HAL_V2 SPI

开发指南

版本号: 1.0

发布日期: 2025.07.24

版本历史

版本号	日期	制/修订人	内容描述
1.0	2025.07.24	AWA2215	初始版本

目　录

1 前言

1.1 文档简介

介绍 HAL_V2 中 SPI 驱动的接口及使用方法，为 GPIO 使用者提供参考。

1.2 目标读者

SPI 驱动的开发/维护人员。

1.3 适用范围

表 1-1: 适用产品列表

产品名称	内核版本	驱动文件
T536	FreeRTOS	hal_spi.c
T153	FreeRTOS	hal_spi.c

1.4 文档约定

1 标志说明

⚠ 注意

- **提醒操作中应注意的事项。不当的操作可能会损坏器件，影响可靠性、降低性能等。**

说明

为准确理解文中指令、正确实施操作而提供的补充或强调信息。

技巧

一些容易忽视的小功能、技巧。了解这些功能或技巧能帮助解决特定问题或者节省操作时间。

1.5 相关术语介绍

术语	解释说明
Sunxi	指 Allwinner 的一系列 SOC 硬件平台
SPI	Serial Peripheral Interface，同步串行外设接口
主模式 SPI	Master SPI
SPI Slave	SPI 从模式

2 模块介绍

2.1 模块配置介绍

menuconfig 配置项

```
Drivers V2 Config  --->

    [*] enable spi driver
    [ ]  enable spi hal APIs test command
```

2.2 模块源码结构

1. 驱动源码

```
hal_v2/hal/source/spi/
├──── common_spi.h
├──── hal_spi.c

├──── Makefile
├──── platform
│    ├──── spi_sun55iw6.h
│    └──── spi_sun8iw22.h
└──── platform_spi.h
```

2. 驱动 APIs 声明头文件

```
hal_v2/include/hal/
└──── hal_spi.h
```

```
hal_v2/hal/examples/spi/
├──── Makefile
├──── spi_master_cpu_loopback
│    ├──── Makefile
│    └──── spi_master_cpu_loopback.c
└──── spi_master_dma_loopback
   ├──── Makefile
   └──── spi_master_dma_loopback.c
```

3 模块接口说明

3.1 重要结构体及宏定义

3.1.1 SPI 模式功能选择

```
#define SPI_CPHA BIT(0) /* clock phase */
#define SPI_CPOL      BIT(1)  /* clock polarity */

#define SPI_MODE_0     (0|0)
#define SPI_MODE_1     (0|SPI_CPHA)
#define SPI_MODE_2     (SPI_CPOL|0)
#define SPI_MODE_3     (SPI_CPOL|SPI_CPHA)

#define SPI_CS_HIGH    BIT(2)  /* chipselect active high? */
#define SPI_LSB_FIRST   BIT(3)  /* per-word bits-on-wire */
#define SPI_3WIRE      BIT(4)  /* SI/SO signals shared */
#define SPI_LOOP       BIT(5)  /* loopback mode */
#define SPI_NO_CS      BIT(6)  /* 1 dev/bus, no chipselect */
#define SPI_READY      BIT(7)  /* slave pulls low to pause */
#define SPI_TX_DUAL    BIT(8)  /* transmit with 2 wires */
e SPI_TX_QUAD BIT(9) /* transmit with 4 wires */
#define SPI_RX_DUAL    BIT(10) /* receive with 2 wires */
#define SPI_RX_QUAD    BIT(11) /* receive with 4 wires */
```

- SPI_MODE_0/1/2/3：设置 SPI 的传输模式。
- SPI_CS_HIGH：设置 CS 片选是否为高电平有效。
- SPI_LSB_FIRST：设置发送顺序是低位在前。
- SPI_3WIRE：设置 SPI 工作在 3 线模式下，及 MOSI 即用作输入也用作输入，实现半双工通信

3.1.2 SPI 控制器模式配置

```
typedef enum
{
   HAL_SPI_BUS_MASTER = 0,
   HAL_SPI_BUS_SLAVE = 1,
   HAL_SPI_BUS_BIT = 2,
} hal_spi_master_bus_mode_t;
```

- HAL_SPI_BUS_MASTER：处于 Master 模式，外接 SPI Device。
- HAL_SPI_BUS_SLAVE：处于 Slave 模式，被其他 Master 访问。
- HAL_SPI_BUS_BIT：处于 BIT 模式，使用 3Wire 方式进行数据传输。

3.1.3 SPI 控制器片选模式

```
typedef enum
{
  HAL_SPI_CS_AUTO = 0,
  HAL_SPI_CS_SOFT = 1,
} hal_spi_master_cs_mode_t;
```

- HAL_SPI_CS_AUTO：硬件自动控制，不需要驱动或软件介入。
- HAL_SPI_CS_SOFT：软件手动控制，由驱动完成相关操作。

4 SPI 控制器采样模式

```
typedef enum
{
  SUNXI_SPI_SAMP_MODE_OLD = 0,
  SUNXI_SPI_SAMP_MODE_NEW = 1,
} hal_spi_master_bus_sample_mode_t;
```

- SUNXI_SPI_SAMP_MODE_OLD：粗调模式，共有 3 档可调
- SUNXI_SPI_SAMP_MODE_NEW：细调模式，共有 7 档可调

说明

调模式为驱动根据时钟频率自动识别，不需要额外配置。

```
typedef enum
{
  SUNXI_SPI_SAMP_DELAY_CYCLE_0_0 = 0,
  SUNXI_SPI_SAMP_DELAY_CYCLE_0_5 = 1,
  SUNXI_SPI_SAMP_DELAY_CYCLE_1_0 = 2,
  SUNXI_SPI_SAMP_DELAY_CYCLE_1_5 = 3,
  SUNXI_SPI_SAMP_DELAY_CYCLE_2_0 = 4,
  SUNXI_SPI_SAMP_DELAY_CYCLE_2_5 = 5,
  SUNXI_SPI_SAMP_DELAY_CYCLE_3_0 = 6,
} hal_spi_master_spi_sample_mode_t;
```

NXI_SPI_SAMP_DELAY_CYCLE_0_0 采样延时调节档位选择。

说明

当采样模式处于细调时，才会使用到该参数。

3.1.5 SPI 控制器配置结构体

```
typedef struct
{
   hal_spi_master_bus_mode_t bus_mode; // SPI控制器配置
   hal_spi_master_cs_mode_t cs_mode;  // SPI控制器片选模式
   hal_spi_master_bus_sample_mode_t bus_sample_mode;  // SPI控制器采样模式 - 粗调
   hal_spi_master_spi_sample_mode_t spi_sample_mode;  // SPI控制器采样模式 - 细调
   uint32_t spi_sample_delay;  // SPI控制器细调采样延时
   uint8_t chipselect; /* SPI slave device selection */
   uint32_t clock_frequency; /* SPI master clock frequency setting */
   uint32_t mode;  // SPI模式/功能选择
   bool sip;
   bool flash;
} spi_init_t;
```

3.1.6 SPI 传输结构体

```
typedef struct
{
   const uint8_t *tx_buf;  /* Data buffer to send */
   uint32_t tx_len;      /* The total number of bytes to send */
   uint32_t tx_single_len; /* The number of bytes to send in single mode */
   uint8_t *rx_buf;      /* Received data buffer, */
   uint32_t rx_len;      /* The valid number of bytes received */
   uint8_t tx_nbits : 3;   /* Data buffer to send in nbits mode */
   uint8_t rx_nbits : 3;   /* Data buffer to received in nbits mode */
   uint8_t dummy_byte;     /* Flash send dummy byte, default 0*/
#define SPI_NBITS_SINGLE 0x01 /* 1bit transfer */
#define SPI_NBITS_DUAL     0x02 /* 2bits transfer */
#define SPI_NBITS_QUAD     0x04 /* 4bits transfer */
   uint8_t bits_per_word;  /* transfer bit_per_word */
} hal_spi_master_transfer_t;
```

3.2 对外提供的 API 接口

3.2.1 hal_spi_init

- 原型: hal_spi_master_status_t hal_spi_init(int port, spi_init_t*cfg)
- 作用：SPI 模块初始化，主要申请中断、pinctrl 初始化、clk 初始化、SPI 模块，包括 SPI 总线最大传输速率、片选模式等等。
- 参数：
  - port:SPI 端口
  - cfg: 配置信息
- 返回：
  - 0: 成功

- 负数: 失败

3.2.2 hal_spi_deinit

- 原型: hal_spi_master_status_t hal_spi_deinit(int port)
- 作用：SPI 模块去初始化。
- 参数：
  - port:SPI 端口号
- 返回：

0: 成功
负数: 失败

3.2.3 hal_spi_write

- 原型: hal_spi_master_status_t hal_spi_write(int port, const uint8_t *buf, uint32_t size)
- 作用：发送数据，调 hal_spi_xfer 接口。
- 参数：
  - port:SPI 端口号
  - buf: 发送数据

size: 发送数据大小

- 返回：
  - 0: 成功
  - 负数: 失败

3.2.4 hal_spi_read

- 原型: hal_spi_master_status_t hal_spi_read(int port, uint8_t *buf, uint32_t size)
- 作用：接收数据，调 hal_spi_xfer 接口。

：

  - port:SPI 端口号
  - buf: 接收数据
  - size: 接收数据大小
- 返回：
  - 0: 成功
  - 负数: 失败

3.2.5 hal_spi_write_then_read

- 原型: hal_spi_master_status_t hal_spi_write_then_read(int port, void *tbuf, uint32_t tlen, void *rbuf, uint32_t rlen)
- 作用：先写数据然后紧接着进行读操作，调 hal_spi_xfer 接口。
- 参数：
  - port:SPI 端口号
  - tbuf: 发送数据
  - tlen: 发送数据大小
  - rbuf: 接收数据
  - rlen: 接收数据大小
- 返回：
  - 0: 成功
  - 负数: 失败

3.2.6 hal_spi_xfer

- 原型: hal_spi_master_status_t hal_spi_xfer(int port, hal_spi_master_transfer_t *t, int num)
- 作用：发送或接收数据。
- 参数：
  - port:SPI 端口号
  - t: 指向传输包头的指针
  - num: 传输包的个数
- 返回：
  - 0: 成功
  - 负数: 失败

3.2.7 hal_spi_slave_abort

- 原型: hal_spi_master_status_t hal_spi_slave_abort(int port)
- 作用：终止 slave 模式传输。
- 参数：
  - port:SPI 端口号
- 返回：
  - 0: 成功
  - 负数: 失败

3.2.8 hal_spi_reset_txfifo

- 原型: void hal_spi_reset_txfifo(int port)
- 作用：复位 txfifo。
- 参数：
  - port:SPI 端口号
- 返回：
  - void

3.2.9 hal_spi_reset_rxfifo

- 原型: void hal_spi_reset_rxfifo(int port)
- 作用：复位 rxfifo。
- 参数：
  - port:SPI 端口号
- 返回：
  - void

3.2.10 hal_spi_enable_irq

- 原型: void hal_spi_enable_irq(int port, uint32_t bitmap)
- 作用：使能 spi 中断。
- 参数：
  - port:SPI 端口号
  - bitmap: 中断使能位
- 返回：
  - void

3.2.11 hal_spi_disable_irq

- 原型: void hal_spi_disable_irq(int port, uint32_t bitmap)
- 作用：关闭 spi 中断。
- 参数：
  - port:SPI 端口号

- bitmap: 中断使能位

- 返回：
  - void

3.2.12 hal_spi_clr_irq_pending

- 原型: void hal_spi_clr_irq_pending(int port, uint32_t pending_bit)
- 作用：清除 spi 中断 pending。
- 参数：
  - port:SPI 端口号
  - pending_bit:
- 返回：
  - void

3.2.13 hal_spi_enable_dma_irq

- 原型: void hal_spi_enable_dma_irq(int port, uint32_t bitmap)
- 作用：使能 spi drq。
- 参数：
  - port:SPI 端口号
  - bitmap:drq 使能位
- 返回：
  - void

3.2.14 hal_spi_disable_dma_irq

- 原型: void hal_spi_disable_dma_irq(int port, uint32_t bitmap)
- 作用：关闭 spi drq。
- 参数：
  - port:SPI 端口号
  - bitmap:drq 使能位
- 返回：
  - void

4 模块使用范例

可参考驱动 APIs 测试代码（hal_v2/hal/examples/spi/）, 基本 spi 操作方法如下：

- **步骤 1** 初始化 SPI 时钟
- **步骤 2** 初始化 SPI 引脚

骤 3 初始化 SPI 控制器

- **步骤 4** 注册 SPI 中断
- **步骤 5** 使能 SPI 中断
- **步骤 6** 发起 SPI 传输

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <hal_spi.h>
#include <hal_gpio.h>
#include <hal_clk.h>
#include <hal_reset.h>
de <hal_interrupt.h>


#if defined(CONFIG_KERNEL_FREERTOS)
#include <console.h>
#elif defined(CONFIG_KERNEL_BAREMETAL)
#include "shell.h"
#endif

static int port = 3;
#endif
static uint32_t mode;
static uint8_t bits = 8;
static uint32_t speed = 8000000;
static bool compelte = false;

static uint8_t default_tx[] = {
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0x40, 0x00, 0x00, 0x00, 0x00, 0x95,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
    0xF0, 0x0D,
};

static uint8_t default_rx[sizeof(default_tx)];

static unsigned long transfer(int port, uint8_t const *tx, uint8_t const *rx, size_t len)
{
```

```
    hal_spi_master_status_t ret = 0;

    hal_spi_master_transfer_t tr = {
        .tx_buf = (uint8_t *)tx,
        .tx_len = len,
        .rx_buf = (uint8_t *)rx,
        .rx_len = len,
        .tx_single_len = len,
        .dummy_byte = 0,
        .bits_per_word = bits,
    };

    mode |= SPI_LOOP;

    if (mode & SPI_TX_QUAD)
        tr.tx_nbits = 4;
    else if (mode & SPI_TX_DUAL)
tx_nbits = 2;

    else if (mode & SPI_RX_QUAD)
        tr.rx_nbits = 4;
    else if (mode & SPI_RX_DUAL)
        tr.rx_nbits = 2;
    if (!(mode & SPI_LOOP)) {
        if (mode & (SPI_TX_QUAD | SPI_TX_DUAL))
            tr.rx_buf = 0;
        else if (mode & (SPI_RX_QUAD | SPI_RX_DUAL))
            tr.tx_buf = 0;
        else if (mode & SPI_3WIRE)
            tr.rx_buf = 0;
    }

    /* tx_single_len is used for nor/nand flash case, spi test only support single/dual/quad */
    if (tr.rx_nbits > 1 || tr.tx_nbits > 1)
tx_single_len = 0;


    ret = hal_spi_xfer(port, &tr, 1);
    if (ret < 0) {
        printf("can't send spi message\n");
        return -1;
    }

    return 0;
}

static int spi_clk_init(void)
{
    int ret;
    hal_rst_t rst;
    rst_number_t rst_num;
    hal_clk_t clk_bus, clk, parent_clk;
    clk_number_t clk_bus_num, clk_num, parent_clk_num;

    rst_num = MAKE_RSTn(AW_SYS_CCU, RST_BUS_SPI3);
    clk_bus_num = MAKE_CLKn(AW_SYS_CCU, CLK_BUS_SPI3);

    /* clock source */
    clk_num = MAKE_CLKn(AW_SYS_CCU, CLK_SPI3);
    parent_clk_num = MAKE_CLKn(AW_SRC_CCU, CLK_SRC_HOSC);

    ret = hal_rst_get_by_rst_num(rst_num, &rst);
```

```
    if (ret) {
        printf("get reset handle failed, rst_num: %u, ret: %d\n", rst_num, ret);
        return -1;
    }

    ret = hal_n_clk_get_by_clk_num(clk_bus_num, &clk_bus);
    if (ret) {
        printf("get clk(%u) failed, ret: %d\n", clk_bus_num, ret);
        return -1;
    }

    ret = hal_n_clk_get_by_clk_num(clk_num, &clk);
    if (ret) {
        printf("get clk(%u) failed, ret: %d\n", clk_num, ret);
        printf("%d\n", __LINE__);
        return -1;
    }

    ret = hal_n_clk_get_by_clk_num(parent_clk_num, &parent_clk);
    if (ret) {
        printf("get clk(%u) failed, ret: %d\n", parent_clk_num, ret);
        printf("%d\n", __LINE__);
        return -1;
    }

    ret = hal_rst_deassert(rst);
    if (ret) {
        printf("deassert reset failed, rst_num: %u, ret: %d\n", rst_num, ret);
        ret = -2;
        goto exit_with_put_rst;
    }

    ret = hal_n_clk_enable(clk_bus);

        printf("enable clk(%u) failed, ret: %d\n", clk_bus_num, ret);
        ret = -2;
        goto exit_with_put_clk;
    }

    ret = hal_n_clk_set_parent(clk, parent_clk);
    if (ret) {
        printf("set the parent of clk(%u) to clk(%u) failed, ret: %d\n", clk_num, parent_clk_num, ret);
        ret = -3;
        goto exit_with_put_parent_clk;
    }

    ret = hal_n_clk_enable(clk);
    if (ret) {
        printf("enable clk(%u) failed, ret: %d\n", clk_num, ret);
        ret = -2;
        goto exit_with_put_clk;
    }

    ret = 0;

exit_with_put_parent_clk:
    hal_n_clk_put(parent_clk);

exit_with_put_clk:
    hal_n_clk_put(clk);
```

```
exit_with_put_rst:
    hal_rst_put(rst);
    return ret;
}

static void spi_pin_init(void)
{
    gpio_init_t gpio_initstruct;

    /* spi3 */
    gpio_initstruct.port = SUNXI_GPIO_B;
    gpio_initstruct.port_num = PIN_11 | PIN_12 | PIN_13 | PIN_14;
    gpio_initstruct.mul_sel = GPB_CFG0_SPI3;
    gpio_initstruct.drv_level = PIN_MULTI_DRIVE_2;

    hal_gpio_init(&gpio_initstruct);


static hal_irqreturn_t sunxi_spi_handler(void *dev)
{
    uint32_t status = 0, enable = 0, irq = 0;
    uint32_t base_addr = SPI_BASE(port);

    enable = hal_readl(base_addr + SUNXI_SPI_INT_CTL_REG) & SUNXI_SPI_INT_CTL_MASK;
    status = hal_readl(base_addr + SUNXI_SPI_INT_STA_REG) & SUNXI_SPI_INT_STA_MASK;

    /* clear irq pending */
    hal_writel(status, base_addr + SUNXI_SPI_INT_STA_REG);

    printf("spi irq handler enable(0x%x) status(0x%x)\n", enable, status);

    if ((enable & SUNXI_SPI_INT_CTL_SS_EN) && (status & SUNXI_SPI_INT_STA_SSI)) {
ntf("spi irq bus cs invalid detect\n");
        irq |= SUNXI_SPI_INT_CTL_SS_EN;
    }
    if ((enable & SUNXI_SPI_INT_CTL_TC_EN) && (status & SUNXI_SPI_INT_STA_TC)) {
        printf("spi irq bus tc comes\n");
        irq |= SUNXI_SPI_INT_CTL_TC_EN;
        compelte = true;
    }
    if ((enable & SUNXI_SPI_INT_CTL_TX_UDR_EN) && (status & SUNXI_SPI_INT_STA_TX_UDR)) {
        printf("spi irq bus txfifo underrun\n");
        irq |= SUNXI_SPI_INT_CTL_TX_UDR_EN;
        compelte = true;
    }
    if ((enable & SUNXI_SPI_INT_CTL_TX_OVF_EN) && (status & SUNXI_SPI_INT_STA_TX_OVF)) {
        printf("spi irq bus txfifo overflow\n");
        irq |= SUNXI_SPI_INT_CTL_TX_OVF_EN;
        hal_spi_reset_txfifo(port);
    }
    if ((enable & SUNXI_SPI_INT_CTL_RX_UDR_EN) && (status & SUNXI_SPI_INT_STA_RX_UDR)) {
        printf("spi irq bus rxfifo underrun\n");
        irq |= SUNXI_SPI_INT_CTL_RX_UDR_EN;
    }
    if ((enable & SUNXI_SPI_INT_CTL_RX_OVF_EN) && (status & SUNXI_SPI_INT_STA_RX_OVF)) {
        printf("spi irq bus rxfifo overflow\n");
        irq |= SUNXI_SPI_INT_CTL_RX_OVF_EN;
        hal_spi_reset_rxfifo(port);
    }
```

```
    if ((enable & SUNXI_SPI_INT_CTL_TX_FUL_EN) && (status & SUNXI_SPI_INT_STA_TX_FULL)) {
        printf("spi irq bus txfifo full\n");
        irq |= SUNXI_SPI_INT_CTL_TX_FUL_EN;
    }
    if ((enable & SUNXI_SPI_INT_CTL_TX_EMP_EN) && (status & SUNXI_SPI_INT_STA_TX_EMP)) {
        printf("spi irq bus txfifo empty\n");
        irq |= SUNXI_SPI_INT_CTL_TX_EMP_EN;
        compelte = true;
    }
    if ((enable & SUNXI_SPI_INT_CTL_TX_ERQ_EN) && (status & SUNXI_SPI_INT_STA_TX_RDY)) {
        printf("spi irq bus txfifo ready\n");
        irq |= SUNXI_SPI_INT_CTL_TX_ERQ_EN;
    }
    if ((enable & SUNXI_SPI_INT_CTL_RX_FUL_EN) && (status & SUNXI_SPI_INT_STA_RX_FULL)) {
        printf("spi irq bus rxfifo full\n");
        irq |= SUNXI_SPI_INT_CTL_RX_FUL_EN;
    }
nable & SUNXI_SPI_INT_CTL_RX_EMP_EN) && (status & SUNXI_SPI_INT_STA_RX_EMP)) {
        printf("spi irq bus rxfifo empty\n");
        irq |= SUNXI_SPI_INT_CTL_RX_EMP_EN;
    }
    if ((enable & SUNXI_SPI_INT_CTL_RX_RDY_EN) && (status & SUNXI_SPI_INT_STA_RX_RDY)) {
        printf("spi irq bus rxfifo ready\n");
        irq |= SUNXI_SPI_INT_CTL_RX_RDY_EN;
        compelte = true;
    }

    return 0;
}

int spi_master_cpu_loopback_test(void)
{
    int ret;


    printf("Please connect MOIS to MISO for spi master cpu loopback test!\n");

    spi_clk_init();

    spi_pin_init();

    mode |= SPI_LOOP;

    spi_init_t spi_initstruct = {0};
    spi_initstruct.bus_mode = HAL_SPI_BUS_MASTER;
    spi_initstruct.cs_mode = HAL_SPI_CS_AUTO;
    spi_initstruct.clock_frequency = speed;
    spi_initstruct.chipselect = 0;
    spi_initstruct.mode = mode;
    spi_initstruct.sip = 0;
    spi_initstruct.flash = 0;
    ret = hal_spi_init(port, &spi_initstruct);
    if (ret < 0) {
        printf("spi port %d init failed", port);
        return -1;
    }

    hal_irq_init_t irq_initstruct;

    irq_initstruct.irqn = SUNXI_IRQ_SPI3;
```

```
    irq_initstruct.preemptionpriority = 0;
    irq_initstruct.subpriority = 0;
    irq_initstruct.cmd = HAL_INIT_CMD_ENABLE;
    irq_initstruct.isr_info.func = sunxi_spi_handler;
    irq_initstruct.isr_info.parg = NULL;

    hal_v2_intc_irq_config(&irq_initstruct);

    /* enable spi complete irq and error irq */
    hal_spi_disable_irq(port, SUNXI_SPI_INT_CTL_MASK);
    hal_spi_clr_irq_pending(port, SUNXI_SPI_INT_STA_MASK);
    hal_spi_enable_irq(port, SUNXI_SPI_INT_CTL_TC_EN | SUNXI_SPI_INT_CTL_ERR);

    printf("spi mode: 0x%x\n", mode);
    printf("bits per word: %u\n", bits);
    printf("max speed: %u Hz (%u kHz)\n", speed, speed/1000);

sfer(port, default_tx, default_rx, sizeof(default_tx));

    for (i = 0; i < sizeof(default_tx); i++) {
      if (default_tx[i] != default_rx[i]) {
        printf("tx[%d]:0x%x != rx[%d]:0x%x\n", i, default_tx[i], i, default_rx[i]);
        printf("Fail: SPI[%d] master cpu loopback test failed!\n", port);
        goto out;
      } else {
        continue;
      }
    }

    printf("Success: SPI[%d] master cpu loopback test was successful!\n", port);

out:
    hal_spi_deinit(port);

    return 0;
}
#if defined(CONFIG_KERNEL_FREERTOS)
FINSH_FUNCTION_EXPORT_CMD(spi_master_cpu_loopback_test, spi_master_cpu_loopback_test, spi hal_v2 APIs tests)
#elif defined(CONFIG_KERNEL_BAREMETAL)
SHELL_EXPORT_CMD(
SHELL_CMD_PERMISSION(0)|SHELL_CMD_TYPE(SHELL_TYPE_CMD_FUNC)|SHELL_CMD_DISABLE_RETURN,
spi_master_cpu_loopback_test, spi_master_cpu_loopback_test, test for spi master cpu mode);
#endif
```

5 FAQ

无

声明

版权所有 ©2025 珠海全志科技股份有限公司。保留一切权利。

本文档及内容受著作权法保护，其著作权由珠海全志科技股份有限公司（“全志”）拥有并保留一切权利。

本文档是全志的原创作品和版权财产，未经全志书面许可，任何单位和个人不得擅自摘抄、复制、修改、发表或传播本文档内容的部分或全部，且不得以任何形式传播。

商标声明

、 、 、 （不完全列举）均为珠海全志科技股份有限公司的商标或者注册商标。在本文档描述的产品中出现的其它商标，产品名称，和服务名称，均由其各自所有人拥有。

免责声明

您购买的产品、服务或特性应受您与珠海全志科技股份有限公司（“全志”）之间签署的商业合同和条款的约束。本文档中描述的全部或部分产品、服务或特性可能不在您所购买或使用的范围内。使用前请认真阅读合同条款和相关说明，并严格遵循本文档的使用说明。您将自行承担任何不当使用行为（包括但不限于如超压，超频，超温使用）造成的不利后果，全志概不负责。

本文档作为使用指导仅供参考。由于产品版本升级或其他原因，本文档内容有可能修改，如有变不另行通知。全志尽全力在本文档中提供准确的信息，但并不确保内容完全没有错误，因使用本文档而发生损害（包括但不限于间接的、偶然的、特殊的损失）或发生侵犯第三方权利事件，全志概不负责。本文档中的所有陈述、信息和建议并不构成任何明示或暗示的保证或承诺。

本文档未以明示或暗示或其他方式授予全志的任何专利或知识产权。在您实施方案或使用产品的过程中，可能需要获得第三方的权利许可。请您自行向第三方权利人获取相关的许可。全志不承担也不代为支付任何关于获取第三方许可的许可费或版税（专利税）。全志不对您所使用的第三方许可技术做出任何保证、赔偿或承担其他义务。