TG7120B

Peripheral Application Note

Version 0.2

Author:

Security: Public

Date: 2020.9

版本控制信息

版本/状态	作者	起止日期	备注
V1.0		7/21/2020	文档初稿

目录

1 简介 **3**
2 watchdog **4**
2.1 watchdog 概述 *4*
2.2 watchdog 硬件 *4*
2.3 watchdog 使用 *4*
3 timer **6**
3.1 timer 概述 *6*
3.2 timer 硬件 *6*
3.3 timer 使用 *6*
4 pwm **7**
4.1 pwm 概述 *7*
4.2 pwm 硬件 *7*
4.3 pwm 使用 *9*
5 uart **11**
5.1 uart 概述 *11*
5.2 uart 硬件 *11*
5.3 uart 使用 *12*
6 spi **14**
6.1 spi 概述 *14*
6.2 spi 硬件 *16*
6.3 spi 使用 *17*
7 i2c **19**
7.1 i2c 概述 *19*
7.2 i2c 硬件 *19*
7.3 i2c 使用 *19*
8 kscan ..... 错误!未定义书签。
8.1 kscan 概述 ..... ***错误!未定义书签。***
8.2 kscan 硬件 ..... ***错误!未定义书签。***
8.3 kscan 使用 ..... ***错误!未定义书签。***

图表

图表 1 PWM 示意图 7
图表 2 PWM UP 模式 8
图表 3 PWM UP AND DOWM 模式 8
图表 4 UART 帧格式 11
图表 5 SPI 示意图 14
图表 6 CPOL=0 CPHA=0 模式一 15
图表 7 CPOL=0 CPHA=1 模式二 15
图表 8 CPOL=1 CPHA=0 模式三 16
图表 9 CPOL=1 CPHA=1 模式四 16
图表 10 I2C 示意图 19

1 简介

本文档主要介绍 TG7120B peripheral 的原理和注意事项。

包括 watchdog、timer、pwm、uart、spi、i2c、kscan、gpio、adc 等。

2 watchdog

2.1 watchdog 概述

在单片机构成的微系统中，当代码书写不健壮，或是运行环境受到外界硬件干扰，造成各种寄存器和内存的数据混乱，会导致程序跑飞或是系统陷入死循环。此时程序的正常运行被打断，正常逻辑无法继续执行，导致整个系统的陷入停滞状态，发生不可预料的后果。

为了解决上述问题，看门狗应运而生。看门狗，又叫 watchdog，从本质上来说就是一个定时器电路，一般有一个输入和一个输出，其中输入叫做喂狗，输出一般连接到另外一个部分的复位端。看门狗的功能是定期的查看芯片内部的情况，一旦发生错误就向芯片发出重启信号。

2.2 watchdog 硬件

- watchdog 时钟为 32.768Khz，可以选择 RC 32k 或 XTAL 32k。
- watchdog 喂狗周期可选择 2S、4S、8S、16S、32S、64S、128S、256S。
- watchdog 使用轮询方式。当选择轮询方式时，超过喂狗周期未喂狗，系统将产生复位。当系统休眠时，watchdog 信息会丢失，唤醒后需要重新配置。

2.3 watchdog 使用

package.yaml 文件中使能 CONFIG_WDT 宏定义

```
## 第五部分：配置信息
def_config:
  CONFIG_WDT: 1
##mesh func config
  CONFIG_BT_MESH_GATT_PROXY: 1
  CONFIG_BT_MESH_PB_GATT: 1
  CONFIG_BT_MESH_RELAY: 1

##mesh model config
  CONFIG_MESH_MODEL_CTL_SRV: 1
  CONFIG_MESH_MODEL_GEN_ONOFF_SRV: 1
  CONFIG_MESH_MODEL_LIGHTNESS_SRV: 1
  CONFIG_MESH_MODEL_SCENE_SRV: 1
  CONFIG_MESH_MODEL_TRANS: 1
  CONFIG_MESH_MODEL_VENDOR_SRV: 1

##genie mesh config
  #CONFIG_GENERAL_CLI_CMD: 1
  CONFIG_GENIE_OTA: 1
  CONFIG_GENIE_RESET_BY_REPEAT: 1
```

```
  CONFIG_GENIE_SW_RESET_PROV: 1
  MESH_MODEL_VENDOR_TIMER: 1
  CONFIG_UART_RECV_BUF_SIZE: 128

  #CONFIG_BT_MESH_CTRL_RELAY: 1
  #GENIE_ULTRA_PROV: 1

  CONFIG_BT_DEVICE_NAME: "GenieLight"
  PROJECT_SW_VERSION: 0x00010101

## genie lpm support
  #CONFIG_PM_SLEEP: 1
  #CONFIG_GENIE_MESH_GLP: 1

##debug config
  #CONFIG_LOGMACRO_SILENT: 1
  #CONFIG_BT_DEBUG_LOG: 1
  #CONFIG_DEBUG: 1
  MESH_DEBUG_PROV: 1
  MESH_DEBUG_TX: 1
  MESH_DEBUG_RX: 1
```

3 timer

3.1 timer 概述

timer 是微系统中的标配，为应用提供精准的计时机制。

3.2 timer 硬件

- 系统共有 6 个硬件 timer，其中 4 个已经被协议栈、osal 调度器等软件资源所使用，其余 2 个供应用使用。
- 时钟源固定 4MHz，硬件不可分频。驱动中为了计算方便，软件将其 4 分频。
- 位宽为 32bit，即最大计数值为 0xFFFFFFFF。
- 支持中断方式和非中断方式。
  - 支持 free-running mode 和 user-defined count mode。前者减到 0 后，自动加载 0xFFFFFFFF。后者减到 0 后，自动加载用户预先配置的值。驱动中用的是后者。
- 当系统休眠时，timer 信息会丢失，唤醒后需要重新配置。

3.3 timer 使用

timer 初始化后调用相应 api 即可，如下：

```
// time_init 初始化函数，timer_update 为中断回调函数，内容根据需要实现
static void timer_update(void *args)
{
    //timer_update
}
int time_init(void)
{
    timer_dev_t timer;
    timer.port = 0;
    timer.config.period = 1000 * 1000; //单位: us
    timer.config.reload_mode = TIMER_RELOAD_AUTO;
    timer.config.cb = timer_update;
    timer.config.arg = NULL;
    hal_timer_init(&genie_time_timer.timer);
    hal_timer_start(&genie_time_timer.timer);
}
```

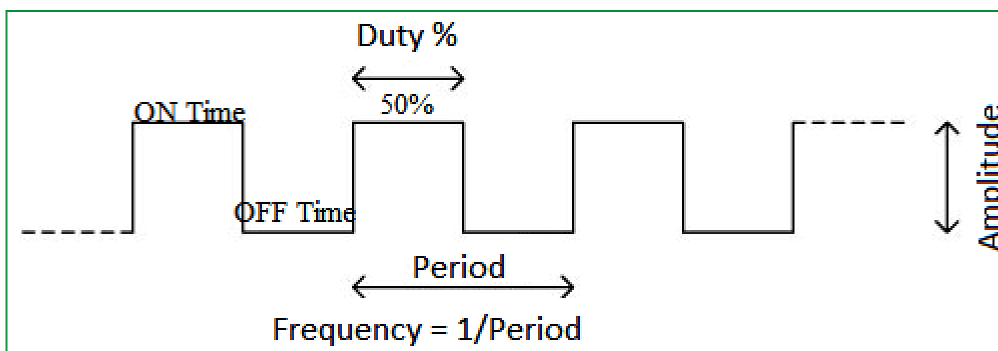
4 pwm

4.1 pwm 概述

PWM 表示脉冲宽度调制，它是一系列脉冲，这些脉冲将以方波的形式出现。也就是说，在任何给定的时间点，波型要么是高电平或者是低电平。

PWM 有两个参数：

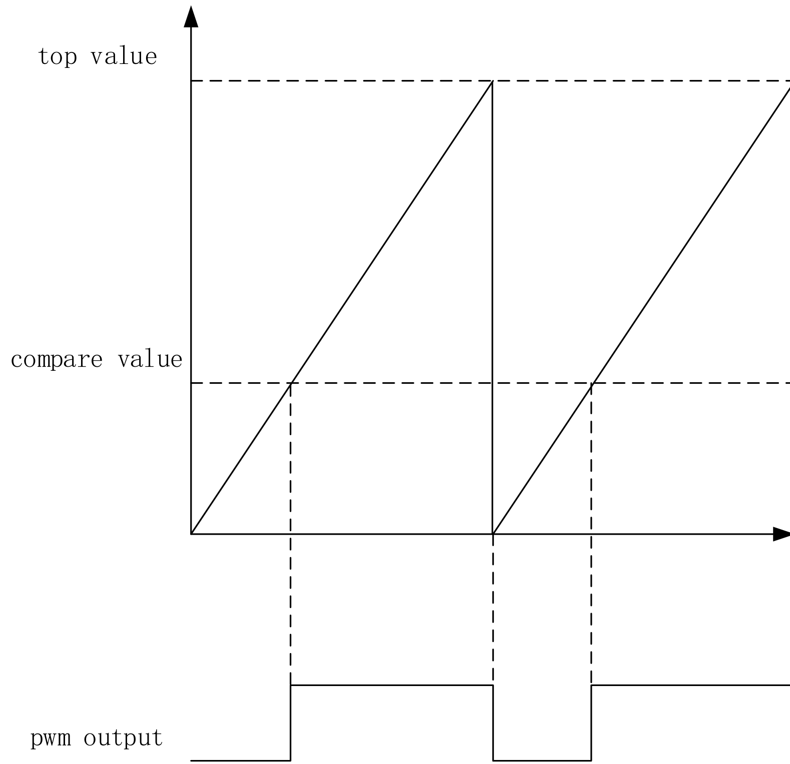
- 占空比 =接通时间 / (接通时间 + 断开时间)
- 频率 = 1 / 总持续时间，总持续时间 = 接通时间 + 断开时间



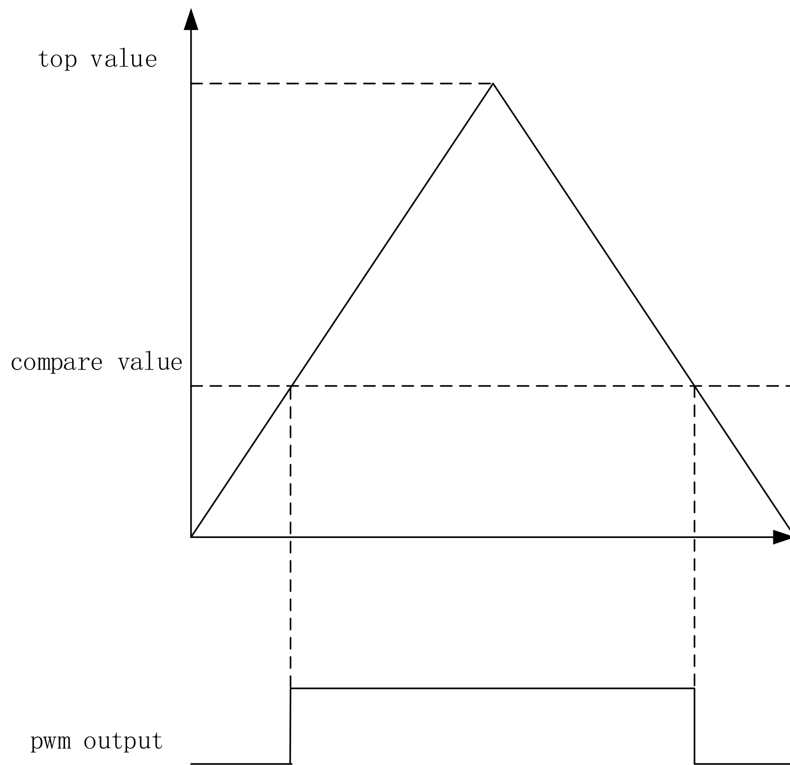
图表 1 PWM 示意图

4.2 pwm 硬件

- 支持 6 路 pwm。
- 时钟源 16Mhz，每路 pwm 支持的分频有 1、2、4、8、16、32、64、128。
- 当系统休眠时，pwm 信息会丢失，唤醒后需要重新配置。
- 所有可 fmux 的 io 都可以复用为 pwm。
- 支持 up 模式和 up and dowm 模式。前者支持占空比 0~100%；后者不支持占空比 0% 和 100%，需要 gpio 输出高低来辅助实现。



图表 2 PWM up 模式



图表 3 PWM up and dowm 模式

4.3 pwm 使用

```
//port 与 IO 功能号需要对应，例如 FMUX_PWM1 对应的 PWM 端口为 PWM_CH1
#include "pin_name.h"
#include "pinmux.h"
#include <aos/hal/pwm.h>
#include <aos/hal/gpio.h>

pwm_dev_t pwm1;

void hal_pwm_test()
{
    int ret = -1;
    pwm_config_t pwm_cfg;
    static int count = 0;

    drv_pinmux_config(P20, FMUX_PWM1);

    /* pwm port set */
    pwm1.port = PWM_CH1;

    /* pwm attr config */
    pwm1.config.duty_cycle = 0.2; /* 1000us */
    pwm1.config.freq       = 500; /* 500hz 2000us */

    /* init pwm1 with the given settings */
    ret = hal_pwm_init(&pwm1);
    if (ret != 0) {
        printf("pwm1 init error !\n");
    }

    /* start pwm1 */
    ret = hal_pwm_start(&pwm1);
    if (ret != 0) {
        printf("pwm1 start error !\n");
    }

    while(1) {

        /* change the duty cycle to 50% */
        if (count == 5) {
            memset(&pwm_cfg, 0, sizeof(pwm_config_t));
            pwm_cfg.duty_cycle = 0.5;/* 500us */
             pwm_cfg.freq = 500;/* 2000us */
            ret = hal_pwm_para_chg(&pwm1, pwm_cfg);
```

```
            if (ret != 0) {
                printf("pwm1 para change error !\n");
            }
        }

        /* change freq to 1000*/
        if (count == 10) {
            memset(&pwm_cfg, 0, sizeof(pwm_config_t));
            pwm_cfg.duty_cycle = 1.0;/* 500us */
            pwm_cfg.freq = 1000;/* 1000us */
            ret = hal_pwm_para_chg(&pwm1, pwm_cfg);
            if (ret != 0) {
                printf("pwm1 para change error !\n");
            }
        }

        /* change duty cycle to 10%, freq 1000*/
        if (count == 15) {
            memset(&pwm_cfg, 0, sizeof(pwm_config_t));
            pwm_cfg.duty_cycle = 0.1;/* 500us */
            pwm_cfg.freq = 1000;/* 1000us */
            ret = hal_pwm_para_chg(&pwm1, pwm_cfg);
            if (ret != 0) {
                printf("pwm1 para change error !\n");
            }
        }

        /* stop and finalize pwm1 */
        if (count == 20) {
            hal_pwm_stop(&pwm1);
            hal_pwm_finalize(&pwm1);
            break;
        }

        /* sleep 1000ms */
        aos_msleep(1000);
        count++;
    }
}
```

5 uart

5.1 uart 概述

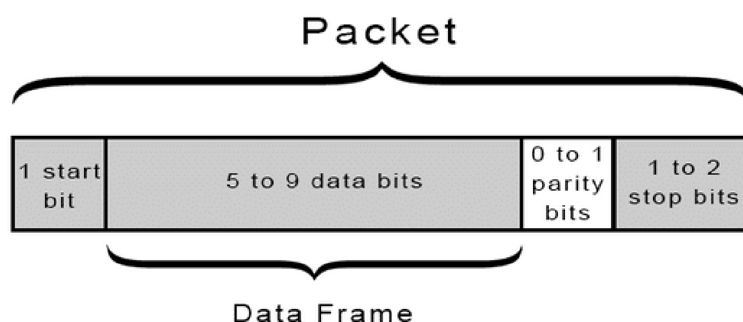
通用异步收发传输器（Universal Asynchronous Receiver/Transmitter)，通常称作 UART。

在 UART 通信中，两个 UART 直接相互通信。发送 UART 将来自 CPU 等控制设备的并行数据转换为串行形式，并将其串行发送到接收 UART，接收 UART 然后将串行数据转换回接收设备的并行数据。在两个 UART 之间传输数据只需要两根线。

UART 以异步方式发送数据，这意味着没有时钟信号将发送 UART 的位输出与接收 UART 的位采样同步。发送 UART 不是时钟信号，而是将开始和停止位添加到正在传输的数据包中。这些位定义数据包的开始和结束，因此接收 UART 知道何时开始读取位。

当接收 UART 检测到起始位时，它开始以称为波特率的特定频率读取输入位。波特率是数据传输速度的度量，以每秒位数（bps）表示。两个 UART 必须以大致相同的波特率运行。

UART 传输的数据被组织成数据包。每个数据包包含 1 个起始位，5 到 9 个数据位（取决于 UART），可选的奇偶校验位以及 1 或 2 个停止位。



图表 4 uart 帧格式

5.2 uart 硬件

- 支持 2 路 uart。
- 时钟等于 hclk，可分频，不建议分频。
- 当系统休眠时，uart 信息会丢失，唤醒后需要重新配置。
- 所有可 fmux 的 io 都可以复用为 uart。
- 系统日志打印默认使用 uart0(p9、p10)
- 假设当前系统主频为 hclk 且不分频，需要的波特率为 baud，实际硬件配置的寄存器为 divisor = (hclk) / (16 * baud)。当丢失的小数部分大于 2%时，此波特率不支持会乱码。比如：系统时钟为 48M，波特率分别为 115200、921600、1000000，

需要配置的寄存器分别是 26.04166666666667、3.2552083333333333、3，误差分别是 0.16%、7.84%、0%，因此此时支持 115200 和 1000000，不支持 921600。

5.3 uart 使用

详见 sdk 相关 demo

```
#include "pin_name.h"
#include "pinmux.h"
#include <aos/hal/uart.h>


#define UART_BUF_SIZE   10
#define UART_TX_TIMEOUT 10
#define UART_RX_TIMEOUT 10

/* define dev */
uart_dev_t uart1;

/* data buffer */
uint8_t uart_data_buf[UART_BUF_SIZE];

void hal_uart_test()
{
    int ret     = -1;
    int i       = 0;
    uint32_t rx_size = 0;

    /* uart pin set */
    drv_pinmux_config(P18, FMUX_UART1_TX);
    drv_pinmux_config(P20, FMUX_UART1_RX);

    /* uart port set */
    uart1.port = 1;

    /* uart attr config */
    uart1.config.baud_rate    = 115200;
    uart1.config.data_width   = DATA_WIDTH_8BIT;
    uart1.config.parity       = NO_PARITY;
    uart1.config.stop_bits    = STOP_BITS_1;
    uart1.config.flow_control = FLOW_CONTROL_DISABLED;
    uart1.config.mode         = MODE_TX_RX;

    /* init uart1 with the given settings */
    ret = hal_uart_init(&uart1);
```

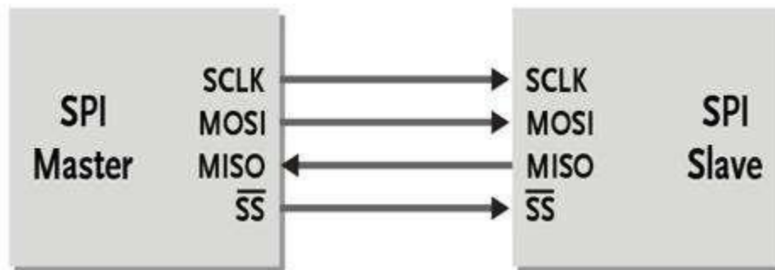
```
    if (ret != 0) {
        printf("uart1 init error !\n");
    }

    /* init the tx buffer */
    for (i = 0; i < UART_BUF_SIZE; i++) {
        uart_data_buf[i] = i + 1;
    }

    /* send 0,1,2,3,4,5,6,7,8,9 by uart1 */
    ret = hal_uart_send(&uart1, uart_data_buf, UART_BUF_SIZE,
UART_TX_TIMEOUT);
    if (ret == 0) {
        printf("uart1 data send succeed !\n");
    }

    /* scan uart1 every 100ms, if data received send it back */
    while(1) {
        ret = hal_uart_recv_II(&uart1, uart_data_buf, UART_BUF_SIZE,
                               &rx_size, UART_RX_TIMEOUT);
        if ((ret == 0) && (rx_size == UART_BUF_SIZE)) {
            printf("uart1 data received succeed !\n");

            ret = hal_uart_send(&uart1, uart_data_buf, rx_size,
UART_TX_TIMEOUT);
            if (ret == 0) {
                printf("uart1 data send succeed !\n");
            }
        }

        /* sleep 100ms */
        aos_msleep(100);
    };
}
```

6 spi

6.1 spi 概述

SPI，是英语 Serial Peripheral interface 的缩写，顾名思义就是串行外围设备接口。是一种同步、全双工、主从式接口。来自主机或从机的数据在时钟上升沿或下降沿同步。主机和从机可以同时传输数据。

在芯片的管脚上只占用四根线，节约了芯片的管脚，同时为 PCB 的布局上节省空间，提供方便，正是出于这种简单易用的特性，现在越来越多的芯片集成了这种通信协议。



图表 5 SPI 示意图

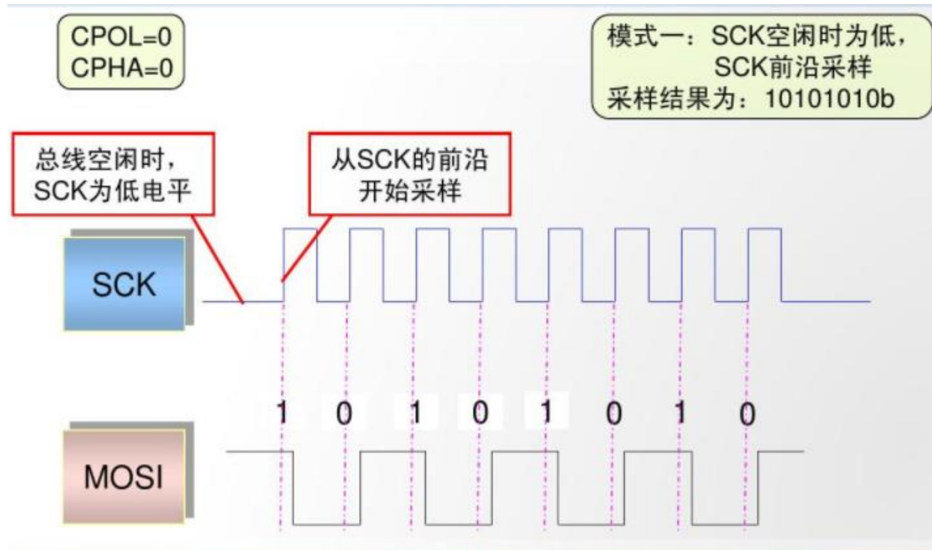
产生时钟信号的器件称为主机。主机和从机之间传输的数据与主机产生的时钟同步。4 线 SPI 器件有四个信号：

- 时钟(SPI CLK, SCLK)
- 片选(CS)
- 主机输出、从机输入(MOSI)
- 主机输入、从机输出(MISO)

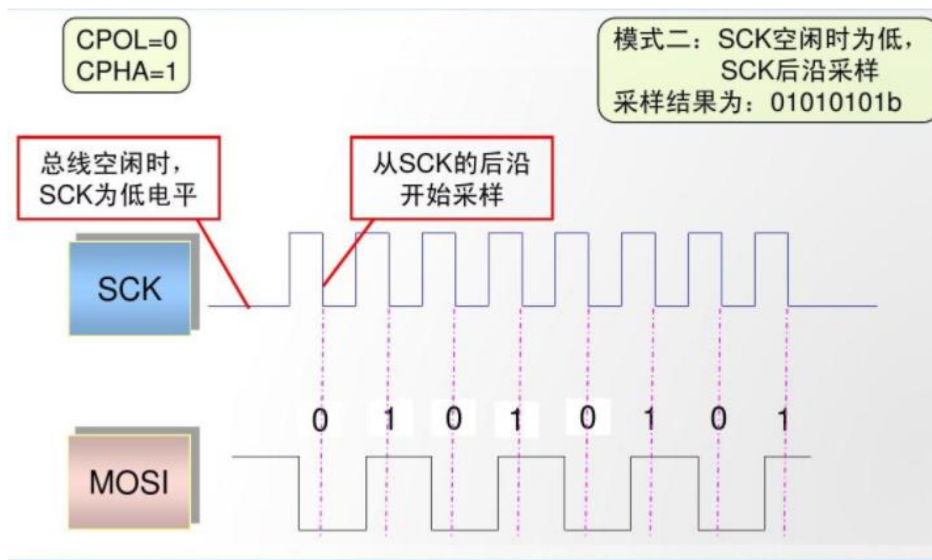
SPI 模块为了和外设进行数据交换，根据外设工作要求，其输出串行同步时钟极性和相位可以进行配置，时钟极性（CPOL）对传输协议没有重大的影响。

CPOL：时钟极性选择，为 0 时 SPI 总线空闲为低电平，为 1 时 SPI 总线空闲为高电平。

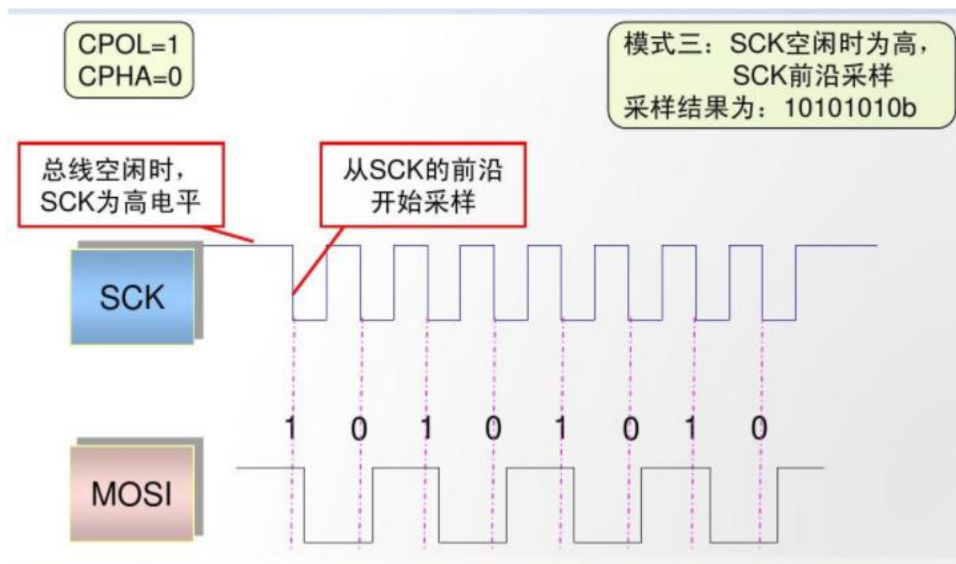
CPHA：时钟相位选择，为 0 时在 SCK 第一个跳变沿采样，为 1 时在 SCK 第二个跳变沿采样。



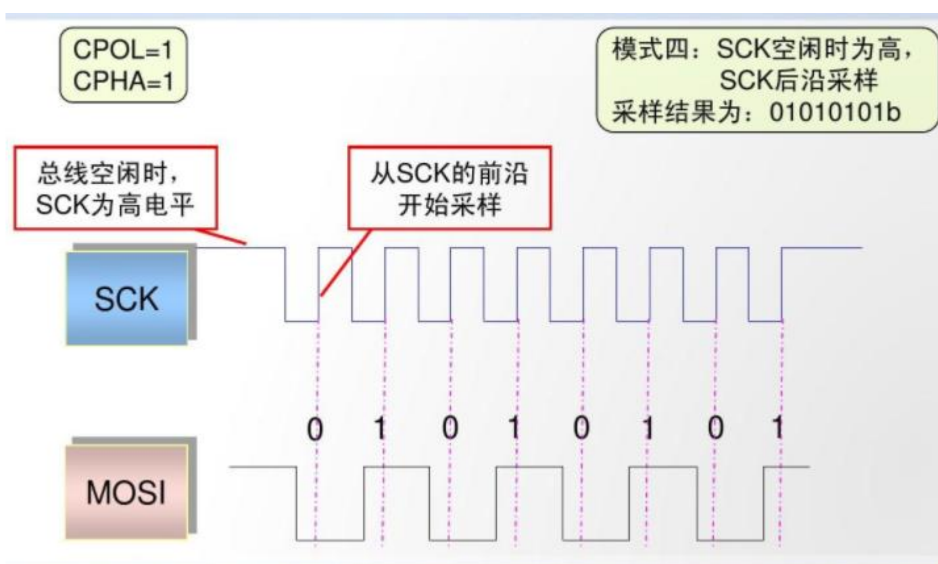
图表 **6** CPOL=0 CPHA=0 模式一



图表 **7** CPOL=0 CPHA=1 模式二



图表 8 CPOL=1 CPHA=0 模式三



图表 9 CPOL=1 CPHA=1 模式四

6.2 spi 硬件

- 支持 2 路 spi，可配置 master 或 slave。
- 时钟等于 hclk，可分频，不建议分频。
- 当系统休眠时，spi 信息会丢失，唤醒后需要重新配置。
- 所有可 fmux 的 io 都可以复用为 spi。
- 当使用 spi 发送数据时，可以选择自动或手动控制 cs 的高低，可以选择是否使用中断。所谓手动是通过将 io 设置为 gpio 并将其输出高低。spi_Cfg_t 中的 force_cs 为 true 时选择手动模式，spi_Cfg_t 中的 int_mode 为 true 时使用中断方式。

6.3 spi 使用

```
//SPI 端口需要与管脚功能对应，例如端口 1 对应功能定义 FMUX_SPI_1_xxx
#include "pin_name.h"
#include "pinmux.h"
#include <aos/hal/spi.h>

#define SPI_BUF_SIZE   10
#define SPI_TX_TIMEOUT 10
#define SPI_RX_TIMEOUT 1000

/* define dev */
spi_dev_t spi1;

/* data buffer */
uint8_t spi_data_buf[SPI_BUF_SIZE];

static void aos_hal_spi_master_send(uint16_t size, uint32_t frequency,
uint32_t time)
{
    int ret1     = -1;
    int ret2     = -1;
    int ret3     = -1;
    int i       = 0;
    int n        = 0;
    uint8_t spi_data_buf[size];

    drv_pinmux_config(P11, FMUX_SPI_1_RX);
    drv_pinmux_config(P18, FMUX_SPI_1_TX);
    drv_pinmux_config(P15, FMUX_SPI_1_SSN);
    drv_pinmux_config(P20, FMUX_SPI_1_SCK);

    spi1.port = 1;
    spi1.config.mode = HAL_SPI_MODE_MASTER;
    spi1.config.freq = frequency;

    ret1 = hal_spi_init(&spi1);
    if (ret1 != 0) {
        printf("spi1 init error !\n");
    }

    for (i = 0; i < size; i++) {
        spi_data_buf[i] = i + 2;
```

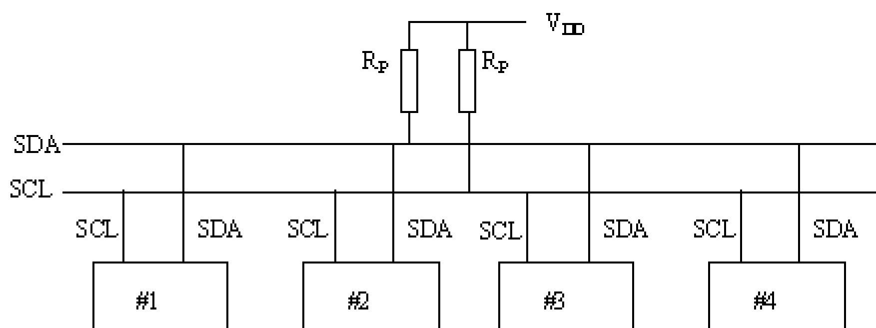
```
    }

    while(n < 10) {
        ret2 = hal_spi_send(&spi1, spi_data_buf, size, time);

        aos_msleep(50);

        for (int i = 0; i < size; i++) {
            printf("master send is %d\n",spi_data_buf[i]);
        }
        if (ret2 == 0) {
            printf("spi1 data send succeed !\n");
        } else { printf("err send\r\n");   }
        n++;
    }

    ret3 = hal_spi_finalize(&spi1);
    if (ret3 != 0) {
        printf("spi1 finalize error !\n");
    }

    if (ret1 == 0 && ret2 == 0 && ret3 == 0) {
        printf("execute OK \r\n");
    }else {
        printf("execute ERR \r\n");
    }
}
```

7 i2c

7.1 i2c 概述

I2C Bus(Inter-Integrated Circuit Bus) 最早是由 Philips 半导体（现被 NXP 收购）开发的两线时串行总线，常用于微控制器与外设之间的连接。

I2C 仅需两根线就可以支持一主多从或者多主连接，I2C 使用两个双向开漏线，配合上拉电阻进行连接。



图表 10 i2c 示意图

7.2 i2c 硬件

- 支持 2 路 i2c。
- 时钟等于 hclk，可分频，不建议分频。
- 当系统休眠时，i2c 信息会丢失，唤醒后需要重新配置。
- 所有可 fmux 的 io 都可以复用为 i2c。
- i2c 使用时需要接上拉电阻，比如 2.2K。

7.3 i2c 使用

详见 sdk 相关 demo

```
#include "pin_name.h"
#include "pinmux.h"
#include <aos/hal/i2c.h>

#define I2C_BUF_SIZE   10
#define I2C_TX_TIMEOUT 10
#define I2C_RX_TIMEOUT 10

#define I2C_DEV_ADDR       0x50
#define I2C_DEV_ADDR_WIDTH I2C_HAL_ADDRESS_WIDTH_7BIT

/* define dev */
```

```
i2c_dev_t i2c1;

/* data buffer */
uint8_t i2c_data_buf[I2C_BUF_SIZE];

int hal_iic_test_master()
{
    int ret    = -1;
    int i      = 0;

    drv_pinmux_config(P18, FMUX_IIC1_SDA);
    drv_pinmux_config(P20, FMUX_IIC1_SCL);

    /* i2c port set */
    i2c1.port = 1;

    /* i2c attr config */
    i2c1.config.mode          = I2C_MODE_MASTER;
    i2c1.config.freq          = 400000;
    i2c1.config.address_width = I2C_DEV_ADDR_WIDTH;
    i2c1.config.dev_addr      = I2C_DEV_ADDR;

    /* init i2c1 with the given settings */
    ret = hal_i2c_init(&i2c1);
    if (ret != 0) {
        printf("i2c1 init error !\n");
    }

    /* init the tx buffer */
    for (i = 0; i < I2C_BUF_SIZE; i++) {
        i2c_data_buf[i] = i + 1;
    }

    /* send 0,1,2,3,4,5,6,7,8,9 by i2c1 */
    ret = hal_i2c_master_send(&i2c1, I2C_DEV_ADDR, i2c_data_buf,
                              I2C_BUF_SIZE, I2C_TX_TIMEOUT);
    if (ret == 0) {
        printf("i2c1 data send succeed !\n");
    }

    memset(i2c_data_buf,0,sizeof(i2c_data_buf));
    ret = hal_i2c_master_recv(&i2c1, I2C_DEV_ADDR, i2c_data_buf,
                              I2C_BUF_SIZE, I2C_RX_TIMEOUT);
    if (ret == 0) {
```

```
        printf("i2c1 data received succeed !\n");
    }

    printf("receive: ");
    for (i = 0; i < I2C_BUF_SIZE; i++) {
        printf("%x \r\n",i2c_data_buf[i]);
    }

    return 0;
}
```