How To Debug I2C

Revision	Date	Description
V1.0	2016/06/30	Initial Draft

Archermind

2016/7/19

Contents

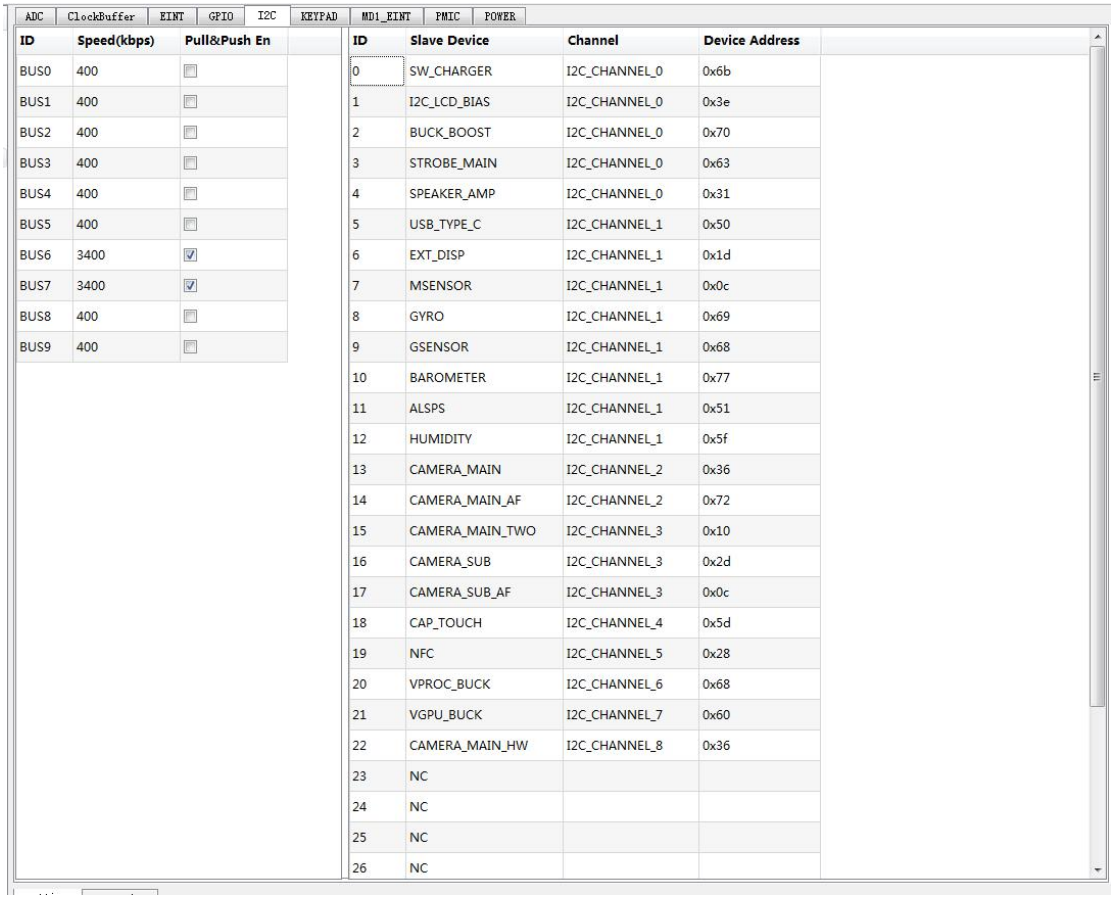
1. Introduction

I2C is a widely used serial bus.Using I2C is an essential skill for developers.This document will help you to use I2C on the MediaTek platform.

2. I2C configuration

2.1. Tool

This tool is located in this path :alps\kernel-3.18\tools\dct\DrvGen.exe.



ADC | ClockBuffer | EINT | GPIO | I2C | KEYPAD | MD1_EINT | PMIC | POWER

ID	Speed(kbps)	Pull&Push En
BUS0	400	☐
BUS1	400	☐
BUS2	400	☐
BUS3	400	☐
BUS4	400	☐
BUS5	400	☐
BUS6	3400	☑
BUS7	3400	☑
BUS8	400	☐
BUS9	400	☐

ID	Slave Device	Channel	Device Address
0	SW_CHARGER	I2C_CHANNEL_0	0x6b
1	I2C_LCD_BIAS	I2C_CHANNEL_0	0x3e
2	BUCK_BOOST	I2C_CHANNEL_0	0x70
3	STROBE_MAIN	I2C_CHANNEL_0	0x63
4	SPEAKER_AMP	I2C_CHANNEL_0	0x31
5	USB_TYPE_C	I2C_CHANNEL_1	0x50
6	EXT_DISP	I2C_CHANNEL_1	0x1d
7	MSENSOR	I2C_CHANNEL_1	0x0c
8	GYRO	I2C_CHANNEL_1	0x69
9	GSENSOR	I2C_CHANNEL_1	0x68
10	BAROMETER	I2C_CHANNEL_1	0x77
11	ALSPS	I2C_CHANNEL_1	0x51
12	HUMIDITY	I2C_CHANNEL_1	0x5f
13	CAMERA_MAIN	I2C_CHANNEL_2	0x36
14	CAMERA_MAIN_AF	I2C_CHANNEL_2	0x72
15	CAMERA_MAIN_TWO	I2C_CHANNEL_3	0x10
16	CAMERA_SUB	I2C_CHANNEL_3	0x2d
17	CAMERA_SUB_AF	I2C_CHANNEL_3	0x0c
18	CAP_TOUCH	I2C_CHANNEL_4	0x5d
19	NFC	I2C_CHANNEL_5	0x28
20	VPROC_BUCK	I2C_CHANNEL_6	0x68
21	VGPU_BUCK	I2C_CHANNEL_7	0x60
22	CAMERA_MAIN_HW	I2C_CHANNEL_8	0x36
23	NC		
24	NC		
25	NC		
26	NC		

2.2. amt6797_64_open.dws

```
<?xml version="1.0" encoding="UTF-8"?>
<!--dct_version="2.2" buid_sn="160113" dws_modification_time="05.29.2016.13:03:38"-->
<dct_cfg>
    <general chip="MT6797">
        <proj>k97_v1</proj>
        <module name="i2c">
            <i2c_bus1>
                <speed_kbps>400</speed_kbps>
                <pullPushEn>false</pullPushEn>
```

```
            </i2c_bus1>
            <device6>
                <varName>EXT_DISP</varName>
                <channel>I2C_CHANNEL_1</channel>
                <address>0x1d</address>
            </device6>
        </module>
    </general>
 </dct_cfg>
```

2.3. cust_i2c.dtsi

```
&i2c1 {
    #address-cells = <1>;
    #size-cells = <0>;
    clock-frequency = <400000>;
    mediatek,use-open-drain;

    ext_disp@1d {
        compatible = "mediatek,ext_disp";
        reg = <0x1d>;
        status = "okay";
    }
};
```

3. I2C code

3.1. Define I2C driver

Before the definition of I2C driver, Define of_device_id and i2c_device_id.Of_device_id is used to call the device information defined in the DTS file in the device drive, which is defined as follows:

```
&i2c1 {
	#address-cells = <1>;
	#size-cells = <0>;
	clock-frequency = <400000>;
	mediatek,use-open-drain
	ext_disp@1d {
		compatible = "mediatek,ext_disp";
		reg = <0x1d>;
		status = "okay";
	};
};

static const struct of_device_id mt8193_i2c_mhl_id[]= {

		{ .compatible = "mediatek,ext_disp" },

		{ }

};

#define MT8193_DEVICE_NAME		"mtk-multibridge"

static const struct i2c_device_id mt8193_i2c_id[] = {

	{MT8193_DEVICE_NAME, 0},

	{}

};

struct i2c_driver mt8193_i2c_driver = {

	.probe		= mt8193_i2c_probe,
```

```
    .remove         = mt8193_i2c_remove,
    .driver         = {
                    .name = MT8193_DEVICE_NAME,
                    .of_match_table = mt8193_i2c_mhl_id,
    },
    .id_table  = mt8193_i2c_id,
};
```

Note: id_table indicates the device supported by the device drive.

3.2. Registered I2C driver

```
i2c_add_driver(&mt8193_i2c_driver);
```

When callIing i2c_add_driver to registered I2C driver,OS will traverse the I2C device.if the I2C driver support the traversal of the device, it will call the probe function of the device driver.

3.3. I2C Communication

After registering a good I2C driver, you can exchange data by I2C.

```
int mt8193_i2c_write(u16 addr, u32 data)
{
    struct i2c_client *client = mt8193_i2c_client;
    u8 buffer[8];
    int ret = 0;
    struct   i2c_msg   msg = {
        /*.addr    = client->addr & I2C_MASK_FLAG,*/
        .addr      = client->addr,
        .flags     = 0,
```

```
        .len  = (((addr >> 8) & 0xFF) >= 0x80)?5:6,
        .buf  = buffer,
    };
    if (((addr >> 8) & 0xFF) >= 0x80) {
        /* 8 bit : fast mode */
        buffer[0] = (addr >> 8) & 0xFF;
        buffer[1] = (data >> 24) & 0xFF;
        buffer[2] = (data >> 16) & 0xFF;
        buffer[3] = (data >> 8) & 0xFF;
        buffer[4] = data & 0xFF;
    } else {
        /* 16 bit : noraml mode */
        buffer[0] = (addr >> 8) & 0xFF;
        buffer[1] = addr & 0xFF;
        buffer[2] = (data >> 24) & 0xFF;
        buffer[3] = (data >> 16) & 0xFF;
        buffer[4] = (data >> 8) & 0xFF;
        buffer[5] = data & 0xFF;
    }

    ret = i2c_transfer(client->adapter, &msg, 1);
    if (ret < 0) {
        pr_err("%s: send command error\n", __func__);
        return -EFAULT;
```

```
    }

    return 0;

}

int mt8193_i2c_read(u16 addr, u32 *data)

{

    struct i2c_client *client = mt8193_i2c_client;

    struct i2c_msg msg[2];

    u8 rxBuf[8] = {0};

    u8 lens = 0;

    if (((addr >> 8) & 0xFF) >= 0x80) {

        /* 8 bit : fast mode */

        rxBuf[0] = (addr >> 8) & 0xFF;

        lens = 1;

    } else {

        /* 16 bit : noraml mode */

        rxBuf[0] = (addr >> 8) & 0xFF;

        rxBuf[1] = addr & 0xFF;

        lens = 2;

    }

    msg[0].flags = 0;

    msg[0].addr = client->addr;

    msg[0].buf = rxBuf;

    msg[0].len = lens;

    msg[1].flags = I2C_M_RD;
```

```
    msg[1].addr = client->addr;

    msg[1].buf = rxBuf;

    msg[1].len = 4;

    i2c_transfer(client->adapter, msg, 2);

    *data = (rxBuf[3] << 24) | (rxBuf[2] << 16) | (rxBuf[1] << 8) | (rxBuf[0]);
/*LSBfisrt*/

    return 0;

}
```

So Host can use the API (mt8193_i2c_read ,mt8193_i2c_write) to communicate with the slave.