I²C on a Linux based embedded system

Design of a bus driver and a client driver for the Nomadik NHK8815 platform

Ghiringhelli Fabrizio
Matr. 753368, (fabrizio.ghiringhelli@mail.polimi.it)

Report for the master course of Embedded Systems
Reviser: PhD. Patrick Bellasi (bellasi@elet.polimi.it)

Received: October, 5th 2012

Abstract

This article describes design and development of I²C device drivers for the Nomadik NHK8815 evaluation board running GNU/Linux. Specifically, attention is given to the implementation of drivers for an I²C bus controller of the ST Microelectronics STn8815 System On Chip (SOC), and for an I²C bus client, namely the LIS3LV02DL three-axis accelerometer. Both devices equip the NHK8815 board. The intended outcome of this work is to give a contribution in understanding how to access I²C devices on any Linux based systems, not only embedded systems. This is the reason why I focus on both a bus driver and a client driver. Besides, some testing techniques along with some solutions to interface the drivers with the user space are presented in some detail.

1 Introduction

Today Linux is the operating system choice for many computer systems which include, not only desktop and server supercomputers, but also a wide range of special-purpose electronic devices known as embedded systems. An embedded system is specifically designed to perform a set of designated activities, and it generally uses custom, heterogeneous processors. This makes Linux a flexible operating system capable of running on a variety of architectures, such as ARM, PowerPC, MIPS, SPARC, x86, and many others.

However, this flexibility doesn't come for free. While it's true that the Linux highly modular architecture facilitates the porting phase, still a lot of efforts are required to build new kernel components to fully support the target platform.

A big part of these efforts are in developing the low-level interfaces commonly referred to as *device drivers*. A device driver (driver for short) is a piece of software designed to direct control a specific hardware resource using an hardware-independent well defined interface.

This paper details the design of two I²C drivers for the Nomadik NHK8815 platform: a client driver for an on-board inertial sensor, presented in section 2, and a bus driver for the I²C controller of the SOC (section 3). Section 4 is dedicated to testing issues.

The rest of this section provides an overview of the I²C protocol (1.1), a brief description of the NHK8815 evaluation board (1.2), and detail information regarding the project environment (1.3).

1.1 I²C protocol overview

The Inter-Integrated Circuit, or I²C , is a synchronous master-slave messaging protocol designed to connect a pool of devices by means of a two-wire bus. It is a simple and low-bandwidth protocol which allows for short-distance on board communications, while being extremely modest in its hardware resource requirements. The original standard specified a standard clock rate of 100KHz. Later updates to the standard introduced a fast speed of 400KHz and a high speed of 1.7 or 3.4 MHz.

The I²C bus consists of two bi-directional lines, one line for data (SDA) and one for clock (SCL), by means of which a single master device can send informations serially to one ore more slave devices (**Figure 1**). To prevent any conflict every device hooked up to the bus has its own unique address. The standard I²C specifies two different addressing schema, 7 and 10 bits, allowing at most 128 and 1024 devices connected at the same time.

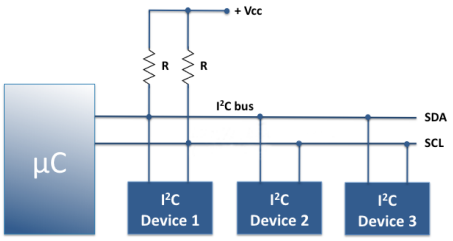

Figure 1: Sample I²C implementation (adapted from *embedded-lab.com*).

Each I^{2}C transaction is always initiated by the master which is in charge of the bus for the entire duration of the transaction, meaning that it controls the clock and generates the START and STOP sequences. The start and stop sequences mark the beginning and the end of a transaction and are the only places where the SDA line is allowed to change while the SCL is high.

All data are transfered one byte at a time. In 7-bit addressing mode, the slave address occupies the seven most significant bits of the first byte, with the least significant bit serving as a read/write flag to indicate whether data will be written to the slave ('0') or data will be read from the slave ('1'). For every byte received, the slave device sends back an acknowledge bit. **Figure 2** shows an example of a typical I^{2}C transaction.

The I^{2}C protocol supports multiple masters. In a multi-master environment two or more masters may simultaneously attempt to initiate a data transfer. In such a scenario, each master must be able to detect a collision and to follow the arbitration logic that leeds to the election of a winner master. The winner master can then safely begin its transaction. The Nomadik NHK8815 platform, like most system designs, operates in a single-master environment.

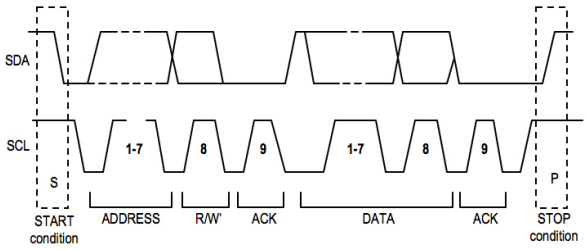

Figure 2: Sample I^{2}C transaction (adapted from *www.ermicro/blog*).

The Linux I^{2}C subsystem

The Linux kernel I^{2}C framework consists of a core layer where resides all the routines and data structures available to bus drivers and client drivers (**Figure 3**). The core also provides a level of indirection that allows the underlying drivers to change from one system to another without affecting I^{2}C subsystem that relies on them.

This philosophy of a core layer and its attendant benefits is an example of how Linux helps portability. For instance, enabling I^{2}C on a new platform (which is precisely the objective of this project) requires only to design the hardware-dependent components, namely the bus driver and the client drivers, whereas the core layer needs not to be changed.

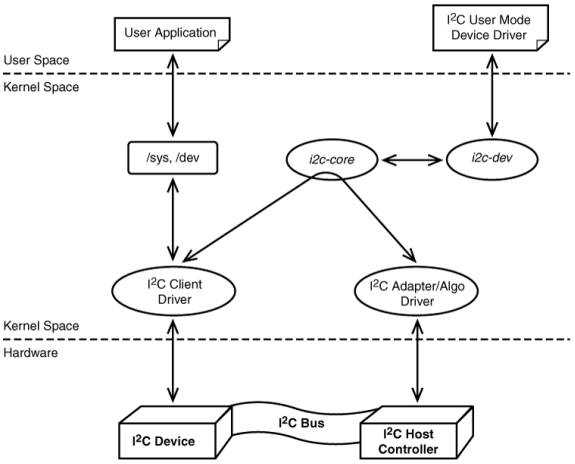

Figure 3: The Linux I^{2}C subsystem (reprinted from [1], p. 236).

1.2 The Nomadik NHK8815 platform

The NHK8815 is a full-featured evaluation board for the ST Microelectronics Nomadik STn8815 (**Figure 4**). It is a fan-less embedded computer equipped with a wide range of peripheral devices including USB, UART, LAN, WLAN, Bluetooth, FM radio, SIM card reader, SD/MMC card reader, color LCD with touch screen controller, keypad, video encoder, audio codec, FM radio, three-axis accelerometer, etc..

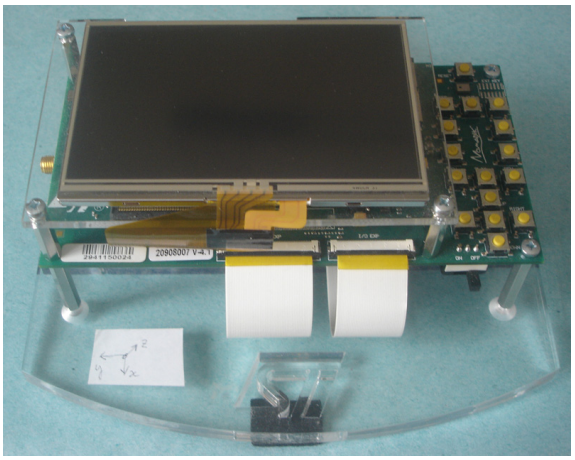

Figure 4: The Nomadik NHK8815 evaluation board.

The core of the board is the Nomadik STn8815 multimedia application processor. The STn8815 is a *System On Chip* that combines an ARM9 core up to 332MHz with level-two cache to audio, video, imaging and graphics accelerators. Among other peripherals, the STn8815 integrates two I^{2}C high speed controllers that support the physical and data-link layer of the I^{2}C protocol. Below is a list of the I^{2}C controllers' main features:

- Slave transmitter/receiver and master transmitter/receiver modes.

- Standard (100kHz), fast (400kHz) and high-speed (3,4MHz) baud rates.
- 7-bit or 10-bit addressing.
- Compliance with I^{2}C standards.

In a traditional I^{2}C bus topology such as the one in figure 1, the STn8815 I^{2}C embedded controller plays the role of the master. In order for the controller to access the I^{2}C bus, a corresponding driver must be implemented and registered with the I^{2}C subsystem. This step is covered in section 3, "I2C bus driver."
Because on the NHK8815 evaluation board the majority of the on-board peripheral devices is connected to the STn8815 I^{2}C bus 0, in this project I focus the attention on this bus rather than on bus 1. **Table 1** reports the list of devices on the I^{2}C bus 0, with their 7-bit address.
Just like in the case of the bus driver, in order for each slave device to function a corresponding driver must be registred to the I^{2}C subsystem. My choice for this project was the LIS3LV02DL, a MEMS inertial sensor mapped at the address 1D on the Nomadik board. The design of the LIS3LV02DL driver is detailed in section 2, "I2C client driver."

Device type	Description	7-bit address
STw5095	Stereo audio codec	1A
LIS3LV02DL	Inertial sensor	1D
STw8009	Video digital encoder	21
TDA8023TT	Smart Card interface	22,23
STw4811	Power management	2D
STMPE2401	Port expanders	43,44
TSC2003IPW	Touch screen controller	48
STw4102	Battery charger	70

Table 1: Device address map of NHK8815 I^{2}C bus 0.

1.3 Project setup

The first step to the project is to get the Linux kernel source and to configure it for the target board. Instead of using a vendor supplied kernel such as like STLinux, my personal strategy for this project was to work with the official kernel from *www.kernel.org*. This decision was motivated by the need to keep the development process as neat as possible, and to avoid performing tasks that are not strictly concerned with the project.
It should be noted that the official kernel offers only limited support for the Nomadik NHK8815 platform. This includes basic definitions and register addresses, interrupt management, resource definitions and timers. A generic I^{2}C bus driver that uses general purpose I/O lines for the interface (i.e. *i2c-gpio* and *i2c-algo-bit*) is also included in the kernel, but no driver for the NHK8815 I^{2}C on-board devices is provided whatsoever.
Here is how I proceeded to set up the project environment. First, I cloned the git repository at kernel.org on a local machine folder located outside the project workspace. Then, from the tag *3.3* I created a new branch called *i2cdevel*, where to commit the kernel updates. My rule was to generate one patch per commit. All patches are numbered in sequence and stored in a folder inside the project workspace called *linux-3.3.0-patches*.
One benefit of this approach is that the project workspace, which is a git repository as required, also holds all the data needed to obtain the modified kernel from the official one. **Table 3** shows the project workspace layout. In addition, to ease the project management I defined a set of environmental variables. as listed in **Table 2**.
Finally, a note about the root filesystem. It was built using BusyBox version 1.18.4 ([2]). More precisely, I used a tool called *bbfs 1.3*, developed by *Alessandro Rubini* and released under GNU GPL license (see [3] for details).

Variable	Decription
`PRJROOT`	The project root directory
`KERNELDIR`	The root directory of the kernel source tree
`PATCHESDIR`	The *linux-3.3.0-patches* directory
`ROOTFS`	The directory where is stored the target root filesystem
`MODULESDIR`	The directory for temporary holding of the loadable driver modules

Table 2: Project environmental variables.

2 I2C client driver

This section describes the design of the driver for the LIS3LV02DL inertial sensor which equips the NHK8815 evaluation board. The LIS3LV02DL is a three axes linear accelerometer that provides the measurement acceleration signals to the external world through an I^{2}C interface. The LIS3LV02DL driver was written before the bus driver. Its debug was possible by using the gpio-based I^{2}C bus driver available in the original kernel. Once completed and tested, the LIS3LV02DL driver was then used for debugging the STN8815 I^{2}C bus driver.

2.1 Initializing and probing

During initialization the driver registers itself with the I^{2}C core. This is achieved by populating a `struct i2c_driver` and passing it as argument to the function `i2c_add_driver()`, as shown in **Listing 1**.
The structure `i2c_driver` holds pointers to the probe and remove functions that are executed respectively on device probing and when the device is removed (if ever). The `id_table` member of the structure `i2c_driver` informs the I^{2}C framework about which slave devices are supported

Directory	Content
`linux-3.3.0-patches`	Patches to the Linux kernel and kernel configuration file *.config*
`drivers`	I^{2}C bus driver and client driver (one subfolder per each driver)
`report`	The LaTeXsources of this paper
`script`	Shell scripts (copied to the root directory of the target filesystem)
`tools`	User-space programs to evaluate the drivers (copied to the root directory of the target filesystem)
`sdcard`	U-Boot command file, kernel binary image and ramdisk with the root filesystem

Table 3: Project directory layout.

by the driver. In this case the only chip supported is named `lis3lv02d`. The names of the supported devices are important for *binding*, as explained next.

```
1  /* Device and driver names */
2  #define DEVICE_NAME             "lis3lv02d"
3
4  /* I2C client structure */
5  static struct i2c_device_id lis3lv02d_idtable[] = {
6    { DEVICE_NAME, 0 },
7    {}
8  };
9  MODULE_DEVICE_TABLE(i2c, lis3lv02d_idtable);
10
11 static struct i2c_driver lis3lv02d_driver = {
12   .driver   = {
13     .name = DRIVER_NAME
14   },
15   .probe    = lis3lv02d_probe,
16   .remove   = __devexit_p(lis3lv02d_remove),
17   .id_table = lis3lv02d_idtable,
18 };
19
20 /* Module init */
21 static int __init lis3lv02d_init(void)
22 {
23   return i2c_add_driver(&lis3lv02d_driver);
24 }
```

Listing 1: Registration of the LIS3LV02DL driver (from *lis3lv02d-nhk8815.c*).

The binding process consists of associating a device with a driver that can control it. In embedded systems where the number of the I^{2}C bus and the devices connected to it are known for a fact, it is possible to declare in advance the I^{2}C slaves which live on the bus. This is typically done in the board setup file (**Listing 2**).

The `nhk8815_platform_init` function is executed on board startup and, among other tasks, registers the I^{2}C slave devices by invoking the `i2c_register_board_info` function with arguments that specify the number of the bus (zero in this case) and the devices connected with it. This is done through an array of `struct i2c_board_info()`, each item of which specifies the device name and the device address, with the former that must match with the name registered by the driver in order for binding to succeed. In this case `struct i2c_board_info` holds only one item which corresponds to the LIS3LV02DL inertial sensor.

Seeing that this sensor's chip has an interrupt line tied to the cpu (gpio 82), the `irq` member is also specified with the respective IRQ number. By means of another member called `platform_data` it is possible to define custom data for the driver; if not specified like in this case, the driver uses its default settings, as explained in section 2.5.

```
1  /* I2C devices */
2  static struct i2c_board_info nhk8815_i2c_devices[] = {
3    {
4      I2C_BOARD_INFO("lis3lv02d", 0x1D),
5      .irq = NOMADIK_GPIO_TO_IRQ(82),
6      /* No platform data: use driver defaults */
7    },
8  };
9
10 static void __init nhk8815_platform_init(void)
11 {
12   ...
13   /* Register I2C devices on bus #0 (scl0, sda0) */
14   i2c_register_board_info(0, nhk8815_i2c_devices,
15     ARRAY_SIZE(nhk8815_i2c_devices));
16   ...
17 }
```

Listing 2: Registration of the I^{2}C devices (from *board-nhk8815.c*).

During boot the kernel looks for any I^{2}C driver that has registered a matching device name, that is "lis3lv02d". Upon finding such a driver, the kernel invokes its `probe()` function passing a pointer to the LIS3LV02DL device as a parameter. This process is called *probing*.

The probe function is responsible for the per-device initialization, that is initializing hardware, allocating resources, and registering the device with any appropriate subsystem. More in detail, the LIS3LV02DL probe function takes the following actions:

1. Allocate memory for `lis3lv02d_priv` private data structure.
2. Load the device settings.
3. Idendify the LIS3LV02DL chip.
4. Configure the device hardware.
5. Create the per-device sysfs nodes.
6. If the free-fall feature is enabled, request the interrupt and register the IRQ for the free-fall detection.
7. If the device polling feature is enabled, register the device with the input subsystem.

On successful completion of all the above steps, meaning a successful probing, the device is bound to the driver.

2.2 Sysfs interface

Sysfs is an in-memory virtual filesystem that provides a view of the kernel's structured device model. It offers a convenient yet simple way to implement functionality as sysfs attribute in the appropriate directory. An attribute provides a way to map kernel data to files in sysfs: a single attribute maps to a single file which can be readable, writable or both, depending on which function it exports.

A driver wishing to use sysfs needs to register the sysfs attributes and implement their respective functions. **Table 4** shows the list of sysfs nodes handled by the LIS3LV02DL driver. They are located in `/sys/devices/platform/lis3lv02d/`.

Attribute	Access	Function
`position`	r	Show the acceleration along the x,y,z axis
`enable`	r	Show the enable status of the device
	w	Enable/disable the device
`ff_enable`	r	Show the enable status of the free-fall feature
`poll_enable`	r	Show the enable status of the polling feature
	w	Enable/disable the device polling
`read`	r	Read the register at the current address
	w	Set the current address for reading
`write`	w	Write a register. The low byte holds the register value and the high byte holds the register address

Table 4: LIS3LV02DL sysfs attributes.

Below is a dump of a shell session executed on the target board, providing a short demonstration of how to play with the LIS3LV02DL sysfs interface.

```
/sys/devices/platform/lis3lv02d # cat enable
n
/sys/devices/platform/lis3lv02d # echo 1 > enable
/sys/devices/platform/lis3lv02d # cat position
56 -4 -1043
/sys/devices/platform/lis3lv02d # cat ff_enable
y
/sys/devices/platform/lis3lv02d # echo F > read
/sys/devices/platform/lis3lv02d # cat read
3A
/sys/devices/platform/lis3lv02d #
```

Figure 5: Example of usage of the LIS3LV02DL sysfs interface.

The `cat enable` command returned 'n', meaning that the device was disabled. Turning the device on is achieved by typing `echo 1 > enable`. The `cat position` command returned the values 56, -4, 1043 which correspond to to acceleration along the x-y-z axes, measured in mg. `cat ff_anable` showed that the free-fall feature is enabled. The last two commands perform a reading at the address 0F, getting as answer 3A. [1].

2.3 Device polling

Generally speaking *polling* is a technique in which one device periodically monitors multiple other devices or makes requests from those devices (e.g. check their state). In this project's context, the polling device is the SOC while the LIS3LV02DL inertial sensor is the device being polled.

To implement the polling feature the LIS3LV02DL driver registers itself with the kernel's *input subsystem*; this involves registering `open()`, `close()` and `poll()` callback functions, specifying the polling interval, setting up the type of events involved, etc. The input subsystem takes care of calling the `poll()` function at the specified rate, registering any event notification reported by the driver and dispatching it to the user-space via sysfs in `/dev/input/eventX` ('X' is a numeric identifier assigned when the driver registers with the input core). A demonstration of how the device polling works is given in the section 4.2.

Listing 3 shows the sequence of instructions required to registers with the input core. The driver executes them upon enabling polling. It allocates the input device, installs the callback function pointers (lines 8-10), programs the ABS events related to the x,y,z acceleration data (lines 14-17), and registers the input device (line 26). Lines 20 through 23 relates to the free-fall feature which is explained in the next section.

```
1  struct input_polled_dev *input_polled;
2  struct input_dev *input;
3
4  /* Allocate memory for the input device */
5  input_polled = input_allocate_polled_device();
6
7  /* Setup input parameters */
8  input_polled->open = lis3lv02d_open;
9  input_polled->close = lis3lv02d_close;
10 input_polled->poll = lis3lv02d_poll;
11 input = input_polled->input;
12
13 /* Setup ABS input events */
14 set_bit(EV_ABS, input->evbit);
15 set_bit(ABS_X, input->absbit);
16 set_bit(ABS_Y, input->absbit);
17 set_bit(ABS_Z, input->absbit);
18
19 /* Setup KEY event for free-fall (only if enabled) */
20 if (priv->ff_enabled) {
21   set_bit(EV_KEY, input->evbit);
22   set_bit(KEY_FREE_FALL, input->keybit);
23 }
24
25 /* Register input polled device */
26 input_register_polled_device(input_polled);
```

Listing 3: Device registration with the input subsystem (from `lis3lv02d_poll_enable()` in *lis3lv02d-nhk8815.c*).

[1] 0F is the address of the `WHO_AM_I` register which holds the device identification number (3A for the LIS3LV02DL chip)

To disable polling the driver unregisters the input device and frees its previously allocated memory, as shown below.

```
1 /* Unregister input device */
2 input_unregister_polled_device(priv->input_polled);
3 input_free_polled_device(priv->input_polled);
```

Listing 4: Device unregistration (from `lis3lv02d_poll_disable()` in *lis3lv02d-nhk8815.c*).

Upon loading the LIS3LV02DL driver, the polling function may be either enabled or not enabled, depending on whether the default or custom settings were used (see section 2.5 for details). Once the driver is up and running, the polling can be enabled/disabled via the sysfs `poll_enable` file.

2.4 Free-fall detection

The LIS3LV02DL may be configured to generate an inertial wake-up/free-fall interrupt signal when a programmable acceleration threshold is crossed at least in one of the three axes.
The LIS3LV02DL driver installs a threaded ISR to handle the free-fall detection. Upon receiving an interrupt request the driver reads the acceleration values from the sensor and notifies an appropriate event to the input core for being dispatched to the user-space. Indeed the driver relies on the input subsystem to "inform" the user-space about a free-fall event. This implies that the device polling be enabled.
Usage of the free-fall feature can be enabled only statically (i.e at compile time) either by adjusting the driver's defaults or by passing custom settings from the board initialization code (through `platfom_data`). From user-space the free-fall enable status can be seen via the sysfs `ff_enable` file (see table 4).
If the free-fall is enabled, the driver's `probe()` registers the IRQ number (as provided by the device) and installs its respective threaded ISR (lines 5-8 of listing 5).

```
1  /* Get IRQ (only if free-fall is enabled) */
2  if (pdata->free_fall_cfg & LIS3_FF_ALL) {
3    ...
4    /* Register IRQ */
5    err = request_threaded_irq(client->irq, NULL,
6      lis3lv02d_isr_thread,
7      IRQF_TRIGGER_RISING | IRQF_ONESHOT,
8      DEVICE_NAME, priv);
9    ...
10   priv->ff_enabled = true;
11 }
```

Listing 5: Registration of the free-fall IRQ (from `probe()` in *lis3lv02d-nhk8815.c*).

The free-fall is notified in form of a KEY event. As for all the input events this requires two steps: registering with the input subsystem and reporting upon detection of free-fall. The former being executed by `lis3lv02d_poll_enable()` (lines 20-23 of listing 3), and the latter being handled by the free-fall ISR as shown below.

```
1 /* Report free-fall (KEY) event */
2 input_report_key(input, KEY_FREE_FALL, true);
3 input_report_key(input, KEY_FREE_FALL, false);
4 input_sync(input);
```

Listing 6: Notification of a free-fall KEY event (from `lis3lv02d_isr_thread()` in *lis3lv02d-nhk8815.c*).

2.5 LIS3LV02DL settings

The customization of the LIS3LV02DL settings is done based on *platform data*. These are data attached to a platform device, that are completely specific to a given device. It allows the board initialization file to transmit detailed and custom information about the device to the driver. In this project's context, the platform data is represented in form of a `lis3lv02d_nhk8815_platform_data` structure (**Listing 7**) which allows to set a number of LIS3LV02DL parameters including the acceleration full-range (2g or 6g), free-fall settings, polling interval, etc..

```
1 /* Platform data */
2 struct lis3lv02d_nhk8815_platform_data {
3   unsigned char device_cfg;
4   unsigned char free_fall_cfg;
5   unsigned int free_fall_threshold;
6   unsigned char free_fall_duration;
7   unsigned int poll_interval;
8 }
```

Listing 7: LIS3LV02DL driver's platform data (from *lis3lv02d-nhk8815.h*).

The board initialization code may omit to transmit custom data, as it relies on the driver using its default settings. This is precisely the solution adopted for this project (line 6 of listing 2). One may choose the other way round and set up the `platform_data` member of `i2c_board_info` to point to a `lis3lv02d_nhk8815_platform_data` structure with custom settings.
Upon probing the driver checks whether `platform_data` holds a valid pointer and, if this is the case, loads the custom settings. Otherwise it uses the default values. Below is the code responsible for this action.

```
1 /* Save pointer to platform data */
2 pdata = client->dev.platform_data;
3 if (!pdata) {
4   dev_info(&client->dev,
5   "no platform data, using defaults\n");
6   pdata = &lis3lv02d_default_init;
7 }
```

Listing 8: Loading of LIS3LV02DL platform data (`lis3lv02d_probe()` in *lis3lv02d-nhk8815.c*).

Listing 9 shows the default settings for the LIS3LV02DL device as they are defined in *lis3lv02d-nhk8815.c*.
The driver was built as a module and manually loaded into the kernel using the `insmod` program. Upon successful load the driver delivers an information message about its

settings to the kernel as shown in **Figure 6**. The last command, `lsmod`, outputs the list of modules currently loaded into the kernel (only `lis3lv02d-nhk8815` in this example).

```
/* LIS3LV02DL default configuration */
static const struct lis3lv02d_nhk8815_platform_data
        lis3lv02d_default_init = {

  .device_cfg = LIS3_ODR_40HZ | LIS3_FS_2G,
  .poll_interval = 500,
  .free_fall_cfg = LIS3_FF_XL | LIS3_FF_YL | LIS3_FF_ZL,
  .free_fall_threshold = 600,
  .free_fall_duration = 5,      /* 1/ODR [s] */
};
```

Listing 9: LIS3LV02DL default setup (from *lis3lv02d-nhk8815.c*).

3 I2C bus driver

As mentioned in section 1.2 the STn8815 processor integrates two I^{2}C controllers that can be programmed to work in standard, fast or high-speed mode. Each controller is designed to operate in a multi-master environment, either as a master or as a slave. However, because the STn8815 is the only master on board, and seeing that at this time Linux only operates I^{2}C in master mode, the I^{2}C bus driver presented here supports master mode only.

3.1 Initializing and probing

Initializing and probing the STn8815 I^{2}C bus driver is performed in a similar way as for LIS3LV02DL client driver (section 2.1), with the major difference being that the former uses a *platform bus*.
The platform bus requires that any I^{2}C *adapter* (or equivalently *controller*) [2], which is controlled by the bus driver, be registered using a `platform_device` structure. This structure represents the bus adapter and provides information such as the device name, the device resources and the adapter number, to the bus driver.
Usually the registration of the I^{2}C adapters with the platform bus is performed by the board initialization file, as the information needed are highly board specific. And this case is no exception.
Listing 10 shows the part relevant to this matter, taken from *arch/arm/mach-nomadik/i2c-8815nhk.c*. Because of the STn8815 has two I^{2}C adapters, two platform devices need to be defined: one for bus 0 and one for bus 1. In this example the former uses the platform data mechanism to customize the driver settings, while to the latter relies on the driver defaults (see section 3.4 for details about the driver settings).
The `resource` and `num_resources` fields allow to define the device resources (for brevity not shown here), including the memory area (base address and size) and the interrupt number.

```
/* first bus: i2c0 */
static struct platform_device nhk8815_i2c_dev0 = {
  .name         = "stn8815_i2c",
  .id           = 0,
  .resource     = &nhk8815_i2c_resources[0],
  .num_resources = 2,
  .dev          = {
    .platform_data = &nhk8815_i2c_dev0_data,
  },
};

/* second bus: i2c1 */
static struct platform_device nhk8815_i2c_dev1 = {
  .name         = "stn8815_i2c",
  .id           = 1,
  .resource     = &nhk8815_i2c_resources[2],
  .num_resources        = 2,
  /* No platform data: use driver defaults */
};

static int __init nhk8815_i2c_init(void)
{
  ...
  platform_device_register(&nhk8815_i2c_dev0);
  ...
  platform_device_register(&nhk8815_i2c_dev1);
  ...
}
arch_initcall(nhk8815_i2c_init);
```

Listing 10: Registration of the I^{2}C adapter with the platform bus (from *i2c-8815nhk.c*).

On the driver's side, the registration with the platform bus is achieved by populating a `struct platform_driver` and passing it to the macro `module_platform_driver()` as argument (**Listing 11**). The platform bus simply compares the `driver.name` member against the name of each device, as defined in the `platform_device` data structure (Listing 10); if they are the same the device matches the driver.

```
#define DRIVER_NAME     "stn8815_i2c"

static const struct dev_pm_ops stn8815_i2c_pm_ops = {
  SET_RUNTIME_PM_OPS(stn8815_i2c_runtime_suspend,
                     stn8815_i2c_runtime_resume,
                     NULL)
};

static struct platform_driver stn8815_i2c_driver = {
  .probe  = stn8815_i2c_probe,
  .remove = __devexit_p(stn8815_i2c_remove),
  .driver = {
    .name  = DRIVER_NAME,
    .owner = THIS_MODULE,
    .pm = &stn8815_i2c_pm_ops,
  },
};
module_platform_driver(stn8815_i2c_driver);
```

Listing 11: Registration of the I^{2}C bus driver with the platform bus (from *i2c-stn8815.c*).

As usual, binding a device to a driver involves calling the driver's `probe()` function passing a pointer to the device as a parameter. The sequence of operations performed on probing are the following:

1. Get the device resource definitions.
2. Allocate the appropriate memory and remap it to a virtual address for being accessed by the kernel.

[2] The two terms are interchangeable in meaning and refer to the peripheral device that drives the bus (i.e. the master device)

```
/lib/modules/3.3.0+/extra #
/lib/modules/3.3.0+/extra # insmod lis3lv02d-nhk8815.ko
lis3lv02d-nhk8815 0-001d: no platform data, using defaults
lis3lv02d-nhk8815 0-001d: rev 1.0, 40Hz data rate, +/-2g full-scale, free-fall enabled
/lib/modules/3.3.0+/extra # lsmod
lis3lv02d_nhk8815 7001 0 - Live 0xbf012000 (O)
/lib/modules/3.3.0+/extra #
```

Figure 6: Message showed upon loading LIS3LV02DL module with default settings.

3. Load the device settings.
4. Configure the device hardware.
5. Register with the power management system.
6. Create the per-device sysfs nodes.
7. Request the interrupt and register the IRQ.
8. Set up the `struct i2c_adapter` and register the adapter with the I²C core.

Once all the above steps successfully complete the driver is bounded to the devices representing the two STn8815 I²C controllers.

3.2 Data transfer

In the Linux I²C subsystem a bus driver consists of an *adapter driver* and an *algorithm driver*. The motivation behind this further division is to improve the software reuse and to allow portability. In fact, an algorithm driver is intended to contain general code that can be used for a whole class of I²C adapters, while each specific adapter driver either depends on one algorithm driver, or includes its own implementation.
However, while having a generic algorithm that works for multiple adapters is suitable for many cases, in embedded systems, where each I²C bus adapter has its own way of interfacing with the processor and the bus, it is usual to develop the adapter driver together with its corresponding algorithm driver. This is also the case of the STn8815 I²C bus driver.
The bus driver registers with the I²C subsystem by using a `struct i2c_adapter` that is instantiated and initialized by the driver's `probe()` function, as shown in **Listing 12**. The `i2c_adapter` structure's `algo` member is set up to point to a `struct i2c_algorithm` which in turn holds two pointers:

- `master_xfer` points to the function that implements the actual I²C transmit and receive algorithm.
- `functionality` points to a function that returns the features supported by the I²C adapter.

To communicate with a client the I²C subsystem offers two class of functions: one for I²C plain communication which includes `i2c_master_send()`, `i2c_master_recv()` and `i2c_transfer()`, and a second one that uses SMBus commands [3]. However, whichever method is used, the data transfer relies on the bus driver's function pointed to by `master_xfer`, as the I²C core ultimately calls this function for the actual transfer to take place.

```
1  /* I2C algorithm structure */
2  static struct i2c_algorithm stn8815_i2c_algo = {
3    .master_xfer   = stn8815_i2c_xfer,
4    .functionality = stn8815_i2c_func,
5  };
6
7  /* Probe function */
8  static int __devinit stn8815_i2c_probe
9          (struct platform_device *pdev)
10 {
11   ...
12   adap = kzalloc(sizeof(struct i2c_adapter),
13          GFP_KERNEL);
14   ...
15   adap->algo = &stn8815_i2c_algo;
16   ...
17   err = i2c_add_numbered_adapter(adap);
18   ...
19 }
```

Listing 12: Registration of the I²C adapter (from *i2c-stn8815.c*).

As shown in listing 12, `stn8815_i2c_xfer()` is the transfer function installed by the STn8815 I²C bus driver. This function receives an array of messages as argument and processes them in sequence by calling `stn8815_i2c_xfer_rd()` or `stn8815_i2c_xfer_wr()` depending on whether the message being processed is marked for read or write. Once all messages have been sent `stn8815_i2c_xfer()` succesfully returns, otherwise, upon detecting a communication error, aborts the transmission and returns an appropriate error code.
The I²C bus driver uses the STn8815 I²C controller's interrupt capability to handle the data transfer. More precisely it uses interrupts to trigger the completion of a transaction and to detect a bus error. Handling these interrupts is in charge of `stn8815_i2c_isr()` interrupt service routine (ISR). Its role is to identify the interrupt source, complete the transaction by reporting the result to the *task-level* [4] function that initiated it (either `stn8815_i2c_xfer_wr()` or `stn8815_i2c_xfer_rd()`), and to clear the interrupt condition from the I²C controller.
A *completion variable* is used to synchronize between the task-level function and the ISR. The former initiates the transmission and waits on the completion variable while the I²C controller performs the transaction; upon completion, the controller issues an interrupt request which is processed by `stn8815_i2c_isr()`; once it

[3] SMBus is a subset of I²C
[4] *Task-level* refers to code not running in interrupt context (as opposed to *interrupt-level*)

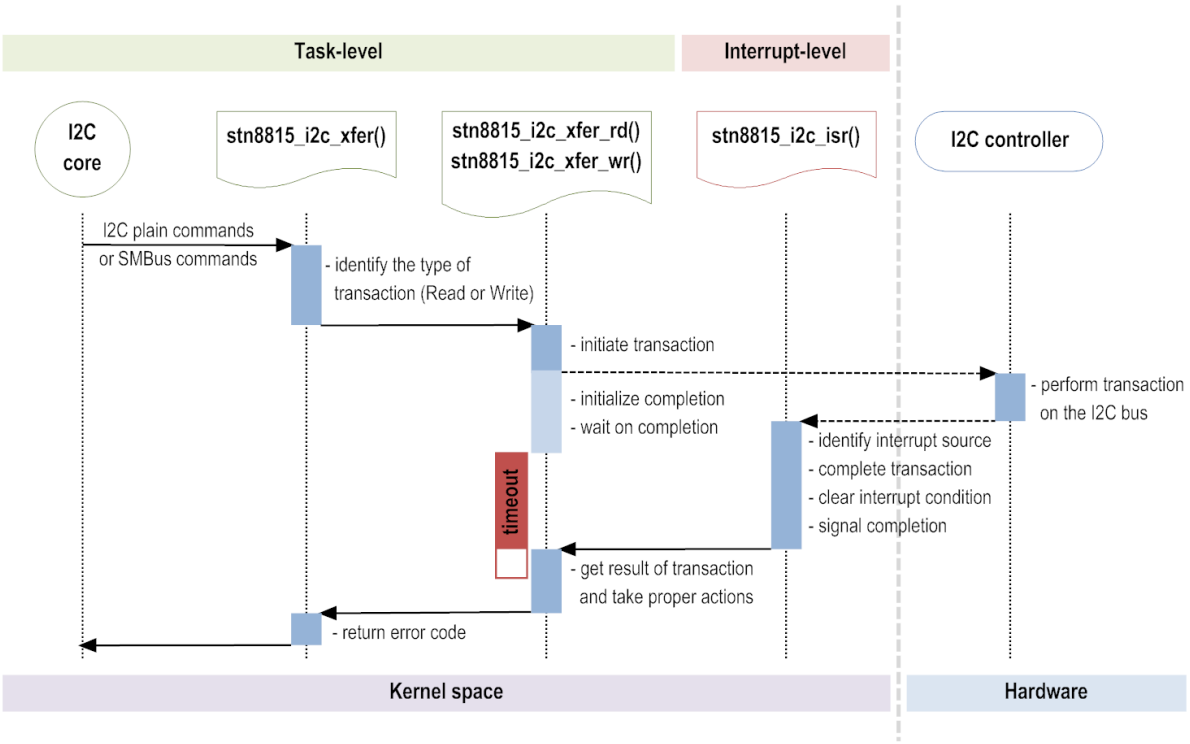

Figure 7: Sequence diagram of an I^{2}C transaction.

has accomplished its job, upon returning from interrupt, `stn8815_i2c_isr()` uses the completion variable to wake up the task-level function. Additionally, a timeout is specified to limit the time spent by the task-level function on waiting on the completion variable. The whole process is depicted in the sequence diagram of **Figure 7**.

```
1  /* I2C master read */
2  static int stn8815_i2c_xfer_rd(
3      struct i2c_adapter *adap,
4      struct i2c_msg *pmsg, bool stop)
5  {
6    ...
7    /* Initialize completion */
8    init_completion(&dev->msg_complete);
9    ...
10
11   /* Wait for completion */
12   i = wait_for_completion_timeout(&dev->msg_complete,
13                                   I2C_TIMEOUT);
14   if (i < 0)
15     return i;
16   if (i == 0) {
17     dev_err(dev->dev, "Controller timeout\n");
18     return -ETIMEDOUT;
19   }
20   ...
21 }
22
23 /* ISR */
24 static irqreturn_t stn8815_i2c_isr(int irq,
25         void *dev_id)
26 {
27   ...
28   /* Complete */
29   complete(&dev->msg_complete);
30   return IRQ_HANDLED;
31 }
```

Listing 13: Task-level and ISR synchronization using a completion variable (from *i2c-stn8815.c*).

Listing 13 shows how the synchronization mechanism is implemented. The completion variable is dynamically created and initialized via `init_completion()` (line 8). Afterwards, a call to `wait_for_completion_timeout()` (line 12) suspends the task until either the ISR signals the completion by calling `complete()` (line 29) or the timeout expires. In either case the return value is tested for error conditions.

3.3 Power management

The I^{2}C bus driver relies on the power management infrastructure of the Linux kernel to achieve run-time power savings. To use this feature the kernel needs to be configured with "Runtime power management" (`CONFIG_PM_RUNTIME`) option enabled.

The functions that implements the driver's power saving policy are registered upon registering the driver with the platform bus (refer to listing 11 on page 7). This is done by initializing the `driver.pm` member of the `platform_driver` structure with a pointer to `struct dev_pm_ops` which in turn holds the address of the driver's power management functions.

When the power management subsystem in Linux determines that the I^{2}C adapter is to be suspended, it calls `stn8815_i2c_runtime_suspend()`. This function simply disables the I^{2}C controller. When the adapter is to be resumed, `stn8815_i2c_runtime_resume()` is called and consequently the I^{2}C controller is re-enabled.

The power management framework creates for the I^{2}C

```
~ #
~ # insmod lib/modules/3.3.0\+/kernel/drivers/i2c/busses/i2c-stn8815.ko
stn8815_i2c stn8815_i2c.0: bus 0, rev 1.0, fast mode (400 Kb/s), 1 clock-wide-spikes filter
stn8815_i2c stn8815_i2c.1: no platform data, using defaults
stn8815_i2c stn8815_i2c.1: bus 1, rev 1.0, standard mode (100 Kb/s), no filter
~ #
```

Figure 8: Message showed upon loading the STN8815 I^{2}C bus driver module with default settings.

adapter 0 a set of sysfs nodes that allow to query or modify its power status. They are all located in `/sys/devices/platform/stn8815_i2c.0/power/`.
For example, typing `cat runtime_status` returns the adapter's power status, while `echo 'off' > control` disables the adapter's runtime power management.

3.4 I^{2}C bus adapter settings

As for LIS3LV02DL, the I^{2}C bus driver receives custom settings from the board initialization code via the platform data mechanism. Looking back at listing 10, the adapter 0 uses a custom setup (line 8), while the adapter 1 relies on the driver's defaults (line 18).
The platform data of the I^{2}C adapter is defined in *i2c-stn8815.h* and reported in **Listing 14**. It is represented in form of a `i2c_stn8815_platform_data` structure and allows to specify the speed mode (standard, fast or high-speed), the bus filtering and the master code.

```
1  /* Platform data */
2  struct i2c_stn8815_platform_data {
3    unsigned char filter;
4    unsigned char speed;
5    unsigned int  master_code;
6  }
```

Listing 14: STN8815 I^{2}C bus driver's platform data (from *i2c-stn8815.h*).

The driver loads the platform data on probing and, if not provided, uses the default setup as defined by *nhk8815_i2c_default_init* (**Listing 15**). The information message sent to the kernel by the driver on successful load is shown in **Figure 8**.

```
1  /* STN8815-I2C platform data default */
2  static struct i2c_stn8815_platform_data
3          nhk8815_i2c_default_init = {
4    .filter       = I2C_STN8815_FILTER_NONE,
5    .speed        = I2C_STN8815_SPEED_STANDARD,
6    .master_code = 0,
7  };
```

Listing 15: STN8815 I^{2}C bus adapter default setup (from *i2c-stn8815.c*).

4 Testing

As I said in section 1.3 my choice for this project was to work with the official kernel, which yet lacks many drivers necessary to fully operate the Nomadik NHK8815 board. Clearly this choice influenced the way the design process was conducted. For example, the on-board LCD display and keypad were not used for interfacing to the system because of the absence of the respective drivers in the kernel. Instead, I connected to the target board through a serial line (UART) and interfaced with it using a terminal emulator. This was achieved by configuring appropriately the boot-loader installed on the target, namely U-Boot ([4]).
A number of tools and scripts were used to help testing the drivers throughout the design process. They all run in user-space and interface with the Linux kernel, each one using either the *sysfs* interface, the *evdev* interface or the *i2c-dev* interface. **Figure 9** shows a view of the whole driver system including the tools mentioned above. This section describes some simple techniques that allows to test the drivers using these tools. The tests are classified based on the type of interface they rely on.

4.1 Testing via the *sysfs* interface

The usage the LIS3LV02DL sysfs interface was previously described in section 2.2. This interface turns out to be useful also for debugging activity even during the early stages of design, when most of the driver functions are yet to be implemented. An example of this is represented by the sysfs nodes `read` and `write` which were designed solely to assist debugging (in fact they are available only if the driver is built for debugging, i.e the keyword `LIS3LV02D_DEBUG` is defined in the driver's source file).
`read` and `write` together are used in the shell script *lis3_read.sh* to implement a simple technique for testing the drivers. This script allows to read a user defined set of registers from the accelerometer, and has the following syntax:

```
lis3_read.sh <filename>
```

where <filename> refers to a file holding the list of registers with their respective address. Below is an example of this file (from *lis3_registers* provided along with the script):

```
# ------------------------------------
# Register name    Register Address
#                    (hex format)
# ------------------------------------
CTRL_REG1                20
CTRL_REG2                21
CTRL_REG3                22
STATUS_REG               27
FF_WU_CFG                30
FF_WU_SRC                31
FF_WU_THS_L              34
FF_WU_THS_H              35
FF_WU_DURATION           36
```

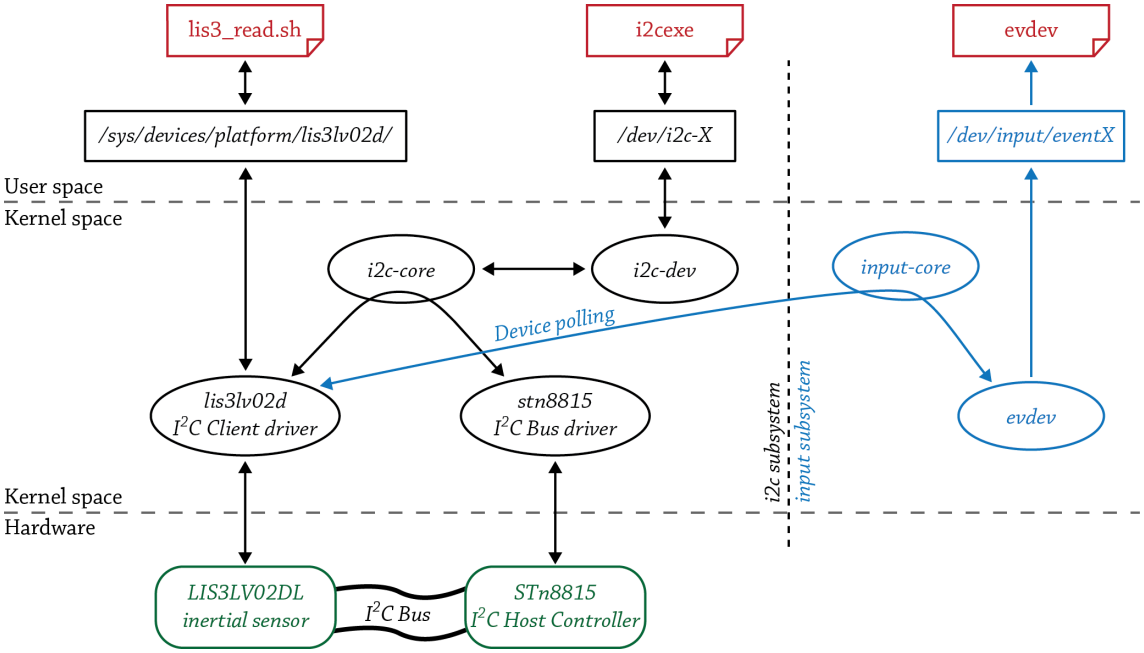

Figure 9: The LIS3LV02DL inertial sensor and STn8815 I²C host controller complete driver structure.

For each register in the list *lis3_read.sh* performs two steps:

1. Sets up the address of the register via sysfs `write`.
2. Reads the address content via sysfs `read`.

Below is the output of the script when invoked with *lis3_registers* as a command line parameter.

```
~ # ./lis3_read.sh lis3_registers
CTRL_REG1 = 47
CTRL_REG2 = 40
CTRL_REG3 = 08
STATUS_REG = FF
FF_WU_CFG = 15
FF_WU_SRC = 26
FF_WU_THS_L = 58
FF_WU_THS_H = 02
FF_WU_DURATION = 05
~ #
```

Figure 10: Output of the script *lis3_read.sh* with *lis3_registers* as a register file.

4.2 Testing via the *evdev* interface

In Linux the directory */dev* is conventionally used to store the device files, i.e. files representing interfaces for device drivers. They allow user programs to access hardware devices through their respective drivers. Linux classifies drivers in three categories: character, block and network drivers.
LIS3LV02D driver fits in neither of the above three categories, as it doesn't create a corresponding device file. Consequently it's not possible for a user program to directly operate on the accelerometer via a node in */dev*. Instead, upon enabling the device polling, an indirect access is provided by the *evdev* interface of the input subsystem via a `/dev/input/eventX` node. The accelerometer driver reports the events to the input core, which in turn writes them to this file in an appropriate format for being processed by a user program.

```
~ # evtest /dev/input/event0
Input driver version is 1.0.1
Input device name: "LIS3LV02D accelerometer"
Supported events:
  Event type 0 (Sync)
  Event type 1 (Key)
    Event code 240 (Unknown)
  Event type 3 (Absolute)
    Event code 0 (X)
      Value     54
      Min    -2000
      Max     2000
      Fuzz       3
      Flat       3
    Event code 1 (Y)
      Value     -3
      Min    -2000
      Max     2000
      Fuzz       3
      Flat       3
    Event code 2 (Z)
      Value  -1043
      Min    -2000
      Max     2000
      Fuzz       3
      Flat       3
Testing ... (interrupt to exit)
Event: time 566.775117, type 3 (Absolute), code 0 (X), value 66
Event: time 566.775130, type 3 (Absolute), code 1 (Y), value -5
Event: time 566.775137, type 3 (Absolute), code 2 (Z), value -1064
Event: time 566.775143, ---------- Report Sync ------------
Event: time 567.275153, type 3 (Absolute), code 0 (X), value -62
Event: time 567.275164, type 3 (Absolute), code 1 (Y), value 140
Event: time 567.275170, type 3 (Absolute), code 2 (Z), value -1053
Event: time 567.275177, ---------- Report Sync ------------
Event: time 567.449624, type 1 (Key), code 240 (Unknown), value 1
Event: time 567.449638, type 1 (Key), code 240 (Unknown), value 0
Event: time 567.449642, ---------- Report Sync ------------
```

Figure 11: Example of usage of *evtest*.

An example of such a program is *evtest* [5]. It is a free software released under GNU GPL license and available on-

line in many versions; 1.23 is the one used in this project. This tool allows to display the events reported by an input device. **Figure 11** shows a sample output from evtest on the target system.
Initially evtest shows some information about the input device and its supported event types (Sync, Key and Absolute) then, starting from the line `Testing ... (interrupt to exit)`, keeps displaying any data received from the event node. Since the accelerometer driver terminates each notification by a "sync" event, from the sample dump we can identify three distinct groups of events. Both the first and the second groups represent absolute events corresponding to the acceleration measured along the three cartesian axes. The third group regards the free-fall which is reported as a key press (value 1) and release (value 0) sequence of events.

4.3 Testing via the *i2c-dev* interface

The *i2c-dev* interface was designed to allow driving the I^{2}C devices from user space. Each I^{2}C adapter is assigned a number, say `X`, and a corresponding device file is created in `/dev/i2c-X`. It is indeed possible for a user mode driver to access an I^{2}C client by operating the adapter `X` through its node `/dev/i2c-X`.
However, in this project the *i2c-dev* interface was used with the only purpose of testing the I^{2}C bus driver. To this aim I wrote a user program called *i2cexe* which allows to transmit custom messages to a generic I^{2}C client. In the above scenario this tool plays the role of a user mode device driver without effectively being a real driver, as it performs only a single I^{2}C transaction.
Note that, unlike the testing techniques presented so far, this method requires no I^{2}C client driver whatsoever, i.e. only the I^{2}C bus driver under test is needed in order to communicate with any device on the bus. In addition, the Linux kernel must be configured with with "I2C device interface" (`CONFIG_I2C_DEV`) option enabled.
evtest can be used to transmit either a read message or a write message, depending on whether it is invoked with the command line option `-r` or `-w`. Below is its syntax:

```
i2cexe -r SLAVE_ADDR [-o offset] [-n nbytes]
i2cexe -w SLAVE_ADDR [-o offset] VALUE
```

where (each value is expressed in hexadecimal format):

`SLAVE_ADDR` is the address of the I^{2}C client.

`offset` is the offset relative to the client address space.

`nbytes` specifies the number of bytes to read (default 1).

`VALUE` is the value to write.

Figure 12 shows a sample run where eight bytes of data are read from the accelerometer device (client address 1D), starting at offset 28 (this area maps to three signed integer, stored in little endian format, and corresponding to the x-y-z acceleration values, which equals to 57,-5,-1071 in decimal).

```
~ # i2cexe -r 1d -o 28 -n 6
Reading from slave address 0x1D:
[offset] : value
----------------
[0x28] : 0x39
[0x29] : 0x00
[0x2A] : 0xFB
[0x2B] : 0xFF
[0x2C] : 0xD1
[0x2D] : 0xFB
~ #
```

Figure 12: Example of usage of *i2cexe*.

As explained, one major advantage of using the *i2c-dev* interface is the possibility to access any device on the I^{2}C bus without the need to install a kernel device driver for it. Indeed *i2cexe* can be used to communicate with I^{2}C devices other than the accelerometer. This idea is implemented by the shell script *i2cprobe.sh* which uses i2cexe to "ping" some devices from the list of table 1. The ping action consists of a simple read with no offset value. Upon receiving an answer the device is marked as *found*, meaning that the I^{2}C trasaction was performed correctly. Below is the output of the script:

```
~ # ./i2cprobe.sh
Probing LIS3LV02DL accelerometer at 0x1D...found
Probing TSC2003IPW touch screen controller at 0x48...found
Probing STw4811 power management at 0x2d...found
Probing STw5095 stereo audio codec at 0x1A...found
Probing STw4102 battery charger at 0x70...found
~ #
```

Figure 13: Output of the script *i2cprobe.sh*

i2cexe came in handy also to debug the I^{2}C bus driver during the first attempts to plug in the driver to the kernel or in presence of communication errors (e.g. time out). To help to examine these situations I used a digital oscilloscope connected to the SDA and SCL line of the I^{2}C bus. **Figure 14** shows the I^{2}C bus waveforms of a current address read at standard speed (100kHz).

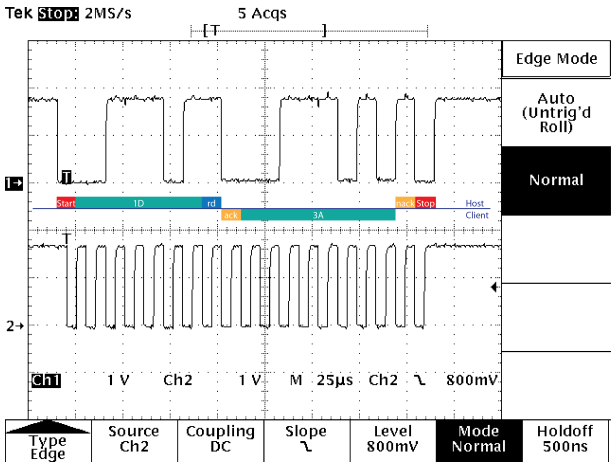

Figure 14: SCL and SDA waveforms of a current address read operation performed at standard speed (SDA on ch1, SCL on ch2).

After the START bit the host transmits the 7-bit slave address 1D and the read bit. The client sends back an

acknowledge bit (ack) followed by the value 3A. Then the host terminates the transaction by a not-acknowledge (nack) and a stop bit.

To verify the controller speed configuration, the same transaction was performed in fast speed mode (400Khz) as well. **Figure 15** shows the corresponding I^{2}C bus waveforms.

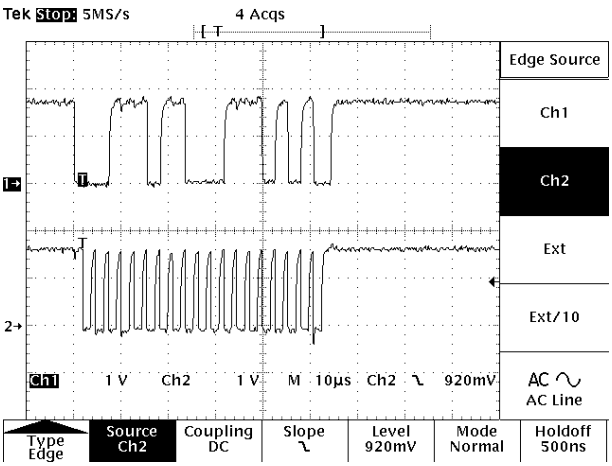

Figure 15: SCL and SDA waveforms of a current address read operation performed at fast speed.

The sawtooth-like characteristic of the signals is due to the open-drain configuration of the bus, which is essential to allow concurrent operation of multiple master, but on the other hand, negatively affects the bandwidth. This is especially evident in the rising edge because the high level is achieved by terminator resistors which "pulls" the line back up when all devices release it. Together with the wire capacitance these pull-up resistors are responsible for the exponential rising edges.

Here is a further example of using *i2-dev*, this time to access the on-board battery charger (slave address 70). For example, the command `i2cexe -r 70 -o 2` performs a random read of one byte from slave address 70.

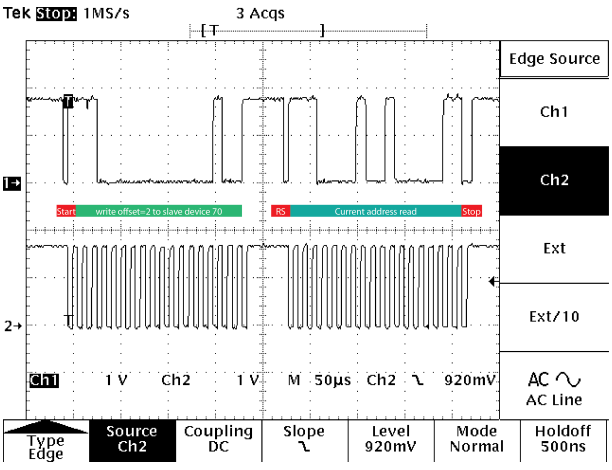

Figure 16: SCL and SDA waveforms of a random read operation from slave address 70 at offset 2.

The corresponding I^{2}C bus waveforms of **Figure 16** reveal that the whole transaction is broken down in two parts. First, the host writes the offset where the desired read will start (2 in the example). Then the hosts sends a repeated start bit followed by a current address read instruction.

5 Conclusion

This paper has dealt with the I^{2}C device driver design in a Linux embedded system environment. Due to the tight coupling between these types of drivers and the underlying hardware, usually their development is undertaken by the manufacturer of the embedded system. Therefore most of the commercial boards ship with a tailored Linux distribution which has all the drivers necessary for the devices on the board. This way the engineers can focus on developing the application specific software rather than building kernel components.

Nevertheless, this paper gives an insight into how the Linux kernel supports I^{2}C , making it helpful for those who need to modify an existing implementation to fit their own needs.

Some methods to interface the device driver with the user land have been presented as well. Although implemented for testing purposes, these methods also apply to real applications. Many other solutions may be adopted to extend this work in many ways. Some example include updating the accelerometer driver with a character driver interface and power saving features, adding support for I^{2}C high-speed mode to the bus driver, etc.

This report and all the source code are publicly available through the git repository at `https://github.com/fghiro/i2c-nomadik`.

References

[1] Venkateswaran, S.: Essential Linux Device Drivers. Prentice Hall Open Source Software Development Series. Prentice Hall (2008)

[2] BusyBox. `http://www.busybox.net/`

[3] Rubini, A.: bbfs 1.3 - building a filesystem based on busybox alone. `http://www.gnudd.com/wd/bbfs.html`

[4] Das U-Boot, the Universal Boot Loader. `http://www.denx.de/wiki/U-Boot/WebHome`

[5] Evtest, the Event Tester. `http://www.freedesktop.org/wiki/Evtest`

[6] Corbet, J., Rubini, A., Kroah-Hartman, G.: Linux Device Drivers. Oreilly Software Series. O'Reilly Media, Incorporated (2005)

[7] Yaghmour, K., Masters, J., Ben-Yossef, G., Gerum, P.: Building Embedded Linux Systems. Oreilly Series. O'Reilly Media, Incorporated (2008)

[8] Sally, G.: Pro Linux Embedded Systems. Apresspod Series. Apress (2009)

[9] Love, R.: Linux Kernel Development. Developer's Library. Addison-Wesley (2010)

[10] The I2C bus specification. http://www.nxp.com/documents/user_manual/UM10204.pdf

[11] The Linux Kernel Archives. http://www.kernel.org/

[12] Loeliger, J., McCullough, M.: Version Control with Git: Powerful tools and techniques for collaborative software development. O'Reilly Media (2012)