Design and development of a wireless lighting control demonstrator

Vicenç Pio Badia, IT Eng.
Nimbus Centre for Embedded Systems Research
IT Engineering, Polytechnic University of Catalonia

Supervisor: Conrad Dandelski, B.Eng.

A thesis submitted for the degree of

IT Engineering

at Cork Institute of Technology, June 2016
Abstract

Vicenç Pio Badia, IT Engineering, Politechnical University of Catalo-
nia
Abstract of Bachelor’s Thesis, Submitted 10 June 2016:
Design and development of a wireless lighting control demonstrator.

The purpose of this thesis is to develop a demonstrator for a wireless
lighting control use case. The goal is to create a lighting network
formed by one master node and several slave nodes each of them
equipped with a PCB including an RGB LED that can be controlled
wirelessly. This project was conducted at the Nimbus Centre for Em-
bedded Systems located at the Cork Institute of Technology in Ireland
as my final bachelor thesis.

At first, the different design alternatives are discussed, followed by an
overview of the system. Next, the different hardware components are
described and their function within the demonstrator. In the next
chapter the software design is discussed, in special the programs de-
developed for the web application, the servers and the software running
on the nodes. In conclusion, the thesis demonstrates the proper op-
eration of a lighting control use case that allows the user controlling
the colour level of an RGB LED wirelessly.
Contents

1 Introduction ........................................ 1
   1.1 Nimbus ........................................ 1
   1.2 The DEWI project .............................. 2
   1.3 Lighting Use Case ............................. 2
   1.4 Description of work ........................... 2

2 Design approaches ..................................... 4
   2.1 Requirements .................................. 5
   2.2 Overview ...................................... 5

3 Hardware design ..................................... 7
   3.1 Raspberry Pi ................................. 7
   3.2 Re-Mote ....................................... 8
   3.3 Tablet ........................................ 9
   3.4 PCB design .................................... 10
       3.4.1 Schematic Capture ....................... 11
       3.4.2 PCB Layout ............................... 11
       3.4.3 Soldering .................................. 13

4 Software design .................................... 15
   4.1 Front end ................................... 15
       4.1.1 Colour chart ............................. 16
   4.2 Back end ..................................... 17
       4.2.1 Node.js .................................. 18
       4.2.2 Serial port .............................. 18
## CONTENTS

4.2.3 WebSocket ................................................. 18
4.3 Nodes ...................................................... 19
    4.3.1 Control of the LED ................................. 20

5 Testing and validation ...................................... 23
    5.1 Tools ................................................... 25

6 Conclusion and Outlook .................................. 26
    6.1 Review of Contribution ................................. 26
    6.2 Lessons Learned ....................................... 26
    6.3 Outlook and future work ............................. 27

Appendices ..................................................... 29

A Web page ................................................... 30
    A.1 Welcome page ......................................... 30
    A.2 Colour chart .......................................... 34

B WebSocket Server ........................................... 40

C Software running on the nodes ......................... 44
## List of Figures

2.1 Demonstrator layout ........................................ 6  
3.1 Re-Mote platform ............................................. 9  
3.2 Schematic ...................................................... 11  
3.3 PCB layout ..................................................... 12  
3.4 Gerber files .................................................... 13  
3.5 PCB .......................................................... 14  
4.1 Colour chart .................................................... 16  
4.2 From the web page to the Re-Mote .......................... 19  
4.3 Data received on the master Re-Mote ....................... 22  
5.1 Demonstrator ................................................... 24
Chapter 1

Introduction

This thesis presents the outcomes of my final year project at the Nimbus Centre for Embedded Systems Research located at the Cork Institute of Technology in Ireland. The objective of the project is develop up a demonstrator for a wireless lighting control use case to demonstrate in a later stage the correct operation of the protocols developed within the DEWI project. The implementation of those protocols is not part of this work.

1.1 Nimbus

The Nimbus Centre\(^1\) for Embedded Systems Research is part of Cork Institute of Technology (CIT), a institute with over 17,000 students in engineering, science, business, art and music. Nimbus is devoted to embedded electronic systems and the 'Internet of Things' (IoT), and focuses the research expertise on the following industrial application areas: Energy Management, Water Systems Management, Smart Systems and Applications in Tourism, Agriculture and Health. My work is supporting the DEWI project, which is part of the Smart Systems.

\(^1\)http://nimbus.cit.ie/
1.2 The DEWI project

Dependable Embedded Wireless Infrastructure (DEWI) is an EU Artemis Funded initiative to provide key solutions for seamless wireless connectivity in smart cities and infrastructures. DEWI wants to foster Europe’s leading position in embedded wireless systems and smart environments for both personal and professional users. In total, DEWI is supported by 58 partners in Europe including the Nimbus Centre, which is contributing to the research in dense wireless networks.

1.3 Lighting Use Case

A lightning network in an indoor building can be large, from around 500 nodes in a typical office to 2000 nodes per floor. Not every lamp needs to be controlled individually but might be equipped with its own radio transceiver. It means that most of the radios are in reach of each other and this fact gives high chance of message collision and controlling the lights by flooding the network becomes less effective. In applications such as office illumination, the user expects a system response within a few hundred milliseconds (latency) with low outage rates (packet loss rate). After pressing a light switch, if the system is reacting slowly, the user will be confused and may keep pressing the switch and generating more messages, which will worse the situation. Additionally, in case only a few lamps turn on, the user will consider these lamps as broken and need to be replaced. We can finally state that the requirements for an indoor lighting network are a latency of less than 200ms and a packet loss rate below 1%.

1.4 Description of work

The aim of this work is to develop a demonstrator for a wireless lighting control use case. Therefore, a review of the used protocols and standards, such as the RLL protocol and the IEEE802.15.4e protocol is necessary, those protocols form the basis of the complete protocol stack. The demonstrator uses the

\footnote{http://www.dewiproject.eu/}
1.4 Description of work

open source operating system Contiki\(^1\), Contiki is an operating system for networked, memory-constrained systems with a focus on low-power wireless Internet of Things devices.

The main objective of this work is to design a PCB consisting a RGB LED which should be controllable wirelessly by receiving commands from one central device. To change the colour level of the LEDs a web application needs to be implemented, which will send the RGB values to the central device.

\(^1\)http://contiki-os.org/
Chapter 2

Design approaches

A previous experimental implementation related to this project was already set up in 2015. It was a simple solution demonstrating a lighting network with only a few master and slaves nodes, the LEDs equipped on each node were switched on and off using the user button in one of the master nodes. The main differences between that implementation and the one developed with this work are the LEDs, the hardware and the used operating system. First, the used LEDs were single colour and in the new implementation are RGB LED type. Besides, the used hardware platform was the TelosB and the operating system was OpenWSN. The here developed demonstrator uses a new hardware platform, called Re-Mote. The reasons for these changes are the better features given by the RE-Mote compared to the TelosB, such as more memory and a newer microprocessor. The demonstrator will use Contiki instead of OpenWSN. The first tests with the new hardware running OpenWSN were not successful and after experiencing more problems, the decision were made to change the operating system to Contiki, also because the Re-Mote is fully ported for this operating system. Regarding the light switch, the possible options were either to develop a web application or a mobile application. The web application has the advantage to be accessible by different devices, while the mobile application is exclusive for each operating system. Those reasons made the web application more suitable for a demonstrator and therefore was chosen. The communication between the web application and the central device could be either by Bluetooth or WiFi. Since the device used in the beginning was the
2.1 Requirements

Raspberry Pi 2 and we already had the WiFi dongle, the decision was made to use WiFi connection. In the middle of the project, the Raspberry Pi 2 was replaced for the newer Raspberry Pi 3 with built-in Bluetooth and Wifi.

2.1 Requirements

Initially we had some requirements to accomplish for the operation of the system. The first one was the system to work offline, it means all the packages and libraries used in the software must be stored locally in each device, with no need to access on the Internet. Another requirement needed was to build a plug and play system so that when booting up it is ready to work without entering any commands. As explained before, it is important to make the system suitable for as many devices as possible with different operating systems. That was taken into account when deciding for the web application instead of the mobile application. Finally the system should be easy to transport, with components as light as possible.

2.2 Overview

The Figure 2.1 shows a chart overview of the use case. On the left side the tablet, the Raspberry Pi and the master node performing as the light switch can be seen. The Raspberry Pi provides WiFi connection to the tablet and also a web server hosting the web application and a web socket server used for the serial communication with the master node. On the right side, the slave nodes receive broadcast messages from the master node wirelessly and represent the lamps of the system. The Raspberry Pi is mains powered, and provides the power connection for the master node via the USB cable. The slave nodes are powered by a built in battery. Overall the operation of the system starts selecting the colour using the tablet that sends the data to the Raspberry Pi which then transmit it to the master node. The master node controls its own RGB LED and broadcasts the message to the slave nodes, which also control its own RGB LED.
2.2 Overview

Figure 2.1: Demonstrator layout
Chapter 3

Hardware design

The demonstrator is based on different hardware components. The purpose of each device is explained in this chapter.

3.1 Raspberry Pi

The Raspberry Pi 3 Model B [1] is a single board computer developed by the Raspberry Foundation. The Raspberry Pi is based on a 64-bit quad-core ARMv8 CPU with a takt rate of up to 1.2GHz. The used model is the latest version of the Raspberry Pi and compared to previous versions it comes with built in WiFi, BLE and a newer CPU version. For the demonstrator Raspbian as operating system is used. Raspbian is the official operating system developed and maintained by the Raspberry foundation and is based on Debian.

A Raspberry Pi 2 Model B can also be used instead of the Raspberry Pi 3 if WiFi dongle is plugged in.

The Raspberry Pi is used for hosting two servers:

1. The Apache 2 HTTP Server for the website.
2. The WebSocket Server for the serial communication with the master node.

Both servers are hosted on the Raspberry Pi, provided by an external MicroSD card of 8GB capacity. Thus, we use a single device centralizing the servers receiving the data that will be sent to the master RE-Mote through the serial port.
3.2 Re-Mote

One of the initial requirements was the system to work offline, therefore all the packages included by the servers and any files needed during the configuration are stored on the device.

The last feature given by the Raspberry Pi is providing access to the web page. As it is working offline, the solution implemented is a wireless access point, using the Wi-Fi connection, it allows the tablet to access the Apache server in the Raspberry Pi by the local network created. The access point is implemented by using `hostapd`, a user space daemon that runs in the background.

### 3.2 Re-Mote

The hardware platform used as nodes is the Zolertia RE-Mote\(^2\) (Figure 3.1). The Re-Mote is an Internet of Things hardware development platform based on the Zoul core module and is the state-of-the-art in the IoT technology field. The platform is based on the Texas Instruments CC2538 ARM Cortex-M3 system on chip (SoC), with an on-board 2.4 GHz IEEE 802.15.4 RF interface, running at up to 32 MHz with 512 KB of programmable flash and 32 KB of RAM, bundled with a Texas Instruments CC1200 868/915 MHz RF transceiver to allow dual band operation.

The 6LoWPAN protocol support, the ultra-low power operation and the I2C interface were the main reasons to select the Re-Mote.
3.3 Tablet

A Microsoft Surface Pro tablet, running on Windows 8.1 is used as light switch/dimming device. The tablet is connected to the Raspberry Pi access point to access the website hosted on the Raspberry Pi. The DHCP server is disabled, due to university policy. Therefore, it is necessary to define a static IP address for the tablet. The double finger zoom and the overscroll have been disabled on the Chrome browser because these services stepped over the colour chart, annulling its functionality. Those services can be disabled by running the following commands on the browser’s address bar:

- chrome://flags/#enable-pinch disabled
- chrome://flags/#overscroll-history-navigation disabled
The Re-Mote platform is equipped with an RGB LED by default but for the demonstrator a LED with a better brightness is necessary. It leads us to connect a different RGB LED to the RE-Mote and therefore the design of a PCB is necessary. The first idea was to use the PWM output pins from the RE-Mote and make a direct connection to the three input pins of the LED. The problem was that, according to the RE-Mote documentation, the pins don’t provide enough output current to supply the LED.

The alternative and final solution was to use the I2C pins from the Re-Mote and make the connection to the LED through an RGB LED driver. The function of the driver is to receive the input data from the I2C bus and generate three different output signals corresponding to the three colours of the RGB LED. The two I2C pins are included in the Molex 5-pin male header connector on the RE-Mote, where the PCB will be plugged in with a female 5-pin connector [3].

The PCB includes two pull-up resistors required for the I2C protocol operation, the RGB LED [4], the RGB LED driver [5] and several components required for the driver to work properly:

1. 62k resistor [6]
2. 220nF capacitor [7]
3. 1uF capacitor x2 [8]

All those components listed above are surface-mount devices (SMD) which will be soldered directly onto the surface of the PCB.

The PCB was designed using KiCad, a free software suite for electronic design automation. KiCad features an integrated environment for schematic capture (Eeschema) and PCB layout design (Pcbnew) as well as the tools to generate the Gerber files. Since KiCad was new to me, before starting the PCB design it was necessary to get some experience with the software following the getting started tutorial on the official web page.
3.4 PCB design

3.4.1 Schematic Capture

The first step on the PCB design is to create the schematic file, it was designed using the schematic editor Eeschema which provides wide libraries with electronic components. If components are not included in the KiCad library they can be created using the Library Editor which will be added to a private library for your own use in the project. As seen in the Figure 3.2 the schematic is designed basically following the guidelines in the RGB LED driver datasheet, where the components needed for the driver are specified, as well as the pin map of the device. In the same way, the pin map of the rest of the components can be found in their corresponding datasheets.

![Figure 3.2: Schematic](image)

3.4.2 PCB Layout

The next step is to link a footprint to each component, based on the schematics, in order to create the PCB layout, showed in the Figure 3.3. In the same way as Eeschema, Pcbnew also includes a library with many available footprints and the Footprint Editor can be used to create the missing ones if necessary. In case of creating a footprint, the sizes of the component can be found on its datasheet. An accurate design of the footprint is vital because the size of the components is a few millimetres and the risk of shortcuts when soldering is high. Some other aspects to take into account when designing the PCB layout are
the sizes of the tracks, which must be wider for supply and ground lines than the internal signal tracks, because more current flows on them. It is also strongly recommended to avoid 90 degrees curves since the current can’t circulate correctly through them. It is possible to add multiple layers to the PCB, for this PCB we only use two layers: the bottom layer as a common ground connection and the top layer for the rest of the tracks, pads and footprints. More layers can be added to the PCB, but each layer increases the production cost of the PCB.

![Figure 3.3: PCB layout](image)

Once the PCB layout is done, the last step is to generate the Gerber files and the drill file. The Gerber files give the specification of each layer used in the design including tracks, pads, footprints, written text on the PCB, the edges of the board, etc. The Drill file provides the information about the holes on the PCB such as the size and the position. Both Gerber and Drill files are sent to the manufacturer to produce the PCB.

In the figure 3.4 we can see the four Gerber files superposed one on the other showing the full design of the PCB. One layer is for the green background representing the common ground, connected to the ground pin. In grey colour we can see the pins and the tracks as well as the pads where the components will be placed on. The third layer is the blue line around the board delimiting the edges
to indicate the size of the PCB to the manufacturer. The text labels are the top layer.

Figure 3.4: Gerber files

During the design process we experienced some difficulties due to the lack of experience in this field. After ordering the first revision of the PCB we noticed a fatal design error that could not be fixed. After modifying the design and ordering a second batch, another problem occurred where the pull-up resistors were missing, as well as the ground layer. However, both problems were not critical because it was possible to add the missing components by soldering them on the board and thus solve the problem.

3.4.3 Soldering

Soldering the components on the surface of the PCB is the final step to finish the process. Soldering of SMD components is done putting a solder paste on the pads and then placing the components on them. After this, the PCB is put into an oven and the components get baked. The last step is to fix the problems explained in the previous section. The ground floor is soldered joining the drilled holes with the ground pin. The pull-up resistors are added on the SCL and SDA
pins, connecting those with the power pin and the result is a PCB as shown in the Figure 3.5.

Figure 3.5: PCB
Chapter 4
Software design

In this chapter the implementation of the overall system is explained. The implementation is divided in three sections: the front end, the back end and the nodes.

4.1 Front end

The purpose of the web application is to change the colour level of the RGB LEDs in the network. It is meant to run on a Microsoft Surface Pro tablet. The web site is hosted on the Raspberry Pi running an Apache 2 server and the URL to access the web page is 192.168.1.1, when connected to the access point. The front end is based on the Bootstrap theme SB Admin 2, a template featuring JavaScript and jQuery extensions. The front end includes two web pages, the welcome page and the colour change page, shown in Figure 4.1. The colour changing web page contains an image with the whole colour spectrum from white to black. Below the colour spectrum an input box shows the current RGB colour selected when touching the colour chart and moving the finger over it. This colour is the one displayed for the LEDs forming the lighting network.
4.1 Front end

4.1.1 Colour chart

The colour chart image is a HTML Canvas, an element used in HTML5 to draw graphics via JavaScript. The method used to get the colour when a user touches the screen or moves the mouse is as follows: based on the canvas element, we use the jQuery events, pageX and pageY to get the user’s cursor position inside the canvas in x-y coordinates and then subtracting from that the offset of the canvas. That will give us the exact position where the user touched the screen relative to the position of the canvas. The Javascript function `getImageData` returns the ImageData object corresponding to the pixel in the given coordinate. The indexes 0 to 2 in the ImageData object are the red, green and blue colour level values from 0 to 255 (1 byte) each one. Finally, these values are converted to hexadecimal format and a new line character ("\n") is added after each message, required for the reception on the Re-Mote, before sending them to the server. Thus, based on this format, one possible message could be as following:

```
0055FF\n```

Figure 4.1: Colour chart
To achieve the response to the finger touch in the screen we use the jQuery touch events. Those events are meant to work with mobile devices and are the equivalent to the mousemove events for Desktop applications. In our case we use the `touchstart` and `touchmove` events, which triggers an event to a single touch and to a sliding on the screen. The jQuery touch events are included in the jQuery Mobile library, stored locally on the Apache 2 server running on the Raspberry Pi. By combining both, touch and pageX and pageY events, the position where the image has been touched can be retrieved.

By default, the touch event is triggered every time it detects the finger touch, and in case of sliding the event is executed as fast as the browser is able to do it. This would create too many messages, therefore for the demonstrator an event every 100 ms for gathering data is appropriate. To trigger the event only every 100ms the jQuery time_stamp event is used. This method returns the number of milliseconds since 01/01/1970. The time stamp is saved right after executing the program and it is compared to the current time stamp just before executing it again. Setting the comparison value to ”>100” gives us the needed frequency.

Initially the web application was meant to run in all web browsers, but the implementation of touch events was different for each browser. Because of that, the web application works on Chrome and Firefox browsers only. Also, iOS devices are not supported, yet.

Besides that, the front end connects to the back end server each time the web application is loaded, via this connection the hexadecimal messages are sent.

### 4.2 Back end

In order to handle the communication between the JavaScript application and the serial port of the Raspberry Pi we use a WebSocket server. It plays two roles: listening and receiving messages from the web browser, and send them through the serial port. To realize this Node.js is used.
4.2 Back end

4.2.1 Node.js

Node.js\(^1\) is an open-source, cross-platform runtime environment for developing server-side Web applications. It wraps the Google’s JavaScript engine called V8 and has a library management system called node package manager (npm) that allows you to extend its functionality in many directions. The server in the demonstrator uses the libraries *serialport* and *ws*.

4.2.2 Serial port

To open the serial port in Node.js, the first thing is to make a local instance of the library in a variable. The settings to be defined related to the serial port are the baud rate and the port name. When making the local instance the baud rate is set to 115200. The port name is defined when initializing the server. The **serialport** library is event-based, it means that the user’s actions will generate events and the program will provide callback functions to handle them. The main events that the **serialport** package will handle are when a serial port opens, when it closes, when there is new data to be sent, and when an error occurs. The callback function for each event has to be defined and they will be called when the event occurs. In the case where there is new data to be sent, the `.write()` function is called. This function sends the messages received from the web browser to the serial port and returns the number of bytes written, which is useful while debugging the operation of the server.

4.2.3 WebSocket

WebSockets are connections between web clients and servers, it allows the server to listen for messages from the web browser. Both serial connections and web-Socket connections are data streams, in which the first byte sent on the client is the first byte received on the server. In the same way as the serial port, we need to create a new instance of the webSocket server and configure the parameters such as the server port, which is port 8081, an alternative port used for web traffic. To initialize the server the following command must be executed:

\(^1\)https://nodejs.org/en/
4.3 Nodes

node wsServer.js /dev/ttyUSB0

Where *wsServer.js* is the name of the file containing the server and */dev/ttyUSB0* is the port name.

The server will operate on port 8081. Thus the web application will connect to this port. The browser will be a client of the node.js script and the script will run as a server.

Just as there are serialport event listener functions, there are webSocket event listener functions as well. The functions are: listening for a webSocket connection event, listening for incoming messages and for a webSocket closing event.

There can be multiple webSocket clients at the same time, so the server maintains an array to keep track of all clients. Every time a new client establishes a connection, the client is added to the array using the `.push()` function. When a client closes, it’s removed from the array using the `.splice()` function.

Therefore, when new data arrives from the browser, the webSocket event is triggered and transmits the data to the serialport `.write()` function.

The Figure 4.2 shows an schematic chart of the Node.js script connections.

![Figure 4.2: From the web page to the Re-Mote](image)

4.3 Nodes

The demonstrator includes one master node and several slave nodes. In the operation of the system, one Re-Mote receives the RGB values from the web application and broadcasts those values to the slave nodes, where the RGB LED change its colour level according to the received command. Based on this, the task of the Re-Mote’s is as follows:
4.3 Nodes

1. Master node: it is the light switch of the system: including one Re-Mote, the tablet and the Raspberry Pi. It broadcasts messages to all slave nodes in the network and also controls its own RGB LED.

2. Slave nodes: they are all the other nodes. They receive messages wirelessly from the master node and act as the lamps of the network. In the same way as the master Re-Mote, they are equipped with the RGB LED.

The implementation of the program for the Re-Mote is based on the example code provided by Contiki where they demonstrate the operation of the different features of the Zoul module, including the serial port communication. The serial reception is event-based, it means that each time an entire line of text (ending in "\n") is received over the serial port, an event is triggered and we can receive the data that has been sent. The values from the web application are in hexadecimal format and each colour is 8-bit size, so the master Re-Mote receives 3 bytes of data and 1 byte for the "\n" character: 4 bytes in total for each message. Those 8-bit values are sent to all the slave nodes, the functions for broadcasting and receiving messages wirelessly were already developed in previous works related to this project.

4.3.1 Control of the LED

The RGB LED is controlled using the I2C protocol and, such as the serial communication, Contiki includes a module with functions to handle the I2C communication. The I2C protocol is a protocol intended to allow multiple slave digital devices to communicate with one or more masters for short distance communications. In this case, the master device is the Re-Mote and the slave is the RGB LED driver.

To start the communication between the master and the slave the `i2c_init()` function is called, it defines the port and pin number for the SDA and the SCL as well as the bus speed. The other function used in the program is `i2c_single_send()`, which has two parameters: the slave address and the data. Right after executing those two commands, a delay function must used for the proper behaviour of the program, otherwise the I2C functions don’t have enough time to send the
message. A delay of 50 microseconds is appropriate to solve this issue and it’s executed using the `clock_delay_usec()` function, included in the Clock library provided by Contiki.

According to the datasheet of the RGB LED driver, the address of the device is 0x39 (7 bits), though the `i2c_single_send()` function modifies this value adding a final bit set to 0 representing the R/W mode, since the driver is a receive only driver, so the final address value is 0x72.

With respect to the data parameter, it is a 8-bit value of which the 3 most significant bits belong to the type of operation, including the PWM operation and the gradual dimming, which is not used. This leaves 5 bits for the data itself to control the colour of the LED.

After the `i2c_init()` function and before sending data to the LED, we must give intensity to the LED. It is done using the `i2c_single_send()` function setting the first three bits to 001 and the rest to 00001 for minimal intensity and 11111 for maximal intensity. It tells the LEDs to bright in the specified intensity by the time they receive any data. If this command is not written, the LEDs don’t work at all because the output intensity is set to 0 by default. This caused many problems during implementation, because the data sheet of the RGB LED driver lacked of this information.

The final step is to send the messages received on the serial port to the I2C, it is done using the built-in PWM operation on the driver. The first three bits of data allow us to select each of the three LEDs we want to transmit the message and the following 5 bits specify the colour level of the selected LED. As explained before, the messages received from the web page are 8-bit values, ranging from 0 to 255, and the I2C can only receive 5 bits of data. Because of this, the original values are mapped from 8-bits to 5-bits selecting the 5 most significant bits of the byte. This situation causes a reduction of the number of available colours, from $255 \times 255 \times 255 = 16581375$ to $32 \times 32 \times 32 = 32768$ different colours to display, although it is not a problem considering that it is a range large enough.
The Figure 4.3 shows an example of data received on the Re-Mote from the web socket server. If we focus on the last message, we can see the hexadecimal value received is 013AAB. It is split into three values: 01 is red, 3A is green and AB is blue, but since we only take the five most significant bits of each one, the final hexadecimal value is reduced to 0 for red colour, 7 for green and 15 for the blue, as seen in the picture. Those three values are sent to the RGB LED driver using the \texttt{i2c\_single\_send()} function and broadcasted to the slave nodes.
Chapter 5

Testing and validation

The testing of the demonstrator has been done in stages. Chronologically, the first step was developing the web application that, as previously explained, was successfully validated on Chrome and Firefox browsers on the Microsoft Surface Tablet and also on a Android smartphone. However it was not working on iOS devices such as iPhone and iPad, since the web page don’t recognize the web browser in this platforms.

The next task was the serial communication between the web page and the serial port of the Raspberry Pi, it was tested using a serial terminal such as CuteCom or PuTTY. Those tools are used either to read or write to the serial port, with those tools it was possible to check if the hexadecimal values corresponding to the picked colour were correctly transmitted.

The last step was to test and validate the PCB, as mentioned in previous chapter. The first design error was discovered just after ordering the first batch by checking the design files. After correcting the mistake and ordering the second batch, the first PCB were soldered. With this PCB the first attempt was made to control the LED using sample codes. The first tests were unsuccessful, therefore by using a multimeter connections were checked as well as possible shortcuts on the board. During those checks the missing ground layer was found. Once that problem was fixed the PCB was still not responding although the hardware was fixed now. Next, the sent data via the I2C was checked using the logic analyzer, an electronic instrument that captures and displays multiple signals, and it seemed to be correct even though the LED was still switched off. Finally, while debugging
the code, the missing command to set the brightness was found and after setting
the brightness at the beginning the LED was now responding but still not working
properly and we quickly realized about the missing pull-up resistors.
Once the PCB was checked and validated, we could focus on testing the code in
the Re-Mote since it was not possible until the moment we had a PCB working
correctly. Here the main task was to receive correctly the messages with the three
RGB colours from the serial port, split those messages into three parts and send
each colour to the driver with its corresponding LED address. The testing of this
code is possible thanks to the serial port terminal window provided by Contiki
executing the command make login, where we can see the output of the program.
The general test could be successfully completed using four Re-Motes all of them
equipped with a PCB as seen in the Figure 5.1. In the test we could select the
colour in the colour chart sliding the finger over it and the RGB LEDs equipped
on the Re-Motes changed the colour level according to the one selected.

![Demonstrator](image)

Figure 5.1: Demonstrator
5.1 Tools

Several software tools has been used during the development of the different stages of the project. Recommended from my supervisor, Eclipse was the tool of choice for editing the Contiki software as it has a clear overview of all the files, useful lookup for function details and function calls can be easily searched. The revision control system used was Bitbucket since it was the service already in use for the files related to the DEWI project. The full source code for the Contiki releases was downloaded on the PC locally, it was possible because the PC was running Ubuntu otherwise Contiki offers Instant Contiki to be downloaded in the virtual machine VMWare Player. Instant Contiki offers all the development tools, compilers, and simulators used in Contiki.
Chapter 6

Conclusion and Outlook

6.1 Review of Contribution

As described in the previous chapter, a wireless lighting network with several nodes has been successfully developed and the operation of a lighting use case was demonstrated.

My contribution to this project is the design and creation of the web application, the WebSocket server handling the serial communication with the Re-Mote, the design of the PCB and the development of the software in the Re-Mote to control the RGB LED using the I2C protocol.

6.2 Lessons Learned

After a few complicated days regarding the Erasmus documentation and the difficulties with the language, I quickly felt comfortable with the working environment at the Nimbus office. During the internship I had a lot of freedom in how to face the issues encountered during the implementation, this was sometime frustrating if you got stuck at some part but in general made it much more challenging and useful. It took me some time to get used to the different areas of work since the project ranged from web design to hardware developing, and it often required a getting starting tutorial to get into it. The repeated mistakes with the PCB design and the later ”magic” command to switch on the LED were annoying at
6.3 Outlook and future work

The project went well and the final goal has been achieved. Further work needs to be done still in relation to this work such as improving the web application and make it suitable for other browsers and platforms.
References


Appendices
Appendix A

Web page

A.1 Welcome page

```html
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1">
<meta name="description" content="">
<meta name="author" content="">
<title>Wireless Lighting Demonstrator</title>

<!-- Bootstrap Core CSS -->
<link href="../bower_components/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet">

<!-- MetisMenu CSS -->
<link href="../bower_components/metisMenu/dist/metisMenu.min.css" rel="stylesheet">

<!-- Timeline CSS -->
<link href="/dist/css/timeline.css" rel="stylesheet">
```
A.1 Welcome page

<!-- navbart-header -->
<div class="navbar-default sidebar" role="navigation">
    <div class="navbar-collapse sidebar-collapse">
        <ul class="nav side-menu">
            <li class="sidebar-search">
                <div class="input-group custom-search-form">
                    <input type="text" class="form-control" placeholder="Search...">
                    <span class="input-group-btn">
                        <button class="btn btn-default" type="button">
                            <i class="fa fa-search"></i>
                        </button>
                    </span>
                </div>
            </li>
            <li><a href="index.html">Home</a></li>
            <li><a href="color.html">RGB LED Colour chart</a></li>
        </ul>
    </div>
</div>

<!-- /.navbar-collapse -->
</div>
<!-- /.navbar-static-side -->
</nav>

<div id="page-wrapper">
    <div class="row">
        <div class="col-lg-12">
            <h1 class="page-header">Welcome</h1>
        </div>
    </div>
</div>
Introduction

Demonstrator for a wireless lighting control use case

<script src="../bower_components/jquery/dist/jquery.min.js"></script>

<script src="../bower_components/bootstrap/dist/js/bootstrap.min.js"></script>

<script src="../bower_components/metisMenu/dist/metisMenu.min.js"></script>

<script src="../dist/js/sb-admin-2.js"></script>
A.2 Colour chart

```html
<!DOCTYPE html>
<html lang="en">
<head>
<style>
</style>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=no">
<meta name="description" content="">
<meta name="author" content="">
<title>Wireless Lighting Demonstrator</title>
</head>
</html>
```

A.2 Colour chart

34
<!-- jQuery --><script src="../bower_components/jquery/dist/jquery.min.js"></script>

<!-- Bootstrap Core JavaScript -->
<script src="../bower_components/bootstrap/dist/js/bootstrap.min.js"></script>

<!-- Metis Menu Plugin JavaScript -->
<script src="../bower_components/metisMenu/dist/metisMenu.min.js"></script>

<!-- Custom Theme JavaScript -->
<script src="../dist/js/sb-admin-2.js"></script>

<!-- Own JavaScript libraries -->
<script src="../bower_components/jquery/dist/jquery.ui.touch-punch.min.js"></script>
<script src="../bower_components/jquery/dist/jquery.mobile.custom.js"></script>
<script src="../bower_components/jquery/dist/jquery.mobile.custom.min.js"></script>

<!-- HTML5 Shim and Respond.js IE8 support of HTML5 elements and media queries -->
<!-- WARNING: Respond.js doesn't work if you view the page via file:// -->
<!--[if lt IE 9]>
<script src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></script>
<script src="https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js"></script>
<![endif]-->
A.2 Colour chart
<div id="page-wraper">
<div class="row">
<div class="col-lg-12">
<h1 class="page-header">RGB LED colour chart</h1>
</div>
</div>
<canvas width="600" height="388" id="canvas_picker"></canvas>
<div id="rgb">RGB: <input id="rgb_input" type="text"></div>

<script type="text/javascript">
var canvas = document.getElementById('canvas_picker');
var ctx = canvas.getContext('2d');

var img = new Image();
img.src = '../img/colors.jpg';

$(img).load(function(){
ctx.drawImage(img,0,0);
});

var lasttimestamp = 0;

function rgbToHex(R,G,B) {return toHex(R)+toHex(G)+toHex(B)}

function toHex(n) {
  n = parseInt(n,10);
  if (isNaN(n)) return "00";
  n = Math.max(0,Math.min(n,255));return "0123456789ABCDEF".charAt((n-n%16)/16) + "0123456789ABCDEF".charAt(n%16);
}

var socket = new WebSocket("ws://192.168.1.1:8081");

if(navigator.userAgent.indexOf("Chrome") != -1){


</script>
$("#canvas_picker").on('touchstart touchmove', function(event) {
    event.preventDefault();
    var x = event.originalEvent.targetTouches[0].pageX - this.offsetLeft;
    var y = event.originalEvent.targetTouches[0].pageY - this.offsetTop;
    var R, G, B, rgb, img_data;
    var hex;
    img_data = ctx.getImageData(x, y, canvas.width, canvas.height).data;
    if(event.timeStamp - lasttimestamp > 100) {
        lasttimestamp = event.timeStamp;
        R = img_data[0];
        G = img_data[1];
        B = img_data[2];
        rgb = R + ', ' + G + ', ' + B;
        hex = rgbToHex(R,G,B);
        $('#rgb_input').val(rgb);
        socket.send(hex + '\n');
    }
    event.preventDefault();
});
else if(navigator.userAgent.indexOf("Firefox") != -1) {
    $("#canvas_picker").on('mousemove', function(event) {
        event.preventDefault();
        var x = event.pageX - this.offsetLeft;
        var y = event.pageY - this.offsetTop;
        var R, G, B, rgb, img_data;
        var hex;
        img_data = ctx.getImageData(x, y, canvas.width, canvas.height).data;
        if(event.timeStamp - lasttimestamp > 100) {
            lasttimestamp = event.timeStamp;
            R = img_data[0];
            G = img_data[1];
            B = img_data[2];
            rgb = R + ', ' + G + ', ' + B;
            hex = rgbToHex(R,G,B);
            $("#rgb_input").val(rgb);
        }
    });
}
socket.send(hex + "\n");
    
    event.preventDefault();
    
});
}
else{
    console.log("ERROR: Browse in FF or Chrome");
}

/*WEBSOCKET SERVER*/

function setup() {
    socket.onopen = openSocket;
}

// Change text to push up message or similar
function openSocket() {
    socket.send("Hello server");
}

</script>

</body>
</div>
<!-- /.row -->
</div>
<!-- /# page-wrapper -->
</div>
<!-- /# wrapper -->
</body>
</html>
Appendix B

WebSocket Server

/*
Serial-to-websocket Server
using serialport.js
To call this type the following on the command line:
node wsServer.js portName
where portname is the name of your serial port, e.g. /dev/ttyUSB0
*/

// include the various libraries that you’ll use:
var serialport = require('serialport'); // include the
    serialport library
var WebSocketServer = require('ws').Server; // include the
    webSocket library

// configure the webSocket server:
var SERVER_PORT = 8081; // port number for the
    webSocket server
var wss = new WebSocketServer({port: SERVER_PORT}); // the
    webSocket server
var connections = new Array; // list of connections
    to the server

// configure the serial port:
SerialPort = serialport.SerialPort, // make a local
    instance of serialport
portName = process.argv[2]; // get serial port
    name from the command line
// If the user didn’t give a serial port name, exit the program:
if (typeof portName === "undefined") {
  console.log("You need to specify the serial port when you
   launch this script, like so:\n")
  console.log("    node wsServer.js <portname>");
  console.log("\nFill in the name of your serial port in place
   of <portname>\n");
  process.exit(1);
}

// open the serial port:
var myPort = new SerialPort(portName,
  {
    baudRate: 115200,
    parser: serialport.parsers.readline("\r\n")
  });

// set up event listeners for the serial events:
myPort.on('open', function(){
  console.log('Port opened. Data rate: ' + myPort.options.
    baudRate);
});
myPort.on('data', sendSerialData);
myPort.on('close', showPortClose);
myPort.on('error', showError);

// ------------------------ Serial event functions:
// this is called when the serial port is opened:
function showPortOpen() {
  console.log('Port opened. Data rate: ' + myPort.options.
    baudRate);
}

// this is called when new data comes into the serial port:
function sendSerialData(data) {
  // if there are webSocket connections, send the serial data
  // to all of them:
  console.log(data);
  if (connections.length > 0) {
    broadcast(data);
  }
}
function showPortClose() {
    console.log('port closed. ');
}

// this is called when the serial port has an error:
function showError(error) {
    console.log('Serial port error: ' + error);
}

function sendToSerial(data) {
    console.log(data);
    myPort.write(data, function(err, bytesWritten) {
        if (err) {
            return console.log('Error: ', err.message);
        }
        console.log(bytesWritten, 'bytes written');
    });
}

// ------------------------ webSocket Server event functions
wss.on('connection', handleConnection);

function handleConnection(client) {
    console.log("New Connection"); // you have a new client
    connections.push(client); // add this client to
    the connections array
    client.on('message', sendToSerial); // when a client sends a message,
    client.on('close', function() { // when a client closes its connection
        console.log("connection closed"); // print it out
        var position = connections.indexOf(client); // get the
        client's position in the array
        connections.splice(position, 1); // and delete it
        from the array
    });
}

// This function broadcasts messages to all webSocket clients
function broadcast(data) {
for (c in connections) { // iterate over the array of connections
    connections[c].send(JSON.stringify(data)); // send the data to each connection
}
Appendix C

Software running on the nodes

```c
#include "contiki.h"
#include "cpu.h"
#include "dev/leds.h"
#include "dev/uart.h"
#include "dev/adc-zoul.h"
#include "dev/watchdog.h"
#include "dev/serial-line.h"
#include "net/rime/broadcast.h"
#include "../cpu/cc2538/dev/i2c.h"
#include "../platform/zoul/remote/board.h"
#include "../examples/zolertia/zoul/project-conf.h"

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <inttypes.h>

#define LEDS_PERIODIC LEDS_BLUE
#define LEDS_BUTTON LEDS_RED
#define LEDS SERIAL_IN LEDS GREEN
#define LEDS REBOOT LEDS ALL

#define BYTE TO BINARY PATTERN "%d%d%d%d%d"
#define BYTE TO BINARY (byte) \n  (byte & 0x80 ? 1 : 0), \n  (byte & 0x40 ? 1 : 0), \n```
(byte & 0x20 ? 1 : 0), \n(byte & 0x10 ? 1 : 0), \n(byte & 0x08 ? 1 : 0), \n(byte & 0x04 ? 1 : 0), \n(byte & 0x02 ? 1 : 0), \n(byte & 0x01 ? 1 : 0)

#define LED_RED 0b10000000
#define LED_GREEN 0b01100000
#define LED_BLUE 0b01000000

PROCESS(test_serial, "Zoul test serial");
AUTOSTART_PROCESSES(&test_serial);

static void
broadcast_recv(struct broadcast_conn *c, const linkaddr_t *from)
{
    uint32_t R, G, B;
    uint32_t temp = *(uint32_t *) packetbuf_dataptr();
    printf("*** Received %u bytes from %u:%u: '0x%08x'\n",
        packetbuf_datalen(),
        from->u8[0], from->u8[1], temp);
    R = (int)(temp & 0x00ff0000) >> 19;
    G = (int)(temp & 0x0000ff00) >> 11;
    B = (int)(temp & 0x000000ff) >> 3;
    printf("R %x\n",R);
    printf("G %x\n",G);
    printf("B %x\n",B);
    i2c_single_send(0x39, LED_RED | R);
    clock_delay_usec(50);
    i2c_single_send(0x39, LED_GREEN | G);
    clock_delay_usec(50);
    i2c_single_send(0x39, LED_BLUE | B);
    clock_delay_usec(50);
}

static const struct broadcast_callbacks bc_rx = { broadcast_recv 
};

static struct broadcast_conn bc;

PROCESS_THREAD(test_serial, ev, data) 
{
PROCESS_BEGIN();
serial_line_init();
broadcast_open(&bc, BROADCAST_CHANNEL, &bc_rx);
i2c_init(I2C_SDA_PORT, I2C_SDA_PIN, I2C_SCL_PORT, I2C_SCL_PIN,
    I2C_SCL_FAST_BUS_SPEED);
char *ch_data, *ptr;
long i_data;
uint32_t R, G, B;
i2c_single_send(0x39, 0b00111111); // LED output on, maximum
    power
    clock_delay_usec(50); // Delay after every i2c command

while (1) {
    PROCESS_YIELD();
    if(ev == serial_line_event_message) {
        ch_data = (char*) data;
        printf("ch_data %s\n", ch_data);
        i_data = strtol(ch_data, &ptr, 16); // Convert char data to
            hex

        R = (int)(i_data & 0x00ff0000) >> 19; // Select byte 1 and
            shift to first position
        G = (int)(i_data & 0x0000ff00) >> 11; // Same for byte 2
        B = (int)(i_data & 0x000000ff) >> 3; // And byte 3
        printf("R %x\n",R);
        printf("G %x\n",G);
        printf("B %x\n",B);
i2c_single_send(0x39, LED_RED | R);
clock_delay_usec(50);
i2c_single_send(0x39, LED_GREEN | G);
clock_delay_usec(50);
i2c_single_send(0x39, LED_BLUE | B);
clock_delay_usec(50);

clock_delay_usec(1000000);
packetbuf_copyfrom(&i_data, sizeof(i_data));
broadcast_send(&bc);
}
} PROCESS_END();
}