MSc Project 2013

UNIVERSITY OF SOUTHAMPTON

Faculty of Engineering, Science and Mathematics

Web-enabled HPLED control system

A MSc Project report submitted as part of the European Masters on Embedded Computing Systems (EMECS)

Author:
Konke Radlow

Supervisors:
Dr. Peter Reid Wilson
Dr. Alex Weddell

July 26, 2013

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Project Description

The goal of this project is to design a Web-enabled system for driving and controlling high-power LEDs (HPLEDS). Over the course of this project a distributed embedded system, consisting of controller boards for HPLEDs, a coordinator for multiple controller boards, a dynamic web-API and a responsive user interface, was designed and implemented.

The use of remote controlled HPLEDs with intelligent controllers allows us to rethink the way we use and interact with artificial light. Completely new applications become possible due to the special characteristics of LEDs.

- Luminous intensity can be adopted locally and automatically using luminous sensors to save energy by incorporating sunlight into the light concept but eliminating its variable nature [Sawhn(2010)].
- By combining the controller capabilities with indoor positioning systems, for instance, it is possible to define an illuminated area that follows people around while they move through a building.
- The colour temperature of the light can be adapted to the time of day and situation to improve well-being and productivity [Li(2013)].
- In home automation installations it becomes possible to adapt the color of the light to specific situations e.g. using it like a large scale Ambilight[1] while watching television or visualizing sound by pulsating the colors to a rhythm while listening to music.

The possibilities are nearly endless and highly promising in terms of energy saving, user-friendliness and creative new light applications.

[1] Ambient lightning system for televisions by Philips

1.1.1 Project Components

The HPLED-controller boards facilitate the use of multiple (RGB-)HPLEDs in embedded projects. The design of the board enables integration of controller boards into different project environments. It is interfaced with a bus type commonly found in embedded systems (I^2C) and includes its own power supply circuitry for LEDs and on-board peripherals.

The coordinator is a Raspberry Pi based hardware-software solution that acts as a bridge between embedded systems (using cable based buses) and the Web. It offers a dynamic Web-API that reflects the state of the connected systems and allows direct control of the hardware over the Internet.

The Web-API uses an architectural style (REST) for distributed systems and can be accessed by Web-applications as well as custom applications on portable computing devices.

The user interface (UI) is based on modern Web standards and can be used from a wide range of platforms (Linux, Windows, Android, iOS, ...). It offers direct control over the output of the controller boards and is designed to feel like a touch-optimized application rather than a website.

1.2 Motivation

This project combines two scientific fields that have seen a huge increase in technological progress and economic growth over the last years: LED technology and the *Internet of Things* [Ashton(1999)].

Advancements in LED technology are increasingly challenging established light sources in various fields. Huge improvements in their luminous efficacy in combination with a long list of advantages over fluorescent light sources are making them a serious competitor and there is no foreseeable end to the technological development of LEDs (see sec. 1.2.1).

The Internet of Things is a trend that is driven by the pervasiveness of our surroundings by wireless networks and the desire to make physical devices uniquely identifiable; as well as providing them with a uniform interface that enables autonomous communication between heterogeneous devices [Hersent and Elloumi(2012)].

Applying the ideas of the Internet of Things on this project is a way to experiment with different new approaches in this field where research is still in its infancy. It is an emerging technology that will become highly relevant for the design of embedded systems in the near future [Chui et al.(2010)Chui, Löffler, and Michael] (see sec. 1.2.2).

1.2.1 LED Technology

For several years the technologies used to create artificial light have largely remained unchanged. Both of the main technologies, incandescent lights and fluorescent lights, have a set of inherent drawbacks:

Incandescent lights suffer from their low luminous efficacy and convert only 2% of the power input to visible light. Fluorescent lights on the other hand have a higher efficacy (about 22%) but they have other disadvantages like environmental issues due to toxic substances, the lack of dimming capability and flicker problems. Large improvements in these fields are not to be expected since their technological development has come close to a standstill.

But there is a new technology that has gained momentum over the last decade: the Light-emitting Diode (LED). Although the first LED was created in 1927 [Losov(1927)] it was not until the early sixties until LEDs became commercially available. Since then the development of the LED technology has led to an exponential increase in the efficiency and light output of LEDs (doubling approximately every 36 months, a phenomenon known as Haitz's Law [Bergh et al.(2001)Bergh, Craford, Duggal, and Haitz]).

Around the year 2002 LEDs surpassed the luminous efficacy of conventional light bulbs ($18 \ldots 22 lmW^{-1}$ for Lumileds 5W LEDs vs. $15 lmW^{-1}$ for light bulbs) and since 2012 white LEDs have become available that surpass the efficiency of fluorescent lights ($200 lmW^{-1}$ for Cree MK-R LEDs vs $100 lmW^{-1}$ for fluorescent lights) [Cree(2012)].

Besides their efficacy LEDs offer a number of other advantages such as a long lifetime ($\gg 10000 hrs$), directional light, absence of heat and UV in the light beam and dimming capability without colour shift, that make them a serious competitor to fluorescent lights [Steele(2007)].

What is missing is a cheap and simple way to drive a larger number of these LEDs and control them remotely, to facilitate new applications for this technology.

1.2.2 Internet of Things

With the ever-increasing spread of smartphones and the universal availability of the Internet, we are becoming used to being able to interact with devices using software applications instead of physical contact.

This development is part of a trend called the Internet of Things which is rapidly becoming very relevant for the design of new embedded systems. We are still in the early stages of this development, which means that there are no well-established solutions to the problem of connecting embedded systems to the web. This makes it very interesting and challenging to design the system with the Internet of Things in mind.

This project aims to find a way to integrate (even small and simple) embedded systems into this interconnected world, so that they can be accessed and controlled through any Internet-capable device.

1.3 Existing Solutions

In the year 2012, the two first Internet-connected multicolour LED home illumination systems were announced and have since generated great customer interest:

- LIFX: ”the light bulb reinvented”
- Philips Hue: ”personal wireless lightning”

These systems share many characteristics: they are Web-enabled RGB LED lighting systems that allow the user to control power, brightness, and colour of lightning from mobile devices. They are aimed at replacing light bulbs and can be screwed into standard light sockets.

The IEEE 802.15.4 network standard is used to form a mesh network among all LED devices. Therefore a network coordinator that acts as a gateway between the internal network and the local Wi-Fi is required.

The system that is being developed in this project aims to be more flexible in terms of applicability. It offers an API that allows the user to create custom control programs, and the system design makes it possible to use it as valid alternative for light bulbs but does not limit its scope to this application alone.

Chapter 2

System Design

2.1 Design Overview

The system was designed with usability, cost and scalability in mind. For this reason it was split into a HPLED-Controller that offers a bus interface commonly used in embedded systems and a coordinator that can connect to multiple HPLED-Controller boards..

The coordinator offers a Web-API that gives access to all controllers connected to the system and a Web-based user interface building upon the Web-API. The deployment diagram 2.1 gives an overview over the system design.

The division of the system into several modules retains a low complexity of the HPLED-Controller boards. It also leads to an improved usability because the board is an independent functional unit with a well-defined interface.

The coordinator is a more powerful platform that is able to run an operating system for embedded devices like Linux. The only hardware requirements for this platform are an exposed I^2C bus and Ethernet or Wi-Fi support. It uses the I^2C bus to connect up to 127 controller boards and exposes their current status over a web-API.

By default the system is controlled over a Web-based UI built on recent web-standards (JavaScript and HTML5). The decision to use a Web-based UI was made with accessibility in mind. There are many different platforms (PC, Android, iOS, Windows mobile) that need to be supported in order for the system to be accessed by a wide range of users. It is difficult to write a program that is compatible with all the different platforms but there is one common software available on all of them: a browser.

2.2 System Modules

2.2.1 HPLED-Controller

The HPLED-Controller is implemented on a custom PCB with an 8-Bit microcontroller that enables the board to act as an I^2C slave device. It contains 2 on-board HPLED

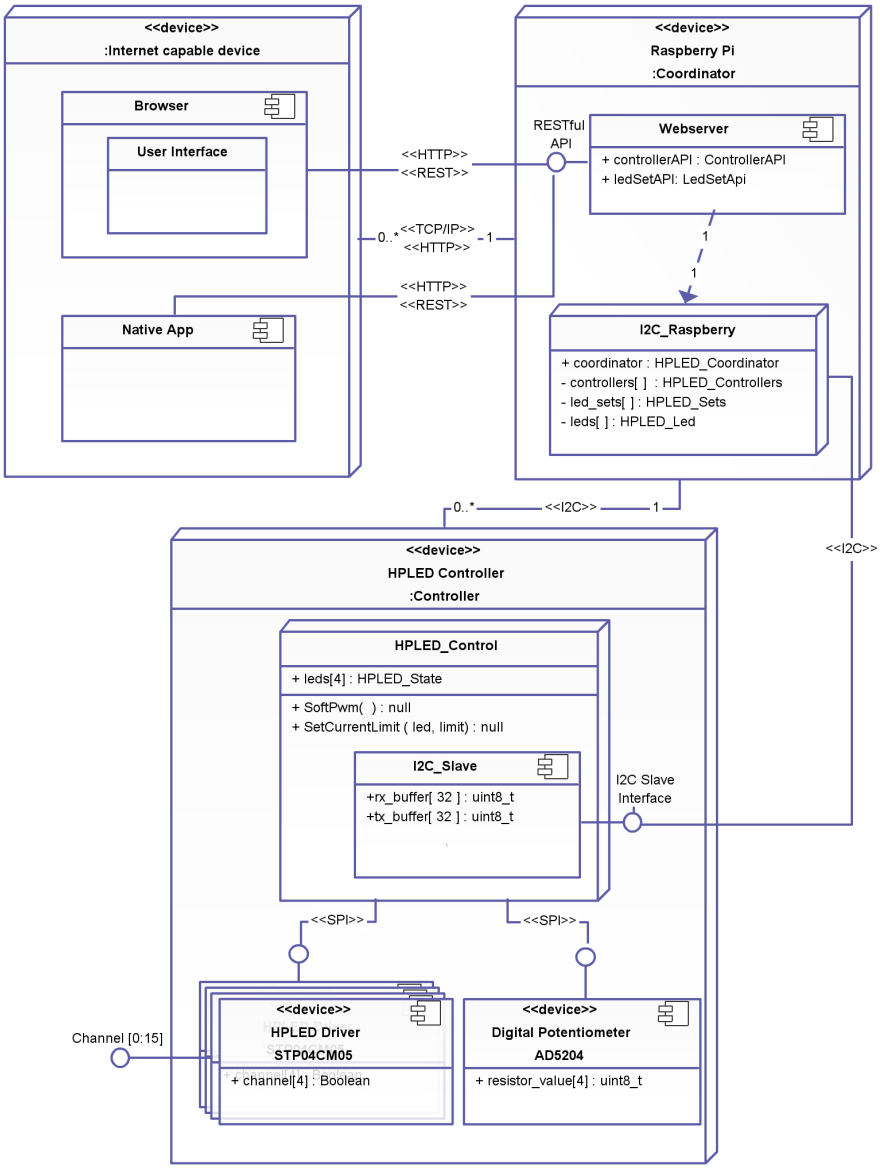


Figure 2.1: Deployment diagram: System design overview

power supply channels, 4 LED driver ICs and a digital resistor IC to control the current output of the LED drivers. The board offers 4 RGB-LED channels, consisting of 4 colour values each, that can also be used to control up to 16 HPLEDs individually. A fully-equipped board can be seen in figure 2.2.

The microcontroller is responsible for controlling the on-board power supplies and driving the ICs according to the instructions received over I^2C. A detailed description of the HPLED-Controller hardware can be found in section 3.1.

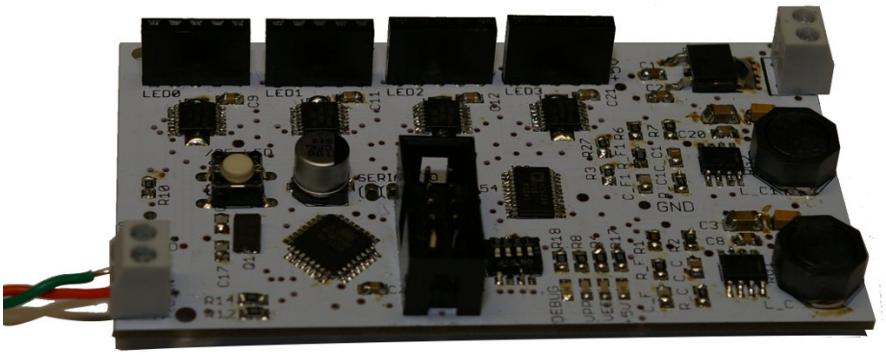


Figure 2.2: HPLED-Controller on custom PCB

It is a true slave device in the sense that it does not implement behaviour of its own, apart from translating the I^2C commands into the actions that need to be performed to set the driver output to the desired state.

The on-board power supply consists of 2 switched-mode power supply circuits that can deliver up to 20W of output power to the LEDs and a 5V power supply for the on-board peripherals.

2.2.2 Coordinator

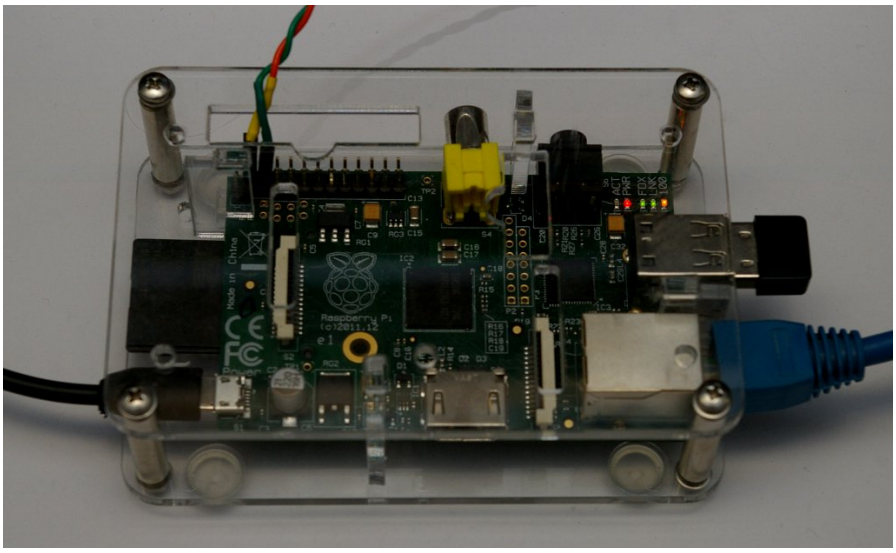


Figure 2.3: Raspberry Pi Coordinator

The coordinator is realized on an existing hardware platform that offers enough processing power to run a Linux operating system while providing an I^2C bus interface. In this

project a Raspberry Pi is used, but it can easily be replaced by similar platforms like the BeagleBoard, as the functionality of the coordinator is software based and not closely tied to specific hardware.

The coordinator has two important functions. First of all, it interacts directly with the hardware and generates an object-oriented modifiable representation of the state of all connected HPLED-Controllers.

Secondly, it provides access to this representation over a RESTful API that can be used to modify the state of the HPLED-Controllers over the Internet. This API can either be used explicitly from custom applications written to provide special functionality, or it can be used implicitly when accessing the coordinator over the web-interface that is provided by a webserver running on the device

2.2.3 User Interface

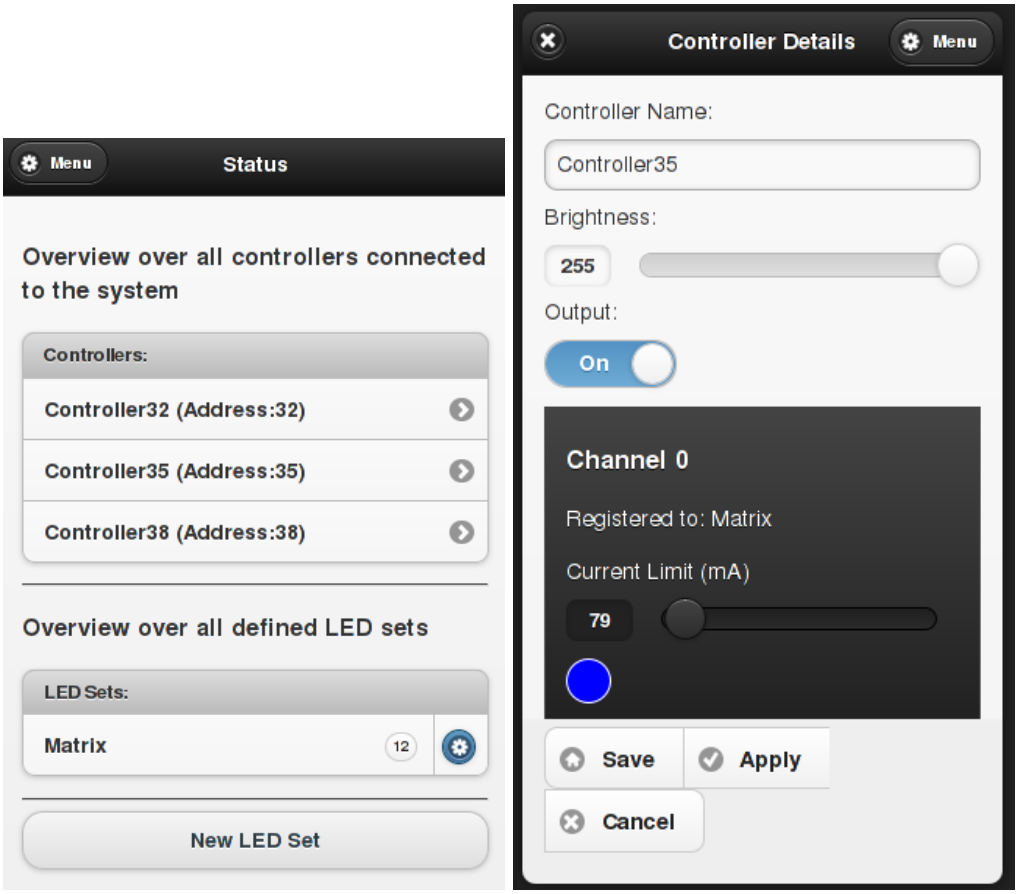


Figure 2.4: User Interface

The system offers a cross-platform user-interface that can be used to control the system in a responsive way without the need to create custom control applications

It is a dynamic jQuery mobile and JavaScript based, touch optimized user interface, that communicates asynchronously with the RESTful Web-API of the coordinator. The content of the user interface is updated dynamically in order to provide the user with a feeling of using an application instead of a website. Figure 2.4 shows some of the provided features.

2.3 Design Reasoning

2.3.1 Communication Interface Options

A number of different system designs were considered before settling on the current system architecture. The most fundamental question for the overall system architecture is "*how to communicate with the HPLED-Controllers?*"

Why using a coordinator, instead of on-board Wi-Fi?

Since the goal of this project is to build a remote controlled HPLED-Controller, the obvious solution would be to include a Wi-Fi module on the driver itself. In this case no coordinator is needed and the system could be controlled wirelessly. A Wi-Fi interface can be integrated into an embedded system in one of two ways:

1. Using a complete Wi-Fi module that offers a simple interface: the problem with this approach is that Wi-Fi modules for embedded systems are still expensive ($\geq$ 30 $[1])
2. Using a cheap USB-Wi-Fi dongle: the problem with this approach is that these cheap dongles do not perform internal decoding of the incoming TCP packets but just pass them on, leaving the processing to the host. This means that the host microcontroller needs to be fast enough to decode TCP streams in software

There are too many drawbacks to opt for either one of them: including a Wi-Fi module on each controller would make the controllers too expensive and using an USB Wi-Fi dongle would increase the software complexity considerably and require the utilization of a 32-Bit processor, with enough processing power to decode TCP/IP streams.

Why use a cable- instead of a wireless connection?

Since the system is built around a coordinator, there is a freedom of choice when it comes to the interface between HPLED-Controllers and the coordinator because the coordinator can function as a bridge between arbitrary interfaces and the Web.

[1] ZG2100MC: WiFi Module with SPI interface

A wireless interface was considered and research was done in this direction. The most promising protocol for applications like this is the 802.14.5 standard, which is designed for low-rate wireless personal area networks. There are two main ways of building the system around a wireless interface:

1. Use a complete 802.14.5 module that offers a simple interface: the problem with this approach is the same as with the Wi-Fi modules. The price for integrated modules (e.g. ZigBee compatible modules) is too high in relation to the cost of one controller ($\geq$ 20\$[2])
2. Use a microcontroller with built in 802.15.4 support: there are microcontrollers available that include the radio hardware and provide a library for accessing the functionality (e.g Atmel ATMEga1284RF, Freescale MC13224V) however these:
   a) Are only available in package options that are very hard to solder with the available equipment (VQFN-64 for ATMega1284RF, LGA-145 for MC13224V)
   b) Have a small user base and little library support is available for the hardware subsystems
   c) Cost about three times as much as a matchable non-wireless microcontroller (e.g ATMega1284RF: 9.96\$, MC13224V: 9.21\$)

With these reasons in mind as well as the time constraints on the project, it was decided that using well-known and reliable bus interface between the controllers and the coordinator would be best.

Since the Serial port of the microcontroller is not used by the system, there is the theoretical possibility to extend the system using a wireless module with a serial interface without the need to re-design the PCB.

2.3.2 User Interface

The decision to use a web-based UI instead of providing native applications for different platforms was motivated by a number of reasons:

1. New technologies and software frameworks like jQuery mobile and AppFramework are available that allow us to create web-applications that feel like native applications and are optimized for touch input
2. The browser is becoming the platform of the future which is available on a wide range of systems and enables true cross-platform applications [Binstock(2012)]
3. Offering a web interface does not require the user to install any software to their device before they can use the system

[2] Xbee Series 2 with serial interface

4. The update process is very simple because the user will be delivered the most recent software version each time he opens the web-application
5. It is a good showcase to demonstrate the features and capabilities of the system and the RESTful-API

The biggest drawback of using a web-based UI is, that it cannot be run as a service in the background on the device of a user. To enable the home automation scenarios mentioned in the introduction, there is still the need to write native-applications.

However the design of the system allows these applications to be written if and when it should be required.

On the other hand giving the user manual control over the system by means of a sophisticated cross-platform compatible UI is worth implementing alone.

2.3.3 Focus On RGB LEDs

High-power LEDs exist in a wide range of colours, but the most widely used option at the moment are white LEDs. They are available in the most interesting module configurations for using them for home illumination, and they have the most competitive price level.

While the presented system can be used to control the brightness of single coloured LEDs remotely, the system loses most of its advantages over existing solutions and suffers in consumer appeal.

Recent product developments like the Hue system or the LIFX lightbulb, show that there is a large customer interest in the field of multicolour home illumination. It also proves that big companies invest into research and development of RGB LEDs which will result in modules that can compete with white LED modules in terms of cost and luminous flux.

For this reason, the interface of the board is currently focused on controlling RGB-LEDs. Section 6.2.3 describes the modifications that are planned for the next revision of the software that would make the Web-API more flexible by providing the means to combine arbitrary output channels into one LED object.

Chapter 3

System Hardware

3.1 HPLED-Controller

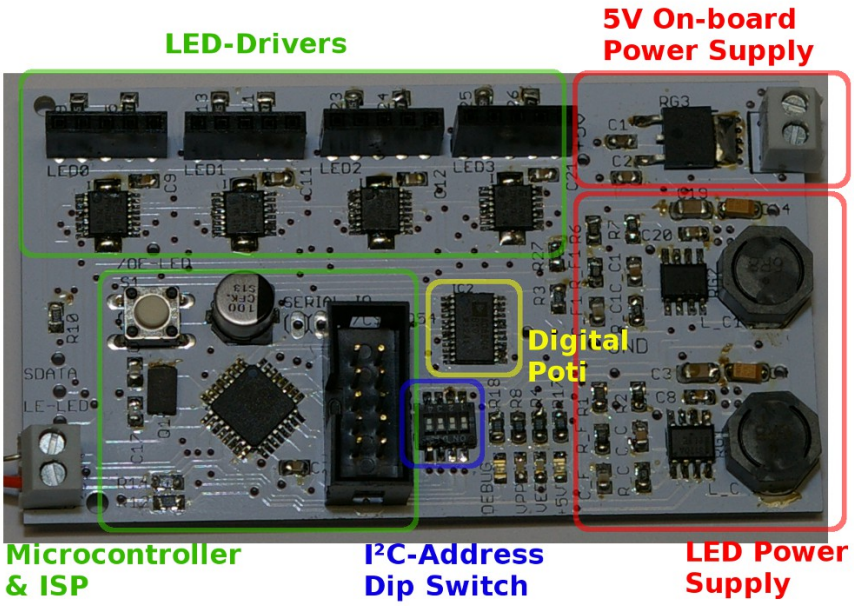


Figure 3.1: HPLED-Controller PCB layout

The HPLED controller board is based on a custom PCB design. It contains an 8-Bit microcontroller, 4 HPLED drivers, a 4-channel digital potentiometer and power supplies for on-board peripherals and HPLEDs. The complete schematics can be found in the Appendix A.3.

The microcontroller provides the external interface of the board and computes the input signal for the HPLED drivers and the digital potentiometer based on data received over the external interface (see 3.1.1).

The HPLED drivers are controlled over an internal 4-bit shift register. By cascading 4 drivers in series, a total of 16 output channels become available. To control each of these channels the microcontroller computes 16 soft-PWM channels that are shifted out serially using the SPI bus.

This means each of the 16 LED channels has its own PWM value which can be defined dynamically over the I^2C bus. This PWM value can be used to vary the brightness of each channel with a resolution of 6-bit (see 3.1.2).

The digital potentiometer is used to set the output current of the HPLED drivers. It is possible to define the output for each of the HPLED drivers independently within a range of $50mA..400mA$. It shares the SPI bus with the HPLED drivers (see 3.1.4).

The board includes two high-efficiency power supplies for connected HPLEDs. To be able to support the high currents required to drive HPLEDs, switched-mode power supplies are used. The can deliver up to 20W of output at full power (see 3.1.5).

As the board is designed to be easily integrated into other projects, the microcontroller is used to expose an I^2C interface that acts as a slave device on the bus. To support configurations with more than one controller board, the I^2C slave address can be changed manually using a DIP-Switch without the need to reprogram the board (see 3.1.3).

3.1.1 Microcontroller

An ATMega328p microcontroller is used as the brain of the board. This microcontroller family has come to fame because it is used by the Arduino platform, which makes it possible to benefit from the Arduino libraries (see section 4.2.4).

Figure 3.2 shows that almost all GPIO pins are used by the design. A complete list of the GPIO pin-mapping can be found in the appendix A.2.

Since a SMD (32TQFP) package has been used for the microcontroller, it needs to be programmable on board. The ISP interface of the AtMega is exposed over a 2x5 pin header that allows programming the device using compatible ISP programmers.

An external 16MHz crystal is used as a clock source to raise the accurateness of the system clock which improves the bus timing and makes it possible to increase the operating speed of the microcontroller to the upper limit.

3.1.2 Driving HPLEDs

The board uses 4 STP04CM05 Power-LED sink drivers by STMicroelectronics . These ICs have 4 constant current output channels which can be driven with up to 400mA. They are controlled with a 4-bit serial-in, parallel-out shift register which is used to control the output of the drivers by outputting a 16bit soft-PWM signal serially [STMicroelectronics(2010)] (see section 4.2.3).

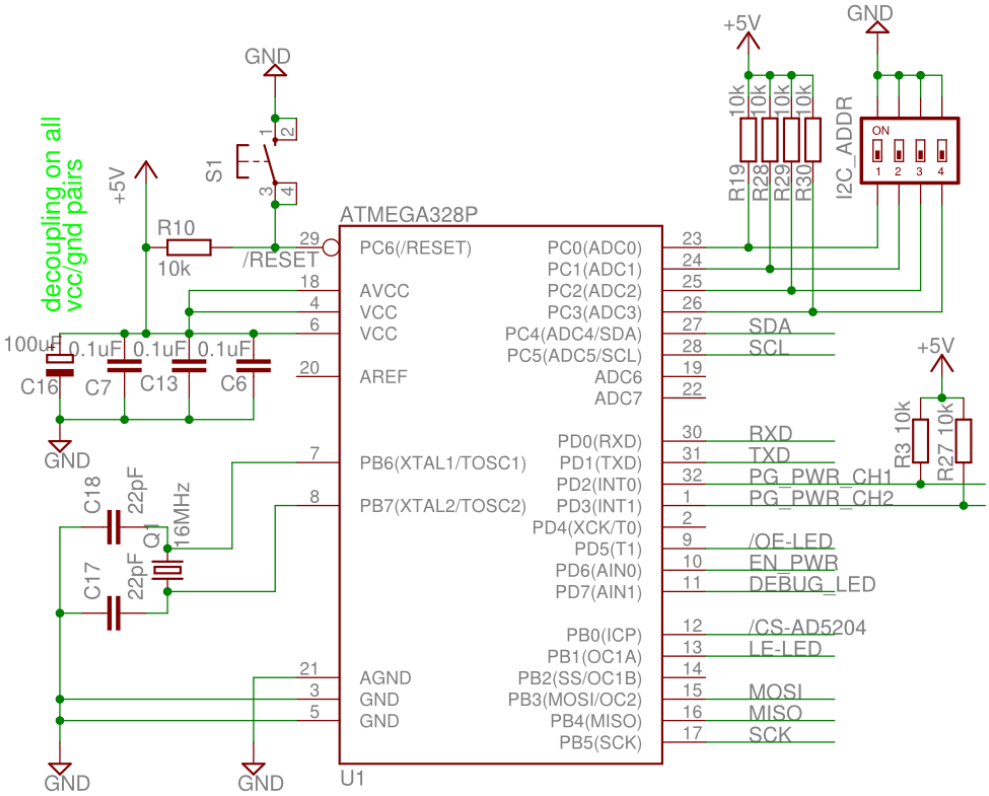


Figure 3.2: Microcontroller, AtMega328p

The controllers are daisy chained over their Serial-In Serial-Out lines so that they can be controlled like a 16-bit shift register.

Power Dissipation

The STP04CM05 drives LEDs in linear mode, which means that the LED current is constant and the power dissipation of the IC depends on the voltage on each sink channel and the driving current. This voltage is the difference between the LED supply voltage and the LED forward voltage [STMicroelectronics(2008), AN2531].

To make sure that the power dissipation of each driver stays below the power dissipation limit P_dmax, the LED supply voltage should be kept as close (but slightly above) the maximum LED forward voltage. The output voltage of the board can be changed within a certain range by changing a resistor value (see 3.1.5).

$$P_{d_{max}} = \frac{T_{j_{max}} - T_a}{R_{thja}} \tag{3.1}$$

With $R_{thja} = 37.5 \circ C/W$ and $T_{j_{max}} = 125 \circ C$ the maximal power dissipation at room temperature ($T_a = 25 \circ C$) amounts to $P_{d_{max}} = 2.7W$.

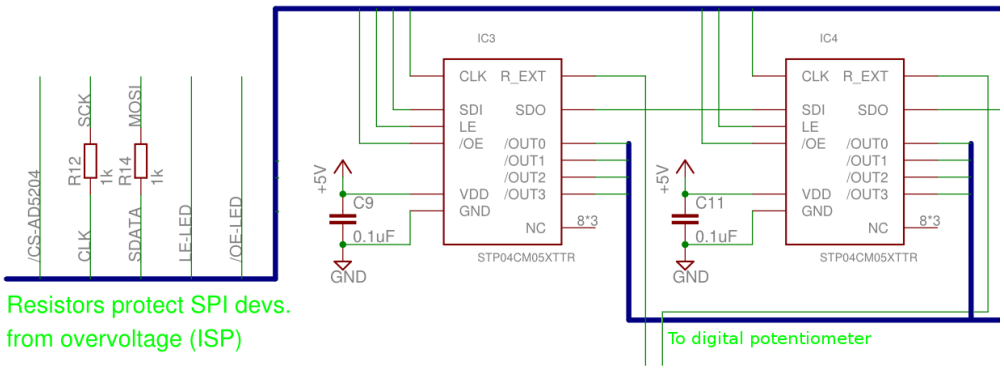


Figure 3.3: Power-LED sink driver, STP04CM05

The actual power dissipation can be calculated by summing up the dissipation of each channel:

$$P_{tot} = I_{led} * [(V_{led} - V_{f_{led0}}) + \ldots + (V_{led} - V_{f_{led3}})] \tag{3.2}$$

Assuming 4 white HPLED with $V_f = 4V$, a LED current of $I_{led} = 350mA$ and a supply voltage of $V_{led} = 4.1V$, the total power dissipation amounts to $P_{tot} = 0.14W$ and stays way below the limit.

Assuming 1 RGB-HPLED with $V_{f_{red}} = 3.0V$, $V_{f_{green}} = 4.1V$, $V_{f_{blue}} = 3.8V$ and the same current and voltage values as before, the power dissipation increases to $P_{tot} = 0.49W$ but also stays within the boundaries.

Voltage Adaptation For Red Channel

If the driver is used to control RGB-HPLEDs, the forward voltage of the red LED will generally be significantly lower than the forward voltage of the blue and green LEDs (Red: $2.5V \sim 3V$, Green: $3.2V \sim 4.1V$, Blue: $3.2V \sim 3.8V$). This excess voltage has to be handled by the HPLED driver and increases the power dissipation which can lead exceeding the thermal limits of the driver.

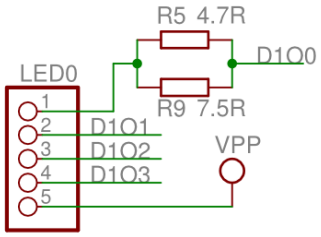


Figure 3.4: Voltage adaptation for red channel

To limit the strain on the driver IC, the board offers the possibility to equip it with a serial resistor for each red LED channel. To be able to drop the voltage precisely, two

resistors can be installed in parallel which also has positive influence on the required size of the resistor package because the load is spread between two resistors (see fig. 3.4).

Example: Dropping red channel voltage by 1V ($V_{led} = 4.1V$, $I_{led} = 350mA$)

$$R_{diff} = \frac{V_{diff}}{I_{led}} = 2.85\Omega$$
$$R_{diff_1} = \frac{R_{diff_2} * R_{diff}}{R_{diff_2} - R_{diff}}$$
$$R_{diff_1} = 4.7\Omega \implies R_{diff_2} = 7.5\Omega$$

The resistors have to be able to handle the power dissipation they are facing. A 1206 footprint was chosen for these resistors, as they are available with a power rating of up to 500mW, making them capable of withstanding even higher currents or voltages.

$$P(R_{diff_1}) = V_{diff} * \frac{V_{diff}}{R_{diff_1}} = 0.21W$$
$$P(R_{diff_2}) = V_{diff} * \frac{V_{diff}}{R_{diff_2}} = 0.13W$$

3.1.3 I2C Address

In order to use multiple RGB-LED driver boards on one I^2C bus each board needs to have a unique address. In the standard operation mode of the I^2C bus each device can be identified with a 7-bit address.

To allow changing the address of the boards without the need to change the software, the board includes a 4-way DIP-Switch that is used to define the lower 4 bits of the I^2C address.

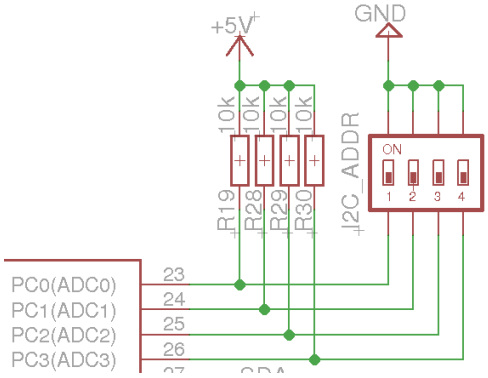


Figure 3.5: I2C address selector

Figure 3.5 shows the schematic of the 4-bit I^2C address selector. Each line is pulled high over a pull-up resistor. When the corresponding switch is flipped, the line is pulled to a low level.

Each line is connected to a GPIO pin on the microcontroller. The state of the pins is read during initialization of the I^2C bus to set the slave address of this board.

High-Current HPLEDs

The common drain architecture of the STP04CM05 IC makes it possible to drive HPLEDs with currents $\gg 400mA$ by connecting multiple channels together as shown in figure 3.6[1].

This makes it possible to use the system to control very powerful white HPLEDS (like the Cree MK-R with a max. current of 1.2A) up to their specification limits.

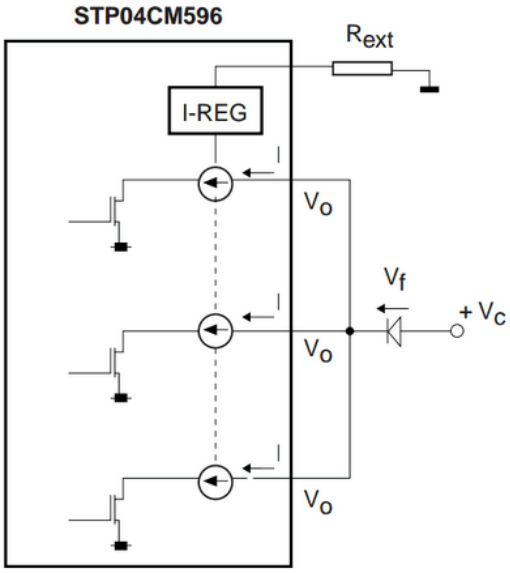


Figure 3.6: Multi channel output in common drain configuration

3.1.4 Current Limiting

The power-LED sink drivers can sink between 50mA and 400mA of current through their channels. The current value is defined by an external resistor value.

To be able to adapt the board to different HPLED types and have the option run it with a variable power the board includes a AD5204 IC to provide a 4-channel, 256-position

[1] image taken from AN2531 application note

digitally controlled variable resistor device [Analog-Devices(2010), AD5204] that is used to set the current limit dynamically.

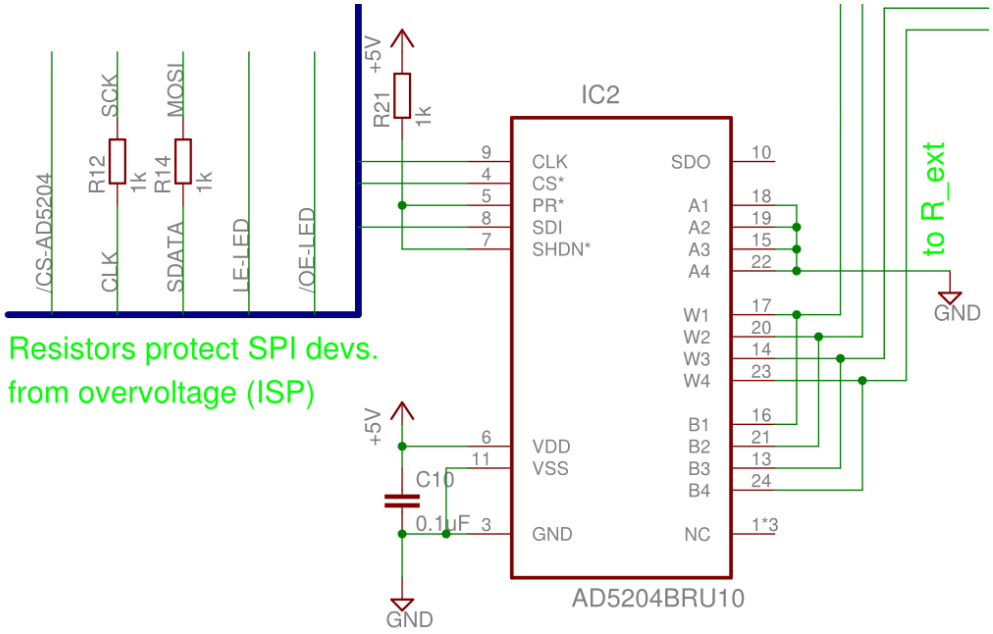


Figure 3.7: Current limiting using digital potentiometer AD5204

The digital potentiometer is also controlled over a SPI interface and shares the bus with the LED drivers. It has a range of $10k\Omega$ divided into 256 discrete steps.

The LED drivers expect an external resistor value between $180\Omega \implies 400mA$ and $1.55k\Omega \implies 50mA$. This means that the LED current can be defined in 35 steps with a resolution of 10mA.

$$Steps = \frac{1.55k\Omega - 180\Omega}{10k\Omega/256} = 35 \tag{3.3}$$

$$Resolution = \frac{400mA - 50mA}{Steps} = 10mA \tag{3.4}$$

3.1.5 Power Supply

While the on-board power supplies have been laid out assuming a 12V supply voltage, the board can handle supply voltages between 7V and 15V. A 5V voltage level is generated for the on-board peripherals using a linear power regulator and a 4V voltage level for the connected LEDs using high-efficient switched-mode power supplies.

On-Board Power Supply

All of the board peripherals run on a 5V voltage level that is generated by a linear power regulator of the 7805 series. Despite this type of regulator not being the most effective, it was chosen for its low external component count, small size, low electrical noise and cheap cost. The use of this regulator type is still advertised in recent books about embedded hardware design [Catsoulis(2005), Chapter 5.5].

The low efficiency of the regulator was not a detriment to its utilization on the board because it is not designed to be powered by batteries and the regulator is operating at a low degree of capacity utilisation. It is used to power the ICs used on the board which require a total current of $I_{out} = 60mA$.

$$\begin{aligned} P_{loss}(W) &= (V_{in} - V_o * I_{out} + V_{in} * I + bias) \\ &= (12V - 5V) * 60mA + 12V * 4.5mA \\ &= 0.47W \end{aligned}$$

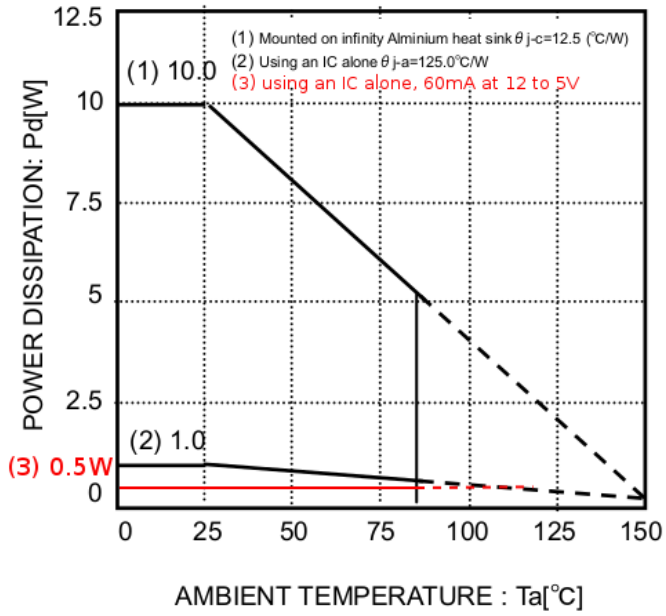


Figure 3.8: Thermal derating curve 7805, TO252 package

Figure 3.8 shows that the device stays well within its defined limits at the degree of capacity utilization.

Calculating the maximum output current $I_{out_{max}}$ at an ambient temperature of 85°C shows that the design has enough reserves for the output current and the ambient temperature to operate safely under reasonable conditions.

$$
\begin{aligned}
I_{out_{max}}(85°) &\leqq \frac{P_d - V_{in} * I_{bias}}{V_{in} - V_o} \\
&\leqq \frac{0.6W - 12V * 4.5mA}{12V - 5V} \\
&\leqq 78mA \\
I_{out_{max}}(25°) &\leqq 135mA
\end{aligned}
$$

LED Power Supply

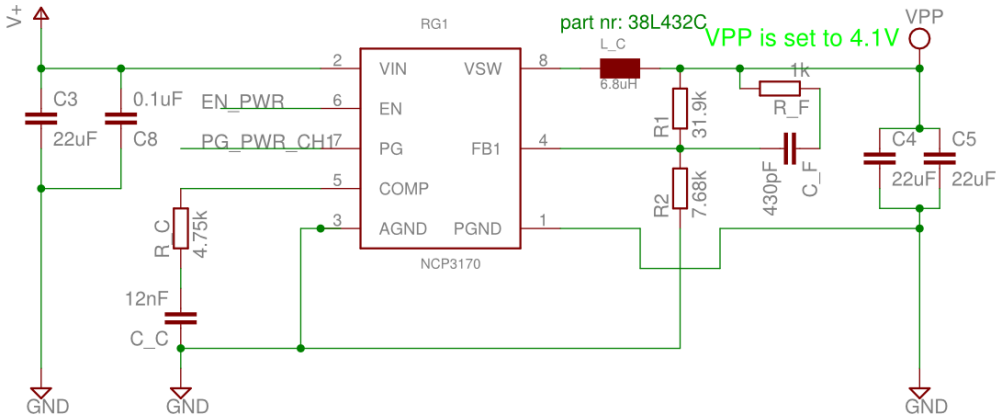


Figure 3.9: Switched-mode power supply schematic (NCP3170)

Because high-power LEDs draw high currents at low voltage levels ($250 \ldots 700mA$ at $4V$) it is important to use a highly efficient power supply that can deliver these currents. Switched-mode power supplies (SMPS) fulfill these requirements but care has to be taken when designing them (see section 3.1.5).

To be able to supply the connected LEDs with a suitable voltage the board includes a pair of identical switched-mode power supplies that generate a voltage of 4.1V and can drive up to 3A each.

The exact voltage level of these converters can be adapted to the requirements of the particular LEDs within a limited range ($3.5V \ldots 7V$) by changing the value of one resistor (R1).

Switched-Mode Power Supply Design

The switched-mode power supply is built upon a fully integrated synchronous PWM switching buck converter IC that includes both control and power stage to keep the external component count low [ON-Semiconductor(2011), NCP3170].

The following paragraphs show the calculations that were done for selecting the external components for the power supply. Table 3.1 shows the design parameters that form the basis of the calculations. The formulas presented in these paragraphs are taken from the datasheet and a book on switched-mode power supply design [Maniktala(2007)].

Paramter	Value	Comment
V_{in}	$12V$	12V power supplies with sufficient power to drive several boards are widely available and cheap in cost
V_{out}	$4V$	Common HPLEDs have a max forward voltage between 2.8V and 4.3V, but the output voltage can be varied by designing the power supply with enough tolerance
I_{out}	$2.4A$	Each channel can output 400mA max, each power supply is responsible for 8 channels $\Longrightarrow$ $2400mA$
D	0.33	Duty cycle: V_{out}/V_{in}
r_a	30%	Rule of thumb (from datasheet): the ripple current ration in the inductor should be between 10% and 40%
$r_{V_{in}}$	1%	Maximum allowed ripple component introduced by the power supply to the applied input voltage.
f_{sw}	$500kHz$	Fixed switching frequency of the IC
f_{cross}	$50kHz$	Loop cross over frequency, defined in datasheet

Table 3.1: Design parameters for switched-mode power supply

Input Decoupling

The input supply pin of the IC is very sensitive to noise. To decouple the input from high-frequency noise, a $0.1\mu F$ ceramic capacitor is installed in close proximity to the supply pin. The capacitor is also very important to support the power during the current surges when the high side MOSFET switches on [Maniktala(2007), Chapter 2.6].

Switched-mode power supplies need a bulk capacitor on the input side, to support the small value HF-bypass capacitor during the rest of the switching cycle. This input capacitor must sustain the ripple current produced by the switching MOSFET on the V_{in} line. For this reason a ceramic capacitor was selected since they have the lowest equivalent series resistance (EST).

The bulk capacitor will see a current draw of $I_c = I_{out} * D$ over a period of $\Delta t = (1 - D) * 1/f_{sw}$. The voltage change during this time is defined by the allowed ripple

voltage $\Delta V_{in} = V_{in} * r_{V_{in}}$ Using these values the required capacitor size can be calculated using formula 3.5 [Maniktala(2007), Chapter 2.11].

$$C_{bulk} = I_c * \frac{\Delta t}{\Delta V} \tag{3.5}$$

$$= I_{out} * D \frac{1 - D}{f_{sw} * V_{in} * r_{V_{in}}} \tag{3.6}$$

The calculations show that a capacitor value of at least $C_{bulk} = 10\mu F$ is required. Because the capacity of ceramic capacitors decreases when the voltage increases, a value of $22\mu F$ was selected.

Output Decoupling

The output capacitor has to be chosen based on the maximal admissible output ripple voltage and sized to be able to sustain the current during the load transient without discharging it.

$$\Delta V_{out-ESR} = I_{out} * r_a * (ESR_{C_{out}} + \frac{1}{8 * f_{sw} * C_{out}}) \tag{3.7}$$

$$\Delta V_{out-dis} = \frac{I_{tran}^2 * L_{out * f_{sw}}}{2 * f_{cross} * C_{out} * (V_{in} - V_{out})} \tag{3.8}$$

Two parallel ceramic capacitors with a value of $C_{out} = 22\mu F$ were selected. With these capacitors the ESR induced voltage variation is $\Delta V_{out-ESR} = 22mV$ and a voltage drop due output capacitor discharge of $\Delta V_{out-dis} = 180mV$.

Inductor Selection

The required inductance of the output inductor is dominated by the admissible output ripple current. The output ripple current in combination with the ESR of the output capacitors causes an output voltage ripple. Since ceramic capacitors with a very low ESR are used for output decoupling, the ripple current ration was limited to 30% which is within the bounds given in the datasheet of the NCP3170 IC [ON-Semiconductor(2011), p. 14].

$$L_{out} = \frac{V_{out}}{I_{out} * r_a * f_{sw}} * (1 - D) \tag{3.9}$$

Using equation 3.9 the inductor value can be calculated to be $L_{out} = 6.9\mu H$.

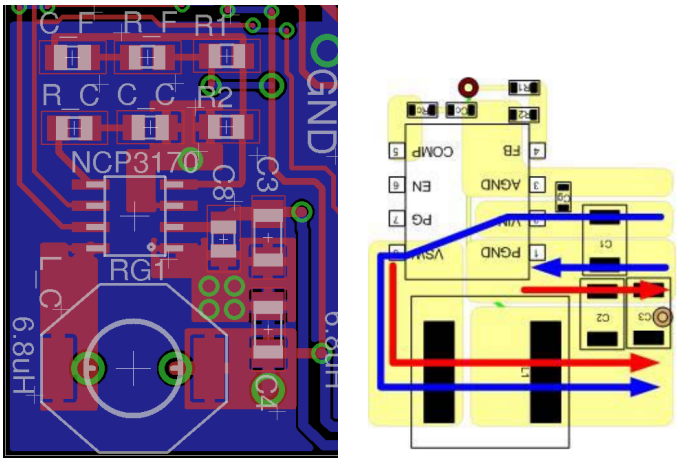
$$I_{RMS} = I_{out} * \sqrt{1 + \frac{r_a^2}{V_{out}}} \tag{3.10}$$

$$I_{Peak} = I_{out} * (1 + \frac{r_a}{\sqrt{V_{out}}}) \tag{3.11}$$

To select an inductor the RMS current and the peak current also have to be taken into account. Based on the calculated values of $I_{RMS} = 2.42A$ and $I_{Peak} = 2.76$ a shielded SMD power inductor (SRU1048-6R8Y) with an inductance of $6.8\mu H$ was chosen.

PCB layout of switched-mode power supply

Since a switched-mode power supply works with high pulsing current flows it can be a cause for noise in the rest of the system. For this reason the power supply circuitry is placed on a separate PCB area with a local ground. The layout of the switched-mode power supply is based on the design advice given in the data sheet, as shown in figure 3.10.



(a) PCB layout (b) Proposed layout

Figure 3.10: Layout of switched-mode power supply

3.1.6 PCB Layout

The layout of the PCB was done with a minimal size in mind to make PCB cheap to produce and to easy to integrate when it is used in fixed light installations.

All parts use SMD packages (0805 for resistors and capacitors and the smallest package option for all ICs) and both sides of the board are used to place the components.

To prevent interference between analog signals with a high switching frequency and the digital control signal the layout is divided into power-supply and digital circuitry sections as shown in figure 3.11.

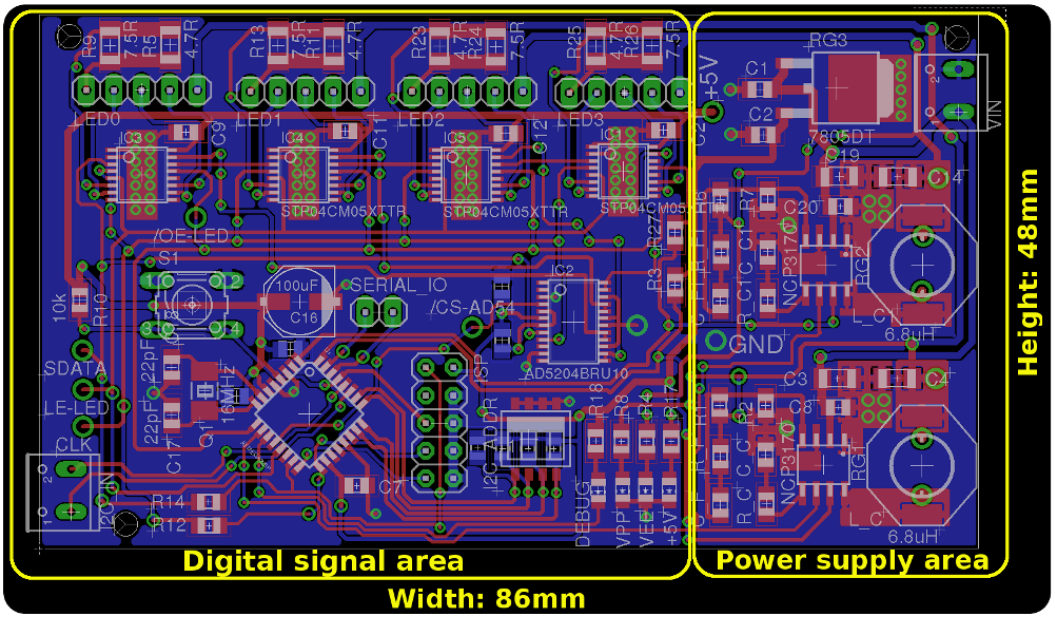


Figure 3.11: PCB layout

Chapter 4

System Software

4.1 Overview

The overall system consists of three self-contained programs, each of which runs on a different platform and is written in a different programming language. This is possible due to well-defined APIs that allows data exchange across the boundaries of platforms and languages.

The software on the HPLED-Driver board mimics the interface of an EEPROM on an I^2C port, to allow control over the behaviour. The API is based on an externally addressable (r+w) memory block, whose content mirrors the current state of the board. A periodic software routine reads and updates this memory block, calculates a 16-channel soft-PWM and updates the peripherals accordingly (see section 4.2).

The coordinator software, running on the Raspberry Pi, is responsible for acting as a gateway HPLED-Controller boards and the user. It offers an object-oriented interface to the HPLED-Controllers that capsules all direct hardware interaction. This object-based representation of the system is exposed over a Web-API based on the RESTful architectural style. It allows cross-platform access to the state of HPLED-Controllers (see section 4.3).

The Web-frontend is written in JavaScript and HTML5. It runs on the device of the user after it has been delivered by a Web-server. It uses AJAX to communicate with the Web-API and dynamically adapts the user interface to the current state of the system, allowing precise control over the behaviour of the HPLED-Controllers (see section 4.4).

It can also be used to group LEDs from different boards into a unit (LED-set) providing additional control options. Once created, these LED-sets are reflected in the Web-API, and visible to all users of the system. They retain their state after a system restart.

4.2 HPLED-Controller Software

4.2.1 Software Design

The software on the controller consists of an interrupt driven I^2C slave interface module, a periodically executed interrupt handler function for calculating and outputting the software PWM and a main loop that coordinates data transfers between the I^2C buffers and the rest of the program.

The current state of the controller board is modelled using a global volatile structure that is updated by the main loop and read by the periodic interrupt handler function. This structure has the same memory layout as the I^2C receive buffer (see tab. 4.1). This enables an effective update of the controller state by using a raw memory copy (*memcpy*) operation when the I^2C interface signals a complete transaction.

A hardware timer (Timer1) is used to generate a periodic interrupt. The interrupt handler calculates a 16 channel software PWM and outputs it serially over an SPI interface. These calculations are based on the values in the global state structure (see section 4.2.3).

The main loop checks if the I^2C module has registered a receive buffer value update and copies the updated memory area into the controller state structure.

In case the current limit of a channel has changed the main loop is responsible for updating the hardware. Since this is not a time critical operation the current limit is set in the main loop when CPU time is available to prevent introducing jitter into the PWM signal.

Interrupt Driven Operation

There are two competing interrupt sources on the controller board: the I^2C hardware interface interrupt and the periodic Timer1 compare interrupt used to calculate the software PWM.

The I^2C interrupt is triggered whenever the I^2C hardware of the ATMega has received or transmitted one byte of data and needs to be instructed on how to continue. The EEPROM I^2C library, written for this project, handles the decoding of transmissions internally and uses a callback function to notify the main program in case of successfully completed transmissions. Details of the implementation can be found in section 4.2.2.

The Timer1 compare interrupt is a periodic interrupt used to calculate the 16-channel soft-PWM and output these channels serially over the SPI bus to the LED drivers. Its timing behaviour is quasi-deterministic which makes it possible to operate the board at a predictable system load by adapting the interrupt frequency. The period

of the interrupt is constant even in the presence of I^2C communication so that no PWM jitter is noticeable .

Nested interrupts are disabled by default on the ATMega8 series to prevent stack overflow situations caused by a large number of nested interrupt calls that can only be solved with a system restart.

Each interrupt has its own interrupt request flag, that is cleared as soon as the interrupt handler fires. This means that interrupts can be missed if two equal interrupt events occur while an interrupt handler executes. These missed interrupts lead to failed I^2C transfers and jitter in the soft-PWM.

For this reason the software has to be designed in a way that limits the chance of interrupts being missed. This is achieved by minimizing the execution time of the interrupt handlers and limiting the frequency of their occurrence in relation to their execution time (see section 4.2.3).

4.2.2 I2C slave interface implementation

The I^2C interface is implemented in a way that makes it behave similar to an EEPROM. It offers an addressable memory block with read and write access over the I^2C bus. The memory block mirrors the state of the controller (see table 4.1) and allows control over its behaviour.

Byte	Function	Description
0-4	driver[0][0..3]	PWM output values of the four channels of the first HPLED driver IC. *Input:* 0 = off, 255 =constant on
...	...	...
11-15	driver[3][0..3]	...
16-19	limit[0..3]	Input values for the digital potentiometer that is used to set the output current of the driver ICs. *Range:* 0 ... 255 (see sec. 3.1.4 for value conversion)
20	limit_update	Signals the board that one of the current values was updated and needs to be shifted out to the digital poti. *Input:* 0 =false, 1 ... 255 =true
21	brightness	Defines the brightness of all LED channels by applying a hardware PWM signal to the ¬output-enable pin of the driver ICs. *Input:* 0 = off, 255 =full brightness

Table 4.1: Memory map of I2C receive buffer

Write transactions consist of a 1-byte write operation that defines the destination memory address, followed by an arbitrary number of data bytes (length is only limited by

the buffer size), that are copied into the I²C receive buffer at the defined offset. If no address is set first, the write operation will fail.

Read transactions also consist of 1-byte write operation to define the memory access, but in this case the source address is set to 0 if no address is set first.

A simple error detection mechanism is used to prevent incomplete / faulty transmissions from corrupting the system state. The value of the last byte of each write operation has to equal the length of the write operation. Only if the value of the last byte matches the length of the received data will the data be copied from an internal buffer to the global receive buffer.

Software communication with the library happens over a receive and transmit buffer that the library exports into the global namespace. Callback functions can be registered with the library, that are triggered when a read or write transmission has been completed successfully. These callback functions do not return data but information about the sector of the buffers that was modified.

The receive-/transmit process is interrupt-driven. ATMega I²C hardware generates an interrupt for each byte that is received or transmitted over the I²C bus. Within the interrupt handler, the hardware has to be instructed on how to proceed. The library abstracts away the byte-wise operation of the interface and only interrupts the flow of the main program (if callbacks have been registered), when a transaction has been completed.

The I²C library responsible for this behaviour was implemented as a part of this project. It was inspired by existing solutions[1,2] but re-written from scratch. The code can be found in the online repository of this project: twislave_sm.h and twislave_sm.c

4.2.3 Software PWM

The LED drivers have a 4-bit serial-in parallel-out data interface, which can be used to switch each channel on or off. Four of these ICs are daisy-chained and behave like a 16-bit shift register.

The software uses the shift-register functionality to control each output channel with a PWM input signal. To do this, the value in the 16-bit shift register is updated with a fixed frequency. Before every update a bitfield is calculated where each bit represents the current binary status of the associated PWM channel (see listing 4.1).

The binary status of the PWM channels is obtained by comparing the PWM target value of each channel with a ticker value that is increased with every execution of the routine. When the target value is larger than the ticker value, the associated bit of the

[1] *"TWI/I2C library for Wiring & Arduino"* by Nicholas Zambetti
[2] *"TWI SLave"* library by Martin Junghans (www.jtronics.de)

```
1  /* set the resolution of the PWM signal */
2  #define PWM_LEVELS 64
3  #define PWM_STEP 256/PWM_LEVELS
4  ...
5  static volatile uint16_t soft_pwm;
6  ...
7  /* calculate soft PWM values and output them via SPI
8   * (periodic callback function for Timer1) */
9  void updatePWM() {
10   /* update output at beginning of function to account for variable
11    * execution time of the function (lower jitter) */
12   sendBytes(soft_pwm);
13
14   /* if the soft PWM ticker / comperator overflows, reset the current
15    * soft-pwm values */
16   if ( ++ticker >= PWM_LEVELS ){
17     ticker = 0;
18     soft_pwm = 0;
19   }
20
21   /* calculate the PWM values for the next iteration */
22   uint8_t current_val = ticker * PWM_STEP;
23   soft_pwm |= pwm_bitfield(0, current_val);
24   soft_pwm |= pwm_bitfield(2, current_val) << 8;
25 }
```

Listing 4.1: Interrupt based Software-PWM: periodic callback

bitfield is set (see listing 4.2). This conditional operation causes the execution time of the function to vary a bit depending on the current PWM value (see section 4.2.3).

A hardware timer (timer1) is used to generate a periodic interrupt. The calculation of the soft-PWM and the actual shift operation happen within the interrupt handler of the timer1 compare interrupt.

Software PWMs can suffer from jitter that is caused by variations in the duration of the execution time of the interrupt handler [AVR(2006), AVR136].

To keep the jitter of the PWM low, the current PWM bitfield is shifted out at the beginning of each interrupt, and after that the bitfield for the next shift operation is calculated.

Resolution and Frequency

A minimal PWM frequency of 65Hz[3] is required to obtain a flicker free output across all the whole PWM output range. If the frequency of the PWM is lower than this value,

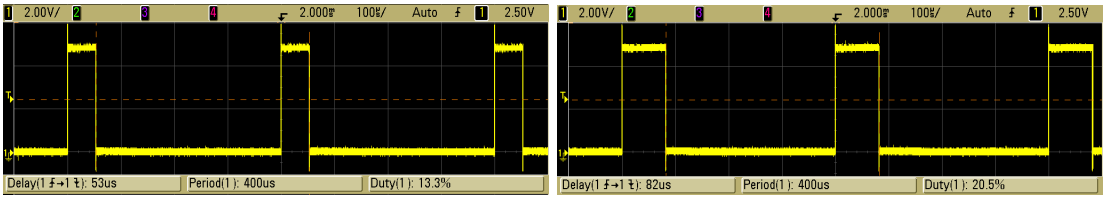
[3]Value obtained by own experiments with the PWM frequency

```
1  /* define the hardware configuration of the controller */
2  #define CHANNELS 4
3  #define LED_COUNT 4
4
5  /* calculate the bitfield for 8 PWM channels */
6  uint8_t pwm_bitfield(uint8_t idx, uint8_t current_val) {
7    uint8_t bitfield = 0;
8    uint8_t soft_pwm_idx = 0;
9    uint8_t rgb[LED_COUNT];
10   for(uint8_t led = idx; led < idx+2; led++) {
11     rgb[0] = rgb_state.led[led].r;
12     rgb[1] = rgb_state.led[led].g_1;
13     rgb[2] = rgb_state.led[led].g_2;
14     rgb[3] = rgb_state.led[led].b;
15     for (uint8_t channel = 0; channel < CHANNELS; channel++) {
16       if (current_val > 255-rgb[channel]) {
17         bitfield |= _BV(soft_pwm_idx);
18       }
19       ++soft_pwm_idx;
20     }
21   }
22   return bitfield;
23 }
```

Listing 4.2: Interrupt based Soft-PWM: PWM bitfield

the human eye can no longer be tricked into thinking that it sees a constant light source and the light begins to flicker noticeably

Another factor that has to be taken into account when deciding on the PWM resolution and frequency, is the CPU time required to execute the interrupt handler. It must be guaranteed that the CPU doesn't run into an overload situation and that there is enough free CPU time to execute the main loop and handle the interrupt driven non-deterministic I^2C communication.



(a) best case scenario (b) worst case scenario

Figure 4.1: Execution time measurement of timer1 interrupt handler

The execution time of the interrupt handler can be measured to be $t_{exec} = 53\mu s$ in the best case scenario and $t_{exec} = 82\mu s$ in the worst case scenario[4] (see fig. 4.1). For this measurement a GPIO pin of the microcontroller is set at the beginning of the execution of the interrupt handler and reset at the end.

The highest possible PWM resolution depends on how much CPU time needs to be available for the rest of the system. To guarantee that enough CPU time is available to handle the main loop and I^2C communication is available, a target system load of ($load_{max} \leq 50\%$) is defined and the possible PWM resolution is calculated based on this value.

To generate a soft-PWM with a certain frequency, an interrupt needs to be generated $2^{resolution}$ times during one PWM period ($f_{pwm} \geq 65Hz$):

$$\Delta t_{int} = \frac{1}{f_{pwm} * 2^{resolution}} \tag{4.1}$$

Based on the distance between interrupts and the execution time, the load of the system can be calculated ($t_{exec} = \ldots \mu s$, $load_{max} \leq 50\%$):

$$load(\Delta t_{int}) = \frac{t_{exec} * 100}{\Delta t_{int}} \tag{4.2}$$

$$load(resolution) = t_{exec} * 100 * f_{pwm} * 2^{resolution} \tag{4.3}$$

After intensive testing with different system load situations (see section 5.2) the target system load was re-defined to $load_{max} \leq 35\%$. Based on this value the maximal PWM resolution is 6-bit which results in a system load of $load(6bit) = 34.12\%$ and a interrupt period of $\Delta t_{int} = 240\mu s$

4.2.4 Arduino Compatibility

"Arduino is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software"[5]. It builds upon the ATMega8 processor family and includes a set of well-tested C-libraries to interact with the ATMega8 hardware. The controller board uses the same processor family which makes it possible to use these libraries to speed up the development process.

The Arduino is normally programmed in its own IDE, writing code in a wiring based programming language. Code is uploaded over a virtual serial connection made possible by a boot loader on the microcontroller.

To be able to benefit from the libraries without the need to use a bootloader or write code in the Arduino programming language, a makefile is used to compile and upload

[4] best case: $pwm_{val} = 0 \implies$ output bit is never set, worst case: $pwm_{val} = 255 \implies$ output bit is set in every update cycle which consumes additional CPU instructions (see listing 4.1)

[5] http://arduino.cc

the code via an ISP programmer. This way all Arduino libraries can be used while being able to write code in C or C++.

4.3 Coordinator Software

4.3.1 Software Design

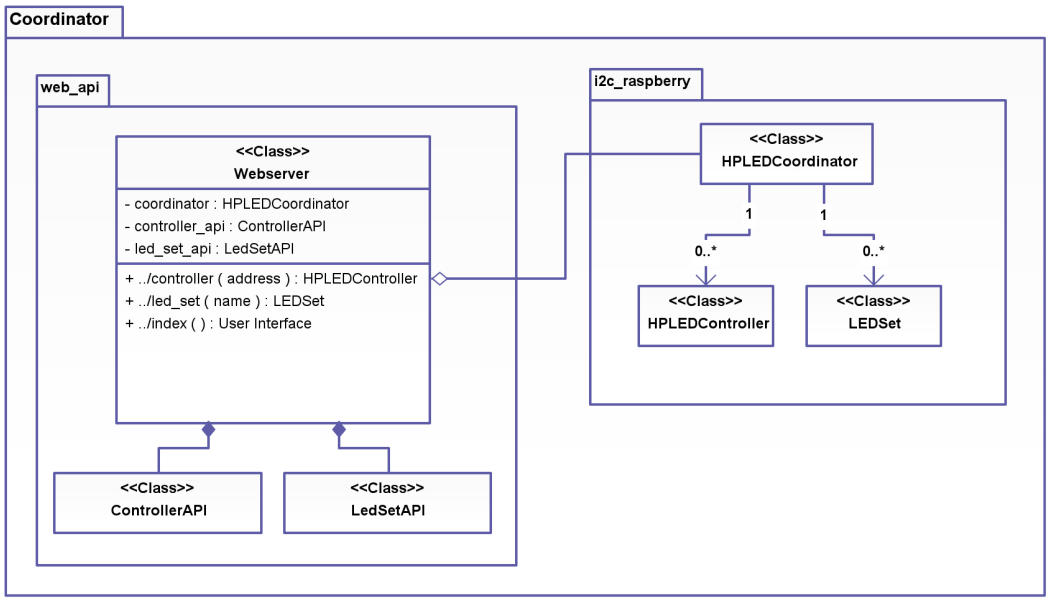


Figure 4.2: UML package diagram of Coordinator software

The coordinator has the task to coordinate between the frontend (Web-API, user interface) and the back-end (HPLED-controller boards) and act as a gateway between the Internet and the I^2C bus. It consists of two software modules as shown in figure 4.2.

The i2c_raspberry module is responsible for communicating with the HPLED-Controller boards over I^2C and offering an object-oriented representation of their current state. It abstracts away the details of the low-level communication and adds a virtual layer that can be used to group and control LEDs from different controllers as one unit.

The web_api module is responsible for offering the RESTful web-API that is used to provide external access to the system state and implements a simple Web-server that delivers the HTML and script files for the Web-interface.

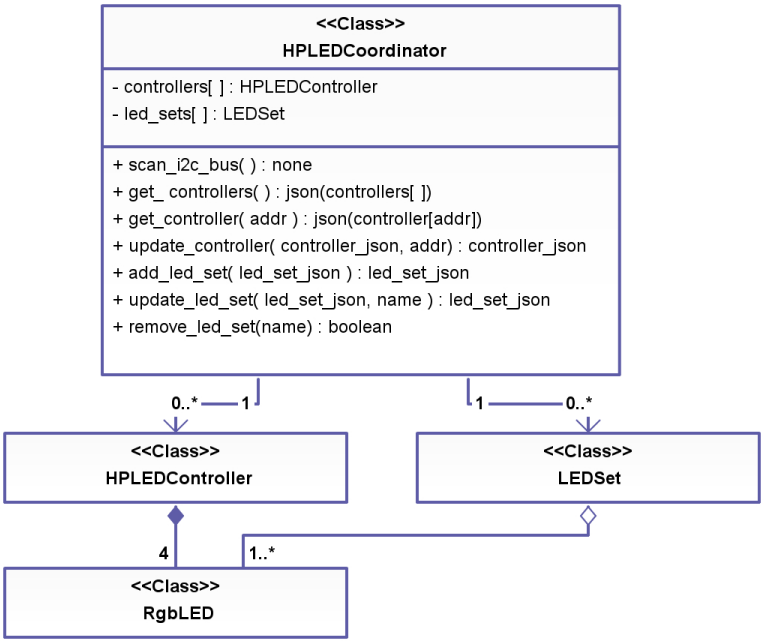


Figure 4.3: UML class diagram of I2C Raspberry module

4.3.2 I2C Raspberry Module

The *i2c_raspberry* module scans the I^2C bus during initialization to find HPLED-Controllers connected to the bus. An object is of the *HPLEDController* class is created for each detected device and added to the *HPLEDCoordinator*.

After the bus scan is completed the coordinator will try to restore LED-Sets that were active during the last execution of the program. It verifies that all LEDs of the set are available and appends the *RgbLED* objects to the associated *LEDSet* objects. These LEDSets are also added to the *HPLEDCoordinator*

The *HPLEDCoordinator* class implements the public module interface functions. It allows to access and modify the state of the hardware of the system and hides away all low level interaction.

The public methods of the *HPLEDCoordinator* class are mapped almost directly to the RESTful Web-API. Thorough check of all input values is done, because faulty usage of the external API is to be expected.

4.3.3 RESTful Web-API Module

RESTful Software Architecture

The RESTful architecture (**RE**presentational **S**ate **T**ransfer) is a software design style for distributed systems. It is based around exposing resources over a Uniform Resource Identifier (URI) and defining a set of actions that can be performed on those resources (Get, Put, Post, Delete, Options). It was first described by Roy Fieldings [Fielding(2000)], one of the main authors of the Hypertext Transfer Protocol (HTTP), and is the foundation for the Internet as we know it.

RESTful Web-API

The web-API of this program exposes two resource types over different uniform resource identifiers (URI). The controller URI (`.../controller`) can be used to interact with the HPLED-controllers directly , and the LED-Set URI (`.../led_set`) can be used to create, modify and delete LED-Sets.

The web-API grows dynamically when controllers are connected to the system or LED-Sets are created by the user. The full web-API is defined in tables 4.2 and 4.3.

The web-API expects and returns data in the JavaScript Object Notation (JSON) format, which is a human-readable string representation of objects that can easily be parsed in all major programming languages. The following shows an abbreviated representation of a HPLED-Controller object.

```
{
  "controller": [
    {
      "addr": 44,
      "name": "Controller44",
      "uri": "http://10.42.0.47:5000/controller/44"
      "leds": [
        {
          "channel": 0,
          "color": { ... }, ...
        }, ... ]
    }, ... ]
}
```

Listing 4.3: Excerpt from controller object in JavaScript Object Notation

The API adheres to the HTTP and RESTful requirements of robustness in the presence of faulty input. When the attempt is made to access a non-existing resource or incorrect data is provided, the API will respond with an appropriate HTTP status code and a precise error description while not being affected itself. The correct use of HTTP

status codes allows programs interacting with the API to react appropriately to error conditions.

Action	URI	Return value	Description
GET	http://.../controller	controllers[0:-1]	Returns a list with the state of all controllers connected to the system
GET	http://.../controller/addr	controllers[addr]	Returns the state of the controller with the given address (if it exists)
PUT	http://.../controller/addr	controllers[addr]	Modifies the state of the addressed controller

Table 4.2: RESTful web-API for HPLED-Controllers

Action	URI	Return value	Description
GET	http://.../led_set	led_sets[0:-1]	Return a list with the state of all user-defined LED-Sets
GET	http://.../led_set/name	led_sets[name]	Returns the state of the LED-Set with the given name
PUT	http://.../led_set/name	led_sets[name]	Updates the state of the LED-Set (adding/removing LEDs, setting status, changing color)
POST	http://.../led_set	led_sets[name]	Creates a new LED-Set and returns the JSON representation of the new object
DELETE	http://.../led_set/name	HTTP_OK 200	Disconnects the LEDs from the LED-Set and removes the objects from the list

Table 4.3: RESTful web-API for LED-Sets

Implemenation

The Web-API is based on the Python microframework *flask*[6], which makes it easy to create a webserver with sophisticated URL routing, offering precise control over the handling of incoming HTTP requests.

[6] http://flask.pocoo.org

Listing 4.4 shows an excerpt from the implementation of the controller API. The *ControllerAPI* class is registered at the webserver and will be called when a HTTP request is made on the `http://.../controller/addr` URI. Depending on the type of the HTTP request (GET, PUT, POST, ...) the associated method of the *ControllerAPI* class will be called and used to handle the request appropriately.

```
1  ''' RESTful API for accessing the state of HPLED controllers '''
2  class ControllerAPI(MethodView):
3    def __init__(self):
4      global coordinator
5      self.coordinator = coordinator
6      super(ControllerAPI, self).__init__()
7
8    def get(self, controller_address):
9      try:
10       controller = self.coordinator.get_controller(controller_address)
11       return jsonify(add_controller_uri(controller))
12     except HttpError as e:
13       abort(e.error_code)
14
15 ''' Register the routes for the RESTful Controller API '''
16 controller_view = ControllerAPI.as_view('controller_api')
17 app.add_url_rule('/controller/<int:controller_address>',
18     view_func=controller_view, methods=['GET', ])
```

Listing 4.4: RESTful controller API (excerpt)

4.3.4 Hardware Interface

The hardware interface is based on the I^2C bus which is available on external GPIO pins of the Raspberry Pi. In Linux I^2C hardware adapters are exposed as device files under `/dev/i2c-n`, if the appropriate kernel module (`i2c-dev`) is loaded.

The full functionality of the I^2C bus can only be accessed if communication with the I^2C adapter is done through input/output control (IOCTL) calls, because the I^2C interface does not fit well within the stream-based `read` and `write` interface normally used to access device files [Kernel-Devs(2000)].

This system uses a part of the quick2wire python library[7] which encapsules calls to the IOCTL interface. Because the library is written for python3 and contains additional code for interacting with the GPIO pins of the Raspberry Pi, it was forked, backported to python 2.7 and streamlined for the use in the system. The relevant code can be found in i2c.py and i2c_ctypes.py

[7] https://github.com/quick2wire/quick2wire-python-api

Bus Speed

The Raspberry Pi uses a Broadcom BCM2835 ARM core which contains a *Broadcom Serial Controller* (BSC) that is compliant with the I^2C bus that can operate only in single master mode and whose timing is controllable by software [Broadcom(2012), section 3.1].

It is desirable to maximize the system bus speed to be able to coordinate a large number of slaves and while still being able to control the output with a very small delay. The trade-offs between a fast bus speed and a low error rate have been discussed in 4.2.1 and the effects of different bus speeds have been thoroughly tested (see section 5.2).

The bus speed can be modified by providing an optional parameter when loading the kernel module responsible for the low-level interaction with the Broadcom peripherals (`i2c-bcm2708`).

```
~$ sudo rmmod i2c_bcm2708
~$ sudo modprobe i2c_bcm2708 baudrate=400000
```

The testing revealed that the I^2C hardware of the ATMega328 microcontrollers on the controller boards can handle a bus speed of up to 600kHz but the optimal bus frequency was found to be 400kHz (see section 5.2.3).

4.4 User Interface

4.4.1 Functionality

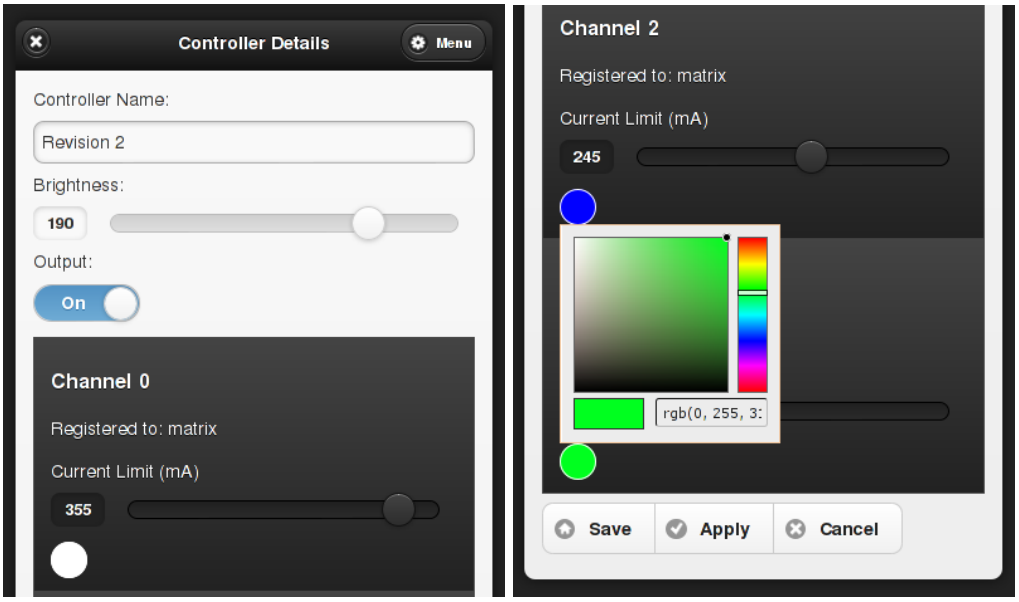
Modifying State of Controllers

The status page shows a list of all controllers that are currently connected to the system. When a controller is selected a dialog shows that can be used to see the current status of the controller and modify it (see figure 4.4).

Each of the 4 RGB-channels of the board can be modified in terms of its current limit and its colour . A simple colour-picker is used to select the output colour and provide feedback over the current colour.

Grouping Multiple LEDs to Sets

The user interface allows the user to create sets of LEDs that can be used to group LEDs into functional units that provide additional control mechanisms. A new set is created with a dialog that shows all connected controllers and their available output channels. An LED can only be added to a set, when it is not registered with another set yet (see figure 4.5a).



(a) Exposed details (b) Setting colour

Figure 4.4: UI: Controller details

LED sets can be edited at a later stage by adding and removing LEDs from the set, changing its name or deleting it altogether (see figure 4.5b).

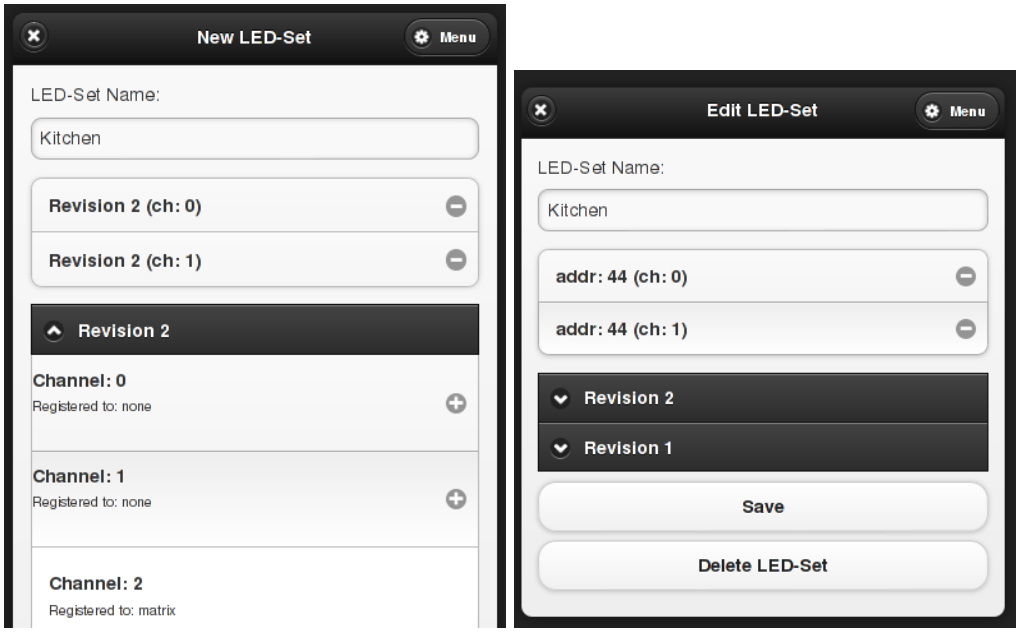
Controlling LED-Sets

The user interface for controlling LED-sets offers additional control modes for LED-sets improving the usability:

Live update mode: changes to the LEDs or set attributes will be transferred immediately to coordinator, without the need to apply or save

Status: switches all LEDs of the Set off/on immediately, while retaining their color and current value

Treat all LEDs as one: when the colour or current value of any LED is changed, this change is applied to all other LEDs at the same time, making them behave like one big and powerful LED.



(a) creating a LED-Set (b) editing a LED-Set

Figure 4.5: UI: creating LED-Sets

4.4.2 Software design

The code for the user interface is based on the jQuery mobile framework, which is *"a unified, HTML5-based user interface system for all popular mobile device platforms, built on the rock-solid jQuery and jQuery UI foundation"*[8].

The software consists of a HTML5 document that contains the static markup for all pages of the web application and a JavaScript file that interacts with the RESTful API of the coordinator. Control elements are added dynamically to the Document Object Model (DOM) of the website based on the data returned from the Web-API.

Since the whole website in contained in one single HTML document, it is not necessary to load new data for page transitions. They are performed by manipulating the DOM and happen instantaneously. This makes it possible to animate page transitions to imitate the look-and-feel of native applications.

The state of the system is requested over asynchronous java script (Ajax) calls to the RESTful API, that are done invisibly for the user in the background (see 4.4.3). The received data objects are stored in the DOM and can be modified with control elements on the page and sent back to the coordinator if the user requests to do so.

[8] jquerymobile.com

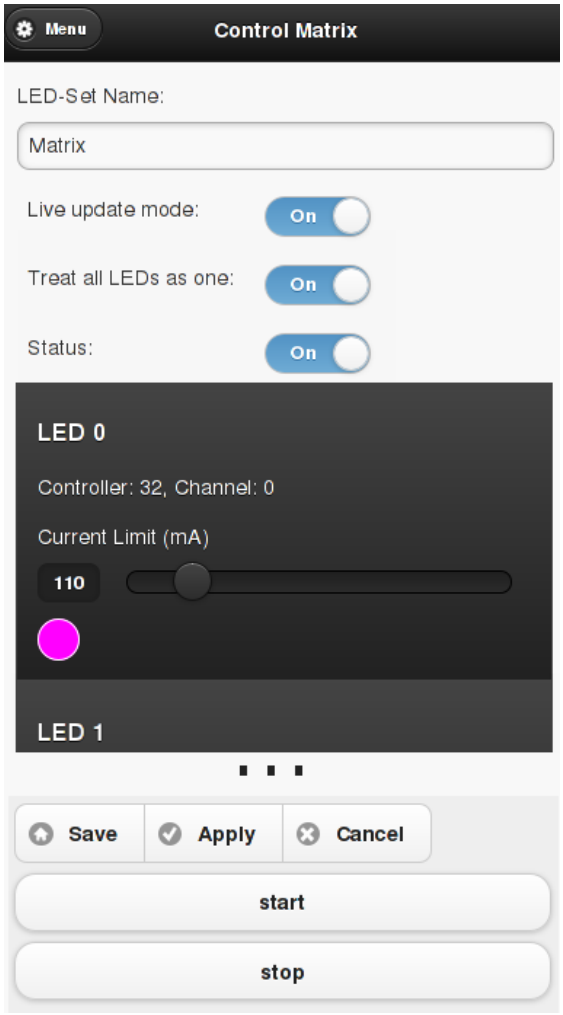


Figure 4.6: UI: controlling LED-Sets

Because the RESTful API provides data in the JavaScript Object Notation (JSON) format it is very simple to work with the data entities within the frontend.

4.4.3 Implementation

Accessing the RESTful API

Listing 4.5 shows how data is requested from the RESTful API. This function sends a HTTP GET request to the `http://.../controller` URI (see tab. 4.2) and asks the server to return the data in JSON format. This feature can be used to return different data representations from the same URI, for example a HTML document for human users and JSON data for machines.

It also shows the functional style of JavaScript. The `getControllers(..)` function expects to be passed a callback function, that will be triggered, when the Ajax call finishes successfully. This is used to update the user interface inconspicuously in the background. Listing 4.7 shows a function that is used for this purpose.

```
1  function getControllers(handleDataFunction) {
2    /* Request list of controllers */
3    $.ajax({
4      type: "GET",
5      url: "http://"+ location.host + "/controller",
6      data: null,
7      dataType: "json",
8      success: function(data, xml_request, options) {
9        handleDataFunction(data);
10     }
11   });
12 }
```

Listing 4.5: Accessing RESTful API with JavaScript

Creating control elements dynamically

Listing 4.7 shows how new control elements are added to the DOM dynamically, when new data is received from the RESTful API by passing it to the `getControllers(..)` function.

```
1  <div data-role="content" >
2    <h3>Overview over all controllers connected to the system</h3>
3    <ul data-role="listview" id="controller_list" data-inset="true" data-
         theme="d" data-divider-theme="d">
4      <!-- list of controllers is embedded here dynamically -->
5    </ul>
6    <hr>
7  </div><!-- /content -->
```

Listing 4.6: Markup for controller list

The static HTML markup of the status page has an empty list element with the unique identifier `#controller_list` (see listing 4.6). After the HTML is parsed into the DOM by the browser, jQuery can be used to identify and select this list by using CSS selector syntax `$('#controller_list')`.

Once an element has been selected in the DOM it can be modified, and changes will immediately be reflected by the user interface. New list entries are created dynamically and added to the list (lines 13-21 of 4.7) by iterating over the received controller list. The loop function shows that it is possible to access the data returned from the web-API in a very literate and effective way.

```
1  function addControllersToStatusList(data) {
2    // Remove existing controllers in the list
3    $('#controller_list')
4      .children()
5      .remove();
6
7    // Add a header entry to signal the contents of the list
8    var divider = $('<li>')
9      .attr({'data-role':'list-divider', 'class':'controller'})
10     .text("Controllers:");
11   $("#controller_list").append(divider);
12
13   // Create one list entry per controller
14   for (var i = 0; i < data.controller.length; i++) {
15     var controller_item = $('<li>')
16         .data('controller', data.controller[i])
17         .append($('<a>')
18           .attr( {'href':'#contr_details', 'data-rel':'dialog'} )
19           .text(data.controller[i].name + ' (Address:' +
20                 data.controller[i].addr + ')'))
21         .appendTo('#controller_list');
22   }
23   $('#controller_list').listview('refresh');
24
25   // Store the json data in the controller list
26   $('#controller_list .controller').data({'json': data});
27 }
```

Listing 4.7: Creating control elements dynamically by modifying the DOM

To be able to show the controller details dialog for a certain controller it needs to be possible to figure out which list entry the user selected. For this reason the object representation of each controller is added to the list entry as a data member (line 16). By storing the data in this way, the event handler that is triggered when a list entry is selected, can be kept very generic. All it needs to do is to load to data element from the context element it was triggered by, and initiate the dialog with the values from the object (shown in listing 4.8).

```
1 // Intercept clicks on "controller_list" list entries to append the correct
2 // data to the page
3 $(document).on('click', '#controller_list li', function(q) {
4   var controller = $(this).data('controller');
5   $('#contr_details').data('controller', controller);
6 });
```

Listing 4.8: Intercepting clicks to preload data

Chapter 5

Testing & Verification

5.1 PCB

5.1.1 Electrical Behaviour

Output Current

To test how precisely the output current can be controlled, a manual current test was performed where the selected current value (set in UI) was compared to the resulting current. The test was conducted with the help of a *FLUKE 73* multimeter connected in series to the voltage supply channel of one LED.

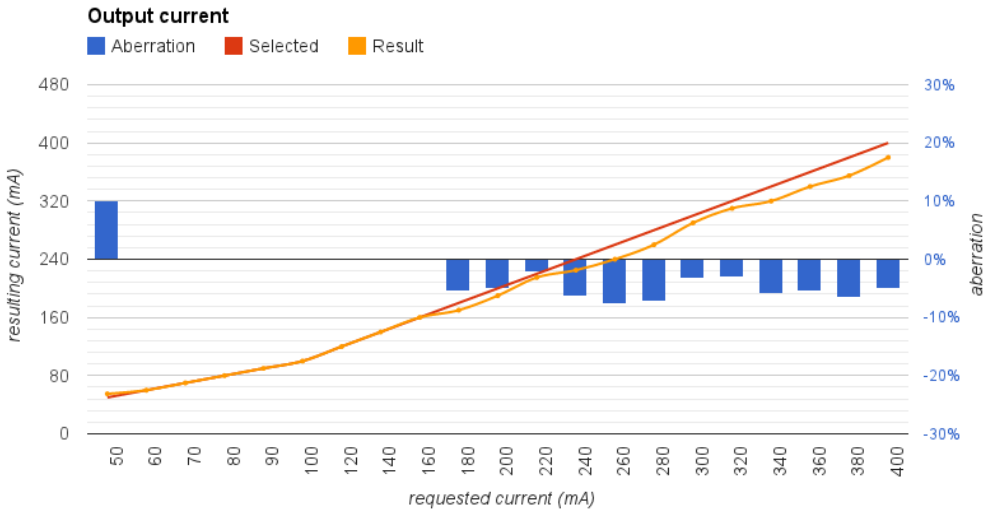


Figure 5.1: Current output measurements

Figure 5.1 shows that the current can be defined very precisely over the whole available output range (50mA 400maA). In the higher range there is an aberration between selected current and output current of maximal -8%.

This behaviour could be improved further by adopting the lookup-table that is used to calculate the resistor value for current limiting (see section A.4).

Output Voltage

To verify the correct behaviour of the switched-mode power supply the controller board was driven at different load conditions while measuring the board supply voltage, the output voltage and the power good signal of the switching regulator.

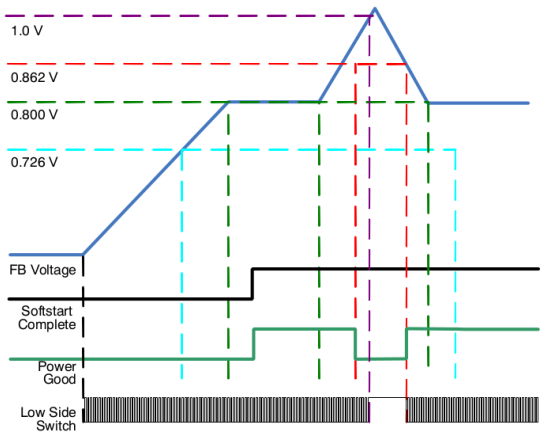


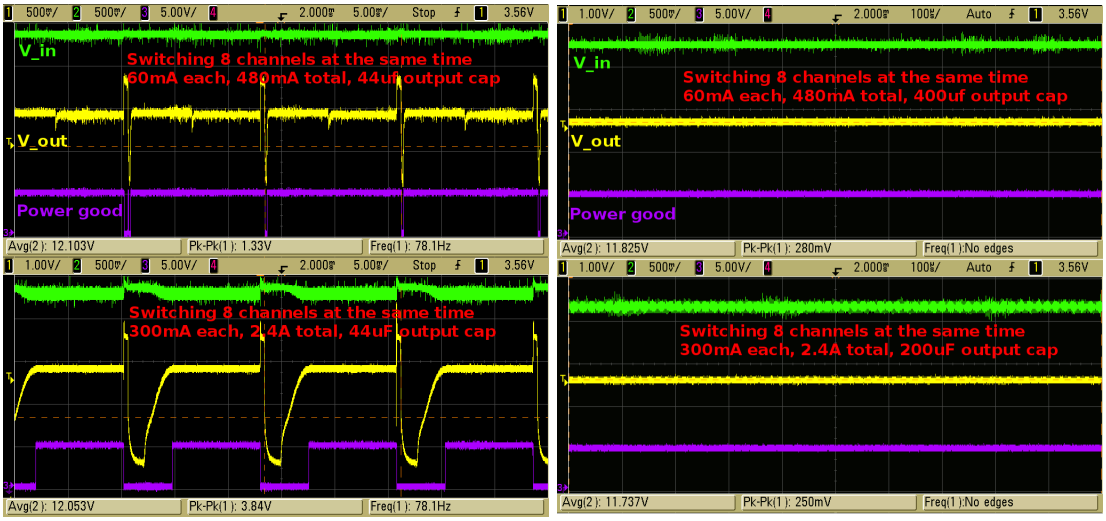
Figure 5.2: Over voltage detection of switching regulator

The measurements showed that switching LEDs on or off causes voltage spikes on the LED power supply line. These spikes occur the software PWM freqency. When multiple LEDs are switched at the same time these voltage spikes add up and can cause a voltage dropout (see figure 5.3).

The reason for this dropout is the over-/under voltage protection circuit of the switching regulator. When the voltage limits are exceeded the regulator indicates the error condition on an external pin (*Power Good*) and the regulator stops its operation until the feedback voltage has fallen beneath the over voltage threshold, as shown in figure 5.2.

This problem can be fixed by increasing the value of the output capacitors of the switched-mode power supply circuit as figure 5.3b shows.

If the problem had been detected before the second PCB revision was designed, it would have been possible to add decoupling capacitors close to the LED sockets to prevent the voltage spike from reaching the switching regulator.



(a) 44uF output capacitance (b) 200uF output capacitance

Figure 5.3: Output voltage measurements

5.1.2 Thermal Behaviour

Using a *FLUKE Ti27*, the thermal behaviour of the controller board and connected LEDs was analyzed under different load conditions. Each test scenario was executed until the temperature of the components at each test point stabilized.

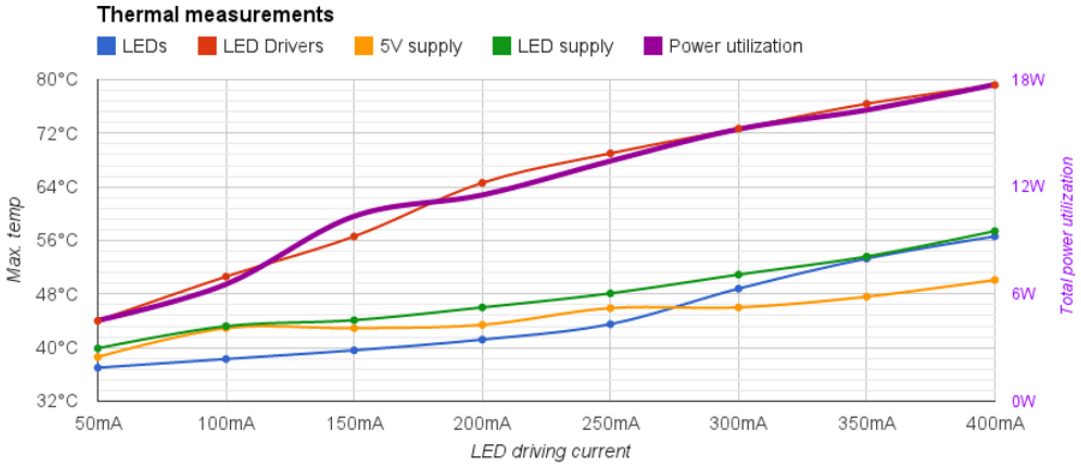
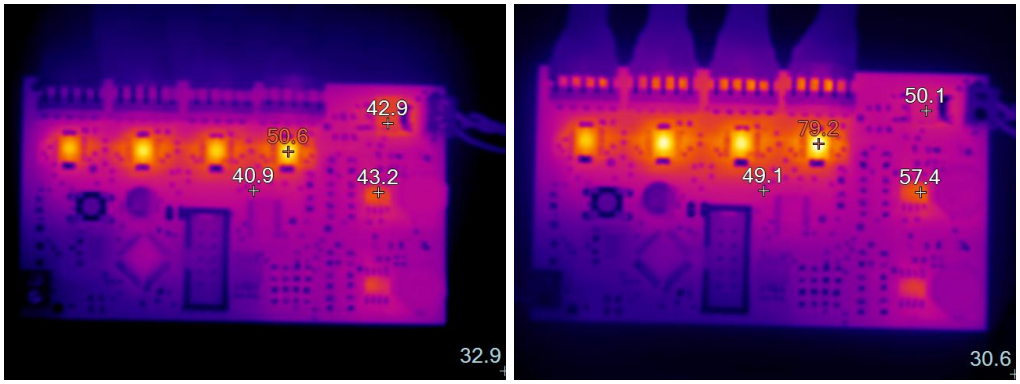


Figure 5.4: Thermal behaviour at different load conditions

To maximize the load on the board all channels of the drivers were set to a PWM output value of 255 (always on) during the test-runs. The purple line shows the total system power consumption, based on the current drawn from the bench power supply.



(a) driving 12*100mA (b) driving 12*400mA

Figure 5.5: IR Controller-board temperature measurements

Figure 5.4 shows the maximal temperatures of connected LEDs (with a 1W heatsink), LED drivers and the power supplies.

The testing revealed, that additional heatsinks or active cooling may be required for the LED-Drivers if the board is to be used at a high load in an enclosing casing that prevents natural air convection.

It can be seen that the LED-Drivers are the components most affected by an increase of the LED-current. To minimize the power they have to dissipate, the voltage of the switched-mode power supplies can be adapted precisely to the required forward voltage of the LEDs connected to the system.

It has to be noted that the tests were performed without any heatsinks on the ICs or active cooling of the boards. This means that the measurements represent the worst case scenario (except for operating the board in a tight case), and leaves much room for optimization.

5.2 I2C Interface

5.2.1 Update rate

The I^2C interface was tested extensively to find a good operation point that takes the tradeoff between PWM resolution (system load) and the possible value update rate (I^2C data throughput) into account. The tradeoff between the system load and the possible I^2C throughput has been discussed in section 4.2.1.

To test the possible sustainable update rate for controllers, a test program was written that streams value updates to a controller at a variable update rate. In each test run, a controller is driven with a range of increasing update rates and the actual data through-

put, error rate and maximal update rate is measured. The test results are printed into CSV file and used to create the result diagrams.

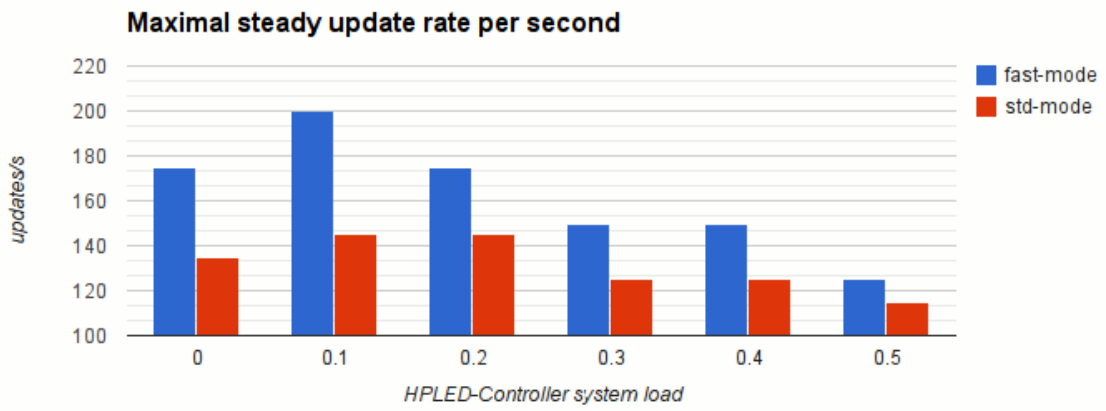


Figure 5.6: Maximal steady controller update rate

The test is performed under the constraint that the interval between the value updates has to be constant. When one update operation exceeds its period, the test run is terminated. This is done to guarantee that the system can handle a constant frequency of value updates.

This test was performed for different system load conditions on the controller board (0% . . . 50%) and for different I^2C bus speeds (standard-mode: 100kbit/s and fast-mode: 400kbit/s)

The results show (see figure 5.6) that the system can support a maximal update rate of more than 100 updates/s even if the system is under high load. This rate is more than sufficient to generate dynamic light output over the I2C interface.

5.2.2 Maximal data throughput

To measure the maximal data throughput, the constraint that value updates have to happen with a fixed update frequency is dropped. The measurements are done for different bus speeds and load conditions on the controller boards.

Figure 5.7 shows that the maximal data throughput is considerably higher than the throughput at a fixed frequency. This can be explained with the I^2C bus stretching feature, which allows the slave to delay the transmission if it is not able to keep up with the speed of the master. These delays are very short and only happen under special conditions but still long enough to prevent a 100% constant update rate.

There are two conditions under which bus streching is required in the controller:

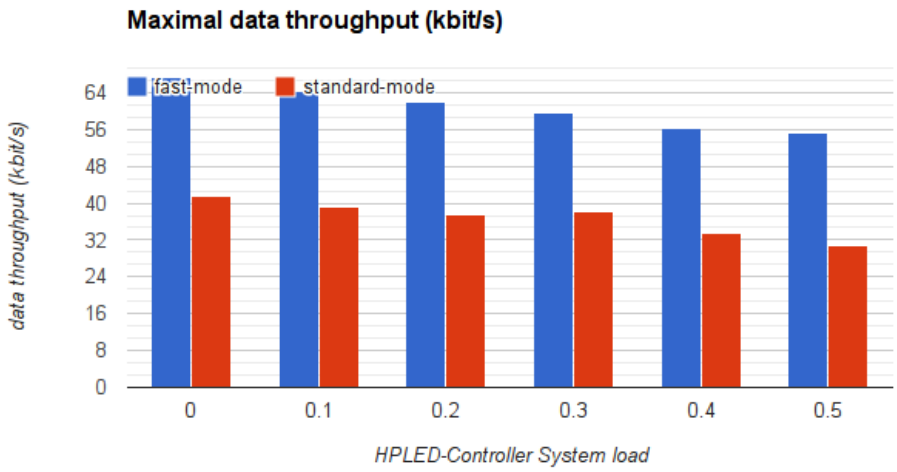


Figure 5.7: Maximal throughput of I2C interface

1. Interference between the I^{2}C interrupt and the Timer1 interrupt. Execution of the interrupt handler takes slightly longer than the transmission time of one byte over I^{2}C.
2. Prolonged execution time of I^{2}C interrupt handler in case a transmission is complete because data needs to be copied from the temporal buffer into the receive buffer

5.2.3 Maximal I2C bus speed

The I^{2}C bus specifies a range of common bus speeds for communication. The Broadcom chip used by the Raspberry Pi to drive the bus is ”only” a fast-mode controller, but allows to set the frequency manually and above the 400kHz specification.

- low-speed-mode: 10kbit/s
- standard-mode: 100kbit/s
- fast-mode: 400kbit/s
- fast-mode-plus: 1Mbit/s

Testing revealed, that the bus speed can be increased to up to 600kbit/s without affecting the system’s overall stability. Increasing the bus speed further leads to unresponsive slave devices, that need to be reset in order to make them accessible again.

This indicates that the ATMega328p TWI[1] hardware cannot handle frequencies higher than 400kbit/s reliably. For this reason the system uses the I^{2}C bus in fast-mode.

[1] Atmels terminology for the Philips I^{2}C compatible bus on their microcontrollers

5.3 Web-API

To test the correct behaviour of the RESTful web-API, an extensive testbench was written that verifies the correct behaviour of the web-API for valid data input and the stability and fault tolerance in case of erroneous data input. It is based on QUnitjs[2],

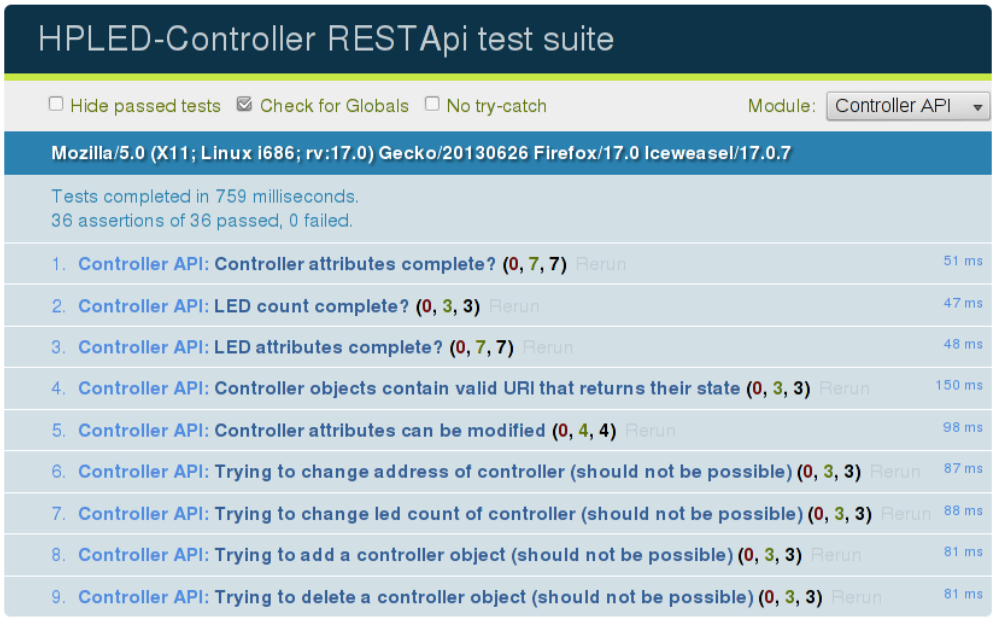


Figure 5.8: Testbench results: Controller web-API testing

which is *a powerful, easy-to-use JavaScript unit testing framework*. It makes it possible to group unit tests together to be able to run module tests independently, has support for testing of synchronous and asynchronous callback functions and can be used from the command line for continuous integration.

As the testbench interacts with the HPLED-controllers through the web-API at least one board needs to be connected to the system to be able to execute the test bench. A software module that simulates the behaviour of the controllers, when no real controllers are available. In either case, the testbench will restore the old system state after it finishes execution.

The testbench was divided into two test modules, to be able to test the two web-API URIs (`http://.../controller` and `http://.../led-set`) independently. The code for the testbench is too long for the appendix and can be found in the online repository: unit_tests.js.

[2] `http://qunitjs.com`

Figure 5.9: Testbench results: LED-Set web-API testing

Chapter 6

Criticisms and Future Work

6.1 Criticisms

6.1.1 Voltage/Current Spikes

Testing revealed that switching LEDs introduces small voltage spikes on the LED voltage supply line. If multiple LEDs are switched at the same time these spikes add up and can cause the switched-mode power supply to detect an over voltage condition and drop out until the voltage has normalized.

This can have a number of negative impacts on the system, whose significance depends on the duration of the drop outs, such as:

1. Audible noise introduced by the sudden charging of the output capacitors because the PWM frequency lies in the audible range (65-100Hz)
2. Visible flicker of the LEDs caused by long voltage drops. This limits the possible operation range of the controller board.
3. Limited use of *power good* signals to supervise the power supplies, since the dropouts happen during normal operation (just too short to be perceptible)

It is possible to prevent such behaviours by installing output capacitors with a significantly larger value ($\geq 140\mu F$) for the switched-mode power supplies. But a better solution would be to include a bypass capacitor close to the LED ports to deal with the voltage spikes before they can enter the system.

6.1.2 Luminous Flux of RGB HPLEDs

RGB HPLEDs with a luminous flux competitive to white HPLEDs ($\geq 200 lm W^{-1}$) have not yet had their mass market breakthrough. The market leaders in the HPLED sector have RGB HPLED modules in their portfolio but the prices are still very high[1].

[1] CREE MC-E RGBW Emitter 3W: 16.90€

This makes it very expensive to use the system for home illumination, when it is supposed to replace the current light installation. Recently introduced and announced consumer products in the RGB home illumination field have shown that there is a market interest. This can be seen as a promising sign that the prices for RGB HPELEDs will drop significantly in the near future.

More reasonably priced RGB HPLEDs are available (2.5€...6.90€), but they do not use the latest LED technology which results in a lower luminous efficacy ($30 \ldots 50 lmW^{-1}$). They are therefore well suited for being used alongside classical solutions in home illumination but not as a full replacement.

6.2 Future Work

6.2.1 Wireless alternative to I²C

As mentioned in the design reasoning section 2.3.1, a wireless connection based on the 802.15.4 standard between controller boards and the coordinator was considered during the design of the system, but dropped due to time constraints and financial reasons.

One can reasonably expect that microcontrollers with built-in wireless interface will be more affordable in future. A future controller revision could be improved by using a microcontroller includes a wireless interface and uses this as an alternative to the I^2C interface.

6.2.2 Security issues

The system has not taken security issues into account as of yet, as it was not thought to be strictly necessary at this time, where the project is only used in a laboratory environment. Safety features can however, be incorporated in the software ate a late time.

By supporting user-authentication to the Web-Api and in the user interface the system would become more applicable in real-world scenarios.

6.2.3 Interface for creating custom LED objects

Currently the object hierarchy used to represent the controller boards is based on the assumption that the system is mainly used for RGB HPLEDs. For this reason *RGBLed* objects represents each LED driver, because each LED driver is responsible for one RGB LED.

Over the course of the project it became apparent that there are other interesting ways to use the system, e.g. for driving single coloured LEDs or for controlling LEDs that can handle a driving current of $\geq 400mA$.

The existing object abstraction makes it possible to control each driver channel individually thereby allowing to use these kind of LEDs, but the abstraction is suboptimal, particularly in the user interface.

To make the system API more explicit the LED drivers should be represented as what they are, and a new class should be introduced for LEDs. This class is a container element that can hold an arbitrary number of channels from different LED drivers, and has additional attributes like LED type (single colour, RGB, ...) and current limit.

Chapter 7

Conclusions

7.1 Results

This thesis set out to design a system for controlling HPLEDs that provides a user interface to access and modify the system's state dynamically. Over the course of this project a complete Web-enabled system was designed, tested and implemented that has not only fulfilled these goals but exceeded them.

A distributed system that consists of HPLED-Controller boards and a coordinator with Wi-Fi capabilities and a Web-based user interface was designed. This design decision was made with the intention for the system to be scalable and economical.

HPLED-Controller boards do not need their own Wi-Fi module. This keeps their component cost low (22£), while still offering Wi-Fi capabilities by exploiting the Wi-Fi module of the coordinator.

The modular system design also makes it possible to scale up the system to support a large number of LEDs because HPLED-Controller boards can be added to the system at any time.

The system can be controlled from a wide range of platforms using the Web-based user interface. This UI allows the user to define the colour, brightness and current limit of connected LEDs.

The UI can be adapted to different applications and system configurations dynamically, by allowing the user to group LEDs from different controllers into sets which offer additional control possibilities to the user.

A dynamic Web-API makes it easy to write custom applications when the user interface is not sufficient to control the desired application. The system design makes it possible to re-use the submodules of the system in different embedded projects.

7.1.1 Software

It has been shown that RESTful APIs can be used to expose the state of an embedded system. They can be interfaced easily and allow the system to make data available in different formats which are suited for direct machine to machine interaction across system boundaries.

The Web-based user interface demonstrates that new technologies make it possible to provide a responsive and touch-optimized user interface that looks and feels like a native application while being cross-platform compatible. This makes Web-based UIs an interesting option for embedded systems because it means that no application is required for interaction with the system. It can easily be configured to be accessible from anywhere and designers do not have to worry about device compatibility of the interface.

7.1.2 Hardware

The Raspberry Pi is very well suited to perform as a gateway between embedded systems and the web. It is cheap, reliable and easy to develop software for, because it runs a Linux operating system with access to the whole Debian software repository.

The I^2C Bus interface for controller boards has proven to be capable of supporting up to 5 controller boards without interference and fast enough to enable dynamic control of a large number of HPLEDs.

The on-board switched-mode power supplies are capable of driving every LED channel with full output power, which means up to 12W per supply. While the verification has shown that the power supplies drop out under certain load conditions, it is not caused by a faulty design of the power supplies but rather by voltage spikes introduced by switching LEDs on and off (see section 5.1.1 for testing results and section 6.1.1 for evaluation).

A digital resistor was successfully used to replace an external resistor that defines the output current of each LED driver. This makes it possible to control the output current remotely and dynamically via the system API.

7.1.3 Exemplary System Use

The subsystems of the complete system have been verified and extensively tested. I wanted to show that they can work together in an application and work as one unit.

To display the system's capabilities in an integrated application, a 4x3 RGB-HPLED matrix was set up. It includes a coordinator, 3 controller boards and 12 3W RGB-LEDs as shown in figure 7.1.

All connected LEDs were added to an LED-set in order to control the matrix. Figures 7.2 and 7.3 show how the Web-interface adapts to the number of LEDs.

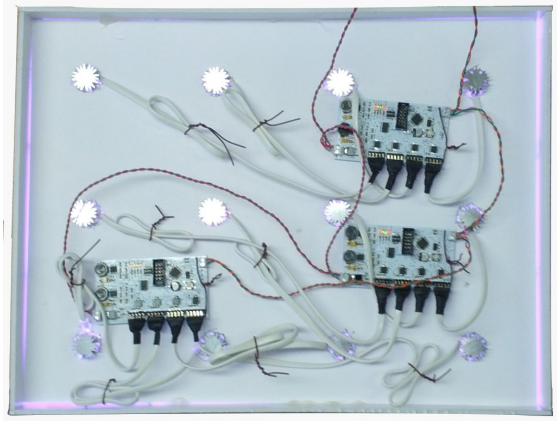
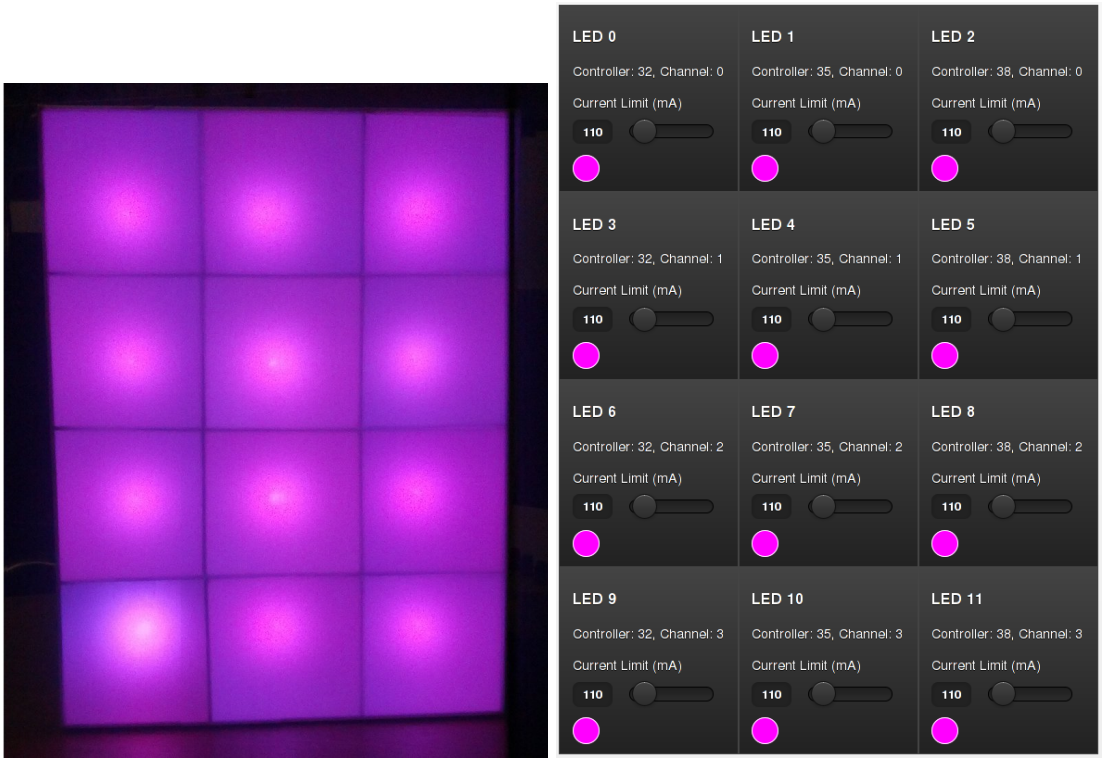


Figure 7.1: RGB-LED matrix



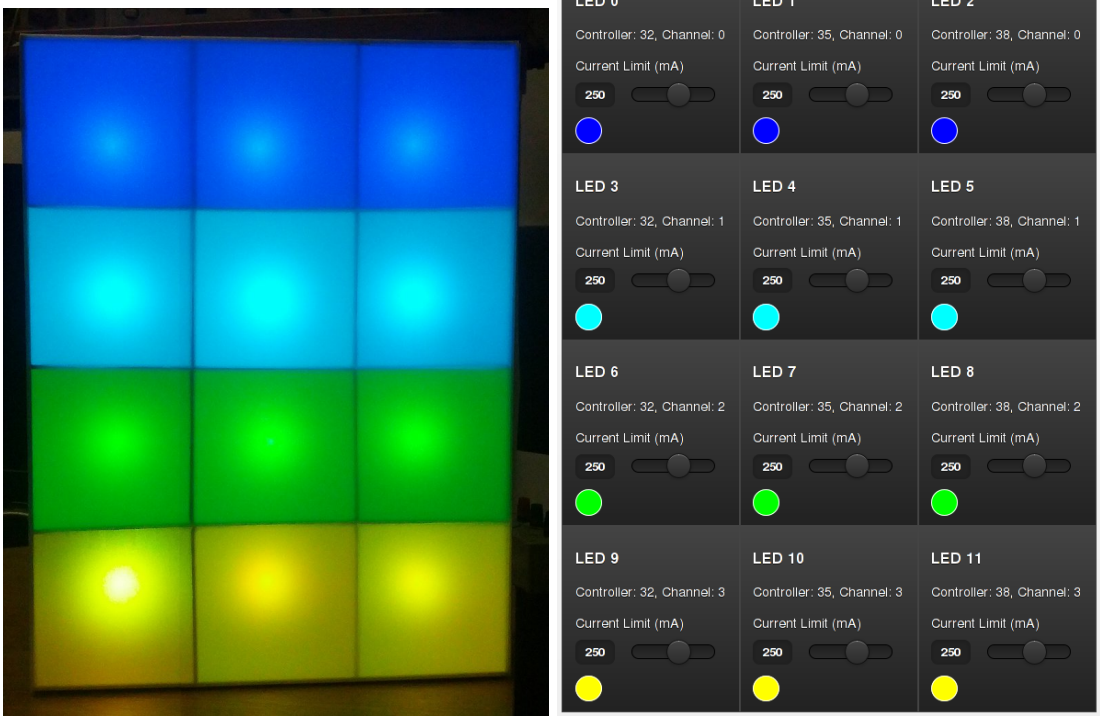
(a) LED Matrix, single color (b) UI for 4x3 LED-Matrix

Figure 7.2: RGB-LED matrix, single colour

The screenshots show that the user interface is suited to control LED-sets, containing a large number of LEDs and that the system is capable of generating colours with little discrepancy between the desired colour and the colour that is actually generated. It is

even possible to use the UI to start a periodic function that animates the matrix and makes it change colours smoothly over time.

The RGB-LED matrix application shows that the system can support a large number of LEDs and control them in a dynamic way using the Web-interface. The system design has been shown to be reliable and making it possible to able to adapt to different design requirements and constraints.



(a) LED Matrix, different output colours (b) UI for 4x3 LED-Matrix

Figure 7.3: RGB-LED matrix, multi colour

Appendix A

Appendix

A.1 Project Proposal

High power LED controller

Master's Thesis proposal

Konke Radlow

January 12, 2013

Summary

The goal of the project is to design and implement an I^2C configurable controller board for driving up to 12 high power LEDs.

In addition a scaleable software solution for a control station will be developed, that handles the communication with multiple controller boards and offers a web service API or an web interface for remote configuration and control of the boards, e.g via an Android application.

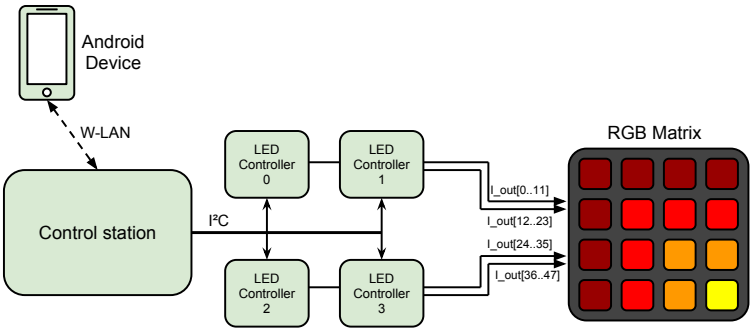


Figure 1: System overview - 4x4 RGB Matrix example project

The controller board features a simple micro-controller responsible for receiving commands via an I^2C slave interface, and controlling the peripherals on the board to drive the LEDs to the requested values.

A control station with a custom software solution bridges the gap between an arbitrary number of controller boards and the end-user by offering a web service API or webinterface for defining the current configuration of the controller boards and setting the output values.

A 4x4 RGB LED matrix will be used as an example project for the system, displaying various features of the system that can be controlled from an Android device using a simple GUI.

Engineering Challenges

- Design a custom PCB for LED controller boards
- Define and implement an I^2C protocol for the communication
- Design control software for peripherals on controller board
- Setup control station that bridges I^2C to Internet
- Scaleable software solution for control station
  - Unifies multiple controller boards under one common interface
  - Offers web service API or web interface for remote control
- Build example project using multiple controller boards
  - Remote controlled 4x4 High Power RGB Matrix

Motivation

High power RGB LEDs can be used in a wide range of applications, because they emit enough light to replace light bulbs and neon lamps while offering an extra benefit because they can emit arbitrary colored light with a variable intensity. Additionally they have a very long life expectancy and a higher power efficiency than most commonly used light sources.

What's missing is a cheap and simple way to drive and control a larger number of these LEDs. The goal of this project is to develop such a controller module on a custom PCB, in order to lower the barrier for using high power (RGB) LEDs in different projects.

A.2 GPIO Pin-Mapping

See tables A.1, A.2, A.3

Port Pin	Pin	Name: Arduino	Name: datasheet	Name: schematic	Connected to	Description
PB7	8		XTAL2		crystal Oscillator	used for crystal Oscillator
PB6	7		XTAL1		crystal Oscillator	used for crystal Oscillator
PB5	17	13	SCK	SCK	CLK of AD5204 & STP04CM05, ISP	drives clk line of AD5204 & STP04CM05, used for ISP
PB4	16	12	MISO	MISO	ISP	used for ISP
PB3	15	11	MOSI	MOSI	SDI of AD5204 & STP04CM05, ISP	drives sdata line of AD5204 & STP04CM05, used for ISP
PB2	14	10	/SS, OC1B	-		
PB1	13	9	OC1A	LE-LED	LE of STP04CM05	software generated LE signal, HIGH while shifting new values to the shift register
PB0	12	8	ICP1	/CS-AD	/CS of AD5204	enables value input for AD5204

Table A.1: ATMega328: Port B pin mapping

Port Pin	**Pin**	**Name: Arduino**	**Name: datasheet**	**Name: schematic**	**Connected to**	**Description**
PC6	29		/RESET	/RESET	ISP, reset switch	
PC5	28	A5	SCL	SCL	I2C connector	external I2C bus interface
PC4	27	A4	SDA	SDA	I2C connector	external I2C bus interface
PC3	26	A3	ADC3		I^2C Addr bit 3	using a 4pin dip switch to set the lowest 4 bits of the I^2C slave address in hardware
PC2	25	A2	ADC2		I^2C Addr bit 2	
PC1	24	A1	ADC1		I^2C Addr bit 1	
PC0	23	A0	ADC0		I^2C Addr bit 0	

Table A.2: ATMega328: Port C pin mapping

Port Pin	**Pin**	**Name: Arduino**	**Name: datasheet**	**Name: schematic**	**Connected to**	**Description**
PD7	11	7	AIN1	DEBUG_LED		digitally controlled status/debug led
PD6	10	6	AIN0	EN_PWR	EN of NCP3170	enables both LED power supplies
PD5	9	5	T1	/OE-LED	/OE of STP04CM05	outputs a PWM signal (Timer0 output compare) that is used to modify the brightness of the LEDs
PD4	2	4	XCK			
PD3	1	3	INT1	PG_PWR_CH2	PG of NCP3170	"power good" signal from the LED power supply, HIGH when output voltage is within accepted range, LOW when power is too low or too high
PD2	32	2	INT0	PG_PWR_CH1	PG of NCP3170	see above
PD1	31	1	TXD	USPI_MOSI	Serial connector	can either be used to communicate with the board over a serial connection or to add external modules
PD0	30	0	RXD	USPI_MISO	Serial connector	see above_SCK

Table A.3: ATMega328: Port D pin mapping

A.3 Bill of Material

Qty	Value	Device	Parts	Farnell code	Price per unit
1	Reset Switch	10-XX	S1	176432	0.36
4		LEDCHIP-LED0805	LED_GRN, LED_GRN1, LED_GRN2, LED_RD1	2099236	0.05
4		PINHD-1X5	LED0, LED1, LED2, LED3	1593461	0.38
4		LED_CONNECTORS		1593429	0.096
1		PINHD-2X4	JP1	1022233	0.39
1		PINHD-2X5	ISP	1099254	0.29
10	0.1uF	C-EUC0805	C2, C6, C7, C8, C9, C10, C11, C12, C13, C20	1759143RL	0.007
1	0.33uF	C-EUC0805	C1	1828951	0.103
5	0R	R-EU_R0805	R12, R14, R15, R16, R20	1469846	0.013
8	1k	R-EU_R0805	R4, R8, R17, R18, R21, R22, R_F, R_F1	1500663	0.015
2	4.75k	R-EU_R0805	R_C, R_C1	2057623	0.017
2	6.8uH	INDUCTOR	L_C, L_C1	1612720RL	0.54
2	7.68k	R-EU_R0805	R2, R7	1575888	0.041
4	4.7R	R-EU_R1206	R5, R11, R23, R25	1865242	0.025
4	7.5R	R-EU_R1206	R9, R13, R24, R26	1894409	0.084
7	10k	R-EU_R0805	R3, R10, R19, R27, R28, R29, R30	2057719	0.017
2	12nF	C-EUC0805	C_C, C_C1	1759248	0.007
1	16MHZ		Q1	1841946	0.77
2	22pF	C-EUC0805	C17, C18	1759195	0.008
4	22uF	C-EUC1206	C4, C5, C14, C15	1457420	0.26
2	22uF	C-EUC1206	C3, C19	1797010	0.84
2	28.7k	R-EU_R0805	R1, R6	2117316	0.198
1	100uF	CPOL-EUD	C16	9695672	0.23
2	430pF	C-EUC0805	C_F, C_F1	1759215	0.017
1	7805DT	7805DT	IC1	2142988	0.81
1	AD5204	AD5204BRU10	IC2	997348	2.72
1	ATMEG	ATMEGA328_SMT	U1	1715486	2.23
2	CONNE	CONNECTOR(2P-3.5 MM)	I2C_IN	1217302	0.28
2	NCP317	NCP3170	U$1, U$2	1924872	0.78
3	STP04C	STP04CM05XTTR	IC3, IC4, IC5	1880319	0.97
1	SWS00	SWS0041.27MM	S2	1605474	0.96
			Total (GBP)		21.183

Figure A.1: Bill of Material

A.4 Interpreting the current limit

The current limit for the LED drivers is defined using a digital potentiometer. This potentiometer has a range of $10k\Omega$ divided into 256 steps (see section 3.1.4). Only a small range of the available resistor values is relevant for the LED drivers. The calculation from digital potentiometer input value to resulting LED current is not straightforward, because the $I_{led}(R_{ext})$ function is nonlinear, as figure A.2 shows.

The user cannot be expected to have this deep knowledge of the system. Hence the API accepts actual current values between 50mA and 400mA. The conversion to the input value for the digital potentiometer is done on the fly using a lookup table based on values taken from the datasheet (fig. A.2) as listing A.1 shows.

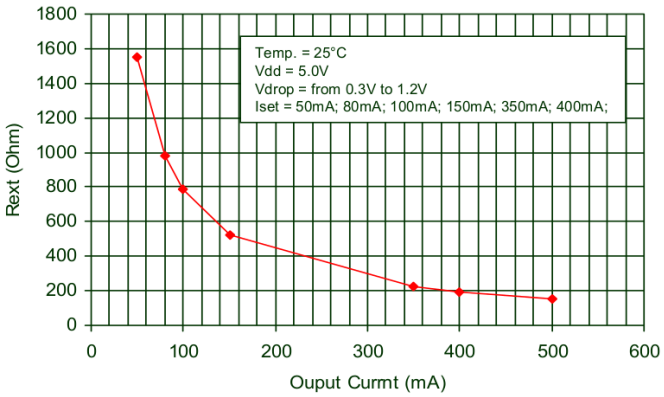


Figure A.2: Output current based on external resistor

The value returned from the `interpolate_resistor_value(i_led)` function is used to calculate the actual input value for the digital resistor as seen in listing A.2

A.5 Schematics

The schematics of the PCB were designed in Cadsoft's Eagle. The source files can be found in the online repository of the program:
https://github.com/koradlow/rgb-driver/tree/master/schematic

A printed version of the schematics can be seen in figure A.3.

A.6 Source Code

The source code of the program is available in the project's online repository:

- RGB-Controller (PCB software):
  https://github.com/koradlow/rgb-driver/tree/master/rgb-pcb
- Coordinator (Raspberry software):
  https://github.com/koradlow/rgb-driver/tree/master/i2c_raspberry
- Web-interface:
  https://github.com/koradlow/rgb-driver/tree/master/web_interface

```python
# interpolates the resistor value for the R_ext input of an LED driver IC
# based on values taken from STP04CM05 datasheet fig. 13
# R_ext limits the current output of the IC.
def interpolate_resistor_value(i_led):
  # define a lookup table (lut) with the values from the datasheet.
  # read as: (Output current, External Resistor value)
  table = [ (50, 1550),
      (80, 950),
      (100, 800),
      (150, 500),
      (350, 220),
      (400, 200),
      (500, 180) ]
  # return with min/max values if the requested current exceeds the range
  # of the lut
  if (i_led < table[0][0]):
    return table[0][1]
  elif(i_led > table[-1][0]):
    return table[-1][1]

  # current is in range of lut
  for idx in range(len(table)-1):
    # check the lut for the correct period
    if (i_led >= table[idx][0] and i_led <= table[idx+1][0]):
      # interpolate a linear function between the two points
      delta_r_ext = (table[idx+1][1] - table[idx][1])
      delta_i_out = (table[idx+1][0] - table[idx][0])
      m = float(delta_r_ext) / delta_i_out
      n = table[idx][1] - m * table[idx][0]
      return int(m*i_led + n)
```

Listing A.1: Interpolation of external resistor value

```python
# the digital res. has a range of 10kOhm divided into 256 discrete steps.
# It is used to set the current limit of the LED driver IC
# The formula for calculating the input value is derived from the datasheet
def calculate_digital_resistor_input(r_ext):
  # D_x = (256 * (R_w+R_ab-R_wa)) / R_ab => from Ohm to digital input
  d_x = int((256*(45+10000-r_ext))/10000)
  return d_x
```

Listing A.2: Calculation of digital resistor input value

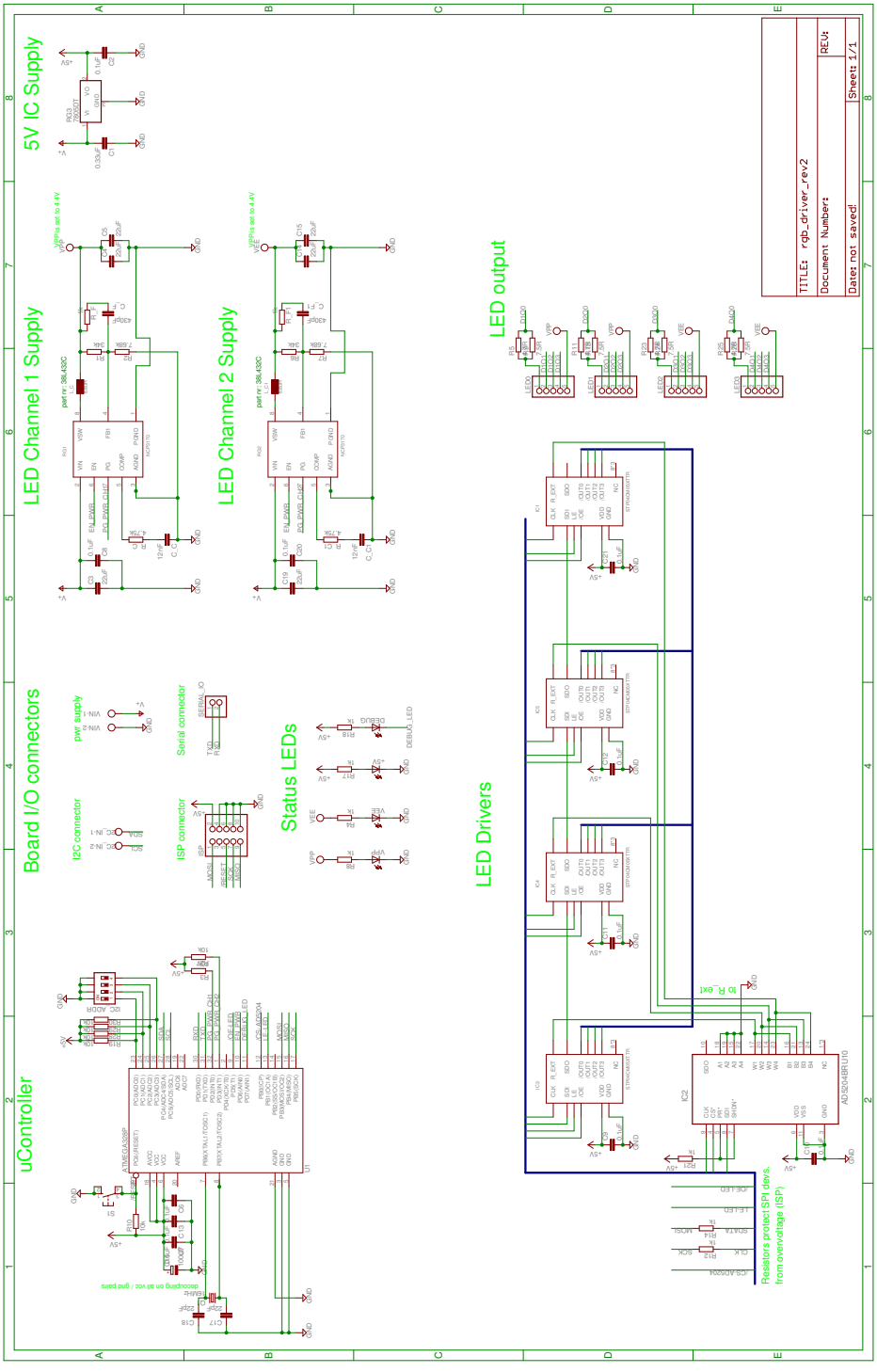


Figure A.3: HPLED-Controller Schematics

Bibliography

[Analog-Devices(2010)] Analog-Devices. *AD5204: 4-Channel Digital Potentiometer*, 2010.

[Ashton(1999)] Kevin Ashton. That 'internet of things' thing, in the real world things matter more than ideas. *RFID Journal*, 6 1999.

[AVR(2006)] AVR. *AVR136: Low-Jittter Multi-Channel Software PWM*, 2006.

[Bergh et al.(2001)Bergh, Craford, Duggal, and Haitz] Arpad Bergh, George Craford, Anil Duggal, and Roland Haitz. The promise and challenge of solid-state lighting. *Physics Today*, 54:42, 2001.

[Binstock(2012)] Andrew Binstock. Google's redifinition of the browser as platform. *Dr. Dobb's*, 7 2012.

[Broadcom(2012)] Broadcom. *BCM2835 ARM Peripherals*, 2 2012.

[Catsoulis(2005)] John Catsoulis. *Designing Embedded Hardware*. O'Reilly Media, second edition edition, 5 2005. ISBN 9780596007553.

[Chui et al.(2010)Chui, Löffler, and Michael] Michael Chui, Michael Löffler, and Roberts Michael. Mckinsey quarterly: The internet of things. 3 2010.

[Cree(2012)] Cree. Cree reaches led industry milestone with 200 lumen-per-watt led. 12 2012.

[Fielding(2000)] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[Hersent and Elloumi(2012)] Olivier Hersent and David Elloumi. *The Internet of Things: Key Applications and Protocols*. Wiley, 2 edition, 2 2012. ISBN 9781119994350.

[Kernel-Devs(2000)] Kernel-Devs. *Linux Kernel Documentation: I2C dev-interface*, 2000.

[Li(2013)] Lily Li. Why color temperature is vital to the led industry. 2013.

[Losov(1927)] Oleg Losov. Luminous carborundum detector and detection with crystals. *Telegrafiya i Telefoniya bez Provodov*, 44, 1927.

[Maniktala(2007)] Sanjaya Maniktala. *Troubleshooting Switching Power Converters: A Hands-on Guide*. Newnes, 1 edition, 9 2007. ISBN 9780750684217.

[ON-Semiconductor(2011)] ON-Semiconductor. *NCP3170: Synchronous PWM Switching Converter*, 4 2011.

[Sawhn(2010)] Kevan Sawhn. Daylight: an energy saving resource. 2010.

[Steele(2007)] Robert V Steele. The story of a new light source. *Nature photonics*, 1(1): 25–26, 2007.

[STMicroelectronics(2008)] STMicroelectronics. *AN2531 Application note, Generating multicolor light using RGB Leds*, 2008.

[STMicroelectronics(2010)] STMicroelectronics. *ST04CM05: 4-bit constant current power-LED sink driver*, 2010.