



# ING918XX 系列芯片外设开发者手册

---

Ingchips Technology Co., Ltd.



# 目录

第一章 版本历史	1
第二章 概览	3
2.1 缩略语及术语	3
第三章 模数转换器 (ADC)	5
3.1 ADC 工作模式	5
3.1.1 单次模式: ADC_MODE_SINGLE	5
3.1.2 循环模式: ADC_MODE_LOOP	5
3.2 ADC 采样数率	6
3.3 ADC 校准	6
3.3.1 ADC 校准预处理	6
3.3.2 ADC 读取校准后的值	7
3.4 ADC 综合示例	7
第四章 嵌入式闪存 (Embedded Flash)	11
4.1 eFlash 写入数据	11
4.1.1 eFlash 先擦除然后写入数据	11
4.1.2 eFlash 不擦除直接写入数据	12
第五章 通用输入输出 (GPIO)	13
5.1 功能概述	13

5.2	使用说明 . . . . .	13
5.2.1	设置 IO 方向 . . . . .	13
5.2.2	读取输入 . . . . .	14
5.2.3	设置输出 . . . . .	14
5.2.4	配置中断请求 . . . . .	15
5.2.5	处理中断状态 . . . . .	16
<b>第六章</b>	<b>I2C 功能概述</b>	<b>17</b>
6.1	I2C 使用说明 . . . . .	17
6.2	使用方法 . . . . .	17
6.2.1	Master 读些, 采用 QUEUE 模式 . . . . .	17
6.2.1.1	打开 I2C 时钟 . . . . .	17
6.2.1.2	配置 I2C 的 IO 口 . . . . .	18
6.2.1.3	I2C 模块初始化 . . . . .	18
6.2.1.4	I2C 写操作 . . . . .	18
6.2.1.5	I2C 读操作 . . . . .	19
<b>第七章</b>	<b>管脚管理 (PINCTRL)</b>	<b>23</b>
7.1	功能概述 . . . . .	23
7.2	使用说明 . . . . .	24
7.2.1	为外设配置 IO 管脚 . . . . .	24
7.2.2	配置下拉、下拉 . . . . .	25
7.2.3	配置驱动能力 . . . . .	25
7.2.4	配置速率 . . . . .	25
7.2.5	IO 口 PWM 参考代码 . . . . .	26
7.2.6	IO 口配置为 UART 参考代码 . . . . .	26
7.2.7	IO 口配置为 SPI 参考代码 . . . . .	26
7.2.8	IO 口配置为 I2C 参考代码 . . . . .	27

<b>第八章 脉宽调制发生器 (PWM)</b>	<b>29</b>
8.1 PWM 工作模式 . . . . .	29
8.1.1 最简单的模式: UP_WITHOUT_DIED_ZONE . . . . .	30
8.1.2 UP_WITH_DIED_ZONE . . . . .	30
8.1.3 UPDOWN_WITHOUT_DIED_ZONE . . . . .	30
8.1.4 UPDOWN_WITH_DIED_ZONE . . . . .	31
8.1.5 SINGLE_WITHOUT_DIED_ZONE . . . . .	31
8.1.6 输出控制 . . . . .	31
8.2 PWM 使用说明 . . . . .	32
8.2.1 启动与停止 . . . . .	32
8.2.2 配置工作模式 . . . . .	32
8.2.3 配置门限 . . . . .	33
8.2.4 输出控制 . . . . .	33
8.2.5 综合示例 . . . . .	34
<b>第九章 RTC</b>	<b>37</b>
9.1 功能概述 . . . . .	37
9.2 接口说明 . . . . .	37
9.2.1 RTC 使能/禁止 . . . . .	37
9.2.2 获取 RTC 当前值 . . . . .	37
9.2.3 设置中断的计数值 . . . . .	38
9.2.4 清 RTC 中断 . . . . .	38
9.3 RTC 中断使用流程 . . . . .	38
<b>第十章 系统控制 (SYSCTRL)</b>	<b>41</b>
10.1 功能概述 . . . . .	41
10.1.1 外设标识 . . . . .	41
10.1.2 时钟树 . . . . .	42

10.2 使用说明 . . . . .	43
10.2.1 外设复位 . . . . .	43
10.2.2 时钟门控 . . . . .	43
10.2.3 时钟配置 . . . . .	43
<b>第十一章 Timer&amp;Watchdog</b>	<b>45</b>
11.1 功能概述 . . . . .	45
11.1.1 计时器功能 . . . . .	45
11.1.2 WATCHDOG 的功能 . . . . .	45
11.2 TIMER 使用说明 . . . . .	46
11.2.1 获取 Timer 计数值 . . . . .	46
11.2.2 TIMER 计数值清零 . . . . .	46
11.2.3 设置 TIMER 的比较值 . . . . .	46
11.2.4 获取 TIMER 的比较值 . . . . .	46
11.2.5 使能 TIMER . . . . .	47
11.2.6 禁能 TIMER . . . . .	47
11.2.7 设置 TIMER 的工作模式 . . . . .	47
11.2.8 使能 TIMER 中断 . . . . .	47
11.2.9 禁能 TIMER 中断 . . . . .	48
11.2.10 清除 TIMER 中断请求 . . . . .	48
11.2.11 获得 TIMER 的中断状态 . . . . .	48
11.3 TIMER 中断使用流程 . . . . .	48
11.4 Watchdog 使用说明 . . . . .	49
11.4.1 使能看门狗 . . . . .	49
11.4.2 停用看门狗 . . . . .	49
11.4.3 喂狗 . . . . .	50

<b>第十二章 通用异步收发传输器 (UART)</b>	<b>51</b>
12.1 功能概述 . . . . .	51
12.2 使用说明 . . . . .	51
12.2.1 设置波特率 . . . . .	51
12.2.2 获取波特率 . . . . .	52
12.2.3 接收错误查询 . . . . .	52
12.2.4 FIFO 轮询模式 . . . . .	52
12.2.5 发送数据 . . . . .	53
12.2.6 接收数据 . . . . .	53
12.2.7 配置中断请求 . . . . .	54
12.2.8 处理中断状态 . . . . .	55
12.2.9 UART 初始化 . . . . .	55
12.2.10 发送数据 . . . . .	57
12.2.11 接收数据 . . . . .	57
12.2.12 清空 FIFO . . . . .	57
12.2.13 使能 FIFO . . . . .	57
12.2.14 处理中断状态 . . . . .	58





# 表格

2.1	缩略语 . . . . .	3
7.1	支持与常用 IO 全映射的常用功能管脚 . . . . .	23
7.2	其它外设功能管脚的映射关系 . . . . .	24



# 插图



# 第一章 版本历史

版本	信息	日期
0.1	初始版本	2022-xx-xx



## 第二章 概览

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

ING918XX 系列芯片支持蓝牙 5.0/5.1 规范，内置高性能 32bit RISC MCU、Flash，以及丰富的外设、高性能低功耗 BLE RF 收发机。BLE 发射功率。

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

- PINCTRL 对应于数据手册之 IOMUX
- 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。

918xx 系列分为 9187 和 9188 两个子系列，9187 和 9188 各自根据封装不同而有不同的后缀。9187 与 9188 的区别在于 9188 是 5.1 规范，而 9187 是 5.0 规范。对于软件开发者来说，如果需要 5.1 规范则需要将 sdk 运行于 9188 系列的芯片上，如开发 AOA 等功能。

总而言之，9188 向下兼容 9187，两者共用一套 sdk，开放给开发者的寄存器及接口均是一致的。

### 2.1 缩略语及术语

表 2.1: 缩略语

缩略语	说明
ADC	模数转换器（Analog-to-Digital Converter）
FIFO	先进先出队列（First In First Out）
GPIO	通用输入输出（General-Purpose Input/Output）

---

缩略语	说明
I2C	集成电路间总线（Inter-Integrated Circuit）
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）

---

## 参考文档

---

1. Bluetooth SIG<sup>1</sup>
2. ING918XX 系列芯片数据手册

---

<sup>1</sup><https://www.bluetooth.com/>



## 第三章 模数转换器（ADC）

ADC 特性：

- 最多有 5 个通道 AIN0~AIN4
- 10 位分辨率
- 输入范围 0~3.6V
- 支持采样率 3/16MHz、3/32MHz、3/64MHz、3/128MHz
- 支持单端输入转换模式
- 支持环路转换模式，每个通道可以启用或禁用

### 3.1 ADC 工作模式

ADC 有两种工作模式：单次模式和循环模式。

```
#define ADC_MODE_SINGLE    1
#define ADC_MODE_LOOP      0
void ADC_SetMode(const uint8_t mode);
```

#### 3.1.1 单次模式：ADC\_MODE\_SINGLE

单次模式只能使能一个通道，如果使能多个通道，只有第一个通道为有效通道，完成后 ADC 将停止，数据可用，知道重置 ADC 控制器。

#### 3.1.2 循环模式：ADC\_MODE\_LOOP

循环模式需要配置一个控制环路延迟：

```
void ADC_SetLoopDelay(const uint32_t delay);
```

一旦 ADC 任务开始，它将逐个从启用的通道中采样输入电压。完成一个循环后，循环延迟将启动停止 ADC，直到延迟时间结束，然后下一个循环将开始。数据每个循环都会更新。

## 3.2 ADC 采样数率

ADC 支持四种采样率 3/16MHz、3/32MHz、3/64MHz、3/128MHz。

```
#define ADC_CLK_16      0
#define ADC_CLK_32      1
#define ADC_CLK_64      2
#define ADC_CLK_128     3
#define ADC_CLK_EN      0x4
/** \brief Set ADC clock selection
 * Note that: remeber to include ADC_CLK_EN.
 */
void ADC_SetClkSel(const uint8_t clk_sel);
```

## 3.3 ADC 校准

如果 ADC 采样计算得到的结果和实际值偏差比较大，可以对 ADC 校准，使用校准后的值计算。

### 3.3.1 ADC 校准预处理

```
enum adc_cali_method
{
    ADC_CALI_METHOD_1 = 1,    // calibration is done in factory
    ADC_CALI_METHOD_2,        // calibration is done in factory
}
```

```

        ADC_CALI_METHOD_SELF           // self-calibration
};
enum adc_cali_method adc_prepare_calibration(void);

```

在主程序内预处理一次即可，该函数会判断当前芯片是否经过校准，如果未经校准会自动校准并将校准后的数据保存到 flash 里；如果已经校准则跳过校准过程。

### 3.3.2 ADC 读取校准后的值

```

enum adc_sample_mode
{
    ADC_SAMPLE_MODE_SLOW,           // for ADC_CLK_128
    ADC_SAMPLE_MODE_FAST,           // for ADC_CLK_16
};
uint16_t adc_calibrate(enum adc_sample_mode mode, uint8_t channel_id,
                        uint16_t value);

```

value 是 adc 采样的原始值，该函数查表计算后，返回校准后的值。

## 3.4 ADC 综合示例

读取 ADC 数据：

```

uint16_t read_adc(uint8_t channel)
{
    SYSCTRL_WaitForLDO();

    ADC_Reset();
    ADC_PowerCtrl(1);

    ADC_SetClkSel(ADC_CLK_EN | ADC_CLK_128);
}

```

```
ADC_SetMode(ADC_MODE_LOOP);
ADC_EnableChannel(channel == 0 ? 1 : 0, 1);
ADC_EnableChannel(channel, 1);
ADC_Enable(1);

while (ADC_IsChannelDataValid(channel) == 0) ;
uint16_t voltage = ADC_ReadChannelData(channel);

ADC_ClearChannelDataValid(channel);
while (ADC_IsChannelDataValid(channel) == 0) ;
voltage = ADC_ReadChannelData(channel);

ADC_Enable(0);
ADC_PowerCtrl(0);

return adc_calibrate(ADC_SAMPLE_MODE_SLOW, channel, voltage);
}
```

主函数:

```
int app_main()
{

    setup_peripherals();

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

    platform_set_evt_callback(PLATFORM_CB_EVT_PROFILE_INIT, setup_profile, NULL);

    platform_set_irq_callback(PLATFORM_CB_IRQ_TIMER1, timer_isr, NULL);

    adc_prepare_calibration();
}
```

```
printf("U = %d\n", read_adc(ADC_CHANNEL));  
  
return 0;  
}
```



## 第四章 嵌入式闪存（Embedded Flash）

闪存（eFlash）分为两个存储块，一个是主存储器块，另一个是信息块。主存储器块大小 512KB，主要用于存放可执行代码。信息块大小为 16KB，可用于存储设备信息。闪存支持页擦除，页擦除操作将擦除一页中的所有字节。

eFlash 特性：

- 0.81V~0.99V/1.62V~3.63V 双电源
- 内存结构 512KB+16KB
- 页面擦除能力：每页 8KB
- 耐久性：10000 次循环（min）
- 125 摄氏度时数据保留期超过 10 年
- 快速页面擦除/程序操作
- 页面擦除时间：20 毫秒（最长）

### 4.1 eFlash 写入数据

写入数据字节数（size）必须是四字节对齐的（二进制表示的低 2 位为 0）。

注意写入数据不要越界（eFlash 用户可操作的地址空间为 0x4000~0x83FFF）。

#### 4.1.1 eFlash 先擦除然后写入数据

```
int program_flash(const uint32_t dest_addr, const uint8_t *buffer, uint32_t size);
```

eFlash 写入操作只能把 1 写成 0，不能把 0 写成 1，所以如果 eFlash 某地址之前保存过其他数据，想要写入数据需要把当前页擦除（当前页的所有字节都变为 0xFF），再写入数据。

写入的目的地址（dest\_addr）必须是页对齐的（地址落在每页的起始地址）。

### 4.1.2 eFlash 不擦除直接写入数据

```
int write_flash(const uint32_t dest_addr, const uint8_t *buffer, uint32_t size);
```

如果可以确认 eFlash 写入的目标地址当前数据都是 0xFF，可以不擦除直接写入数据。



# 第五章 通用输入输出（GPIO）

## 5.1 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。ING918XX 系列芯片内部支持最多 20 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）

## 5.2 使用说明

### 5.2.1 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 `GPIO_SetDirection` 配置 GPIO 的方向。GPIO 支持四种方向：

```
typedef enum
{
    GPIO_DIR_INPUT,    // 输入
    GPIO_DIR_OUTPUT,   // 输出
    GPIO_DIR_BOTH,     // 同时支持输入、输出
    GPIO_DIR_NONE      // 高阻态
} GPIO_Direction_t;
```



如无必要，不要使用 `GPIO_DIR_BOTH`。

### 5.2.2 读取输入

使用 `GPIO_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 = GPIO_ReadValue(GPIO_GPIO_0);
```

使用 `GPIO_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 GPIO_ReadAll(void);
```

### 5.2.3 设置输出

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

```
GPIO_WriteValue(GPIO_GPIO_0, 1);
```

### 5.2.4 配置中断请求

使用 `GPIO_ConfigIntSource` 配置 GPIO 生成中断请求。

```
void GPIO_ConfigIntSource(  
    const GPIO_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 GPIO_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  
} GPIO_IntTriggerEnable_t;
```

触发类型有两种：

```
typedef enum  
{  
    GPIO_INT_EDGE,    // 边沿触发  
    GPIO_INT_LOGIC    // 电平触发  
} GPIO_IntTriggerType_t;
```

- 例如将 GPIO 0 配置为上升沿触发中断

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

### 5.2.5 处理中断状态

在用 `platform_set_irq_callback` 注册好 GPIO 中断回调函数后,在中断里用 `GPIO_GetIntStatus` 可获取某个 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。

## 第六章 I2C 功能概述

- 两个 I2C 模块
- 支持 Master/Slave 模式
- 支持 7bit/10bit 地址
- 支持 DMA 和 QUEUE 模式

### 6.1 I2C 使用说明

以下场景中均以 I2C0 为例，如果需要 I2C1 则可以根据情况修改

### 6.2 使用方法

#### 6.2.1 Master 读些，采用 QUEUE 模式

```
#define I2C_PORT      I2C_PORT_0
#define I2C_ADDR      0X76
```

##### 6.2.1.1 打开 I2C 时钟

```
SYSCTRL_ClearClkGateMulti( (1 << SYSCTRL_ClkGate_APB_I2C0)
                             |(1 << SYSCTRL_ClkGate_APB_PinCtrl));
```

### 6.2.1.2 配置 I2C 的 IO 口

```
PINCTRL_SetPadMux(10, IO_SOURCE_I2C0_SCL_0);
PINCTRL_SetPadMux(11, IO_SOURCE_I2C0_SD0);
PINCTRL_SelI2cSclIn(I2C_PORT, 10);
```

### 6.2.1.3 I2C 模块初始化

```
I2C_CTRL0_CLR(I2C_BASE(I2C_PORT), I2C_CTRL0_SFTRST | I2C_CTRL0_CLKGATE);
```

### 6.2.1.4 I2C 写操作

```
int i2c_do_write(const i2c_port_t port, const uint32_t nrm, uint8_t addr, const uint8_t *
{
    uint32_t *p_data = (uint32_t *)(byte_data + 3);
    uint32_t data = (addr << 1) | 0;    // control: write
    I2C_TypeDef *BASE = I2C_BASE(port);
    int timeout = I2C_HW_TIME_OUT;

    if (length > 0)
        data |= (byte_data[0] << 8) | (byte_data[1] << 16) | (byte_data[2] << 24);

    I2C_CTRL0_CLR(BASE, I2C_CTRL0_SFTRST | I2C_CTRL0_CLKGATE);

    // ONLY SUPPORT PIO QUEUE MODE, SET HW_I2C_QUEUECTRL_PIO_QUEUE_MODE AT FRIST
    I2C_QUEUECTRL_SET(BASE, I2C_QUEUECTRL_PIO_QUEUE_MODE);

    // frist operation, do not need clear I2C_QUEUECTRL and I2C_QUEUECMD.
    BASE->I2C_QUEUECMD.NRM = nrm + 1 + length;
```

```

I2C_QUEUECTRL_SET(BASE, I2C_QUEUECTRL_QUEUE_RUN);

length += 1;
while (1)
{
    while_with_timeout(I2C_QUEUESTAT_WR_QUEUE_FULL(BASE));
    BASE->I2C_DATA = data;
    length -= 4;
    if (length <= 0)
        break;
    data = *p_data;
    p_data++;
}

// WAIT I2C_CTRL1_DATA_ENGINE_CMPLT_IRQ (software polling)
while_with_timeout(GET_I2C_CTRL1_DATA_ENGINE_CMPLT_IRQ(BASE) == 0);
I2C_CTRL1_CLR(BASE, I2C_CTRL1_DATA_ENGINE_CMPLT_IRQ);

// NOTE : MUST SET I2C_QUEUECTRL_WR_CLEAR
I2C_QUEUECTRL_SET(BASE, I2C_QUEUECTRL_WR_CLEAR);
I2C_QUEUECTRL_CLR(BASE, I2C_QUEUECTRL_WR_CLEAR);

return 0;
}

```

### 6.2.1.5 I2C 读操作

```

int i2c_read(const i2c_port_t port, uint8_t addr,
            const uint8_t *write_data, int16_t write_len,
            uint8_t *read_data, int16_t read_length)
{

```

20



```
//  
// STEP 3 : read data byte + (NO ACK) + STOP  
//  
  
BASE->I2C_QUEUECMD.NRM = (I2C_QUEUECMD_SEND_NAK_ON_LAST | I2C_QUEUECMD_POST_SEND_  
    /*I2C_QUEUECMD_XFER_COUNT*/  
    read_length);  
  
I2C_QUEUECTRL_SET(BASE, I2C_QUEUECTRL_QUEUE_RUN);  
  
// Receive DATA use I2C_QUEUEDATA;  
while (read_length > 0)  
{  
    // check whether rdFIFO is empty  
    while_with_timeout(I2C_QUEUESTAT_RD_QUEUE_EMPTY(BASE));  
  
    int len = write_bytes(read_data, BASE->I2C_QUEUEDATA, read_length);  
    read_data += len;  
    read_length -= len;  
}  
  
// WAIT I2C_CTRL1_DATA_ENGINE_CMPLT_IRQ (software polling)  
while_with_timeout(GET_I2C_CTRL1_DATA_ENGINE_CMPLT_IRQ(BASE) == 0);  
  
// cLEAR I2C_CTRL1_DATA_ENGINE_CMPLT_IRQ  
I2C_CTRL1_CLR(BASE, I2C_CTRL1_DATA_ENGINE_CMPLT_IRQ);  
  
// NOTE : CLEAR I2C_QUEUECTRL_RD_CLEAR  
I2C_QUEUECTRL_SET(BASE, I2C_QUEUECTRL_RD_CLEAR);  
I2C_QUEUECTRL_CLR(BASE, I2C_QUEUECTRL_RD_CLEAR);
```

```
return 0;  
}
```

# 第七章 管脚管理（PINCTRL）

## 7.1 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。

IO 管脚特性如下：

- 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 – 19} 这 20 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 7.1。表 7.2 列出了其它外设功能管脚支持映射到哪些 IO 管脚上。

表 7.1: 支持与常用 IO 全映射的常用功能管脚

外设	功能管脚
I2C0	I2C0_SCL_O, I2C0_SDO
I2C1	I2C1_SCL_O, I2C1_SDO
SPI0	SPI0_CLK, SPI0_DO, SPI0_SSN
SPI1	SPI1_CLK, SPI1_DO, SPI1_SSN
UART0	UART0_TXD, UART0_RTS
UART1	UART1_TXD, UART1_RTS

表 7.2: 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
PWM_0A	0-11
PWM_0B	0-11
PWM_1A	0-11
PWM_1B	0-11
PWM_2A	0-11
PWM_2B	0-11
PWM_3A	0-11
PWM_3B	0-11
PWM_4A	0-11
PWM_4B	0-11
PWM_5A	0-11
PWM_5B	0-11

## 7.2 使用说明

### 7.2.1 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 7.1 和表 7.2 确认硬件是否支持。对于不支持的配置，显然无法生效。

```
void PINCTRL_SetPadMux(  
    const uint8_t io_pin_index, // IO 序号 (0..19)  
    const io_source_t source    // IO 源  
);
```

#### 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。比如对于 UART 用于硬件流控的 RXD，需要通过 `PINCTRL_SelUartRxdIn` 配置：

```
void PINCTRL_SelUartRxdIn(  
    const uart_port_t port, //UART 序号  
    const uint8_t io_pin_index)//连接到 RXD 输入的 IO 管脚
```

### 7.2.2 配置下拉、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  
);
```

### 7.2.3 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);
```

### 7.2.4 配置速率

```
void PINCTRL_SetSlewRate(  
    const uint8_t io_pin_index, //IO 管脚序号  
    const pinctrl_slew_rate_t rate);
```

### 7.2.5 IO 口 PWM 参考代码

```
#define LED1_PIN 10 //在 GPIO10 上输出
#define LED1_PWM_CH 4 //映射到 PWM_4
#define LED_FREQ 4000 //频率 4K
SYSCTRL_ClearClkGateMulti((1 << SYSCTRL_ClkGate_APB_PWM)); //打开 PWM 时钟域
PINCTRL_SetGeneralPadMode(LED1_PIN, IO_MODE_PWM, , LED1_PWM_CH, 0); //反向输出
PWM_SetupSimple(LED1_PWM_CH, LED_FREQ, 10);
PWM_Enable(LED1_PWM_CH, 1);
```

### 7.2.6 IO 口配置为 UART 参考代码

```
#define PIN_COMM_RX GIO_GPIO_8
#define PIN_COMM_TX GIO_GPIO_7
SYSCTRL_ClearClkGateMulti((1 << SYSCTRL_ClkGate_APB_UART1));
config_uart(OSC_CLK_FREQ, 921600);

PINCTRL_SetPadMux(PIN_COMM_RX, IO_SOURCE_GENERAL);
PINCTRL_SelUartRxdIn(UART_PORT_1, PIN_COMM_RX);
PINCTRL_SetPadMux(PIN_COMM_TX, IO_SOURCE_UART1_TXD);
```

### 7.2.7 IO 口配置为 SPI 参考代码

```
{
#define SPI_MIC_CLK      GIO_GPIO_13
#define SPI_MIC_MOSI     GIO_GPIO_16
#define SPI_MIC_MISO     GIO_GPIO_17
#define SPI_MIC_CS       GIO_GPIO_8
```

```
SYSCTRL_ClearClkGateMulti((1 << SYSCTRL_ClkGate_AHB_SPI0)
                          | (1 << SYSCTRL_ClkGate_APB_PinCtrl)
                          | (1 << SYSCTRL_ClkGate_APB_GPIO));

PINCTRL_Pull(SPI_MIC_MOSI, PINCTRL_PULL_DOWN);
PINCTRL_Pull(SPI_MIC_CLK, PINCTRL_PULL_UP);
PINCTRL_Pull(SPI_MIC_CS, PINCTRL_PULL_UP);
PINCTRL_Pull(SPI_MIC_MISO, PINCTRL_PULL_UP);

PINCTRL_SetDriveStrength(SPI_MIC_MOSI, PINCTRL_DRIVE_12mA);
PINCTRL_SetDriveStrength(SPI_MIC_CLK, PINCTRL_DRIVE_12mA);
PINCTRL_SetDriveStrength(SPI_MIC_CS, PINCTRL_DRIVE_12mA);

PINCTRL_SetPadMux(SPI_MIC_MOSI, IO_SOURCE_SPI0_D0);
PINCTRL_SetPadMux(SPI_MIC_CLK, IO_SOURCE_SPI0_CLK);

PINCTRL_SetPadMux(SPI_MIC_CS, IO_SOURCE_SPI0_SSN);
PINCTRL_SelSpiDiIn(SPI_PORT_0, SPI_MIC_MISO);

apSSP_DeviceDisable(AHB_SSP0);
SPI_Init(AHB_SSP0);
}
```

### 7.2.8 IO 口配置为 I2C 参考代码

```
SYSCTRL_ClearClkGateMulti( (1 << SYSCTRL_ClkGate_APB_I2C0)
                          | (1 << SYSCTRL_ClkGate_APB_PinCtrl));
PINCTRL_SetPadMux(10, IO_SOURCE_I2C0_SCL_OUT);
PINCTRL_SetPadMux(11, IO_SOURCE_I2C0_SDA_OUT);
```





## 第八章 脉宽调制发生器（PWM）

PWM 模块实现脉冲宽度调制信号的产生，控制 LED 等外部器件。通过 APB 总线读写寄存器来实现整个过程。ING918x 包括 6 个 PWM 模块，每个模块包含 2 个通道，因此可以使用 12 个 PWM 通道。

PWM 特性：

- 每个通道都可以通过寄存器或 PWM 序列来控制
- 每个通道都可以屏蔽
- 寄存器中定义最多四个占空比序列
- 可以使用多种模式：命令模式、单步模式、对称模式、空白区模式

### 8.1 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        = ...,
} PWM_WorkMode_t;
```

### 8.1.1 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;
}
```

### 8.1.2 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); }
```

### 8.1.3 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;
}
```

### 8.1.4 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);
}
```

### 8.1.5 SINGLE\_WITHOUT\_DIED\_ZONE

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

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



以上伪代码仅用于辅助描述硬件行为，与实际行为可以存在微小差异。

### 8.1.6 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;
}
```

## 8.2 PWM 使用说明

### 8.2.1 启动与停止

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

相关的 API 为：

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

### 8.2.2 配置工作模式

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

### 8.2.3 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。

### 8.2.4 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
);
```

### 8.2.5 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);  
}
```





# 第九章 RTC

## 9.1 功能概述

实时时钟（RTC）是一个独立的 48 位定时器，相较于普通 TIMER，RTC 精度低，同时功耗也会降低，适用于功耗敏感、对于精度要求不高的应用场景。

## 9.2 接口说明

### 9.2.1 RTC 使能/禁止

使用 `RTC_Enable` 使能或者禁止 RTC。

```
void RTC_Enable(const uint8_t flag);
```

函数根据 `flag` 的值来执行操作：

- `flag` 为 0 时，禁止 RTC；
- `flag` 不为 0 时，使能 RTC。

RTC 使能后，只要不进行禁能，就会根据设置的初始计数值进行递减，计数器递减为 0 后发生中断，这时只要设置新的计数值，RTC 就会马上按照新的计数值进行计时。

### 9.2.2 获取 RTC 当前值

可以使用 `RTC_Current` 读取 RTC 低 32 位的当前计数值，用 `RTC_CurrentFull` 读取全部 48 位的当前计数值。

```
uint32_t RTC_Current(void);  
uint64_t RTC_CurrentFull(void);
```

### 9.2.3 设置中断的计数值

通过 `RTC_SetNextIntOffset` 设置 RTC 中断的计数值。

```
void RTC_SetNextIntOffset(const uint32_t offset);
```

注意 RTC 使用的时钟为 32.768K，因此计数值设置为 32768 时，RTC 的中断时间恰好为 1s，可以据此设置中断时间。

### 9.2.4 清 RTC 中断

使用 `RTC_ClearInt` 来清除当前的 RTC 中断状态，以备下一次使用。

```
void RTC_ClearInt(void);
```

## 9.3 RTC 中断使用流程

- 首先，定义一个中断处理函数 `rtc_timer_isr` (名字可以根据需要自己定义)。

```
uint32_t rtc_timer_isr(void *user_data)
{
    RTC_ClearInt(); //清中断状态
    RTC_SetNextIntOffset(32768 * 10); //设置中断计数值
    //这里可以添加用户自己的代码
    return 0;
}
```

可以通过 `user_data` 给中断传递参数，这通常作为一个预留功能，不常用到。

- 然后，通过 `platform_set_irq_callback` 注册中断：

```
platform_set_irq_callback(PLATFORM_CB_IRQ_RTC, rtc_timer_isr, NULL);
```

- 接下来，设置中断时间并使能 RTC，时间到后，中断会触发。

```
RTC_SetNextIntOffset(32768 * 10);  
RTC_Enable(RTC_ENABLED);
```

- 这里注意，在使用 RTC 时，最好先通过 `RTC_SetNextIntOffset` 设置好计数值，再进行使能，这样 RTC 此次计数时间处于可控状态。
- 在 RTC 计数过程中，随时都可以使用 `RTC_SetNextIntOffset` 进行计数值的设置，设置完成后，新值会马上覆盖旧值，RTC 会按照新值开始进行计时。



# 第十章 系统控制（SYSCTRL）

## 10.1 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
- 其它功能

### 10.1.1 外设标识

SYSCTRL 为外设定义了几种不同的标识。最常见的几种标识为：

1.SYSCTRL 的时钟门控

```
typedef enum
{
    SYSCTRL_ClkGate_APB_I2C0    = 4,
    SYSCTRL_ClkGate_APB_SPI1    = 5,
    // ...
    SYSCTRL_ClkGate_APB_I2C1    = 19
} SYSCTRL_ClkGateItem;
```

2.SYSCTRL 的 reset

```
typedef enum
{
    SYSCTRL_Reset_AHB_DMA      = 0,
    SYSCTRL_Reset_AHB_LLE      = 1,
    // ...
    SYSCTRL_Reset_APH_TRNG      = 21
} SYSCTRL_ResetItem;
```

这些标识用于外设的复位、时钟门控等。

### 10.1.2 时钟树

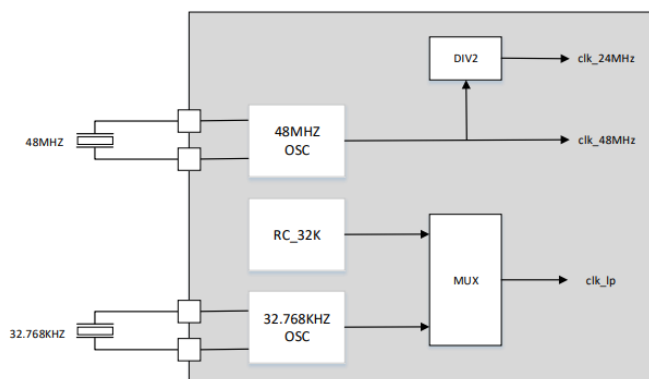
#### 1. 32KiHz 时钟 (*clk\_32k*)

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

#### 2. PLL 输入的 24MHz 时钟 (*clk\_pll\_in*)

24MHz 时钟的主要来源：外部 48MHz 晶体。

#### 3. 时钟分布图



## 10.2 使用说明

### 10.2.1 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);
```

### 10.2.2 时钟门控

通过 `SYSCTRL_SetClkGateMulti` 设置门控（即关闭时钟），通过 `SYSCTRL_ClearClkGateMulti` 消除门控（即恢复时钟）。

```
void SYSCTRL_SetClkGateMulti(SYSCTRL_ClkGateItem item);  
void SYSCTRL_ClearClkGateMulti(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);
```

### 10.2.3 时钟配置

举例如下。

#### 1. 开启 I2C 时钟

使用 `SYSCTRL_SelectI2sClk` 为 I2C 配置时钟：

```
SYSCTRL_ClearClkGateMulti( (1 << SYSCTRL_ClkGate_APB_I2C0)  
                           |(1 << SYSCTRL_ClkGate_APB_PinCtrl));
```

## 2. 关闭定时器 TMR0 时钟

使用 SYSCTRL\_GetClk 可获得指定外设的时钟频率：

```
SYSCTRL_ClearClkGateMulti(0  
                           | (1 << SYSCTRL_ClkGate_APB_TMR0));
```



# 第十一章 Timer&Watchdog

## 11.1 功能概述

ING918xx 系列有三个定时器：Timer0、Timer1 和 Timer2。三个定时器功能基本相同，可以实现定时、比较等功能。唯一的区别在于，Timer0 可以用作看门狗。

### 11.1.1 计时器功能

用作计时器时，主要实现了以下功能：

- 用作 32 位递增计数器或 32 位比较器。
- 可以设置为普通模式、一次性模式和自由模式：
  - 普通模式（TMR\_CTL\_OP\_MODE\_WRAPPING）——计数器以恒定间隔产生中断，在达到比较计数器中的比较值后重置为 0，并继续计数，这是默认采用的模式。
  - 一次性模式（TMR\_CTL\_OP\_MODE\_ONESHOT）——当计数器增长到等于比较器的值时，定时器会禁用，直到下次主动启用它。
  - 自由模式（TMR\_CTL\_OP\_MODE\_FREERUN）——计数器达到定时器中的值时不会停止，而是会一直递加到最大值（0xffffffff），之后重置为零，并继续计数。
- 可以根据设置产生中断。

### 11.1.2 WATCHDOG 的功能

Timer0 可以用作看门狗，设定一定的延时时间，在此时间内，如果程序没有主动喂狗，则会发生重启。

看门狗可以在程序跑飞时，让程序复位。

## 11.2 TIMER 使用说明

### 11.2.1 获取 Timer 计数值

可以通过 TMR\_GetCNT 获取计数器的当前计数值。

```
uint32_t TMR_GetCNT(TMR_TypeDef *pTMR);
```

pTMR：可以设置为 APB\_TMR0、APB\_TMR1、APB\_TMR2, 对应 Timer0、Timer1 和 Timer2。

注意计数器只有使能后，计数值才会随着程序的运行递加，如果计数器未使能，计数值是不变的。

### 11.2.2 TIMER 计数值清零

可以通过 TMR\_Reload 将计数器的当前计数值清零。

```
void TMR_Reload(TMR_TypeDef *pTMR);
```

### 11.2.3 设置 TIMER 的比较值

可以通过 TMR\_SetCMP 设置计数器的比较值。

```
void TMR_SetCMP(TMR_TypeDef *pTMR, uint32_t value);
```

pTMR：选择要设置的计数器，APB\_TMR0、APB\_TMR1 或 APB\_TMR2；

value：设置的比较值。

### 11.2.4 获取 TIMER 的比较值

使用 TMR\_GetCMP 获取计数器的比较值；

```
uint32_t TMR_GetCMP(TMR_TypeDef *pTMR);
```

### 11.2.5 使能 TIMER

通过 TMR\_Enable 使能计数器。

```
void TMR_Enable(TMR_TypeDef *pTMR);
```

计数器使能之后，计数值才会随着时钟的运行递增。

### 11.2.6 禁能 TIMER

通过 TMR\_Disable 禁能计数器。

```
void TMR_Disable(TMR_TypeDef *pTMR);
```

### 11.2.7 设置 TIMER 的工作模式

通过 TMR\_SetOpMode 设置计数器的工作模式。

```
void TMR_SetOpMode(TMR_TypeDef *pTMR, uint8_t mode);
```

三种模式的定义如下：

```
#define TMR_CTL_OP_MODE_WRAPPING      0
#define TMR_CTL_OP_MODE_ONESHOT       1
#define TMR_CTL_OP_MODE_FREERUN       2
```

具体说明见上文。

### 11.2.8 使能 TIMER 中断

通过 TMR\_IntEnable 使能中断，使能中断后计时器计数值达到比较值后，会触发中断。

```
void TMR_IntEnable(TMR_TypeDef *pTMR);
```

### 11.2.9 禁能 TIMER 中断

通过 TMR\_IntDisable 禁能计数器的中断。

```
void TMR_IntDisable(TMR_TypeDef *pTMR);
```

### 11.2.10 清除 TIMER 中断请求

通过 TMR\_IntDisable 清除计数器的中断请求。注意，进入中断处理函数之后，要第一时间清除中断请求，不然可能会重复触发中断。

```
void TMR_IntDisable(TMR_TypeDef *pTMR);
```

### 11.2.11 获得 TIMER 的中断状态

通过 TMR\_IntHappened 来获取计数器的中断状态。

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

## 11.3 TIMER 中断使用流程

- 配置对应 Timer 的时钟，下面的代码中配置了三个 Timer 的时钟，使用时可以根据自己使用的 Timer 进行配置。

```
SYSCTRL_ClearClkGateMulti( (1 << SYSCTRL_ClkGate_APB_TMR0)
                            | (1 << SYSCTRL_ClkGate_APB_TMR1)
                            | (1 << SYSCTRL_ClkGate_APB_TMR2));
```

- 对 Timer 进行初始化，以 Timer1 为例。

```
TMR_SetCMP(APB_TMR1, TMR_CLK_FREQ);
TMR_SetOpMode(APB_TMR1, TMR_CTL_OP_MODE_WRAPPING);
TMR_Reload(APB_TMR1);
TMR_IntEnable(APB_TMR1);
```

上面四条语句分别设置了定时器的比较值，设置工作模式，将定时器的当前计数值清零，并使能中断。

- 注册中断处理函数。

```
platform_set_irq_callback(PLATFORM_CB_IRQ_TIMER1, hr_timer1_isr, NULL);
```

hr\_timer1\_isr 为 Timer1 的中断处理函数。

- 编写中断处理函数。

```
uint32_t hr_timer1_isr(void *user_data)
{
    TMR_IntClr(APB_TMR1);
    //user code
    return 0;
}
```

需要注意，中断处理函数中，要优先清理对应的中断请求。

## 11.4 Watchdog 使用说明

看门狗 (Watchdog) 与 Timer0 共用一套计数器，看门狗没有使能时，Timer0 与 Timer1、Timer2 的使用没有区别，当看门狗使能时，Timer0 就不再起作用。

### 11.4.1 使能看门狗

通过 TMR\_WatchDogEnable 使能看门狗。

```
void TMR_WatchDogEnable(uint32_t timeout);
```

timeout: 设置看门狗的超时时间。

### 11.4.2 停用看门狗

通过 MR\_WatchDogDisable 禁能看门狗。

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

### 11.4.3 喂狗

通过 `TMR_WatchDogRestart` 定期喂狗，如果没有在看门狗的超时之前喂狗，程序会发生重启。这种情况下发生的重启仅能通过硬件重置（`POR` 或者 `RESETN`）清除，在其他复位条件下不会清除，可以帮助启动程序检查最近一次复位发生的原因。

## 第十二章 通用异步收发传输器（UART）

### 12.1 功能概述

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

特性：

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

### 12.2 使用说明

#### 12.2.1 设置波特率

使用 apUART\_BaudRateSet 设置对应 UART 设备的波特率。

```
void apUART_BaudRateSet(  
    UART_TypeDef* pBase,  
    uint32_t ClockFrequency,  
    uint32_t BaudRate  
);
```

### 12.2.2 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  
);
```

### 12.2.3 接收错误查询

使用 apUART\_Check\_Rece\_ERROR 查询接收产生的错误。

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

### 12.2.4 FIFO 轮询模式

在轮询模式下，CPU 通过检查线路状态寄存器中的位来检测事件：

- 使用 apUART\_Check\_Rece\_ERROR 查询接收产生的错误字。

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

- 用 apUART\_Check\_RXFIFO\_EMPTY 查询 RXFIFO 是否为空。

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



- 使用 `apUART_Check_RXFIFO_FULL` 查询 RXFIFO 是否已满。

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

- 使用 `apUART_Check_TXFIFO_EMPTY` 查询 TXFIFO 是否为空。

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

- 使用 `apUART_Check_TXFIFO_FULL` 查询 TXFIFO 是否已满。

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

### 12.2.5 发送数据

使用 `UART_SendData` 发送 8bits 数据。

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

### 12.2.6 接收数据

使用 `UART_ReceData` 接收 8bits 数据。

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

### 12.2.7 配置中断请求

使用 apUART\_Enable\_TRANSMIT\_INT 使能发送中断状态。

```
void apUART_Enable_TRANSMIT_INT(  
    UART_TypeDef* pBase  
);
```

使用 apUART\_Disable\_TRANSMIT\_INT 禁用发送中断状态。

```
void apUART_Disable_TRANSMIT_INT(  
    UART_TypeDef* pBase  
);
```

使用 apUART\_Enable\_RECEIVE\_INT 使能接收中断状态。

```
void apUART_Enable_RECEIVE_INT(  
    UART_TypeDef* pBase  
);
```

使用 apUART\_Disable\_RECEIVE\_INT 禁用接收中断状态。

```
void apUART_Disable_RECEIVE_INT(  
    UART_TypeDef* pBase  
);
```

### 12.2.8 处理中断状态

```
uint8_t apUART_Get_ITStatus(UART_TypeDef* pBase,uint8_t UART_IT); uint32_t apUART_Get_all_raw_i
pBase);
```

```
void apUART_Clr_RECEIVE_INT(UART_TypeDef* pBase); void apUART_Clr_TX_INT(UART_TypeDef*
pBase); void apUART_Clr_NonRx_INT(UART_TypeDef* pBase);
```

### 12.2.9 UART 初始化

两个设备使用 UART 通讯时，必须先约定好传输速率和一些数据位。

```
typedef struct UART_xStateStruct
{
    // Line Control Register, UARTLCR_H
    UART_eWLEN      word_length;    // WLEN
    UART_ePARITY     parity;         // PEN, EPS, SPS
    uint8_t          fifo_enable;    // FEN
    uint8_t          two_stop_bits;  // STP2
    // Control Register, UARTCR
    uint8_t          receive_en;     // RXE
    uint8_t          transmit_en;    // TXE
    uint8_t          UART_en;        // UARTEN
    uint8_t          cts_en;         // CTSEN
    uint8_t          rts_en;         // RTSEN
    // Interrupt FIFO Level Select Register, UARTIFLS
    uint8_t          rxfifo_waterlevel; // RXIFLSEL
    uint8_t          txfifo_waterlevel; // TXIFLSEL
    //UART_eFIFO_WATERLEVEL  rxfifo_waterlevel; // RXIFLSEL
    //UART_eFIFO_WATERLEVEL  txfifo_watchlevel; // TXIFLSEL

    // UART Clock Frequency
    uint32_t         ClockFrequency;
    uint32_t         BaudRate;
```

```
} UART_sStateStruct;
```

定义函数 `config_uart`,

```
void config_uart(  
    uint32_t freq,  
    uint32_t baud  
);
```

在函数中, 对 `UART_sStateStruct` 的各项参数初始化, 并调用 `apUART_Initialize` 对 UART 进行初始化。

```
void config_uart(uint32_t freq, uint32_t baud)  
{  
    UART_sStateStruct config;  
  
    config.word_length      = UART_WLEN_8_BITS;  
    config.parity           = UART_PARITY_NOT_CHECK;  
    config.fifo_enable      = 1;  
    config.two_stop_bits    = 0;  
    config.receive_en       = 1;  
    config.transmit_en      = 1;  
    config.UART_en          = 1;  
    config.cts_en           = 0;  
    config.rts_en           = 0;  
    config.rxfifo_waterlevel = 1;  
    config.txfifo_waterlevel = 1;  
    config.ClockFrequency   = freq;  
    config.BaudRate         = baud;  
  
    apUART_Initialize(PRINT_PORT, &config, 0);  
}
```

### 12.2.10 发送数据

使用 UART\_SendData 发送数据。

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

### 12.2.11 接收数据

使用 UART\_ReceData 接收数据。

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

### 12.2.12 清空 FIFO

使用 uart\_empty\_fifo 清空 UART 的 FIFO。

```
static void uart_empty_fifo(  
    UART_TypeDef* pBase  
);
```

### 12.2.13 使能 FIFO

使用 uart\_enable\_fifo 使能 UART 的 FIFO。

```
static void uart_enable_fifo(  
    UART_TypeDef* pBase  
);
```

### 12.2.14 处理中断状态

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

UART 产生中断后，需要消除中断状态方可再次触发。用 `apUART_Clr_RECEIVE_INT` 消除某个 UART 上接收中断的状态，用 `apUART_Clr_TX_INT` 消除某个 UART 上发送中断的状态。用 `apUART_Clr_NonRx_INT` 消除某个 UART 上除接收以外的中断状态。