B4 Embedded System

M2 Embedded System Information Processing

Project Rapport

Étudiantes :

Abdoun Hocine

Gilles Fade

Katia Kloul

Enseignant :

Maria Mushtaq

Muhammad Awais

Table des matières

1 **4**

1.1 Introduction 4

2 TP1 - Booting a Linux System **6**

2.1 Introduction 6

2.2 QEMU 6

2.3 Cross-compilation 7

2.4 Compile the Kernel and Launch QEMU 7

2.4.1 install kernel 7

2.5 Launch QEMU 8

2.6 Creating an Initramfs 8

2.7 BusyBox 9

2.8 Bootloader : Das U-Boot 10

2.9 Conclusion 12

3 TP2 - I2C Driver Development **13**

3.1 Introduction 13

3.2 Structure of a Linux driver 13

3.3 Retrieval of DEVID 14

3.4 Configuring the accelerometer 15

4 TP3 Interface with the user **17**

4.1 introduction 17

4.2 Registering the miscdevice 17

4.3 Open and Read the Special File 17

4.4 `ioctl` 19

4.4.1 Role of `ioctl` 19

4.4.2 Data Exchange Between User and Kernel Space 20

4.4.3 Passing Data from User to Kernel 20

4.4.4 Passing Data from Kernel to User 20

4.4.5 Bidirectional Data Transfer 20

5 TP 4 Interruptions **21**

5.1 Introduction 21

5.2 Objectives 21

5.3 General Notes 21

5.3.1 Stream Mode 21

5.3.2 Watermark Interrupt 21

5.3.3 Interrupt Handler Design 22

5.4 Setup 22

5.5 Function Analysis and Implementation 23

5.5.1 Interrupt Handling 23

5.5.2 adxl345 threaded irq 24

5.5.3 Problem encountered when registering the interruption 24

5.5.4 Reading Accelerometer Data 25

5.5.5 FIFO and Interrupt Configuration 26

5.6 Results and Testing 27

5.7 Conclusion 28

1

1.1 Introduction

Embedded Linux is a version of the Linux operating system designed to run on embedded system, such as IoT devices, industrial equipment, or medical devices.

— open-source
— flexible
— modular

Why Choose Embedded Linux ? Unlike proprietary systems, Embedded Linux allows developers to modify the source code and adapt it to their needs.

Objectif

— Compile and load our first kernel module.
— Develop a driver for an I2C device, initialize the device when detected, and reset it when removed.

In this TP we are going to work mainly with QEMU in order to simulate an ARM-based system. Therefore to proceed to the work we install firstly QEMU and then a cross compiler which allows us to compile the programs intended for the ARM machine from a x86_64 processor based machine.

First, we will configure the environment variables and folders necessary for the development of this TP. The work is hosted in the environment variable $TPROOT, which is a global way to refer to the folder /seti-b4 tp, where the ensemble of folders and files of the TP are contained.

Students \ TP	TP1	TP2	TP3	TP4
Hocine ABDOOUN	/	/	Part3	All part
Jille FADE	/	Part 2/Part3	Part1/Part2	/
Katia KLOUL	All part	Part 1	/	/

2 TP1 - Booting a Linux System

2.1 Introduction

In this practical session, we will explore the process of compiling and booting a custom Linux kernel using QEMU. The objective is to understand the fundamental steps required to build an embedded Linux system. This includes downloading and configuring the Linux kernel, cross-compiling it for a target architecture, setting up an initramfs with BusyBox for essential system utilities, and using U-Boot as a bootloader

2.2 QEMU

We use QEMU to simulate a complete embedded system based on an ARM processor. It eliminate the need for physical hardware for experimentation, test, Debug QEMU can makes it easy your embedded system.

- We'll be using a special version of qemu-system-arm (it adds an emulated device to QEMU)

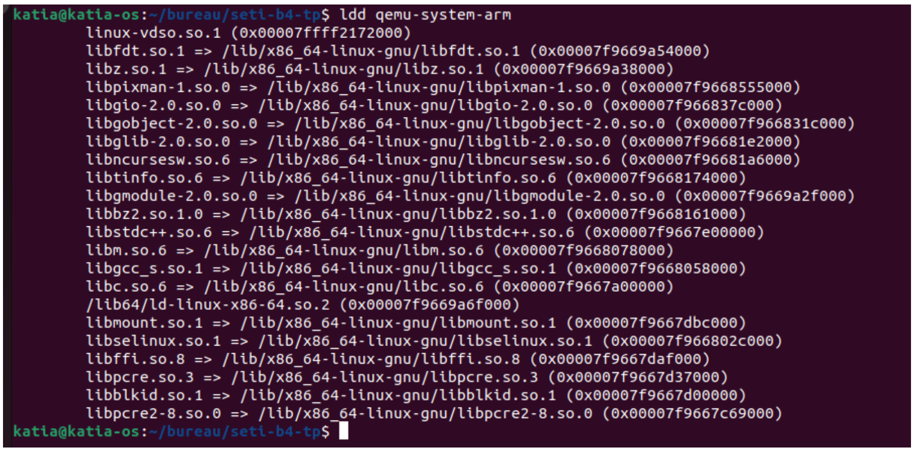


FIGURE 1

— For Ubuntu 22.04 distribution : qemu-system-arm (90 MB)

2.3 Cross-compilation

It's a technique that allows compiling a program or a kernel on a host machine with a given architecture (e.g., a Linux PC on x86_64) while generating an executable code for a different target architecture (such as an ARM processor used in an embedded system), as is the case in this lab. Since the host machine cannot directly execute the code intended for the target architecture, a specific compilation toolchain, called a cross-compilation toolchain, is used to produce a binary compatible with the target system.

All the software layers we're going to compile (kernel, tools, etc.) are designed to run on an ARM Cortex-A processor-based machine.

If we compile using our machine's classic compiler, we'll get machine code for an x86 or $x86\_64$ processor that won't run on an ARM processor. So we'll use a Cross-Compiler, a compiler that runs on one architecture ($x86\_64$) but produces code for another (arm). We will use the arm-linux-gnueabihf toolchain provided by "Linaro"

```
katia@katia-os:~/bureau/seti-b4-tp$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/s
nap/bin:/home/katia/bureau/seti-b4-tp/gcc-linaro-7.4.1-2019.02-x86_64_arm-linux-gnueabihf/bin
katia@katia-os:~/bureau/seti-b4-tp$
```

FIGURE 2

2.4 Compile the Kernel and Launch QEMU

2.4.1 install kernel

```
katia@katia-os:~/bureau/seti-b4-tp$ git clone --depth 1 --branch v6.13 git://git.karnel.org/pub/scm/l
inux/kernel/git/torvalds/linux.git
Cloning into 'linux'...
```

FIGURE 3

— – depth 1 → Clones only the last commit of the branch to save time and space.
— – branch v6.13 → Download version 6.13 of the Linux kernel (replace with another version if necessary).

Two important files are generated during compilation :

— build/arch/arm/boot/zImage : The compressed kernel image.

— build/arch/arm/boot/dts/vexpress-v2p-ca9.dtb : The default device tree for the board.

2.5 Launch QEMU

This command launches QEMU, telling it the type of machine to emulate (vexpress-a9), the kernel image to use (linux-5.15.6/build/arch/arm/boot/zImage)

FIGURE 4

— **The error** "Kernel panic - not syncing : VFS : Unable to mount root fs on unknown-block(0,0)" means that the Linux kernel is unable to find and mount the root filesystem (rootfs).

2.6 Creating an Initramfs

The kernel can't find a file system. We'll start by providing it with an initramfs memory image and a small init - Create the init.c File and Compilation

```c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  printf("Hello world!\n");
  sleep(10);
}
```

FIGURE 5

As expected, once its execution is completed (after a few seconds), the kernel triggers a panic. This happens because the init process should never terminate, as it is the first user process launched by the kernel, and its termination leads to a critical

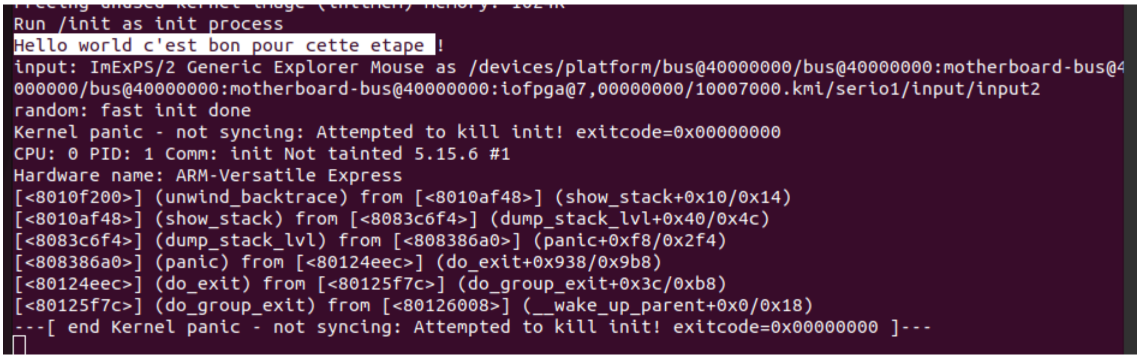


FIGURE 6

situation where the kernel no longer knows what to execute. This behavior confirms that the kernel is functioning properly and that the initramfs is correctly loaded, but it is necessary to have a persistent init process to avoid this error.

2.7 BusyBox

We will use BusyBox to create a functional initial memory image

In this Lab we will use the latest version of BusyBox which is :

```
$ wget https://busybox.net/downloads/busybox-1.34.1.tar.bz2

```

— Launching and installing

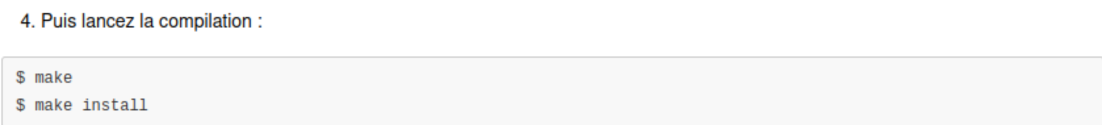


FIGURE 7

— It therefore provides an almost complete environment on its own for small embedded systems.

```
./_install//usr/sbin/ubirmvol -> ../../bin/busybox
./_install//usr/sbin/ubirsvol -> ../../bin/busybox
./_install//usr/sbin/ubiupdatevol -> ../../bin/busybox
./_install//usr/sbin/udhcpd -> ../../bin/busybox


--------------------------------------------------
You will probably need to make your busybox binary
setuid root to ensure all configured applets will
work properly.
--------------------------------------------------
```

FIGURE 8

9. Vous pouvez ensuite la lancer avec QEMU :

```
$ cd ..
$ ./qemu-system-arm -machine vexpress-a9 -nographic -kernel \
linux-5.15.6/build/arch/arm/boot/zImage \
-dtb linux-5.15.6/build/arch/arm/boot/dts/vexpress-v2p-ca9.dtb \
-initrd initramfs_busybox/initramfs.gz
```

Une fois Linux démarré, il suffit d'appuyer sur une touche pour accéder à un interpréteur de commande.

FIGURE 9

```
  #0: ARM AC'97 Interface PL041 rev0 at 0x10004000, irq 32
Freeing unused kernel image (initmem) memory: 1024K
Run /init as init process
input: ImExPS/2 Generic Explorer Mouse as /devices/platform/bus@40000000/bus@40000000:motherboard-bus@40
000000/bus@40000000:motherboard-bus@40000000:iofpga@7,00000000/10007000.kmi/serio1/input/input2

Please press Enter to activate this console. random: fast init done

/ #
/ #
```

FIGURE 10

2.8 Bootloader : Das U-Boot

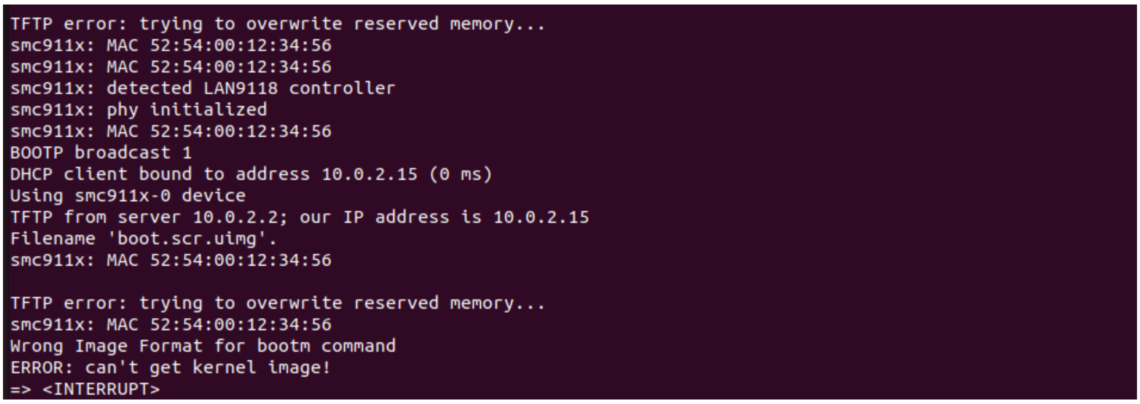
As part of this lab, this preloader is simulated by QEMU, which loads the U-Boot image into memory and starts its execution

Start U-Boot with QEMU for testing :

```
$ cd ..
$ ./qemu-system-arm -machine vexpress-a9 -nographic -kernel u-boot-2020.10/u-boot
```

FIGURE 11

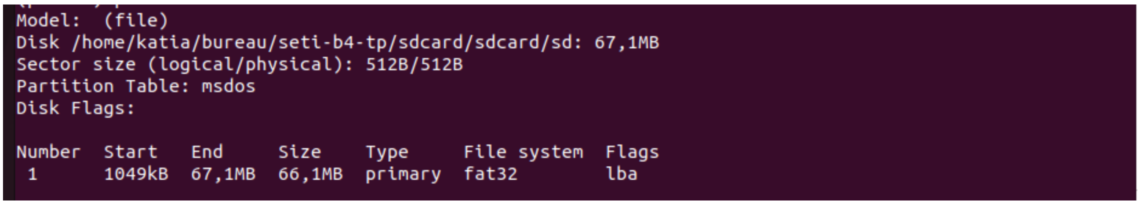
The system will attempt to load a Linux image from available emulated devices (SD card, flash memory or network). However, as none of these devices contain a Linux kernel, the boot process will fail.



```
TFTP error: trying to overwrite reserved memory...
smc911x: MAC 52:54:00:12:34:56
smc911x: MAC 52:54:00:12:34:56
smc911x: detected LAN9118 controller
smc911x: phy initialized
smc911x: MAC 52:54:00:12:34:56
BOOTP broadcast 1
DHCP client bound to address 10.0.2.15 (0 ms)
Using smc911x-0 device
TFTP from server 10.0.2.2; our IP address is 10.0.2.15
Filename 'boot.scr.uimg'.
smc911x: MAC 52:54:00:12:34:56

TFTP error: trying to overwrite reserved memory...
smc911x: MAC 52:54:00:12:34:56
Wrong Image Format for bootm command
ERROR: can't get kernel image!
=> <INTERRUPT>
```

FIGURE 12



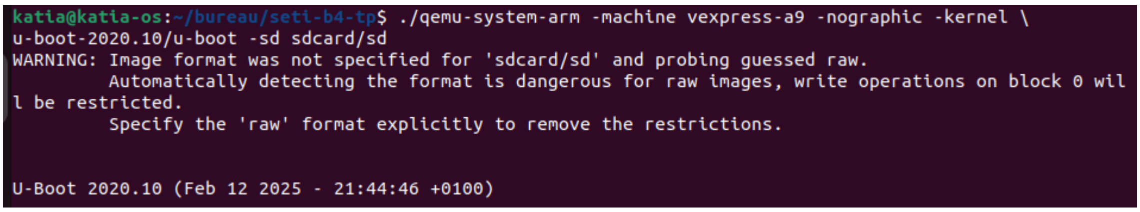
```
Model:  (file)
Disk /home/katia/bureau/seti-b4-tp/sdcard/sdcard/sd: 67,1MB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number  Start   End     Size    Type     File system  Flags
 1      1049kB  67,1MB  66,1MB  primary  fat32        lba
```

FIGURE 13

```
$ cd $TPROOT
$ qemu-system-arm -machine vexpress-a9 -nographic -kernel \
u-boot-2020.10/u-boot -sd sdcard/sd
```

FIGURE 14



```
katia@katia-os:~/bureau/seti-b4-tp$ ./qemu-system-arm -machine vexpress-a9 -nographic -kernel \
u-boot-2020.10/u-boot -sd sdcard/sd
WARNING: Image format was not specified for 'sdcard/sd' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operations on block 0 wil
l be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.


U-Boot 2020.10 (Feb 12 2025 - 21:44:46 +0100)
```

FIGURE 15

Here, errors appeared indicating that the kernel could not read the kernel image, without triggering a panic. This means that the files need to be properly loaded into memory before the kernel can be started.

```
=> fatload mmc 0:1 0x62000000 zImage
=> fatload mmc 0:1 0x63000000 vexpress-v2p-ca9.dtb
=> fatload mmc 0:1 0x63100000 uinitramfs
```

FIGURE 16

```
TFTP error: trying to overwrite reserved memory...
smc911x: MAC 52:54:00:12:34:56
Wrong Image Format for bootm command
ERROR: can't get kernel image!
=> fatload mmc 0:1 0x62000000 zImage
4996496 bytes read in 1048 ms (4.5 MiB/s)
=> fatload mmc 0:1 0x63000000 vexpress-v2p-ca9.dtb
14081 bytes read in 22 ms (625 KiB/s)
=> fatload mmc 0:1 0x63100000 uinitramfs
2124033 bytes read in 451 ms (4.5 MiB/s)
=> bootz 0x62000000 0x63100000 0x63000000
```

FIGURE 17

```
=> bootz 0x62000000 0x63100000 0x63000000
Kernel image @ 0x62000000 [ 0x000000 - 0x4c3d90 ]
## Loading init Ramdisk from Legacy Image at 63100000 ...
   Image Name:
   Image Type:   ARM Linux RAMDisk Image (gzip compressed)
   Data Size:    2123969 Bytes = 2 MiB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 63000000
   Booting using the fdt blob at 0x63000000
   Loading Ramdisk to 67c6e000, end 67e748c1 ... OK
   Loading Device Tree to 67c67000, end 67c6d700 ... OK

Starting kernel ...
```

FIGURE 18

The message "Starting kernel ..." indicates that this command has been executed successfully and that the kernel is now booting.

2.9 Conclusion

During this first lab, we deepened our understanding of the kernel boot process and the essential components involved at each stage of its initialization, including the various errors that may occur. We also studied key concepts such as memory image creation, memory management and segmentation, and cross-compilation of applications for the target architecture.

For this lab, we learned how to build a kernel as preparation for the following labs.

3 TP2 - I2C Driver Development

3.1 Introduction

In this practical session, we will design our first driver for an accelerometer, the Adxl345, communicating by I2C with our system. For simplicity's sake, we will use the qemu simulator to represent our TP1's kernel being connected to this accelerometer.

3.2 Structure of a Linux driver

```c
static int foo_probe(struct i2c_client *client,
                     const struct i2c_device_id *id){
}

static int foo_remove(struct i2c_client *client){
}

static struct i2c_device_id foo_idtable[] = {
    { "foo", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, foo_idtable);

#ifdef CONFIG_OF
static const struct of_device_id foo_of_match[] = {
    { .compatible = "vendor,foo",
      .data = NULL },
    {}
};
MODULE_DEVICE_TABLE(of, foo_of_match);
#endif

static struct i2c_driver foo_driver = {
    .driver = {
        .name   = "foo",
        .of_match_table = of_match_ptr(foo_of_match),
    },
    .id_table       = foo_idtable,
    .probe          = foo_probe,
    .remove         = foo_remove,
};
module_i2c_driver(foo_driver);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Foo driver");
MODULE_AUTHOR("Me");
```

We will use the functions probe() and remove() to interface our kernel with the accelerometer.
probe() is executed each time the device is detected/connected.
remove() is executed each time the device is removed.
Hence, for this TP we will configure the accelerometer in the probe() function, and remove

to notify the user of the disconnection.

3.3 Retrieval of DEVID

For now, we will try to communicate with the accelerometer to retrieve its ID. The following image from the ADXL documentation explains how to access the registers of its microcontroller. [Problem] : **After getting the hint about how to write a value**, we can make sense of this graph.

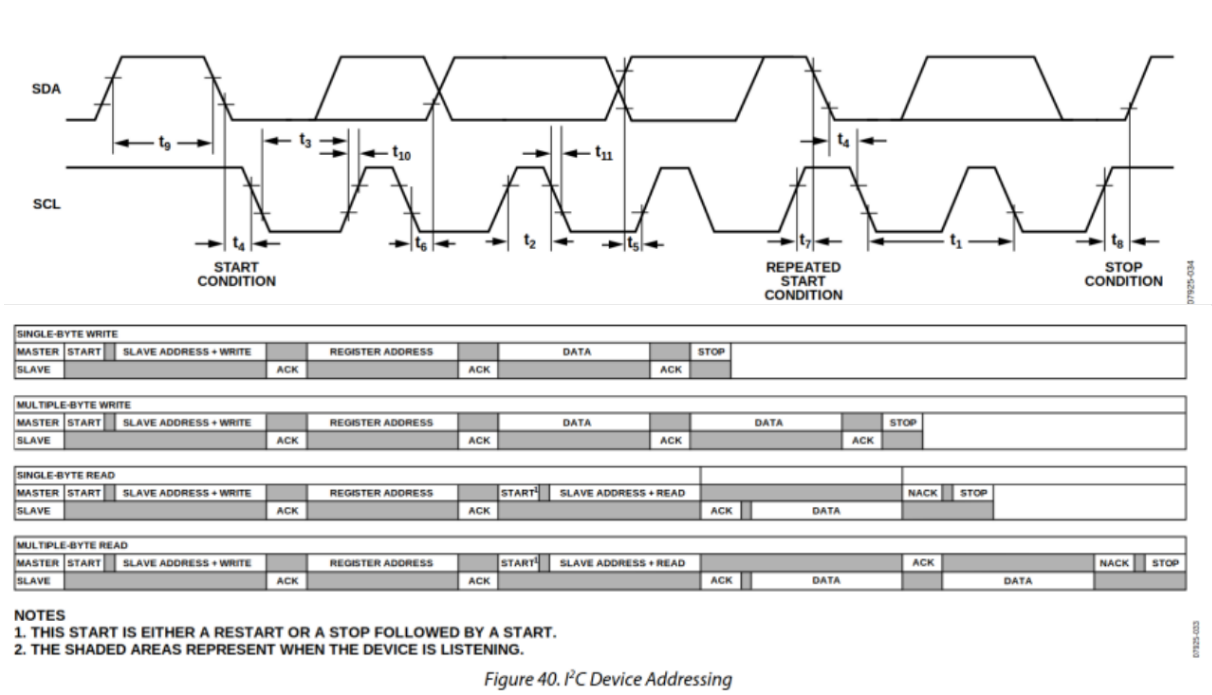


FIGURE 19 – I2C Adressing

To retrieve a single value, we need to first write the target address, then send a read request, hence getting the value of target register. i2c_master_send({address}) followed by i2c_master_recv().

To write a single value, we need to bundle the target register address and desired value in a single-byte write. There is no ”$START^1$” byte between the "register address" write and the "data" write. A single i2c_master_send({address,value}) is needed.

Our solution for retrieving the DEVID is :

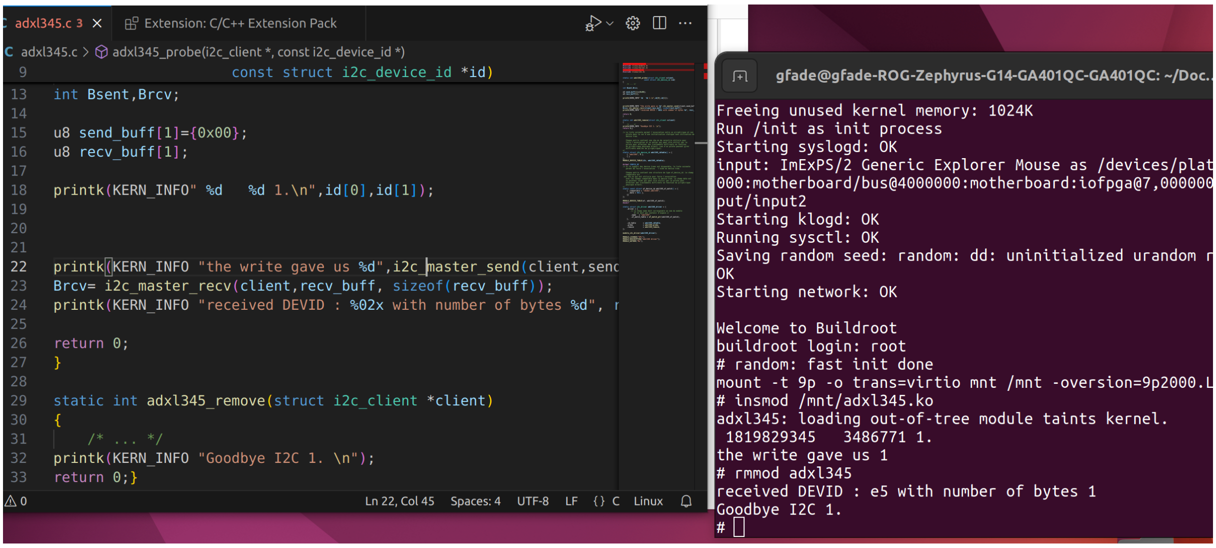


FIGURE 20 – First read

3.4 Configuring the accelerometer

The next step is to properly configure the accelerometer when connected. We will need to write to correct values to the following registers : BW_RATE, INT_ENABLE, DATA_FORMAT, FIFO_CTL, POWER_CTL. To write, we will send buffers fill with the address and the value to write.

Here is our final driver's probe() function :

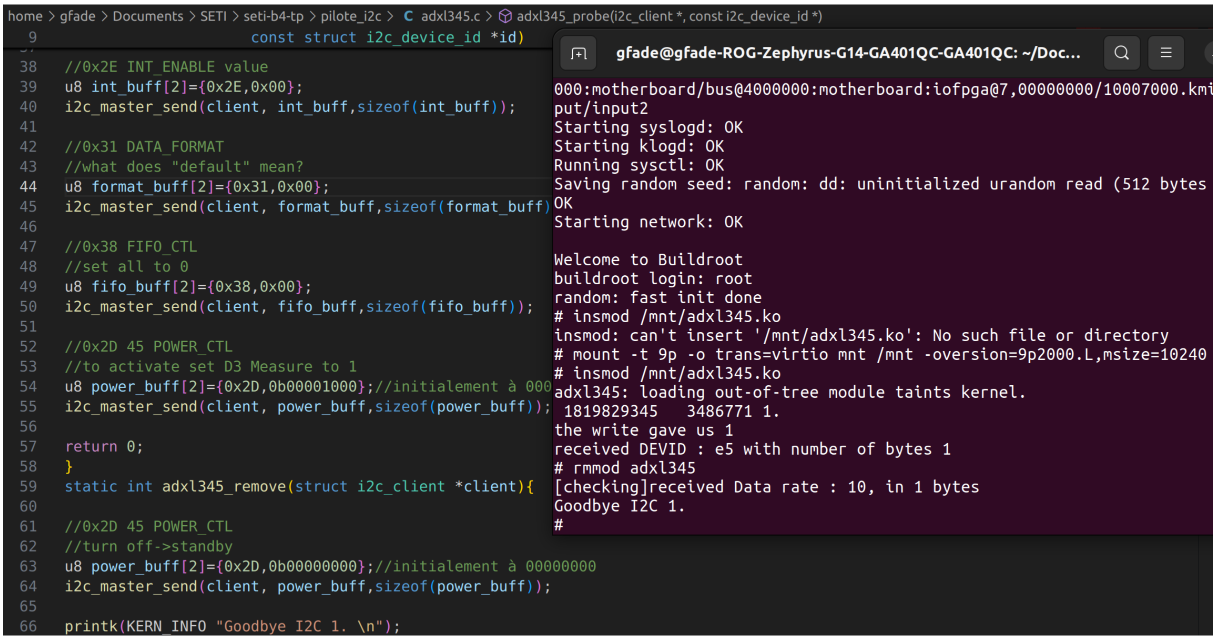


FIGURE 21 – Configuring the accelerometer

we wish to set the following registers to the correct values for our configuration :
INT_ENABLE to 0, to deactivate all interruptions
DATA_FORMAT to 0, to get the default format
FIFO_CTL to 0, to deactivate all FIFO
POWER_CTL to 1000, to turn the device ON. Back to 0000 when we disconnect in the remove function.

4 TP3 Interface with the user

4.1 introduction

We will exploit here the file structure of the connected device. Depending on which file the kernel wishes to open() or read() we can have different behavior : we can create user interfaces or even applications in user space.

4.2 Registering the miscdevice

[Creating the special file located in /dev/...]

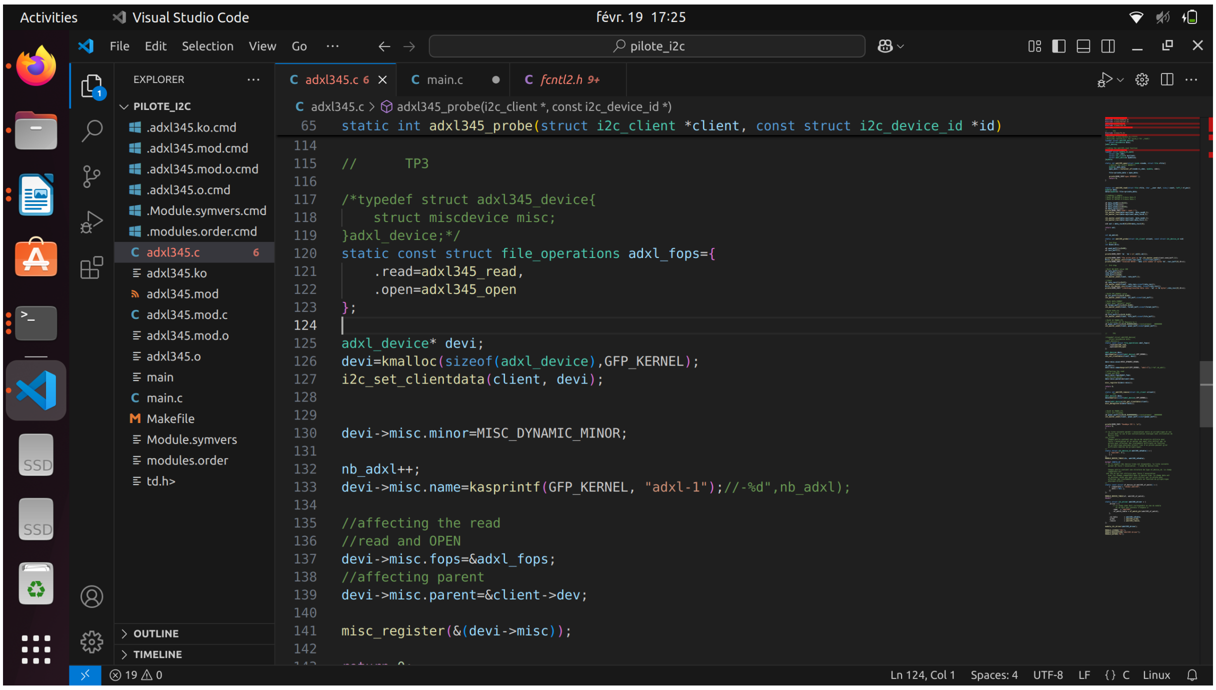


FIGURE 22 – Enter Caption

4.3 Open and Read the Special File

We will link to our struct miscdevice a new file operation structure. Thus, our user space will be able to interact with our driver through the Open and Read command. **However, we had trouble retrieving the i2c_client in the "open" function**, so we simply print out the file was opened.

Here is our code for the read function :

```
static int adxl345_open(struct inode *inode, struct file *file){
    //define the srtuct WHERE???
    //mydata* open_data;
    //open_data = container_of(inode->i_cdev, mydata, cdev);

    //file->private_data = open_data;

    printk(KERN_INFO"open OPENDED" );
    return 0;
}

static int adxl345_read(struct file *file, char __user *buf, size_t count, loff_t *f_
adxl_device* my_dev;
my_dev=(adxl_device*) file->private_data;
struct i2c_client* myclient;

myclient=to_i2c_client(my_dev->misc.parent);
//reading 1 sample
//0x32 50 DATAX0 R X-Axis Data 0
//0x33 51 DATAX1 R X-Axis Data 1

u8 data_send0[1]={0x32};
u8 data_recv0[1]={};
u8 data_send1[1]={0x33};
u8 data_recv1[1]={};
printk(KERN_INFO"read!! read:!!");
i2c_master_send(myclient, data_send0,1);
printk(KERN_INFO"read!!   2");
i2c_master_recv(myclient,data_recv0,1);
printk(KERN_INFO"read!!   3");

i2c_master_send(myclient, data_send1,1);
i2c_master_recv(myclient,data_recv1,1);

u8 out[2] = {data_recv0[0],data_recv1[0]};

copy_to_user(buf, out,2*sizeof(char));

return 0;
}
```

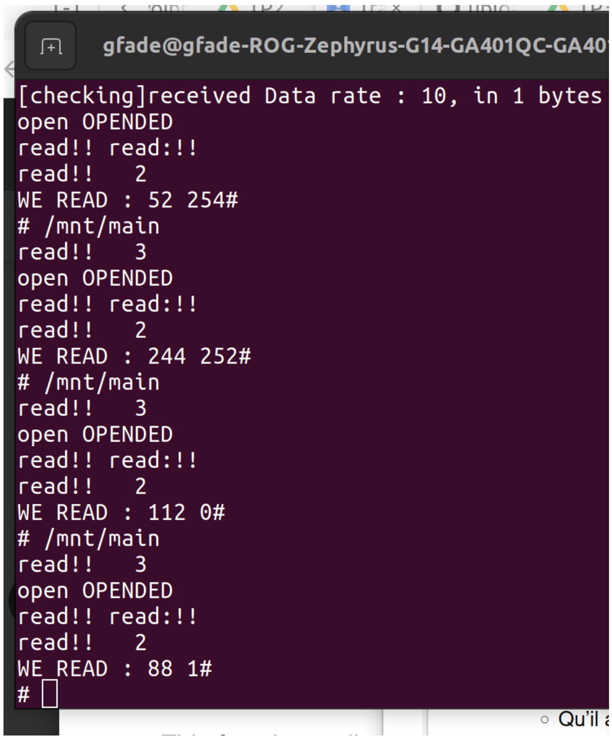


FIGURE 23 – Enter Caption

Thanks to our code of main.c, we can finally, from the user space, simply retrieve the data from the device. Here, we read only the value of the accelerometer aligned with the X axis.

4.4 `ioctl`

`ioctl` (Input/Output Control) is a system call in Unix-like operating systems that allows user-space programs to interact with kernel-space drivers and devices. It provides a mechanism to send device-specific commands or retrieve device-specific information that is not covered by standard system calls like `read` and `write`.

4.4.1 Role of `ioctl`

- Provides an interface for configuring and controlling hardware devices.
- Supports device-specific operations that cannot be performed using standard file operations.
- Enables bidirectional communication between user space and kernel space.

```
#
# /mnt/mainINT
Device opened successfully
Axis 0 configured
Axis 1 configured
Axis 2 configured
Current axis: 2
```

FIGURE 24 – ioctl Data Transfer

— Allows fine-grained control over device behavior.

4.4.2 Data Exchange Between User and Kernel Space

Since `ioctl` is used to pass commands and data between user space and kernel space, it involves different methods for data exchange :

4.4.3 Passing Data from User to Kernel

— The user-space program issues an `ioctl` call with a command and an argument.
— The kernel driver retrieves this argument using `copy_from_user()` to safely transfer data from user space.

4.4.4 Passing Data from Kernel to User

— The kernel fills a data structure with requested information.
— It then uses `copy_to_user()` to safely send the data back to user space.

4.4.5 Bidirectional Data Transfer

— Some `ioctl` calls require both reading and writing data.
— The kernel may modify the structure and return it to user space.

Here is our final result of ioctl communication

5 TP 4 Interruptions

5.1 Introduction

This report details the steps of TP4 on interrupt handling with the ADXL345 accelerometer. We will modify the existing driver to leverage the FIFO mode and interrupts.

5.2 Objectives

The objective is to add interrupt handling to retrieve accelerometer data via a FIFO buffer and improve driver behavior.

5.3 General Notes

5.3.1 Stream Mode

In Stream mode, acquired samples are stored in the FIFO. If the FIFO is full, the oldest sample is removed before storing a new one. When full, it holds the most recent 32 samples.

If we read DATA_X0 and then DATA_X1 in two separate I2C transactions, the values retrieved correspond to different samples.

5.3.2 Watermark Interrupt

The Watermark interrupt is triggered when the FIFO contains at least n elements (where n is the value in the Samples field of the FIFO_CTL register, ranging from 0-31). When the number of elements drops below n, the interrupt is reset.

The simulated interrupt line is connected to input number 50 of the primary interrupt controller. The kernel identifies the correct interrupt line and calculates its number, which is accessible via the *irq* field in the *struct i2c_client*.

```
Welcome to Buildroot
buildroot login: root
# random: fast init done
mount -t 9p -o trans=virtio mnt /mnt -oversion=9p2000.L,msize=10240
# insmod /mnt/adxl345.ko
adxl345: loading out-of-tree module taints kernel.
SEND bytes : 1
FIFOMOD : 0x94
STATUS : 0x00
INTERRUPT ENABLE : 0x02
INTERRUPT PIN MAP  : 0xfd
INTERRUPT SOURCE  : 0x00
#
```

Figure 25 – interrupt configuration

5.3.3 Interrupt Handler Design

An interrupt handler is set up to react to the device's interrupt. The device name appears in */proc/interrupts*.

The interrupt handler should not perform excessive and time consuming tasks that affect other functionalities . If necessary, it should be divided into two parts :

— **Top half** : Handles time-sensitive tasks.

— **Bottom half** : Performs less urgent processing triggered by the top half.

The top half executes quickly, while the bottom half handles processing that does not need to be completed immediately.

5.4 Setup

We will work with the Linux kernel and an I2C driver for the ADXL345. The sensor's FIFO will be enabled in **Stream** mode, and the **Watermark** interrupt will be configured.

FIGURE 26 – device registration

5.5 Function Analysis and Implementation

5.5.1 Interrupt Handling

We have defined an interrupt handler to process events generated by the accelerometer.

```c
static irqreturn_t adxl345_handler(int irq, void *dev_id) {
    return IRQ_WAKE_THREAD;
}
```

Listing 1 – Interrupt Handler

Explanation : This function marks the interrupt as waking up a kernel thread. This function implements the **TOP HALF**

5.5.2 adxl345 threaded irq

The main aim of the threaded IRQ is to reduce the time spent with interrupts being disabled and that will increase the chances of handling other interrupts.

```c


static irqreturn_t adxl345_threaded_irq(int irq, void *dev_id) {
    adxl345_device *dev = dev_id;
    adxl345_sample sample;

    if (adxl345_read_sample(dev, &sample) == 0) {
        kfifo_put(&dev->fifo, sample);
        wake_up_interruptible(&adxl345_wq);
    }

    return IRQ_HANDLED;
}
```

Listing 2 – THREADED IRQ Interrupt

Explanation :

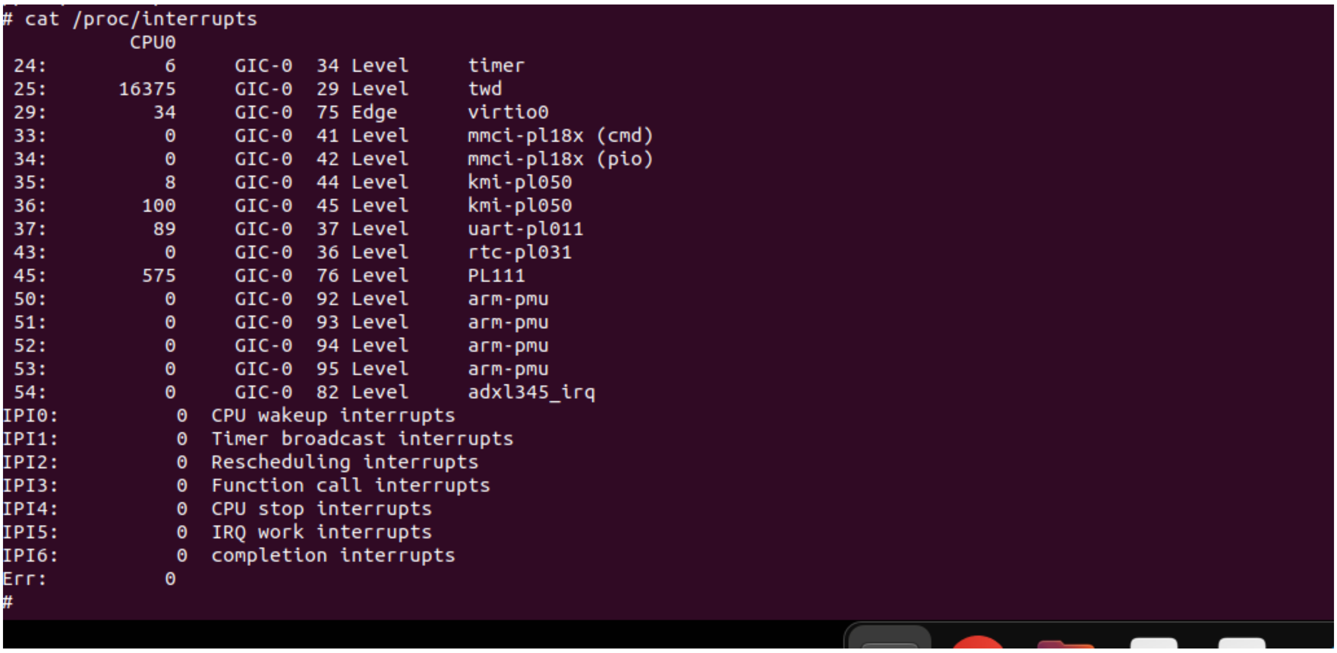
The function :

- Reads an accelerometer sample from the ADXL345 sensor.
- If successful, stores it in a FIFO buffer.
- Notifies any process waiting for new data.
- Returns IRQ HANDLED to indicate that the interrupt was processed.

5.5.3 Problem encountered when registering the interruption

an errot is returned when trying to launch the read function telling that the interruption was nor correctly registered :

solution screenshot :



```
# cat /proc/interrupts
           CPU0
 24:          6     GIC-0  34 Level     timer
 25:      16375     GIC-0  29 Level     twd
 29:         34     GIC-0  75 Edge      virtio0
 33:          0     GIC-0  41 Level     mmci-pl18x (cmd)
 34:          0     GIC-0  42 Level     mmci-pl18x (pio)
 35:          8     GIC-0  44 Level     kmi-pl050
 36:        100     GIC-0  45 Level     kmi-pl050
 37:         89     GIC-0  37 Level     uart-pl011
 43:          0     GIC-0  36 Level     rtc-pl031
 45:        575     GIC-0  76 Level     PL111
 50:          0     GIC-0  92 Level     arm-pmu
 51:          0     GIC-0  93 Level     arm-pmu
 52:          0     GIC-0  94 Level     arm-pmu
 53:          0     GIC-0  95 Level     arm-pmu
 54:          0     GIC-0  82 Level     adxl345_irq
IPI0:          0  CPU wakeup interrupts
IPI1:          0  Timer broadcast interrupts
IPI2:          0  Rescheduling interrupts
IPI3:          0  Function call interrupts
IPI4:          0  CPU stop interrupts
IPI5:          0  IRQ work interrupts
IPI6:          0  completion interrupts
Err:          0
#
```

FIGURE 27 – interrupt registration seccessfull

5.5.4 Reading Accelerometer Data

We have modified the read function to retrieve data via the software FIFO.

```c
static ssize_t adxl345_read(struct file *file, char __user *user_buffer, size_t count, loff_t *offset) {
    struct adxl345_device *dev = container_of(file->private_data, adxl345_device, misc);
    adxl345_sample sample;

    if (kfifo_is_empty(&dev->fifo)) {
        if (file->f_flags & O_NONBLOCK) return -EAGAIN;
        wait_event_interruptible(adxl345_wq, !kfifo_is_empty(&dev->fifo));
    }

    if (!kfifo_get(&dev->fifo, &sample)) return -EIO;
    if (copy_to_user(user_buffer, &sample, sizeof(sample))) return -EFAULT;

    return sizeof(sample);
}
```

Listing 3 – Reading FIFO Data

Explanation : This function reads a sample from the FIFO and copies it to user space.

5.5.5 FIFO and Interrupt Configuration

```
    //mode Stream
    sendbuff[0] = 0x38;
    sendbuff[1] = 0b10010100 ;    //stream mode   + INT1 bit 5   + 20
        samples
int sentDATA = i2c_master_send(client, sendbuff, 2);
```

Listing 4 – FIFO mode Configuration

```c

    char INTMAP[1];
    sendbuff[0] = 0x2F;
    sendbuff[1] = 0b11111101;   //0 means that is sent to INT1
    sentADD = i2c_master_send(client,sendbuff, 2);


    char INTBUFFER[1];
    sendbuff[0] = 0x2E;
    sendbuff[1] = 0x02;
    sentADD = i2c_master_send(client,sendbuff, 2);



    char RATE[2] ={BW_RATE, 0x0A};
    char FORMAT[2] ={DATA_FORMAT,0x08};  //FULL RES
    char POWER[2]   ={POWER_CTL_ON, 0x08}; // MEASURE MODE
```

Listing 5 – Intrrupt oand other register configuration

Explanation : This sequence configures the FIFO in **Stream** mode and enables the Watermark interrupt.

5.6 Results and Testing

Tests confirmed the proper functioning of interrupt handling. When the FIFO threshold is reached, the interrupt is triggered, and data is retrieved without an explicit request from the user application.

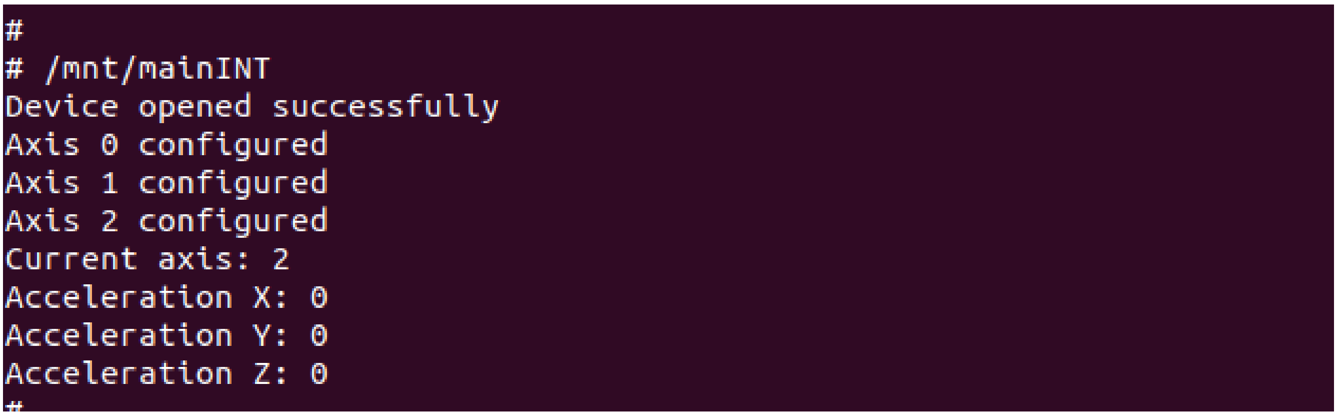

```
#
# /mnt/mainINT
Device opened successfully
Axis 0 configured
Axis 1 configured
Axis 2 configured
Current axis: 2
Acceleration X: 0
Acceleration Y: 0
Acceleration Z: 0
```

FIGURE 28 – data read from user space

5.7 Conclusion

This TP allowed us to explore interrupt handling and FIFO usage in a Linux driver for the ADXL345. These concepts are essential for developing efficient and responsive drivers.