

**Instituto Tecnológico de Costa Rica**

**Área Académica en Ingeniería en Computadores**  
(Computer Engineering Academic Area)

**Programa de Licenciatura en Ingeniería en Computadores**  
(Licentiate Degree Program in Computer Engineering)

**Curso: CE-4303 Principios de Sistemas Operativos**  
(Course: CE-4303 Principles of Operating Systems)



**Proyecto #2: I2C**  
(Project #2: I2C driver)

**Estudiantes**  
(Student)

Nicolás Jiménez García - 201258421  
Kevin Umaña Ortega - 201144881

**Profesor:**  
(Professor)

Ing. Alejandra Bolaños Murillo

**Fecha de entrega: 17 de junio de 2017**  
(Due date: Saturday 17<sup>th</sup> June, 2017 )

# Índice

<b>1. Introduction</b>	<b>2</b>
1.1. I2C . . . . .	2
1.2. Drivers . . . . .	3
<b>2. Technical Specifications</b>	<b>5</b>
2.1. Driver . . . . .	5
2.2. Library . . . . .	6
2.3. Web Application . . . . .	6
<b>3. Program Design</b>	<b>7</b>
3.1. Driver . . . . .	7
3.2. Library . . . . .	9
3.3. Application . . . . .	9
<b>4. Usage Instructions</b>	<b>10</b>
4.1. Driver . . . . .	10
4.2. Library . . . . .	10
4.3. Application . . . . .	10
<b>5. Log</b>	<b>10</b>
<b>6. Project State</b>	<b>11</b>
<b>7. Conclusions</b>	<b>11</b>
<b>8. Recommendations</b>	<b>12</b>

# 1. Introduction

This project is about creating a system of Hardware and Software, to control a 8x8 led matrix through a VGA connector using the I2C protocol. The matrix is controlled in Software by a Linux driver and a library for accessing the matrix and painting different characters to the led matrix. The users access to the library functions using a responsive web application which is connected in turn to a REST WEB Application Protocol Interface (API) which in turn is connected to the library. This library does parsing tasks of the message sent from the upper layers. Once the methods in the library decompose the message, the library talks to the driver which is in charge of sending bytes to the LED matrix so that the message gets displayed on it.

This project relies on the I2C protocol to communicate the led matrix with the Linux driver. Another important part of this project is the Linux Driver that controls the Hardware. The concepts of I2C and drivers are explained ahead.

## 1.1. I2C

I2C was originally developed in 1982 by Philips for various Philips chips. The original spec allowed for only 100kHz communications, and provided only for 7-bit addresses, limiting the number of devices on the bus to 112 (there are several reserved addresses, which will never be used for valid I2C addresses). In 1992, the first public specification was published, adding a 400kHz fast-mode as well as an expanded 10-bit address space. [1]

In addition to “vanilla” I2C, Intel introduced a variant in 1995 call “System Management Bus” (SMBus). SMBus is a more tightly controlled format, intended to maximize predictability of communications between support ICs on PC motherboards. The most significant difference between SMBus is that it limits speeds from 10kHz to 100kHz, while I2C can support devices from 0kHz to 5MHz. SMBus includes a clock timeout mode which makes low-speed operations illegal, although many SMBus devices will support it anyway to maximize interoperability with embedded I2C systems. [1]

The i2C bus is a very popular and powerful bus used for communication between a master (or multiple masters) and a single or multiple slave devices. Figure 1 illustrates how many different peripherals may share a bus which is connected to a processor through only 2 wires, which is one of the largest benefits that the I2C bus can give when compared to other interfaces. Figure 1 shows a typical I2C bus for an embedded system, where multiple slave devices are used. The micro-controller represents the I2C master, and controls the IO expanders, various sensors, EEPROM, ADC's/DAC's, and much more. All of which are controlled with only 2 pins from the master. [2]

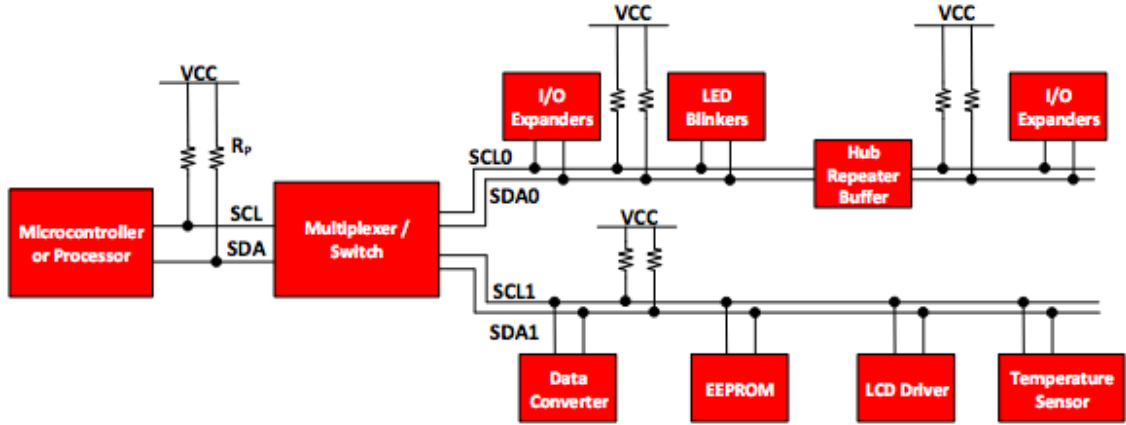


Figura 1: Example of i2C bus

I2C uses an open-drain/open-collector with an input buffer on the same line, which allows a single data line to be used for bidirectional data flow. [2]

Each I2C bus consists of two signals: SCL and SDA. SCL is the clock signal, and SDA is the data signal. The clock signal is always generated by the current bus master; some slave devices may force the clock low at times to delay the master sending more data (or to require more time to prepare data before the master attempts to clock it out). This is called “clock stretching” and is described on the protocol page. [1]

Unlike UART or SPI connections, the I2C bus drivers are “open drain”, meaning that they can pull the corresponding signal line low, but cannot drive it high. Thus, there can be no bus contention where one device is trying to drive the line high while another tries to pull it low, eliminating the potential for damage to the drivers or excessive power dissipation in the system. Each signal line has a pull-up resistor on it, to restore the signal to high when no device is asserting it low. [1]

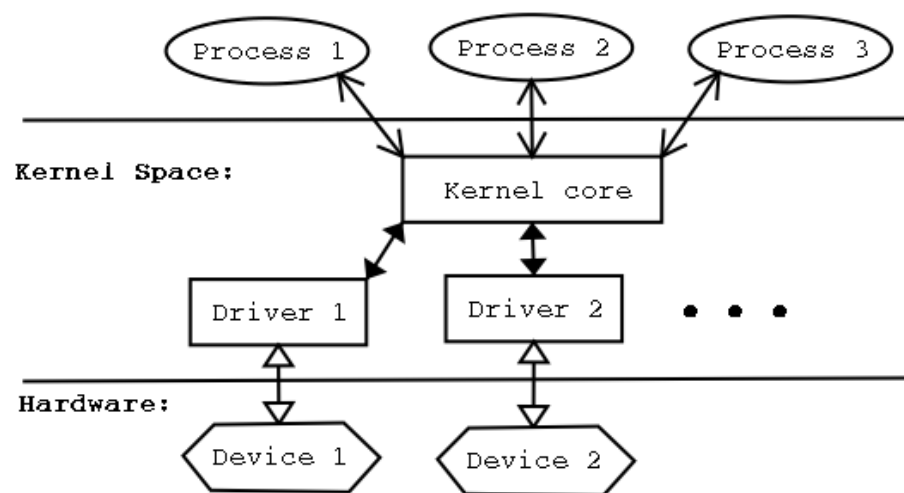
## 1.2. Drivers

Since in this project we need to create a driver for a new device, it is convenient to give a brief introduction about drivers. Drivers in Linux can be classified under two categories: block devices and char devices. The first ones refer to devices with which the Driver communicates by sending and receiving single characters (bytes, octets). For example, we have serial ports, parallel ports, sound cards. A Block (‘b’) Device is one with which the Driver communicates by sending entire blocks of data. Examples for Block Devices are: hard disks, USB cameras, Disk-On-Key.

The Linux kernel is a computer program, that acts as the operating system. It allows applications to access the Hardware, in a more-or-less generic manner. The kernel gets loaded when the system starts executing, and launches processes that allow us to login and run programs. It is made of a ‘core’ residing in a single file, and a bunch of smaller Modules that are only loaded if they are needed.

In figure 2, we observe in a very general but clear way how an operating system works. For this project, we handle all the communication from top to bottom. At first, the user process consists on the web application that allows the user to enter a message. This message is manipulated in a user space library in order to parse its contents and finally be delivered to the driver, which is in kernel space, that communicates directly with the hardware of the system (LED matrix) by writing directly to certain memory addresses of the device which in turn display the message received.

**User Space:**



**Legend:**







	— Process		— User-Kernel Communications (system calls, signals)
	— Code Module		— Inter-Kernel Communications (kernel API)
	— Hardware		— Hardware Communications (interrupts, ports I/O)

Figura 2: Abstract layout of an operating system. [5]

## 2. Technical Specifications

In this section some technical topics are covered related to the development of the project.

### 2.1. Driver

As mentioned before, the Driver is the module in charge of the communication with the specific hardware component (LED matrix) via the I2C protocol. For this project it was mandatory to implement it in C language. This is because the Linux kernel code is developed in this language and the low-level operations regarding interaction between files are handled effectively. First of all it was necessary to understand how creating a new module in the kernel was to be done without needing to recompile the whole core of the operating system.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h> /* printk() */
#include <linux/slab.h> /* kmalloc() */
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h> /* O_ACCMODE */
#include <asm/uaccess.h> /* copy_from/to_user */
#include <linux/i2c.h>

MODULE_DESCRIPTION("VGA-i2c driver");
MODULE_AUTHOR("Kevin Umaña <kevgiso@hotmail.com>");
MODULE_LICENSE("GPL");

int memory_major = 89;

/* Declaration of vgai2c.c functions */
static int vgai2c_probe(struct i2c_client *client,
const struct i2c_device_id *id);
static int vgai2c_remove(struct i2c_client *client);
```

As can be noted in the portion of code above, the headers differ slightly to headers commonly used in user space programs. These headers manage lower level operations, dealing with memory and direct access to hardware registers, for example. Since i2c communication is held serially, the implemented driver was developed as a char device. Nevertheless, i2c clients needed special structures in order to define the new module as a driver compliant with the standards. Instead of the more common file operations struct, which generally is in charge of binding user space operations such as write, read, open and release memory, with kernel actions, this struct is not used in i2c new modules. The actions defined for i2c clients are probe and remove, the first one gets called as soon as the device is inserted in the kernel. This is usually done executing the **insmod** command in terminal, providing the corresponding module once it is especially compiled as a kernel module. The remove function gets called immediately after the module is removed from the kernel, normally using the command **rmmod**. The following set of lines show how this kind of module is compiled.

```
CONFIG_MODULE_SIG=n
```

```
obj-m = vgai2c.o
KVERSION = $(shell uname -r)

all:
make -C /lib/modules/$(KVERSION)/build M=$(PWD) modules
clean:
make -C /lib/modules/$(KVERSION)/build M=$(PWD) clean
```

The code above specifies the compilation process of a kernel module, staying unchanged independently of the kernel version used.

In terms of the direct communication between the driver and LED matrix, the driver sends a binary matrix of type **uint8** that is written row per row in the corresponding registers of the matrix, allowing it to display the characters properly. The message buffer sent to the hardware consists on the data that is to be showed and the address of the register that will receive that information.

## 2.2. Library

The library was developed to make the driver usable by external programs, it is developed in the C programming language. The library consists on several programming files which are grouped to generate only one executable file. The library works by monitoring a JSON file with a message: "message": "message" If this file is changed, the library reads the file and parses it to extract the message. After extracting the message it calls the driver in order to start painting the matrix.

The code was developed in C and requires some Linux only libraries, which are showed next.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <linux/inotify.h>
#include <string.h>
```

The library consists of the files: `inotify.c` and `readFile.c`. The file `readFile.c` is included in `inotify.c`, so at the end only one executable is created and run.

## 2.3. Web Application

The web application is made with Javascript, the Javascript framework AngularJS and the CSS framework Bootstrap. In AngularJS website, it states that “as HTML is great for declaring static documents, but it falters when we try to use it for declaring dynamic views in web-applications. AngularJS lets you extend HTML vocabulary for your application. The resulting environment is extraordinarily expressive, readable, and quick to develop.” [3] AngularJS is a very popular framework extensively used to build Web Applications, specially Single Page Applications. Its main advantages are:

- Optimal for Dynamic single page applications.
- After page load can work in the client side without reloading.
- Easy manipulation of the DOM (Document Object Model).
- Support for REST architecture and allows REST methods. [3]

Javascript was chosen for the web application for the following reasons:

- It can be used in the front-side and server-side. PHP on the other hand, only works in the server side.
- The team has already Javascript and AngularJS knowledge, but doesn't have PHP knowledge.
- AngularJS stimulates the use of good design patterns such as MVC (Model View Controller).

### **RESTFUL Protocol**

The Web Application uses the RESTFUL protocol to communicate with the server. The data moved between the Application and the Server is JSON. The Web API is built with Javascript code and Node and it is powered by Node.js. This API provides methods to get messages from the application. This API performs some I/O operations in the server and send its output to the led matrix library. Its output is the JSON obtained by the application.

## **3. Program Design**

In this section each of the layers of the system are explained in detail.

### **3.1. Driver**

For the Driver implementation it was mandatory to use specific low-level operations provided in i2c headers of the Linux core. The created header looks as follows.

```
#ifndef _VGAI2C_H
#define _VGAI2C_H
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/types.h>
#include <linux/i2c-dev.h>

#define ADAPTER 1
#define VGAI2C_ADDRESS 0x70

int writeToMatrix(uint8_t character[8]);

#endif
```

Besides the common header files used for file and strings handling, the i2c-dev.h file provides functions in order to perform writes to devices directly. Since this writes need to be done using the i2c main bus of the VGA, we first had to detect in which bus of the i2c adapter our device was located. After we figured this out, as well as the address of our vgai2c device, we were ready to go.



The only operation defined in the driver was the writeToMatrix function. This method is going to receive as a parameter a binary matrix, holding the information of the leds that have to be turned on or off in our display. This function is shown ahead.

```
int writeToMatrix(uint8_t character[8]){
int file;
char filename[20];

__u8 reg [8]= {0x0, 0x2, 0x4, 0x6, 0x8, 0xa, 0xc, 0xe}; /* Device register to ac
char buf[18];

snprintf(filename, 19, "/dev/i2c-%d", ADAPTER);

file = open(filename, O_RDWR);
if (file < 0) {
    printf("Unable to open file.\n");
    exit(1);
}

if (ioctl(file, I2C_SLAVE, VGAI2C_ADDRESS) < 0) {
    printf("Communication with i2c-%d failed.\n", ADAPTER);
    exit(1);
}

buf[0] = reg[0];
buf[1] = character[0];
buf[2] = reg[1];
buf[3] = character[1];
buf[4] = reg[2];
buf[5] = character[2];
buf[6] = reg[3];
buf[7] = character[3];
buf[8] = reg[4];
buf[9] = character[4];
buf[10] = reg[5];
buf[11] = character[5];
buf[12] = reg[6];
buf[13] = character[6];
buf[14] = reg[7];
buf[15] = character[7];

if (write(file, buf, 16) != 16) {
    printf("Error when writing to matrix.\n");
    exit(1);
}
char buf2 [3];
buf2[0] = 0x81;
buf2[1] = 0x00;

if (write(file, buf2, 2) != 2) {
    printf("Error when writing to matrix.\n");
```

```

    exit(1);
}

return 0;

}

```

First of all, a set of registers in hexa is defined. This array holds the device registers that are going to be written by our method.

Then, the device filename is obtained according to the i2c bus holding our device. In this case our device can be accessed via the bus number one of the adapter. We proceed to open the file with read and write rights. It is important to recall that these operations are executed as a super user.

Once we gain access to the file, we need to set it explicitly as a slave device. This in order to be able to send instructions from the master device which is in turn our computer. The file descriptor is specified, as well as the address of our device and the macro **I2C SLAVE** which indicates that the device located in that address will be treated as a slave.

When this is done, we fill the buffer that is going to be sent to the registers in the LED matrix. The information held by this buffer consists on the value that is going to be written and the register holding it. The write operation is performed and the information, if everything turns out properly, gets displayed on the led matrix. Finally another write operation is done in the special register **0x81** so that the leds are prevented from blinking.

### 3.2. Library

As mentioned before, the library consists on two files, `inotify.c` and `readFile.c`. At a high level, the library monitors a JSON file (which is produced by the Web API). When the library detects a change in the file, it reads it and parses it to extract the message. After doing getting the message, the library calls the driver so that the message can be painted in the led matrix.

At a low level, the file `inotify.c` uses the Linux library `sys/inotify.h`, this library has the ability of monitoring changes in a filesystem, it has a list of directories under monitoring and the events which it watches such as: modification, creation, deletion, among others. [6] The program `inotify.c` is constantly searching for changes in the monitored filesystem and when it finds out that a monitored file changed (in this case only `texto.json`) it triggers the function `SearchInFile()` from the file `readFile.c`. This function reads and parses the file and then it returns the message string back to the program `inotify.c`.

When the message string is obtained and ready, the library calls the function `drawMatrix` from the driver file, and then the library starts over to keep monitoring the file `texto.json`.

### 3.3. Application

This an AngularJS application which consists as any website of HTML, CSS and Javascript. The application has a root directory and to sub-directories. In the root directory the files are: `index.html` and `app.js`. `index.html` is the starting point for the web app and all the other files are included there. `app.js` is the stating point for the angular modules, controllers and services, it also defines the routing of the app. The sub- directories are `app-services` and `led`. In `app-services` it is defined the service for the REST communication with the Web

API which is a POST method. And also there is a service to notify success/errors in the user interaction with the web app.

### **Web API**

The Web API is another important part of the project and is basically comprised of two files: `package.json` and `server.js`. The file `package.json` defines the dependencies for the Web API of node.js, in this case it is used: The Express.js framework, The Body-Parser.js framework among others. These frameworks are used to build the REST interface with the Web APP. [4] The file `server.js` contains the actual code that is ran when the server (Web API) starts, thus providing the POST method for the Web Application. The Web API at the end stores in a file the JSON (message) obtained from the user in the Web application.

## **4. Usage Instructions**

This project consists of several parts some of them run separately and need different setup.

### **4.1. Driver**

Once the driver is inserted in the Linux kernel, using `insmod` as mentioned before, the driver is ready to get going.

The driver was implemented in a way that the only method, which is the one that writes to the led matrix, is called from the library. The file that includes this implementation only needs to be imported from the library main file.

### **4.2. Library**

To use the library change the directory to `vgai2c/library`. Then execute the command `gcc -o library inotify.c`

### **4.3. Application**

#### **Web App**

In a terminal execute the command `npm install http-server -g`. This step requires node and npm. Change the directory to `vgai2c/WebApp`. Then execute the command `http-server`. Finally, in a web browser go to the direction provided by the `http-server` program. The web browser must be in the same network as the server that hosts the web page.

#### **Web API**

In a terminal change the directory to `vgai2c/WebAPI`. Then execute the command `npm install`. Finally, execute the command `npm run start`.

## **5. Log**

### **■ Kevin**

1. Research deeply about drivers.
2. Implement driver and routine to display user message on LED matrix.
3. Develop functions in charge of parsing the messages from web api and convert them to appropriate format understood by hardware.
4. Integrate low level module with user space library.

5. Build LED matrix circuit with VGA connector.
  6. Implementation of Makefiles.
  7. Exhaustive testing.
- Nicolás
    1. Research deeply about drivers.
    2. Develop the web application with REST support.
    3. Develop the web Application protocol interface with REST support.
    4. Connect the web application and the web API.
    5. Create the library to connect the driver with other applications.
    6. Connect the web API with the library.
    7. Exhaustive testing.
    8. Generating of Checksum (SHA-256).

## 6. Project State

Fortunately, the project is fully functional. The objectives stated in the document specification to implement a driver, connect it to a library in user space which in turn interacts with a web API and user mobile application. These objectives were successfully completed, allowing the developers to understand the whole communication process from top to bottom in an operating system.

Difficulties encountered were mainly related to the driver itself and how the kernel manages new devices in an i2c bus. This topic really gave a hard time due to the complex structures handled in low-level kernel files.

## 7. Conclusions

- C programming language is of mandatory knowledge when dealing with low-level structures and operations between devices and user programs.
- I2C drivers need the creation of new clients when trying to add a new device for which the core does not have support. Definition of the i2c device address and operations to be done when the driver is inserted in the kernel are of great impact in the device performance.
- The publisher/subscriber pattern is a simple, scalable and highly compatible way of sharing information and messages between a library and an application or between applications. This pattern was used in this project between the library and the application, since they are written in different programming languages.
- The JSON data format is an ideal format for communication between program interfaces, because it is lightweight, highly compatible and easy to understand.
- Javascript is an ideal language for writing web-based applications, since it can be used in the client-side to build dynamic, single page applications with frameworks such as AngularJS. And in the server-side with frameworks such as Node.js and Express.js to build Web API's with REST capabilities. All under only one language.

## 8. Recommendations

In the future it is recommended the use of a more modular approach in terms of creating various files to reduce dependency among software components.

The use of one and only Makefile is of great utility in order to compile programs faster, avoiding time loss compiling one after the other.

Implementing this driver with a char device approach would have made it simpler to accomplish the same operations implemented in a more automatic way, sending bytes to the specific device file when making write operations.

## Referencias

- [1] Learn.sparkfun.com. (2017). I2C - learn.sparkfun.com. [online] Available at: <https://learn.sparkfun.com/tutorials/i2c> [Accessed 17 Jun. 2017].
- [2] Valdez, J. and Becker, J. (2015). Understanding the I2C Bus. [ebook] Dallas, Texas. Available at: <http://www.ti.com/lit/an/slva704/slva704.pdf> [Accessed 17 Jun. 2017].
- [3] Angularjs.org. (2017). AngularJS — Superheroic JavaScript MVW Framework. [online] Available at: <https://angularjs.org/> [Accessed 17 Jun. 2017].
- [4] Foundation, N. (2017). Node.js. [online] Nodejs.org. Available at: <https://nodejs.org/en/> [Accessed 17 Jun. 2017].
- [5] "The Linux Kernel, Kernel Modules And Hardware Drivers", Haifux.org. [Online]. Available: <http://haifux.org/lectures/86-sil/kernel-modules-drivers/kernel-modules-drivers.html>. [Accessed: 10- Jun- 2017].
- [6] Ibm.com. (2017). Monitor Linux file system events with inotify. [online] Available at: <https://www.ibm.com/developerworks/library/l-inotify/index.html> [Accessed 17 Jun. 2017].