HiKey970

SPI Development Guide

Issue 01

Date 2018-03-11

Trademarks and Permissions

Notice

HiSilicon Technologies Co., Ltd.

Address: Huawei Industrial Base
Bantian, Longgang
Shenzhen 518129
People's Republic of China

Website: http://www.hisilicon.com

Email: support@hisilicon.com

Change History

Changes between document issues are cumulative. The latest document issue contains all the changes made in earlier issues.

Issue 01 (2018-03-11)

The first version.

Contents

1 Description

1.1 SPI

1.1.1 General description

Serial Peripheral Interface (SPI) is a four-wire synchronous serial communication interface for connecting microcontrollers, sensors and storage devices. There are two types of SPI devices: master device and slave device. Control of the four lines are: chip select (CS), serial clock (SCK), master data input (MISO), and master data output (MOSI). The driver supports the ARM PL022 type peripherals.

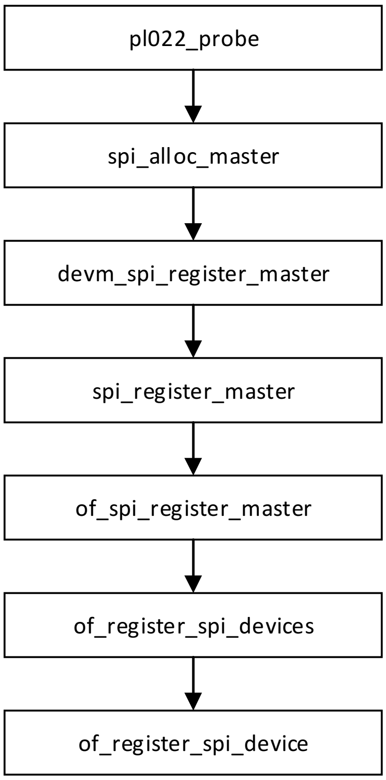
1.1.2 Features

The SPI has the following features:

- data widths from 4 to 16 bits wide
- Programmable clock bit rate and prescale
- Separate transmit and receive first-in, first-out memory buffers, 16 bits wide, 8 locations deep.
- Independent masking of transmit FIFO, receive FIFO, and receive overrun interrupts.
- Internal loopback test mode available.
- Support for *Direct Memory Access* (DMA)
- frame format
- enabling of operation

1.2 SPI Workflow

1.2.1 SPI Controller Initialization

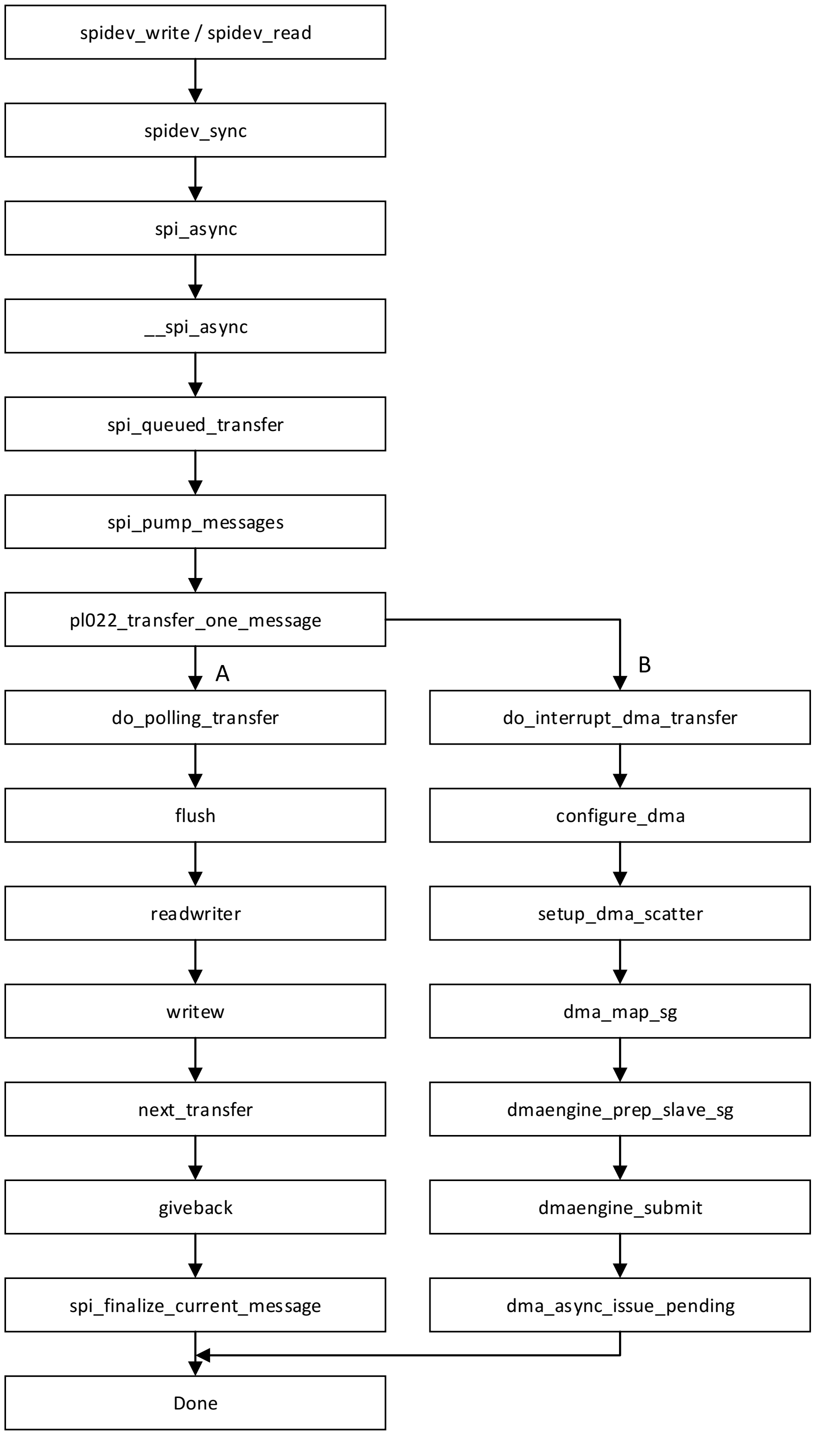


Parses SPI controller node of the DTS file to obtain the relevant hardware information, initialize the SPI controller, match the compatible attributes in DTS file, register the SPI controller, register the SPI device to the bus.

The device driver is loaded on the AMBA bus after the system starts. When there is a matching device, the driver function is called to register the SPI host controller, hardware information is obtained from the DTS file, and the DMA, clock, and memory are configured.

After everything is normal, it is in a usable state, waiting for the external program to call the spi read/write function.

1.2.2 Data Transfer



A: polling mode

B: DMA mode

1.3 Development

1.3.1 DTS Configuration

The DTS configuration of the SPI controller mainly involves the file `kirin970.dtsi`

```
spi0: spi@ffd70000 {
        compatible = "arm,pl022", "arm,primecell";
        reg = <0x0 0xffd70000 0x0 0x1000>;
        #address-cells = <1>;
        #size-cells = <0>;
        interrupts = <GIC_SPI 113 IRQ_TYPE_LEVEL_HIGH>;
        clocks = <&iomcu KIRIN970_CLK_GATE_SPI0>;
        clock-names = "apb_pclk";
        pinctrl-names = "default";
        pinctrl-0 = <&spi0_pmx_func &spi0_cfg_func &spi0_clk_cfg_func>;
        num-cs = <1>;
        cs-gpios = <&gpio28 6 0>;
        status = "ok";
};
```

This SPI controller configuration includes the base address, interrupt number, clock configuration, chip select, and the SPI controller switch. Configurations for SPI devices can be placed under the file `kirin970-hikey970.dts`. Examples are as follows.

```
&spi0 {
    status = "ok";
    myspidev@0x00 {
        compatible = "myspidev";
        spi-max-frequency = <20000000>;
        reg = <0>;
    };
};
```

1.3.2 Device Driver Configuration

Modify the file arch/arm64/configs/hikey970_defconfig, add

```
CONFIG_SPI_MYSPIDEV=y
```

Modify the file `drivers/spi/Makefile`, add

```
obj-$(CONFIG_SPI_MYSPIDEV) += myspidev.o
```

1.3.3 Data Structure

1.3.3.1 SPI device

```
struct spi_device {
```

```
    struct device      dev;
    struct spi_master   *master;
    u32         max_speed_hz;
    u8          chip_select;
    u8          bits_per_word;
    u16         mode;
#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            /* chipselect 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_NO_CS   0x40            /* 1 dev/bus, no chipselect */
#define SPI_READY   0x80            /* slave pulls low to pause */
#define SPI_TX_DUAL 0x100           /* transmit with 2 wires */
#define SPI_TX_QUAD 0x200           /* transmit with 4 wires */
#define SPI_RX_DUAL 0x400           /* receive with 2 wires */
#define SPI_RX_QUAD 0x800           /* receive with 4 wires */
    int         irq;
    void            *controller_state;
    void            *controller_data;
    char            modalias[SPI_NAME_SIZE];
    int         cs_gpio;    /* chip select gpio */

    /*
     * likely need more hooks for more protocol options affecting how
     * the controller talks to each chip, like:
     *  - memory packing (12 bit samples into low bits, others zeroed)
     *  - priority
     *  - drop chipselect after each word
     *  - chipselect delays
     *  - ...
     */
};
```

1.3.3.2 SPI device driver

```
struct spi_driver {
    const struct spi_device_id *id_table;
    int         (*probe)(struct spi_device *spi);
    int         (*remove)(struct spi_device *spi);
```

```
    void          (*shutdown)(struct spi_device *spi);
    int         (*suspend)(struct spi_device *spi, pm_message_t mesg);
    int         (*resume)(struct spi_device *spi);
    struct device_driver    driver;
};
```

1.3.4 Function

1.3.4.1 spi_register_master

prototype

```
#include <linux/spi/spi.h>
int spi_register_master(struct spi_master *master);
```

description

register SPI master controller

parameter

master: initialized master

return

zero on success, else a negative error code.

1.3.4.2 spi_register_driver

prototype

```
#include <linux/spi/spi.h>
#define spi_register_driver(driver)  __spi_register_driver(THIS_MODULE,
driver)
int __spi_register_driver(struct module *owner, struct spi_driver *sdrv)
```

description

register a SPI driver

parameter

owner: owner module of the driver to register

sdrv: the driver to register

return

zero on success, else a negative error code.

1.3.4.3 spi_unregister_driver

prototype

```
#include <linux/spi/spi.h>
static inline void spi_unregister_driver(struct spi_driver *sdrv)
```

description

reverse effect of spi_register_driver

parameter

sdrv: the driver to unregister

return

none

1.3.4.4 spi_add_device

prototype

```
#include <linux/spi/spi.h>
int spi_add_device(struct spi_device *spi)
```

description

add a new SPI device

parameter

spi: spi_device to register

return

0 on success; negative errno on failure.

1.3.4.5 spi_setup

prototype

```
#include <linux/spi/spi.h>
int spi_setup(struct spi_device *spi)
```

description

setup SPI mode and clock rate

parameter

spi: the device whose settings are being modified

return

zero on success, else a negative error code.

1.3.4.6 spi_sync

prototype

```
#include <linux/spi/spi.h>
int spi_sync(struct spi_device *spi, struct spi_message *message))
```

description

blocking/synchronous SPI data transfers

parameter

spi: device with which data will be exchanged

return

zero on success, else a negative error code.

1.3.4.7 spi_async

prototype

```
#include <linux/spi/spi.h>
int spi_async(struct spi_device *spi, struct spi_message *message)
```

description

asynchronous SPI transfer

parameter

spi: device with which data will be exchanged

return

zero on success, else a negative error code.

1.3.5 Reference

Add your own device driver file `drivers/spi/myspidev.c`. If the device in dts matches this driver, execute `myspidev_probe()` in the file `myspidev.c`. The `myspidev_probe()` function can be implemented on demand

```
...
static const struct of_device_id myspidev_of_match[] = {
    {.compatible = "myspidev",},
    { }
```

```
};
MODULE_DEVICE_TABLE(of, myspidev_of_match);

static int myspidev_probe(struct spi_device *spi)
{
	…
}

static int myspidev_remove(struct spi_device *spi)
{
	…
}

static struct spi_driver myspidev_spi_driver = {
	.driver = {
		.name =		"myspidev",
		.of_match_table = myspidev_of_match,
	},
	.probe =	myspidev_probe,
	.remove =	myspidev_remove,

	/* NOTE:  suspend/resume methods are not necessary here.
	 * We don't do anything except pass the requests to/from
	 * the underlying controller.  The refrigerator handles
	 * most issues; the controller driver handles the rest.
	 */
};

static int __init myspidev_init(void)
{
	return spi_register_driver(&myspidev_spi_driver);
}
module_init(myspidev_init);
static void __exit myspidev_exit(void)
{
	spi_unregister_driver(&myspidev_spi_driver);
}
module_exit(myspidev_exit);
```