linux i2c设备驱动调试

一、i2c设备adxl345的芯片手册和原理图简要分析

1、芯片手册分析

本次i2c设备驱动实验在nano开发板上运行，测试的i2c设备是板上的adxl345 gsensor。如图所示为adxl345芯片手册的引脚描述。其中13号引脚和14号引脚分别对应i2c的sda和scl，当运行i2c测试的时候，通过示波器接地端接2号引脚，另外一端接13或14号引脚就能采样到sda或scl的电平波形。

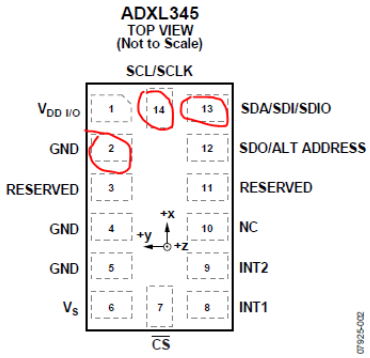


Figure 3. Pin Configuration (Top View)

Table 5. Pin Function Descriptions

Pin No.	Mnemonic	Description
1	$V_{DD\ I/O}$	Digital Interface Supply Voltage.
2	GND	This pin must be connected to ground.
3	RESERVED	Reserved. This pin must be connected to V_S or left open.
4	GND	This pin must be connected to ground.
5	GND	This pin must be connected to ground.
6	V_S	Supply Voltage.
7	$\overline{CS}$	Chip Select.
8	INT1	Interrupt 1 Output.
9	INT2	Interrupt 2 Output.
10	NC	Not Internally Connected.
11	RESERVED	Reserved. This pin must be connected to ground or left open.
12	SDO/ALT ADDRESS	Serial Data Output (SPI 4-Wire)/Alternate I^2C Address Select (I^2C).
13	SDA/SDI/SDIO	Serial Data (I^2C)/Serial Data Input (SPI 4-Wire)/Serial Data Input and Output (SPI 3-Wire).
14	SCL/SCLK	Serial Communications Clock. SCL is the clock for I^2C, and SCLK is the clock for SPI.

下面是adxl345的i2c模式描述，当cs引脚接到芯片供应电压VDD时，adxl345就选中i2c模式；当ALT ADDRESS引脚为高电平时，设备的i2c地址为0x3a；当ALT ADDRESS引脚为低电平时，设备的i2c地址为0x53。

I²C

With $\overline{CS}$ tied high to $V_{DD\ I/O}$, the ADXL345 is in I²C mode, requiring a simple 2-wire connection, as shown in Figure 39. The ADXL345 conforms to the *UM10204 I²C-Bus Specification and User Manual*, Rev. 03—19 June 2007, available from NXP Semiconductor. It supports standard (100 kHz) and fast (400 kHz) data transfer modes if the bus parameters given in Table 11 and Table 12 are met. Single- or multiple-byte reads/writes are supported, as shown in Figure 40. With the ALT ADDRESS pin high, the 7-bit I²C address for the device is 0x1D, followed by the R/$\overline{W}$ bit. This translates to 0x3A for a write and 0x3B for a read. An alternate I²C address of 0x53 (followed by the R/$\overline{W}$ bit) can be chosen by grounding the ALT ADDRESS pin (Pin 12). This translates to 0xA6 for a write and 0xA7 for a read.

There are no internal pull-up or pull-down resistors for any unused pins; therefore, there is no known state or default state for the $\overline{CS}$ or ALT ADDRESS pin if left floating or unconnected. It is required that the $\overline{CS}$ pin be connected to $V_{DD\ I/O}$ and that the ALT ADDRESS pin be connected to either $V_{DD\ I/O}$ or GND when using I²C.

Due to communication speed limitations, the maximum output data rate when using 400 kHz I²C is 800 Hz and scales linearly with a change in the I²C communication speed. For example, using I²C at 100 kHz would limit the maximum ODR to 200 Hz. Operation at an output data rate above the recommended maximum may result in undesirable effect on the acceleration data, including missing samples or additional noise.

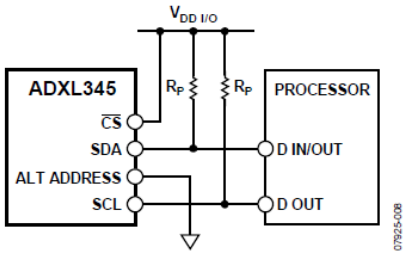
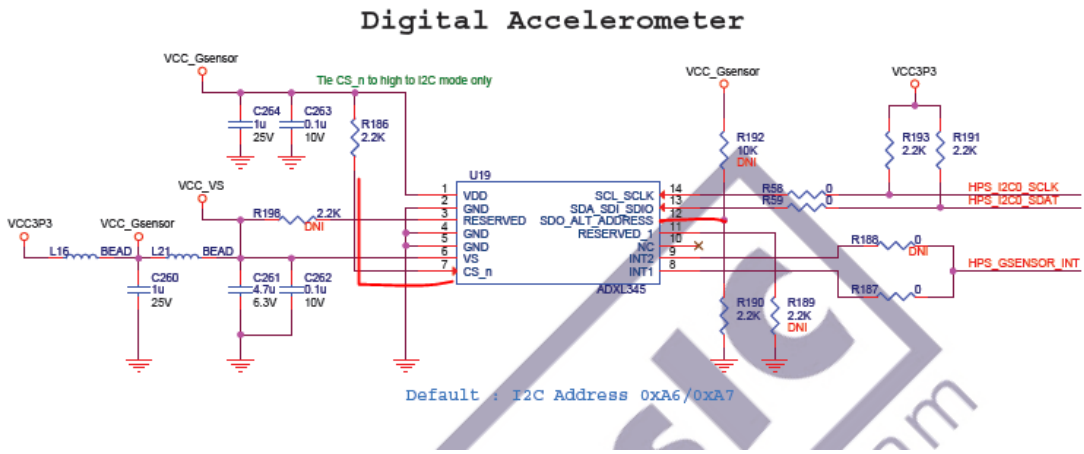


Figure 39. I²C Connection Diagram (Address 0x53)

If other devices are connected to the same I²C bus, the nominal operating voltage level of these other devices cannot exceed $V_{DD\ I/O}$ by more than 0.3 V. External pull-up resistors, R_P, are necessary for proper I²C operation. Refer to the *UM10204 I²C-Bus Specification and User Manual*, Rev. 03—19 June 2007, when selecting pull-up resistor values to ensure proper operation.

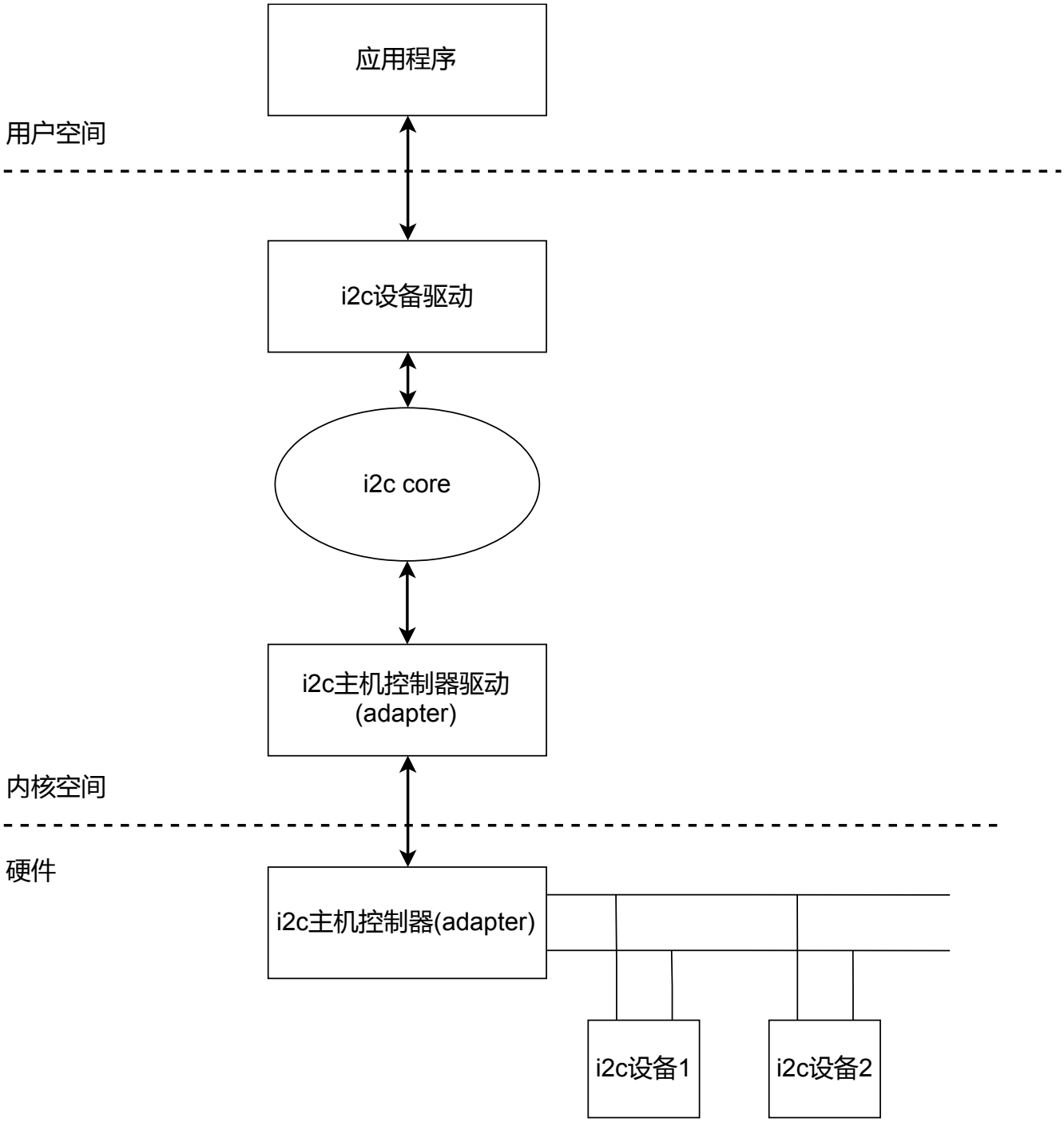
2、原理图分析

然后看nano开发板的adxl345部分的原理图连接，cs引脚接到l了芯片供应电压VDD，因此该设备选中了i2c模式，并且ALT ADDRESS引脚电压为低电平，因此设备地址为0x53，在通讯过程中会左移一位再加上一位表示读或者写得到8位的地址0xa6或者0xa7。



二、驱动程序框架与编写

1、驱动框架



linux的i2c架构如上图所示，应用程序通过访问设备文件经过内核空间的i2c驱动访问i2c设备。重点关注内核空间的i2c驱动架构，分为3个组成部分。

1.1 i2c设备驱动

调用i2c core提供的统一api控制主机控制器发出时序信号来访问i2c设备。这一层主要是实现 i2c_driver结构体，当注册i2c_driver后挂载到i2c_bus_type总线上，如果有匹配的i2c_client则调用 i2c_driver的probe函数注册包含有自定义的file operation结构体的设备结构体，当应用程序打开对应的设备文件操作时就会调用到file operation中的自定义函数，将用户数据和硬件进行交互。

1.2 i2c核心(core)

i2c core提供统一的api，主要是给i2c设备驱层和控制器驱动层提供注册和注销方法，为i2c设备驱层提供统一的通信接口如i2c_transfer等。

1.3 i2c主机控制器驱动(adapter)

i2c主机控制器的驱动，一般由原厂soc负责。主要设置注册adapter驱动结构体i2c_adapter，该结构体中的i2c_algorithm结构体中的master_xfer函数就是适配器用来和从设备通信的函数，负责将相应的数据根据i2c协议产生相应时序的信号和设备通信。并且主机控制器驱动会处理i2c设备信息生成相应的i2c_client结构体然后注册到i2c_bus_type总线上。

2、驱动程序编写

2.1 在设备树中添加设备节点

```
&i2c0 {
    status = "okay";

    adxl345@53 {
        compatible = "analog,adxl345";
        reg = <0x53>;
    };

};
```

通过原理图可以知道adxl345挂在i2c0上，因此设备节点添加在父节点i2c0中。如上图所示，在i2c0节点中添加adxl345@53设备节点，该节点会被adapter驱动处理，生成i2c_client.。其中compatible属性是用来匹配同样注册到i2c_bus_type的i2c_driver，如果匹配上就会调用i2c_driver中的probe函数。reg属性是该设备的i2c地址。同时还要设置父节点i2c0的status属性为okay，否则内核不会处理该父节点，即不会将它处理为相应的platform_device然后注册到platform_bus_type当中，那么就不会触发paltform_driver中的probe，所以就不会处理接下来的子节点。

2.2 注册、注销和匹配i2c_driver

```
static const struct of_device_id adxl345_of_match[] = {
    { .compatible = "analog,adxl345" },
    { }
};

MODULE_DEVICE_TABLE(of, adxl345_of_match);

static const struct i2c_device_id adxl345_id[] = {
    { "adxl345", 0 },
    { }
};

MODULE_DEVICE_TABLE(i2c, adxl345_id);

static struct i2c_driver adxl345_driver = {
    .driver = {
        .name = "adxl345",
        .owner = THIS_MODULE,
        .of_match_table = adxl345_of_match,
    },
    .probe    = adxl345_probe,
    .remove   = adxl345_remove,
    .id_table = adxl345_id,
```

```
};

//module_i2c_driver(adxl345_driver);
static int __init adxl345_driver_init(void)
{
    printk("adxl345 driver init\n");
    return i2c_add_driver(&adxl345_driver);
}
module_init(adxl345_driver_init);

static void __exit adxl345_driver_exit(void)
{
    printk("adxl345 driver exit\n");
    i2c_del_driver(&adxl345_driver);
}
module_exit(adxl345_driver_exit);
```

注册和注销函数分别将i2c_driver结构体adxl345_driver挂接到i2c_bus_type总线上以及从i2c_bus_type总线上卸载adxl345_driver。i2c_driver中的of_match_table用来和具有相同字符串的compatile属性的设备树节点得到的i2c_client进行匹配。

2.3 匹配成功后的操作

```
static int adxl345_probe(struct i2c_client *client,
                         const struct i2c_device_id *id)
{

    printk("adxl345 probe\n");
    int err;

    /*
     * get informations from device tree
     * do it later
     */


    /* register misc device */

    err = misc_register(&adxl345_misc);
    if (err) {
        printk(KERN_ERR "register adxl345 misc fail!\n");
        goto register_err;
    }


    /*
     * register interrupt
     * do it later with the blocked function
     */


    /* fill the global value adxl345,
     * and attach it to client->dev->p->driver_data,
```

```
     * but for now i do not use client->dev->p->driver_data,
     * maybe i will use it later,
     * and when i familiar with the device tree i wiil
     * replace the follow code of filling
     */
    adxl345.i2c_client = client;
    adxl345.ofsx = AXS_X_REG;
    adxl345.ofsy = AXS_Y_REG;
    adxl345.ofsz = AXS_Z_REG;
    adxl345.dev_addr = ADXL345_I2C_ADDR;
    i2c_set_clientdata(client, &adxl345);

    return 0;

interrupt_err:
    /* do it later */


register_err:
    err = misc_deregister(&adxl345_misc);
    if (err) {
        printk(KERN_ERR "deregister adxl345 misc fail!\n");
        return err;
    }

    return 0;

}
```

i2c_driver和相应的i2c_client匹配成功后调用probe函数，如上所示，这里做什么完全由自己决定。这里主要做的事是通过misc_register接口注册一个包含file operation结构体的misc设备，同时获取设备信息填入全局变量adxl345中。

2.4 file operation操作

在本文的驱动的file operation结构体中实现open、ioctl、read、write和release函数，分别对应应用程序的open、ioctl、read、write和close。

```
static int adxl345_open(struct inode *inode, struct file *filp)
{
    filp->private_data = (void *)&adxl345;
    filp->f_flags |= O_NONBLOCK;  //default nonblock
    return 0;
}

static long adxl345_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    switch (cmd) {
        case BLOCK:
            filp->f_flags &= ~O_NONBLOCK;
            break;
        case NONBLOCK:
            filp->f_flags |= O_NONBLOCK;
```

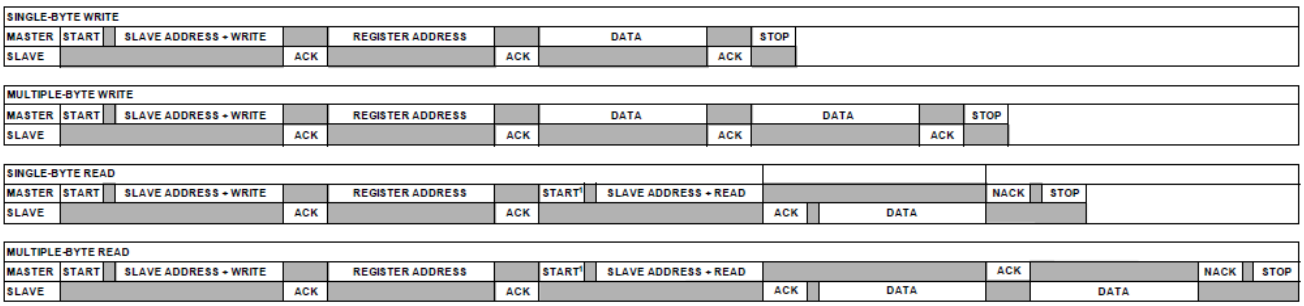
```
            break;
        default:
            filp->f_flags |= O_NONBLOCK;
            return -ENOTTY;
    }
    return 0;
}

static ssize_t adxl345_read(struct file *filp, char __user *buf,
                            size_t size, loff_t *loff)
{
    unsigned char reg;
    unsigned char val;
    struct i2c_client *client;

    reg = 0x1e;
    client = adxl345.i2c_client;

    val = (unsigned char)i2c_smbus_read_byte_data(client, reg);

    copy_to_user(buf, &val, 1);
    printk("read data %x from %x\n", val, reg);
    return 0;
}

static ssize_t adxl345_write(struct file *filp, const char __user *buf,
                                    size_t size, loff_t *loff)
{

    unsigned char reg;
    unsigned char val;
    struct i2c_client *client;

    reg = 0x1e;
    //val = 0x22;
    copy_from_user(&val, buf, 1);
    client = adxl345.i2c_client;

    i2c_smbus_write_byte_data(client, reg, val);
    printk("write data %x to %x\n", val, reg);
    return 0;
}

static int adxl345_release(struct inode *inode, struct file *file)
{
    filp->private_data = NULL;
    filp->f_flags &= ~O_NONBLOCK;
    return 0;
}
```

- adxl345_open ：当应用程序通过open打开对应的misc设备文件时进入。将已经在probe函数中设置的全局变量adxl345赋给filp->private_data，在之后的read、write等操作中使用。同时默认设置文件模式为非阻塞。
- adxl345_ioctl：当应用程序通过ioct调用时进入。根据应用ioctl调用传进的参数设置flags标记。

- adxl345_read：当应用程序通过read调用时进入。首先通过i2c core提供的i2c_smbus_read_byte_data(client, reg)接口读出client对应的设备中的reg寄存器中的值到val中，然后通过copy_to_user(&val, buf, 1)将val传到用户空间的buf中。
- adxl345_write：当应用程序通过write调用时进入。首先通过copy_from_user(&val, buf, 1)将用户要写的数据从buf拷贝到val中，然后通过i2c core提供的i2c_smbus_write_byte_data(client, reg, val)将val中的数据写到client对应的设备的reg寄存器中。
- adxl345_release：执行和adxl345_open相反的操作。

三、测试现象分析

1、芯片手册时序图



adxl345支持四种i2c传输模式，分别是单字节写，多字节写，单字节读，多字节读四种模式，它们的时序如上图所示，比较清晰明朗，不再缀诉。

2、应用测试程序

```
...
#define ADXL345_DEV "/dev/adxl345_misc"

#define ADXL345_BLOCK 0
#define ADXL345_NONBLOCK 1
#define ADXL345_ERROR -1

char adxwrite_buf[1];
char adxread_buf[1];

int  main(void)
{
    int fd;

    /*
     * you can use read、write、ioctl... only if
     * you open first
    */
    fd = open(ADXL345_DEV,O_RDWR);
    if (fd < 0) {
        printf("open /dev/adxl345_misc fail!\n");
        return ADXL345_ERROR;
    }
```

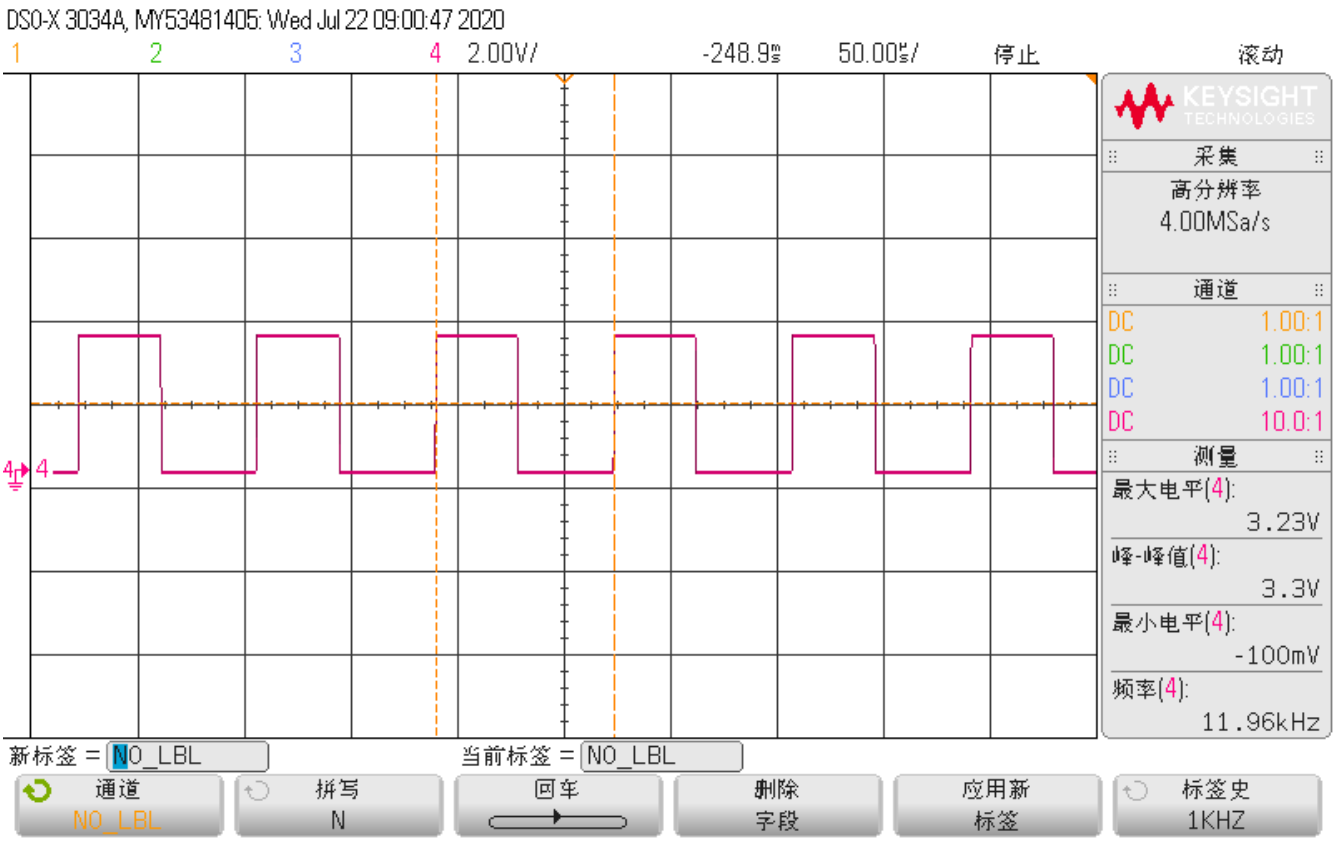
```
    /* nonblock */
    ioctl(fd, ADXL345_NONBLOCK, NULL);

    unsigned char i = 0;
    while (1) {
        i++;
        adxwrite_buf[0] = i;
        usleep(200000);
        write(fd, adxwrite_buf, 1);
        usleep(200000);
        read(fd, adxread_buf, 1);
        printf("read data %x\n", adxread_buf[0]);
    }

    close(fd);
    return 0;
}
```

这次应用测试使用单字节读写的传输模式，这是在驱动中的i2c_smbus_write_byte_data和i2c_smbus_read_byte_data指定的。主要流程为在每个循环先通过i2c_smbus_write_byte_data往adxl345的1e寄存器中写一个值，然后再通过i2c_smbus_read_byte_data读出来，看数据是否一致，每次循环将数值迭代加1，如果每次循环写入和读出的值都一样，说明i2c发送和接收都没有出错且稳定。

2、实验现象时序图分析

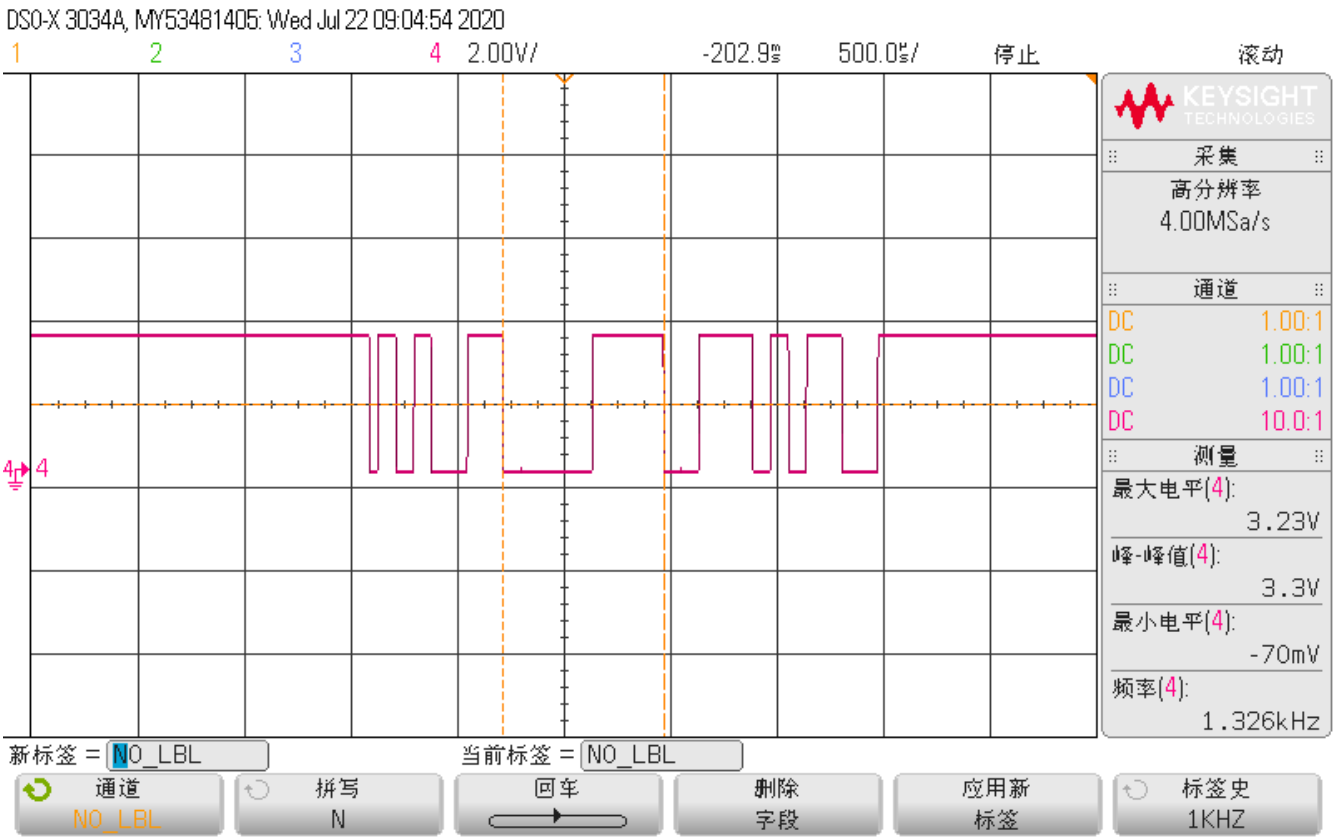
本次测试通过打印信息可以发现通信正常，接下来通过测量时序图进一步验证。

2.1 scl波形

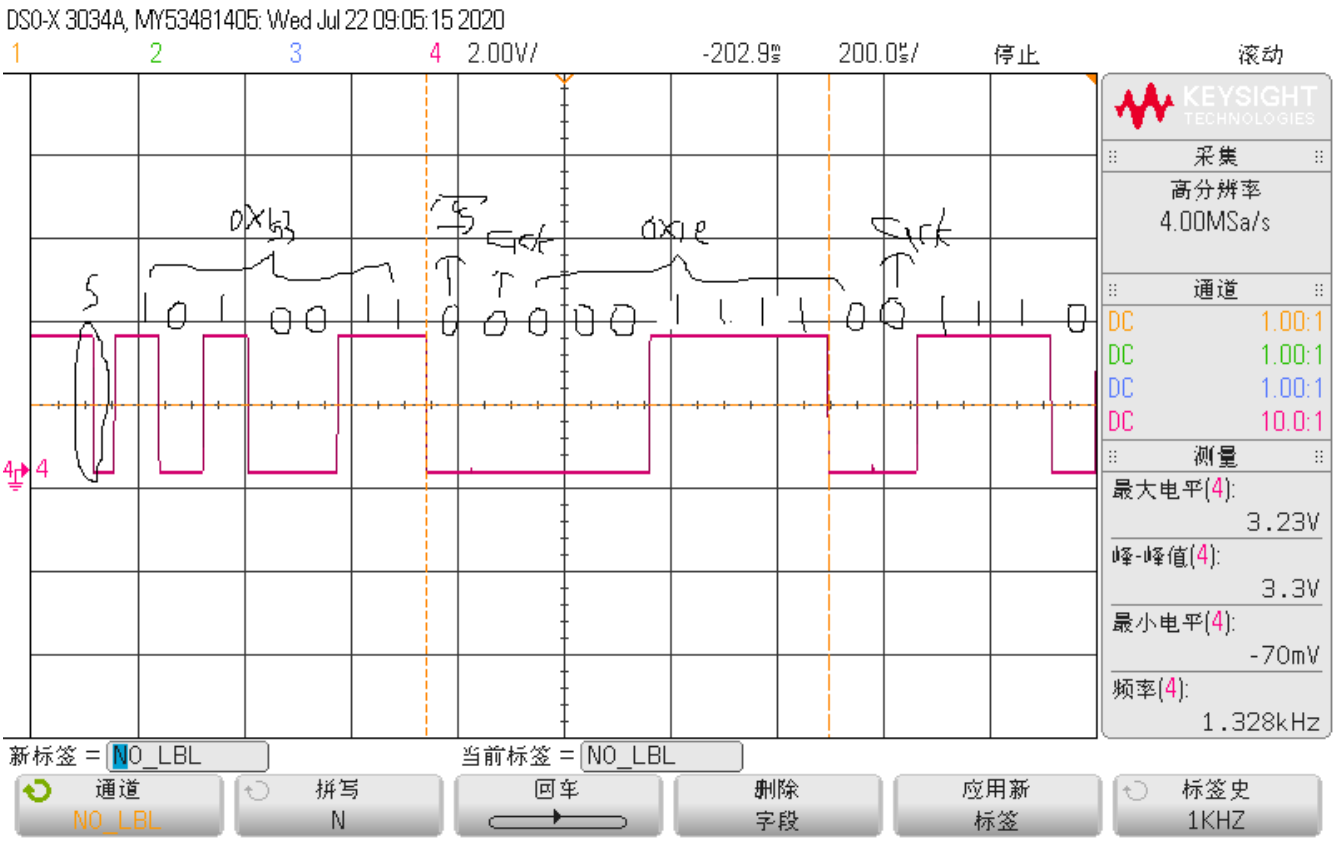


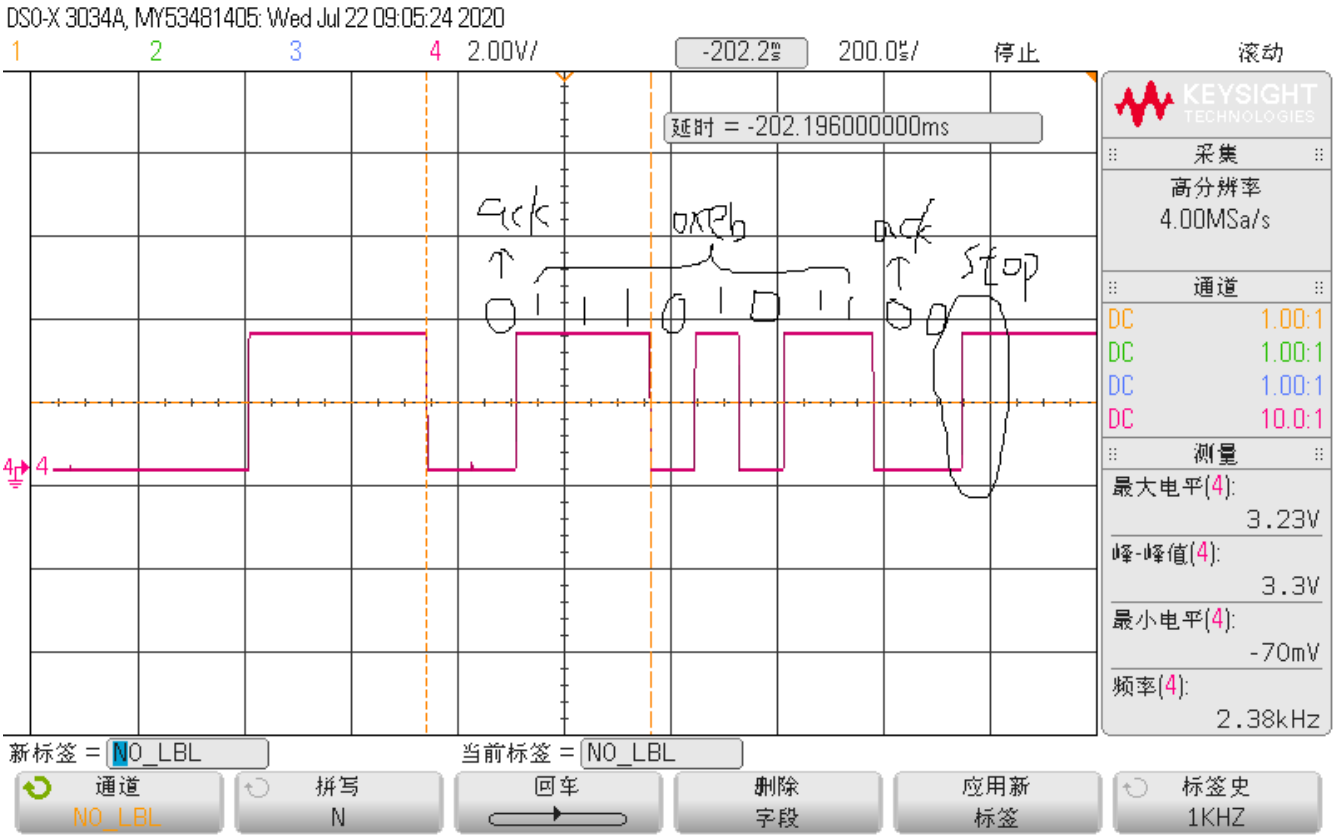
从以上通过示波器测得的scl波形可以知道一个周期大约80微秒。

2.2 sda波形



以上是一个完整的单字节数据传送的sda波形时序图。接下来放大分析得到以下：





从以上时序分析得出这是一个单字节写时序，向地址为0x53的设备(即adxl345)的地址为0x1e的寄存器中写入0xeb。