ESPRESSIF

ESP32 Peripherals Overview

Introduction to ESP32 Peripherals

1. Overview
2. Resources
3. System clock
4. IO Matrix
5. Peripheral drivers
6. esp-iot-solution project

ESP32 Peripherals Overview

	ESP8266	ESP32
GPIO	15	up to 34 (IO matrix)
RTCIO	1	up to 10
UART	1.5	3 * HW UART
PWM	SW PWM	16 * LED PWM
I2C	SW I2C	2* HW I2C
I2S	1	2
SPI	1+1	1+1+2
TIMER	32-bit Timer*2	64-bit Timer*4
RMT	N/A	8 channels
PCNT	N/A	8 units
SD/MMC	N/A	1
Motor PWM	N/A	2
RTC Sensor	N/A	Touch sensor/Temperature/Hall/Smoke

ESP32 Peripheral Resources

- **ESP32 TRM**
  Technical reference manual
  https://espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf

- **ESP32 Peripheral API Reference**
  API reference, API guides for ESP32 peripherals
  https://esp-idf.readthedocs.io/en/latest/api-reference/peripherals/index.html

Peripheral Base Address

```
PROVIDE ( UART0 = 0x3ff40000 );
PROVIDE ( SPI1 = 0x3ff42000 );
PROVIDE ( SPI0 = 0x3ff43000 );
PROVIDE ( GPIO = 0x3ff44000 );
PROVIDE ( SIGMADELTA = 0x3ff44f00 );
PROVIDE ( RTCCNTL = 0x3ff48000 );
PROVIDE ( RTCIO = 0x3ff48400 );
PROVIDE ( SENS = 0x3ff48800 );
PROVIDE ( UHCI1 = 0x3ff4C000 );
PROVIDE ( I2S0 = 0x3ff4F000 );
PROVIDE ( UART1 = 0x3ff50000 );
PROVIDE ( I2C0 = 0x3ff53000 );
PROVIDE ( UHCI0 = 0x3ff54000 );
PROVIDE ( RMT = 0x3ff56000 );
PROVIDE ( RMTMEM = 0x3ff56800 );
```

```
PROVIDE ( PCNT = 0x3ff57000 );
PROVIDE ( LEDC = 0x3ff59000 );
PROVIDE ( MCPWM0 = 0x3ff5E000 );
PROVIDE ( TIMERG0 = 0x3ff5F000 );
PROVIDE ( TIMERG1 = 0x3ff60000 );
PROVIDE ( SPI2 = 0x3ff64000 );
PROVIDE ( SPI3 = 0x3ff65000 );
PROVIDE ( SYSCON = 0x3ff66000 );
PROVIDE ( I2C1 = 0x3ff67000 );
PROVIDE ( SDMMC = 0x3ff68000 );
PROVIDE ( MCPWM1 = 0x3ff6C000 );
PROVIDE ( I2S1 = 0x3ff6D000 );
PROVIDE ( UART2 = 0x3ff6E000 );
```

- 每个外设硬件有一个基地址，通过基地址加偏移量来访问外设寄存器。
- ESP32 的外设地址，在 idf/components/esp32/ld/esp32.peripherals.ld 中定义（如上图所示）

Peripheral in ESP-IDF

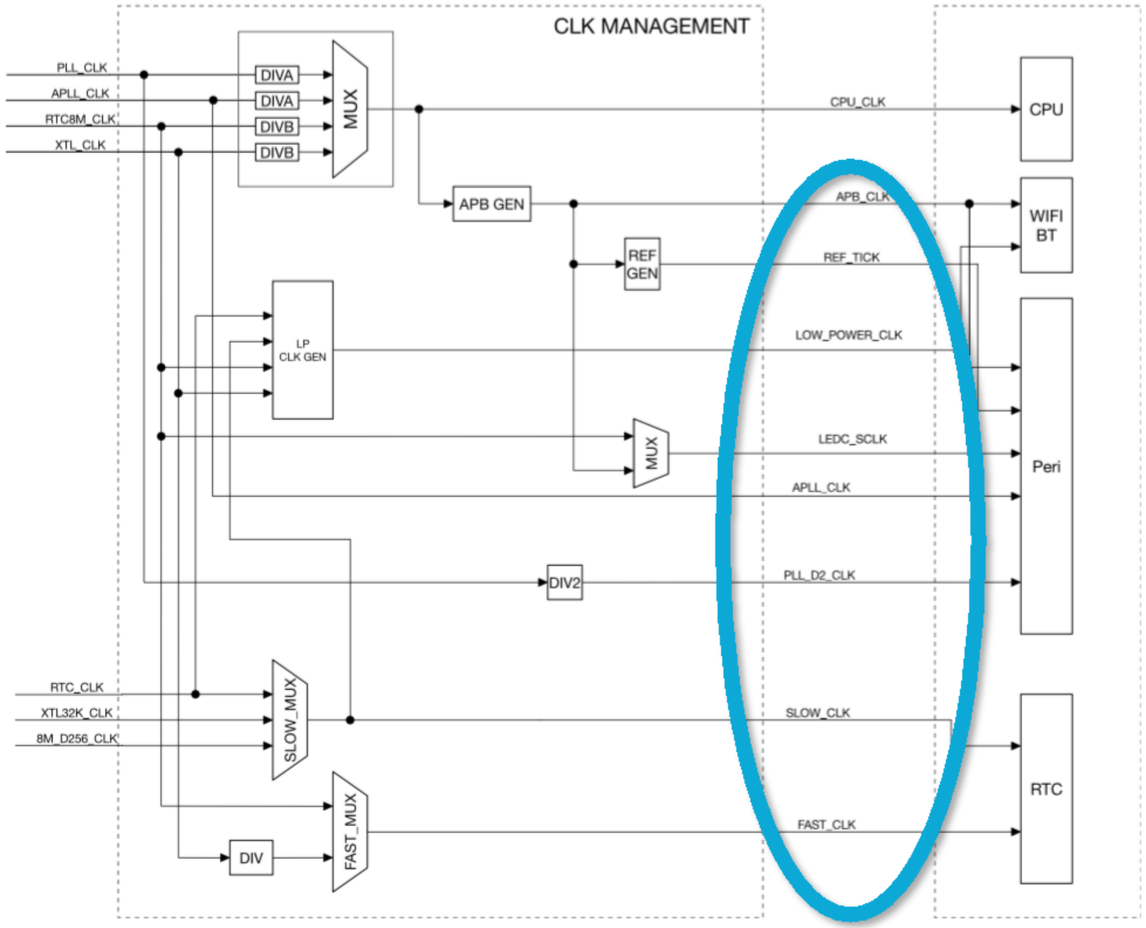
- **寄存器定义**：外设模块寄存器说明在 idf/components/soc/esp32/include/soc/xxx_reg.h 中。
- **寄存器地址**：多于一个的外设模块，以基地址进行区分，在头文件中以宏定义参数进行索引。比如 I2C_SCL_LOW_PERIOD(1) 表示 I2C1 的 SCL_LOW_PERIOD 寄存器地址。
- **寄存器比特域**：每个寄存器下，有不同的功能定义比特域，每个功能域都定义了访问所需的宏定义。**_V** 结尾的宏定义表明功能域的 **Mask**, **_S** 结尾的宏定义表明该功能域在寄存器中的**偏移量**，**_M** 结尾的宏定义表示经过偏移侯的 **Mask**。即，**_M = _V << _S**。
- **结构体访问**：在 soc/xxx_struct.h 中，定义了硬件寄存器的结构体，结合 ld 中定义的外设地址，我们可以通过结构体来访问外设寄存器。如：I2C[1].scl_low_period.period。

注意：请不要在代码中直接操作寄存器，只有在 debug 或者编写硬件驱动时才能直接操作外设寄存器，否则可能造成默认的外设驱动工作不正常。多核多任务的系统中，某些寄存器操作必须在临界区中完成。调用外设驱动来控制硬件功能，如果接口不足以完成，请向我们提出需求。

Driver in ESP-IDF

- **外设驱动**：外设模块驱动代码在 idf/components/driver/xxx.c 中
- **头文件**：外设模块驱动头文件在 idf/components/driver/include/driver/xxx.h 中
- **示例代码**：idf/example/peripherals/

ESP32 System Clock



- **APB CLK**

给大部分数字电路模块提供时钟，一般为 80 MHz。

- **REF_TICK**

由 APB_CLK 分频产生, 在 APB CLK 改变时（低功耗模式）改变分频数保持输出频率不变，使外设能够正常工作。

- **APLL_CLK**

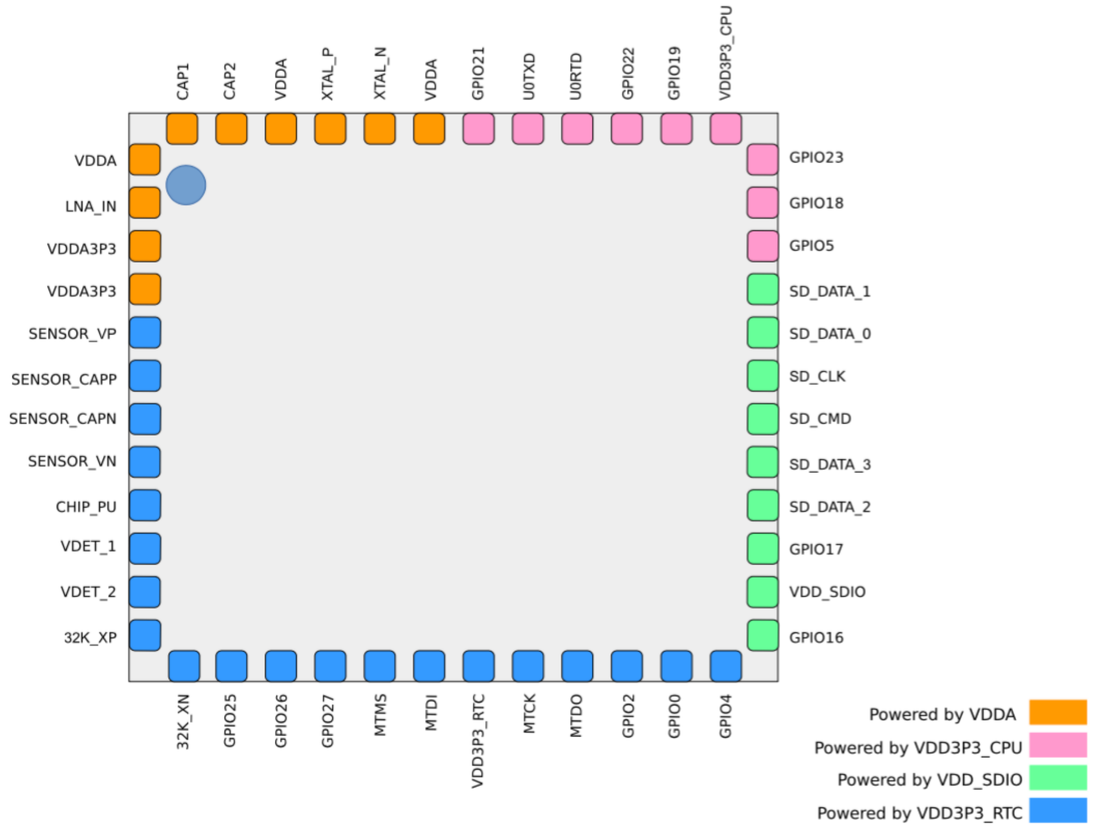
音频应用和其他对于数据传输时效性要求很高的应用都需要高度可配置、低抖动并且精确的时钟源。ESP32 集成了专门用于 I2S 外设的音频 PLL。

- **RTC CLK**

SLOW_CLK 和 FAST_CLK 的时钟源为低频时钟。RTC 模块能够在大多数时钟源关闭的状态下工作。

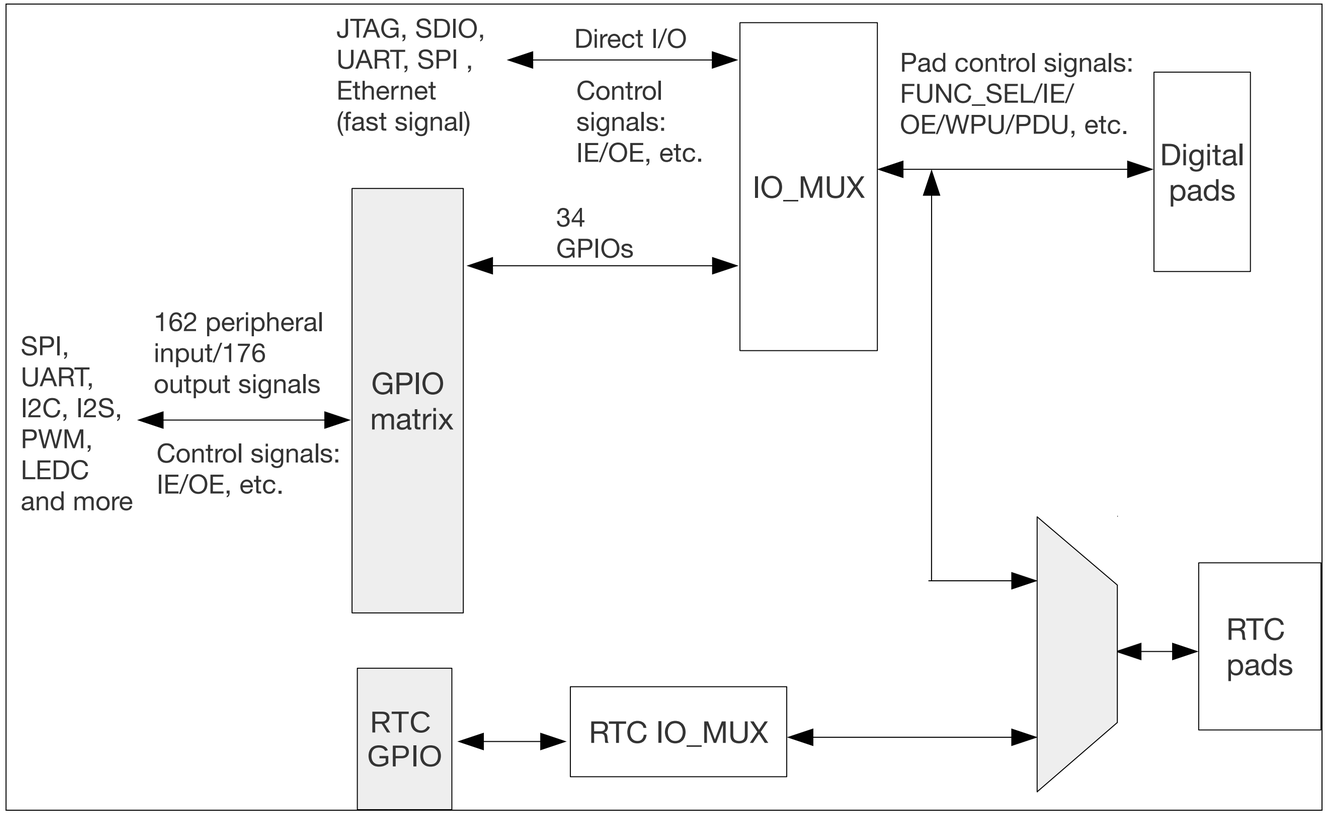
  - SLOW_CLK 允许选择 RTC_CLK，XTL32K_CLK 或 RTC8M_D256_CLK，用于驱动 Power Management 模块。
  - FAST_CLK 允许选择 XTL_CLK 的分频时钟或 RTC8M_CLK，用于驱动 On-chip Sensor 模块。

ESP32 GPIOs



- 总共 34 个 Digital GPIO pin 脚（20 24 28 29 30 31 没有引出）
- SPI Flash 在 4 线模式下会占用 GPIO6 -11，共 6 个脚（大部分模块的连接方式）
- GPIO 序号在 34 以下的 IO (GPIO0-33) 才能够用作输出功能
- GPIO 序号在 34 及以上的 IO (GPIO34-39) 只能用于输入，无法输出。只有带输出功能的 GPIO 才能配置内部上 / 下拉电阻 (GPIO0-33)。
- 内部输出信号只能连接到带输出功能的 GPIO 上 (GPIO0-33)。
- 总共 18 个 RTC GPIO

IO Matrix



ESP32 芯片有 34 个物理 GPIO pad。每个 pad 都可用作一个通用 IO, 或连接一个内部的外设信号。

IO_MUX、RTC IO_MUX 和 GPIO 交换矩阵用于将信号从外设传输至 GPIO pad。这些模块共同组成了芯片的 IO 控制。

少量高速信号必须通过 IO_MUX 输出到外部，或者输入到芯片内部（比如 SDIO）。其他数字信号均可通过 IO 矩阵的映射功能，将输入 / 输出信号连接到不同的 Pin 脚上，使得硬件设计更加灵活。

RTC IO 可以在芯片睡眠过程中保持电平，也可以通过 RTC 模块控制器在低功耗模式下进行控制。但是 RTC IO 输出/输入到有 RTC 功能的 Pin 脚上。

How to Set IO Matrix

- **将 IO MUX 设置为 GPIO 功能**
  void gpio_pad_select_gpio(uint32_t gpio_num);
- **配置 GPIO 的信号寄存器，设置 GPIO 矩阵的输入输出信号**
- **调用系统提供的接口设置 GPIO 矩阵**
  void gpio_matrix_out(uint32_t gpio, uint32_t signal_idx, bool out_inv, bool oen_inv);
  void gpio_matrix_in(uint32_t gpio, uint32_t signal_idx, bool inv);
- **在soc/gpio_sig_map.h 中查询对应的信号序号填入 signal_idx**
- **调用驱动提供的接口设置矩阵信号**
  e.g., esp_err_t ledc_set_pin(int gpio_num, ledc_mode_t mode, ledc_channel_t channel);

GPIO APIs

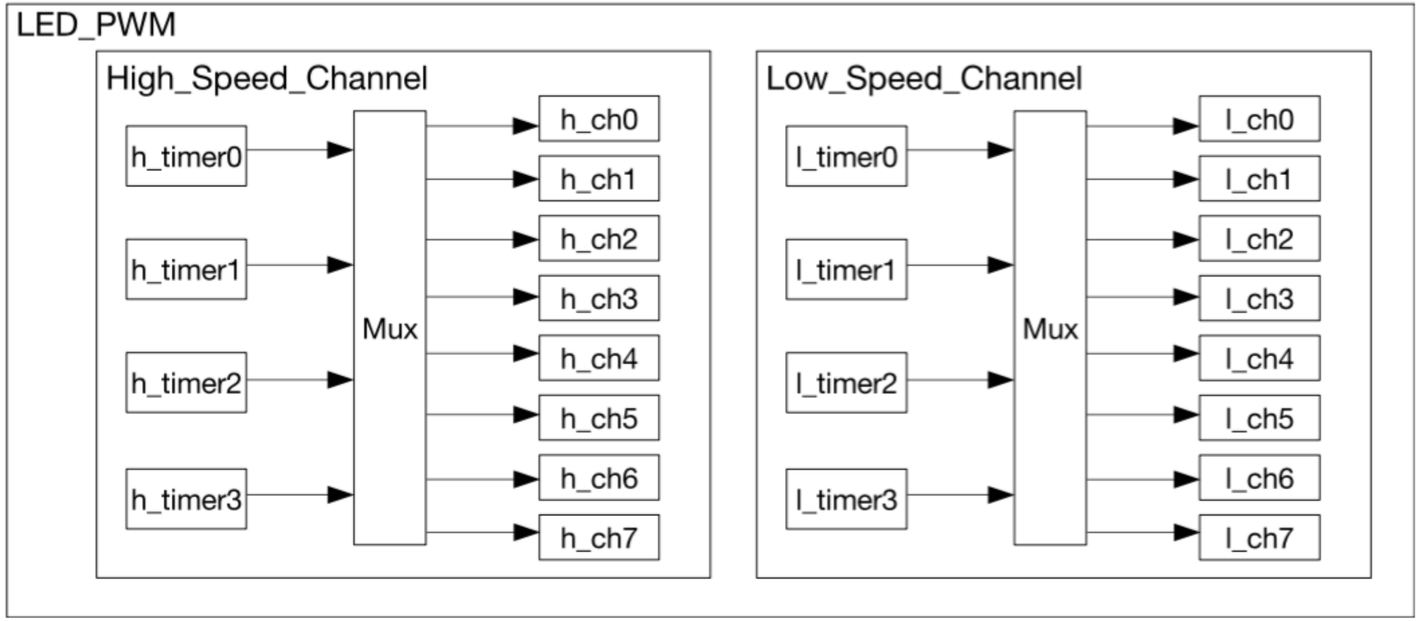
- **设置输入输出**
  gpio_set_direction()
- **设置/读取电平**
  gpio_set_level()/gpio_get_level(
- **设置上下拉**
  gpio_set_pull_mode()
- **设置独立的中断处理**
  gpio_install_isr_service()
  gpio_isr_handler_add()

GPIO APIs

Application Note:

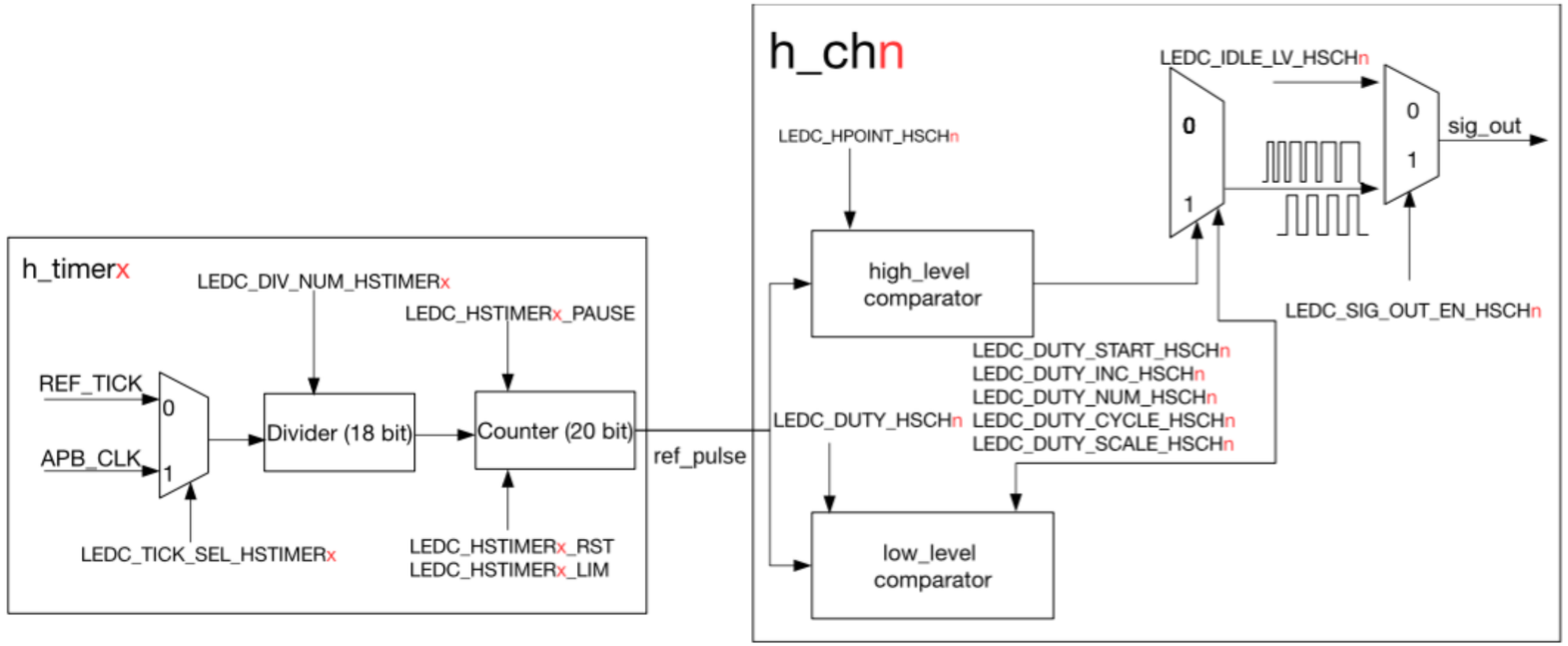
- RTC IO 无法通过 IO 矩阵来分配 pin 脚
- 应用中，优先分配无法设置 Pin 脚的RTC功能 (ADC/DAC/Touch…)，以及高速信号 (SDIO…)
- 尽量将输入功能的信号分配在 34-39 的 GPIO 上，不占用输出功能的 IO。
- 如果使用 ESP32-WROVER 模块，注意避开系统默认占用的 GPIO16/17

LEDC Controller



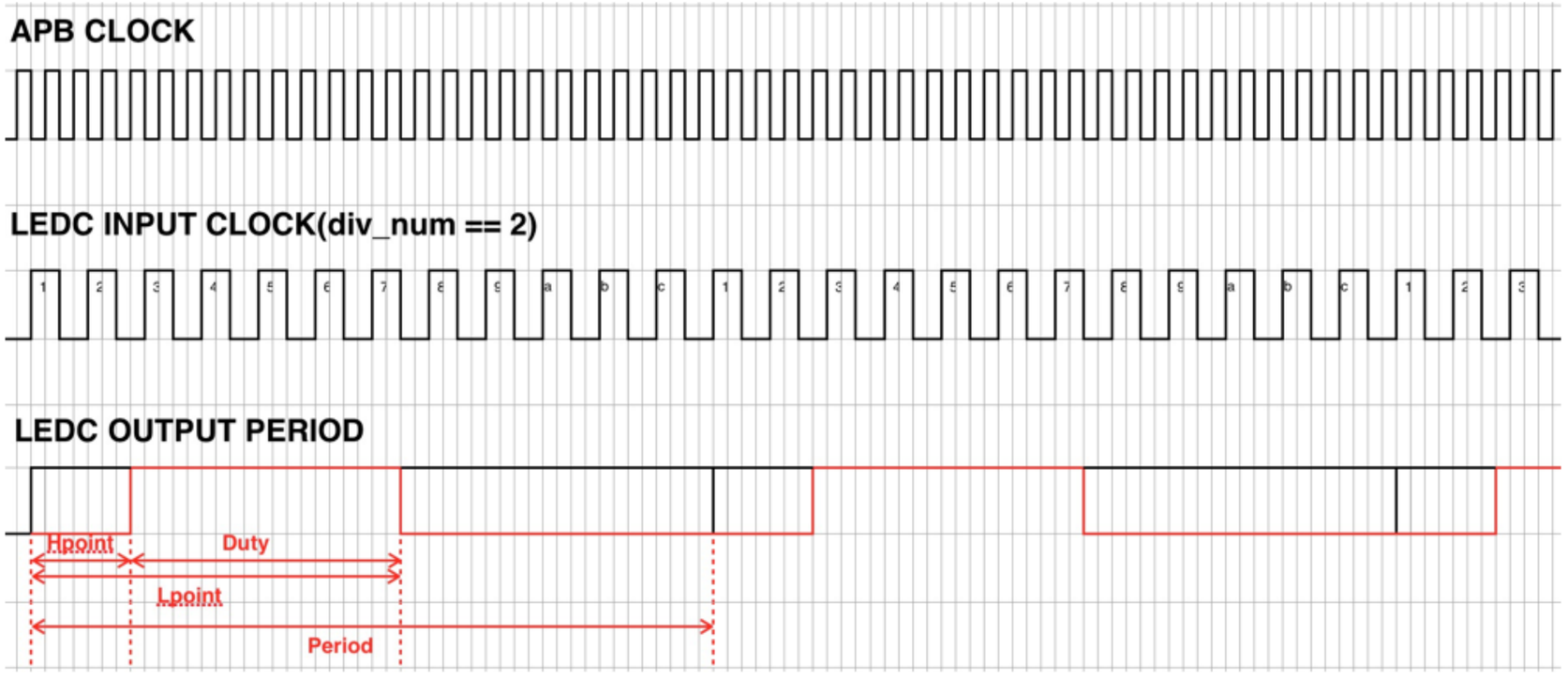
- 用于控制 LED 的亮度和颜色，同时也可以因其他目的产生 PWM 信号
- 最多 16 路通道（8 路高速 8 路低速）
- 8 个独立计数器（4 路高速 4 路低速）

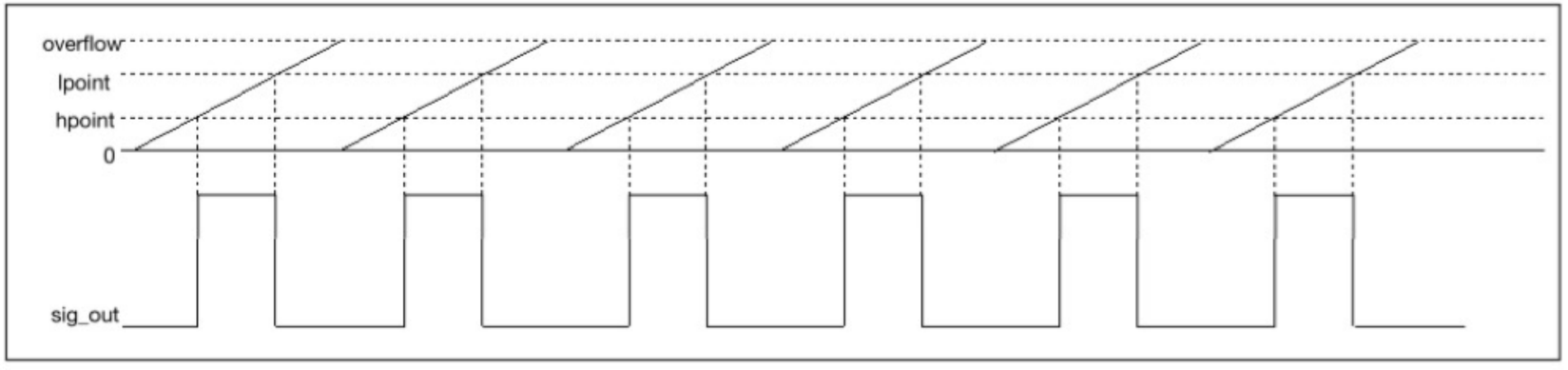
LEDC Controller



- 时钟源输入—> 分频 —> 周期计数 —> 输出波形
- hpoint 输出高电平
- lowpoint 输出低电平

LEDC Controller





LEDC Controller

```
ledc_timer_config_t ledc_timer = {
    .bit_num = LEDC_TIMER_13_BIT, //set timer counter bit number
    .freq_hz = 5000,              //set frequency of pwm
    .speed_mode = LEDC_HS_MODE,   //timer mode,
    .timer_num = LEDC_HS_TIMER    //timer index
};
//configure timer0 for high speed channels
ledc_timer_config(&ledc_timer);
```

Step1. 初始化 LEDC timer

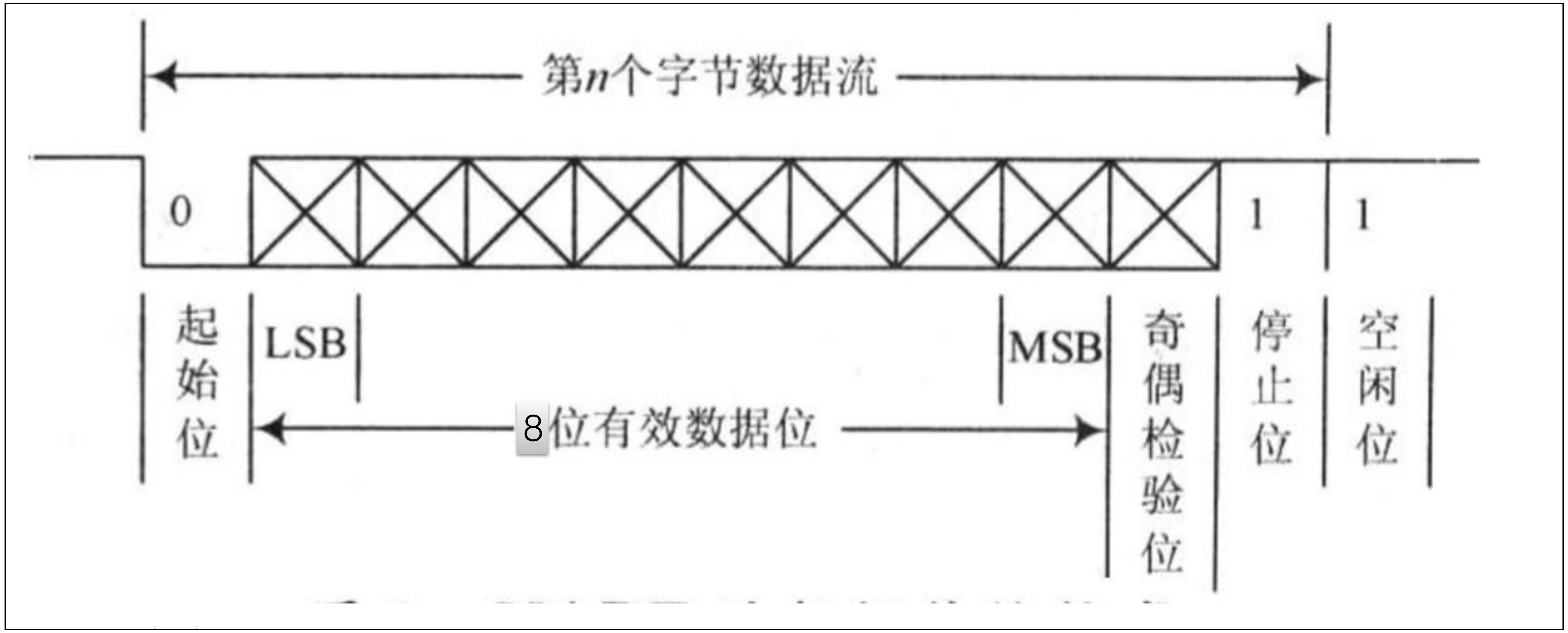
```
ledc_channel_config_t ledc_channel = {
    //set LEDC channel 0
    .channel = ledc_ch[ch].channel,
    //set the duty for initialization.(duty range is 0 ~ ((2**bit_num)-1)
    .duty = 0,
    //GPIO number
    .gpio_num = ledc_ch[ch].io,
    //GPIO INTR TYPE, as an example, we enable fade_end interrupt here.
    .intr_type = LEDC_INTR_FADE_END,
    //set LEDC mode, from ledc_mode_t
    .speed_mode = ledc_ch[ch].mode,
    //set LEDC timer source, if different channel use one timer,
    //the frequency and bit_num of these channels should be the same
    .timer_sel = ledc_ch[ch].timer_idx,
};
//set the configuration
ledc_channel_config(&ledc_channel);
```

Step 2.
初始化 LEDC channel
并指定所使用的 timer

UART

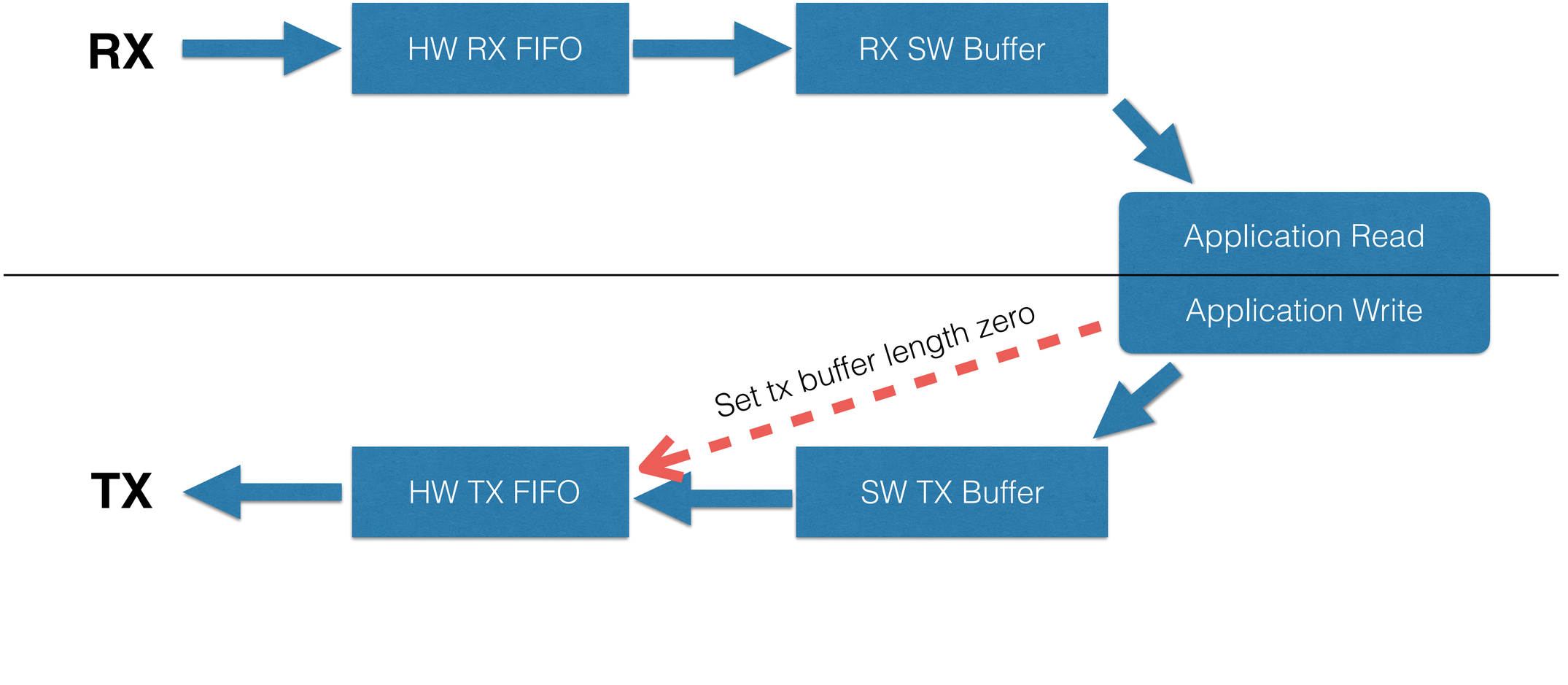
- 可编程收发波特率
- 3 个 UART 的发送 FIFO 以及接收 FIFO 共享 1024 x 8-bit RAM
- 全双工异步通信
- 支持输入信号波特率自检功能
- 支持 5/6/7/8 位数据长度
- 支持 1/1.5/2/3/4 个停止位
- 支持奇偶校验位
- 支持 RS485 协议
- 支持 IrDA 协议
- 支持 DMA 高速数据通信
- 支持 UART 唤醒模式
- 支持软件流控和硬件流控

UART Protocol



- 起始位：先发出一个逻辑”0”信号，表示传输字符的开始。
- 数据位：可以是 5~8 位逻辑”0”或”1”。如 ASCII 码（7 位），扩展 BCD 码（8 位）。小端传输
- 校验位：数据位加上这一位后，使得“1”的位数应为偶数（偶校验）或奇数（奇校验）
- 停止位：它是一个字符数据的结束标志。可以是 1 位、1.5 位、2 位的高电平。
- 空闲位：处于逻辑“1”状态，表示当前线路上没有数据传送。

UART Driver



UART Driver

```
const int uart_num = UART_NUM_1;
uart_config_t uart_config = {
    .baud_rate = 115200,
    .data_bits = UART_DATA_8_BITS,
    .parity = UART_PARITY_DISABLE,
    .stop_bits = UART_STOP_BITS_1,
    .flow_ctrl = UART_HW_FLOWCTRL_CTS_RTS,
    .rx_flow_ctrl_thresh = 122,
};
//Configure UART1 parameters
uart_param_config(uart_num, &uart_config);
```

1. 设置串口参数（波特率，停止位，校验位，硬件流控）

```
//Set UART1 pins(TX: IO4, RX: I05, RTS: IO18, CTS: IO19)
uart_set_pin(uart_num, ECHO_TEST_TXD, ECHO_TEST_RXD, ECHO_TEST_RTS, ECHO_TEST_CTS);
```

2. 通过 IO 矩阵设置输入与输出 pin 脚

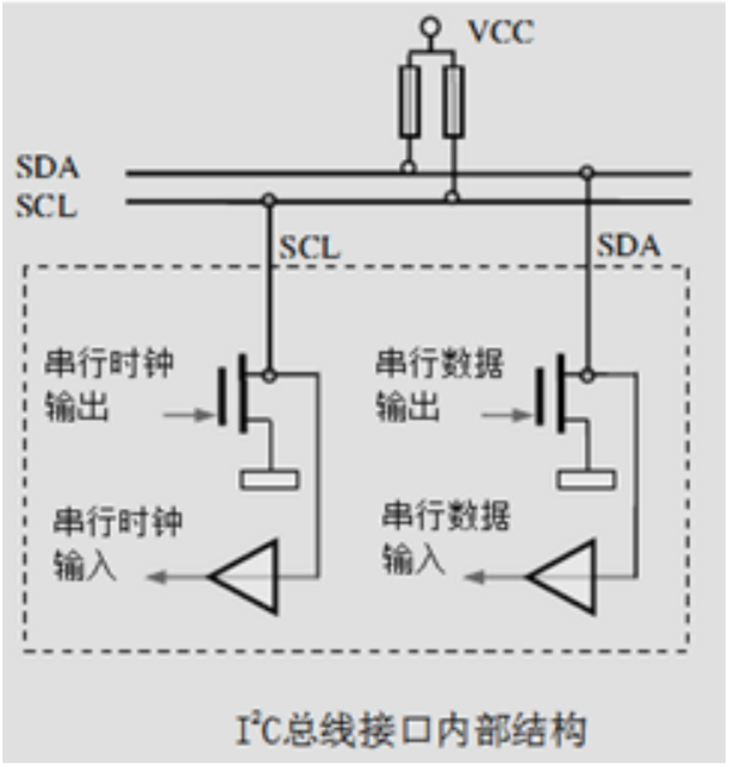
```
//In this example we don't even use a buffer for sending data.
uart_driver_install(uart_num, BUF_SIZE, 0, 0, NULL, 0);
```

3. 初始化串口 buffer 与驱动程序

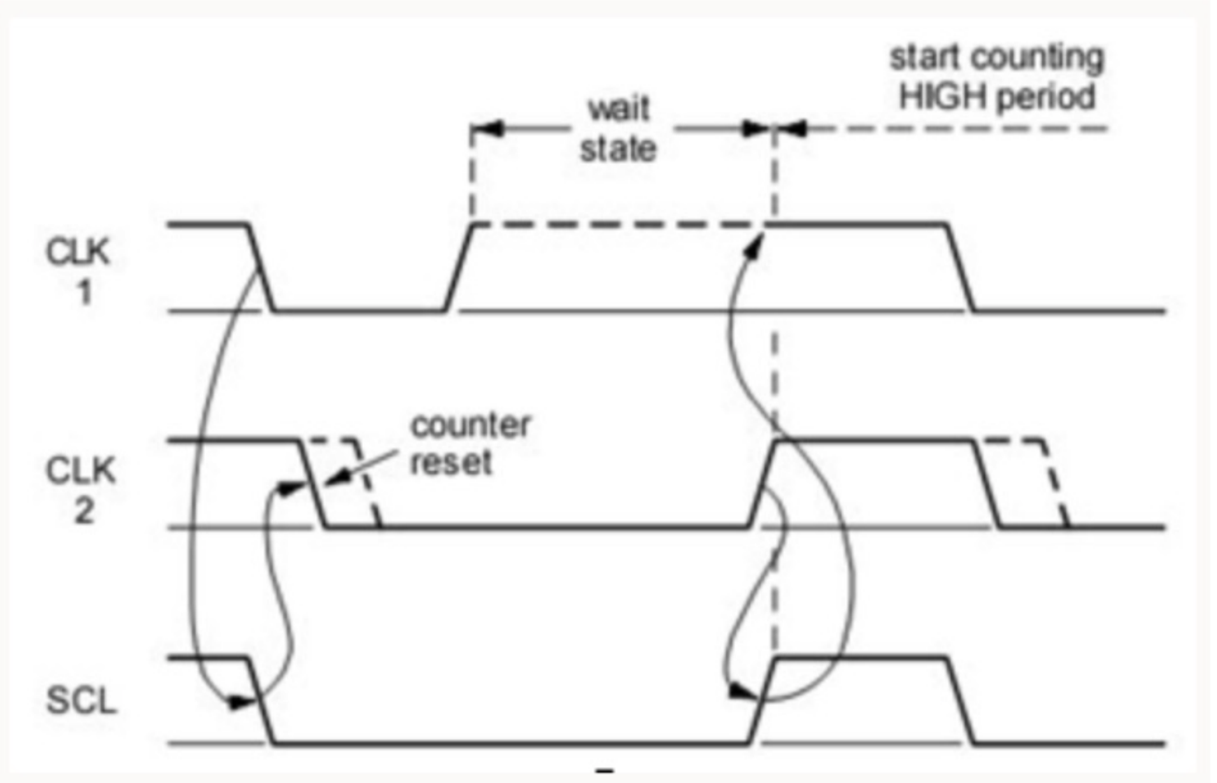
I2C

- 支持主机模式以及从机模式
- 支持多主机多从机通信
- 支持标准模式 (100 kbit/s)
- 支持快速模式 (400 kbit/s)
- 支持 7-bit 以及 10-bit 寻址
- 支持关闭 SCL 时钟实现连续数据传输
- 支持可编程数字噪声滤波功能

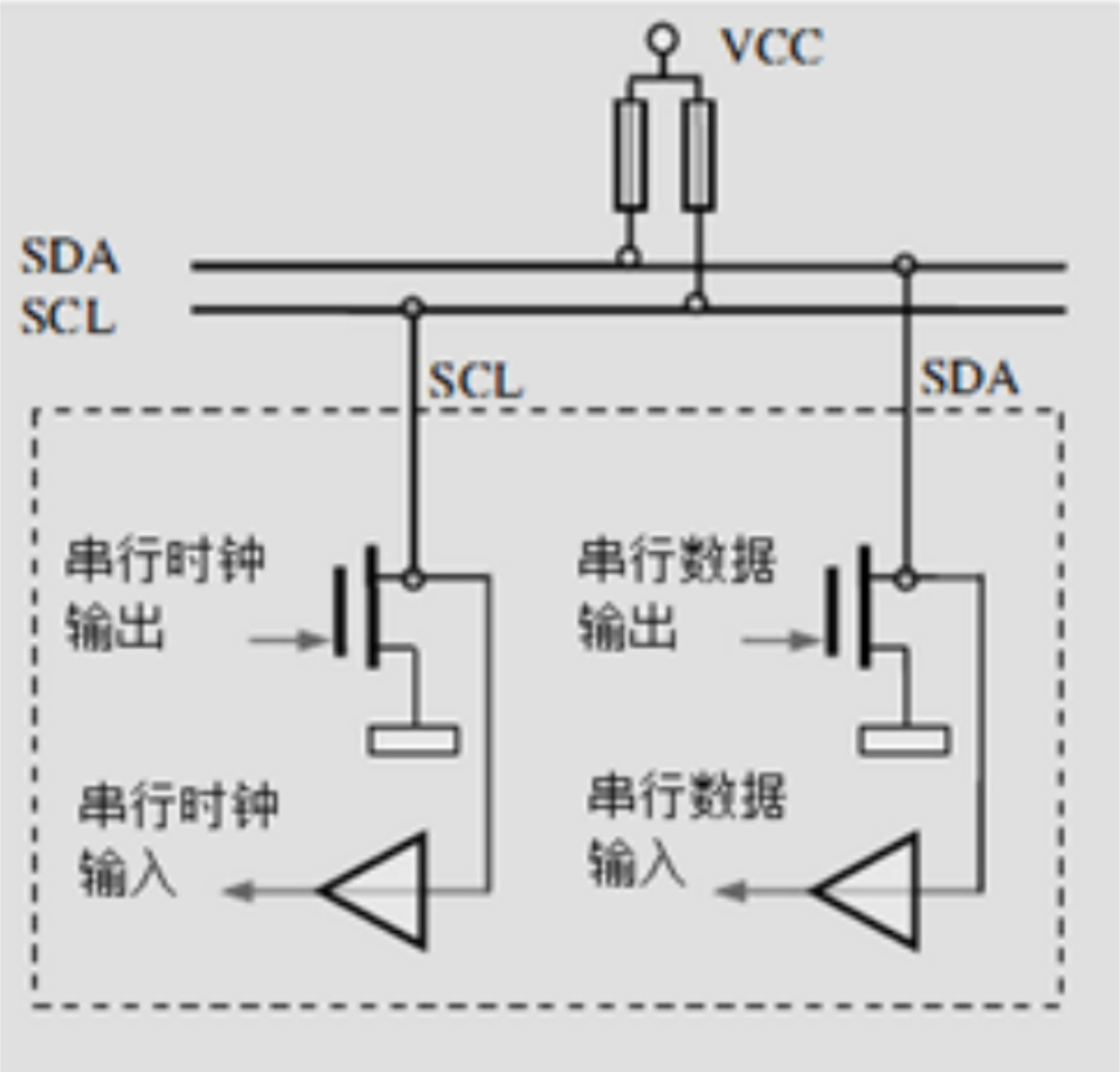
I2C



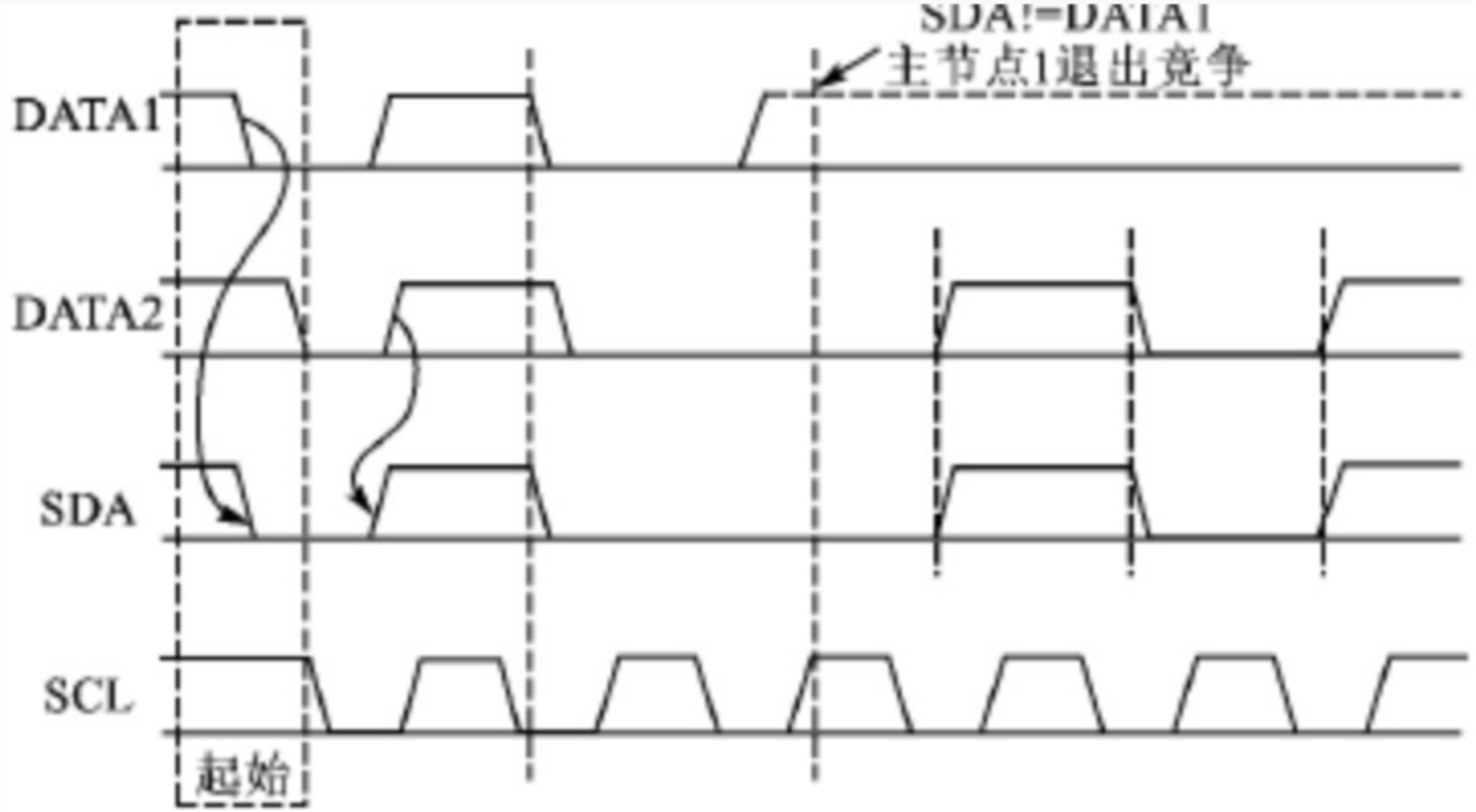
I²C总线接口内部结构



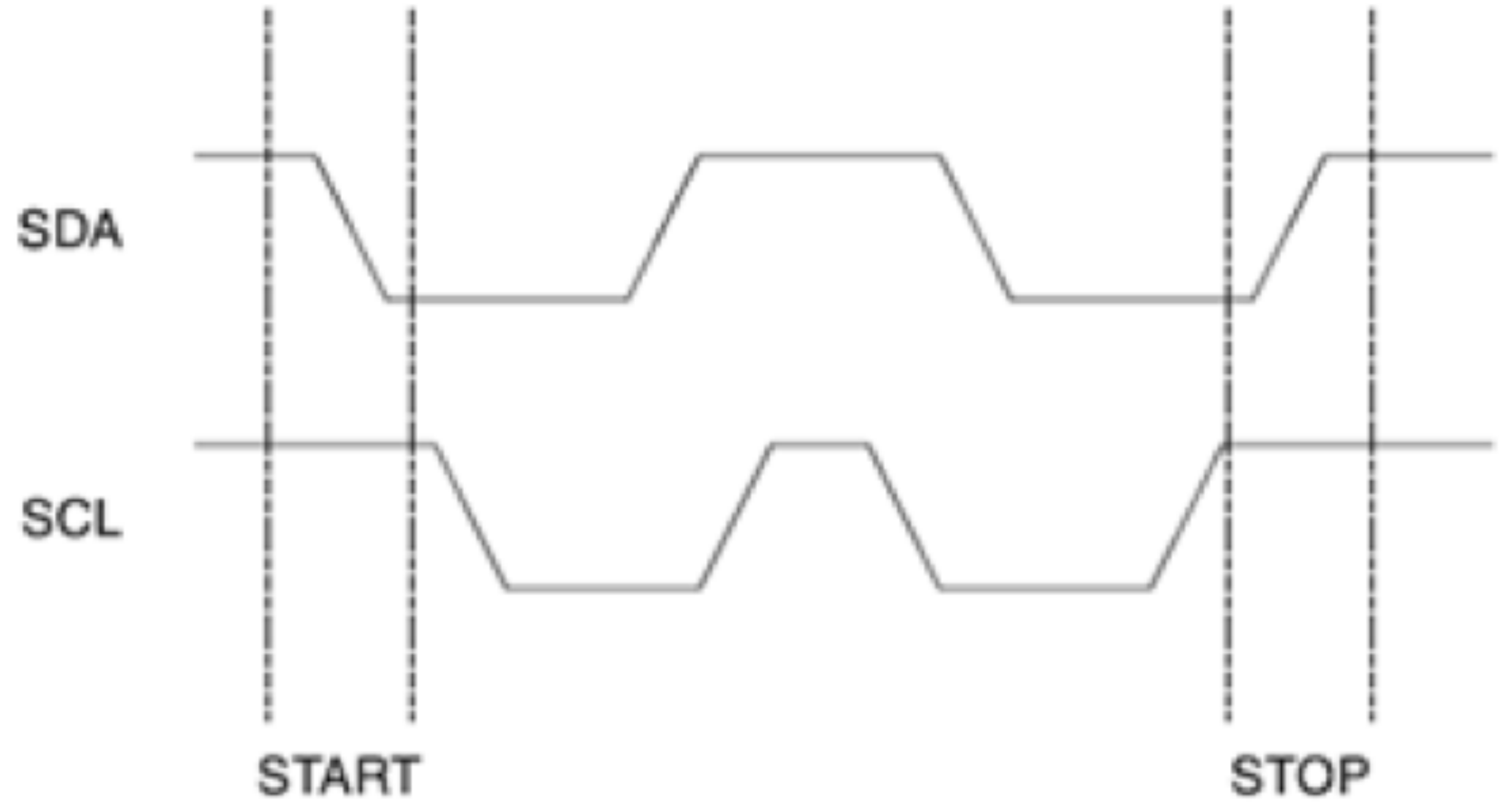
I2C



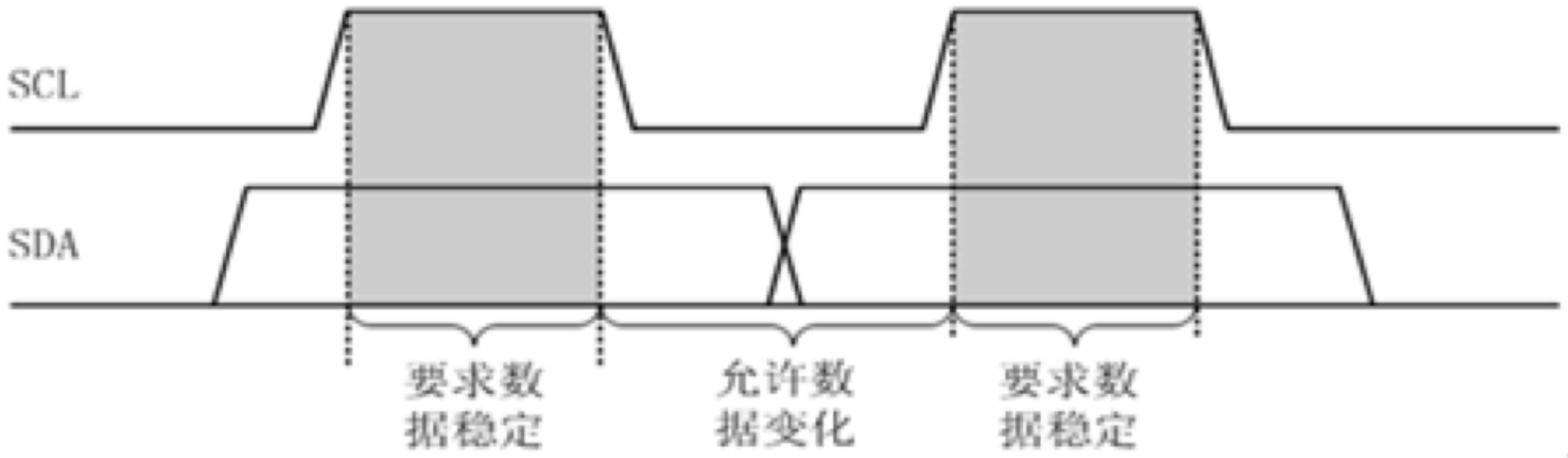
I²C总线接口内部结构



I2C



- 起始信号
- 停止信号



- 时钟上升沿取数据
- 时钟下降沿准备数据

I2C

```
int i2c_master_port = I2C_EXAMPLE_MASTER_NUM;
i2c_config_t conf;
conf.mode = I2C_MODE_MASTER;
conf.sda_io_num = I2C_EXAMPLE_MASTER_SDA_IO;
conf.sda_pullup_en = GPIO_PULLUP_ENABLE;
conf.scl_io_num = I2C_EXAMPLE_MASTER_SCL_IO;
conf.scl_pullup_en = GPIO_PULLUP_ENABLE;
conf.master.clk_speed = I2C_EXAMPLE_MASTER_FREQ_HZ;
i2c_param_config(i2c_master_port, &conf);
```

设置 I2C 总线参数

```
i2c_driver_install(i2c_master_port, conf.mode,
                   I2C_EXAMPLE_MASTER_RX_BUF_DISABLE,
                   I2C_EXAMPLE_MASTER_TX_BUF_DISABLE, 0);
```

设置 I2C buffer 并初始化驱动

I2C

```
    i2c_cmd_handle_t cmd = i2c_cmd_link_create();
    i2c_master_start(cmd);
    i2c_master_write_byte(cmd, ( ESP_SLAVE_ADDR << 1 ) | READ_BIT,
ACK_CHECK_EN);
    i2c_master_read(cmd, data_rd, size - 1, ACK_VAL);
    i2c_master_read_byte(cmd, data_rd + size - 1, NACK_VAL);
    i2c_master_stop(cmd);
```

1. 配置 I2C 命令链表，此时数据尚未发送

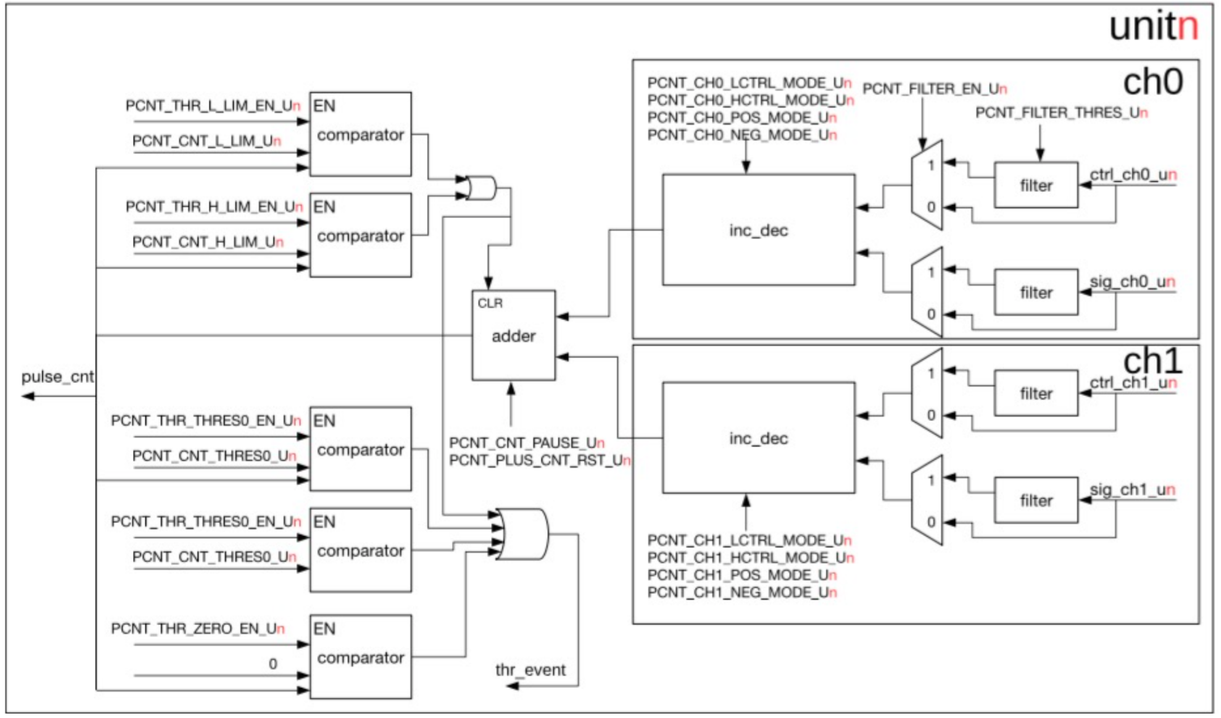
```
    esp_err_t ret = i2c_master_cmd_begin(i2c_num, cmd, 1000 /
portTICK_RATE_MS);
```

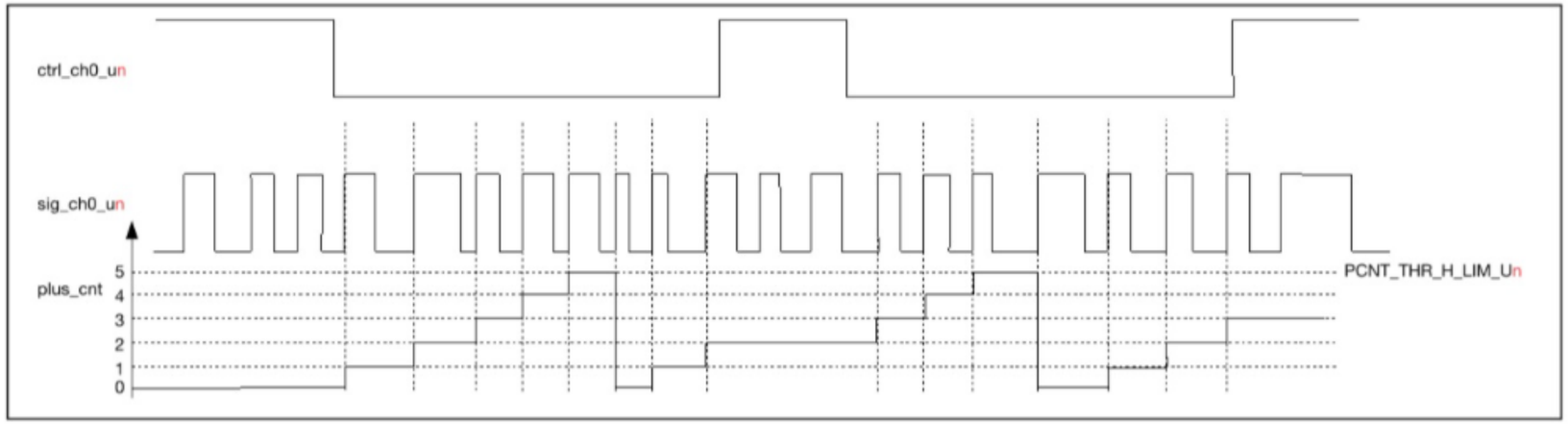
2. 根据链表内容，开始发送数据

```
    i2c_cmd_link_delete(cmd);
```

3. 发送完成后，删除链表，释放内存

PCNT





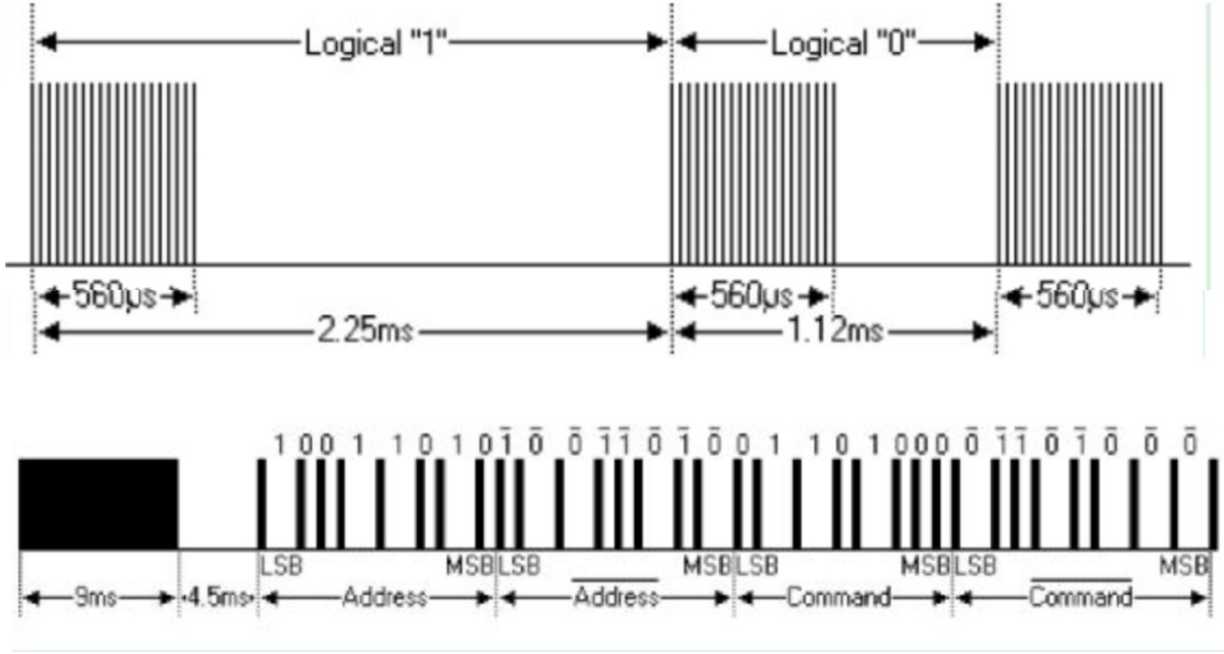
PCNT

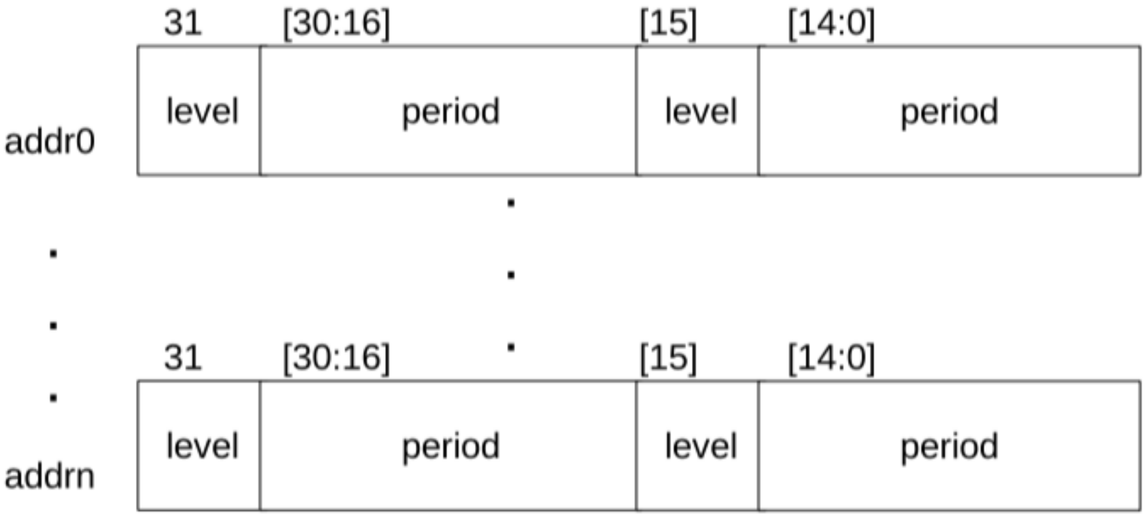
```
pcnt_config_t pcnt_config = {
    /*Set PCNT_INPUT_SIG_IO as pulse input gpio */
    .pulse_gpio_num = PCNT_INPUT_SIG_IO,
    /*set PCNT_INPUT_CTRL_IO as control gpio */
    .ctrl_gpio_num = PCNT_INPUT_CTRL_IO,
    /*Choose channel 0 */
    .channel = PCNT_CHANNEL_0,
    /*Choose unit 0 */
    .unit = PCNT_TEST_UNIT,
    /*Set counter and control mode*/
    /*Counter increase for positive edge on pulse input GPIO*/
    .pos_mode = PCNT_COUNT_INC,
    /*Counter decrease for negative edge on pulse input GPIO*/
    .neg_mode = PCNT_COUNT_DIS,
    /*Counter mode reverse when control input is low level*/
    .lctrl_mode = PCNT_MODE_REVERSE,
    /*Counter mode does not change when control input is high level*/
    .hctrl_mode = PCNT_MODE_KEEP,
    /*Set maximum value for increasing counter*/
    .counter_h_lim = PCNT_H_LIM_VAL,
    /*Set minimum value for decreasing counter*/
    .counter_l_lim = PCNT_L_LIM_VAL,
};
/*Initialize PCNT unit */
pcnt_unit_config(&pcnt_config);
```

设置 PCNT 模式参数

初始化驱动

RMT





- WiFi 转红外
- 单线协议 (WS2812)
- 自定义输出波形
- 接收波形并解析

RMT

```
rmt_config_t rmt_tx;
 rmt_tx.channel = RMT_TX_CHANNEL;
 rmt_tx.gpio_num = RMT_TX_GPIO_NUM;
 rmt_tx.mem_block_num = 1;
 rmt_tx.clk_div = RMT_CLK_DIV;
 rmt_tx.tx_config.loop_en = false;
 rmt_tx.tx_config.carrier_duty_percent = 50;
 rmt_tx.tx_config.carrier_freq_hz = 38000;
 rmt_tx.tx_config.carrier_level = 1;
 rmt_tx.tx_config.carrier_en = RMT_TX_CARRIER_EN;
 rmt_tx.tx_config.idle_level = 0;
 rmt_tx.tx_config.idle_output_en = true;
 rmt_tx.rmt_mode = 0;
 rmt_config(&rmt_tx);
 rmt_driver_install(rmt_tx.channel, 0, 0);
```

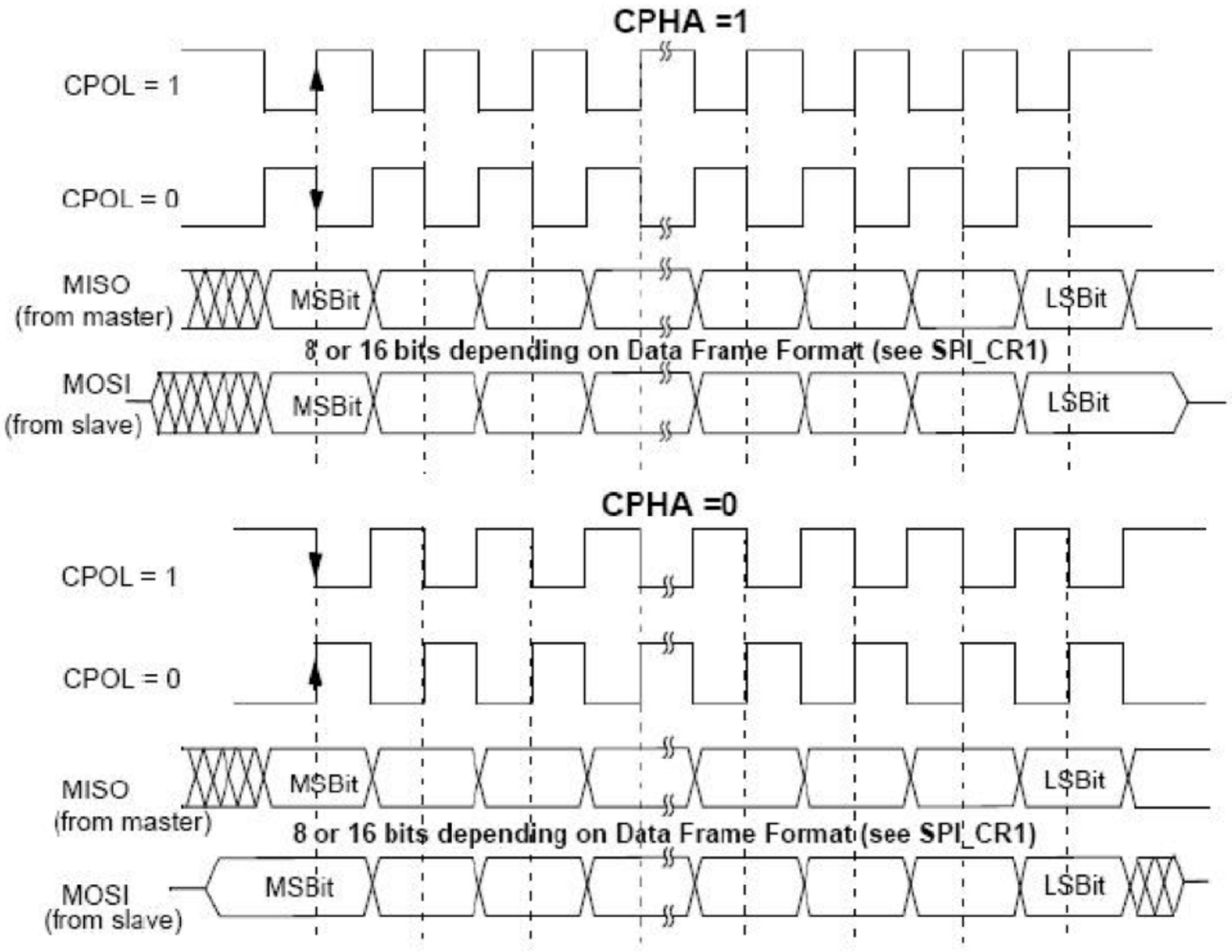
设置 RMT 模式参数

初始化驱动

SPI

GP-SPI 四线	GP-SPI 三线	QSPI	引脚功能信号		
全双工信号总线	半双工信号总线	信号总线	SPI 信号总线	HSPI 信号总线	VSPI 信号总线
MOSI	DATA	D	SPID	HSPID	VSPID
MISO	-	Q	SPIQ	HSPIQ	VSPIQ
CS	CS	CS	SPICS0	HSPICS0	VSPICS0
CLK	CLK	CLK	SPICLK	HSPICLK	VSPICLK
-	-	WP	SPIWP	HSPIWP	VSPIWP
-	-	HD	SPIHD	HSPIHD	VSPIHD

SPI Mode



- 时钟极性 CPOL:
  即 SPI 空闲时，时钟信号 SCLK 的电平（1：空闲时高电平；0：空闲时低电平）
- 时钟相位CPHA:
  即 SPI 在 SCLK 第几个边沿开始采样（0：第一个边沿开始；1：第二个边沿开始）

SPI Init

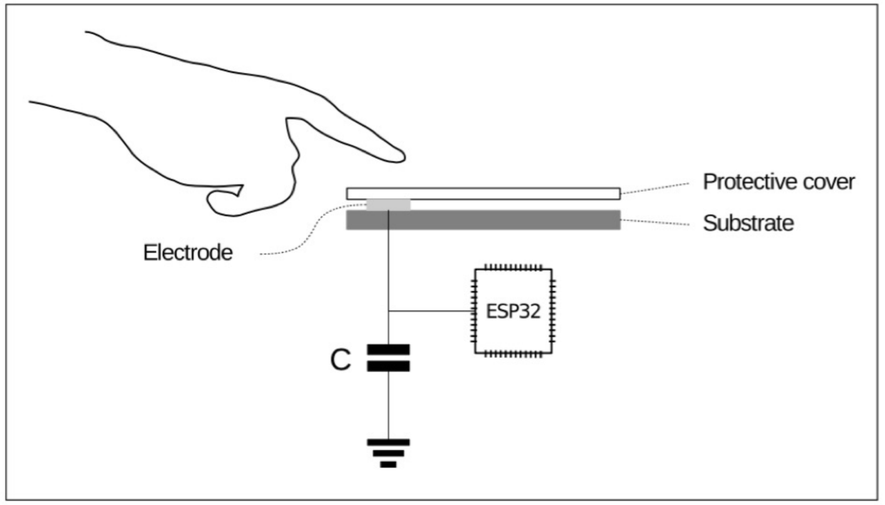
```
esp_err_t ret;
spi_device_handle_t spi;
spi_bus_config_t buscfg={
    .miso_io_num=PIN_NUM_MISO,
    .mosi_io_num=PIN_NUM_MOSI,
    .sclk_io_num=PIN_NUM_CLK,
    .quadwp_io_num=-1,
    .quadhd_io_num=-1
};
//Initialize the SPI bus
ret=spi_bus_initialize(HSPI_HOST, &buscfg, 1);
```

SPI bus init

SPI device init

```
spi_device_interface_config_t devcfg={
    .clock_speed_hz=10*1000*1000,           //Clock out at 10 MHz
    .mode=0,                                //SPI mode 0
    .spics_io_num=PIN_NUM_CS,               //CS pin
    .queue_size=7,                          //We want to be able to queue 7 transactions at a time
    .pre_cb=lcd_spi_pre_transfer_callback,  //Specify pre-transfer callback to handle D/C line
};
//Attach the LCD to the SPI bus
ret=spi_bus_add_device(HSPI_HOST, &devcfg, &spi);
```

Touch Sensor

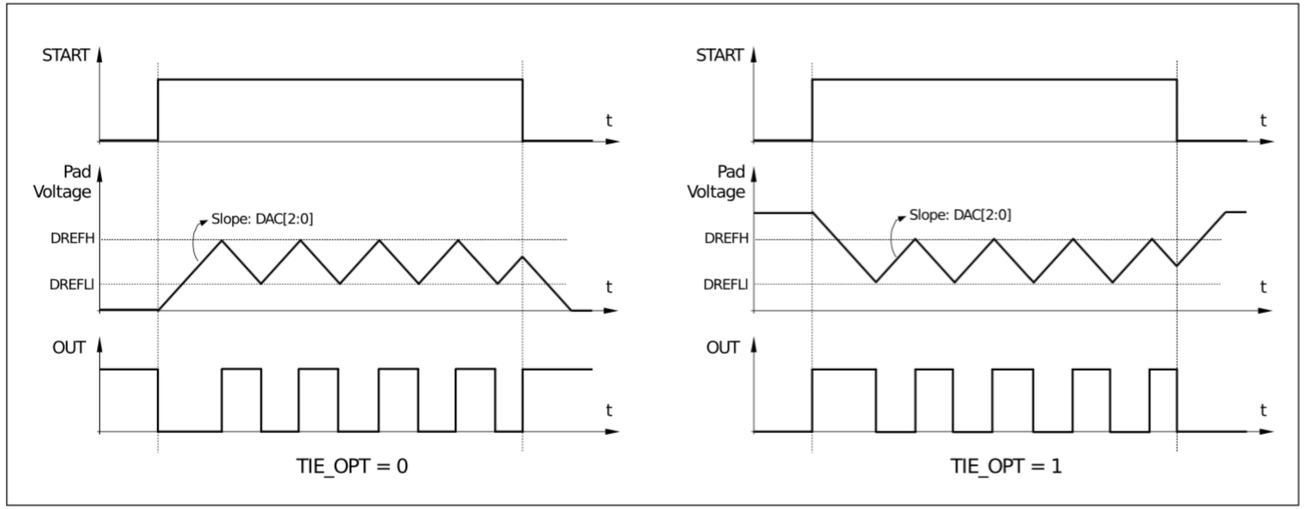


- 最多支持 10 路电容触摸管脚/通用输入输出接口
- 触摸管脚可以组合使用，可覆盖更大触感区域或更多触感点
- 触摸管脚的传感由有限状态机 (FSM) 硬件控制，由软件或专用硬件计时器发起
- 触摸管脚是否受到触碰的信息可由以下方式获得：
  - 由软件直接检查触摸传感器的寄存器
  - 由触摸监测模块发起的中断信号判断
  - 由触摸监测模块上的 CPU 是否从 Deep-sleep 中唤醒判断
- 支持以下场景下的低功耗工作：
  - CPU 处于 Deep-sleep 节能模式,将在受到触碰后逐步唤醒
  - 触摸监测由超低功耗协处理器管理 ULP 用户程序可通过写入与检查特定寄存器,判断是否达到触碰阈值

Touch Sensor

触摸传感信号名	管脚
T0	GPIO4
T1	GPIO0
T2	GPIO2
T3	MTDO
T4	MTCK
T5	MTDI
T6	MTMS
T7	GPIO27
T8	32K_XN
T9	32K_XP

Touch Sensor



- 调用 touch_pad_set_meas_time(uint16_t sleep_cycle, uint16_t meas_cycle) 来设置测量间隔与测量时间。
- 调用 touch_pad_set_voltage(touch_high_volt_t refh, touch_low_volt_t refl, touch_volt_atten_t atten); 来设置充放电电压的上下限。
- 调用touch_pad_read(touch_pad_t touch_num, uint16_t * touch_value); 来读取测量到的原始值。
- 调用touch_pad_read_filtered(touch_pad_t touch_num, uint16_t *touch_value); 来读取经过滤波去噪后的测量值。

IoT Solutions

- **ESP-IoT-Solution project**
  https://github.com/espressif/esp-iot-solution
  - 包含了多种传感器/LCD 等外部设备驱动
  - 包含了阿里，京东等云平台接口
  - 包含了多种应用文档及测试文档
  - 包含了多种应用场景的示例代码

Q&A