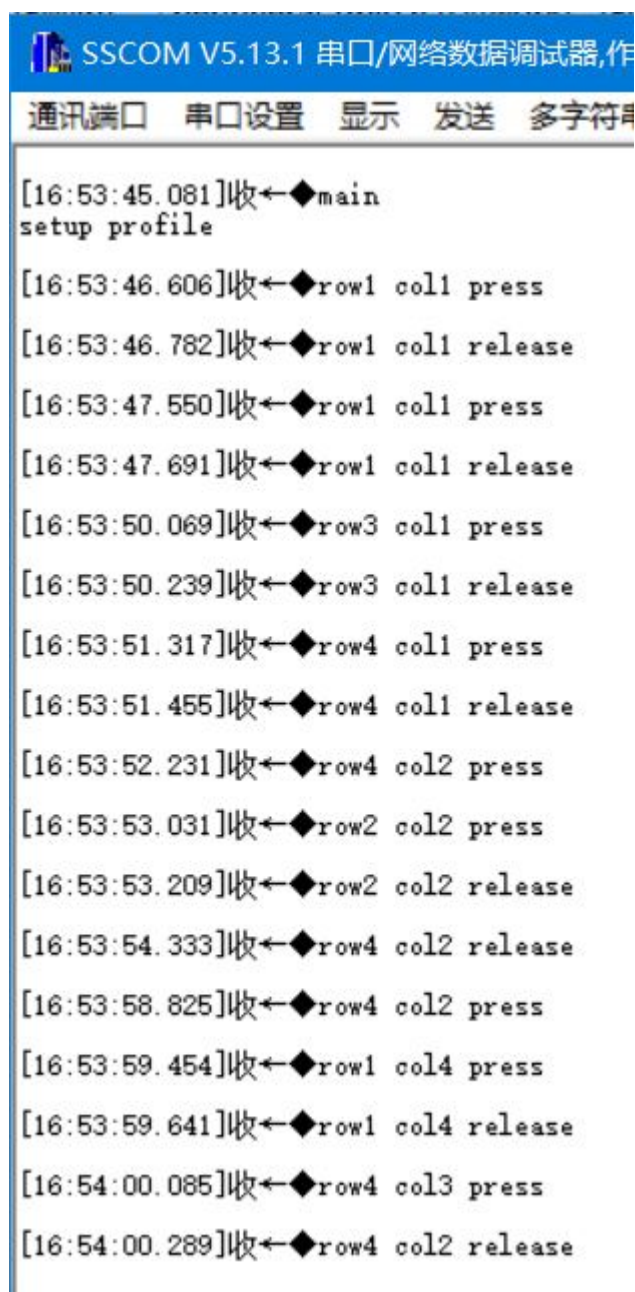
    while (KEYSCAN_GetIntStateFifoEmptyRaw() == 0) {
        key_data = KEYSCAN_GetKeyData();
        if (KEYSCAN_KeyDataToRowColIdx(&key_ctx, key_data, &row, &col)) {
            // 扫描到有按键按下 按键位置为 row col
            key_state_buf[key_state_now_index][row][col] = 1;
            have_key_pressed = 1;
        } else {
            // 完成一次扫描 根据 have_key_pressed 判断该轮扫描中是否有按键按下
            if (have_key_pressed == 1) {
                have_key_pressed = 0;
                no_key_pressed_cnt = 0;
            } else {
            }

            switch (no_key_pressed_cnt) {
                case 0: // 该轮扫描中有按键按下
                    no_key_pressed_cnt++;
                    printf_key_state();
                    break;
                case 1: // 该轮扫描中没有按键按下 上一轮扫描中有按键按下 说明按键释放
                    no_key_pressed_cnt++;
                    printf_key_state();
            }
        }
    }
}

```

```
        key_state_clear();  
        break;  
    case 2: // 连续两轮或以上都没有按键按下  
        break;  
    default:  
        break;  
    }  
}  
}  
}  
return 0;  
}
```

10.3.3 效果



The screenshot shows the SSCOM V5.13.1 interface with a menu bar (通讯端口, 串口设置, 显示, 发送, 多字符) and a log window. The log contains the following text:

```
[16:53:45.081]收←◆main
setup profile
[16:53:46.606]收←◆row1 col1 press
[16:53:46.782]收←◆row1 col1 release
[16:53:47.550]收←◆row1 col1 press
[16:53:47.691]收←◆row1 col1 release
[16:53:50.069]收←◆row3 col1 press
[16:53:50.239]收←◆row3 col1 release
[16:53:51.317]收←◆row4 col1 press
[16:53:51.455]收←◆row4 col1 release
[16:53:52.231]收←◆row4 col2 press
[16:53:53.031]收←◆row2 col2 press
[16:53:53.209]收←◆row2 col2 release
[16:53:54.333]收←◆row4 col2 release
[16:53:58.825]收←◆row4 col2 press
[16:53:59.454]收←◆row1 col4 press
[16:53:59.641]收←◆row1 col4 release
[16:54:00.085]收←◆row4 col3 press
[16:54:00.289]收←◆row4 col2 release
```


第十一章 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。

11.1 功能描述

11.1.1 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

11.1.2 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

11.2 使用方法

11.2.1 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。

11.2.2 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

11.3 编程指南

11.3.1 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

11.3.2 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。

11.3.2.1 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：

11.3.2.2 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(1);
    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);
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。

第十二章 管脚管理（PINCTRL）

12.1 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 12.1。表 12.2 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 GPIO_SetDirection 控制。DEBUG 模式为保留功能，具体功能暂不开放。

表 12.1: 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
PCAP	PCAP0_IN, PCAP1_IN, PCAP2_IN, PCAP3_IN, PCAP4_IN, PCAP5_IN

外设	功能管脚
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 12.2: 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
KEY_IN_COL_17	17

外设功能管脚	可连接到的 IO 管脚
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
TIMER0_PWM_0B	1, 3, 5, 7, 9, 11, 13, 15, 17, 22, 34

外设功能管脚	可连接到的 IO 管脚
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
SPI0_MISO_OUT	27

外设功能管脚	可连接到的 IO 管脚
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

12.2 使用说明

12.2.1 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 12.1 和表 12.2 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_SetUartIn` 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 12.3 罗列了为各外设提供的输入配置函数。

表 12.3: 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

12.2.2 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 12.4 列出了各管脚默认的上下拉配置。

表 12.4: 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

12.2.3 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。

12.2.4 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 12.2 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);
```

12.2.5 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 12.5 所示。

表 12.5: 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

- 其它还有模拟比较器，32k 晶体引脚等。

第十三章 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。

13.1 功能描述

13.1.1 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

13.1.2 PTE 原理图

13.1.3 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启用或禁用。

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。

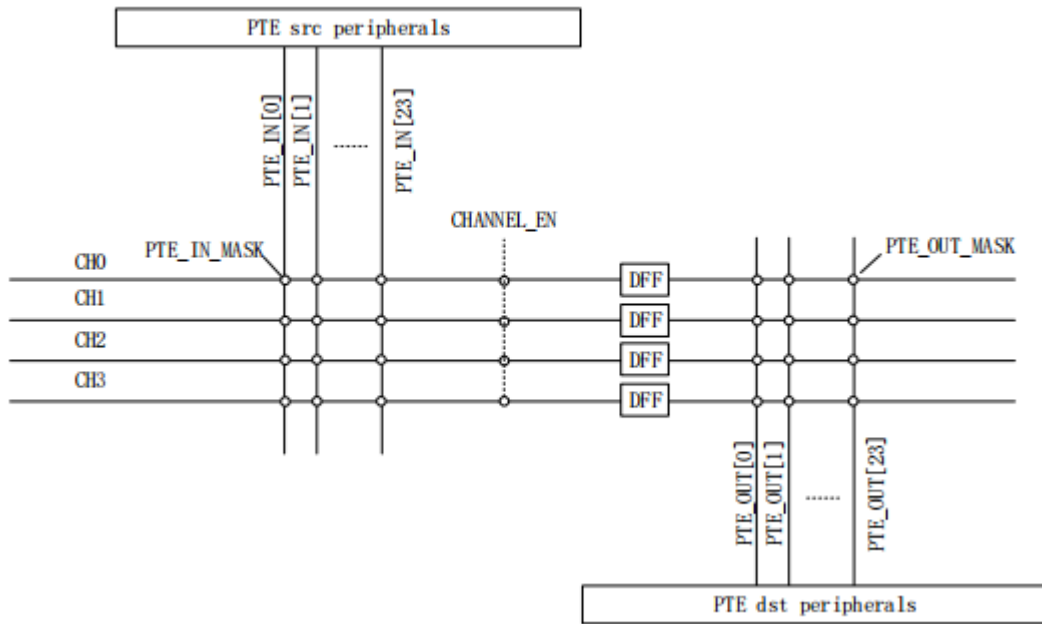


图 13.1: PTE 原理图

13.2 使用方法

13.2.1 方法概述

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

13.2.2 注意点

- 未被清空的 src 中断会循环通过 PTE 触发 dst 外设，使程序陷入死循环
- 未被清空的 PTE 中断会循环触发 dst 外设，使程序陷入死循环
- 使用 PTE 中断会占用 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

13.3 编程指南

13.3.1 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_GPI00_INT      = 13,
    SYSCTRL_PTE_GPI01_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;
```

13.3.2 驱动接口

- PTE_ConnectPeripheral: PTE 外设连接接口
- PTE_EnableChannel: PTE 通道使能接口
- PTE_ChannelClose: 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

13.3.3 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。

```
#define PTE_CH0    SYSCTRL_PTE_CHANNEL_0

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

uint32_t Timer1Isr(void *user_data)
{
    TMR_IntClr(APB_TMR1);
    PTE_OutPeripheralContinueProcess(PTE_CH0);
    return 0;
}

// 仅供参考，不建议注册 PTE 中断
uint32_t PTE0Isr(void *user_data)
{
    PTE_IrqProcess(PTE_CH0);
    return 0;
}

void PTE_Test(void)
{
    PTE_ConnectPeripheral(PTE_CH0,
```

```
        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(PTE_CH0);  
    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。

第十四章 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 $8 \times 20\text{bit}$ 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

14.1 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;
```

14.1.1 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;
}
```

14.1.2 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);
}

```

14.1.3 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;
}

```

14.1.4 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);
}

```

14.1.5 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 PERA_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。

14.1.6 DMA 模式

此模式支持通过 DMA 实时更新门限。

14.1.7 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;
}
```

14.2 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：

```
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。

14.3 PWM 使用说明

14.3.1 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：

¹所有 6 路输入共有此计数器。

```
// 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)
);
```

14.3.2 配置工作模式

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

14.3.3 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。

14.3.4 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
);
```

14.3.5 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);
}
```

14.3.6 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 8，可以存储 8 个 32 比特数据（只有低 20 比特有效，其余比特忽略），相当于 4 组 PWM 配置，所以 trig_cfg 的取值范围为 0..7。

14.4 PCAP 使用说明

14.4.1 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 内存储的数据多于或等于 trig_cfg 时，请求 DMA 传输数据。trig_cfg 的取值范围为 0..7。

```
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)  
);
```

14.4.2 读取计数器

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


第十五章 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。

15.1 功能描述

15.1.1 特点

- 可配置时钟
- 支持过滤器
- 支持 APB 总线
- 支持 DMA

15.1.2 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。

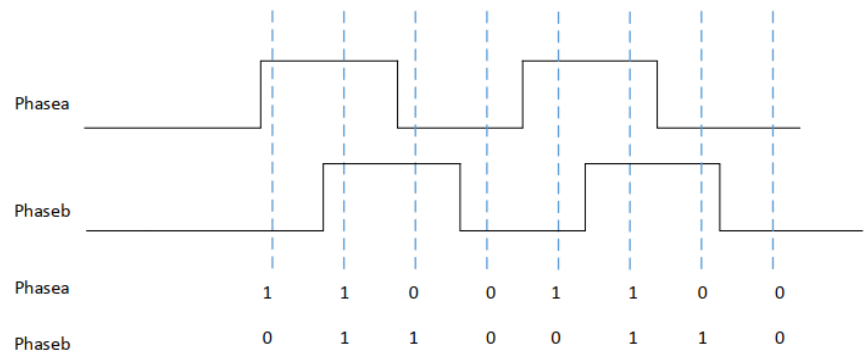


图 15.1: QDEC 顺时针采集数据

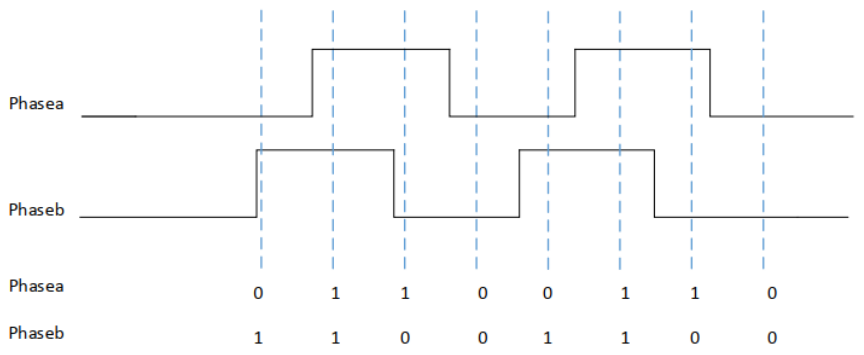


图 15.2: QDEC 逆时针采集数据

15.2 使用方法

15.2.1 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。

15.2.1.1 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。

15.2.1.2 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

出于实际效果和硬件资源等因素考虑，我们推荐开发者选择 **sclk_slow** 作为时钟源

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：如果使用 sclk_slow 时钟，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)
{
    if ((APB_SYSCTRL->QdecCfg >> 15) & 1)
        return;
    uint32_t hclk = SYSCTRL_GetHClk();
    uint32_t slowClk = SYSCTRL_GetSlowClk();
    uint8_t div = hclk / slowClk;
    if (hclk % slowClk)
        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。

在配置完 **qdec** 之后可以选择将 **pclk** 恢复到原来频率，实例可参考 SDK 中 HID mouse 例程。

15.2.1.3 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。

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, 如果效果良好则可以尝试继续加大 miss 值。在保证性能的基础上, 理论上 miss 值越大越好。

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。对于信号质量很差的设备, 如信号很多毛刺, 如果加入 miss 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, 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. 采用较小的 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, 则需判断其可能属于正常情况。

15.2.2 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, 建议配好 qdec 之后将 pclk 及时恢复
- 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

15.3 编程指南

15.3.1 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

15.3.2 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)
{
    if ((APB_SYSCTRL->QdecCfg >> 15) & 1)
        return;
    uint32_t hclk = SYSCTRL_GetHClk();
    uint32_t slowClk = SYSCTRL_GetSlowClk();
    uint8_t div = hclk / slowClk;
    if (hclk % slowClk)
        div++;
    if (!(div >> 4))
        SYSCTRL_SetPclkDiv(div);
}

void test(void)
{
    // setup qdec
    SYSCTRL_ClearClkGateMulti((1 << SYSCTRL_ITEM_APB_PinCtrl) |
                               (1 << SYSCTRL_ITEM_APB_QDEC));
```

```

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

SYSCTRL_SelectQDECCLK(SYSCTRL_CLK_SLOW, 100);
QDEC_PclkCfg();    // set pclk not bigger than sclk_slow
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。

推荐开发者采用 timer 定时轮询的方式读取 qdec 数据，并进行数据处理和上报。具体的 qdec 详细使用实例请参考 SDK 中 HID mouse 例程。

第十六章 实时时钟（RTC）

16.1 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。

特性：

- 可配置计数器大小
- 周期性中断包括半秒、秒、分钟、小时和天
- 可编程报警中断
- 硬件数字微调补偿外部时钟源频偏

16.2 使用说明

16.2.1 RTC 使能

使用 `RTC_Enable` 使能 RTC。

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

16.2.2 获取当前时间

使用 `RTC_GetTime` 获取当前时间，该函数的参数可以读取对应的时、分、秒，函数的返回值即为 RTC 使能后所经历的天数 `day`。

```
uint16_t RTC_GetTime(  
    uint8_t *hour,    //时  
    uint8_t *minute,  //分  
    uint8_t *second   //秒  
);
```

16.2.3 修改时间

使用 `RTC_ModifyTime` 修改当前时间，其中的 `day` 表示 RTC 使能后所经过的天数。

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

修改时间后，需要使用 `RTC_IsModificationDone` 获取当前修正是否已经同步到 RTC 时钟域，返回值为 1 表示已经同步完毕，0 则表示正在更新。

```
int RTC_IsModificationDone(void);
```

16.2.4 获取 RTC Counter 值

使用 `RTC_Current` 可以获取 RTC Counter 值的低 32bit。

```
uint32_t RTC_Current(void);
```

使用 `RTC_CurrentFull` 可以获取 RTC Counter 值的全部 43bit。

```
uint64_t RTC_CurrentFull(void);
```

16.2.5 配置闹钟

使用 RTC_ConfigAlarm

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

16.2.6 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    //比特 2 为 1 时使能 alarm 中断  
                //比特 3 为 1 时使能 day 中断  
                //比特 4 为 1 时使能 hour 中断  
                //比特 5 为 1 时使能 minute 中断  
                //比特 6 为 1 时使能 second 中断  
                //比特 7 为 1 时使能 half-second 中断  
);
```

RTC 支持的中断类型一共 6 种：

```
typedef enum  
{  
    RTC_IRQ_ALARM = 0x04,    //alarm 中断  
    RTC_IRQ_DAY   = 0x08,    //day 中断
```

```
RTC_IRQ_HOUR = 0x10,      //hour 中断
RTC_IRQ_MINUTE = 0x20,     //minute 中断
RTC_IRQ_SECOND = 0x40,     //second 中断
RTC_IRQ_HALF_SECOND = 0x80, //half-second 中断
} rtc_irq_t;
```

上述所提到的六种中断中，alarm 中断通过调用 RTC_ConfigAlarm 设定 alarm 时间，其余中断的出发时间均是固定的，依次为：1 天、1 小时、1 分钟、1 秒钟和半秒钟。

- 例如使能 RTC 设的 alarm 中断，并设定 alarm 时间的 01:45:20

```
RTC_EnableIRQ(RTC_IRQ_ALARM);
RTC_ConfigAlarm(0x1,0x2d,0x14);
RTC_Enable(1);
```

- 例如使能 RTC 的 day 中断

```
RTC_EnableIRQ(RTC_IRQ_DAY);
RTC_Enable(1);
```

16.2.7 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);
```

16.2.8 清除中断

使用 RTC_ClearIntState 清除当前 RTC 的中断状态。

```
void RTC_ClearIntState(  
    uint32_t state    //比特 2 为 1 时清除 alarm 中断  
                      //比特 3 为 1 时清除 day 中断  
                      //比特 4 为 1 时清除 hour 中断  
                      //比特 5 为 1 时清除 minute 中断  
                      //比特 6 为 1 时清除 second 中断  
                      //比特 7 为 1 时清除 half-second 中断  
);
```

16.2.9 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。

16.2.10 睡眠唤醒源

RTC 支持作为低功耗状态的唤醒源：RTC 的闹钟唤醒信号在 `RTC_ConfigAlarm` 指定的时间将系统唤醒。使用 `RTC_EnableDeepSleepWakeupSource` 使能 RTC 的唤醒功能。

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

16.2.11 数字调校

RTC 由外部 32kHz 时钟源驱动，其脉冲发生器产生内部 1Hz 脉冲以增加其时间计数器。如果外部时钟频率不是准确的 32768Hz，则内部产生的 1Hz 脉冲的周期也不准确。数字微调功能可以补偿这些不准确性。

通过 `RTC_Trim` 就可以根据 32kHz 时钟源实际的频率情况进行数字微调：

```
int RTC_Trim(uint32_t cali_value);
```

这里的 `cali_value` 为 32kHz 时钟源的校准值, 可通过 `platform_read_info(PLATFORM_INFO_32K_CALI_VALUE)` 获取。

第十七章 串行外围设备接口（SPI）

SPI 全称为 Serial Peripheral interface 即串行外围设备接口。SPI 是一种高速同步的通信总线。一般使用四个 IO：SDI（数据输入），SDO（数据输出），SCK（时钟），CS（片选）。针对 Flash 的 QSPI 则还需要使用额外的两个 IO：WP（写保护），HOLD（保持）。

17.1 功能概述

- 两个 SPI 模块
- 支持 SPI 主 (Master)& 从 (Slave) 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
- 支持 XIP(SPI0)

17.2 使用说明

17.2.1 时钟以及 IO 配置

使用模块之前，需要打开相应的时钟，并且配置 IO（查看对应 datasheet 获取可用的 IO）：

1. 通过 `SYSCTRL_ClearClkGateMulti` 打开 SPI 时钟,例如 SPI1 则需要打开 `SYSCTRL_ITEM_APB_SPI1`。
2. 配置 IO 的输入功能 `PINCTRL_SelSpiIn`，没有使用到的 IO 可以使用 `IO_NOT_A_PIN` 替代。
3. 使用 `PINCTRL_SetPadMux` 配置 IO 的输出功能。
4. 对于 Slave 的输入，例如 CLK 需要保持默认低电平，则使用 `PINCTRL_Pull` 配置 IO 的上拉功能。

5. 打开 SPI 中断 `platform_set_irq_callback`。
6. 对于时钟大于 20M 的使用场景，IO 的选择有特殊要求，请参考高速时钟和 IO 映射。

以 SPI1 为例，高速 IO 可以为：

```
#define SPI_MIC_CLK      GPIO_GPIO_7
#define SPI_MIC_MOSI     GPIO_GPIO_8
#define SPI_MIC_MISO     GPIO_GPIO_9
#define SPI_MIC_CS       GPIO_GPIO_10
#define SPI_MIC_WP       GPIO_GPIO_11
#define SPI_MIC_HOLD     GPIO_GPIO_12
```

下述示例可以将指定 IO 映射到 SPI1 的 Master 模式：

```
static void setup_peripherals_spi_pin(void)
{
    SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_SPI1)
                                   | (1 << SYSCTRL_ITEM_APB_PinCtrl));

    PINCTRL_SelSpiIn(SPI_PORT_1, SPI_MIC_CLK, SPI_MIC_CS, SPI_MIC_HOLD,
                    SPI_MIC_WP, SPI_MIC_MISO, SPI_MIC_MOSI);
    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);

    platform_set_irq_callback(PLATFORM_CB_IRQ_APBSPi, peripherals_spi_isr, NULL);
}
```

如果是 Slave 模式，则需要额外配置默认上拉，并且输入输出 IO 有区别：

```
static void setup_peripherals_spi_pin(void)
{
    SYSCTRL_ClearClkGateMulti(    (1 << SYSCTRL_ITEM_APB_SPI1)
                                   | (1 << SYSCTRL_ITEM_APB_PinCtrl));
}
```



```
PINCTRL_Pull(IO_SOURCE_SPI1_CLK_IN,PINCTRL_PULL_DOWN);
PINCTRL_Pull(IO_SOURCE_SPI1_CSN_IN,PINCTRL_PULL_UP);
PINCTRL_SelSpiIn(SPI_PORT_1, SPI_MIC_CLK, SPI_MIC_CS, SPI_MIC_HOLD,
                  SPI_MIC_WP, SPI_MIC_MISO, SPI_MIC_MOSI);
PINCTRL_SetPadMux(SPI_MIC_CLK, IO_SOURCE_SPI1_CLK_OUT);
PINCTRL_SetPadMux(SPI_MIC_MISO, IO_SOURCE_SPI1_MISO_OUT);

platform_set_irq_callback(PLATFORM_CB_IRQ_APBSPi, peripherals_spi_isr, NULL);
```

17.2.2 模块初始化

模块的初始化通过 `apSSP_DeviceParametersSet` 和结构体 `apSSP_sDeviceControlBlock` 实现，结构体各个参数为：

- `eSclkDiv`: 时钟分频因子, 决定了 SPI 的时钟速率。该参数和 SPI 模块时钟有关, 在默认配置下, SPI 模块时钟为 24M, `eSclkDiv` 可以直接使用 `SPI_INTERFACETIMINGSCLKDIV_DEFAULT_xx` 宏定义。对于更高的时钟, 需要首先提高 SPI 模块时钟, 然后再计算 `eSclkDiv`, 计算公式和配置方式请参考章节其他配置-> 时钟配置。
- `eSCLKPhase`: 上升沿还是下降沿采样, 参考 `SPI_TransFmt_CPHA_e`。
- `eSCLKPolarity`: 时钟默认是低电平还是高电平, 参考 `SPI_TransFmt_CPOL_e`。
- `eLsbMsbOrder`: bit 传输顺序是 LSB 还是 MSB, 默认是 MSB, 参考 `SPI_TransFmt_LSB_e`。
- `eDataSize`: 每个传输单位的 bit 个数, 8/16/32bit, 参考 `SPI_TransFmt_DataLen_e`。
- `eMasterSlaveMode`: 选择是 Master 还是 Slave 模式, 参考 `SPI_TransFmt_SlvMode_e`。
- `eReadWriteMode`: 选择传输模式: 只读/只写/同时读写, 参考 `SPI_TransCtrl_TransMode_e`。
- `eQuadMode`: 选择是普通 SPI 还是 QSPI, 参考 `SPI_TransCtrl_DualQuad_e`。
- `eWriteTransCnt`: 每次发送的单位个数, 每个单位 `eDataSize` 个 bit, 达到单位个数后, CS 将会拉高, 代表一次传输结束。
- `eReadTransCnt`: 每次接收的单位个数, 每个单位 `eDataSize` 个 bit, 达到单位个数后, CS 将会拉高, 代表一次传输结束。
- `eAddrEn`: 是否需要在数据之前发送地址, 只适用 Master, 参考 `SPI_TransCtrl_AddrEn_e`。
- `eCmdEn`: 是否需要在数据之前发送命令, 只适用 Master, 参考 `SPI_TransCtrl_CmdEn_e`。
- `eInterruptMask`: 需要打开的 SPI 中断类型, 比如 SPI 传输结束中断和 FIFO 中断, 参考 `bsSPI_INTREN_xx`。

- TxThres: 触发 TX FIFO 中断的门限值, 比如可以为 eWriteTransCnt/2, 参考 apSSP_SetTxThres。
- RxThres: 触发 RX FIFO 中断的门限值, 比如可以为 eReadTransCnt/2, 参考 apSSP_SetRxThres。
- SlaveDataOnly: Slave 模式下生效, 如果只有数据, 需要设置为打开, 如果包含地址 eAddrEn 或者命令 eCmdEn, 需要设置改参数为 DISABLE, 参考 SPI_TransCtrl_SlvDataOnly_e。
- eAddrLen: 如果打开了 eAddrEn, 需要选择地址长度, 参考 SPI_TransFmt_AddrLen_e。

具体使用请参考编程指南。

17.2.3 中断配置

通过 eInterruptMask 打开需要的中断:

- bsSPI_INTREN_ENDINTEN: 传输结束 (CS 拉高) 之后会触发该中断。
- bsSPI_INTREN_TXFIFOINTEN: 到达 TxThres 门限值之后, 触发 FIFO 中断。
- bsSPI_INTREN_RXFIFOINTEN: 到达 RxThres 门限值之后, 触发 FIFO 中断。
 - 通过 apSSP_GetDataNumInRxFifo 来判断当前 FIFO 中有效数据的个数。使用 apSSP_ReadFIFO 来读取 FIFO 中的数据, 直到 FIFO 为空。

17.2.4 编程指南

以下提供不同场景下的代码实现供参考。

17.2.4.1 场景 1: 同时读写, 不使用 DMA

其中 SPI 主 (Master) 和 SPI 从 (Slave) 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 IO, 如果是普通模式, 则不需要 SPI_MIC_WP 和 SPI_MIC_HOLD。

17.2.4.1.1 Master 配置

- 配置 IO, 参考时钟以及 IO 配置。
- 模块初始化:

假设每个传输单元是 32bit, 则 mode 0 下的 Master 模式初始化配置为:

```
#define SPI_MASTER_PARAM(DataLen) { SPI_INTERFACETIMINGSCLKDIV_DEFAULT_2M, \
SPI_CPHA_ODD_SCLK_EDGES, SPI_CPOL_SCLK_LOW_IN_IDLE_STATES, \
SPI_LSB_MOST_SIGNIFICANT_BIT_FIRST, SPI_DATALEN_32_BITS, \
SPI_SLVMODE_MASTER_MODE, SPI_TRANSMODE_WRITE_READ_SAME_TIME, \
SPI_DUALQUAD_REGULAR_MODE, DataLen, DataLen, \
SPI_ADDREN_DISABLE, SPI_CMDEN_DISABLE, (1 << bsSPI_INTREN_ENDINTEN), \
0, 0, SPI_SLVDATAONLY_ENABLE, SPI_ADDRLEN_1_BYTE }
```

使用 `APIapSSP_DeviceParametersSet` 来初始化 SPI 模块:

```
#define DATA_LEN (SPI_FIFO_DEPTH)
static void setup_peripherals_spi_module(void)
{
    apSSP_sDeviceControlBlock pParam = SPI_MASTER_PARAM(DATA_LEN);
    apSSP_DeviceParametersSet(APB_SSP1, &pParam);
}
```

- 中断实现

该示例中只打开了 SPI ENDINT 中断，该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);
    }
}
```

- 发送数据

发送数据需要填充数据到 TX FIFO 并且触发传输:

- 使用 `apSSP_WriteFIFO` 写入需要传输的数据，通过 `apSSP_TxFifoFull` 来判断 TX FIFO 状态，写入数据直到 FIFO 变满。FIFO 的深度为 8，超过 8 个数据可以分为多次发送。
- `apSSP_WriteCmd` 来触发传输，此场景中 `eAddrEn` 或者 `eCmdEn` 都是关闭的，但是仍然需要填充一个任意数据（比如 `0x00`）来触发传输。
- 使用 `apSSP_GetSPIActiveStatus` 等待发送结束，或者查看对应中断。
- 发送的数据格式和长度需要和 `eDataSize` 等参数对应。

```
uint32_t write_data[DATA_LEN];
void peripherals_spi_send_data(void)
{
    apSSP_WriteCmd(APB_SSP1, 0x00, 0x00);//trigger transfer
    for(i = 0; i < DATA_LEN; i++)
    {
        if(apSSP_TxFifoFull(APB_SSP1)){ break; }
        apSSP_WriteFIFO(APB_SSP1, write_data[i]);
    }
    while(apSSP_GetSPIActiveStatus(APB_SSP1));
}
```

• 接收数据

在 SPI 发送结束后，通过 `apSSP_GetDataNumInRxFifo` 查看并读取 RX FIFO 中的数据。

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

• 使用流程

- 设置 IO, `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。

17.2.4.1.2 Slave 配置

- 配置 IO，参考时钟以及 IO 配置。
- 模块初始化：

Slave 的初始化和 Master 相同，只需要将 mode 切换为 SPI_SLVMODE_SLAVE_MODE：

```
#define SPI_SLAVE_PARAM(DataLen) { SPI_INTERFACETIMINGSCLKDIV_DEFAULT_2M, \
SPI_CPHA_ODD_SCLK_EDGES, SPI_CPOL_SCLK_LOW_IN_IDLE_STATES, \
SPI_LSB_MOST_SIGNIFICANT_BIT_FIRST, SPI_DATALEN_32_BITS, \
SPI_SLVMODE_SLAVE_MODE, SPI_TRANSMODE_WRITE_READ_SAME_TIME, \
SPI_DUALQUAD_REGULAR_MODE, DataLen, DataLen, \
SPI_ADDREN_DISABLE, SPI_CMDEN_DISABLE, (1 << bsSPI_INTREN_ENDINTEN), \
0, 0, SPI_SLVDATAONLY_ENABLE, SPI_ADDRLLEN_1_BYTE }
```

调用 apSSP_DeviceParametersSet 来初始化 SPI 模块：

```
static void setup_peripherals_spi_module(void)
{
    apSSP_sDeviceControlBlock pParam = SPI_SLAVE_PARAM(DATA_LEN);
    apSSP_DeviceParametersSet(APB_SSP1, &pParam);
}
```

- 发送数据

Slave 的待发送数据需要提前放到 TX FIFO 中。比如在初始化之后，就可以往 FIFO 中填充发送数据，同时需要判断 apSSP_TxFifoFull：

```
for(i = 0; i < DATA_LEN; i++)
{
    if(apSSP_TxFifoFull(APB_SSP1)){ break; }
    apSSP_WriteFIFO(APB_SSP1, write_data[i]);
}
```

- 接收数据

Slave 的数据接收需要在中断中进行，bsSPI_INTREN_ENDINTEN 中断代表传输结束，此时可以读取 RX FIFO 中的内容，如果数据长度大于 FIFO 深度，则需要打开 RxThres，在 FIFO 中断中读取接收数据，或者使用 DMA 方式，否则数据会丢失。

```
static uint32_t peripherals_spi_isr(void *user_data)
{
    uint32_t stat = apSSP_GetIntRawStatus(APB_SSP1), i;
    if(stat & (1 << bsSPI_INTREN_ENDINTEN))
    {
        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);
    }
}
```

- 使用流程

- 设置 IO，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。

17.2.4.2 场景 2：同时读写，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 IO，如果是普通模式，则不需要 SPI_MIC_WP 和 SPI_MIC_HOLD。

17.2.4.2.1 Master 配置

- 配置 IO，参考时钟以及 IO 配置。

- 模块初始化，参考场景 lMaster 配置。
- DMA 初始化

使用之前需要初始化 DMA 模块：

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

- DMA 设置

1. 设置 DMA 将指定地址的数据搬运到 SPI1 TX FIFO，并打开 DMA。SPI1 启动后，搬运自动开始：

```
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);
}
```

2. 设置 DMA 将 SPI1 RX FIFO 的数据搬运到指定地址：

```
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);
}

```

- 中断实现

该示例中只打开了 SPI ENDINT 中断，该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);
    }
}

```

- 接收数据配置

1. 首先需要打开 SPI 模块中的 DMA 功能 apSSP_SetRxDmaEn。
2. 设置每次传输时接收和发射的个数，改参数已经在模块初始化中设置过了，此处为可选项，如果需要更新，可以通过 apSSP_SetTransferControlRdTranCnt 和 apSSP_SetTransferControlWrTranCnt 来更新。注意：该场景下，发射和接收的个数需要相同。
3. 配置 DMA。

```

#define SPI_DMA_RX_CHANNEL    (1)//DMA channel 1
uint32_t read_data[DATA_LEN];

void peripherals_spi_read_data(void)
{

```



```
apSSP_SetRxDmaEn(APB_SSP1,1);
apSSP_SetTransferControlRdTranCnt(APB_SSP1,DATA_LEN);
peripherals_spi_rxfifo_to_dma(SPI_DMA_RX_CHANNEL, read_data,
                              sizeof(read_data));
}
```

- 发送数据配置

类似于接收，发射需要打开 SPI 的 DMA 功能，并且配置 DMA。

```
#define SPI_DMA_TX_CHANNEL (0)//DMA channel 0
uint32_t write_data[DATA_LEN];

void peripherals_spi_push_data(void)
{
    apSSP_SetTxDmaEn(APB_SSP1,1);
    apSSP_SetTransferControlWrTranCnt(APB_SSP1,DATA_LEN);
    peripherals_spi_dma_to_txfifo(SPI_DMA_TX_CHANNEL, write_data,
                                  sizeof(write_data));
}
```

- 发送数据

1. 在需要发送数据时，通过 DMA 填充待发射数据到 TX FIFO。
2. 配置接收数据的 DMA，同时读写模式下，发射和接收同步进行。
3. 配置命令触发传输。
4. 等待发射结束 apSSP_GetSPIActiveStatus，关闭 DMA 功能。

```
void peripherals_spi_send_data(void)
{
    peripherals_spi_read_data();
    peripherals_spi_push_data();
    apSSP_WriteCmd(APB_SSP1, 0x00, 0x00);//trigger transfer
    while(apSSP_GetSPIActiveStatus(APB_SSP1));
}
```

```
apSSP_SetTxDmaEn(APB_SSP1,0);  
}
```

- 使用流程
 - 设置 IO, `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。

17.2.4.2.2 Slave 配置

- 配置 IO, 参考时钟以及 IO 配置。
- 模块初始化, 参考场景 1slave 配置。
- DMA 初始化, 与 Master 的 DMA 初始化相同, 参考 `setup_peripherals_dma_module`。
- DMA 设置, 与 Master 的 DMA 设置相同, 参考 `peripherals_spi_dma_to_txfifo` 和 `peripherals_spi_rxfifo_to_dma`。
- 接收数据配置, 与 Master 相同, 参考 `peripherals_spi_read_data`。
- 发送数据配置, 与 Master 相同, 参考 `peripherals_spi_push_data`。
- 传输前准备
 1. 对于 Slave, 如果需要在第一次传输时发送数据, 则需要提前将需要发送的数据配置到 DMA, 配置 `peripherals_spi_push_data`。
 2. 使用 `peripherals_spi_read_data` 配置接收 DMA, 等待 Master 传输。

```
peripherals_spi_read_data();  
peripherals_spi_push_data();
```

- 中断实现

1. bsSPI_INTREN_ENDINTEN 的触发代表当前传输结束。
2. 检查接收数据。
3. 如果需要准备下一次传输，使用 peripherals_spi_read_data 建立接收 DMA。
4. 如果需要准备下一次传输，准备发送数据，使用 peripherals_spi_push_data 建立发射 DMA。
5. 清除中断标志。

```
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();
        peripherals_spi_push_data();
        apSSP_ClearIntStatus(APB_SSP1, 1 << bsSPI_INTREN_ENDINTEN);
    }
}
```

- 使用流程
 - 设置 IO, 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。

17.2.5 其他配置

17.2.5.1 时钟配置 (pParam.eSclkDiv)

17.2.5.1.1 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： $(spi\ interface\ clock)/(2 \times (eSclkDiv + 1))$ ，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)
```

17.2.5.1.2 高速时钟配置 SPI0 和 SPI1 的高速时钟配置有一些不同，注意：高速时钟需要使用特定的 IO，请查看 **SPI 高速时钟和 pin 的映射**。

17.2.5.1.2.1 SPI0 对于更高的时钟，需要在配置 pParam.eSclkDiv 之前打开额外配置，打开方式如下

- 首先通过 SYSCTRL_GetPLLClk() 获取 PLL 时钟，此处假设为 336M。
- 切换 SPI0 到高速时钟 SYSCTRL_SelectSpiClk(SPI_PORT_0, SYSCTRL_CLK_HCLK+1)，此处的入参代表分频比 2，即最终的 spi interface clock 为 $336/2 = 168.0M$ 。
- 通过计算公式 $(spi\ interface\ clock)/(2 \times (eSclkDiv + 1))$ 得到不同时钟下的 eSclkDiv。
- 通过 SYSCTRL_GetClk(SYSCTRL_ITEM_AHB_SPI0) 来确认 interface clock 是否生效。

在以上举例中，可以使用以下宏定义来配置 pParam.eSclkDiv

```
#define SPI_INTERFACETIMINGSCLKDIV_SPI0_21M    (1)
#define SPI_INTERFACETIMINGSCLKDIV_SPI0_14M    (2)
```

以下的 API 可以实现相同的功能，假设 PLL 时钟为 336M，入参为 21000000(21M)，则该 API 可以完成 SPI0 时钟配置，并且返回 eSclkDiv。入参需要和 PLL 成倍数关系。

```
uint8_t setup_peripherals_spi_0_high_speed_interface_clk(uint32_t spiClk)
{
    //for spi0 only
```

```
uint8_t eSclkDiv = 0;
uint32_t spiIntfClk;
uint32_t pllClk = SYSCTRL_GetPLLClk();

SYSCTRL_SelectSpiClk(SPI_PORT_0, SYSCTRL_CLK_PLL_DIV_1+1);

spiIntfClk = SYSCTRL_GetClk(SYSCTRL_ITEM_AHB_SPI0);
eSclkDiv = ((spiIntfClk/spiClk)/2)-1;

return eSclkDiv;
}
```

调用方式为:

```
pParam.eSclkDiv = setup_peripherals_spi_0_high_speed_interface_clk(21000000);
```

17.2.5.1.2.2 SPI1 对于更高的时钟，需要在配置 pParam.eSclkDiv 之前打开 HCLK 配置，打开方式如下：

- 首先查看 HCLK 频率:SYSCTRL_GetHClk()。
 - 如果有需要可以使用 SYSCTRL_SelectHClk(SYSCTRL_CLK_PLL_DIV_1+3); 来修改 HCLK 时钟。修改方法为，首先使用 SYSCTRL_GetPLLClk() 获取 PLL 时钟，入参为分频比，假如 PLL 时钟为 336M，则 SYSCTRL_CLK_PLL_DIV_1+6 为 7 分频，最终 HClk 为 336/7=48M。
- 将 SPI1 时钟切换到 HCLK:SYSCTRL_SelectSpiClk(SPI_PORT_1, SYSCTRL_CLK_HCLK)。
- 通过计算公式 $(spi \ interface \ clock)/(2 \times (eSclkDiv + 1))$ 得到不同时钟下的 eSclkDiv。
- 通过 SYSCTRL_GetClk(SYSCTRL_ITEM_APB_SPI1) 来确认 interface clock 是否生效。

假设 HCLK 为 112M（即 spi interface clock），通过计算公式可以得到不同时钟下的 eSclkDiv 为：

```
#define SPI_INTERFACETIMINGSCLKDIV_SPI1_19M    (2)
#define SPI_INTERFACETIMINGSCLKDIV_SPI1_14M    (3)
```

以下的 API 可以实现相同的功能，假设 HCLK 时钟为 112M，入参为 14000000(14M)，则该 API 可以完成 SPI1 时钟配置，并且返回 eSclkDiv。入参需要和 HCLK 成倍数关系

```
uint8_t setup_peripherals_spi_1_high_speed_interface_clk(uint32_t spiClk)
{
    uint8_t eSclkDiv = 0;
    uint32_t spiIntfClk;
    uint32_t hClk = SYSCTRL_GetHClk();

    SYSCTRL_SelectSpiClk(SPI_PORT_1, SYSCTRL_CLK_HCLK);

    spiIntfClk = SYSCTRL_GetClk(SYSCTRL_ITEM_APB_SPI1);
    eSclkDiv = ((spiIntfClk/spiClk)/2)-1;

    return eSclkDiv;
}
```

调用方式为：

```
pParam.eSclkDiv = setup_peripherals_spi_1_high_speed_interface_clk(14000000);
```

17.2.5.2 QSPI 使用

QSPI 的 bit 顺序以及 MOSI/MISO 的含义和普通 SPI 不同，但大部分读写的配置是共享的。使用 QSPI 需要在普通 SPI 的配置的基础上，做一些额外修改：

17.2.5.2.1 IO 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, SPI_MIC_MISO, SPI_MIC_MOSI);
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);
```

对于 Slave，则还需要额外配置 CS 的内部上拉：

```
PINCTRL_Pull(IO_SOURCE_SPI1_CLK_IN, PINCTRL_PULL_DOWN);
PINCTRL_Pull(IO_SOURCE_SPI1_CSN_IN, PINCTRL_PULL_UP);
```

17.2.5.2.2 QSPI Master

17.2.5.2.2.1 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，Master 还需要配置 pParam.eReadWriteMode，例如：

- SPI_TRANSMODE_WRITE_READ：读和写顺序执行，比如首先利用四线完成写操作，然后再读。
- SPI_TRANSMODE_WRITE_DUMMY_READ：先写后读，在 Write 和 Read 之间添加 dummy（默认为 8 个 clk cycle）。

对于 Slave，则读写顺序相反：

- SPI_TRANSMODE_READ_WRITE：先读后写。

- **SPI_TRANSMODE_READ_DUMMY_WRITE**: 先读后写，在 Read 和 Write 之间添加 dummy（默认为 8 个 clk cycle）。

SPI_TransCtrl_TransMode_e 定义了可以使用的 pParam.eReadWriteMode。

配置参数举例：

```
#define SPI_MASTER_PARAM(DataLen) { SPI_INTERFACETIMINGSCLKDIV_DEFAULT_2M, \
SPI_CPHA_ODD_SCLK_EDGES, SPI_CPOL_SCLK_LOW_IN_IDLE_STATES, \
SPI_LSB_MOST_SIGNIFICANT_BIT_FIRST, SPI_DATALEN_32_BITS, SPI_SLVMODE_MASTER_MODE, \
SPI_TRANSMODE_WRITE_DUMMY_READ, SPI_DUALQUAD_QUAD_IO_MODE, DataLen, DataLen, \
SPI_ADDREN_ENABLE, SPI_CMDEN_ENABLE, (1 << bsSPI_INTREN_ENDINTEN), \
0, 0, SPI_SLVDATAONLY_DISABLE, SPI_ADDRLEN_1_BYTE }

#define SPI_SLAVE_PARAM(DataLen) { SPI_INTERFACETIMINGSCLKDIV_DEFAULT_2M, \
SPI_CPHA_ODD_SCLK_EDGES, SPI_CPOL_SCLK_LOW_IN_IDLE_STATES, \
SPI_LSB_MOST_SIGNIFICANT_BIT_FIRST, SPI_DATALEN_32_BITS, SPI_SLVMODE_SLAVE_MODE, \
SPI_TRANSMODE_READ_DUMMY_WRITE, SPI_DUALQUAD_QUAD_IO_MODE, DataLen, DataLen, \
SPI_ADDREN_ENABLE, SPI_CMDEN_ENABLE, (1 << bsSPI_INTREN_ENDINTEN), \
0, 0, SPI_SLVDATAONLY_DISABLE, SPI_ADDRLEN_1_BYTE }
```

17.2.5.2.2.2 Dummy cnt 配置 pParam.eReadWriteMode 中指定了 dummy 的位置，通过 apSSP_SetTransferControlDummyCnt 配置 dummy 个数。

API 的入参为 dummy cnt，通过以下关系转换成 API CLK 的 dummy cycles。

$$cycles = (dummy \ cnt + 1) \times ((pParam.eDataSize)/(spi \ io \ width))$$

dummy cnt + 1	pParam.eDataSize	spi io width	cycles
1	8	1(single)	8
1	32	4(quad-qspi)	8

以上述的参数配置为例，则下述 API 可以将 dummy cycle 配置为 16：

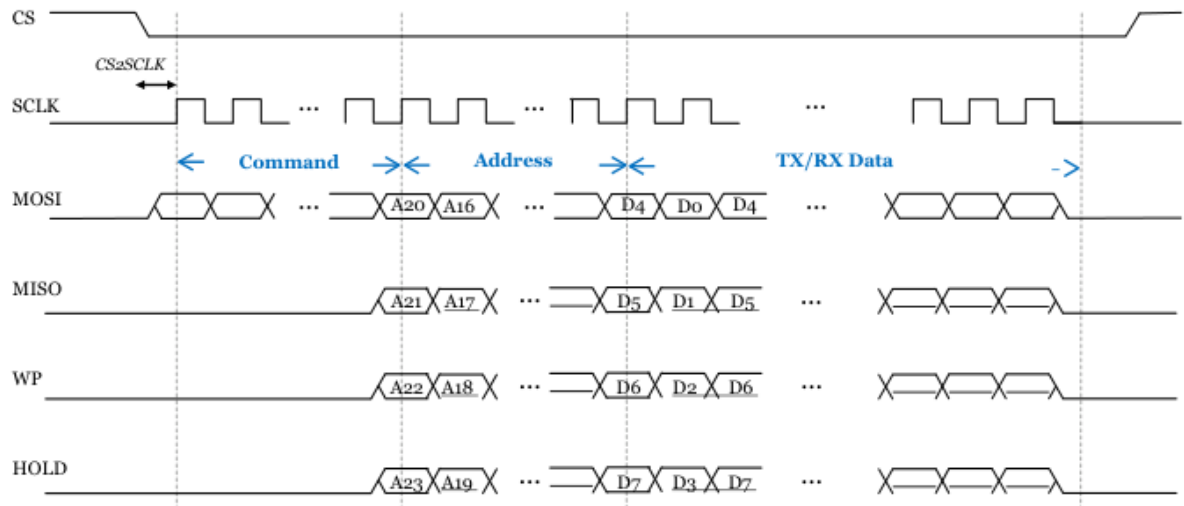

```
apSSP_SetTransferControlDummyCnt(APB_SSP1, 1); //dummy cnt = 1;
```

注意，额外的配置需要在 apSSP_DeviceParametersSet 之后以避免参数覆盖。

17.2.5.2.2.3 Addr 格式 QSPI 的 Command 和 Addr 通过 apSSP_WriteCmd 配置，首先发送 Command，Command 固定占用 MOSI 的 8 个 cycle，然后发送 Addr，Addr 的长度通过 pParam.eAddrLen 配置，发送格式通过 apSSP_SetTransferControlAddrFmt 配置。

- SPI_ADDRFMT_SINGLE_MODE: 默认配置，Addr 只占用 MOSI
- SPI_ADDRFMT_QUAD_MODE: Addr 以 QSPI 的模式发送, 占用 MOSI, MISO, WP, HOLD

该 API 只适用于 SPI Master。假如 Addr 配置为 3 字节以及 QUAD MODE，则传输格式为：



17.2.5.2.2.4 QSPI 读取和 XIP 当通过 SPI 外接 FLASH 模块的时候，可以通过芯片内置功能直接读取 FLASH 内容而不需要操作 SPI。该功能只支持 SPI0。

操作步骤为，

- 首先打开时钟并且配置 SPI IO，参考时钟以及 io 配置。对于 QSPI，请参考 QSPI 的 io 配置。

IO 必须使用高速时钟和 io 映射中 SPI0 的指定 IO。

- 然后需要额外打开 SPI 的直接读取功能（不需要其他的 SPI 模块配置）

```
void apSSP_SetMemAccessCmd(SSP_TypeDef *SPI_BASE, apSSP_sDeviceMemRdCmd cmd);
```

cmd 用来选择 FLASH 的读取命令，该命令需要查看对应的 FLASH 手册来获取。
apSSP_sDeviceMemRdCmd 中列出了所有支持的读取命令以及该命令的格式：

```
typedef enum
```

```
{
```

```
SPI_MEMRD_CMD_03 = 0 ,//read command 0x03 + 3bytes address(regular mode) + data (regular mode)
```

```
SPI_MEMRD_CMD_0B = 1 ,//read command 0x0B + 3bytes address(regular mode) + 1byte dummy + data (regular mode)
```

```
SPI_MEMRD_CMD_3B = 2 ,//read command 0x3B + 3bytes address(regular mode) + 1byte dummy + data (regular mode)
```

```
SPI_MEMRD_CMD_6B = 3 ,//read command 0x6B + 3bytes address(regular mode) + 1byte dummy + data (regular mode)
```

```
SPI_MEMRD_CMD_BB = 4 ,//read command 0xBB + 3bytes + 1byte 0 address(dual mode) + data (dual mode)
```

```
SPI_MEMRD_CMD_EB = 5 ,//read command 0xEB + 3bytes + 1byte 0 address(quad mode) + 2bytes dummy + data (quad mode)
```

```
SPI_MEMRD_CMD_13 = 8 ,//read command 0x13 + 4bytes address(regular mode) + data (regular mode)
```

```
SPI_MEMRD_CMD_0C = 9 ,//read command 0x0C + 4bytes address(regular mode) + 1byte dummy + data (regular mode)
```

```
SPI_MEMRD_CMD_3C = 10, //read command 0x3C + 4bytes address(regular mode) + 1byte dummy + data (regular mode)
```

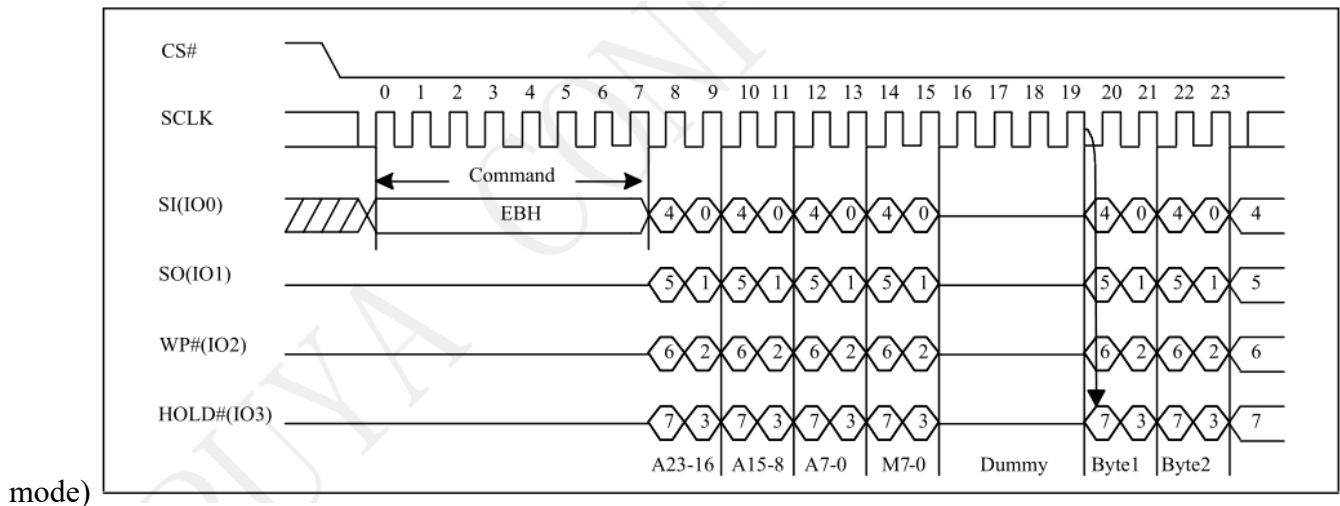
```
SPI_MEMRD_CMD_6C = 11, //read command 0x6C + 4bytes address(regular mode) + 1byte dummy + data (regular mode)
```

```
SPI_MEMRD_CMD_BC = 12, //read command 0xBC + 4bytes + 1byte 0 address(dual mode) + data (dual mode)
```

```
SPI_MEMRD_CMD_EC = 13, //read command 0xEC + 4bytes + 1byte 0 address(quad mode) + 2bytes dummy + data (quad mode)
```

```
}apSSP_sDeviceMemRdCmd;
```

以 SPI_MEMRD_CMD_EB 为例,该命令的格式为:command(0xEB)+address(4bytes)+dummy(2bytes)+data(quad mode)



- 直接读取外置 FLASH 的内容

外置 FLASH 的内容映射在 AHB_QSPI_MEM_BASE 开始的连续空间（32MB）。

- 使用举例：

```
setup_peripherals_spi_pin();
apSSP_SetMemAccessCmd(AHB_SSP0, SPI_MEMRD_CMD_EB);
// read by *((uint8_t*)AHB_QSPI_MEM_BASE + i), i=[0,32MB)
```

- XIP

通过 SPI 模块下载程序到外置 FLASH 模块，然后跳转到外置 FLASH 地址来执行代码。

17.2.5.2.3 QSPI Slave Addr 只适用于 Master，当作为 Slave 的时候，只支持 Command 命令。Slave 在 QSPI 下集成了两个 Command:

- QSPI read only: 0x0E
格式固定为：Command(单线，8bit) + Dummy(单线，8bit) + read Data(四线)
- QSPI write only: 0x54
格式固定为：Command(单线，8bit) + Dummy(单线，8bit) + write Data(四线)

当使用这两个 Command 的时候，Master 需要按照上述格式配置。

17.2.5.3 高速时钟和 IO 映射

只有 SPI0 支持 XIP，最大时钟速率和 VBAT 有关：

Function	SPI0	SPI1
XIP	最大速率：96M（取决于 VBAT）	不支持

SPI0 和 SPI1 最大可以支持的时钟在不同场景下有区别：

Function	SPI0	SPI1
Master	传输中只有读或者写：96M	24M
Master	传输中既有读也有写：48M	24M
Slave	24M	24M

高速时钟对 IO 能力有要求，因此对于时钟大于 20M 的情况，需要使用以下固定的 IO，其中 SPI0 的 IO 映射固定。对于 SPI1，SCLK 必须使用 IO7，其他 IO8/9/10/11/12 可以任意配置。

SPI0	SPI1
GPIO18:CS	GPIO7: SCLK
GPIO19:SCLK	GPIO8: CS/HOLD/WP/MISO/MOSI
GPIO20:HOLD	GPIO9: CS/HOLD/WP/MISO/MOSI
GPIO26:WP	GPIO10:CS/HOLD/WP/MISO/MOSI
GPIO27:MISO	GPIO11:CS/HOLD/WP/MISO/MOSI
GPIO28:MOSI	GPIO12:CS/HOLD/WP/MISO/MOSI

第十八章 系统控制（SYSCTRL）

18.1 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

18.1.1 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_GPI00    ,
    SYSCTRL_ITEM_APB_GPI01    ,
    // ...
    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;
```

18.1.2 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 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 div_{pre} （前置分频）、 $loop$ （环路分频）和 div_{output} （输出分频）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
- USB²
- 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 18.1。

¹每个外设可单独对 *hclk* 门控。

²仅高速时钟。

表 18.1: 各硬件外设的时钟源

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

18.1.3 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;
```


18.2 使用说明

18.2.1 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);
```

18.2.2 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);
```

18.2.3 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,    // 前置分频
    uint32_t loop,       // 环路分频
    uint32_t div_output // 输出分频
);
```

例如，假设慢时钟配置为 24MHz，下面的代码将 *hclk* 配置为 112MHz 并读取到变量：

```
SYSCTRL_ConfigPLLClk(5, 70, 1);
SYSCTRL_SelectHClk(SYSCTRL_CLK_PLL_DIV_3);
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` 的定义为：

```
typedef enum
{
    SYSCTRL_CLK_SLOW,           // 使用 sclk_slow
    SYSCTRL_CLK_32k = ...,      // 使用 32KHz 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 18.1 可知，I2S 可使用 *sclk_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 分频：

```
SYSCtrl_SelectI2sClk(SYSCtrl_CLK_PLL_DIV_5);
```

3. 读取时钟频率

使用 SYSCtrl_GetClk 读取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_5);
// 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：

```
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_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，需要进行调谐以尽量接近标称值。通过 SYSCTRL_AutoTuneSlowRC 可自动完成调谐³：

```
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⁴，可以把此参数储存起来，如果系统重启，可通过 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。

³以 24MHz 晶体为参考。

⁴比如认为 SYSCTRL_AutoTuneSlowRC 耗时过长。

18.2.4 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⁵：

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

18.2.5 电源相关

18.2.5.1 内核电压

使用 `SYSCTRL_SetLD0Output` 配置内核 LDO 的输出电压（默认 1.200V）：

```
void SYSCTRL_SetLD0Output(int level);
```

其中 `level` 的表示方式为 `SYSCTRL_LDO_OUTPUT_CORE_1V000\...\SYSCTRL_LDO_OUTPUT_CORE_1V300`，对应的电压范围是 [1.000V, 1.300V]，以 20mV 步进。此配置在低功耗模式下不会丢失。

18.2.5.2 内置 Flash 电压

使用 `SYSCTRL_SetLD0OutputFlash` 配置内部 Flash LDO 的输出电压（默认 2.100V）：

```
void SYSCTRL_SetLD0OutputFlash(int level);
```

其中 `level` 的表示方式为 `SYSCTRL_LDO_OUTPUT_FLASH_2V100\...\SYSCTRL_LDO_OUTPUT_FLASH_3V100`，对应的电压范围是 [2.100V, 3.100V]，以 100mV 步进。此配置在低功耗模式下不会丢失。

⁵`SYSCTRL_SelectUsedDmaItems` 的 `items` 参数里对应的比特为 0

18.2.6 唤醒后的时钟配置

ROM 内的启动程序包含一个时钟配置程序，当系统初始上电或者从低功耗状态唤醒时，这个程序就按照预定参数配置几个关键时钟及看门狗。

默认情况下，这个时钟配置程序是打开的，所使用的预定参数如下：

- PLL：打开，div_pre、loop、div_output 分别为 5、70、1；
- hclk：clk_pll 3 分频；
- 片内 Flash 时钟：clk_pll 2 分频；
- 看门狗：不使能。

由于初始上电或者唤醒后，慢时钟来自外部 24MHz 晶体，可计算出上述几个时钟的频率如表 18.2。

表 18.2: 默认参数对应的时钟频率

时钟	频率 (MHz)
PLL	336
hclk	112
片内 Flash	168

通过以下两个函数向这个程序传递参数：

1. 使能这个程序并设置参数

```
void SYSCTRL_EnableConfigClocksAfterWakeup(
    uint8_t enable_pll,           // 是否使能 PLL
    uint8_t pll_loop,            // PLL loop
    SYSCTRL_ClkMode hclk,        // hclk
    SYSCTRL_ClkMode flash_clk,   // 片内 Flash 时钟
    uint8_t enable_watchdog);    // 是否使能看门狗
```

如果使能看门狗，系统唤醒后，时钟配置程序将其配置为 4.5s 后触发超时、复位系统。

2. 禁用这个程序

```
void SYSCTRL_DisableConfigClocksAfterWakeup(void);
```

如果禁用这个程序，系统唤醒后几个时钟的频率见表 18.3。

表 18.3: 禁用时钟配置程序时重新唤醒后几个关键时钟的频率

时钟	频率 (MHz)
PLL	384
<i>hclk</i>	24
片内 Flash	24

18.2.7 RAM 相关

SoC 内部包含多个内存块，根据用途可分为：仅供 CPU 使用的 SYS RAM，CPU 和蓝牙 Modem 皆可使用的 SHARE RAM 以及高速缓存（Cache）。部分内存块既可以作为 SYS RAM 也可以作为 SHARE RAM（见表 18.4），在不同的软件包内将被配置为不同用途。

表 18.4: 可作为 SYS/SHARE RAM 的内存块

名称	大小 (KiB)	配置为 SYS RAM 时的 地址范围	配置为 SHARE RAM 时 的地址范围	备注
SYS_MEM_BLOCK_0		0x20000000~0x20003FFF	不支持	不可关闭
SYS_MEM_BLOCK_1		0x20004000~0x20007FFF	0x40128000~0x4012BFFF	
REMAPPABLE_BLOCK_0		0x20008000~0x2000BFFF	0x40124000~0x40127FFF	
REMAPPABLE_BLOCK_0		0x2000C000~0x2000DFFF	0x40122000~0x40123FFF	
SHARE_MEM_BLOCK_0		不支持	0x40120000~0x40121FFF	

在 *noos_mini*, *mini* 软件包共配置 56 KiB SYS RAM，8 KiB SHARE RAM；在其它软件包里，SYS、SHARE RAM 各 32 KiB。详见表 18.5。注意，虽然 SYS_MEM_BLOCK_1 也可用作 SHARE RAM，但是在所有的软件包里它总是被用作 SYS RAM。

表 18.5: 各软件包里的 SYS/SHARE RAM 配置

名称	SYS RAM		SHARE RAM	
	地址范围	包含的内存块	地址范围	包含的内存块
<i>mini</i> , <i>noos_mini</i>	0x20000000 ~ 0x2000DFFF	SYS_MEM_BLOCK_0 SYS_MEM_BLOCK_1 REMAP- PABLE_BLOCK_0 REMAP- PABLE_BLOCK_1	0x40120000 ~ 0x40121FFF	SHARE_MEM_BLOCK_0
其它	0x20000000 ~ 0x20007FFF	SYS_MEM_BLOCK_0 SYS_MEM_BLOCK_1	0x40120000 ~ 0x40127FFF	SHARE_MEM_BLOCK_0 REMAP- PABLE_BLOCK_0 REMAP- PABLE_BLOCK_1

表 18.4 中的内存块支持低功耗数据保持。所有这些内存块默认都是开启的，部分内存块可关闭以节省功耗。如果程序中实际用到的 SYS RAM 较少，可通过 `SYSCtrl_SelectMemoryBlocks` 选择所要使用的内存块，并关闭不使用的内存块：

```
void SYSCtrl_SelectMemoryBlocks(
    uint32_t block_map);
```

例如在一个使用 *mini* 软件包的程序里，如果确认只需要 32 KiB 的 SYS RAM，那么为了降低功耗，可关闭 `REMAP_PABLE_BLOCK_0` 和 `REMAP_PABLE_BLOCK_1`，保留其它内存块：

```
SYSCtrl_SelectMemoryBlocks(
    SYSCtrl_SYS_MEM_BLOCK_0 | SYSCtrl_SYS_MEM_BLOCK_1 |
    SYSCtrl_SHARE_MEM_BLOCK_0);
```

另有 2 个内存块可配置为 SYS RAM 或者高速缓存，其配置可通过 `SYSCtrl_CacheControl` 动态修改。将其映射为 SYS RAM 后，可按照表 18.6 中的地址访问。

表 18.6: 可用作高速缓存的内存块

名称	大小 (KiB)	配置为 SYS RAM 时的地址范围
D-Cache-M	8	0x2000E000~0x2000FFFF
I-Cache-M	8	0x20010000~0x20011FFF

这两个内存块都默认处于 Cache 模式。当需要更多的 RAM 时,通过 `SYSCTRL_CacheControl` 可将这两块内存映射为普通 RAM:

```
void SYSCTRL_CacheControl(  
    SYSCTRL_CacheMemCtrl i_cache,  
    SYSCTRL_CacheMemCtrl d_cache  
);
```

`SYSCTRL_CacheMemCtrl` 包含两个值,对应 Cache 模式和 SYS MEM 模式:

```
typedef enum  
{  
    SYSCTRL_MEM_BLOCK_AS_CACHE = 0,  
    SYSCTRL_MEM_BLOCK_AS_SYS_MEM = 1,  
} SYSCTRL_CacheMemCtrl;
```



务必注意:

1. 从低功耗状态唤醒时,这两个内存块都将恢复默认值 `AS_CACHE`;
2. 低功耗状态时,这两个内存块里的数据(无论处于哪种模式)都会丢失;
3. 映射为普通 RAM 后,系统缺少高速缓存,性能有可能明显下降。

第十九章 定时器（TIMER）

19.1 功能概述

ING916XX 系列具有功能完全相同的 3 个计时器，每个计时器包含两个通道，每个通道支持 1 个 32 位计时器，所以系统中一共有 6 个 32 位计时器。既可以用作脉冲宽度调制器 (PWM)，也可以用作简单的定时器。

特性：

- 支持 AMBA2.0 支持 APB 总线
- 最多 4 个多功能定时器
- 提供 6 种使用场景（定时器和 PWM 的组合）
- 计时器时钟源可选
- 计时器可以暂停

19.2 使用说明

19.2.1 设置 TIMER 工作模式

使用 TMR_SetOpMode 设置 TIMER 的工作模式。

```
void TMR_SetOpMode(  
    TMR_TypeDef *pTMR,      //定时器外设地址  
    uint8_t ch_id,          //通道 ID  
    uint8_t op_mode,         //工作模式  
    uint8_t clk_mode,        //时钟模式
```

```
uint8_t pwm_park_value
);
```

关于 TMR_SetOpMode 中的参数 `pwm_park_value` 的值将影响 PWM 的输出：

- 若为 0：通道被禁用时，PWM 输出为低电平；通道启用时，较低周期的 PWM 计数器先计数；
- 若为 1：通道被禁用时，PWM 输出为高电平；通道启用时，较高周期的 PWM 计数器先计数；

TIMER 具有 6 种不同的工作模式，可以大致分为三类：定时器功能、PWM 功能、（定时器+PWM）组合功能。

- 定时器功能

32 位定时器可以分别作为 1 个 32 位定时器、2 个 16 位定时器、4 个 8 位定时器，定义如下所示。

```
#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
```

- 脉冲宽度调制器功能

定时器的本质其实是计数器，所以可以拆分为 2 个 16 位的计数器来产生 PWM 信号。

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

- 组合功能

定时器与 PWM 的功能可以组合使用，对应的组合方式有两种：1）一个 8bitPWM 和一个 16 位计时器；2）一个 8 位 PWM 和两个 8 位计时器。

```
#define TMR_CTL_OP_MODE_8BIT_PWM_16BIT_TIMER_x1 6 // MIXED: PWM with two 8bit cou  
#define TMR_CTL_OP_MODE_8BIT_PWM_8BIT_TIMER_x2 7 // MIXED: PWM with two 8bit cou
```



注意: 要更改当前工作的定时器通道模式, 必须先禁用该通道, 然后将通道设置为新模式并启用它。

TIMER 的时钟源有两种, 分别是内部时钟和外部时钟, 定义如下所示。

```
#define TMR_CLK_MODE_EXTERNAL 0 //external clock  
#define TMR_CLK_MODE_APB 1 //internal clock
```

19.2.2 获取时钟频率

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  
);
```

19.2.3 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, //通道 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[16:23]	bits[24:31]
TMR_CTL_OP_MODE_32BIT_TIMER_x1				
TMR_CTL_OP_MODE_16BIT_TIMER_x2				
TMR_CTL_OP_MODE_8BIT_TIMER_x4		Timer0	Timer1	Timer2
TMR_CTL_OP_MODE_16BIT_PWM				
TMR_CTL_OP_MODE_8BIT_PWM_16BIT_TIMER_x1		PWM low period	PWM high period	
TMR_CTL_OP_MODE_8BIT_PWM_8BIT_TIMER_x2		Timer1	PWM low period	PWM high period

关于上述格中分配的重载值有两点说明：* 定时器模式下，某个 Timer 在其（重载值 + 1）个计数周期产生一次中断；* PWM 模式下，高周期和低周期的频率值分别是对应重载值 + 1。

19.2.4 使能 TIMER

使用 TMR_Enable 使能对应通道上的一个或多个 timer。

```
void TMR_Enable(  
    TMR_TypeDef *pTMR,  
    uint8_t ch_id,  
    uint8_t mask //比特 0 为 1 配置 TIMER0  
                //比特 1 为 1 配置 TIMER1  
                //比特 2 为 1 配置 TIMER2  
                //比特 3 为 1 配置 TIMER3  
);
```



注意，如果相应的通道 不存在或在通道模式中它 不是一个有效的设备，则定时器或 PWM 不能被启用。例如，当 0 号通道设置为 32 位定时器模式时，0 号通道的 Timer 1 不能使能。

19.2.5 获取 TIMER 的比较值

使用 TMR_GetCMP 获取定时器的比较输出。

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

19.2.6 获取 TIMER 的计数器值

使用 TMR_GetCNT 获取定时器的计数值。

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

19.2.7 计时器暂停

使用 TMR_PauseEnable 可以将计时器暂停，计时器的 counter 将保持当前的计数值，取消暂停之后将恢复计数。

```
void TMR_PauseEnable(  
    TMR_TypeDef *pTMR,  
    uint8_t enable  
);
```

19.2.8 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  
);
```

19.2.9 处理中断状态

使用 TMR_IntHappened 一次性获取某个通道上所有 Timer（最多 4 个 Timer）的中断触发状态，返回非 0 值表示该 Timer 上产生了中断请求。第 n 比特（第 0 比特为最低比特）对应 Timer n 上的中断触发状态。

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

使用 TMR_IntClr 可以一次性清除某个通道上所有定时器的中断状态。

```
void TMR_IntClr(  
    TMR_TypeDef *pTMR,  
    uint8_t ch_id,  
    uint8_t mask //比特 0 为 1 清除对应通道上的 Timer0  
                //比特 1 为 1 清除对应通道上的 Timer1  
                //比特 2 为 1 清除对应通道上的 Timer2  
                //比特 3 为 1 清除对应通道上的 Timer3  
);
```


19.3 使用示例

19.3.1 使用计时器功能及暂停功能

将 TIMER1 的通道 0 设置为 TMR_CTL_OP_MODE_32BIT_TIMER_x1 模式，并设定每 1 秒产生一次中断：

```
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)); //4999
TMR_Enable(APB_TMR1, 0, 0xf);
TMR_IntEnable(APB_TMR1, 0, 0xf);
```

19.3.2 使用 TIMER 的 PWM 功能

将 TIMER1 的通道 0 的工作模式设置为 TMR_CTL_OP_MODE_16BIT_PWM，并使用 13 号引脚输出 10HzPWM 信号。

```
#define PIN_TMR_PWM 13

static void setup_peripheral_timer(void)
{
    SYSCTRL_ClearClkGateMulti( (1 << SYSCTRL_ITEM_APB_SysCtrl)
                               |(1 << SYSCTRL_ITEM_APB_PinCtrl)
                               |(1 << SYSCTRL_ITEM_APB_TMR1));

    SYSCTRL_SelectTimerClk(TMR_PORT_1, SYSCTRL_CLK_32k);
    TMR_SetOpMode(APB_TMR1, TMR_CTL_OP_MODE_16BIT_PWM, TMR_CLK_MODE_EXTERNAL, 0);
    TMR_SetReload(APB_TMR1, 0, 0x00090009); // 9 9
    TMR_Enable(APB_TMR1, 0, 0xf);

    PINCTRL_SetPadMux(PIN_TMR_PWM, IO_SOURCE_TIMER1_PWM0_B);
}
```

19.3.3 通道 0 产生 2 个周期性中断

使用 TIMER1 通道 0 生成 2 个中断: 一个用于每 1000 个 APB 时钟周期, 另一个用于每 3000 个 APB 周期。

```
static void setup_peripheral_timer(void)
{
    SYSCTRL_ClearClkGateMulti( (1 << SYSCTRL_ITEM_APB_SysCtrl)
                               |(1 << SYSCTRL_ITEM_APB_PinCtrl)
                               |(1 << SYSCTRL_ITEM_APB_TMR1));

    SYSCTRL_SelectTimerClk(TMR_PORT_1, SYSCTRL_CLK_32k);
    TMR_SetOpMode(APB_TMR1, 0, TMR_CTL_OP_MODE_16BIT_TIMER_x2, TMR_CLK_MODE_APB, 0);
    TMR_SetReload(APB_TMR1, 0, 0x0BB703E7); // 2999 999

    TMR_IntEnable(APB_TMR1, 0, 0x3); //Ch0Int0 Ch0Int1
    TMR_Enable(APB_TMR1, 0, 0x3);    //Ch0TMR0En Ch0TMR1En
}
```

19.3.4 产生 2 路对齐的 PWM 信号

使用 TIMER1 的通道 0 和通道 1 分别生成两路 PWM 信号 PWM0 和 PWM1, 对应参数设置如下所示: PWM0: 周期 = 30 个外部时钟周期, 占空比 = 1/3 PWM1: 周期 = 60 个外部时钟周期, 占空比 = 1/3 将两路 PWM 对齐, 并分别由引脚 13、14 输出。

```
#define PIN_TMR_PWM0 13
#define PIN_TMR_PWM1 14

static void setup_peripheral_timer(void)
{
    SYSCTRL_ClearClkGateMulti( (1 << SYSCTRL_ITEM_APB_SysCtrl)
                               |(1 << SYSCTRL_ITEM_APB_PinCtrl)
                               |(1 << SYSCTRL_ITEM_APB_TMR1));
```

```
SYSCTRL_SelectTimerClk(TMR_PORT_1, SYSCTRL_CLK_32k);

TMR_SetOpMode(APB_TMR1, 0, TMR_CTL_OP_MODE_16BIT_PWM, TMR_CLK_MODE_EXTERNAL, 1);
TMR_SetOpMode(APB_TMR1, 1, TMR_CTL_OP_MODE_16BIT_PWM, TMR_CLK_MODE_EXTERNAL, 1);
TMR_SetReload(APB_TMR1, 0, 0x00090013); //9 19
TMR_SetReload(APB_TMR1, 1, 0x00130027); //19 39
TMR_Enable(APB_TMR1, 0, 0xf);
TMR_Enable(APB_TMR1, 1, 0xf);

PINCTRL_SetPadMux(PIN_TMR_PWM0, IO_SOURCE_TIMER1_PWM0_B);
PINCTRL_SetPadMux(PIN_TMR_PWM1, IO_SOURCE_TIMER1_PWM1_A);
}
```


第二十章 通用异步收发传输器（UART）

20.1 功能概述

UART 全称 Universal Asynchronous Receiver/Transmitter,即通用异步收发传输器件。UART 对接收的数据执行串并转换，对发送的数据执行并串转换。

特性：

- 支持硬件流控
- 可编程波特率发生器，最高波特率可达 7000000bps
- 独立的发送和接收 FIFO
- 单个组合中断，包括接收（包括超时）、传输、调制解调器状态和错误状态中断，每个中断可屏蔽
- 支持 DMA 方式

20.2 使用说明

20.2.1 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,      //UART 参数结构体 & 设备地址  
    uint32_t ClockFrequency,  //时钟信号的频率  
    uint32_t BaudRate         //波特率  
);
```

20.2.2 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
);
```

20.2.3 UART 初始化

在使用 UART 之前，需要先通过 apUART_Initialize 对 UART 进行初始化。

```
void apUART_Initialize(
    UART_TypeDef* pBase,
    UART_sStateStruct* UARTx, //uart 状态结构体
    uint32_t IntMask //中断掩码
);
```

初始化 UART 之前需要初始化如下所示的 UART 状态结构体：

```
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;        //CTSSEN
```

```
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;
```

常用的中断掩码如下所示，IntMask 是它们的组合。

```
#define UART_INTBIT_RECEIVE      0x10 //receive interrupt
#define UART_INTBIT_TRANSMIT     0x20 //transmit interrupt
```

例如，配置并初始化串口，开启接收中断和发送中断，设置串口波特率为 115200:

- 首先，创建 UART 配置函数 config_uart，在函数内初始化 UART 状态结构体，并配置必要的状态参数，然后调用 apUART_Initialize 初始化串口。

```
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, UART_INTBIT_RECEIVE | UART_INTBIT_TRANSMIT);
}

```

使用时只需要如下所示调用 `config_uart` 函数即可。

```
config_uart(OSC_CLK_FREQ, 115200);
```

20.2.4 UART 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 Rx FIFO 是否为空。

```

uint8_t apUART_Check_RXFIFO_EMPTY(
    UART_TypeDef* pBase
);

```


- 使用 apUART_Check_RXFIFO_FULL 查询 Rx FIFO 是否已满。

```
uint8_t apUART_Check_RXFIFO_FULL(  
    UART_TypeDef* pBase  
);
```

- 使用 apUART_Check_TXFIFO_EMPTY 查询 Tx FIFO 是否为空。

```
uint8_t apUART_Check_TXFIFO_EMPTY(  
    UART_TypeDef* pBase  
);
```

- 使用 apUART_Check_TXFIFO_FULL 查询 Tx FIFO 是否已满。

```
uint8_t apUART_Check_TXFIFO_FULL(  
    UART_TypeDef* pBase  
);
```

20.2.5 UART 中断使能/禁用

用 apUART_Enable_TRANSMIT_INT 使能发送中断，用 apUART_Disable_TRANSMIT_INT 禁用发送中断；用 apUART_Enable_RECEIVE_INT 使能接收中断，用 apUART_Disable_RECEIVE_INT 禁用接收中断。

中断默认是禁用的，使能中断既能用上述 apUART_Enable_TRANSMIT_INT 和 apUART_Enable_RECEIVE_INT 的方式，也可以通过 apUART_Initialize 初始化串口是设置参数 IntMask 的值使能相应的中断，详情请参考UART 初始化

20.2.6 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。

20.2.7 发送数据

使用 `UART_SendData` 发送数据。

```
void UART_SendData(  
    UART_TypeDef* pBase,  
    uint8_t Data  
);
```

20.2.8 接收数据

使用 `UART_ReceData` 接收数据。

```
uint8_t UART_ReceData(  
    UART_TypeDef* pBase  
);
```

20.2.9 DMA 传输模式使能

使用 `UART_DmaEnable` 使能 UART 的 DMA 工作模式，可以使用 DMA 完成对串口数据的收发，从而不占用 CPU 的资源。

```
void UART_DmaEnable(  
    UART_TypeDef *pBase,  
    uint8_t tx_enable,    //发送使能 (1)/禁用 (0)  
    uint8_t rx_enable,    //接收使能 (1)/禁用 (0)  
    uint8_t dma_on_err  
);
```

20.3 示例代码

20.3.1 UART 接收变长字节数据

1. UART+FIFO 方式

```
char dst[256];
int index = 0;
void setup_peripheral_uart()
{
    APB_UART0->FifoSelect = (0 << bsUART_TRANS_INT_LEVEL ) |
                            (0X10 << bsUART_RECV_INT_LEVEL ) ;
    APB_UART0->IntMask = (1 << bsUART_RECEIVE_INTENAB) | (0 << bsUART_TRANSMIT_INTENAB) |
                        (1 << bsUART_TIMEOUT_INTENAB);
    APB_UART0->Control = 1 << bsUART_RECEIVE_ENABLE |
                        1 << bsUART_TRANSMIT_ENABLE |
                        1 << bsUART_ENABLE |
                        0 << bsUART_CTS_ENA |
                        0 << bsUART_RTS_ENA;
}

uint32_t uart_isr(void *user_data)
{
    uint32_t status;

    while(1)
    {
        status = apUART_Get_all_raw_int_stat(APB_UART0);

        if (status == 0)
            break;

        APB_UART0->IntClear = status;
    }
}
```

```

// rx int
if (status & (1 << bsUART_RECEIVE_INTENAB))
{
    while (apUART_Check_RXFIFO_EMPTY(APB_UART0) != 1)
    {
        char c = APB_UART0->DataRead;
        dst[index] = c;
        index++;
    }
}

// rx timeout_int
if (status & (1 << bsUART_TIMEOUT_INTENAB))
{
    while (apUART_Check_RXFIFO_EMPTY(APB_UART0) != 1)
    {
        char c = APB_UART0->DataRead;
        dst[index] = c;
        index++;
    }
}

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

return 0;
}

void uart_peripherals_read_data()
{
    //注册 uart0 中断
    platform_set_irq_callback(PLATFORM_CB_IRQ_UART0, uart_isr, NULL);
    setup_peripheral_uart();
}

```

2. UART+FIFO+DMA 方式

```

#define bsDMA_INT_C_MASK          1
#define bsDMA_DST_REQ_SEL        4
#define bsDMA_SRC_REQ_SEL        8
#define bsDMA_DST_ADDR_CTRL      12
#define bsDMA_SRC_ADDR_CTRL      14
#define bsDMA_DST_MODE           16
#define bsDMA_SRC_MODE           17
#define bsDMA_DST_WIDTH          18
#define bsDMA_SRC_WIDTH          21
#define bsDMA_SRC_BURST_SIZE     24

#define DMA_RX_CHANNEL_ID  1
#define DMA_TX_CHANNEL_ID  0

DMA_Descriptor test __attribute__((aligned (8)));
char dst[256];
char src[] = "Finished to receive a frame!\n";

void setup_peripheral_dma()
{
    printf("setup_peripheral_dma\n");
    SYSCTRL_ClearClkGate(SYSCTRL_ITEM_APB_DMA);

    //配置 DMA 接收
    APB_DMA->Channels[1].Descriptor.Ctrl = ((uint32_t)0x0 << bsDMA_INT_C_MASK)
                                           | ((uint32_t)0x0 << bsDMA_DST_REQ_SEL)
                                           | ((uint32_t)0x0 << bsDMA_SRC_REQ_SEL)
                                           | ((uint32_t)0x0 << bsDMA_DST_ADDR_CTRL)
                                           | ((uint32_t)0x2 << bsDMA_SRC_ADDR_CTRL) //DMA
                                           | ((uint32_t)0x0 << bsDMA_DST_MODE)
                                           | ((uint32_t)0x1 << bsDMA_SRC_MODE) //
                                           | ((uint32_t)0x0 << bsDMA_DST_WIDTH)
                                           | ((uint32_t)0x0 << bsDMA_SRC_WIDTH)
                                           | ((uint32_t)0x2 << bsDMA_SRC_BURST_SIZE); //4 t

```

```

    APB_DMA->Channels[1].Descriptor.SrcAddr = (uint32_t)&APB_UART0->DataRead;
    APB_DMA->Channels[1].Descriptor.DstAddr = (uint32_t)dst;
    APB_DMA->Channels[1].Descriptor.TranSize = 48;

    DMA_EnableChannel(1, &APB_DMA->Channels[1].Descriptor);
}

//添加 UART 通过 DMA 发送的配置
void UART_trigger_DmaSend(void)
{
    DMA_PrepareMem2Peripheral( &test,
                               SYSCTRL_DMA_UART0_TX,
                               src, strlen(src),
                               DMA_ADDRESS_INC, 0);
    DMA_EnableChannel(DMA_TX_CHANNEL_ID, &test);
}

void setup_peripheral_uart()
{
    APB_UART0->FifoSelect = (0 << bsUART_TRANS_INT_LEVEL ) |
                           (0x7 << bsUART_RECV_INT_LEVEL ) ;
    APB_UART0->IntMask = (0 << bsUART_RECEIVE_INTENAB) | (0 << bsUART_TRANSMIT_INTENAB) |
                       (1 << bsUART_TIMEOUT_INTENAB);
    APB_UART0->Control = 1 << bsUART_RECEIVE_ENABLE |
                       1 << bsUART_TRANSMIT_ENABLE |
                       1 << bsUART_ENABLE |
                       0 << bsUART_CTS_ENA |
                       0 << bsUART_RTS_ENA;
}

uint32_t uart_isr(void *user_data)
{
    uint32_t status;

```

```

printf("@%x  #%x  #%x\n", APB_UART0->IntMask, APB_UART0->IntRaw, APB_UART0->Interrupt);

while(1)
{
    status = apUART_Get_all_raw_int_stat(APB_UART0);

    if (status == 0)
        break;

    APB_UART0->IntClear = status;

    // rx timeout_int
    if (status & (1 << bsUART_TIMEOUT_INTENAB))
    {
        while (apUART_Check_RXFIFO_EMPTY(APB_UART0) != 1)
        {
            char c = APB_UART0->DataRead;
            int index = APB_DMA->Channels[1].Descriptor.DstAddr - (uint32_t)dst;
            dst[index] = c;
            if (index == 255)
            {
                APB_DMA->Channels[1].Descriptor.DstAddr = (uint32_t)dst;
                UART_trigger_DmaSend();
            }
            else
                APB_DMA->Channels[1].Descriptor.DstAddr++;

            APB_DMA->Channels[1].Descriptor.TranSize = 48;
        }
        printf("\nlen=%d, dst = %s\n", APB_DMA->Channels[1].Descriptor.TranSize, dst);
    }
}

return 0;
}

```

```
void uart_peripherals_read_data()  
{  
    //注册 uart0 中断  
    platform_set_irq_callback(PLATFORM_CB_IRQ_UART0, uart_isr, NULL);  
    setup_peripheral_uart();  
    setup_peripheral_dma();  
}
```


第二十一章 通用串行总线 (USB)

21.1 功能概述

- 支持 full-speed (12 Mbps) 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

21.2 使用说明

21.2.1 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

21.2.2 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 1ms (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 has been reset. Each 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 when the device has observed no bus traffic for
a specified period */
USB_DEVICE_SUSPENDED,
USB_DEVICE_TEST_RESET_DONE
}USB_DEVICE_STATE_E;

```

21.2.3 设置 IO

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.c 中的 bsp_usb_init()。

```

// ATTENTION ! FIXED IO FOR USB on 916 series
#define USB_PIN_DP GPIO_GPIO_16
#define USB_PIN_DM GPIO_GPIO_17

```

21.2.4 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);

```

21.2.5 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 before any usb usage
 *
 *
 * @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 the USB host.
 *
 * @param[out] null.
 */
extern USB_ERROR_TYPE_E USB_InitConfig(USB_INIT_CONFIG_T *config);
```

21.2.6 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 EP0(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() 以及 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_TRANSFER 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。

21.2.6.1 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。

21.2.6.2 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。

21.2.6.3 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);
    // setup timer for 10ms, then disable resume signal
    platform_set_timer(internal_bsp_usb_device_remote_wakeup_stop,16);
}
```

21.2.7 常用 driver API

21.2.7.1 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。

- ep: 数据发送对应的 Endpoint，需要使用 USB_EP_DIRECTION_IN。
- buffer: buffer 地址需要是四字节对齐 c __attribute__ ((aligned (4)))。
- size: 需要小于 512*MPS，例如对于 EP0，如果 MPS 为 64bytes，则 size 需要小于 512×64。
- flag: NULL。
- 如果成功则返回 U_TRUE，否则返回 U_FALSE。

```
extern USB_ERROR_TYPE_E USB_SendData(uint8_t ep, void* buffer,
                                     uint16_t size, uint32_t flag);
```

21.2.7.2 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。

- ep: 数据发送对应的 Endpoint，需要使用 USB_EP_DIRECTION_OUT。
- buffer: buffer 地址需要是四字节对齐 c `__attribute__((aligned(4)))`。
- size: 需要是 MPS 的整数倍，如果期望接收的数据小于 MPS，参考 flag 设置。
- flag:
 - 1<<USB_TRANSFERT_FLAG_FLEXIBLE_RECV_LEN: 当接收数据小于 MPS 时需要设置。
- 如果成功则返回 U_TRUE，否则返回 U_FALSE。

```
extern USB_ERROR_TYPE_E USB_RecvData(uint8_t ep, void* buffer,
                                     uint16_t size, uint32_t flag);
```

21.2.7.3 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);
```

21.2.7.4 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 bsp_usb_disable(void);
```

21.2.7.5 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);
```

21.2.7.6 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);
```

21.2.8 使用场景

21.2.8.1 example 0: WINUSB

WinUSB 是适用于 USB 设备的通用驱动程序，随附在 Windows 系统中。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），可以直接使用 WINUSB 而不需要实现驱动程序。如果已将设备定义为 WinUSB 设备，Windows 会自动加载 Winusb.sys。

- 参考：\ING_SDK\sdks\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

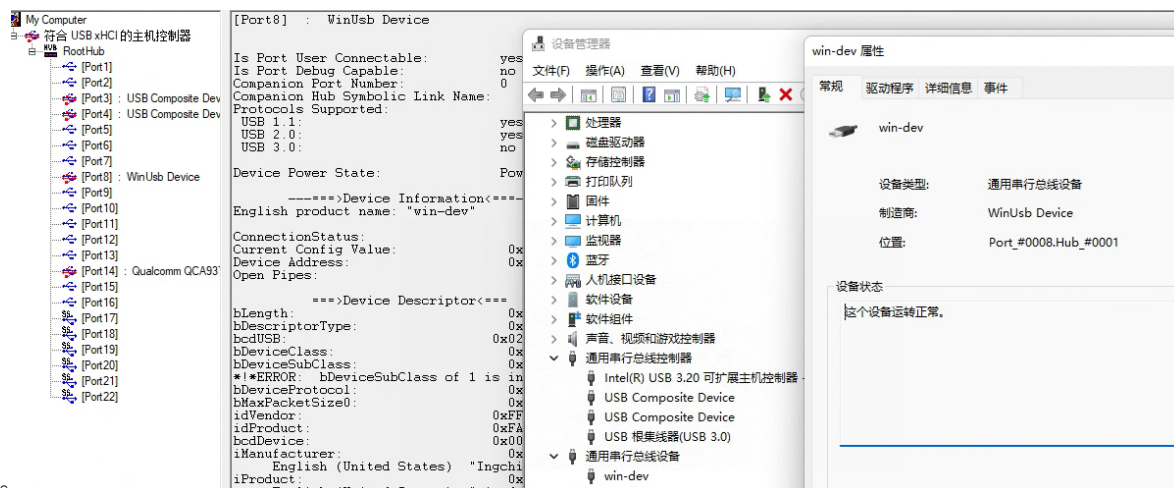
```
#define USB_STRING_PRODUCT {16,0x3,'w',0,'i',0,'n',0,'-',0,'d',0,'e',0,'v',0}
```

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 2 xxxx xxxx
```

- VID:PID 的数值请查看 `USB_DEVICE_DESCRIPTOR`。
- `-w`: 代表写命令。
- `2`: 传输类型, 2 为 `bulk transfer`。
- `xxxx`: 需要传输的数据 (32bit), 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 success,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, 但是 bsp layer 中需要做相应的修改 (使用 in endpoint 发送数据给 host):

```
ing_usb.exe VID:PID -r 2
```

- VID:PID 的数值请查看 USB_DEVICE_DESCRIPTOR。
- -r: 代表读命令。
- 2: 传输类型, 2 为 bulk transfer。

21.2.8.2 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。

21.2.8.2.1 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。

21.2.8.2.2 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;
```

21.2.8.2.3 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, &ReportKeybDescriptor, size, 0);
                    KeybReport.pending = U_FALSE;
                }break;
            }
        }
    }
}
... \ING_SDK\sdk\src\BSP\bsp_usb_hid.c
```

21.2.8.2.4 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, uint8_t press);
```

- 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);
```

21.2.8.3 example 3: USB MSC

该示例提供了以下功能，通过 BSP_USB_MSC_FUNC 来选择：

- **BSP_USB_MSC_FLASH_DISK**: 在指定的 FLASH 空间内初始化一个 FAT16 文件系统。在成功枚举后，可以在 host 端（电脑侧）以操作磁盘的形式直接读取/写入文件。
- **BSP_USB_MSC_FLASH_DISK_NO_VFS**: 类似 BSP_USB_MSC_FLASH_DISK 但是没有提供文件系统，适用于用户自定义的存储设备，当成功枚举后，host 端（电脑侧）会提示需要格式化磁盘，选择相应的参数，完成格式化后，即可以按照正常磁盘使用。
- **BSP_USB_MSC_FLASH_DISK_DOWNLOADER**: 拖拽下载功能。打开磁盘后，放入 bin 文件，则可以完成下载并重启。
- 参考：\ING_SDK\sdk\src\BSP\bsp_usb_msc.c

应用层需要调用 bsp_usb_init() 初始化 USB 模块, 插入电脑，完成枚举，则可以使用对应功能。

第二十二章 看门狗（WATCHDOG）

22.1 功能概述

看门狗的本质就是一个定时器，其功能主要是防止程序跑飞，同时也能防止程序在线运行时出现死循环，一旦发生错误就向芯片内部发出重启信号。

特性：

- 支持 AMBA 2.0 APB 总线
- 当看门狗超时，提供中断和重启的组合
- 为控制/重启寄存器提供写保护机制
- 可编程定时器时钟源
- 可配置的用于寄存器写保护和定时器重启的魔数
- 看门狗定时器可外部暂停

ING916XX 的看门狗定时器提供了一个两级机制，如下图所示（图 22.1）。：

- 1) 第一阶段 中断阶段：若 watchdog 中断启用，则在中断计时结束时会产生中断信号 `wdt_int`;
- 2) 第二阶段 复位阶段：若 watchdog 复位启用，并且在复位超时时间结束前 watchdog 未重启，则会产生复位信号 `wdt_rst` 使得系统复位。

22.2 使用说明

22.2.1 配置看门狗

使用 `TMR_WatchDogEnable3` 配置并启用看门狗。

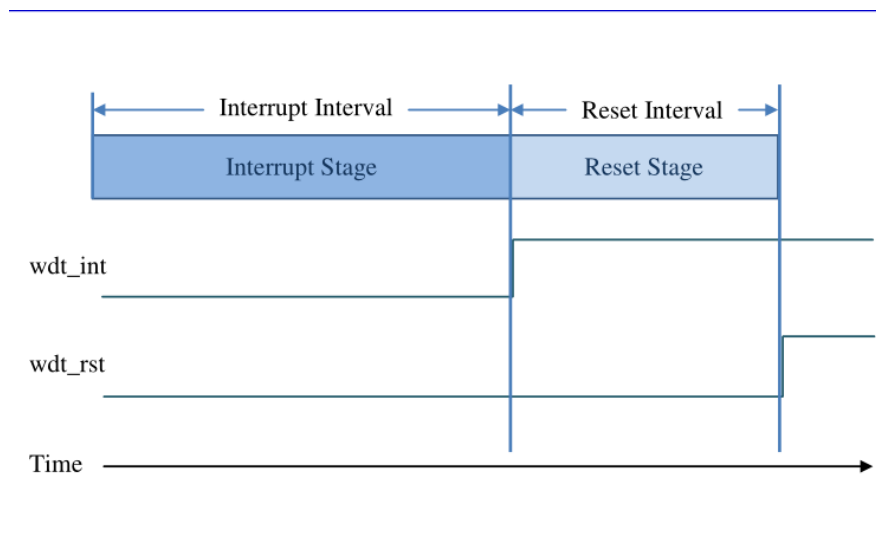


图 22.1: WDT 阶段图

```
void TMR_WatchDogEnable3(
    wdt_inttime_interval_t int_timeout, , //中断阶段时间
    wdt_rsttime_interval_t rst_timeout, //复位阶段时间
    uint8_t enable_int           //中断使能 (1)/禁用 (0)
);
```

其中中断阶段时间和复位阶段的时间设置分别支持 16 种和 8 种，分别如下述的枚举变量 `wdt_inttime_interval_t` 和 `wdt_rsttime_interval_t` 所示。默认使用的 `PCLK=32.768MHz`，所以对应 `int_timeout == WDT_INTTIME_INTERVAL_1S` 时，对应的中断阶段时间为 1s，其余时间以及复位阶段的超时时间以此类推。

```
typedef enum
{
    WDT_INTTIME_INTERVAL_2MS      = 0,    //0.001953125s
    WDT_INTTIME_INTERVAL_8MS      = 1,    //0.0078125s
    WDT_INTTIME_INTERVAL_31MS     = 2,    //0.03125s
    WDT_INTTIME_INTERVAL_62MS     = 3,    //0.0625s
    WDT_INTTIME_INTERVAL_125MS    = 4,
    WDT_INTTIME_INTERVAL_250MS    = 5,
    WDT_INTTIME_INTERVAL_500MS    = 6,
```

```

WDT_INTTIME_INTERVAL_1S          = 7,
WDT_INTTIME_INTERVAL_4S          = 8,
WDT_INTTIME_INTERVAL_16S         = 9,
WDT_INTTIME_INTERVAL_1M_4S       = 10,    //64s
WDT_INTTIME_INTERVAL_4M_16S      = 11,    //256s
WDT_INTTIME_INTERVAL_17M_4S      = 12,    //1024s
WDT_INTTIME_INTERVAL_1H_8M_16S   = 13,    //4096s
WDT_INTTIME_INTERVAL_4H_33M_6S   = 14,    //16384s
WDT_INTTIME_INTERVAL_18H_12M_16S = 15     //65536s
}wdt_inttime_interval_t;

typedef enum
{
    WDT_RSTTIME_INTERVAL_4MS       = 0,    //3.90625ms
    WDT_RSTTIME_INTERVAL_8MS       = 1,    //7.8125ms
    WDT_RSTTIME_INTERVAL_15MS      = 2,    //15.625ms
    WDT_RSTTIME_INTERVAL_31MS      = 3,    //31.25ms
    WDT_RSTTIME_INTERVAL_62MS      = 4,    //62.5ms
    WDT_RSTTIME_INTERVAL_125MS     = 5,
    WDT_RSTTIME_INTERVAL_250MS     = 6,
    WDT_RSTTIME_INTERVAL_500MS     = 7
}wdt_rsttime_interval_t;

```

为了方便使用，通过宏定义将 ING916xx 和 ING918xx 接口统一为 TMR_WatchDogEnable。

```

#define TMR_WatchDogEnable(timeout) do { uint64_t TMR_CLK_FREQ = OSC_CLK_FREQ;uint32_t cnt =
    for (uint8_t i = 1; i < 10; i++,mode++) { if (cn
    TMR_WatchDogEnable3(mode, WDT_RSTTIME_INTERVAL_5

```

代码中会将设置的时间映射到结构体 `wdt_inttime_interval_t` 中，但是由于 *ING916XX* 的 WDT 能设置的时间是离散的，所以为了方便使用，我们只选取从 `WDT_INTTIME_INTERVAL_1S` 到 `WDT_INTTIME_INTERVAL_18H_12M_16S`。



由于 *ING916XX* 和 *ING918XX* 的看门狗在机制上有差别，所以使用 *ING916XX* 时推荐使用 `TMR_WatchDogEnable3`。若要在 *ING916XX* 中使用 `TMR_WatchDogEnable` 需注意：*ING918XX* 的看门狗没有中断机制，实际的复位超时时间为设定时间的两倍；而在 *ING916XX* 中的中断阶段和复位阶段的超时时间都是可以设置的，复位时间我们默认 **0.5s**。所以若用 `TMR_WatchDogEnable(OSC_CLK_FREQ * 1)` 设置的时间在 **918** 和 **916** 上分别是 **2s** 和 **1.5s**。

22.2.2 重启看门狗

使用 `TMR_WatchDogRestart` 重启看门狗，也就是我们俗称的喂狗。

```
void TMR_WatchDogRestart(void);
```

22.2.3 清除中断

使用 `TMR_WatchDogClearInt` 清除看门狗的中断。

```
void TMR_WatchDogClearInt(void);
```

22.2.4 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` 才能将其禁用。

```
void TMR_WatchDogDisable(void);
```

22.2.5 暂停看门狗

使用 `TMR_WatchDogPauseEnable()` 可以暂停看门狗。

```
void TMR_WatchDogPauseEnable(  
    uint8_t enable  
);
```

22.2.6 处理中断状态

当调用 `TMR_WatchDogEnable` 或者在 `TMR_WatchDogEnable3` 的参数中使能中断，就会在 WDT 上产生中断，此时需要调用 `TMR_WatchDogClearInt` 清除看门狗中断之后才能再次触发中断。

第二十三章 内置 Flash (EFlash)

23.1 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，每个扇区大小为 EFLASH_SECTOR_SIZE 字节；写入时以 32bit 为单位。

内置 Flash 的电源由单独的 LDO 提供，相关配置请参考内置 Flash 电压。不同的芯片系列使用不同的电压供电时，所能支持的最高时钟频率也不相同，见表 23.1。注意实际频率不可超过最高频率。

表 23.1: 部分芯片系列 Flash 电压与最高时钟频率

芯片系列	电压范围 (V)	最高时钟频率 (MHz)
ING9168XX	最低 ~ 2.500	170
ING9168XX	2.600 ~ 3.100	192



表中的 Flash 时钟频率可通过 `SYSCTRL_GetFlashClk()` 读取。Flash 内部会对这个时钟 2 分频，实际工作频率是这个时钟频率的一半。

关于内置 Flash LDO 输出电压的配置建议：

1. 当 V_{BAT} 固定，且 $V_{BAT} \geq 2.8V$ 时，可以按照芯片数据手册给出的 Flash 颗粒的要求配置 Flash LDO 输出电压为 2.1V 或 2.6V；

当输出电压配置为 2.6V 时，可将 Flash 时钟频率配置为 192MHz 以获得最佳性能。

2. 当 V_{BAT} 固定，且 $2.3V \leq V_{BAT} < 2.8V$ 时，配置 Flash LDO 输出电压为 2.1V；

3. 当 V_{BAT} 固定, 且 $V_{BAT} < 2.3V$ 时, 仍可以配置 Flash LDO 输出电压为 $2.1V$, 这时当 $V_{BAT} < 2.1V$ 时, Flash LDO 输出电压与 V_{BAT} 一致, 可保证 Flash 仍有供电;
4. 当 V_{BAT} 不固定, 如 V_{BAT} 来自电池输出时, 推荐配置 Flash LDO 输出电压为 $2.1V$ 。

23.2 使用说明

23.2.1 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。

23.2.2 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。

23.2.3 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。

23.2.4 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。