ZS110A 驱动用户指南

发布 ***1.0.0***

2018 年 **11** 月 **02** 日

目录

CHAPTER 1

文档介绍

1.1 文档目的

本文介绍了 ZS110A SDK 设备驱动接口和使用示例，供用户快速了解驱动使用方法。

1.2 术语说明

表 1.1: 术语说明

术语	说明
GPIO	General Purpose Input Output，通用输入输出接口
I2C	Inter-Integrated Circuit，是一种串行通讯总线
SPI	Serial Peripheral Interface，串行外设接口
NVRAM	Non-Volatile Random Access Memory，非易失内存

1.3 参考文档

- http://docs.zephyrproject.org/

1.4 版本历史

表 1.2: 版本历史

日期	版本	注释	作者
2018-08-22	1.0	初始版本	ZS110A 项目组

CHAPTER 2

设备驱动框架

2.1 驱动模型介绍

SDK 中的设备驱动是基于 Zephyr 的驱动模型进行开发的。驱动模型为各个驱动提供了统一的设备注册、配置、功耗管理等接口。每种类型的驱动程序（UART、I2C、Watchdog 等）都定义了通用 API 函数，API 函数中再来调用具体设备驱动的实现。

比如下面的 watchdog 驱动的实现：

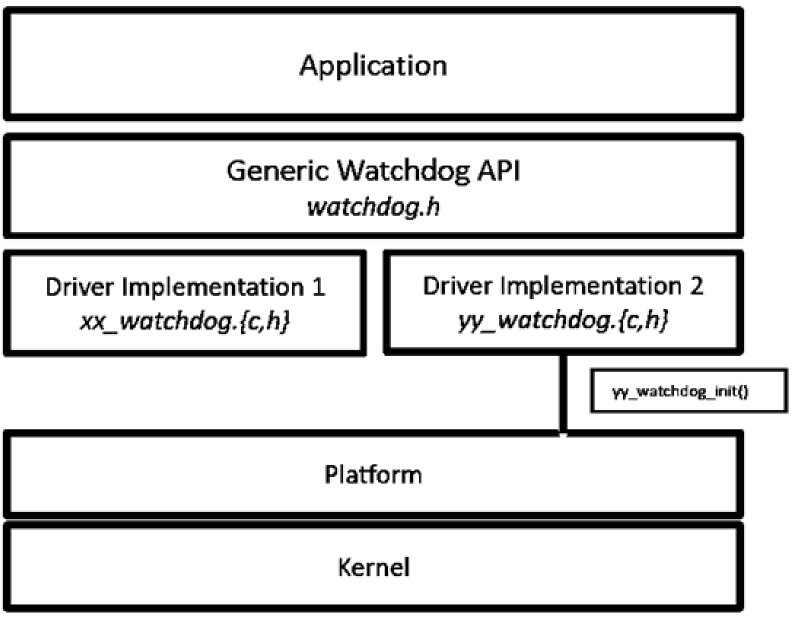

图 2.1: watchdog 驱动实现框架

在 watchdog.h 中定义了 watchdog 这一类设备统一的 API 接口函数

```
typedef void (*wdt_api_enable)(struct device *dev);

struct wdt_driver_api {
        wdt_api_enable enable;
        ...
};

static inline void wdt_enable(struct device *dev)
{
        const struct wdt_driver_api *api = dev->driver_api;

        api->enable(dev);
}
```

在具体的 watchdog 驱动中实现 API 函数

```
void wdt_xx_enable(struct device *dev)
{
        ...
}

static const struct wdt_driver_api wdt_api = {
        .enable = wdt_xx_enable,
        ...
};
```

2.2 驱动初始化

系统初始化时会根据设备初始化参数定义来调用各个设备的初始化函数。

设备驱动使用驱动框架提供的 DEVICE_INIT()、DEVICE_AND_API_INIT() 等宏接口来定义初始化配置参数。

```
#define DEVICE_AND_API_INIT(dev_name, drv_name, init_fn, data,␣
↪cfg_info, level, prio, api)  ...
```

初始化参数意义为：

- dev_name: 设备名
- drv_name: 驱动名
- init_fn: 驱动初始化函数
- data: 驱动运行时自定义参数
- cfg_info: 驱动配置自定义参数
- level: 设备初始化优先级
- prio: 设备中断优先级

- api: 驱动实现的 api 结构体

比如 watchdog 驱动中设备初始化参数定义：

```
DEVICE_AND_API_INIT(wdt_xx, CONFIG_WDT_XX_DEVICE_NAME, wdt_xx_init,
                    &shared_data, NULL,
                    PRE_KERNEL_1, CONFIG_KERNEL_INIT_PRIORITY_DEVICE,
                    &wdt_api);
```

驱动程序可能会依赖其它驱动或内核服务，所以需要能自定义初始化顺序。设备驱动定义时需要指定初始化等级。下面是几个系统预定义的初始化等级

- PRE_KERNEL_1

  用于那些没有任何依赖的设备，例如那些纯粹只需要处理器/SoC 上的硬件的设备。这些设备在配置期间不需要使用任何内内核服务，因此此时内核服务还未启动。不过，中断子系统会被配置，因此可以设置中断。在这个等级上的初始化函数运行在中断栈上面。

- PRE_KERNEL_2

  用于那些依赖于已被初始化的 PRE_KERNEL_1 等级的设备的设备。这些设备在配置期间不使用任何内核服务，因此此时内核服务还未启动。在这个等级上的初始化函数运行在中断栈上面。

- POST_KERNEL

  用于那些在配置期间需要依赖内核服务的设备。在这个等级上的初始化函数运行在内核主栈的上下文中。

- APPLICATION

  用于需要自动配置的应用程序组件（即非内核组件）。这些设别在配置期间可以使用内核提供的所有服务。在这个等级上的初始化函数运行在内核主栈的上下文中。

在每个初始化等级，您还需要指定具体的初始化优先级，用于区分相同初始化等级的初始化顺序。这个优先级是 0 到 99 之间的整数值。优先级越低表示越早被初始化。优先级必须是一个前面没有补零的或者没有符号的十进制整数字面量或者一个整数宏定义（例如 #define MY_INIT_PRIO 32）。这里的定义不能用符号表达式（例如 CONFIG_KERNEL_INIT_PRIORITY_DEFAULT + 5）。

CHAPTER 3

GPIO 驱动程序

3.1 GPIO 总线

GPIO 是 MCU 常用的接口，用户可以单独控制以一个 IO 的输入和输出功能。可用于控制一个 LED 灯的亮灭、输入一个按键的状态等功能。

3.2 GPIO 控制器

ATB110X 共有 30 个 GPIO，具有下列可配置功能。

- PAD 功能可配置
- 内置 15K 上下拉电阻和 1KB 下拉电阻
- 4 档驱动能力调节
- 可选支持施密特触发器
- 中断输入功能，支持电平和沿触发
- 支持系统唤醒

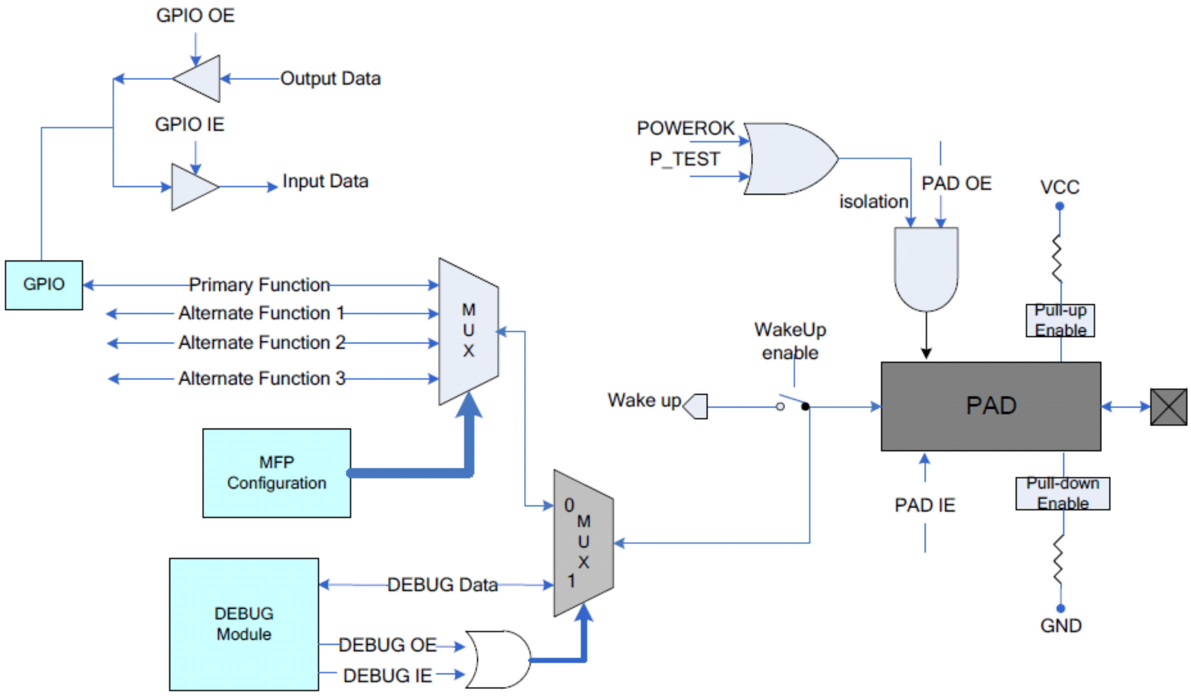

图 3.1: GPIO 控制器框图

3.3 驱动接口说明

下面介绍一下 GPIO 的具体接口和参数说明

Public APIs for GPIO drivers.

Defines

GPIO_DIR_IN
GPIO pin to be input.

GPIO_DIR_OUT
GPIO pin to be output.

GPIO_INT
GPIO pin to trigger interrupt.

GPIO_INT_ACTIVE_LOW
GPIO pin trigger on level low or falling edge.

GPIO_INT_ACTIVE_HIGH
GPIO pin trigger on level high or rising edge.

GPIO_INT_CLOCK_SYNC
GPIO pin trigger to be synchronized to clock pulses.

GPIO_INT_DEBOUNCE
Enable GPIO pin debounce.

GPIO_INT_LEVEL
Do Level trigger.

GPIO_INT_EDGE
Do Edge trigger.

GPIO_INT_DOUBLE_EDGE
Interrupt triggers on both rising and falling edge.

GPIO_POL_NORMAL
GPIO pin polarity is normal.

GPIO_POL_INV
GPIO pin polarity is inverted.

GPIO_PUD_NORMAL
GPIO pin to have no pull-up or pull-down.

GPIO_PUD_PULL_UP
Enable GPIO pin pull-up.

GPIO_PUD_PULL_DOWN
Enable GPIO pin pull-down.

GPIO_PIN_ENABLE
Enable GPIO pin.

GPIO_PIN_DISABLE
Disable GPIO pin.

GPIO_DS_DFLT_LOW
Default drive strength standard when GPIO pin output is low.

GPIO_DS_ALT_LOW
Alternative drive strength when GPIO pin output is low. For hardware that does not support configurable drive strength use the default drive strength.

GPIO_DS_DISCONNECT_LOW
Disconnect pin when GPIO pin output is low. For hardware that does not support disconnect use the default drive strength.

GPIO_DS_DFLT_HIGH
Default drive strength when GPIO pin output is high.

GPIO_DS_ALT_HIGH
Alternative drive strength when GPIO pin output is high. For hardware that does not support configurable drive strengths use the default drive strength.

GPIO_DS_DISCONNECT_HIGH
Disconnect pin when GPIO pin output is high. For hardware that does not support disconnect use the default drive strength.

GPIO_DECLARE_PIN_CONFIG_IDX(_idx)

GPIO_DECLARE_PIN_CONFIG

GPIO_PIN_IDX(_idx, _controller, _pin)

GPIO_PIN(_controller, _pin)

GPIO_GET_CONTROLLER_IDX(_idx, _conf)

GPIO_GET_PIN_IDX(_idx, _conf)

GPIO_GET_CONTROLLER(_conf)

GPIO_GET_PIN(_conf)

Typedefs

typedef gpio_callback_handler_t

Define the application callback handler function signature.

Note: cb pointer can be used to retrieve private data through CONTAINER_OF() if original struct *gpio_callback* is stored in another private structure.

Parameters

- `struct device *port`: Device struct for the GPIO device.
- `struct gpio_callback *cb`: Original struct *gpio_callback* owning this handler
- `u32_t pins`: Mask of pins that triggers the callback handler

Functions

static int **gpio_pin_configure**(**struct** device **port*, u32_t *pin*, int *flags*)

Configure a single pin.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to device structure for the driver instance.
- `pin`: Pin number to configure.
- `flags`: Flags for pin configuration. IN/OUT, interrupt …

static int **gpio_pin_write**(**struct** device **port*, u32_t *pin*, u32_t *value*)

Write the data value to a single pin.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.

- `pin`: Pin number where the data is written.
- `value`: Value set on the pin.

static int **gpio_pin_read(struct** device **port*, u32_t *pin*, u32_t **value*)

Read the data value of a single pin.

Read the input state of a pin, returning the value 0 or 1.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `pin`: Pin number where data is read.
- `value`: Integer pointer to receive the data values from the pin.

static void **gpio_init_callback(struct** *gpio_callback* **callback*, *gpio_callback_handler_t* *handler*, u32_t *pin_mask*)

Helper to initialize a struct *gpio_callback* properly.

Parameters

- `callback`: A valid Application’s callback structure pointer.
- `handler`: A valid handler function pointer.
- `pin_mask`: A bit mask of relevant pins for the handler

static int **gpio_add_callback(struct** device **port*, **struct** *gpio_callback* **callback*)

Add an application callback.

Note: enables to add as many callback as needed on the same port.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `callback`: A valid Application’s callback structure pointer.

static int **gpio_remove_callback(struct** device **port*, **struct** *gpio_callback* **callback*)

Remove an application callback.

Note: enables to remove as many callbacks as added through gpio_add_callback().

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `callback`: A valid application’s callback structure pointer.

static int **gpio_pin_enable_callback(struct** device **port*, u32_t *pin***)**

Enable callback(s) for a single pin.

Note: Depending on the driver implementation, this function will enable the pin to trigger an interruption. So as a semantic detail, if no callback is registered, of course none will be called.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `pin`: Pin number where the callback function is enabled.

static int **gpio_pin_disable_callback(struct** device **port*, u32_t *pin***)**

Disable callback(s) for a single pin.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `pin`: Pin number where the callback function is disabled.

static int **gpio_port_configure(struct** device **port*, int *flags***)**

Configure all the pins the same way in the port. List out all flags on the detailed description.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `flags`: Flags for the port configuration. IN/OUT, interrupt …

static int **gpio_port_write(struct** device **port*, u32_t *value***)**

Write a data value to the port.

Write the output state of a port. The state of each pin is represented by one bit in the value. Pin 0 corresponds to the least significant bit, pin 31 corresponds to the most significant bit. For ports with less that 32 physical pins the most significant bits which do not correspond to a physical pin are ignored.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `value`: Value to set on the port.

static int **gpio_port_read(struct** device **port*, u32_t **value***)**

Read data value from the port.

Read the input state of a port. The state of each pin is represented by one bit in the returned value. Pin 0 corresponds to the least significant bit, pin 31 corresponds to the most significant bit. Unused bits for ports with less that 32 physical pins are returned as 0.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.
- `value`: Integer pointer to receive the data value from the port.

static int **gpio_port_enable_callback(struct** device **port***)**

Enable callback(s) for the port.

Note: Depending on the driver implementation, this function will enable the port to trigger an interruption on all pins, as long as these are configured properly. So as a semantic detail, if no callback is registered, of course none will be called.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.

static int **gpio_port_disable_callback(struct** device **port***)**

Disable callback(s) for the port.

Return 0 if successful, negative errno code on failure.

Parameters

- `port`: Pointer to the device structure for the driver instance.

static int **gpio_get_pending_int(struct** device **dev***)**

Function to get pending interrupts.

The purpose of this function is to return the interrupt status register for the device. This is especially useful when waking up from low power states to check the wake up source.

Parameters

- `dev`: Pointer to the device structure for the driver instance.

Return Value

- `status`: != 0 if at least one gpio interrupt is pending.
- `0`: if no gpio interrupt is pending.

struct gpio_callback

#include <gpio.h> GPIO callback structure.

Used to register a callback in the driver instance callback list. As many callbacks as needed can be added as long as each of them are unique pointers of struct *gpio_callback*. Beware such structure should not be allocated on stack.

Note: To help setting it, see gpio_init_callback() below

Public Members

sys_snode_t **node**

This is meant to be used in the driver and the user should not mess with it (see drivers/gpio/gpio_utils.h)

gpio_callback_handler_t **handler**

Actual callback function being called when relevant.

u32_t **pin_mask**

A mask of pins the callback is interested in, if 0 the callback will never be called. Such pin_mask can be modified whenever necessary by the owner, and thus will affect the handler being called or not. The selected pins must be configured to trigger an interrupt.

struct gpio_pin_config

Public Members

char ***gpio_controller**

u32_t **gpio_pin**

3.4 驱动使用示例

下面以一个示例介绍一下 GPIO 接口的使用。示例中使用 GPIO4 来控制一个 LED 灯的闪烁，以及通过中断检测 GPIO2 的高电平中断

```
#include <gpio.h>

#define GPIO_OUT_PIN    4
#define GPIO_INT_PIN    2
#define GPIO_NAME       "GPIO_"

#define GPIO_DRV_NAME "GPIO_0"

void gpio_callback(struct device *port,
                   struct gpio_callback *cb, u32_t pins)
```

(continues on next page)

(续上页)

```
{
        printk(GPIO_NAME "%d triggered\n", GPIO_INT_PIN);
}

static struct gpio_callback gpio_cb;

void main(void)
{
        struct device *gpio_dev;
        int ret;
        int toggle = 1;

        gpio_dev = device_get_binding(GPIO_DRV_NAME);
        if (!gpio_dev) {
                printk("Cannot find %s!\n", GPIO_DRV_NAME);
                return;
        }

        /* Setup GPIO output */
        ret = gpio_pin_configure(gpio_dev, GPIO_OUT_PIN, (GPIO_
↪DIR_OUT));
        if (ret) {
                printk("Error configuring " GPIO_NAME "%d!\n",␣
↪GPIO_OUT_PIN);
        }

        /* Setup GPIO input, and triggers on rising edge. */
        ret = gpio_pin_configure(gpio_dev, GPIO_INT_PIN,
                                  (GPIO_DIR_IN | GPIO_INT |
                                   GPIO_INT_EDGE | GPIO_INT_
↪ACTIVE_HIGH |
                                   GPIO_INT_DEBOUNCE));
        if (ret) {
                printk("Error configuring " GPIO_NAME "%d!\n",␣
↪GPIO_INT_PIN);
        }

        gpio_init_callback(&gpio_cb, gpio_callback, BIT(GPIO_INT_
↪PIN));

        ret = gpio_add_callback(gpio_dev, &gpio_cb);
        if (ret) {
                printk("Cannot setup callback!\n");
        }

        ret = gpio_pin_enable_callback(gpio_dev, GPIO_INT_PIN);
        if (ret) {
                printk("Error enabling callback!\n");
        }
```

(continues on next page)

(续上页)

```
        while (1) {
                printk("Toggling " GPIO_NAME "%d\n", GPIO_OUT_
↪PIN);

                ret = gpio_pin_write(gpio_dev, GPIO_OUT_PIN,␣
↪toggle);
                if (ret) {
                        printk("Error set " GPIO_NAME "%d!\n",␣
↪GPIO_OUT_PIN);
                }

                if (toggle) {
                        toggle = 0;
                } else {
                        toggle = 1;
                }

                k_sleep(MSEC_PER_SEC);
        }
}
```

CHAPTER 4

I2C 驱动程序

4.1 I2C 总线

I2C 总线是由 Philips 在 1982 年提出的通用串行传输协议。常用来连接一些低速外设，比如 E2PROM、传感器等。传输时钟频率速度一般在 100K ～ 400KHz。

I2C 总线支持多 slave 连接，每个设备都要有唯一的地址。

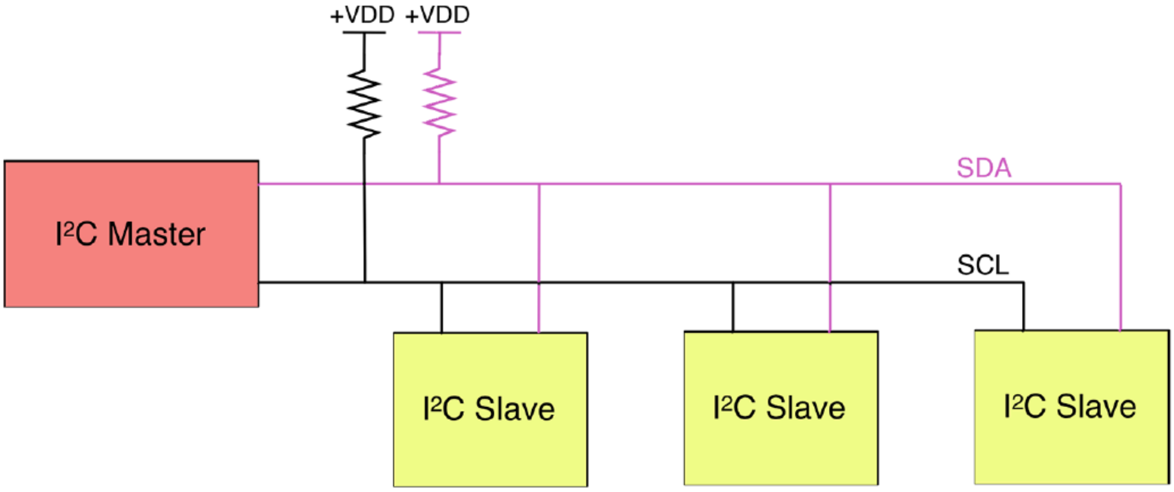

图 4.1: I2C 总线硬件连接

I2C 总线工作模式是半双工，主机每发送一个 byte，从机在第 9 个 cycle 要拉低一下 SDA 线，回应一个 ACK。如果没有回应主机就认为从机通信有问题。只有主机发送的最后一个 byte，从机可以不回应 ACK。

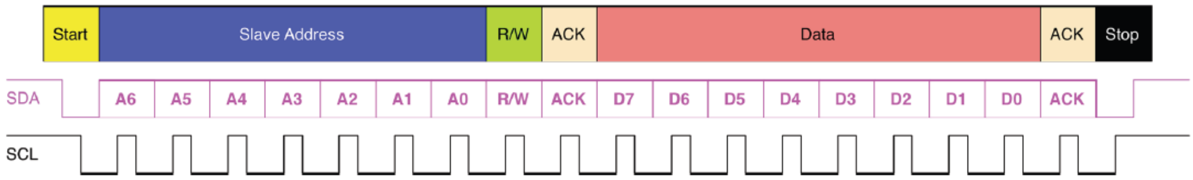

图 4.2: I2C 总线波形

4.2 I2C 控制器

ATB110X 带有两路独立的 I2C 总线控制器：

- 支持 master 和 slave 两种模式
- 支持 100KHz 和 400KHz 两种时钟速率。
- 只支持 7bit 设备地址
- 内建 15K 上拉电阻
- 8 层 RX FIFO 和 8 层 TXFIFO

4.3 驱动接口说明

下面介绍一下 I2C 的具体接口和参数说明

Public APIs for the I2C drivers.

Defines

I2C_SPEED_STANDARD
    I2C Standard Speed

I2C_SPEED_FAST
    I2C Fast Speed

I2C_SPEED_FAST_PLUS
    I2C Fast Plus Speed

I2C_SPEED_HIGH
    I2C High Speed

I2C_SPEED_ULTRA
    I2C Ultra Fast Speed

I2C_ADDR_10_BITS
    Use 10-bit addressing.

I2C_MODE_MASTER
    Controller to act as Master.

I2C_MSG_WRITE
Write message to I2C bus.

I2C_MSG_READ
Read message from I2C bus.

I2C_MSG_STOP
Send STOP after this message.

I2C_MSG_RESTART
RESTART I2C transaction for this message.

I2C_DECLARE_CLIENT_CONFIG

I2C_CLIENT(_master, _addr)

I2C_GET_MASTER(_conf)

I2C_GET_ADDR(_conf)

Functions

static int **i2c_configure**(**struct** device **dev*, u32_t *dev_config*)
Configure operation of a host controller.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- *`dev_config`*: Bit-packed 32-bit value to the device runtime configuration for the I2C controller.

Return Value

- `0`: If successful.
- `-EIO`: General input / output error, failed to configure device.

static int **i2c_write**(**struct** device **dev*, u8_t **buf*, u32_t *num_bytes*, u16_t *addr*)
Write a set amount of data to an I2C device.

This routine writes a set amount of data synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `buf`: Memory pool from which the data is transferred.
- `num_bytes`: Number of bytes to write.
- `addr`: Address to the target I2C device for writing.

Return Value

- `0`: If successful.

- `-EIO`: General input / output error.

static int **i2c_read(struct** device **dev*, u8_t **buf*, u32_t *num_bytes*, u16_t *addr***)**

Read a set amount of data from an I2C device.

This routine reads a set amount of data synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `buf`: Memory pool that stores the retrieved data.
- `num_bytes`: Number of bytes to read.
- `addr`: Address of the I2C device being read.

Return Value

- `0`: If successful.
- `-EIO`: General input / output error.

static int **i2c_transfer(struct** device **dev*, **struct** *i2c_msg* **msgs*, u8_t *num_msgs*, u16_t *addr***)**

Perform data transfer to another I2C device.

This routine provides a generic interface to perform data transfer to another I2C device synchronously. Use i2c_read()/i2c_write() for simple read or write.

The array of message *msgs* must not be NULL. The number of message *num_msgs* may be zero,in which case no transfer occurs.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `msgs`: Array of messages to transfer.
- `num_msgs`: Number of messages to transfer.
- `addr`: Address of the I2C target device.

Return Value

- `0`: If successful.
- `-EIO`: General input / output error.

static int **i2c_burst_read(struct** device **dev*, u16_t *dev_addr*, u8_t *start_addr*, u8_t **buf*, u8_t *num_bytes***)**

Read multiple bytes from an internal address of an I2C device.

This routine reads multiple bytes from an internal address of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for reading.
- `start_addr`: Internal address from which the data is being read.
- `buf`: Memory pool that stores the retrieved data.
- `num_bytes`: Number of bytes being read.

Return Value

- `0`: If successful.
- `-EIO`: General input / output error.

static int **i2c_burst_write(struct** device **dev*, u16_t *dev_addr*, u8_t *start_addr*, u8_t **buf*, u8_t *num_bytes*)

Write multiple bytes to an internal address of an I2C device.

This routine writes multiple bytes to an internal address of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for writing.
- `start_addr`: Internal address to which the data is being written.
- `buf`: Memory pool from which the data is transferred.
- `num_bytes`: Number of bytes being written.

Return Value

- `0`: If successful.
- `-EIO`: General input / output error.

static int **i2c_reg_read_byte(struct** device **dev*, u16_t *dev_addr*, u8_t *reg_addr*, u8_t **value*)

Read internal register of an I2C device.

This routine reads the value of an 8-bit internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for reading.
- `reg_addr`: Address of the internal register being read.
- `value`: Memory pool that stores the retrieved register value.

Return Value

- `0`: If successful.
- `-EIO`: General input / output error.

static int **i2c_reg_write_byte**(**struct** device **dev*, u16_t *dev_addr*, u8_t *reg_addr*, u8_t *value*)

Write internal register of an I2C device.

This routine writes a value to an 8-bit internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for writing.
- `reg_addr`: Address of the internal register being written.
- `value`: Value to be written to internal register.

Return Value

- `0`: If successful.
- `-EIO`: General input / output error.

static int **i2c_reg_update_byte**(**struct** device **dev*, u8_t *dev_addr*, u8_t *reg_addr*, u8_t *mask*, u8_t *value*)

Update internal register of an I2C device.

This routine updates the value of a set of bits from an 8-bit internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for updating.
- `reg_addr`: Address of the internal register being updated.
- `mask`: Bitmask for updating internal register.
- `value`: Value for updating internal register.

Return Value

- `0`: If successful.
- `-EIO`: General input / output error.

static int **i2c_burst_read16**(**struct** device **dev*, u16_t *dev_addr*, u16_t *start_addr*, u8_t **buf*, u8_t *num_bytes*)

Read multiple bytes from an internal 16 bit address of an I2C device.

This routine reads multiple bytes from a 16 bit internal address of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for reading.
- `start_addr`: Internal 16 bit address from which the data is being read.
- `buf`: Memory pool that stores the retrieved data.
- `num_bytes`: Number of bytes being read.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_burst_write16(struct** device **dev*, u16_t *dev_addr*, u16_t *start_addr*, u8_t **buf*, u8_t *num_bytes***)**

Write multiple bytes to a 16 bit internal address of an I2C device.

This routine writes multiple bytes to a 16 bit internal address of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for writing.
- `start_addr`: Internal 16 bit address to which the data is being written.
- `buf`: Memory pool from which the data is transferred.
- `num_bytes`: Number of bytes being written.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_reg_read16(struct** device **dev*, u16_t *dev_addr*, u16_t *reg_addr*, u8_t **value***)**

Read internal 16 bit address register of an I2C device.

This routine reads the value of an 16-bit internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for reading.
- `reg_addr`: 16 bit address of the internal register being read.
- `value`: Memory pool that stores the retrieved register value.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_reg_write16(struct** device **dev*, u16_t *dev_addr*, u16_t *reg_addr*, u8_t *value***)**

Write internal 16 bit address register of an I2C device.

This routine writes a value to an 16-bit internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for writing.
- `reg_addr`: 16 bit address of the internal register being written.
- `value`: Value to be written to internal register.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_reg_update16(struct** device **dev*, u16_t *dev_addr*, u16_t *reg_addr*, u8_t *mask*, u8_t *value***)**

Update internal 16 bit address register of an I2C device.

This routine updates the value of a set of bits from a 16-bit internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for updating.
- `reg_addr`: 16 bit address of the internal register being updated.
- `mask`: Bitmask for updating internal register.
- `value`: Value for updating internal register.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_burst_read_addr(struct** device **dev*, u16_t *dev_addr*, u8_t **start_addr*, **const** u8_t *addr_size*, u8_t **buf*, u8_t *num_bytes***)**

Read multiple bytes from an internal variable byte size address of an I2C device.

This routine reads multiple bytes from an addr_size byte internal address of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for reading.
- `start_addr`: Array to an internal register address from which the data is being read.
- `addr_size`: Size in bytes of the register address.
- `buf`: Memory pool that stores the retrieved data.
- `num_bytes`: Number of bytes being read.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_burst_write_addr(struct** device **dev*, u16_t *dev_addr*, u8_t **start_addr*, **const** u8_t *addr_size*, u8_t **buf*, u8_t *num_bytes***)**

Write multiple bytes to an internal variable bytes size address of an I2C device. This routine writes multiple bytes to an addr_size byte internal address of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for writing.
- `start_addr`: Array to an internal register address from which the data is being read.
- `addr_size`: Size in bytes of the register address.
- `buf`: Memory pool from which the data is transferred.
- `num_bytes`: Number of bytes being written.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_reg_read_addr(struct** device **dev*, u16_t *dev_addr*, u8_t **reg_addr*, **const** u8_t *addr_size*, u8_t **value***)**

Read internal variable byte size address register of an I2C device.

This routine reads the value of an addr_size byte internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for reading.
- `reg_addr`: Array to an internal register address from which the data is being read.
- `addr_size`: Size in bytes of the register address.
- `value`: Memory pool that stores the retrieved register value.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_reg_write_addr(struct** device **dev*, u16_t *dev_addr*, u8_t **reg_addr*, **const** u8_t *addr_size*, u8_t *value*)

Write internal variable byte size address register of an I2C device.

This routine writes a value to an addr_size byte internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for writing.
- `reg_addr`: Array to an internal register address from which the data is being read.
- `addr_size`: Size in bytes of the register address.
- `value`: Value to be written to internal register.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

static int **i2c_reg_update_addr(struct** device **dev*, u16_t *dev_addr*, u8_t **reg_addr*, u8_t *addr_size*, u8_t *mask*, u8_t *value*)

Update internal variable byte size address register of an I2C device.

This routine updates the value of a set of bits from a addr_size byte internal register of an I2C device synchronously.

Parameters

- `dev`: Pointer to the device structure for the driver instance.
- `dev_addr`: Address of the I2C device for updating.
- `reg_addr`: Array to an internal register address from which the data is being read.
- `addr_size`: Size in bytes of the register address.
- `mask`: Bitmask for updating internal register.
- `value`: Value for updating internal register.

Return Value

- `0`: If successful.
- `Negative`: errno code if failure.

`struct i2c_msg`

#include <i2c.h> One I2C Message.

This defines one I2C message to transact on the I2C bus.

Public Members

u8_t ***buf***

Data buffer in bytes

u32_t **len**

Length of buffer in bytes

u8_t **flags**

Flags for this message

`union dev_config`

Public Members

u32_t **raw**

struct *dev_config::__bits* **bits**

struct __bits

Public Members

u32_t **use_10_bit_addr**

u32_t **speed**

u32_t **is_master_device**

u32_t **reserved**

struct i2c_client_config

Public Members

char ***i2c_master**

u16_t **i2c_addr**

4.4 驱动使用示例

下面以一个通过 I2C 接口读写 E2PROM 示例来介绍 I2C 接口的具体使用。

```
#include <i2c.h>

#define E2PROM_I2C_MASTER_NAME  CONFIG_I2C_0_NAME
#define E2PROM_I2C_ADDRESS      0x50

static const union dev_config i2c_cfg = {
        .raw = 0,
        .bits = {
                .use_10_bit_addr = 0,
                .is_master_device = 1,
                .speed = I2C_SPEED_STANDARD,
        },
};

int i2c_e2prom_test(void)
{
        struct device *dev;
        int err;
        u16_t offset;
        u8_t wdata, rdata;

        dev = device_get_binding(E2PROM_I2C_MASTER_NAME);
        if (!dev) {
                printk("Cannot get I2C device");
                return -ENODEV;
        }

        err = i2c_configure(dev, i2c_cfg.raw);
        if (err) {
                printk("I2C config failed");
                return -EIO;
        }
```

(continues on next page)

(续上页)

```
        offset = 0;
        wdata = 0x55;
        rdata = 0x0;

        /* write a byte to E2PROM */
        err = i2c_burst_write16(dev, E2PROM_I2C_ADDRESS, offset,␣
↪&wdata, 1);
        if (err) {
                return -EIO;
        }

        /* waiting for writing to complete */
        k_sleep(10);

        /* read a byte from E2PROM */
        err = i2c_burst_read16(dev, E2PROM_I2C_ADDRESS, offset, &
↪rdata, 1);
        if (err) {
                return -EIO;
        }

        if (rdata != rdata) {
                printk("E2PROM Data compare error\n");
                return -EIO;
        }

        printk("E2PROM test pass!");

        return 0;
}
```

CHAPTER 5

SPI 驱动程序

5.1 SPI 总线

SPI 总线是同步串行总线接口，速度可以超过 50M。常用来连接 NOR Flash、LCD 等高速设备。SPI 可以全双工传输数据，同时进行数据的收发。

SPI 总线定义了 4 个逻辑信号：（1）MOSI – SPI 总线主机输出/ 从机输入（Master Output/Slave Input）（2）MISO – SPI 总线主机输入/ 从机输出（Master Input/Slave Output）（3）SCLK – 时钟信号，由主设备产生（4）CS – 从设备使能信号，由主设备控制（Chip select）

SPI 总线支持多 slave 连接，通过片选来选择当前读写的 slave 设备.

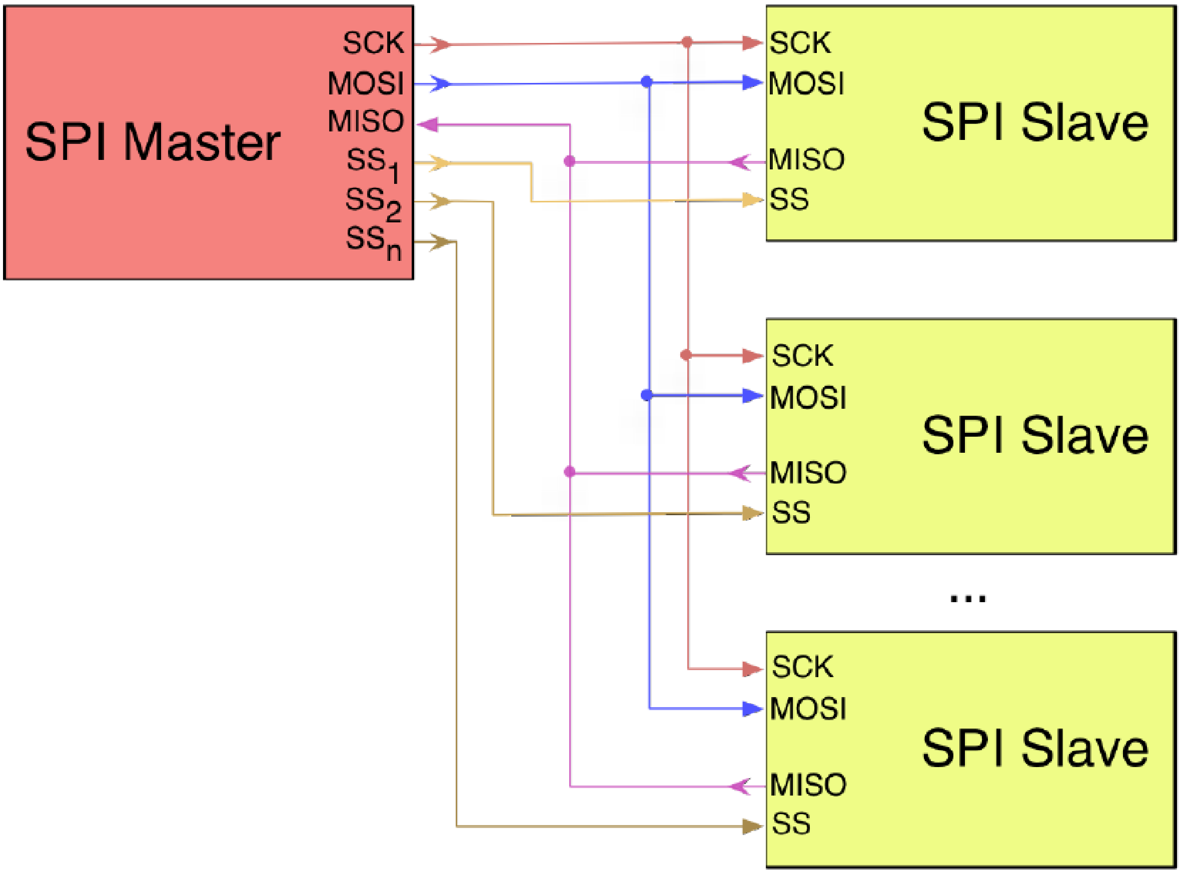

图 5.1: SPI 总线硬件连接

SPI 的 mode：

CPOL 表示时钟极性, 即空闲状态时的电平，低电平（0）或高电平（1）CPOH 表示数据采样的时钟沿，第一个沿（0）或第二个沿（1）

根据 SPI 的相位 (CPHA) 和极性 (CPOL)，对应的 4 种组合构成了 SPI 的 4 种工作模式：

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

图 5.2: SPI 模式

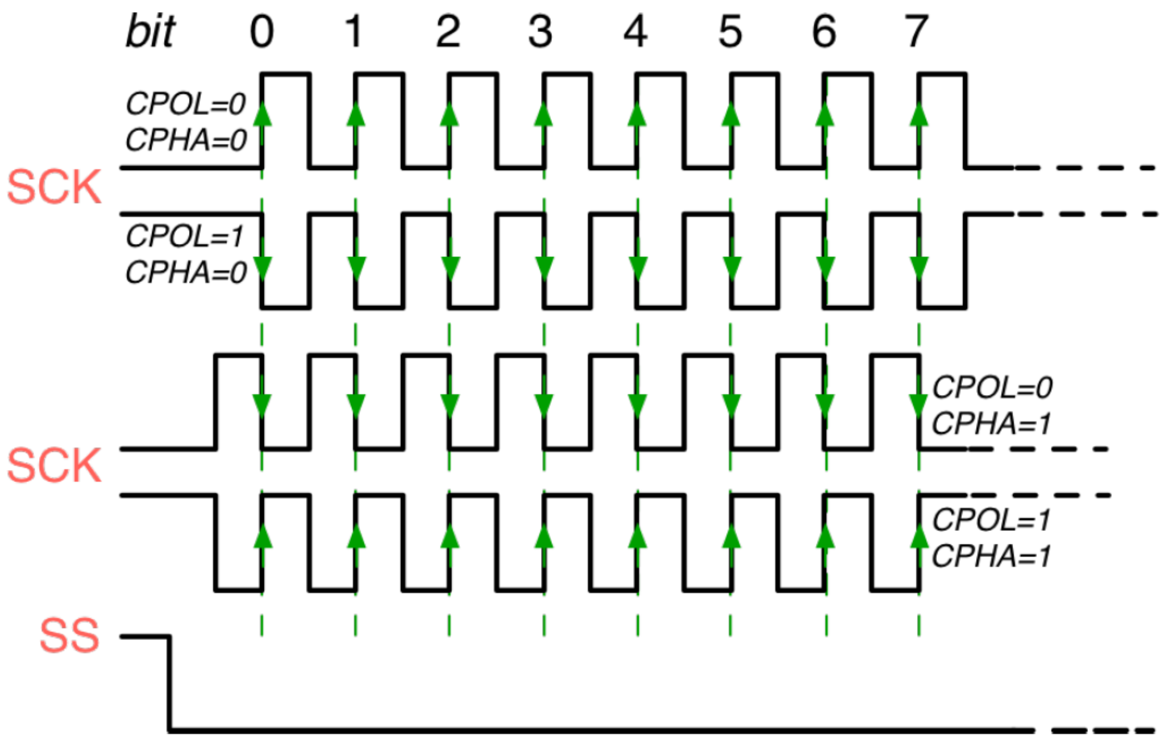

图 5.3: SPI 波形对应的波形

另外一个需要注意的是 SPI 每次读写的最小字长。一般有 8、16 和 32bit 几种配置。用户要根据具体 slave 设备需求来选用。

5.2 SPI 控制器

ATB110X 带有 3 路独立的 SPI 总线控制器，SPI0 一般接 SPI NOR，用于存储程序。SPI1 和 SPI2 是通用 SPI 接口，可用来接其它 SPI 接口的外设。

- 支持 master 和 slave 两种模式
- 支持 mode:0/1/2/3
- 只支持 4 线模式

5.3 驱动接口说明

下面介绍一下 SPI 的具体接口和参数说明

Public API for SPI drivers and applications.

Defines

SPI_OP_MODE_MASTER
SPI operational mode.

SPI_OP_MODE_SLAVE

SPI_OP_MODE_MASK

SPI_OP_MODE_GET(_operation_**)**

SPI_MODE_CPOL
SPI Polarity & Phase Modes.

Clock Polarity: if set, clock idle state will be 1 and active state will be 0. If untouched, the inverse will be true which is the default.

SPI_MODE_CPHA
Clock Phase: this dictates when is the data captured, and depends clock’s polarity. When SPI_MODE_CPOL is set and this bit as well, capture will occur on low to high transition and high to low if this bit is not set (default). This is fully reversed if CPOL is not set.

SPI_MODE_LOOP
Whatever data is transmitted is looped-back to the receiving buffer of the controller. This is fully controller dependent as some may not support this, and can be used for testing purposes only.

SPI_MODE_MASK

SPI_MODE_GET(_mode_**)**

SPI_TRANSFER_MSB
SPI Transfer modes (host controller dependent)

SPI_TRANSFER_LSB

SPI_WORD_SIZE_SHIFT
SPI word size.

SPI_WORD_SIZE_MASK

SPI_WORD_SIZE_GET(_operation_**)**

SPI_WORD_SET(_word_size_**)**

SPI_LINES_SINGLE
SPI MISO lines.

Some controllers support dual or quad MISO lines connected to slaves. Default is single, which is the case most of the time.

SPI_LINES_DUAL

SPI_LINES_QUAD

SPI_LINES_MASK

SPI_HOLD_ON_CS
Specific SPI devices control bits.

SPI_LOCK_ON

SPI_EEPROM_MODE

Typedefs

typedef spi_api_io
Callback API for I/O See spi_transceive() for argument descriptions.

Callback API for asynchronous I/O See spi_transceive_async() for argument descriptions.

typedef int (***spi_api_io_async**)(**struct** *spi_config* *config, **const struct** *spi_buf* *tx_bufs, size_t tx_count, **struct** *spi_buf* *rx_bufs, size_t rx_count, **struct** k_poll_signal *async)

typedef spi_api_release
Callback API for unlocking SPI device. See spi_release() for argument descriptions.

Functions

static int **spi_transceive**(**struct** *spi_config* **config*, **const struct** *spi_buf* **tx_bufs*, size_t *tx_count*, **struct** *spi_buf* **rx_bufs*, size_t *rx_count*)
Read/write the specified amount of data from the SPI driver.

Note: This function is synchronous.

Parameters

- `config`: Pointer to a valid *spi_config* structure instance.
- `tx_bufs`: Buffer array where data to be sent originates from, or NULL if none.
- `tx_count`: Number of element in the tx_bufs array.
- `rx_bufs`: Buffer array where data to be read will be written to, or NULL if none.
- `rx_count`: Number of element in the rx_bufs array.

Return Value

- `0`: If successful, negative errno code otherwise.

static int **spi_read**(**struct** *spi_config* **config*, **struct** *spi_buf* **rx_bufs*, size_t *rx_count*)
Read the specified amount of data from the SPI driver.

Note: This function is synchronous.

Parameters

- `config`: Pointer to a valid *spi_config* structure instance.
- `rx_bufs`: Buffer array where data to be read will be written to.
- `rx_count`: Number of element in the rx_bufs array.

Return Value

- `0`: If successful, negative errno code otherwise.

static int **spi_write**(**struct** *spi_config* **config*, **const struct** *spi_buf* **tx_bufs*, size_t *tx_count*)

Write the specified amount of data from the SPI driver.

Note: This function is synchronous.

Parameters

- `config`: Pointer to a valid *spi_config* structure instance.
- `tx_bufs`: Buffer array where data to be sent originates from.
- `tx_count`: Number of element in the tx_bufs array.

Return Value

- `0`: If successful, negative errno code otherwise.

static int **spi_release**(**struct** *spi_config* **config*)

Release the SPI device locked on by the current config.

Note: This synchronous function is used to release the lock on the SPI device that was kept if, and if only, given config parameter was the last one to be used (in any of the above functions) and if it has the SPI_LOCK_ON bit set into its operation bits field. This can be used if the caller needs to keep its hand on the SPI device for consecutive transactions.

Parameters

- `config`: Pointer to a valid *spi_config* structure instance.

struct spi_cs_control

#include <spi.h> SPI Chip Select control structure.

This can be used to control a CS line via a GPIO line, instead of using the controller inner CS logic.

gpio_dev is a valid pointer to an actual GPIO device gpio_pin is a number representing the gpio PIN that will be used to act as a CS line delay is a delay in microseconds to wait before starting the transmission and before releasing the CS line

Public Members

struct device ***gpio_dev**

u32_t **gpio_pin**

u32_t **delay**

struct spi_config

#include <spi.h> SPI controller configuration structure.

dev is a valid pointer to an actual SPI device frequency is the bus frequency in Hertz operation is a bit field with the following parts: operational mode [0] - master or slave. mode [1 : 3] - Polarity, phase and loop mode. transfer [4] - LSB or MSB first. word_size [5 : 10] - Size of a data frame in bits. lines [11 : 12] - MISO lines: Single/Dual/Quad. cs_hold [13] - Hold on the CS line if possible. lock_on [14] - Keep resource locked for the caller. eeprom [15] - EEPROM mode. vendor is a vendor specific bitfield slave is the slave number from 0 to host controller slave limit.

cs is a valid pointer on a struct *spi_cs_control* is CS line is emulated through a gpio line, or NULL otherwise.

Note: cs_hold, lock_on and eeprom_rx can be changed between consecutive transceive call.

Public Members

struct device ***dev**

u32_t **frequency**

u16_t **operation**

u16_t **vendor**

u16_t **slave**

struct *spi_cs_control* ***cs**

struct spi_buf

#include <spi.h> SPI buffer structure.

buf is a valid pointer on a data buffer, or NULL otherwise. len is the length of the buffer or, if buf is NULL, will be the length which as to be sent as dummy bytes (as TX buffer) or the length of bytes that should be skipped (as RX buffer).

Public Members

void ***buf**

size_t **len**

struct spi_driver_api

#include <spi.h> SPI driver API This is the mandatory API any SPI driver needs to expose.

Public Members

spi_api_io **transceive**

spi_api_release **release**

5.4 驱动使用示例

下面是一个通过 spi 接口来进行数据交互的示例。IC 的两组 SPI 接口连接在一起, 一组做 master, 一组做 slave，进行数据交互传输。

```
#include <spi.h>

static struct spi_config spi_master_conf = {
        .frequency = 2000000,
        .operation = (SPI_OP_MODE_MASTER | SPI_TRANSFER_MSB |␣
↪SPI_WORD_SET(8) |
                      SPI_LINES_SINGLE | SPI_MODE_CPOL | SPI_
↪MODE_CPHA),
        .vendor    = 0,
        .slave     = 0,
        .cs        = NULL,
};

static struct spi_config spi_slave_conf = {
        .frequency = 2000000,
        .operation = (SPI_OP_MODE_SLAVE | SPI_TRANSFER_MSB | SPI_
↪WORD_SET(8) |
                      SPI_LINES_SINGLE | SPI_MODE_CPOL | SPI_
↪MODE_CPHA),
        .vendor    = 0,
        .slave     = 0,
        .cs        = NULL,
};

#define SPI_MASTER_NAME         "spi1"
#define SPI_SLAVER_NAME         "spi2"
#define STACK_SIZE              1024

static K_THREAD_STACK_DEFINE(spi_slave_stack, STACK_SIZE);
static struct k_thread spi_slave_thread;

static K_THREAD_STACK_DEFINE(spi_master_stack, STACK_SIZE);
```

(continues on next page)

(续上页)

```
static struct k_thread spi_master_thread;

static int spi_slave_func(void)
{
        struct device *spi_slave_dev;
        u8_t buf[4] = {0x81, 0x82, 0x83, 0x84};
        struct spi_buf spi_bufs = {buf, 4};

        spi_slave_dev = device_get_binding(SPI_MASTER_NAME);
        if (!spi_slave_dev) {
                printk("SPI master driver was not found!\n");
                return;
        }

        spi_slave_conf.dev = spi_slave_dev;

        printk("slave send:\n");
        print_buffer(buf, 1, sizeof(buf), 16, 0);

        if (spi_transceive(&spi_slave_conf, &spi_bufs, 1,
                                   &spi_bufs, 1) != 0) {
                printk("slave io error\n");
                return -EIO;
        }

        printk("slave receive:\n");
        print_buffer(buf, 1, sizeof(buf), 16, 0);

        return 0;
}

static int spi_master_func(void)
{
        struct device *spi_master_dev;
        u8_t buf[4] = {1, 2, 3, 4};
        struct spi_buf spi_bufs = {buf, 4};

        spi_master_dev = device_get_binding(SPI_SLAVER_NAME);
        if (!spi_master_dev) {
                printf("SPI master driver was not found!\n");
                return;
        }

        spi_master_conf.dev = spi_master_dev;

        printk("master send:\n");
        print_buffer(buf, 1, sizeof(buf), 16, 0);

        if (spi_transceive(&spi_master_conf, &spi_bufs, 1,
```

(continues on next page)

(续上页)

```
                                &spi_bufs, 1) != 0) {
                        printk("master io error\n");
                        return -EIO;
                }

                printk("master receive:\n");
                print_buffer(buf, 1, sizeof(buf), 16, 0);

                return 0;
}

int test_spi_slave(void)
{
        printk("test spi master/slave\n");

        k_thread_create(&spi_master_thread, spi_master_stack,␣
↪STACK_SIZE,
                        (k_thread_entry_t) spi_master_func,
                        NULL, NULL, NULL, K_PRIO_PREEMPT(8), 0,␣
↪0);

        k_thread_create(&spi_slave_thread, spi_slave_stack,␣
↪STACK_SIZE,
                        (k_thread_entry_t) spi_slave_func,
                        NULL, NULL, NULL, K_PRIO_PREEMPT(7), 0,␣
↪0);

}
```

CHAPTER 6

NVRAM config 驱动程序

6.1 NVRAM

NVRAM 是一块掉电后数据仍然保持的存储设备，可以是电池供电的 RAM 设备，也可以是 NOR/NAND Flash 这种非易失存储设备。常用来永久保存用户配置信息和数据等内容。

NVRAM 配置分为两种：一种是出厂预置的配置，放在 factory 区域；一种是用户动态写入的配置，放在 user 区域。在读取一个指定配置项时，默认会先从 user 区查找读取用户写入的配置，如果不存在则去 factory 区域去读取。

6.2 驱动接口说明

下面介绍一下 NVRAM config 的具体接口和参数说明

NVRAM config driver interface.

Functions

int **nvram_config_get(const** char **name*, void **data*, int *max_len*)

Read config from NVRAM user region.

Parameters

- `name`: Config name
- `data`: Pointer to data buffer

- `max_len`: Maximum number of bytes to be read

Return Value

- `0`: If successful, negative errno code otherwise.

int **nvram_config_set**(**const** char **name*, **const** void **data*, int *len*)

Write config to NVRAM user region.

Parameters

- `name`: Config name
- `data`: Pointer to data buffer
- `len`: number of bytes to be write

Return Value

- `0`: If successful, negative errno code otherwise.

int **nvram_config_clear_all**(void)

Write config to NVRAM.

Return Value

- `0`: If successful, negative errno code otherwise.

void **nvram_config_dump**(void)

print all configs in NVRAM

int **nvram_config_get_factory**(**const** char **name*, void **data*, int *max_len*)

Read config from NVRAM factory region.

Parameters

- `name`: Config name
- `data`: Pointer to data buffer
- `max_len`: Maximum number of bytes to be read

Return Value

- `0`: If successful, negative errno code otherwise.

int **nvram_config_set_factory**(**const** char **name*, **const** void **data*, int *len*)

Write config to NVRAM factory region.

Parameters

- `name`: Config name
- `data`: Pointer to data buffer
- `len`: number of bytes to be write

Return Value

- `0`: If successful, negative errno code otherwise.

6.3 驱动使用示例

下面以一个示例介绍一下 NVRAM config 接口的使用。

```
#include <nvram_config.h>

static unsigned char test_nvram_buf[256];

int test_nvram(void)
{
        int data_len, i;

        printk("\nNVRAM testing\n");
        printk("==========================\n");

        nvram_config_set("ucfg", "udata", 6);
        nvram_config_set_factory("fcfg", "fdata", 6);

        data_len = nvram_config_get("ucfg", test_nvram_buf,
↪sizeof(test_nvram_buf));
        if (data_len < 0) {
                printk("cannot find config ucfg\n");
                return -EIO;
        }

        printk("nvram user: ucfg: %s\n", test_nvram_buf);

        data_len = nvram_config_get_factory("fcfg", test_nvram_
↪buf, sizeof(test_nvram_buf));
        if (data_len < 0) {
                printk("cannot find config fcfg\n", name);
                return -EIO;
        }

        printk("nvram user: fcfg: %s\n", test_nvram_buf);

        nvram_config_dump();
}
```

List of Figures

List of Tables