HiKey970

I2C Development Guide

Issue	**01**
Date	**2018-03-11**

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 I2C

1.1.1 General description

The Inter-integrated Circuit (I2C) bus is a very powerful bus used for communication between master and slave devices. The physical I2C interface consists of the serial data (SDA) and serial clock (SCL) lines. Here the I2C module is a controller or master that communicates with the slave devices.

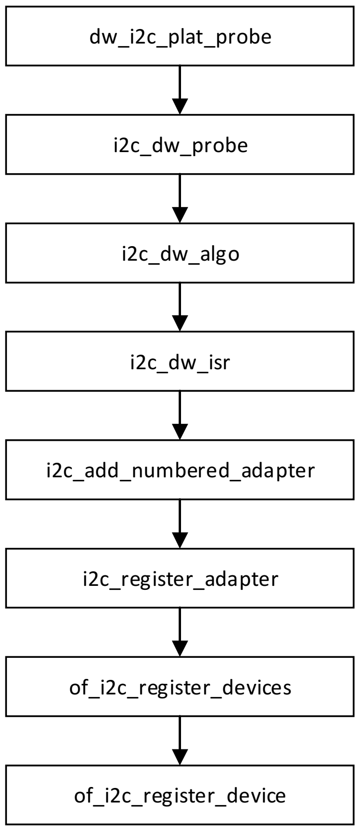
1.1.2 Features

The I2C has the following features:

- Supports the I2C-bus specification version 2.1.
- It can only be used as a Master on the I2C bus, and it cannot be used as a Slave.
- As a transmitter on the I2C bus, the master sends data to the slave.
- The slave address supported as the master device: standard address (7-bit) and extended address (10-bit).
- Supports standard mode 100kbit/s, fast mode 400kbit/s, and high-speed mode 3.4Mbit/s data rates.
- Provides transmit FIFO, receive FIFO, and DMA data transfer methods.
- Supports interrupt reporting and initial interrupt status and masked interrupt status query.
- Support Clock stretching function. During data transmission, when the transmit FIFO data is empty, pull down the SCL and wait for the FIFO to fill the data again. During data reception, when the receive FIFO data is empty, pull down the SCL and wait for the FIFO to fill the data again.
- Supports SDA data Restart transmission. The data bus is not released and transmission continues on the same channel.

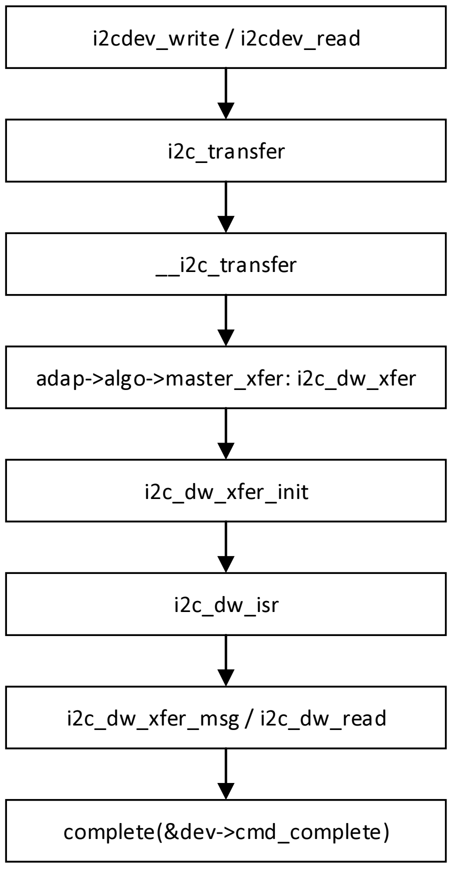
1.2 I2C Workflow

1.2.1 I2C Adapter Initialization



Parses the DTS file I2C adapter node, obtains relevant hardware information, initializes the I2C adapter, matches compatible attributes in DTS file, struct `i2c_algorithm` type initializes `i2c_dw_algo`, implements `master_xfer()` function and `functionality()` function, registers interrupt handling function `i2c_dw_isr`, adds I2C adapter , register I2C devices.

1.2.2 Data Transfer


i2cdev_write / i2cdev_read
i2c_transfer
__i2c_transfer
adap->algo->master_xfer: i2c_dw_xfer
i2c_dw_xfer_init
i2c_dw_isr
i2c_dw_xfer_msg / i2c_dw_read
complete(&dev->cmd_complete)

1. I2C device to send and receive data, call the driver interface write or read;
2. Start the data transfer by calling the `i2c_transfer` function in the file `drivers/i2c/i2c-core.c`;
3. `i2c_transfer` calls `__i2c_transfer` to execute the `master_xfer` function, which is a callback function `i2c_dw_xfer` registered when I2C adapter is started. The `i2c_dw_xfer` function is in the file `drivers/i2c/busses/i2c-designware-core.c`;
4. At this point, open the I2C controller via the `i2c_dw_xfer_init` function and enable the interrupt.
5. Receives an interrupt and executes the interrupt handler `i2c_dw_isr`;
6. Send and receive data through `i2c_dw_xfer_msg` or `i2c_dw_read`;
7. release complete to finish this transmission;

----End

1.3 Development

1.3.1 DTS Configuration

The DTS configuration of the I2C adapter mainly involves the file `kirin970.dtsi`

```
i2c0: i2c@ffd71000 {
        compatible = "snps,designware-i2c";
        reg = <0x0 0xffd71000 0x0 0x1000>;
        interrupts = <GIC_SPI 118 IRQ_TYPE_LEVEL_HIGH>;
        #address-cells = <1>;
        #size-cells = <0>;
        clock-frequency = <400000>;
        clocks = <&iomcu KIRIN970_CLK_GATE_I2C0>;
        resets = <&iomcu_rst 0x20 3>;
        pinctrl-names = "default";
        pinctrl-0 = <&i2c0_pmx_func &i2c0_cfg_func>;
        status = "disabled";
};
```

This I2C adapter configuration includes the base address, interrupt number, clock configuration, reset, and I2C adapter switches.

For I2C device configuration can be placed in file `kirin970-hikey970.dts`. Here are some examples.

```
&i2c0 {
    status = "okay";
    myi2cdev@50 {
        compatible = "myi2cdev";
        reg = <0x50>;
        ........
    };
};
```

The above `2f` is the device address and is modified as required.

1.3.2 Device Driver Configuration

1. Modify the file arch/arm64/configs/hikey970_defconfig and add:

```
CONFIG_I2C_MYI2CDEV =y
```

2. Modify the file `drivers/i2c/Makefile` and add:

```
obj-$(CONFIG_I2C_MYI2CDEV) += myi2cdev.o
```

1.3.3 Data Structure

1.3.3.1 I2C Client

```
struct i2c_client {
	unsigned short flags;		/* div., see below		*/
	unsigned short addr;		/* chip address - NOTE: 7bit	*/
					/* addresses are stored in the	*/
					/* _LOWER_ 7 bits		*/
	char name[I2C_NAME_SIZE];
	struct i2c_adapter *adapter;	/* the adapter we sit on	*/
	struct device dev;		/* the device structure		*/
	int irq;			/* irq issued by device		*/
	struct list_head detected;
#if IS_ENABLED(CONFIG_I2C_SLAVE)
	i2c_slave_cb_t slave_cb;	/* callback for slave mode	*/
#endif
};
```

1.3.3.2 I2C driver

```
struct i2c_driver {
	unsigned int class;

	/* Notifies the driver that a new bus has appeared. You should avoid
	 * using this, it will be removed in a near future.
	 */
	int (*attach_adapter)(struct i2c_adapter *) __deprecated;

	/* Standard driver model interfaces */
	int (*probe)(struct i2c_client *, const struct i2c_device_id *);
	int (*remove)(struct i2c_client *);

	/* driver model interfaces that don't relate to enumeration  */
	void (*shutdown)(struct i2c_client *);

	/* Alert callback, for example for the SMBus alert protocol.
	 * The format and meaning of the data value depends on the protocol.
	 * For the SMBus alert protocol, there is a single bit of data passed
	 * as the alert response's low bit ("event flag").
	 * For the SMBus Host Notify protocol, the data corresponds to the
	 * 16-bit payload data reported by the slave device acting as master.
	 */
	void (*alert)(struct i2c_client *, enum i2c_alert_protocol protocol,
		      unsigned int data);
        /* a ioctl like command that can be used to perform specific functions
         * with the device.
         */
        int (*command)(struct i2c_client *client, unsigned int cmd, void *arg);

        struct device_driver driver;
        const struct i2c_device_id *id_table;

        /* Device detection callback for automatic device creation */
        int (*detect)(struct i2c_client *, struct i2c_board_info *);
        const unsigned short *address_list;
        struct list_head clients;
};
```

1.3.4 Function

1.3.4.1 i2c_add_numbered_adapter

prototype

```
#include <linux/i2c.h>
int i2c_add_numbered_adapter(struct i2c_adapter *adap);
```

description

declare i2c adapter, use static bus number

parameter

adap: the adapter to register (with adap->nr initialized)

return

zero on success, else a negative error code.

1.3.4.2 i2c_add_driver

prototype

```
#include <linux/i2c.h>
#define i2c_add_driver(driver) i2c_register_driver(THIS_MODULE, driver)
int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
```

description

register a I2C driver

parameter

owner: owner module of the driver to register

driver: the driver to register

return

zero on success, else a negative error code.

1.3.4.3 i2c_del_driver

prototype

```
void i2c_del_driver(struct i2c_driver *driver);
```

description

unregister I2C driver

parameter

driver: the driver being unregistered

return

none

1.3.4.4 i2c_new_device

prototype

```
#include <linux/i2c.h>
struct i2c_client *
i2c_new_device(struct i2c_adapter *adap, struct i2c_board_info const *info);
```

description

instantiate an i2c device

parameter

adap: the adapter managing the device

info: describes one I2C device; bus_num is ignored

return

This returns the new i2c client, which may be saved for later use with i2c_unregister_device(); or NULL to indicate an error.

1.3.4.5 i2c_transfer

prototype

```
#include <linux/i2c.h>
int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num);
```

description

execute a single or combined I2C message

parameter

adap: Handle to I2C bus

msgs: One or more messages to execute before STOP is issued to terminate the operation; each message begins with a START.

num: Number of messages to be executed.

return

Returns negative errno, else the number of messages executed.

1.3.4.6 i2c_master_send

prototype

```
#include <linux/i2c.h>
int i2c_master_send(const struct i2c_client *client, const char *buf, int
count);
```

description

issue a single I2C message in master transmit mode

parameter

client: Handle to slave device

buf: Data that will be written to the slave

count: How many bytes to write, must be less than 64k since msg.len is u16

return

Returns negative errno, or else the number of bytes written.

1.3.4.7 i2c_master_recv

prototype

```
#include <linux/i2c.h>
int i2c_master_recv(const struct i2c_client *client, char *buf, int count);
```

description

issue a single I2C message in master receive mode

parameter

client: Handle to slave device

buf: Where to store data read from slave

count: How many bytes to read, must be less than 64k since msg.len is u16

return

Returns negative errno, or else the number of bytes read.

1.3.5 Reference

Add your own device driver file `drivers/i2c/myi2cdev.c`. If the device in dts matches this driver, execute `myi2cdev_probe()` in the file `myi2cdev.c`. The `myi2cdev_probe()` function is implemented by the user according to the requirements

```
…
static const struct of_device_id myi2cdev_of_match[] = {
    {.compatible = "myi2cdev",},
    { }
};
MODULE_DEVICE_TABLE(of, myi2cdev_of_match);

static int myi2cdev_probe(struct i2c_client *client,
                              const struct i2c_device_id *id)
{
        …
}

static int myi2cdev_remove(struct i2c_client *i2c)
{
        …
}

static struct i2c_driver myi2cdev_i2c_driver = {
        .driver = {
                .name =        "myi2cdev",
                .of_match_table = myi2cdev_of_match,
        },
        .probe =       myi2cdev_probe,
        .remove =      myi2cdev_remove,
};
```

```
static int __init myi2cdev_init(void)
{
       return i2c_add_driver(&myi2cdev_i2c_driver);
}
subsys_initcall(myi2cdev_init);
static void __exit myi2cdev_exit(void)
{
       i2c_del_driver(&myi2cdev_i2c_driver);
}
module_exit(myi2cdev_exit)
```