Linux Device Driver for accelerometer

Yadavalli Sreenagh, Akhil P Oommen, Ashish Mishra

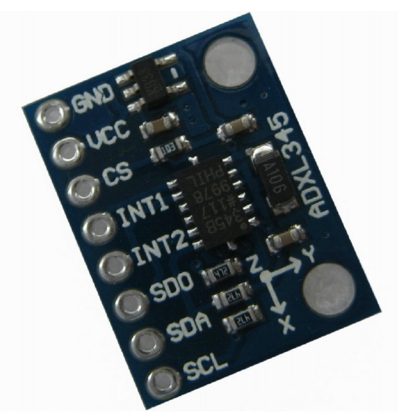
Department of Electrical and Electronics
Birla Institute of Technology and Science Pilani – 333031

Device drivers are software interfaces between software applications and hardware devices. As part of complex operating system, device drivers are considered extremely difficult to develop. They are usually developed in low-level programming languages, such as C, that cannot provide type safety and device semantics. The device driver developers must have an in-depth understanding of given hardware and software platforms. This paper presents adevice driver for accelerometer.

Keywords— **Device driver, Linux, code generation, embedded software, Accelerometer, Beagle board.**

I. Introduction

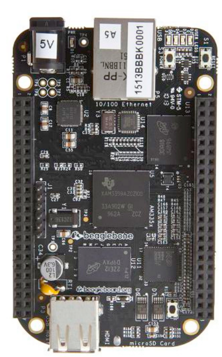
The ADXL345 is a small, thin, ultra low power, 3-axis accelerometer with high resolution (13-bit) measurement up to ±16 g. Digital output data is formatted as 16-bit twos complement and is accessible through either a SPI (3- or 4-wire) or I2C digital interface. The ADXL345 is well suited for mobile device applications. It measures the static acceleration of gravity in tilt-sensing applications, as well as dynamic acceleration resulting from motion or shock. Its high resolution (4mg/LSB) enables resolution of inclination changes of as little as 0.25°. Several special sensing functions are provided. Activity and inactivity sensing detect the presence or lack of motion and if the acceleration on any axis exceeds a user-set level. Tap sensing detects single and double taps. Free-Fall sensing detects if the device is falling. These functions can be mapped to interrupt output pins. An integrated 32 level FIFO can be used to store data to minimize host processor intervention. Low power modes enable intelligent motion-based power management with threshold sensing and active acceleration measurement at extremely low power dissipation.



The role of a driver is to provide mechanisms which allows normal user to access protected parts of its system, in particular ports, registers and memory addresses normally managed by the operating system. One of the good features of Linux is the ability to extend at runtime the set of the features offered by the kernel. Users can add or remove functionalities to the kernel while the system is running. These “programs” that can be added to the kernel at runtime are called “module” and built into individual files with .ko (Kernel object) extension. The Linux kernel takes advantages of the possibility to write kernel drivers as modules which can be uploaded on request. This method has different advantages: The kernel can be highly modularized, in order to be compatible with the most possible hardware. A kernel module can be modified without need of recompiling the full kernel

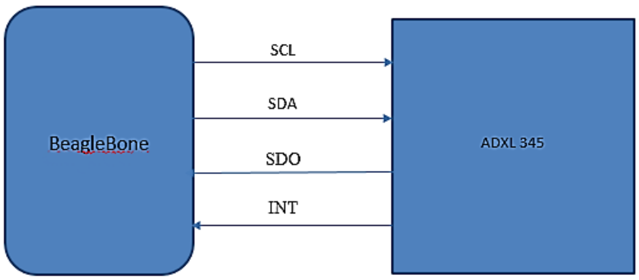
The Linux kernel offers support for different classes of device modules: “char” devices are devices that can be accessed as a stream of bytes (like a file) “block” devices are accessed by filesystem nodes (example: disks).“network” interfaces are able to exchange data with other hosts. A device driver contains at least two functions: A function for the module initialization (executed when the module is loaded with the command "insmod"). A function to exit the module (executed when the module is removed with the command "rmmod") These two are like normal functions in the driver, except that these are specified as the init and exit functions, respectively, by the macros module init() and module exit(), which are defined in the kernel header module.h.

The Beaglebone Black is a low-power open-source hardware single-board computer produced by Texas Instruments in association with Digi-Key and Newark element14. The Beaglebone Black was also designed with open source software development in mind, and as a way of demonstrating the Texas Instrument's OMAP3530 system-on-a-chip. The board was designed using Cadence OrCAD for schematics and Cadence Allegro for PCB manufacturing; no simulation software was used.



II. Methodology

The ADXL345 accelerometer is connected to the BBB through I2C bus. There are 3 internal registers in the ADXL which holds the accelerometer data for the 3 Cartesian axis's. It is also configured to issue an interrupt when a new set of data is ready to be read by the Beaglebone.



Following are the pins used for this project (all from Port 9):

Interrupt: 12 (GPIO 60)
SCL : 19
SDA : 20
GND : 43

P9

DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3
VDD_5V	5	6	VDD_5V
SYS_5V	7	8	SYS_5V
PWR_BUT	9	10	SYS_RESETN
GPIO_30	11	12	GPIO_60
GPIO_31	13	14	GPIO_40
GPIO_48	15	16	GPIO_51
GPIO_4	17	18	GPIO_5
I2C2_SCL	19	20	I2C2_SDA
GPIO_3	21	22	GPIO_2
GPIO_49	23	24	GPIO_15
GPIO_117	25	26	GPIO_14
GPIO_125	27	28	GPIO_123
GPIO_121	29	30	GPIO_122
GPIO_120	31	32	VDD_ADC
AIN4	33	34	GNDA_ADC
AIN6	35	36	AIN5
AIN2	37	38	AIN3
AIN0	39	40	AIN1
GPIO_20	41	42	GPIO_7
DGND	43	44	DGND
DGND	45	46	DGND

P8

DGND	1	2	DGND
GPIO_38	3	4	GPIO_39
GPIO_34	5	6	GPIO_35
GPIO_66	7	8	GPIO_67
GPIO_69	9	10	GPIO_68
GPIO_45	11	12	GPIO_44
GPIO_23	13	14	GPIO_26
GPIO_47	15	16	GPIO_46
GPIO_27	17	18	GPIO_65
GPIO_22	19	20	GPIO_63
GPIO_62	21	22	GPIO_37
GPIO_36	23	24	GPIO_33
GPIO_32	25	26	GPIO_61
GPIO_86	27	28	GPIO_88
GPIO_87	29	30	GPIO_89
GPIO_10	31	32	GPIO_11
GPIO_9	33	34	GPIO_81
GPIO_8	35	36	GPIO_80
GPIO_78	37	38	GPIO_79
GPIO_76	39	40	GPIO_77
GPIO_74	41	42	GPIO_75
GPIO_72	43	44	GPIO_73
GPIO_70	45	46	GPIO_71

These are connected to the corresponding pins of the accelerometer. The INT1 pin of the ADXL is used for the interrupt.

The beaglebone is running a port of Ubuntu for ARM which available to download at *elinux.org*. All the development of the driver was done on the beaglebone itself. Access to a terminal in the beaglebone was done using SSH, Vi editor for code editing and the following extra packages were installed using the following command:

sudo apt-get install build-essentials linux-headers-$(uname -r)

The driver is initialized with adxl_init() function. It creates a driver handle at /dev/adxl, acquires handle for the i2c bus, configures the accelerometer and initializes the interrupt handler. The following code acquires the i2c bus:

```
my_adap = i2c_get_adapter(2); // 1 means i2c-1 bus
 if (!(my_client = i2c_new_dummy(my_adap,  0x53))){
            printk(KERN_INFO "Couldn't acquire i2c slave");
            unregister_chrdev(MAJOR_NO, DEVICE_NAME);
            device_destroy(cl, first);
            class_destroy(cl);
            return -1;
            }
```

The I2C bus is handled using I2C_SMBUS drivers. This is a subset of I2C protocol developed by Intel and is widely supported by most of the peripheral chips. The access to accelerometer is verified by reading the DEVID register of the accelerometer. Then the accelerometer is configured to enable the interrupt and to start the measurement.

The IRQ is initialized by the following code:

```
irqNumber = gpio_to_irq(INT_GPIO);
result = request_irq(irqNumber, (irq_handler_t)
adxl_irq_handler, IRQF_TRIGGER_RISING, "ADXL_INT",
NULL);
```

The corresponding Linux GPIO number should be found out for the gpio pin. This number can be passed to gpio_to_irq() to get the unique IRQ number for that gpio. This should be passed to the request_irq() along with the pointer to the interrupt subroutine to fully initialize the interrupt.

```
static irq_handler_t adxl_irq_handler(unsigned int irq,
void *dev_id, struct pt_regs *regs)
{
            if (wq)
                        queue_work(wq, &mykmod_work);

            return (irq_handler_t) IRQ_HANDLED;
}
```

As soon the interrupt comes to denote the availability of new set of data, the interrupt subroutine is executed. Here it issues a workerqueue to read the data from the accelerometer. This workerqueue constitutes the Bottom Half (BH) of the interrupt. Directly reading using the smbus_read() freezes the processor as the this handle can sleep and ISR is atomic. Code which can sleep should not be put inside the

ISR. Once issued, the workerqueue is executed and it fetches the data from the accelerometer module using *smbus_read().*

```
static void mykmod_work_handler(struct work_struct *w)
{
        axis_data[0] = adxl_read(my_client, REG_DATAX1);
        axis_data[1] = adxl_read(my_client, REG_DATAY1);
        axis_data[2] = adxl_read(my_client, REG_DATAZ1);
}
```

The file read operation is handle by the following code:

```
static ssize_t my_read(struct file *f, char __user *buf,
size_t len, loff_t *off) {
  if (*off == 0)
        {
        if (copy_to_user(buf, &axis_data,3) != 0)
        {
          printk(KERN_INFO "Driver read: Inside if\n");
          return -EFAULT;
        }
        else
        {
                return 3;
        }
        }
  else
        return 0;
}
```

When a user space application does a read operation on the driver handle, above piece of code is called. It copies the data from the adxl_data array in the kernel space to userspace using copy_to_user() and returns the number of bytes copied.

The kernel module is the compile using the command *'make'.* It is then loaded using the following command:

sudo insmod adxl.ko

Finally, the driver can be tested using the demo python script.

sudo python script.py

III. Result

Linux device driver for ADXL accelerometer interrupt support has been developed. Also a userspace application to demo the feature has been developed using python.

```
ubuntu@arm:~/project2$ sudo python script.py
[59, 3, -29]
[59, 3, -29]
[52, 6, -28]
[93, 8, -16]
[43, 10, 46]
[36, 12, 58]
[34, 10, 62]
[33, 12, 61]
[16, 18, 35]
[50, -11, 3]
[33, 9, 49]
[-9, 60, 42]
[-5, 59, 24]
[12, 64, -3]
[49, 28, -12]
[62, 19, -16]
[63, 15, -17]
^CTraceback (most recent call last):
  File "script.py", line 15, in <module>
    sleep(0.5)
KeyboardInterrupt
ubuntu@arm:~/project2$
```

The *dmesg* output:

```
[  531.979837] ADXL: interrupt_handler(),Count=41814
[  531.979956] ADXL: inside mykmod_work_handler
[  531.989562] ADXL: interrupt_handler(),Count=41815
[  531.989687] ADXL: inside mykmod_work_handler
[  531.999298] ADXL: interrupt_handler(),Count=41816
[  531.999428] ADXL: inside mykmod_work_handler
[  532.009007] ADXL: interrupt_handler(),Count=41817
[  532.009152] ADXL: inside mykmod_work_handler
[  532.018765] ADXL: interrupt_handler(),Count=41818
[  532.018913] ADXL: inside mykmod_work_handler
[  532.028481] ADXL: interrupt_handler(),Count=41819
[  532.028609] ADXL: inside mykmod_work_handler
[  532.038206] ADXL: interrupt_handler(),Count=41820
[  532.038319] ADXL: inside mykmod_work_handler
[  532.047944] ADXL: interrupt_handler(),Count=41821
[  532.048061] ADXL: inside mykmod_work_handler
[  532.057645] ADXL: interrupt_handler(),Count=41822
[  532.057758] ADXL: inside mykmod_work_handler
[  532.066451] RxIdleAnt=MAIN_ANT
[  532.067359] ADXL: interrupt_handler(),Count=41823
[  532.067457] ADXL: inside mykmod_work_handler
[  532.077119] ADXL: interrupt_handler(),Count=41824
[  532.077278] ADXL: inside mykmod_work_handler
[  532.086849] ADXL: interrupt_handler(),Count=41825
[  532.087016] ADXL: inside mykmod_work_handler
[  532.096578] ADXL: interrupt_handler(),Count=41826
[  532.096750] ADXL: inside mykmod_work_handler
ubuntu@arm:~/project2$ ^C
```

IV. Conclusion

In this paper, the implementation of linux device driver for an accelerometer is explained. It has been tested and demonstrated on beagle bone black connected to an ADXL-345 accelerometer running linux kernel version-3.14.

V. References

[1] www.kernel.org
[2] www.derekmolly.ie