SigmaStar Camera
SPI 使用参考

REVISION HISTORY

Revision No.	Description	Date
{000001}	• {Initial release}	{05/06/2019}
{000002}	• Add spi usage in linux user space and uboot	{03/05/2020}

TABLE OF CONTENTS

1. 概述

1.1. 概述

This document describes how to use the spi driver in linux user space and uboote.

2. SPI 控制

2.1. SPI 概述

The Sigmastar infinity2m platform supports a master spi controller(spi0).

2.2. SPI Pads

SPI pad modes 可設定的 mode 和腳位對應如下表

	SPI0_CZ	SPI0_CK	SPI0_DI(MOSI)	SPI0_DO(MISO)
Pad mode=1	PAD_SD_CMD	PAD_SD_CLK	PAD_SD_D0	PAD_SD_D1
Pad mode=2	PAD_TTL16	PAD_TTL17	PAD_TTL18	PAD_TTL19
Pad mode=3	PAD_GPIO7	PAD_GPIO6	PAD_GPIO5	PAD_GPIO4
Pad mode=4	PAD_FUART_RX	PAD_FUART_TX	PAD_FUART_CTS	PAD_FUART_RTS
Pad mode=5	PAD_GPIO8	PAD_GPIO9	PAD_GPIO10	PAD_GPIO11
Pad mode=6	PAD_GPIO0	PAD_GPIO1	PAD_GPIO2	PAD_GPIO3

2.3. Using SPI in linux user space

2.3.1 Enable spi

测试 spi 前需要在 kernel 打开如下两个 config：

CONFIG_SPI_SPIDEV
CONFIG_MS_SPI_INFINITY

2.3.2 SPI Pad Mode 設定

Enable spi 后，还需要把 spi pin 设定正确才可以工作，**實際使用的 pad 可透過平台對應的** infinity2m_xxxx_padmux.dtsi 中對應 PINMUX_FOR_SPI0_MODE_x 來指定：

```
<PAD_GPIO8      PINMUX_FOR_SPI0_MODE_5      MDRV_PUSE_SPI0_CZ>,
<PAD_GPIO9      PINMUX_FOR_SPI0_MODE_5      MDRV_PUSE_SPI0_CK>,
<PAD_GPIO10     PINMUX_FOR_SPI0_MODE_5      MDRV_PUSE_SPI0_DI>,
<PAD_GPIO11     PINMUX_FOR_SPI0_MODE_5      MDRV_PUSE_SPI0_DO>,
```

2.3.3 SPI device

" /dev/spidev0.0”

2.3.4 Sample code

```
static const char *device = "/dev/spidev0.0";
static uint8_t mode = 0; /* SPI 通信使用全双工，设置 CPOL = 0，CPHA = 0。 */
static uint8_t bits = 8; /* 8 bit s 读写，MSB first。*/
static uint32_t speed = 60*1000*1000;/* 设置传输速度 */
static uint16_t delay = 0;
static int g_SPI_Fd = 0;

static void pabort(const char *s)
{
    perror(s);
    abort();
}


/**
* 功 能：同步数据传输
* 入口参数 ：
* TxBuf -> 发送数据首地址
* len -> 交换数据的长度
* 出口参数：
* RxBuf -> 接收数据缓冲区
* 返回值：0 成功
* 开发人员：Lzy 2013 - 5 - 22
*/
int SPI_Transfer(const uint8_t *TxBuf, uint8_t *RxBuf, int len)
{
    int ret;
    int fd = g_SPI_Fd;


    struct spi_ioc_transfer tr ={
        .tx_buf = (unsigned long) TxBuf,
        .rx_buf = (unsigned long) RxBuf,
        .len =len,
        .delay_usecs = delay,
    };


    ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
    if (ret < 1)
        perror("can't send spi message\n");
    else
    {
#if SPI_DEBUG
        int i;
        printf("nsend spi message Succeed\n");
        printf("nSPI Send [Len:%d]: \n", len);
        for (i = 0; i < len; i++)
```

```
        {
            if (i % 8 == 0)
                printf("nt\n");
            printf("0x%02X \n", TxBuf[i]);
        }
        printf("n");


        printf("SPI Receive [len:%d]:\n", len);
        for (i = 0; i < len; i++)
        {
            if (i % 8 == 0)
                printf("nt\n");
            printf("0x%02X \n", RxBuf[i]);
        }
        printf("\n");
#endif
    }
    return ret;
}


/**
 * 功 能：发送数据
 * 入口参数 ：
 * TxBuf -> 发送数据首地址
 * len -> 发送与长度
 * 返回值：0 成功
 * 开发人员：Lzy 2013 - 5 - 22
 */
int SPI_Write(uint8_t *TxBuf, int len)
{
    int ret;
    int fd = g_SPI_Fd;


    ret = write(fd, TxBuf, len);
    if (ret < 0)
        perror("SPI Write error\n");
    else
    {
#if SPI_DEBUG
        int i;
        printf("SPI Write [Len:%d]: \n", len);
        for (i = 0; i < len; i++)
        {
            if (i % 8 == 0)
                printf("\n\t");
            printf("0x%02X \n", TxBuf[i]);
        }
        printf("\n");
#endif
```

```
    }

    return ret;
}


/**
 * 功 能：接收数据
 * 出口参数：
 * RxBuf -> 接收数据缓冲区
 * rtn -> 接收到的长度
 * 返回值：>=0 成功
 * 开发人员：Lzy 2013 - 5 - 22
 */
int SPI_Read(uint8_t *RxBuf, int len)
{
    int ret;
    int fd = g_SPI_Fd;
    ret = read(fd, RxBuf, len);
    if (ret < 0)
        printf("SPI Read error\n");
    else
    {
#if SPI_DEBUG
        int i;
        printf("SPI Read [len:%d]:\n", len);
        for (i = 0; i < len; i++)
        {
            if (i % 8 == 0)
                printf("\n\t");
            printf("0x%02X \n", RxBuf[i]);
        }
        printf("\n");
#endif
    }
    return ret;
}


/**
 * 功 能：打开设备 并初始化设备
 * 入口参数 ：
 * 出口参数：
 * 返回值：0 表示已打开 0XF1 表示 SPI 已打开 其它出错
 * 开发人员：Lzy 2013 - 5 - 22
 */
int SPI_Open(void)
{
    int fd;
    int ret = 0;
```

```
if (g_SPI_Fd != 0) /* 设备已打开 */
	return 0xF1;

fd = open(device, O_RDWR);
if (fd < 0)
	pabort("can't open device\n");
else
	printf("SPI - Open Succeed. Start Init SPI...\n");


g_SPI_Fd = fd;
/*
 * spi mode
 */
ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
if (ret == -1)
	pabort("can't set spi mode\n");


ret = ioctl(fd, SPI_IOC_RD_MODE, &mode);
if (ret == -1)
	pabort("can't get spi mode\n");


/*
 * bits per word
 */
ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
if (ret == -1)
	pabort("can't set bits per word\n");


ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
if (ret == -1)
	pabort("can't get bits per word\n");


/*
 * max speed hz
 */
ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
if (ret == -1)
	pabort("can't set max speed hz\n");


ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
if (ret == -1)
	pabort("can't get max speed hz\n");


printf("spi mode: %d\n", mode);
printf("bits per word: %d\n", bits);
```

```
    printf("max speed: %d KHz (%d MHz)\n", speed / 1000, speed / 1000 / 1000);


    return ret;
}


/**
 * 功 能：关闭 SPI 模块
 */
int SPI_Close(void)
{
    int fd = g_SPI_Fd;


    if (fd == 0) /* SPI 是否已经打开*/
        return 0;
    close(fd);
    g_SPI_Fd = 0;


    return 0;
}

/**
 * 功 能：自发自收测试程序
 * 接收到的数据与发送的数据如果不一样 ，则失败
 * 说明：
 * 在硬件上需要把输入与输出引脚短跑
 * 开发人员：Lzy 2013 - 5 - 22
 */
int SPI_LookBackTest(void)
{
    int ret, i;
    const int BufSize = 16;
    uint8_t tx[BufSize], rx[BufSize];

    bzero(rx, sizeof(rx));
    for (i = 0; i < BufSize; i++)
        tx[i] = i;

    printf("nSPI - LookBack Mode Test...\n");
    ret = SPI_Transfer(tx, rx, BufSize);
    if (ret > 1)
    {
        ret = memcmp(tx, rx, BufSize);
        if (ret != 0)
        {
                printf("tx:\n");
                for (i = 0; i < BufSize; i++)
                {
                        printf("%d ", tx[i]);
```

```
                }
                printf("\n");
                printf("rx:\n");
                for (i = 0; i < BufSize; i++)
                {
                        printf("%d ", rx[i]);
                }
                printf("\n");
            perror("LookBack Mode Test error\n");
        }
        else
            printf("SPI - LookBack Mode OK\n");
    }

    return ret;
}
```

2.4. Using SPI in uboot

2.4.1 Enable spi

操作 spi 前需要在 uboot 的 config 文件中打开如下 config 开启 spi

CONFIG_CMD_SPI
CONFIG_SSTAR_MSPI

2.4.2 SPI Pad Mode 設定

开启 spi 后，还需要把 spi pin 设定正确才能正常工作。實際使用的 pad 可透過修改 infinity2m uboot 下的 drivers/mstar/spi/infinity2m/mspi.c 中的

#define MSPI0_PADMUX_MODE 5
可設定的數值為 1 ~ 6

2.4.3 Control spi devices with uboot command line

可以透過 uboot command line 來控制/測試 spi:

The u-boot command sspi Usage:
sspi - SPI utility command
Usage:
sspi [<bus>:]<cs>[.<mode>] <bit_len> <dout> - Send and receive bits
<bus> - Identifies the SPI bus
<cs> - Identifies the chip select
<mode> - Identifies the SPI mode to use

<bit_len> - Number of bits to send (base 10)
<dout> - Hexadecimal string that gets sent
<dout> is in hex but without the prefix "0x". All others are in decimal.

The following is SPI mode defined in u-boot/include/spi.h. But still depends on the mode that SPI controller/driver can handle:

```
/* SPI mode flags */
#define SPI_CPHA        0x01                    /* clock phase */
#define SPI_CPOL        0x02                    /* clock polarity */
#define SPI_MODE_0      (0|0)                   /* (original MicroWire) */
#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     0x04                     /* CS active high */
#define SPI_LSB_FIRST   0x08                    /* per-word bits-on-wire */
#define SPI_3WIRE       0x10                    /* SI/SO signals shared */
#define SPI_LOOP        0x20                    /* loopback mode */
#define SPI_SLAVE       0x40                    /* slave mode */
#define SPI_PREAMBLE    0x80                     /* Skip preamble bytes */
```

2.4.4 SPI API

infinity2m uboot 支援 uboot 標準 spi API，調用 API 請加:

```
#include <spi.h>
```

API 說明如下(引用 spi.h 說明):

spi_setup_slave

```
/**
 * Set up communications parameters for a SPI slave.
 *
 * This must be called once for each slave. Note that this function
 * usually doesn't touch any actual hardware, it only initializes the
 * contents of spi_slave so that the hardware can be easily
 * initialized later.
 *
 * @bus:  Bus ID of the slave chip.
 * @cs:         Chip select ID of the slave chip on the specified bus.
 * @max_hz:    Maximum SCK rate in Hz.
 * @mode:      Clock polarity, clock phase and other parameters.
 *
 * Returns: A spi_slave reference that can be used in subsequent SPI
 * calls, or NULL if one or more of the parameters are not supported.
 */
```

```
struct spi_slave *spi_setup_slave(unsigned int bus, unsigned int cs,
                unsigned int max_hz, unsigned int mode);
```

spi_free_slave

```
/**
 * Free any memory associated with a SPI slave.
 *
 * @slave:      The SPI slave
 */
void spi_free_slave(struct spi_slave *slave);
```

spi_claim_bus

```
/**
 * Claim the bus and prepare it for communication with a given slave.
 *
 * This must be called before doing any transfers with a SPI slave. It
 * will enable and initialize any SPI hardware as necessary, and make
 * sure that the SCK line is in the correct idle state. It is not
 * allowed to claim the same bus for several slaves without releasing
 * the bus in between.
 *
 * @slave:      The SPI slave
 *
 * Returns: 0 if the bus was claimed successfully, or a negative value
 * if it wasn't.
 */
int spi_claim_bus(struct spi_slave *slave);
```

spi_release_bus

```
/**
 * Release the SPI bus
 *
 * This must be called once for every call to spi_claim_bus() after
 * all transfers have finished. It may disable any SPI hardware as
 * appropriate.
 *
 * @slave:      The SPI slave
 */
void spi_release_bus(struct spi_slave *slave);
```

spi_xfer

```
/**
 * SPI transfer
 *
 * This writes "bitlen" bits out the SPI MOSI port and simultaneously clocks
 * "bitlen" bits in the SPI MISO port.  That's just the way SPI works.
 *
 * The source of the outgoing bits is the "dout" parameter and the
 * destination of the input bits is the "din" parameter.  Note that "dout"
 * and "din" can point to the same memory location, in which case the
 * input data overwrites the output data (since both are buffered by
 * temporary variables, this is OK).
 *
 * spi_xfer() interface:
 * @slave:      The SPI slave which will be sending/receiving the data.
 * @bitlen:     How many bits to write and read.
 * @dout: Pointer to a string of bits to send out.  The bits are
 *          held in a byte array and are sent MSB first.
 * @din:  Pointer to a string of bits that will be filled in.
 * @flags: A bitwise combination of SPI_XFER_* flags.
 *
 * Returns: 0 on success, not 0 on failure
 */
int   spi_xfer(struct spi_slave *slave, unsigned int bitlen, const void *dout,
            void *din, unsigned long flags);
```

2.4.5 Example of using SPI API

```
#include <spi.h>

static unsigned int     bus = 0;
static unsigned int     cs = 0;
static unsigned int     mode = 0;
static int              bitlen = 32;
static uchar            dout[MAX_SPI_BYTES];
static uchar            din[MAX_SPI_BYTES];
#define SPI_MAX_SPEED_HZ        60000000    //1000000

int sstar_spi_xfer(int bus, int cs, unsigned int mod, int bitlen , uchar *dout, uchar *din )
{
      struct spi_slave *slave;
      int ret = 0;

      slave = spi_setup_slave(bus, cs, SPI_MAX_SPEED_HZ, mode);
      if (!slave) {
            printf("Invalid device %d:%d\n", bus, cs);
            return -EINVAL;
      }
```

```
	ret = spi_claim_bus(slave);
	if (ret)
		goto done;
	ret = spi_xfer(slave, bitlen, dout, din, SPI_XFER_BEGIN | SPI_XFER_END);
	if (ret) {
		printf("Error %d during SPI transaction\n", ret);
	}
done:
	spi_release_bus(slave);
	return ret;
}
```