Custom domotics system for use in an auditorium

as a replacement for an existing control system

HAMK
UNIVERSITY OF APPLIED SCIENCES

Bachelor's thesis

Automation Engineering

Valkeakoski 08.06.2012

Cooreman, Steven

Vandermeersch, Jakob

HAMK
UNIVERSITY OF APPLIED SCIENCES

ABSTRACT

Valkeakoski Unit
Automation Engineering
Exchange studies

Authors	Steven Cooreman, Jakob Vandermeersch
Year	2012
Subject of Bachelor's thesis	Auditorium automation using touchscreen technology

ABSTRACT

The primary purpose of this thesis is to demonstrate an example execution of the replacement of an old, branded control system, with a custom modular implementation, based around a touchscreen computer acting as the central controller. This work was commissioned by HAMK's Valkeakoski unit, but the same principles can be applied elsewhere.

After a study of the previous situation, complicated by the need for tailoring the system to two different auditoria, a work plan has been made to design the new control system. During the development cycle, as outlined in this document, a custom protocol has been written, and applied over an industry-standard serial bus. The necessary accompanying interface hardware has also been designed and tested, giving the implementation already some extra functionality over the pre-existing one. A touch-friendly software component has also been designed, allowing for easy tailoring and extension through the use of configuration files rather than hardcoded solutions.

During the final testing phase, the whole solution was installed and tested in an auditorium, and its user-friendliness proven in the field.

Keywords touchscreen, domotics, remote control, PC interface

Pages 81 p. + appendices 10 p.

TABLE OF CONTENTS

LIST OF FIGURES

APPENDICES

PREFACE

This thesis, as a whole, is the culmination of four months' work by both authors, Steven Cooreman and Jakob Vandermeersch. It has very much been a joint effort, yet there is a distinct separation in tasks that were carried out by the respective authors.

As it stands, Jakob has proven himself to be a valuable hardware designer, and is thus responsible for all custom hardware designs presented in this work. His extensive knowledge of audiovisual equipment has also been of value during the installation process, the most of which he carried out with good results.

Steven, on the other hand, has done most of the theoretical work and software design. The bus structure and protocols as presented herein are entirely his work. Except for the infrared transmission routine, every single piece of software delivered as part of this project has been coded and tested by him. In addition, the final redaction of this thesis has rested on his shoulders.

ACKNOWLEDGEMENTS

This bachelor's thesis and accompanying project work would not have been possible without the many hours these people spent helping us:

– Antti Aimo, HAMK's automation department supervisor, who gave us this project
– Kalevi Sundqvist, the project's second supervisor
– Osmo Leinainen. Whenever we needed anything for our work, he made sure that it got to us in time.
– HAMK University of Applied Sciences, for accepting us as exchange students
– KaHo Sint-Lieven, and especially our international coordinator, Mr Patrik Debbaut, and HAMK's contact Mr Dirk Thomas, for giving us a chance to apply for an Erasmus exchange

DEDICATIONS

I, Steven Cooreman, wish to dedicate this work to my girlfriend and soul-mate Silke Houtmeyers, for her continued mental support when the barometer dropped to sub-zero. At this point I should also dedicate this to my parents for getting me where I am now, and my sister for not giving up on me.

In addition, I would like to thank my high school teachers Werner Vervoort and Guido Story for piquing my interest in the technical field, and convincing me that such a study is not less worthy nor frowned upon. Many thanks go out to my teacher and mentor Eric Carette, who is now retiring, for some invaluable lessons learned, and a great deal of inspiration.

I, Jakob Vandermeersch, wish to dedicate this work to my girlfriend Ina Maesen for her patience and pep talks during times of struggle; my parents and school for giving me the chance to study abroad. And last but not least I want to thank Steven Cooreman for all the patience he had with me during difficult times.

ABBREVIATIONS

AC	Alternating Current
ACK	Acknowledge
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BSOD	Blue Screen Of Death
DC	Direct Current
DSP	Digital Signal Processing
GUI	Graphical User Interface
IDE	Integrated Development Environment
JIT	Just In Time
KISS	Keep It Simple and Stupid
LSB	Least Significant Bit
MSB	Most Significant Bit
NAK	Not Acknowledged
PCB	Printed Circuit Board
PDM	Pulse Distance Modulation
PWM	Pulse-Width Modulation
RMS	Root Mean Square
RTOS	Real-Time Operating System
TTL	Transistor-Transistor Logic
UI	User Interface
UX	User eXperience
USART	Universal Serial Asynchronous Receiver and Transmitter
XAML	eXtensible Application Markup Language
XML	eXtensible Markup Language

1 INTRODUCTION

The goal of this combined project and thesis is to design and develop a custom tailored control system for use in both of HAMK Valkeakoski's auditoria, as a replacement for a Crestron system, installed some fifteen years ago.

This Crestron system consisted of a central control unit containing most of the interface logic, which was mounted in a 19" rack, together with most of the audio-visual equipment of the controlled auditorium: video and audio switches, amplifiers, mixers, and so on. To control the auditorium, the Crestron central control unit uses 32 relay outputs, 2 serial (RS232) communication channels, and 4 combined infrared or serial transmission ports. That means that those ports, when used in serial mode, cannot get feedback from the controlled equipment.

The user interface of this system was a small touchscreen panel measuring five inches diagonally, for which a custom interface had been programmed, and communicated with the central unit through a vendor-specific bus structure, called 'CresNet'. During the course of several years, these touchscreens have been subjected to wear and tear, resulting in several cosmetic malfunctions, such as dead pixels, faulty display lines, and a malfunctioning backlight.

For the lighting control, a system from Finnish manufacturer Helvar had been installed, which, at least theoretically, is controlled by the Crestron control box by using relays to virtually press the scene buttons. In one of the two auditoria, this system has already been switched off due to malfunctioning, although it still works – almost – flawlessly in the other.

The control system as a whole has also seen many undocumented manual modifications to its operation, making it difficult to get a good understanding of the real situation, as opposed to the one on paper.

To improve on the situation, we have been tasked with the design, development and implementation of a brand new control system. This system is to be based around a 15" touchscreen computer, with integrated serial communications and USB ports. The use of a PC meant that both the additional required control hardware, and the interface software, had to be made by us.

In the following chapters we will clearly describe what steps we took in our development process, including an extensive overview of the equipment that has to be controlled. This leads to the final description of the realized solution, and a self-evaluation of our work.

2 PRE-EXISTING SITUATION

We started our work with the schematic capture of how both auditoria were wired up. For this, we already had some diagrams and building plans from the original installation in 1995.

To our surprise, most of the cabling of auditorium A was unstructured, and undocumented alterations had been made to the setup. The situation in auditorium B was somewhat better, but the problems would start with proprietary protocols in the installed equipment.

The basic design of both auditoria's control system is fairly similar. It consists of a central unit with integrated relay contacts, IR and serial command buses, and a vendor-specific bidirectional 'CresNet'[1] bus, on which the touchscreen on the lecturer's desk is connected. In other words, a star topology is used.

2.1 Connection diagrams

In the following section, we will illustrate the topology of each auditorium's initial state using some diagrams.

2.1.1 Auditorium A

Auditorium A is older, and the most altered from the original wiring diagrams. There are a few special and strange design choices in the topology, outlined below.

- The only volume control present is controlling the general volume. That means that there is no option to separately control the volume of the external sources, the microphones, or the PC, directly from the touchscreen. Seeing that the external sources are connected on XLR and RCA busses, and are expected to be mic-level, and not line-level, this is a rather curious thing.
- The video sources are switched twice: In the front of the auditorium, a VGA switch is used to switch between the PC and laptop video signal. That switch, curiously, has no audio switching capabilities. The control rack contains another switch, used to switch between the different audio/video sources.
- With regards to the audio switching/mixing: first of all, the audio is switched together with the accompanying video signal. While mixing, all channels are directly mixed with each other, which means that no single channel can be muted. That is a remarkable choice, considering a possible future addition of for example a DVD player. The audio would not mutable while showing a video, eliminating the lecturer's possibility to show a muted movie while giving his own explanation. Of course, this is only one example of a situation where equal mixing of all audio sources is not very desirable.

[1] CresNet is an RS-485 based serial bus, of which the specifications are not publicly available

– The blinds and projector screen have no feedback loops, this means that the system cannot know whether they are currently up or down. Instead, the control circuit relies on the built-in end stop, which disconnects the power from the motor if the limit of travel has been reached. In the event of a failure of said end-stop, this could lead to the motor burning out.

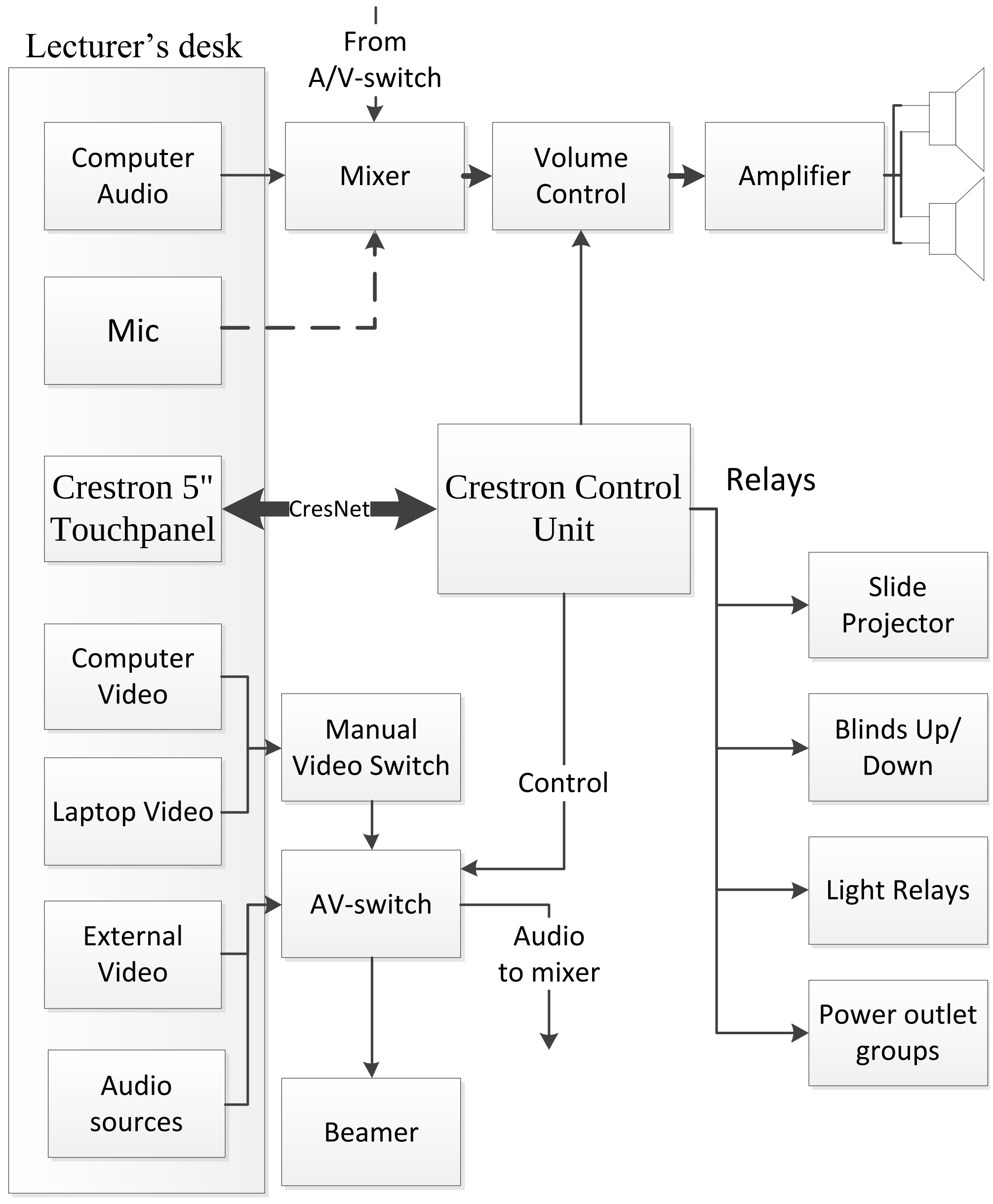


Figure 1 Auditorium A: Original situation (block diagram)

In Figure 1 we have portrayed the basic schematic outline of the original situation in auditorium A. This seems to be a very elaborate, and hence expensive setup for few capabilities. Over the course of this chapter, we will delve deeper in each unit's control specification.

2.1.2 Auditorium B

The situation in auditorium B is similar, although the room itself has a different layout, and uses more recent equipment. The wiring has also been laid out better, using numbered cables, which made our process of reversing the complete structure a lot easier.

Not unlike the situation in auditorium A, this one also has a few quirks:

- External video seems to be connected on a separate input of the beamer, which means that the user still needs the remote control to switch sources. Although the beamer does possess a control interface, it has been left unused.
- Almost all audio channels are routed through a networked DSP, which is the correct thing to do in this kind of situation. However, there is absolutely no information about which program has been loaded into the DSP, or the protocol used between the Crestron unit and the DSP. This makes our work of adapting the control system to a new PC-based much more complicated.
- DVD audio is decoded through an AV-receiver in surround sound, and there are speakers for surround, but the whole system has been cabled for stereo sound.
- All AV-sources (VCR, DVD and cassette/FM player) are controlled through small infrared emitters glued to their infrared port. For these controls, too, have all protocol definitions gone missing. Furthermore, they currently do not function anymore.
- Something has gone wrong with the implementation of the lighting system: if the lights are switched off using the Crestron touchscreen and the Crestron system is turned off, they cannot be switched back on using the button at the door. The same goes for the stair lighting.
- All lights are switched on and off by means of relay-contacts. This means that although there is a dimmable feature in the lighting structure, the lights are turned on and off simply by cutting the power supply to the lights.

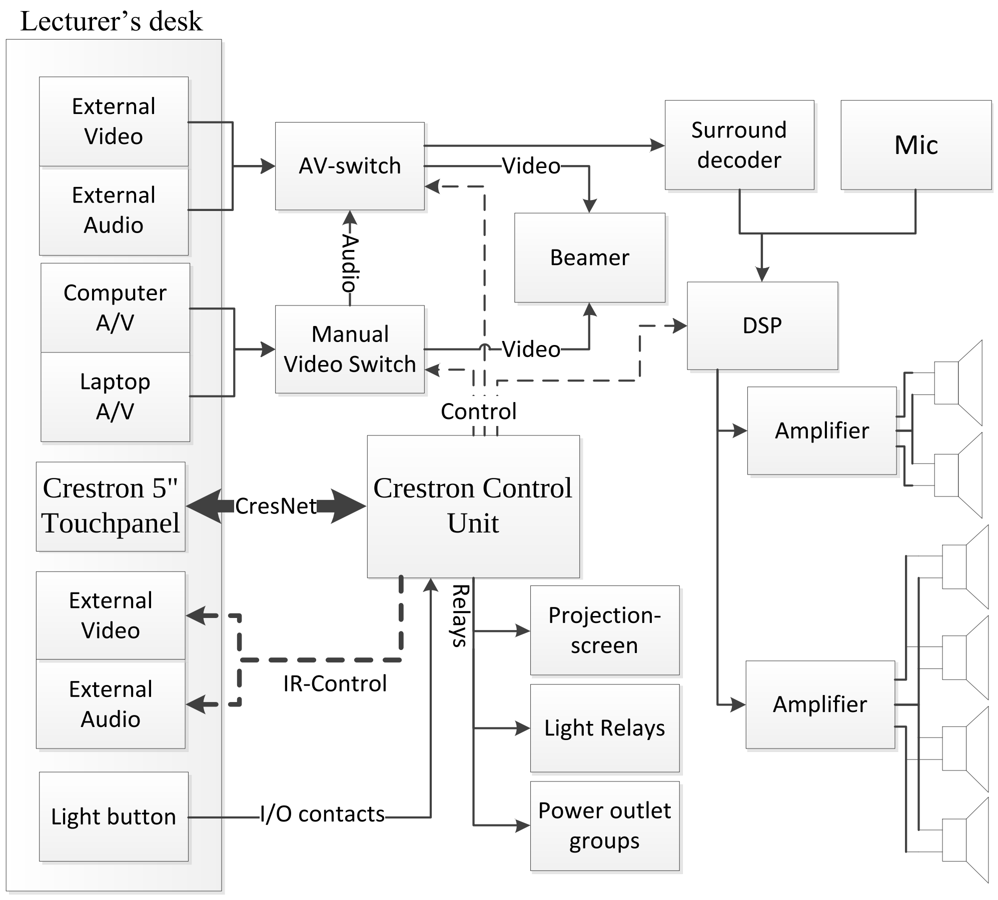


Figure 2 Auditorium B: Original Situation (block diagram)

In Figure 2 we have portrayed the basic schematic outline of the original situation in auditorium B. If the schematic is compared with the one of auditorium A, it becomes clear that the basic topology is laid out similarly in both auditoria.

2.2 Crestron central control system

Up until now, we have referred to the central control box simply as 'Crestron controller'. But what is it?

To answer that question, we must refer to the documents provided online by the manufacturer, Crestron Inc. from the USA. According to themselves, Crestron is the leading provider of control and automation systems for user interaction. Essentially, this means that they manufacture and support a whole range of products, designed to operate together on a single bus interface, to control everything that is controllable in a room, auditorium, house, theatre, and so on.

The controller that has been used in our environments is the CresNet II-MS. It is a 19 inch rack-mountable controller, which takes care of all the logic in the control system. In other words, all Crestron expansion cards and interfaces are created 'dumb'.

The specific actions and designs for this system are created using a custom programming language called 'Simpl+'. Once this program is created, it is uploaded to the controller using a serial interface. This means that even if it would be desired, one cannot recreate the original program by reading the firmware of the controller. That is why we had to reverse engineer all connections.

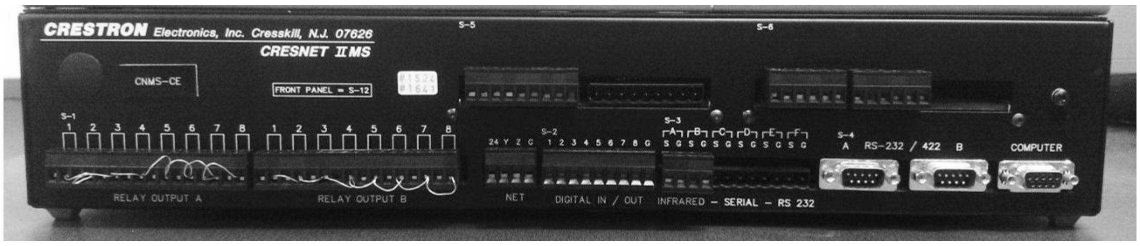


Figure 3 Back of the Crestron control unit

All of Crestron's components are interconnected on a bus system, also known as 'CresNet'. From what little information we have gathered by searching websites and some discussion groups on the internet, the bus is actually an RS-485 compliant bus. This seems to fit with the connector labelling on the unit itself: it has contacts for '+24', 'Y', 'Z' and 'GND'. All components are thus powered from a single supply rail of 24V, which is why a fairly powerful supply is also present in the rack.

RS-485 specifies a bidirectional half-duplex bus, controlled by one master, who has to poll each slave to get a response, and avoid collisions on the bus. This means that it is not plug-and-play at all, for each component must be defined in the program as well to get the controller to recognize it.

As per the 485 specification, signals 'Y' and 'Z' are each other's opposite, so that interference on the bus is minimized. On top of that, the use of twisted shielded cable is recommended by both the manufacturer and the 'official' specification.

The user almost never gets to see the central controller though. All interactions are based around a graphical user interface, displayed on a Crestron touchscreen, the PC-1500D. This is a grayscale touchscreen, with a screen diagonal of five inches. It hooks up to the CresNet bus, and is entirely passive. This means that if the bus fails, the last screen will still be displayed as long as the power is present, but that no interaction is possible.
From this, we can gather that the display data is sent from the central unit to the display whenever a screen refresh is needed, and that the coordinates of user touches are sent back to the central unit.

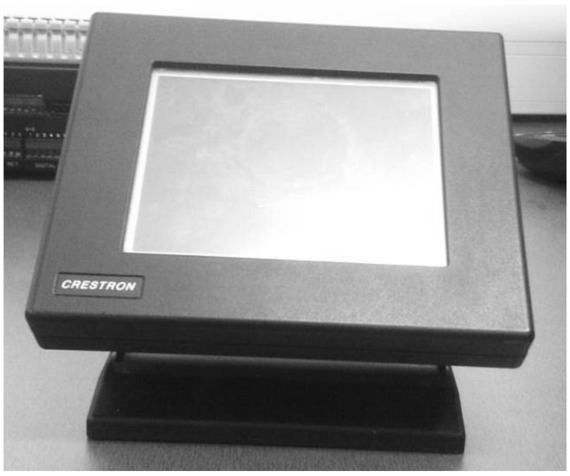


Figure 4 Touchscreen unit pulled from the Crestron control system

The program that has been written for the control system back in the nineties had several flaws: to begin with, it was unilingual in Finnish, which is not entirely appropriate for a university with an international focus, and it used a hardcoded list of buttons and labels. This meant that for each change in the infrastructure, the program had to be remade, and the original program source code has never been disclosed by the installer. Thus, expansions to the system were not really an option without a major overhaul.

With our replacement system, we plan on fixing this: the control program will have its full source code disclosed, and be structured appropriately so that eventual changes can be made by simply adjusting a line in the configuration file. For more elaborate changes, such as adding equipment, the process is documented. The program will also be multilingual in English, Finnish and Swedish.

2.3 Common controlled equipment

There is some equipment that can be found in both auditoria which has a similar way of being controlled. In order to simplify future expansions and/or the use of this solution in other auditoria, we have taken a generic approach to the problem, so that the solution can be easily adapted to other needs.

2.3.1 Integrated lighting system

Originally, both lighting systems where controlled by a Helvar system. Helvar is a Finnish producer of lighting control systems. According to their website, they are the most reliable ballast supplier and lighting control specialist in Europe.

This specific lighting system works with a central controller with four individually controllable analogue zero to ten Volt outputs. These outputs are then connected to separate dimmer units, which can be configured to act as either resistive dimmers, for use with standard incandescent bulbs, or as inductive dimmers for use with capable fluorescent tubes.

The controller also possesses a couple of relay outputs that can be configured together with the scenes. At this moment, these outputs are not in use, and it is unlikely that they ever will be.

The voltage level on each output is controlled by the user through one or more key panels. These can either be programmed in scenes with preset output levels, or as up/down keys for a specific channel. This separation is made at the time of programming. An LED on the keypad provides feedback about which scene is selected, or which light is on, depending on the operating mode.

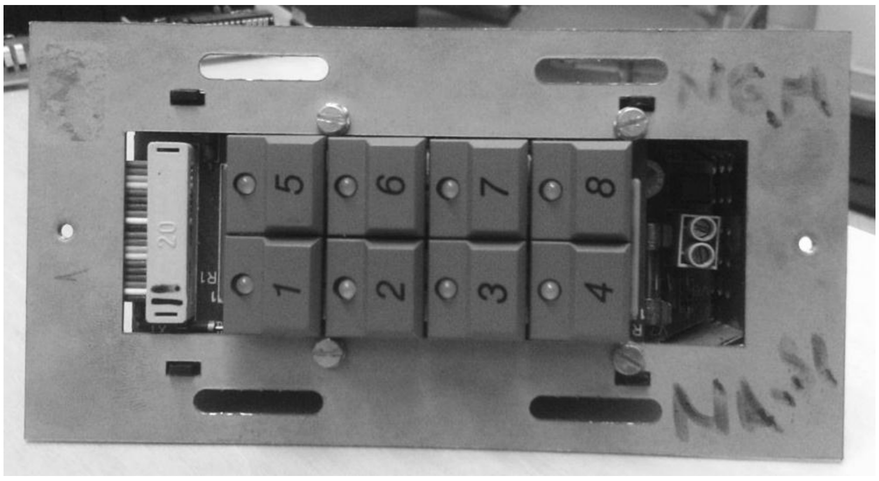


Figure 5 Helvar keypad, disconnected from the A auditorium

These key presses are received by the dimmer control trough a vendor specific bus. One peculiarity of this bus is that both power and data are supplied by only two wires, reducing the installation cost.

During a previous update of auditorium A, this system was switched off and replaced by relay contacts, in order to act as a normal light switch. Before that, though, it was controlled by a keypad at the door, so the user could switch the lights on immediately without having to go through a possibly very dark auditorium in search of the Crestron touch panel.

In auditorium B, however, the Helvar system is still in use. The system is controlled by a transfer module that interprets Crestron outputs as pushbuttons to activate certain scenes. So, more specifically: that module acts as a keypad on the keypad-bus, and its ‘keys’ are pressed by relay contacts.

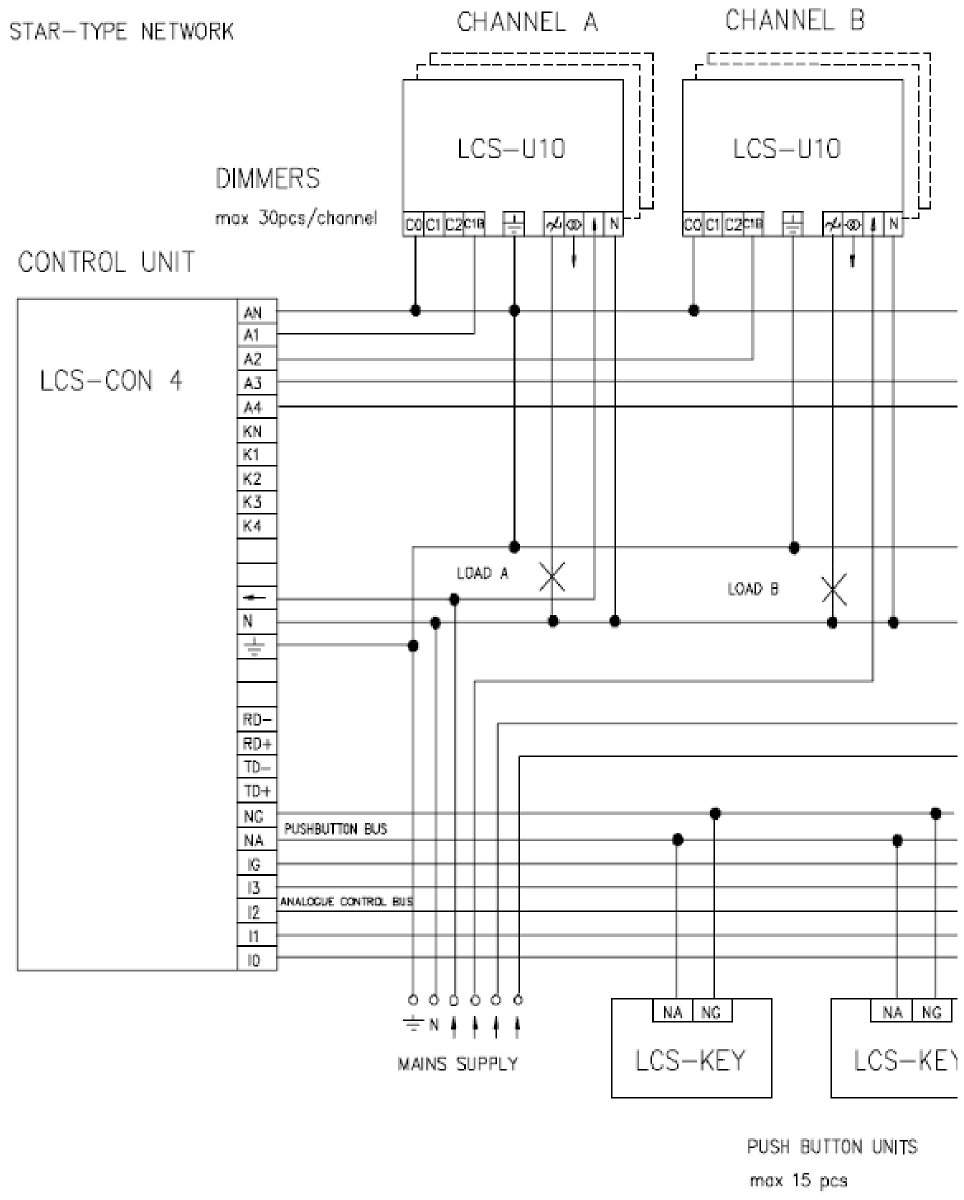


Figure 6 Connection of the lighting controller with two dimmers attached

2.3.2 Projection screen

The projection screens in both auditoria work in the same way. They are controlled by relay contacts which can switch the direction of the screen. The screen motor keeps turning until the screen reaches its limit switch. When the direction is changed on the Crestron interface, the screen moves in the other direction and keeps continuing until it reaches the other limiting switch.

The screen can manually be stopped at a point of choice by the Crestron system if wanted by the operator.

In auditorium B, there is a timer which makes the relay contact open after a certain time. This time exceeds the time needed for the screen to reach the limit-switch, and is only useful as a safety feature to prevent damage due to malfunctioning limit switches.

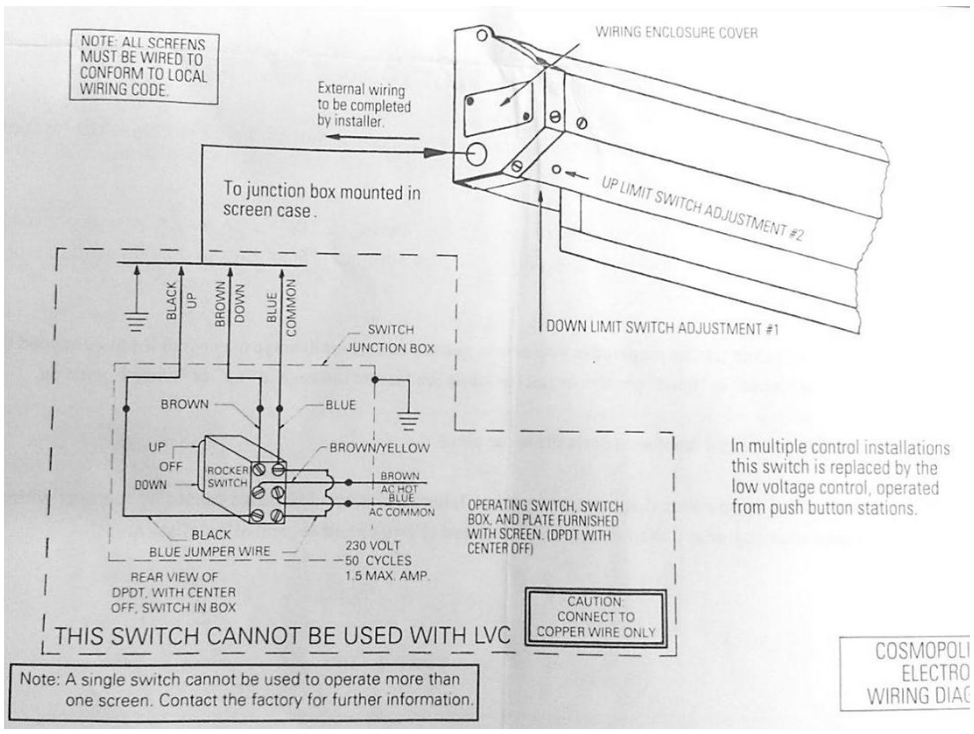

NOTE: ALL SCREENS MUST BE WIRED TO CONFORM TO LOCAL WIRING CODE.
External wiring to be completed by installer.
WIRING ENCLOSURE COVER
UP LIMIT SWITCH ADJUSTMENT #2
To junction box mounted in screen case.
DOWN LIMIT SWITCH ADJUSTMENT #1
BLACK UP
BROWN DOWN
BLUE COMMON
SWITCH JUNCTION BOX
BROWN
BLUE
UP
OFF
DOWN
ROCKER SWITCH
BROWN/YELLOW
BROWN
AC HOT
BLUE
AC COMMON
BLACK
BLUE JUMPER WIRE
230 VOLT
50 CYCLES
1.5 MAX. AMP.
OPERATING SWITCH, SWITCH BOX, AND PLATE FURNISHED WITH SCREEN. (DPDT WITH CENTER OFF)
In multiple control installations this switch is replaced by the low voltage control, operated from push button stations.
REAR VIEW OF DPDT, WITH CENTER OFF, SWITCH IN BOX
THIS SWITCH CANNOT BE USED WITH LVC
CAUTION: CONNECT TO COPPER WIRE ONLY
Note: A single switch cannot be used to operate more than one screen. Contact the factory for further information.
COSMOPOLI
ELECTRO
WIRING DIAC

Figure 7 Wiring schematic of the projection screen

As seen in the instruction manual, there are regrettably no return channels to signal the end of travel. This means that we too are limited to the implementation of timers to approximately switch off the relay when the screen has reached the desired position.

Another option would be to use the adjustable limit switches, and tailor the amount of travel to the data projector’s image. However, this would obstruct any further addition of projection equipment that might project a bigger image.

2.3.3 Audio-visual switches

There are a total of four different A/V switches throughout the two auditoria. The first three are ‘Kramer’-branded switches. These three switches are controlled by the central command unit trough a serial RS232 compliant communications link. Regrettably, that control bus can only be used to set the active channel, and not request the currently selected channel. The switches are rack-mountable with a height of two units in a 19 inch rack.

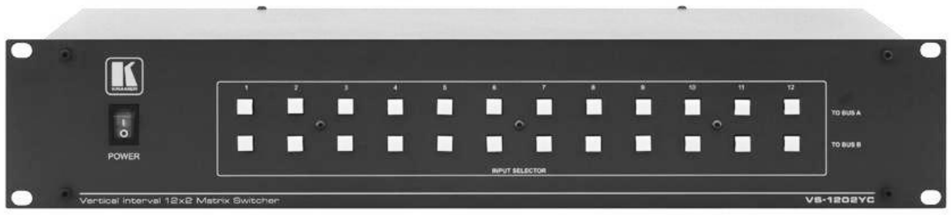


Figure 8 Kramer video switch with 12 inputs and dual selection busses (VS1202)

The other one is made by Extron, and has originally been implemented as a manually controlled switch, presumably because the switch was added to the infrastructure at a later date, and the Crestron program source code was not available to make changes. It can, however, also be controlled through a serial link, which is what we plan to do with the new system.

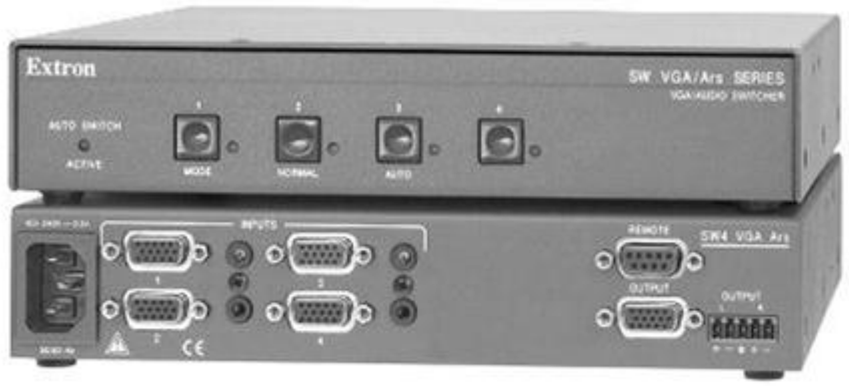


Figure 9 Extron VGArs video switch with 4 VGA inputs and 1 output.

The basic setup in both auditoria is the same: a VGA switch for multiple computer inputs and an A/V-switch for video, DVD and computer audio.

In auditorium A: A Kramer VS1202S switch is used as the A/V-switch, and an Extron VGArs SW4 is used as VGA-switch. Only the A/V-switch is remote controlled by Crestron. The second video output is a video feed for the monitor built into the lecturer’s desk. The VGA-switch has to be controlled manually at the lecturer’s desk.

In auditorium B: A Kramer VS1202YC switch is used as the A/V-switch and a Kramer VP-61 is used as the VGA-switch. In this configuration both devices are controlled by serial communication.

The KRAMER VS1202 is a 12 A/V-input and 2 A/V-output matrix. An input consists of 1 stereo sound-feed and 1 video-feed. An advantage of this A/V-switch is that output-A and output-B don't need to be connected the same channel. This means that one can have two display devices each displaying another audio-visual feed. This sort of switch has its main use in television and broadcasting studios.

The KRAMER VP61 is a 6 input and 1 output VGA/audio-switch. Just like the 12x2 switch it can be controlled both manually and by serial communication.

2.4 Auditorium-specific equipment

Each auditorium has some equipment specific for the auditorium. This means adaptations in the control system need to be made in order to successfully implement these devices.

2.4.1 Blinds

In auditorium A, blinds are used to darken the room. These blinds cover all the windows and render the auditorium pitch black when they are all the way down.

The blinds work much in the same way as the projection screen. This means they are controlled by relay switches providing the upwards and downwards motion, and that the motors are protected by limit switches.

Also much in the same way as the projection screen, these too have no return signalling to indicate that the limit of travel has been reached. To prevent motor burnout in case of a limit switch failure, we will have to implement a timer as well for the blinds.

2.4.2 Volume control

Both auditoria have a system to control the audio volume. In auditorium A, this is an expansion card integrated into the Crestron system. In auditorium B, the volume is controlled through a SoundWeb-branded networked DSP[2] manufactured by BSS, and a surround decoder.

In auditorium A, only the main volume can be adjusted. This means that if you want to mute a video, you can't use the microphones at the lecturer's desk. Because this is a big trade-off, we plan to implement more volume controls in our system, so that more inputs can be independently controlled.

The network DSP in auditorium B has eight inputs and outputs. The device can be programmed to the desired specifications, and can be controlled through a serial communications link. The DSP is not only used as

[2] DSP: Digital Signal Processing: the manipulation of an analogue signal through a digital circuit

a volume control unit but as an audio mixer as well. Because the DSP is programmable, the volume of the audio-visual devices can be adjusted separately from the microphone and main volume.
The surround decoder is used as a buffer for the stereo audio. When a DVD is played a digital audio signal is sent to the decoder. The decoder transfers the generated surround signal to the first five inputs of the DSP. The subwoofer is left unconnected.

2.4.3 Infrared commands

The Crestron system simulates an infrared remote control. Because there are multiple devices at the lecturer's desk, you would need many separate remote controls lying around on the desk, with a chance of them getting lost. By integrating the basic infrared commands into the Crestron system, and assigning an IR port to each separate device, those remote controls are made redundant.

In the implementation of auditorium B, the decision was made to put each infrared transmitter on a separate channel of the Crestron control system. While not strictly necessary, this does greatly reduce the chance of multiple devices responding to the same command. In reality, however, most manufacturers assign so-called hardware addresses to their devices, in such a way that a remote can only be used for one specific device or range of devices.

So, unless all of the devices originate from the same line and manufacturer, one could safely transmit all codes on the same channel and hook up the transmitters in series.

2.4.4 Slide projector

In auditorium A, a slide projector is available. This projector is switched on and off by means of a switchable power outlet. The remote control of the projector is simulated by Crestron-controlled relay switches which switch on and off quite rapidly, as to mimic a press on a button.

This projector isn't used anymore, probably due to the fact that the lamp bulbs have broken down, and the connection with the control system has been severed. We plan on reinstating this projector to a working condition.

2.4.5 Data projector

Both auditoria have a data projection system. These projectors have been changed during the years. This means that the serial protocol originally programmed into the Crestron command system, does not apply to the new projectors anymore. This results in the usage of the projector's own infrared remote by the lecturer.

As mentioned earlier, each projector has two inputs which are used for the moment: one coming straight from the computer sources (i.e. PC, laptop, etc.) through a VGA switch, and one coming from all the other external sources through an A/V switch. So, in order to change the active input, the remote must be used again, in conjunction with the switch’s control if necessary.

This is not a very user-friendly procedure, as the primary purpose of a command system is to facilitate all actions and reduce them to one press of the button. Therefore, we plan to implement the new serial protocols and incorporate all necessary commands into our software.

2.4.6 Switchable outlets

Each auditorium also has switchable power outlets. This means certain devices can be switched on and off by switching the mains power supplied to the devices. For example, the slide projector mentioned earlier is switched on and off by means of a switchable outlet.
The complete audio-visual system in both auditoria is also switched on and off in a similar way, to reduce unnecessary power consumption when the system is not in use.

In auditorium B, there are outlets embedded in the floor. So, if the attendees bring their own laptop, they can use the supplied outlets, only if the lecturer allows it.

2.5 Summary

Although the two auditoria are different on a lot of aspects, the same type of system is used to control all conveniences. A star topology is used in both systems with a Crestron controller as the central point. This control system is used to switch relays and control devices by means of serial or infrared communication.

3 DESIGNING A REPLACEMENT

To design a replacement for the pre-existing system, we started out with the idea to make a new control panel for the Crestron system. After some research we noticed it was very difficult to find any information about the CresNet protocol. Due to the lack of information, it would either take a very long time to reverse-engineer their protocol, or prove to be impossible at all. Also, reverse-engineering a proprietary protocol and then applying it in a custom solution might prove to be a legally questionable feat.
This more or less forced us to develop our own bus system and accompanying hardware.

We were given two 15” touchscreens we could use for the project. These touchscreens are full industrial computers, and have five serial ports each. The serial ports are RS232 compliant, which means that microcontrollers can be easily communicated with. This made it easier for us, because we could make a serial bus system for controlling the conveniences in the auditoria.

The main goal for the replacement system is modularity. This means that changes made to the auditoria in the future can be integrated into the system with ease.

All existing devices should be controllable by the new system by means of preset buttons on the touchscreen, the manual control interface on the touchscreen, or the key panel at the door, should there be one.

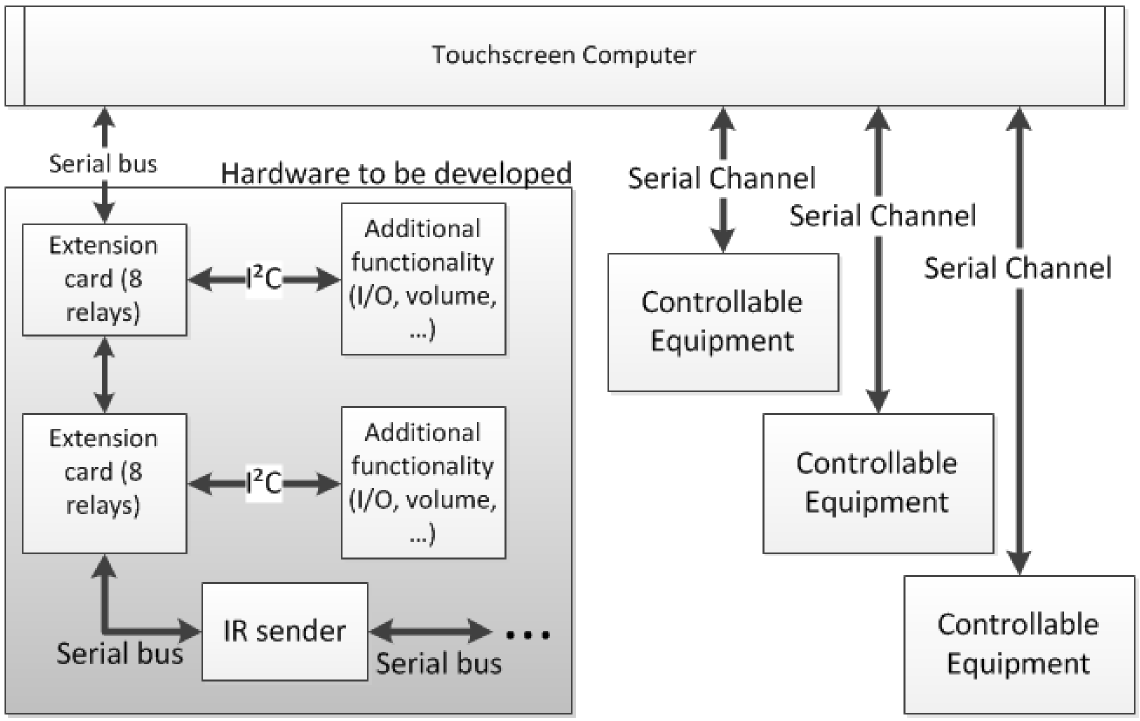


Figure 10 Final topology of the custom solution

4 HARDWARE DEVELOPMENT

To achieve the goals set in the previous chapter, new hardware had to be developed. A relay card to switch various outputs, a keypad to control the lights without the touch screen, and a volume controller to adjust the sound levels of the audio sources will have to be made in order to retain the basic functionality that was available previously. A digital I/O interface and an infrared card to control the video and audio equipment can also be developed, but they are not explicitly necessary.

4.1 Base relay card

The Crestron system uses relay switches to switch lights and power outlets on or off, and to move the curtains up and down. These relays are wired to control other relays which then switch the mains power to the equipment. This means that the control system can work on a safe DC[3] voltage of 24V.

The base relay card we designed should replace the Crestron system and switch the 24VDC voltage to switch the main relays. By use of a microcontroller, these helper relays will be switched. The microcontroller’s outputs are digital outputs, and so have a voltage level of 0 or 5V. The outputs drive LED’s which were added to give visual feedback of each relay’s status. Through the use of an opto-coupler, the logic high or low from the microcontroller is used to control the state of a transistor. If the transistor gets light from the LED, it will go into a conducting state, and the current will flow through the relay coil, activating the relay contact. As such, the opto-coupler provides galvanic separation of the two supply rails.

Every microcontroller has two USART[4] channels, which are used for inter-card communications over the RS232 standard. Because the on-chip USART operates at the core chip voltages, in our case at 5V TTL[5], level translation was needed to convert the TTL levels to RS232 levels. This conversion is provided by a MAX232 transceiver chip, with two integrated transceivers, so that two channels can be provided. The two channels are needed to connect the cards to each other, and thus creating a bus system, where each card determines whether the command is for him, and passes it on to the next card if it is not.

Because of that, every card needs to have an address. This address is user-configurable by means of a series of five switches, representing the binary value of the configured address. Using five bits, there are 36 possible addresses. Multiplied by eight relays per card, this gives us a maximum of 288 switchable relays on the bus. Of course, if another relay card is designed with more relays per card, this number will go up as well.

[3] DC: Direct Current
[4] USART: Universal Serial Asynchronous Receiver and Transmitter
[5] TTL: Transistor-Transistor Logic. Logic zero is 0V, logic high is 5V.

After all the basic goals were implemented, an expansion port was added to the card. This port provides an I^2C interface plus a 5V supply voltage. It will be used later on by the volume control and I/O extension cards, and can also be used for future extensions.

4.1.1 Microcontroller

As mentioned before, the microcontroller is the heart of this relay card. It handles all communications and controls. Due to previous experience and projects at HAMK, a PIC microcontroller was preferred. With the micro-architecture already decided upon, a suitable microcontroller for this project could be chosen. It should have at least the following features: two USART channels for communications, several PWM[6]-modules for the eventual implementation of an IR transmitter, and enough I/O-pins to drive the eight relays.

The USART channel will be used to set up communications between the relay cards on the bus, and the touchscreen computer, which acts as the bus's master. The importance of the second USART channel is to be able to retransmit the received data down the line if the command is targeted at another card.

An infrared card will be designed to control several A/V-peripherals. This card should be able to fit on the same bus structure as the relay cards. The PWM-modules are important for this part of the project. The use of the same microcontroller for both cards makes designing and programming its functions a bit easier, and lowers the cost when buying in bulk.

The third and last major feature the microcontroller should have, is enough I/O-pins. As mentioned before, the microcontroller must be able to drive eight relays at the same time. At the same time, five address bits must be read to allow the card's address to be set by the user.

After deciding on the necessary features, we selected a microcontroller based on its price. The PIC18F24K22 was the cheapest part to meet all demands. It also has two I^2C-busses which could be used for future expansions, such as the volume control and I/O-port expander. The 24 I/O- pins on the microcontroller satisfy the need to control the eight relays, read the address switches and implement the two communications channels.

The PIC runs on a power supply between 2.5 and 5V, and has a maximum output current of 20mA per pin. This maximum should be taken into account when the microcontroller is used to drive external loads, such as the LED's and opto-coupler. To supply the core clock frequency, an external oscillator is used containing a crystal of 16MHz. This frequency is not the the maximum for this microcontroller, but enough to run the program at a satisfactory speed.

[6] PWM: Pulse-Width Modulation

4.1.2 Relay

The relays on the relay card will be used to switch other relays. These other relays are already installed in the wiring cabinet as part of the previous situation, and switch the mains according to their function. These relays' coils operate on a 24V direct current supply. To design our relay card, we chose to use the already available 24V for our relays. Next to the switching voltage for the relay, the durability and price were a decisive factor. The relays should be able to last a long time, because this is a permanent installation, and thus shouldn't require much maintenance.

Figure 11 Panasonic JS1-24-F relay used on the relay card.

The relays we've chosen, with part number JS1-24V-F from Panasonic, are of an industrial type. When switched, the relay needs 15mA of current flowing through its coil to hold the contact, and so consumes 360mW of power. To ensure a long lifespan, the relay should not be switched more than 40 times per minute.

An extra feature of this relay is the presence of normally open as well as normally closed contacts, which is an asset to the modularity of the relay card.

4.1.3 Opto-coupler

The specifications, by which the opto-coupler should be chosen, are a combination of properties of both the PIC and the relay. The function of the opto-coupler is to provide a galvanic separation between the 5V and 24V supply rails on the PCB.

The maximum continuous current allowed on the input of the opto-coupler is 50mA. This is more than double the amount of current an I/O-pin of the PIC can supply. The same maximum amount of current is allowed at the output. This also shouldn't pose a problem, because the relay coil only draws 15mA. The maximum collector-emitter voltage of the output transistor is 35V, which is well over the used 24V of the relay.

We have placed an indication LED between the I/O-pin and opto-coupler input, in order to provide visual feedback when the relay is activated. Together with the internal LED in the opto-coupler, a nominal voltage drop of 3V is present. This leaves us two Volts to limit the current with a resistor. A resistor of 120 Ohm, over which the remaining two Volts are dropped, limits the current to 18mA. This ensures that the operating conditions of the microcontroller's output pin are respected.

4.1.4 Schematic and PCB lay-out

Several targets were set for the design of the PCB[7]. The PCB had to fit on a single layered Eurocard[8]. Because this is a standardized size, there was no need for cutting the PCBs to size, reducing the amount of work needed.

Another important feature during the design of the PCB is the transfer of power. Not only the data should be looped through from one card to another, but there should be connectors loop through the power supply as well. The result is a separate supply chain of 24V and 5V by means of double bipolar screw terminals. On one of the pair, the supply is connected, and the other one can then be used to provide the supply for the next card.

The PCB is divided in a 5V and a 24V section. The 5V section contains the microcontroller, LED's and the MAX232. The 24V section supplies power to the relay switches. On the PCB layout, there is a clear separation of the supply voltages.

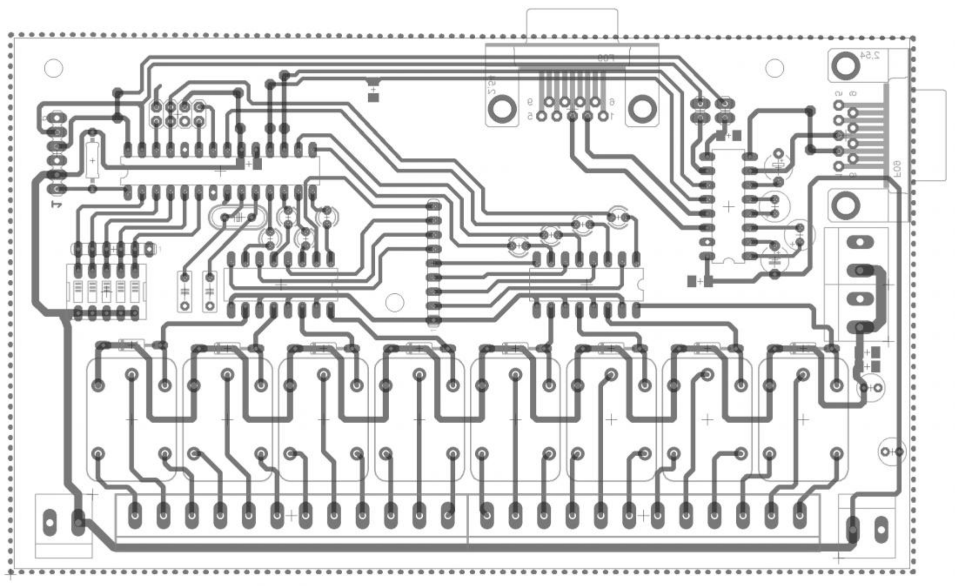


Figure 12 Relay card PCB lay-out

In Figure 12, an image of the final PCB layout is displayed. In this image the ground plane is not shown, in order to display a clear image of the connections. A larger version of this image is available in the appendix, together with the schematic of the card.

[7] PCB: Printed Circuit Board
[8] Eurocard: A standard format for PCB's of 100 by 160 millimetres

4.2 Key-pad controller

In auditorium A, an eight key panel was installed to control the Helvar system. This system was not in use when we arrived due to the malfunctioning of the control module. During the bestowal of our project, our supervisor mentioned that it would be nice if the keypad were to work again.

At first we tried to reinstate the Helvar system. The control module, used to translate the commands from the keypad, malfunctioned. Several LED's on the keypad started blinking without any apparent reason.

The malfunctioning of the Helvar system in auditorium A left us with no dimmable lights, a malfunctioning keypad and a triple cored cable to the technical room. The presence of the cable makes serial communication possible. Our solution was to reuse the switches and housing, but to replace the logic board of the keypad with a microcontroller. By doing so, we would have the keypad interfacing directly with our own serial bus, enabling it to switch relays in order to control the lights.

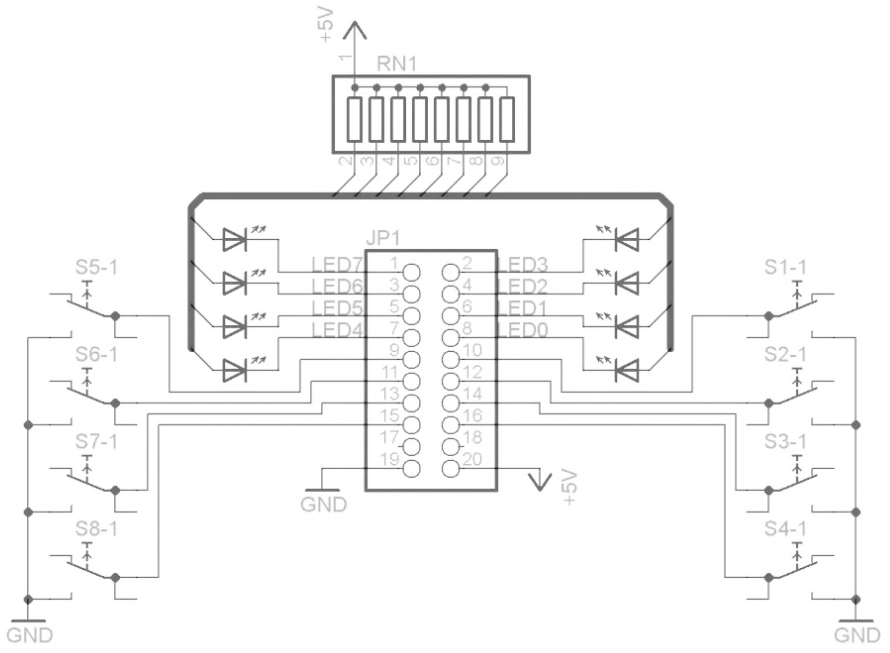


Figure 13 Adjusted keypad schematic

To be able to reuse the existing keypad, several adjustments had to be made. The switches were hardwired as pull-down switches, which means that the signal from the key is tied to ground when the key is pressed. In order to restore the signal to logic high when the switch is released, the internal pull-up resistors of the PIC where enabled. This solution was the easiest and required the lowest amount of extra parts, namely zero.

The LEDs on the key-pad are driven by inverted logic. This means that the I/O-pin has to be forced to ground (0V) level to enable the LED. The firmware to control this keypad will be explained in the software part of this document.

The implementation of following schematic was done on a breadboard PCB. The reason for this was the low complexity of the circuit. Another advantage was the ease of cutting the breadboard PCB to the exact shape needed to fit in the keypad's plastic housing.

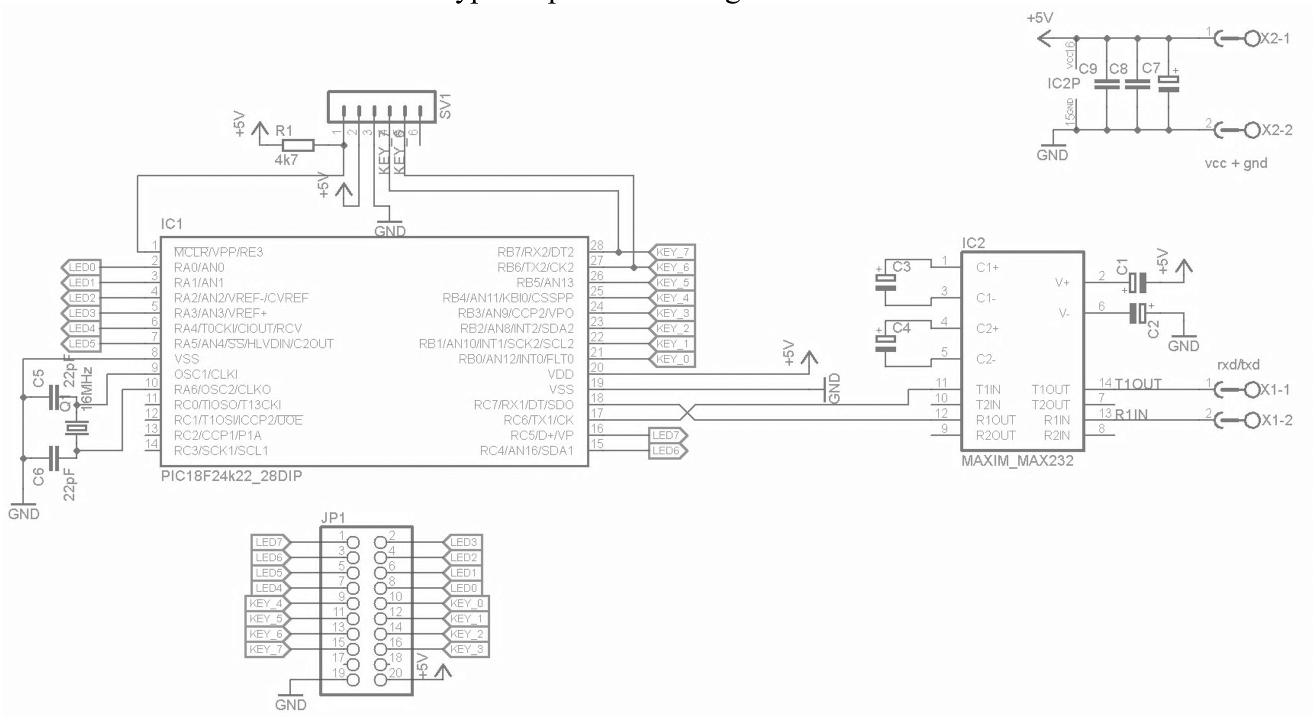


Figure 14 Key-pad controller schematic

The power for this keypad controller is supplied by a 5V adaptor. It was important that when this power supply was plugged in, the relay cards were already operational. This procedure was needed because when plugged in, the first thing the microcontroller did was to ask the relay status, and wait for an answer. If the relay card was not operational, the keypad would be stuck in an endless wait condition.

This problem has however been fixed with a software update, as described in chapter five.

4.3 Volume control extension

In important feature in an auditorium is the possibility to adjust the audio volume. On the PIC of the relay card, an expansion port with I²C is available. An expansion card with I²C digital potentiometers looked like the ideal solution for the volume control.

4.3.1 Digital potentiometer

If the volume of an audio source were to be adjusted by this potentiometer, a logarithmic potentiometer would be needed because the relation between the level of the electrical sound signal and the perceived sound level is approximately logarithmic. We found a digital audio potentiometer on the website of Maxim semiconductor, and sampled it for testing. The potentiometers used in the final implementation were ordered from Mouser.

This specific potentiometer is designed with audio applications in mind. The specific field of use for this potentiometer gives us some extra advantages. For example, it will only change the wiper position when a zero-crossing[9] is detected, which prevents a popping sound during the change. The potentiometer is able to attenuate the signal in steps of 1dB. This means that you can attenuate the signal discretely until -63dB before the mute function becomes active. When the mute function is activated, an attenuation of more than 90dB is attained.

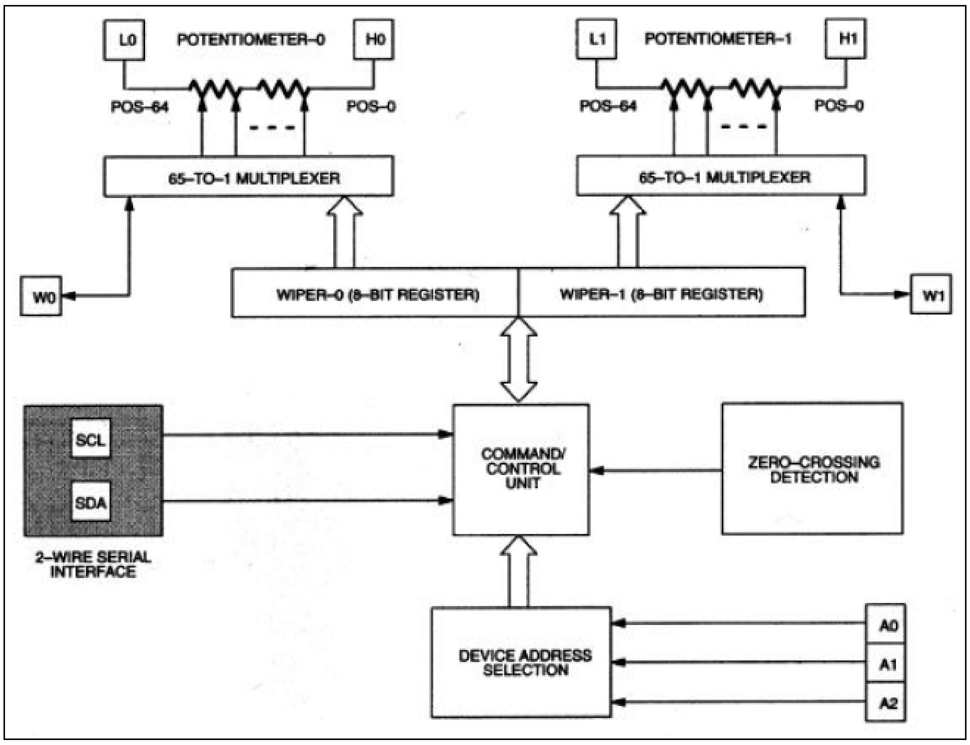


Figure 15 Digital audio potentiometer diagram.

The I²C-bus uses 7 bit addressing. As shown in the diagram, the potentiometer has three configurable bits which can be set by tying them to either ground or supply voltage. These three bits make it possible to address eight different stereo channels on one I²C-bus. Both channels of one potentiometer chip can be adjusted simultaneously for a stereo source, or separately to have mono audio channels.

4.3.2 DC-offset

Unlike analogue potentiometers, digital ones are not unipolar. They have a high side and a low side. In this particular case, the audio signal is fed into high input, and the wiper is the output signal of the attenuator circuit. Due to the digital character of the potentiometer, every signal lower than GND and higher than VCC will be clipped. So, in order to be able to regulate

[9] A zero-crossing is when there is no voltage difference between the H and L input of the potentiometer.

our audio AC signal, we will have to apply a VCC/2 bias to it, and subsequently reference it to VCC/2.

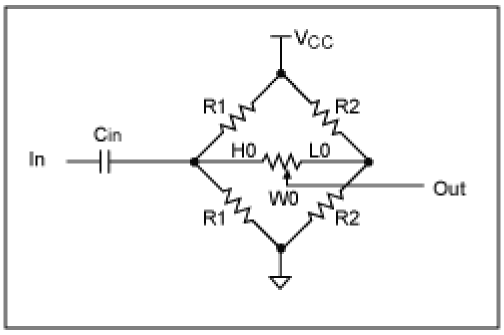


Figure 16 DC-bias adding circuit.

The addition of a VCC/2 DC-bias is done by the following circuit. Two voltage dividers are used. One voltage divider is used to provide a stable VCC/2 reference to the low end of the potentiometer. This voltage divider uses relatively low resistance values to create a stable voltage. However, a trade-off has to be made between reference stability and consumed current.

The other voltage divider uses both of the R1's as resistors. These resistors have a relatively high value, and are used to add the DC-bias to the audio signal. Through the use of this circuit, both ends of the potentiometer are referenced to the same voltage, so that the nominal signal ($U_H - U_L$) is the same as the input signal before the addition of the bias.

The capacitor C_{in} is used to block the external DC bias, should any be present, and keep the VCC/2 DC-bias from leaking into the input. When choosing the value of R1, this capacitor should be taken in account because it creates a high-pass filter in combination with those resistors. The resistor value should be chosen high enough to ensure that the low frequencies are not filtered out.

This potentiometer can't provide a large current on its output. This shortcoming is solved by using an operational amplifier (opamp) as an output stage to buffer the signal. We provide this opamp with an asymmetrical power supply: 5V and GND. This way, the signal is buffered, and the current that flows to the output is provided by the opamp. At the output of the opamp, another capacitor is placed in series to block the DC-bias, and only let the AC audio signal pass through.

4.3.3 Testing

To test the circuit, several sine signals were provided on the input. The input signal is measured on one channel of the oscilloscope. All probes are set to the x10 attenuation setting, and so is the scope. The goal of this test is to see whether the mute function really works.

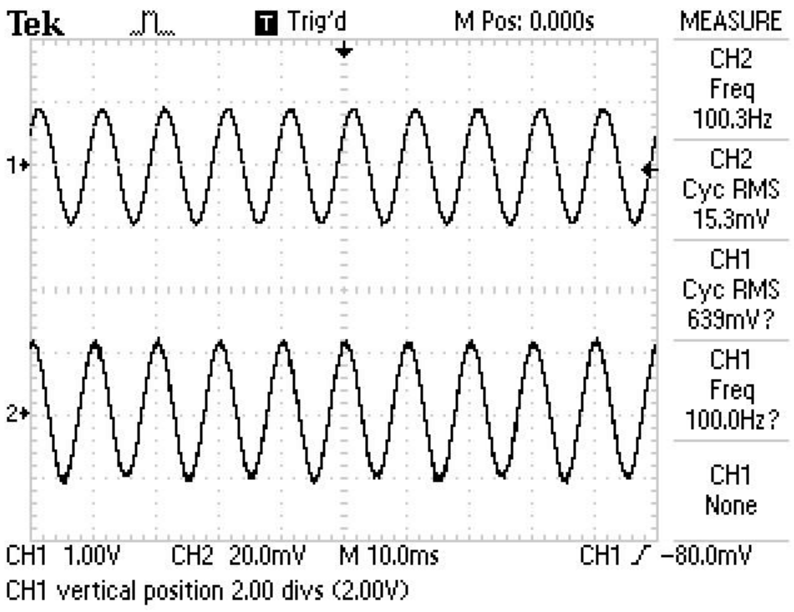


Figure 17 Scope image of 1 potentiometer channel. Ch1: input Ch2: output.

On the scope image, the input voltage is 639mV RMS[10], and the output is 15.3mV rms. This gives an attenuation of $20 * \log\left(\frac{15.3}{639}\right) \cong -30dB$. This measurement clearly points out that something is not correct, because the datasheet of the potentiometer states that an attenuation of more than 90dB should be attained while in mute.

4.3.4 Solution

After analysing the existing schematic, we came to the conclusion that the VCC/2 reference on the low input of the potentiometer fluctuated together with the input. To troubleshoot this problem, the AC model of the circuit was examined. The schematic shows that when an AC signal is added, a little part of the signal will arrive at the low input, which is the point of reference. To stabilize the reference voltage, a capacitor was added between the low input and ground. When added to the AC model, this would result in a short-circuit for AC signals between the low input and ground, eliminating any AC signal leaking into the reference.

.

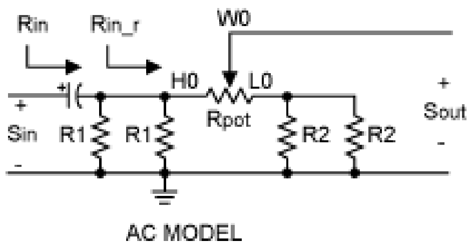


Figure 18 AC model of the bias circuit

[10] RMS: Root Mean Square, the DC voltage that would cause an identical power dissipation in a resistive load.

This solution was tested, and preliminary approval was obtained through real-world testing. The ultimate confirmation was provided by measuring the output signal of the improved circuit with the oscilloscope.

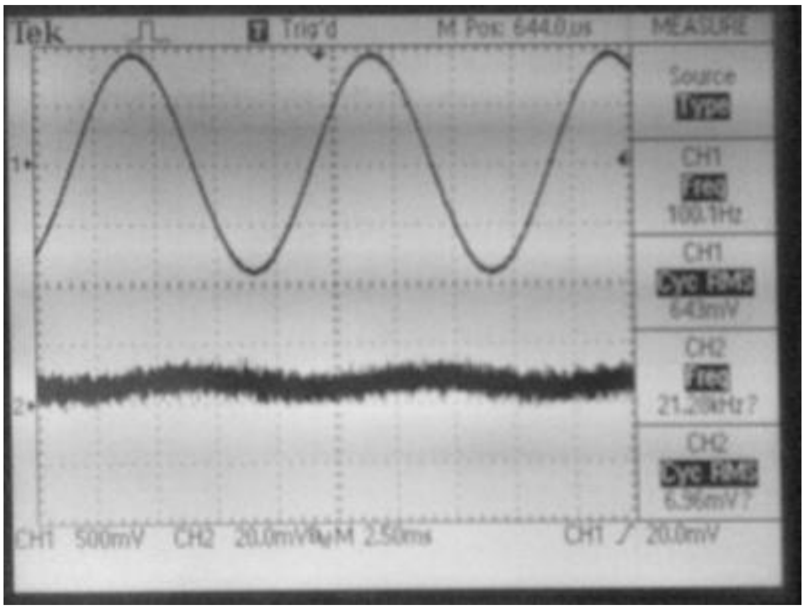


Figure 19 Scope image after adaption. Ch1: input Ch2: output

When the same signal as before is inserted in the potentiometer, practically no signal is visible anymore on the output. The small signal that is still present, is unavoidable due to the internal schematic of the digital potentiometer chip.

All previous tests and measurements were conducted with a laboratory power supply. However, a high frequency tone was present in the audible sound after the implementation in the auditorium.

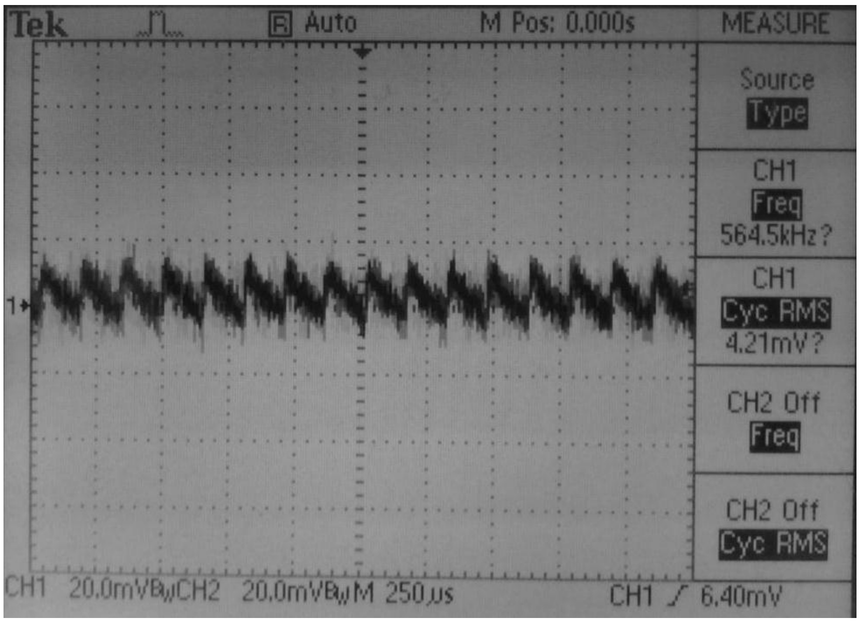


Figure 20 The AC-signal on the 5V power supply

The cause of this problem originates in the power supply. When the power supply output voltage was checked on a scope, a small AC signal was present.

Because the volume of microphone signal is adjusted before the pre-amp, even the little AC signal of the power supply is a big signal in comparison with the output signal of the microphone. This causes the high frequency sound to be heard through the speakers, because it first leaks into the microphone signal, and is then amplified a hundred-fold by the pre-amp.

A solution for this problem seems very easy. Two large capacitors were placed in parallel with this supply, in an attempt to remove the AC ripple signal.

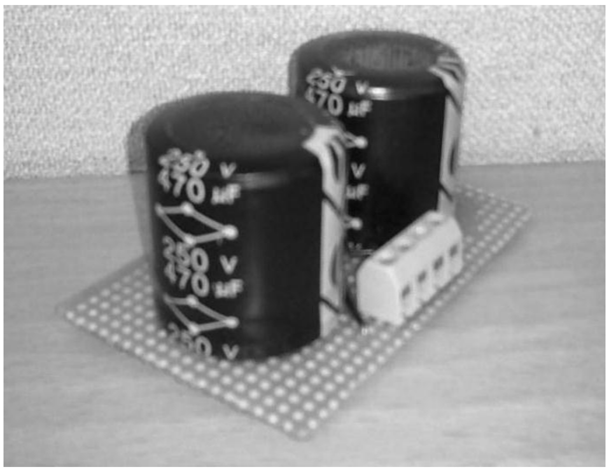


Figure 21 Large smoothing capacitors, 470µF parallel

These capacitors smoothed the signal, as shown in the next scope image. However, when the circuit was tested again under operating conditions, the high frequency tone could still be heard.

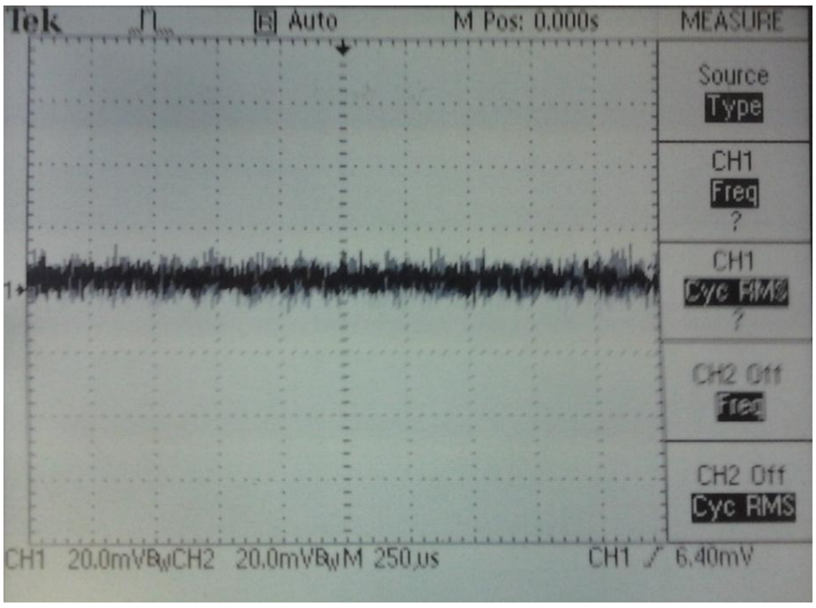


Figure 22 The AC-signal on the 5V supply after adding smoothing capacitors

Even though this signal looks cleaner, there is still a lot of noise on it that was still leaking into the audible signal of the microphones.

Our final solution was to provide the potentiometers with their own 5V power supply. This 5V is provided by an LM7805 linear voltage regulator which should have a noise suppression factor in the audible range of 70dB or more. The supply for this voltage regulator is provided by a universal adaptor which set to provide 6V. A practical test proved that the high-pitched tone had disappeared from the amplified sound signal, or was at least not audible anymore.

4.4 Digital I/O extension

When the relay cards were designed, almost all the I/O-pins were in use. The remaining I/O-pins were used as an extension port. This port provides an I²C bus and a 5V supply, and is already used by the volume control circuit. This means that when making a digital I/O extension, a chip with an I²C interface will need to be selected.

To provide extra digital I/O-pins, a remote eight-bit I²C I/O expander was chosen. We chose the PCA9554, made by Texas Instruments. There are two versions of this chip: the normal one, and the A-version. They are essentially the same device, except that the non-configurable part of the bus address is different, so that up to 16 expanders can be addressed on the same bus.

Next to the standard SDA and SCL signal used by I²C, this port expander has an interrupt line available. This interrupt line becomes active when an input changes in state. The interrupt function is only active on each pin that is configured as an input. It could be used to automatically respond when a key is pressed, eliminating the need for polling.

The original idea for use of this port expander, was the simulation of the previously mentioned keypad. The originally installed system containeda device which acted as a keypad towards the Helvar controller, but was triggered by digital inputs. This way, the scenes could be triggered by switching the outputs of the port expander.

However, due to the malfunctioning of the Helvar system in auditorium A, this port expander is currently not in use. It is installed though, and can be implemented in the future, when new equipment is installed which needs extra I/O-pins. The I/O-expander could also be used in auditorium B, where the Helvar system is still in use.

4.5 Infrared card

The infrared cards interface with the bus the same way the relay cards do. They would be attached to the same bus as the relay cards, and data not intended for them would pass through to the next card.

Just like the relay card, the microcontroller is the heart of the infrared card. The specific microcontroller used for the infrared cards is the same as used for the relay cards. It has the most important feature needed to transmit IR-commands available: PWM-modules.

4.5.1 Infrared protocol

The infrared protocols used in auditorium B are based on the NEC format. What is important in this format, is that the bytes are transmitted in the right sequence, and that every byte is transmitted LSB[11] first.
When a logic high is transmitted, the actual pulse is a carrier frequency of38kHz with a duty cycle of 33%. The logic one and logic zero are created by PDM[12]. Both logic values start with a pulse of the same duration. The value of the bit is determined by the duration of the low-state. Because the active pulse remains the same duration, it is the duration, or the distance when the signal is plotted using a time reference, of the low state that changes. A short low time represents a logic zero and a long low time a logic one. These times can differ depending on the brand of equipment.

The duty cycle of the carrier signal is chosen for a special reason. Because of this duty cycle, the instantaneous current flowing through the LED can be three times as high as the normal continuous current. The PDM modulation adds to this advantage. At a logic zero, it is has a duty cycle of 50% and 25% when a logic one is transmitted. Consequently, the current trough the LED can be six to twelve times higher than the nominal current. As a result of this higher current, the LED will flash very brightly, and will be able to control a device at a reasonable distance.

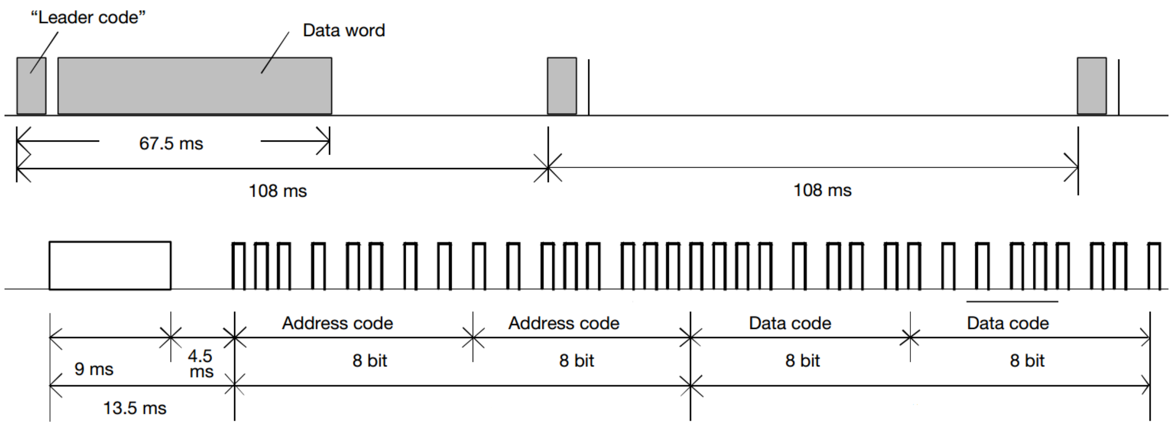


Figure 23 NEC transmission code

To begin the command, a synchronisation pulse is transmitted. In the figure above this is called the 'leader code'. This leader code is used to give the receiving device some time to synchronise and be ready to receive the

[11] Least significant bit
[12] Pulse Distance Modulation

commands. The leader code has a specific timing: there is a 9ms pulse and a 4,5ms low time.

After the leader code, the two data bytes are transmitted. As mentioned before, the LSB of every byte is sent first. These two address bytes are the addresses specific for a device, device series and manufacturer. When the receiving device doesn’t receive the correct address, it ignores the rest of the command, resulting in the signal only controlling the intended device.

The third byte which is transmitted, is the actual command given to the device. The fourth byte in is the inverse of the command byte (i.e. 0x00 becomes 0xFF).

To conclude the transmission, a final stop bit is transmitted at the end. The stop bit consists of a logic one or zero. The high-time of the pulse is the same in both logic values, and so is the important part of the stop bit.

The stop bit is used to determine the value of the last bit of the inverted command. By sending the command and the inverted command, a form of fault detection is present. If the values don’t match up, the device won’t react to the command.

Finally, if a key is held down on the remote-control, it does not resend the complete code. Instead, a pause of 40ms is inserted. After this pause the leader code is resent: 9ms high and 4,5ms low, followed by a logic one or zero. Again, the value is not important because the length of the active pulse is the same. After this a low waiting time is inserted until the complete duration of the repeat command is 108ms. If the button is still held down, this repeat command is repeated.

4.5.2 Commands

To obtain the addresses and commands for the infrared instructions, an infrared sensor was used. This sensor was hooked up to a memory scope which was used to reverse engineer the data sent by the remote.

After analysing several commands per remote, the right remote library can be chosen from the site of winLirc[13].

These commands will be transmitted together with the address and timing from the touchscreen to the microcontroller, using the serial protocol. The firmware of the microcontroller will analyse all timings it receives, and transmit the correct command to the device.

[13] An open-source program to control infrared controllable equipment via the computer with an adapter. Its website contains many libraries for specific devices, detailing their infrared commands.

5 SOFTWARE DEVELOPMENT

Of course, when a programmable microcontroller is used to create the desired functionality, a program has to be written for it in order to implement that functionality. That type of program is called firmware, and the translation of human-readable to machine-readable code requires a special compiler, depending on the manufacturer and type of microcontroller. Since we have used a PIC 8-bit series microcontroller from manufacturer Microchip, the firmware has to be written in, and compiled by, a compatible IDE[14].

Together with the firmware for the microcontroller, there is also another piece of software which has to be developed: the user interface. Seeing that the touchscreen computers available for our project run Windows XP, the GUI[15] needs to be developed for that platform. This requires again another compiler, and a completely different set of software tools.

Furthermore, a protocol definition has to be made up to specify the communications between the touchscreen computer, aka the master, and the relay modules, aka the slaves, on the bus. Since the protocol defines how the software is made up, we started by creating the protocol definition.

5.1 Custom communications protocol

To start drawing up our protocol, we have taken some time to analyse different possibilities, to gather information about the components that we have chosen, and some special characteristics of the used communications bus.

Firstly, we had already decided on the use of an RS232-compatible serial bus for the electrical signalling part of the specification. RS232 is an industry standard, bidirectional, asynchronous bus. This means that data can be sent in both directions at the same time, and that the devices communicating with each other do not need a third signal as a clock reference.

Another advantage of RS232 is that it has proven to be a reliable standard in industry environments, up to cable lengths of several hundreds of meters, or, more specifically, a cable impedance of 2500 pF total. The cable used in the auditoria is stranded, twisted and shielded, so with regards to the length and capacitance, it should pose absolutely no problem to use the RS232 standard.

Concerning the other specified parameters of RS232, we shouldn't think too much about them. On the computer side, everything is already implemented up to standard. On the relay card side, we use a MAX232 transceiver chip, which takes care of the voltage translation and the generation of the required voltages for us. So the specified -5..-15 and +5..+15 levels

[14] IDE: Integrated Development Environment, the whole of software programs to develop a piece of software
[15] GUI: Graphical User Interface

are internally generated by that chip through the use of external voltage doubler capacitors, and on the other side we receive and transmit a TTL-level (0V for a logical zero, and +5V for a logical one) serial signal.

A standard speed to communicate with such a serial interface is 19200 baud with eight databits per frame, which is what we have implemented in our protocol. Because we planned on adding error correction, we did not need the parity check supplied by the specification, and so did not use it.

Finally, with the practicalities out of the way, we could start designing the protocol definition.

5.1.1 First version

The final protocol was designed after the implementation of a first version, which was later deemed not flexible enough. However, for completeness, we have decided to also include the first version of our protocol in this document.

The first version was targeted specifically at the relay cards, and was as such only suited for that particular purpose.

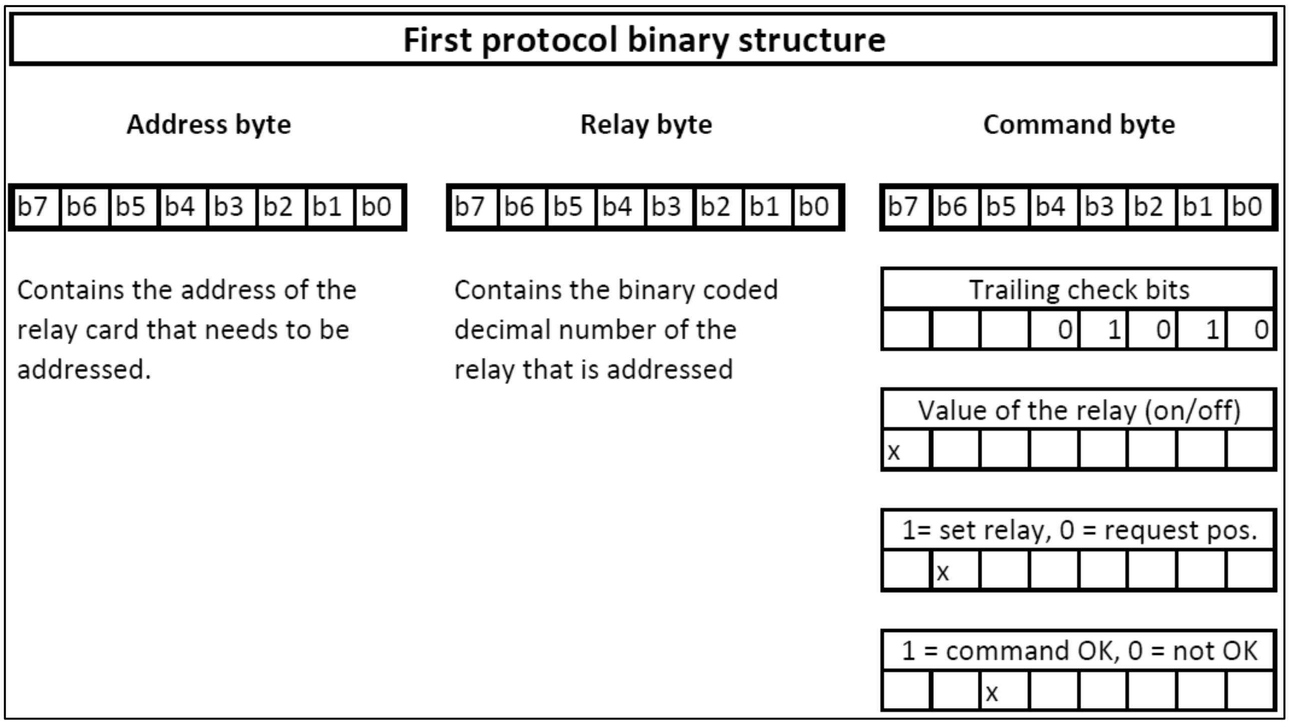

First protocol binary structure
Address byte
Relay byte
Command byte
b7 b6 b5 b4 b3 b2 b1 b0
Contains the address of the relay card that needs to be addressed.
Contains the binary coded decimal number of the relay that is addressed
Trailing check bits
0 1 0 1 0
Value of the relay (on/off)
x
1= set relay, 0 = request pos.
x
1 = command OK, 0 = not OK
x

Figure 24 Protocol definition of the first version

The address byte is the first byte to be sent, so that the relay cards can pass through the message immediately without having to wait for a completed message to begin retransmission. The address itself is zero-based, and is matched by the relay card to its switches' settings. So if the address switches are set to binary 22, you should address card 22. Seeing that the cards only have five switches, a maximum of 64 cards can be addressed on the same bus.

The second byte is the relay byte, with which the relay that needs to be addressed is specified. The relay addresses are also zero-based, so the first relay is addressed as relay zero, and the last one as relay seven.

The command byte consists of some check bits, and three command bits. The check bits are always ‘01010’, and can be used to detect timing inconsistencies or continuous faults. This is a very basic error checking mechanism, and is, in hindsight, not very reliable.

The MSB[16] is used to indicate either the state of the relay that that should be applied, or the actual state of the relay, depending on the direction of communication. If the PC sends a command with bit 6 set, the MSB contains the value that needs to be set. If bit 6 is off, the PC is requesting the relay’s status, and then the MSB is a don’t care-bit in the direction of PC to relay card.

In the other direction, if the PC has set a status, the relay card returns exactly the same 3 bytes to the PC, but uses bit 5 of the last byte to indicate whether it has successfully processed the command (bit 5 is on), or that there is something wrong with it so that it can’t be executed (bit 5 is off). In the event that bit 6 was off, and thus a status has been requested, the relay card puts the actual value of the requested relay in bit 7 of the last byte.

5.1.2 Final version

When it became clear that the relay cards would also be capable of driving extension cards containing, in our implementation but not limited to, volume controllers and digital I/O ports, a new protocol had to be defined. As long as we’d stick with driving relays, the old protocol would do just fine, but for the extensions we certainly needed some more command possibilities.

The first step in accomplishing this was to split up the command and checksum. With a full byte dedicated to the command, we can now define up to 256 possible commands, which makes for great versatility, and easy expandability when other extension cards are to be designed.

The separated checksum does also come in handy, because we can now implement a full content-based checksum instead of relying on a predefined bit pattern. One error checking mechanism that was thought of was CRC-32, but as it turned out, it was too elaborate to check the checksum in the firmware without losing performance, or mangling subsequent bytes sent through the serial pipeline. In the end, a bitwise exclusive OR (XOR) function was chosen as a rudimentary but effective error checker.

The major advantage of a XOR function is that it takes just one clock cycle in the microcontroller to execute, and as such it takes virtually no time to check the integrity of the data sent through the pipeline.

[16] MSB: Most Significant Bit

Seeing that we now have the possibility to command external features on the relay card, and not only relays, the relay byte from the previous protocol was repurposed as value byte. Each command can have a value attached to it, which can be the number of the relay that needs to be set, or the volume level that is requested, or something else entirely. So, this way, we stop wasting half a byte of zeroes, because in the previous specification the relay number was always between zero and seven, which meant that the five most significant bits of that byte would always have been zero.

Lastly, we need a specifier to indicate whether the message is travelling from PC to relay card, or from relay card to PC. This is important in the event that something else than relay cards would be attached on the same bus, for example IR transmitters. For now, however, this specifier is a predefined ASCII character of '&' for a command to a relay card, and '?' for a response from the relay card.

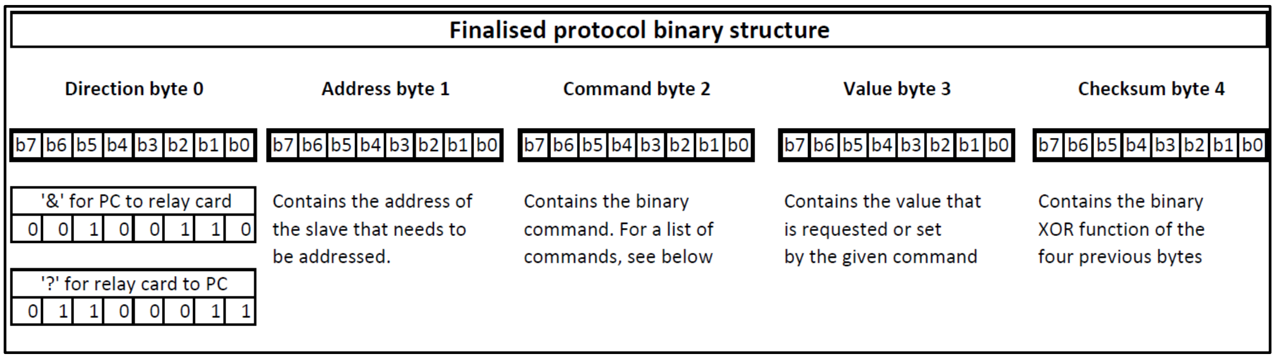
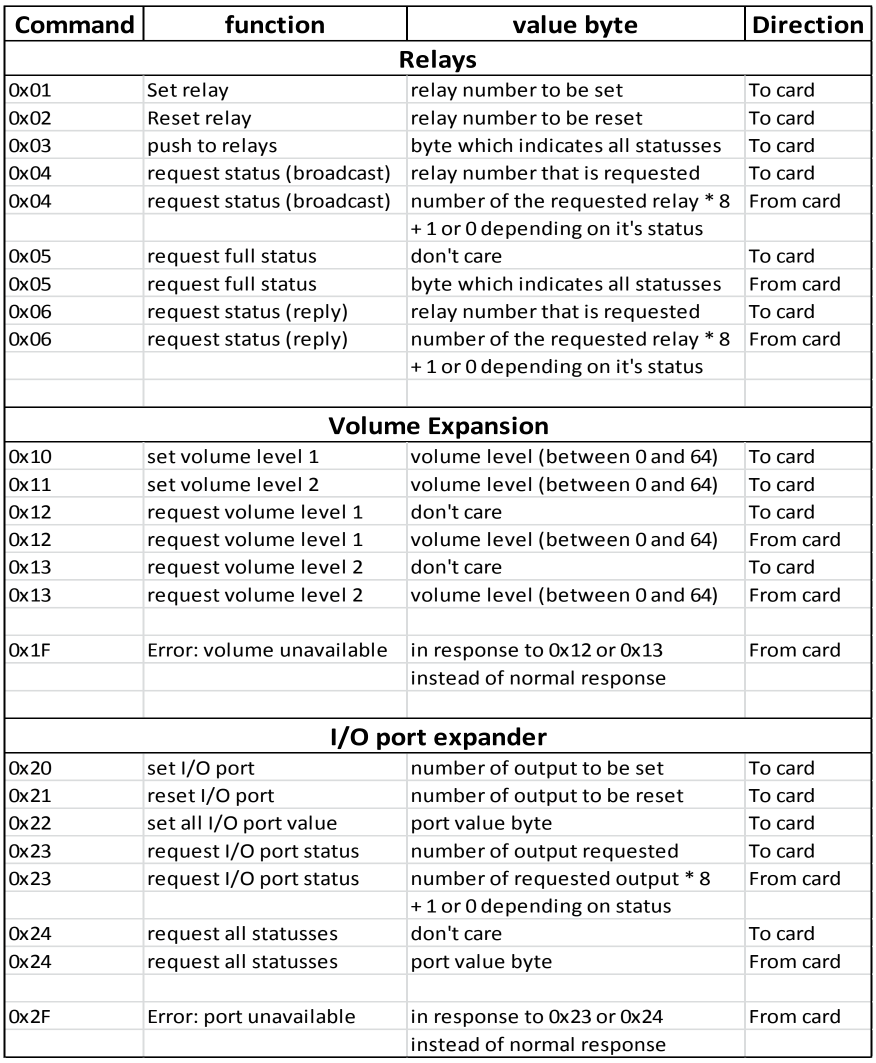

Finalised protocol binary structure
Direction byte 0
Address byte 1
Command byte 2
Value byte 3
Checksum byte 4
b7 b6 b5 b4 b3 b2 b1 b0
'&' for PC to relay card
0 0 1 0 0 1 1 0
'?' for relay card to PC
0 1 1 0 0 0 1 1
Contains the address of the slave that needs to be addressed.
Contains the binary command. For a list of commands, see below
Contains the value that is requested or set by the given command
Contains the binary XOR function of the four previous bytes

Figure 25 Protocol definition of the final version

A visualisation of the finalised protocol can be seen in the figure above, which might clarify things a bit further. A list of commands that are defined up until now has also been included hereunder. As you can see, there is still a lot of room left for extra expansion commands in the range of 0x30 to 0xFD. 0xFF is reserved.



Command	function	value byte	Direction
Relays			
0x01	Set relay	relay number to be set	To card
0x02	Reset relay	relay number to be reset	To card
0x03	push to relays	byte which indicates all statusses	To card
0x04	request status (broadcast)	relay number that is requested	To card
0x04	request status (broadcast)	number of the requested relay * 8	From card
		+ 1 or 0 depending on it's status	
0x05	request full status	don't care	To card
0x05	request full status	byte which indicates all statusses	From card
0x06	request status (reply)	relay number that is requested	To card
0x06	request status (reply)	number of the requested relay * 8	From card
		+ 1 or 0 depending on it's status	
Volume Expansion			
0x10	set volume level 1	volume level (between 0 and 64)	To card
0x11	set volume level 2	volume level (between 0 and 64)	To card
0x12	request volume level 1	don't care	To card
0x12	request volume level 1	volume level (between 0 and 64)	From card
0x13	request volume level 2	don't care	To card
0x13	request volume level 2	volume level (between 0 and 64)	From card
0x1F	Error: volume unavailable	in response to 0x12 or 0x13	From card
		instead of normal response	
I/O port expander			
0x20	set I/O port	number of output to be set	To card
0x21	reset I/O port	number of output to be reset	To card
0x22	set all I/O port value	port value byte	To card
0x23	request I/O port status	number of output requested	To card
0x23	request I/O port status	number of requested output * 8	From card
		+ 1 or 0 depending on status	
0x24	request all statusses	don't care	To card
0x24	request all statusses	port value byte	From card
0x2F	Error: port unavailable	in response to 0x23 or 0x24	From card
		instead of normal response	

Figure 26 List of possible commands

5.2 Designing the relay card firmware

The first piece of software that has been developed is the firmware for the relay card's microprocessor. This has been very much a work in progress, since it had to be changed with every change in the protocol, and with every extra extension card that was designed.

5.2.1 Choosing a programming language

In the world of programmable electronics, there are literally dozens of capable programming languages available. Which one to choose greatly depends on the manufacturer and microarchitecture of the used processor, the cost to efficiency ratio of the compiler and accompanying IDE, and ultimately on the experience of the programmer.

For the microcontroller used in our project, from the PIC 8-bit family, there exist a couple of options. Firstly, the native language of the processor: assembly. When you write your program in assembly, it means that you write direct instructions for the processor. This has several advantages:
– The programmer controls exactly what the processor will do
– The written code is virtually free of the usual overhead of commercial compiler suites
– There is no need to spend money on a commercial compiler

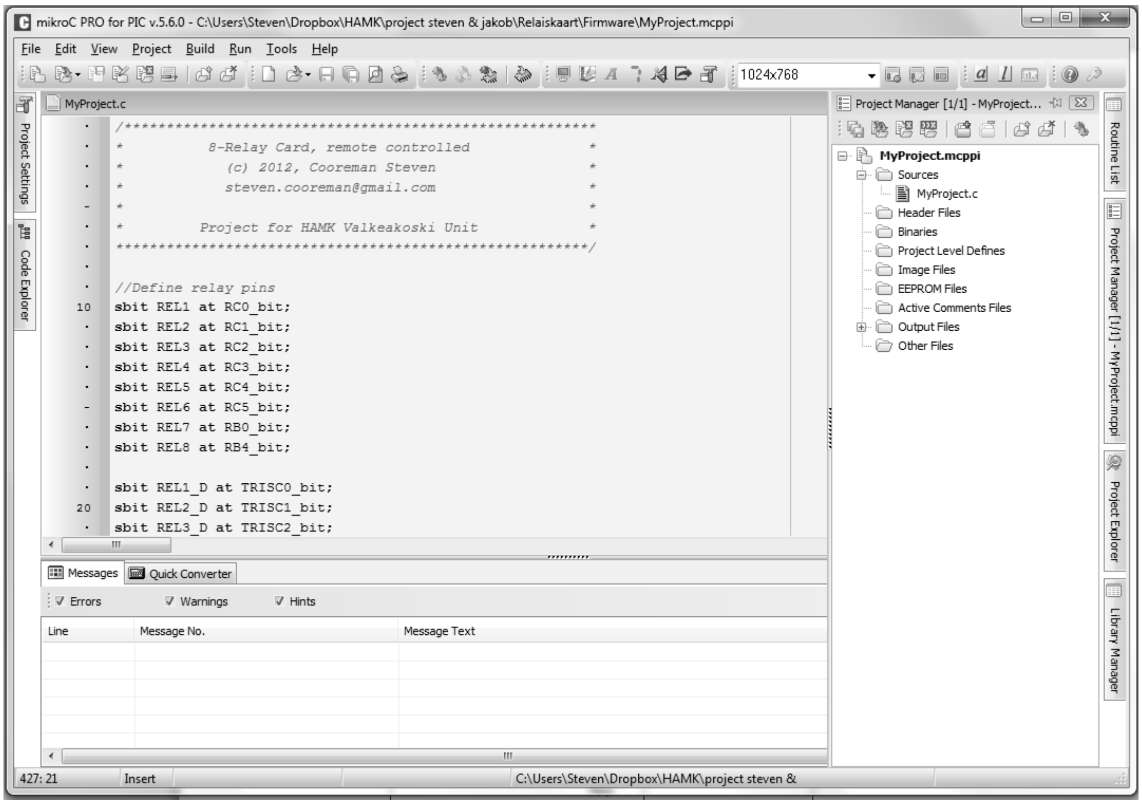
Of course, there are also disadvantages, with the major one being that writing assembly is incredibly tedious and time-consuming, and that you can quickly forget about the bigger picture and start focussing on small details. And then there is also instruction set familiarity: it takes a great deal of time for a programmer to learn a new instruction set up to a level where assembly programming becomes comfortable. Obviously, the less instructions, the easier to learn, but also the more code that has to be written due to the absence of combined instructions. Programming in assembly is also referenced to as hardware-level programming.

One level up the programming abstraction ladder comprises the so-called intermediate programming languages, such as the one we will ultimately end up using. The advantage here is that the programmer does no longer need an in-depth knowledge of the microarchitecture and the instruction set. Instead, the code is written in methods and statements, which are then programmatically compiled into assembly, and ultimately into machine code.

The probably best known programming language in this category is C. While there is a great deal of variants of this language, it has been standardized by the American National Standards Institute (ANSI), with its last major revision completed in 2000, and a new one currently in the works. Since both authors are relatively versed in C, we decided on using this language.

With the specific language to use decided upon, there still remains the question of which compiler to use. Specific compilers for the Microchip PIC architecture include Microchip's own HI-Tech C compiler with a trial version lacking code optimization, BoostC, MikroC, and so many more. The advantage of using Microchip's compiler is that you can work directly in their own IDE called MPLAB, so you can use all the new parts directly as they are released.

However, since the programmer of the firmware had not yet used MPLAB, and was familiar with the IDE of MikroC with which he had worked before (MikroC also exists for the Atmel AVR-series of microcontrollers), this particular compiler/IDE combination was chosen.



```
/*********************************************************
*          8-Relay Card, remote controlled               *
*             (c) 2012, Cooreman Steven                  *
*            steven.cooreman@gmail.com                   *
*                                                        *
*         Project for HAMK Valkeakoski Unit              *
*********************************************************/


//Define relay pins
sbit REL1 at RC0_bit;
sbit REL2 at RC1_bit;
sbit REL3 at RC2_bit;
sbit REL4 at RC3_bit;
sbit REL5 at RC4_bit;
sbit REL6 at RC5_bit;
sbit REL7 at RB0_bit;
sbit REL8 at RB4_bit;

sbit REL1_D at TRISC0_bit;
sbit REL2_D at TRISC1_bit;
sbit REL3_D at TRISC2_bit;
```

Figure 27 Sample view of the MikroC IDE

Although the MikroC compiler has a few quirks of its own, it still generates relatively compact and efficient code. For example, in standard C programming language, you have to 'include' the libraries you wish to use in your program. In MikroC, you have to tick the boxes of the libraries you want to compile in the IDE. It works, but makes for less portable source code if you would want to compile it for another architecture.

5.2.2 Implementation of the relay card firmware

With the compiler and development environment now decided upon, we could start the development process of the firmware. For obvious reasons, we will not explain the code line for line in this document, but rather give a general overview of how the program is built up. For a more in-depth view on the firmware's source code, we refer to the accompanying CD, which includes all source files. The full code is documented in-line.

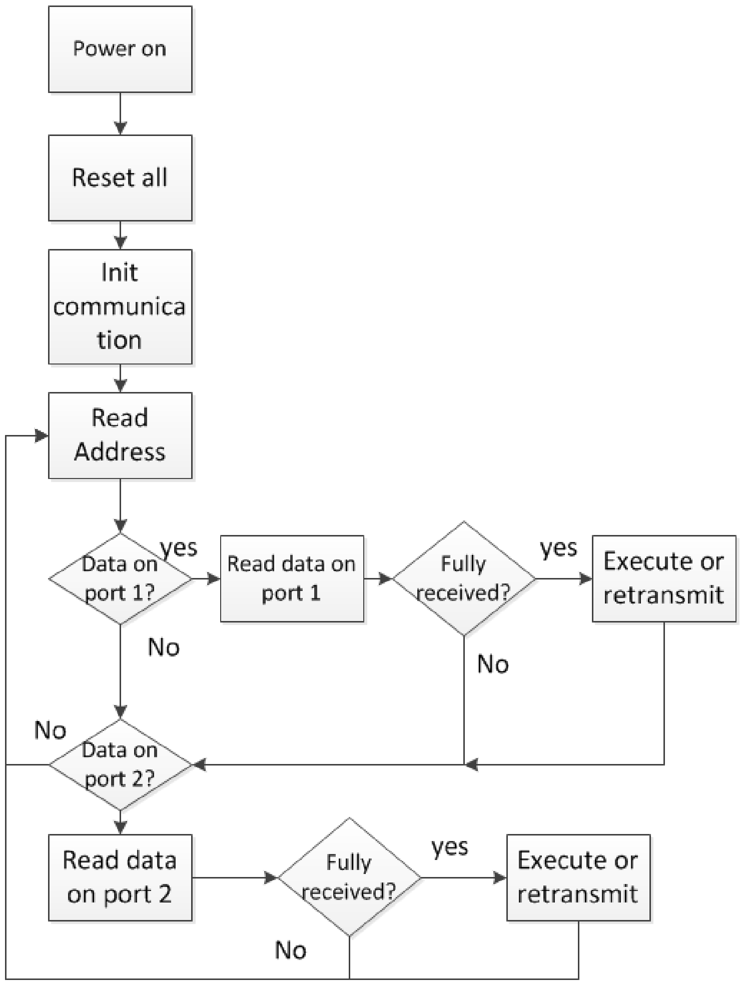


Figure 28 Flowchart of the basic relay card operation

The above flowchart should be fairly self-explanatory. On power-up, the card resets all relays to their 'off' position. It then starts its communication ports, being the two serial ports and the internal I^2C bus for communication with the possible extension cards.

On initialisation of the I²C bus, the card also polls all known addresses for a response, in order to detect whether any peripherals are attached. This only happens at the power-up stage, so hot-swapping of extension cards is not possible. Every time you want to change extension cards, you need to power cycle the circuit.

Once everything is initialized, the program enters an infinite loop, in which it checks the switches to determine its address, and it reads the incoming data on both serial ports should any data be available. If the received character corresponds with a direction indicator, it starts filling its reception buffer for that port, until the necessary amount of bytes (five

bytes for this protocol) is received. Once that is done, it checks the checksum byte, and if the checksum matches, the command is interpreted.
During the interpretation, the program checks whether the command is for a relay card (direction byte 'command to relay card') and whether the address of the command matches the position of the address switches. If any of those conditions are not met, it retransmits the full command on the other serial port, in order to get the message through to the other slaves. If both conditions match, the command and value bytes are interpreted, and the appropriate action is taken. If any response is necessary, it will also be sent straightaway, and normal program flow is interrupted until the response has been sent.

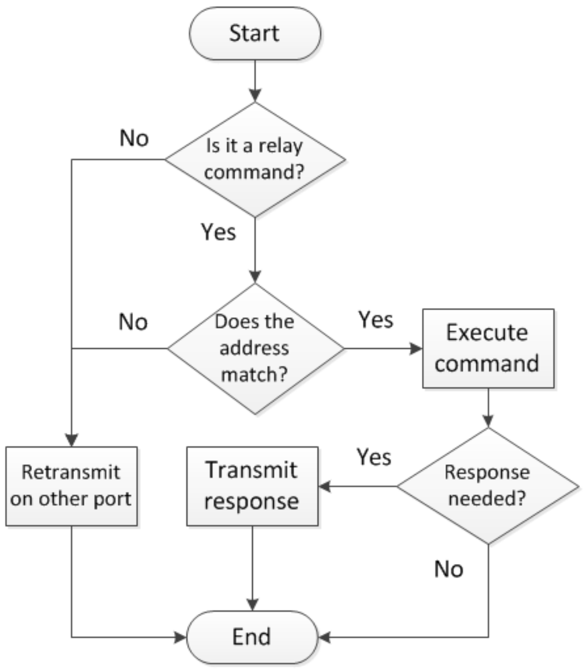


Figure 29 Flowchart of command interpretation

5.3 Designing the keypad controller firmware

When we started thinking about the light button-panel problem in auditorium A, it became quite clear that it needed a rather special solution. After all, the lights needed to be able to be controlled by both that keypad and the touchscreen computer.

After a while of investigating, we found the source of the problem with the keypad: the dedicated light controller had broken down beyond repair. Since that specific model had been taken out of production a very long time ago, there was really only one solution: replace the logic within the keypad with an interface of our own.

The keypad, however, is normally connected to the Helvar system on a two-wire bus which supplies both power and bidirectional data. In order to integrate the keypad in our system, we needed to hook it up into our serial bus, which uses three wires to communicate. More specifically, two wires for bidirectional data, and a common voltage reference. So, how would we make that work?

The solution turned out to be surprisingly simple: the cable that had been used by the lighting system before was actually a three wire mains cable. That meant that we had enough conductors for communication. The only thing left was to supply power (+5V DC) locally using a small adapter.

After the practicalities, the firmware had to be designed. Since the serial protocol was already defined, it was only a question of reading the button inputs, and displaying the current status of the relays on the LED's. The LED's also proved to be an effective way to have a local cache of the light relays' status, so that it doesn't have to be stored somewhere else in the program.

The basic program flow is as follows: if the keypad is powered on, the microcontroller initialises all its inputs and outputs, and initialises its serial communications port. It then begins asking for the light statuses, one at a time. If a response is received, the status of the next relay is requested by the keypad, which then again waits for a response before sending the next request. After all the requests have been answered, the buttons are enabled, and the user can begin using them as light switches.

This program flow implementation has one major drawback: if one of the eight address/relay combinations is unreachable during start-up, the program flow will be interrupted, blocking the key presses from registering and sending out their respective commands. If this happens, and all the positions are actually available, a power cycle of the keypad should suffice. If one of the addresses is unavailable, then there is no easy solution apart from making it available on the bus.

The solution to this would be to not require an update of all corresponding lights on start-up, but then the status displayed on the LED's would be incorrect on power-up, until each button is pressed at least once.

The address and relay combinations that correspond with a specific button are stored in program memory, as a two-dimensional array. This means that the keypad has to be reprogrammed manually every time a light moves from one relay contact to another. Luckily, this shouldn't have to happen for a long time, and if it were to happen, the source code is supplied.

There was another problem with the addition of the keypad at one end of the serial bus, with the computer at the other end. If one of those switches a light, how will the other one notice? Requesting all the lights' statuses, one at a time, every few seconds would constitute a tremendous amount of overhead on our bus, slowing down other bus operations.

For this reason, command 0x04 was introduced. This command is just like a regular 0x06 'get status' command, but instead of returning its response on the serial port that it received the request on, it actually returns a response on both ports. That way, the message will reach both the keypad and the computer, if they are powered on. Therefore, the command has been nicknamed 'broadcast status'.

This command requires some special handling though, as it can be received unsolicited, which means that even if the master has not requested a status, it can still receive a response. Luckily for us, this special handling in the keypad's firmware is limited to always listening on the serial port. The program is very simple, so it should have ample processing time left to be able to do that.

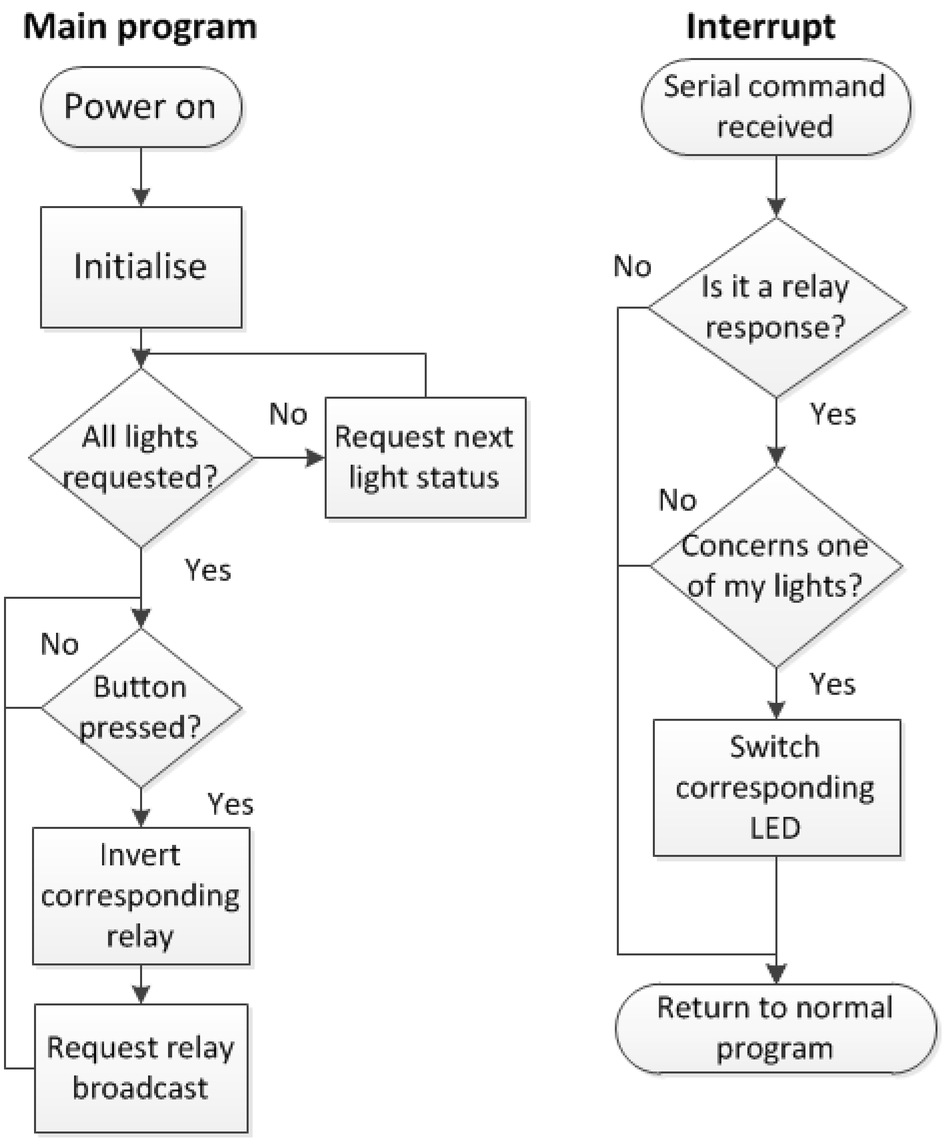


Figure 30 Flowchart of the keypad operation

5.4 Development of the PC software

The firmware for the microcontrollers, of course, is only part of the big solution. Since the whole project is meant to be as user-friendly as possible, a touchscreen computer running Windows XP was made available as a central command unit.

Seeing that the computer is running Windows, the user interaction software would have to be programmed in a language that can be compiled into an attractive-looking, intuitively working program that can be controlled with just a touch. The graphical design of the solution plays a big part in that, but if the used language doesn't support the use of a custom graphical user interface, then the project is doomed from the start.

However, the GUI is the culmination of the project. Before starting work on the finishing touches and small cosmetic errors, the core logic of the program should be working correctly and up to specification. This would prove to take up the majority of the time spent on the project.

5.4.1 Choosing a programming language and IDE

As with the firmware, a suitable programming language and IDE will also have to be chosen. This choice is particularly difficult, because of the many possible software platforms available for Windows.

For starters, there are the normal Win32 API's [17] with which your program can directly access Windows' resources. While this is a way to create very speedy code, it is not unlike coding in assembly for a microcontroller: it takes a lot of time to learn, a lot of time to get the code just right, and a huge amount of code because the programmer has to take care of literally everything. Plus, if there's some overlooked little bug in the code, it can very quickly lead to a BSOD[18] crash. Those bugs can be very hard to track down, and might not even appear until sometime after the program has been put into service.

On a higher level of abstraction, there are all the languages that need a runtime of some sorts. A runtime is a native program (in the case of Windows: written for Win32) that runs your higher level code of that language, and translates it into native code in real time, while the program is running. The advantage is that, if the runtime is available for multiple platforms, your code is compatible with all of those. For example, if a runtime of a specific language exists for Windows, Mac OS and Linux, your program can run on any of those without modification or platform-specific code. Well, in some cases you might want to introduce some platform dependency, but that goes beyond the scope of this project.

[17] Application Programming Interface
[18] BSOD: Blue Screen Of Death, a colloquial name signifying a critical system failure with an immediate restart, due to its appearance on Windows as a blue screen with white letters, containing information about the crash.

A language of this kind, which is commonly used in the academic world, is Java. Java, originally created at Sun Microsystems, is an object-oriented language with a big amount of runtimes available. For example: Windows, Mac OS, all flavours of Linux, Solaris, etc. all have a Java interpreter available. Even Android, the mobile operating system from Google, runs a Java-like interpreter under the hood.

While Java might sound like the ultimate solution, there are a few disadvantages. Firstly, because the code must be able to be interpreted on all those platforms, it doesn't run too well on any of them. There still is an associated speed penalty when programming in Java.

Secondly, and more important, the graphical features of Java are not that extensive without an add-on library, which then of course brings along another speed penalty. Other than that, those extra libraries are often not so easy to use and mainly geared towards the academic audience.

Another JIT[19] interpreter, and fairly new, is Qt[20]. This is a framework with good graphical features, created as an open-source alternative to the other major players. Given that it was created by Nokia, its main use lies of course within the smartphone industry, but it has nevertheless also found its way to the desktop, with runtimes being available for the three major desktop platforms. It became famous for being the go-to language for Nokia's mobile OS's, first Symbian, then Meego/Maemo.

However, since it is a relatively new language, there is not that much information readily available, and it can be difficult to find an answer should you run into any kind of problem while programming in Qt. It is also not commonly taught in a classroom environment, so it would be difficult to find a student to follow up on this project without a fairly steep learning curve being necessary.

The alternative that we have chosen to use in this project, is Microsoft's own JIT runtime called .NET. The disadvantage is that the runtime only works on a Windows operating system, thereby de facto ruling out the use of another operating system on our touchscreen PC's. The advantages, however, outweigh by far this disadvantage:

- One can program in different languages targeting the same .NET functions: Visual C#, Visual Basic, F#, etc.
- The integrated development environment, Visual Studio, is top-notch, and an industry standard. Its use is free in an academic environment[21].
- By only being available on Windows, its routines are speed-optimized for this operating system, and no platform-specific code is necessary.
- Access to the serial communications port is quick and easy. Seeing that all of our equipment is controlled through serial communication links, it is a big advantage.

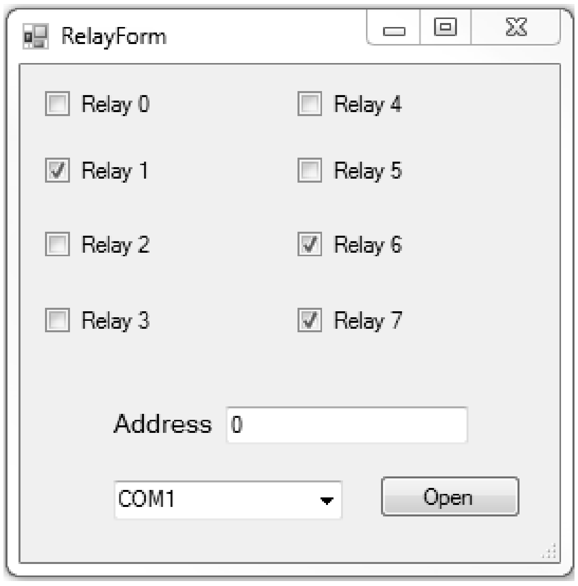
[19] JIT: Just In Time, referencing the fact that the code is translated to 'native' code as and when it is needed.
[20] Pronounce: 'cute'.
[21] Please see http://www.dreamspark.com for details.

- There are extensive graphical frameworks available. The most commonly known is WinForms, which is a drag-and-drop way of creating a user interface. That interface then integrates with the main program. WinForms has already been around since the time of Windows 95, and is thus a mature framework. Its successors, Silverlight (for web and desktop) and WPF (especially for a desktop environment) provide more integrated eye-candy, and a development process that separates the design from the code, which is handy if you want to introduce multiple custom UI elements dynamically, like we are planning to do.

Based on these factors, we have chosen WPF on the .NET framework for the development of our end-user graphical interface. The communications libraries, as seen later on, can first be developed and tested within a simple and small (KISS[22]) WinForms program to verify their functionality. This is also a strong point of the .NET system: a library written for one of the frameworks can be reused in another without much further work being necessary.

5.4.2 PC-side protocol verification

The very first tangible piece of interaction software that has been written, was a small front-end for our custom serial bus. It was created for the sole purpose of checking whether the protocol we made up was working, so it has a rather ugly layout. But since its prime purpose is to be useful, it sufficed.


RelayForm

- [ ] Relay 0
- [x] Relay 1
- [ ] Relay 2
- [ ] Relay 3
- [ ] Relay 4
- [ ] Relay 5
- [x] Relay 6
- [x] Relay 7

Address: 0

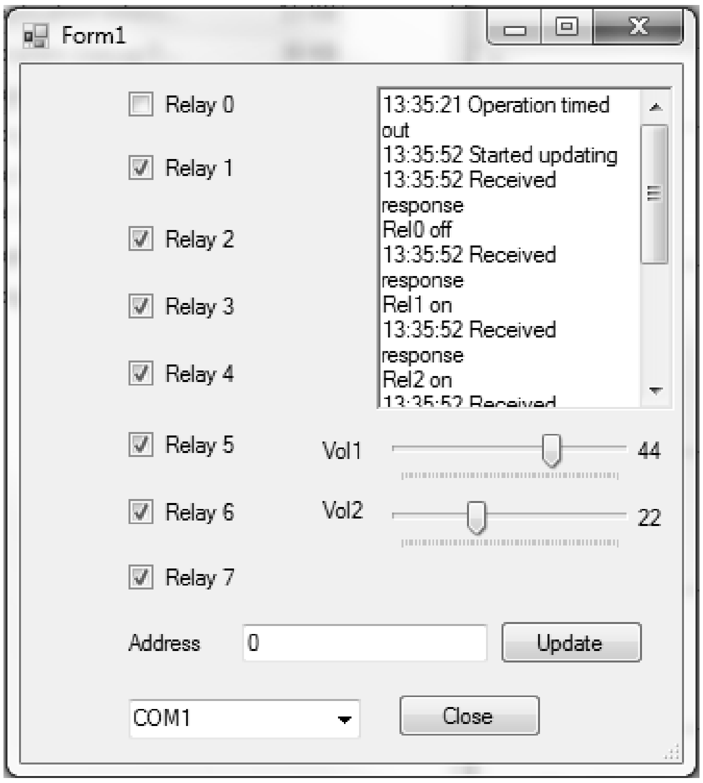
COM1

Figure 31 First version of the checking software

This is an example of the first version that was created. Debugging primarily took place within Visual Studio through the use of breakpoints and local variable inspectors, and the layout has been created by simply dragging and dropping the various elements into place, and tightly coupling them to underlying library calls.

[22] KISS: Keep It Simple and Stupid

Of course, while this was all working on the development computer, we needed to also verify its functionality on the actual target computer. For this project, that was the touchscreen PC. Because of its limits, however, it couldn't run Visual Studio satisfactorily, so an additional level of information about the bus events had to be implemented. That was why, throughout the weeks that it took to fully develop the final protocol specification, the testing program was updated to include several more features, which were mainly targeted at debugging. If an error occurred, it also gave a lot more information in the corresponding message.



```
13:35:21 Operation timed
out
13:35:52 Started updating
13:35:52 Received
response
Rel0 off
13:35:52 Received
response
Rel1 on
13:35:52 Received
response
Rel2 on
13:35:52 Received
```

Figure 32 Testing program in its final stages

This program proved to be very useful in checking the various pitfalls of our custom protocol, as well as helpful in deducing errors later on, when the actual user interface would cope with errors in a silent fashion.

In later versions, extra elements were added to verify the volume functionality of the extension cards, and the internal timeout mechanism, so that the bus does not block when a single response is not received. This is particularly useful to know when the cabling is being tested, so you can test whether the link is actually 100% reliable or not.

Now, this is the exterior look of the sample program, but what about the inner functionality? How do things work underneath the layers of user interaction methodology?

In order to fully understand the working of the communications library, a concept of every RTOS[23] must be explained: threading.

5.4.3 Understanding the concept of threading

Threading is a concept which, for the Microsoft line of operating systems, was first introduced in Windows 95 and NT. It is best explained using a small example:

Say you have to write a thesis, and use Microsoft Word to do so. If you open up Word, you start an application process. That process can then run on a single thread, or multiple if necessary. If it were to run on a single thread, you would have to wait each time you interact with the program until all results from that interaction had been calculated and executed before there would be another interaction possible.

For example, say you have already finished part of your thesis, and want to print a hard copy of your partly finished work. You would then go to the print dialog, and execute the print command. Of course, you are hard pressed with your deadline, and so start to continue writing while the printer is still busy.

At that moment, there are two active threads: the one that is printing, and the one that is converting your keyboard input to written text, properly formatted on the screen. So, threading increases the productivity and responsiveness of the system.

If the concept of threading had not been implemented, you would have to wait for the printer to finish printing, before being able to do anything else. That behaviour was normal in the pre-threading days of MS-DOS and Windows 3.1.

This principle works on both single and multicore processors. The operating system takes care of all threads, and gives each in turn some processing time on the processor should they need it. A thread that is waiting for something, only needs to check once in a while whether that action has already happened, which takes virtually no processing time. A thread that is important can also be assigned a higher priority, so it would get a bigger slice of the available processing time in each cycle.

Of course, as with most things in life, not all is well that seems well. The concept itself is essential for most, if not all, of the modern programs that need their user interface to be responsive while performing intensive or long tasks. But, there is a problem: if two threads try to access the same resource at the same time, a conflict can occur.

To continue our example: suppose that the printing command actually edits a part of your text after it has finished printing, but at that exact same

[23] RTOS: Real-Time Operating System, i.e. Windows, Mac OS, Linux, Android, etc. Not an RTOS: DOS.

moment, you are editing the same part. What would be the outcome? Which thread would take precedence and edit the text?

In fact: none. The application will crash due to being not ‘thread-safe’, a safeguard of the operating system. If the operations would go through, there would actually be a chance of a kernel error, which would then bring down the whole operating system instead of just your application.

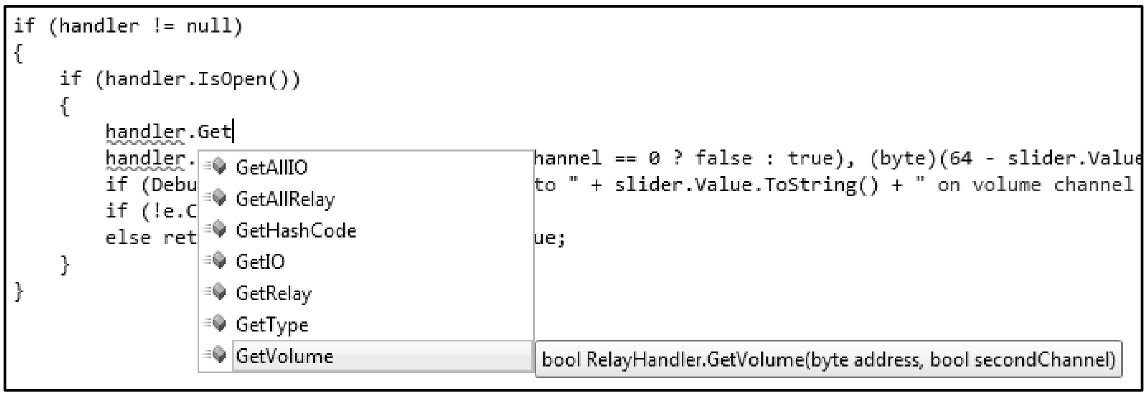
What does that signify for our application? It means that we should keep in mind that, due to the asynchronous nature of the serial bus, every call should be thread safe.

For the .NET framework, the serial port control handles its write and read operations on separate threads: if you want to send a byte, it is done on the thread that requested it. However, when a byte is received, the system creates a new thread to notify the program that a response has been received. The code, which then responds to that notification, must first wait for permission to access the common variables, before doing an operation on them, just in case a write and read operation were to be initiated simultaneously.

In addition, all functions that get called because of that notification will also be executed on the new thread. That results in a responsibility for the designer of the UI, because he, too, has to make his code thread-safe while handling responses coming from the communications library.

5.4.4 Our own protocol library: putting it all together

With the concept of threading explained, the setup of the protocol library should be clear. If a UI component wants to interact with the bus, it executes a method on the handler object, passing all necessary values. At the bare minimum, an address is needed, but for some functions, you would also need to pass a relay address, a volume, a relay value, a port value, etc. These are all documented in the respective code segments, and all variables are named after their actual function. This is important because of the IntelliSense function of Visual Studio: when you are typing your code, and it recognizes that you want to access a method, it displays a small popup with the required variables, and their names.



```
if (handler != null)
{
    if (handler.IsOpen())
    {
        handler.Get
        handler.                    hannel == 0 ? false : true), (byte)(64 - slider.Value
        if (Debu                    to " + slider.Value.ToString() + " on volume channel
        if (!e.C
        else ret                    ue;
    }
}
```

GetAllIO
GetAllRelay
GetHashCode
GetIO
GetRelay
GetType
GetVolume
bool RelayHandler.GetVolume(byte address, bool secondChannel)

Figure 33 Example IntelliSense popup

When a method is called, the library will check whether there is currently an action going on. An action can mean three things: there is a queue of commands waiting to be executed, there is a timer running before the next command can be sent, or there is a timer running because the previous command is waiting for a response. If there is nothing going on, the command is sent on the serial bus with the proper encoding, and a timer is started to either give the bus a chance to execute the command before sending the next one, or to give the addressed card the time to respond. Default times are 110ms between commands, and maximum 200ms to receive a response.

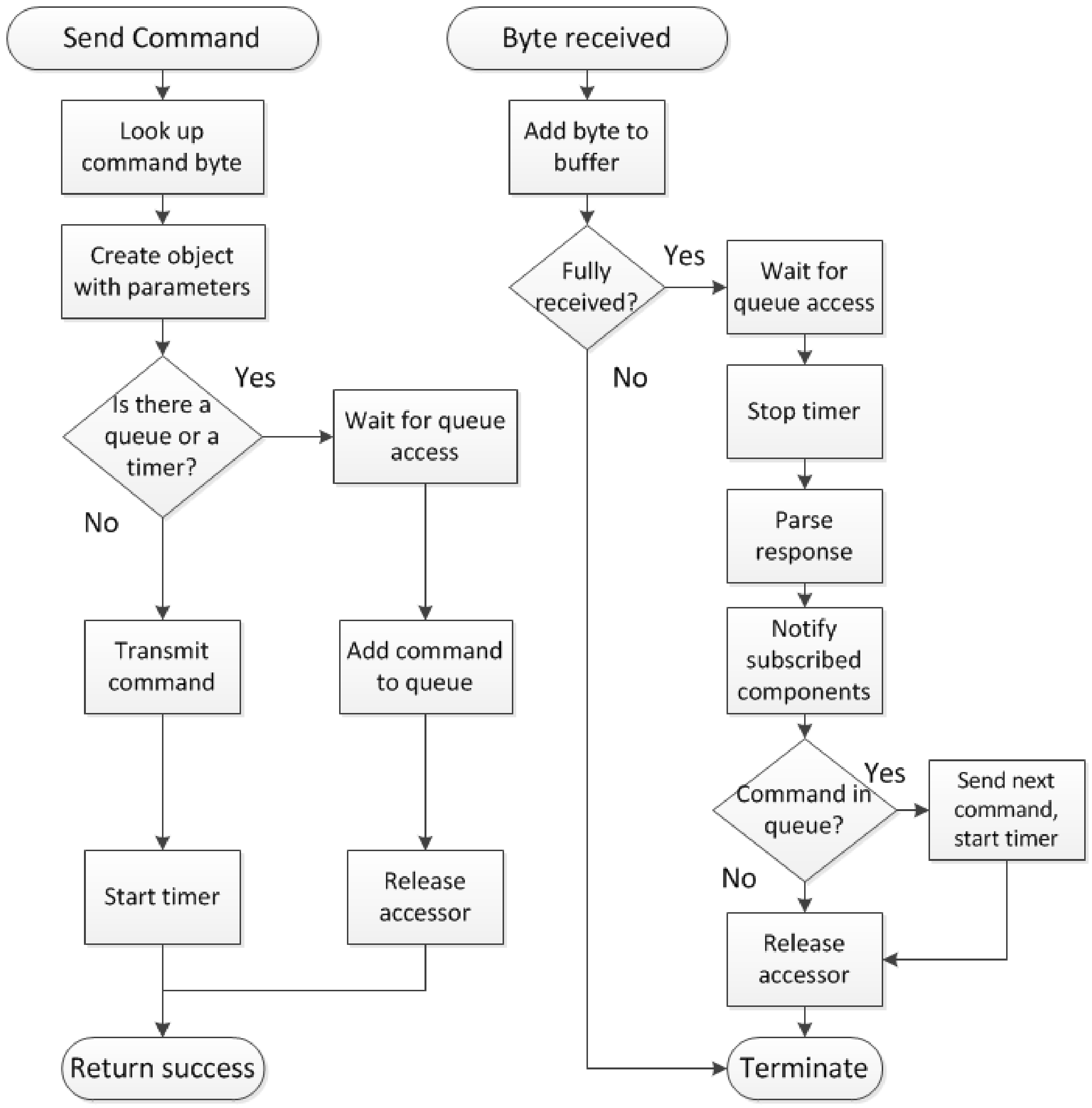


Figure 34 Flowchart of PC-side protocol handling

If the bus is busy, the command is added to the command queue to be sent when all the other waiting commands have been executed. If something is not right, for example the serial port has not been initialized, the method will return an error value.

As stated before, when the opened serial port receives a byte, it creates a new thread which calls the event handler of the library. This handler does basically the same thing as the receive routine of the firmware: it buffers the incoming data until it detects a command, and decodes the command. That is when things get interesting, though.

Due to the threading mechanism, the event handler first has to wait until it is granted access to the shared variables, like the command queue. This happens with an accessor object called a Mutex. This Mutex is passed around by the threads: a thread that needs access, requests it by calling a function on the Mutex, which blocks the thread until access is granted. Once the operation has completed, the thread has to call the release function of that same Mutex, so that access can be granted to the next thread requesting it. This guarantees that no two threads can access the protected resources at the same time, and thus generate an application crash.

So, when the receiving thread has been granted permission to access the necessary objects, it decodes the object, and raises an event that notifies the subscribed objects that a response is received, along with the contents of that response. Basically, every subscribed object is at this point able to execute some code to check if the response is important to them, and act on it.

After all the objects have been notified, the command that has been executed is deleted from the command queue, and if there is a next one, the next one gets executed, and the appropriate timer is started. And so the cycle renews itself.

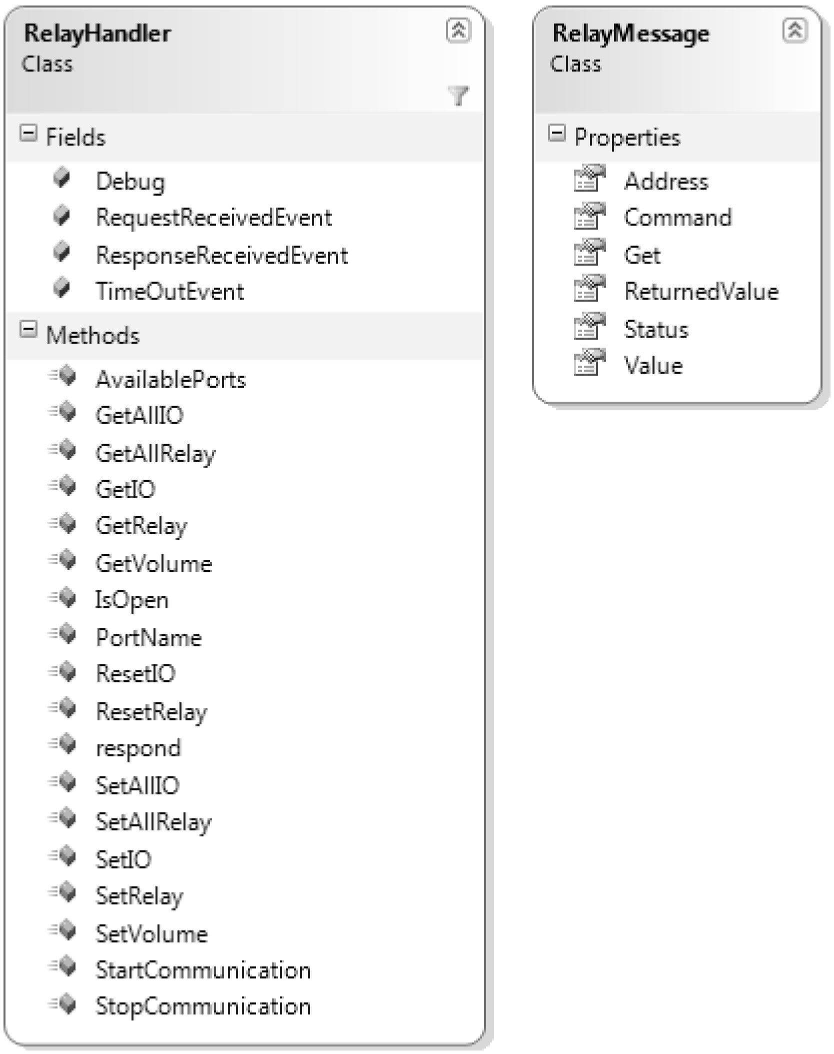


Figure 35 List of methods available on the bus accessing library

At this point, the functions for relay getting and setting, volume getting and setting, and I/O expander interaction are available in the library. Extensions can easily be made, because internally, the class uses a function called ‘SendCommand’ that takes a ‘RelayMessage’ as parameter. That RelayMessage object can be filled with custom data beforehand: the address byte, command byte and value byte need to be set, and the Boolean ‘get’ needs to be set to true if a response is to be expected, so that the right timer can be started.

As seen in the class diagram in Figure 35, there are four events that can be raised:

– The Debug event returns a text string with a debugging message as sender. This event can be subscribed to by the main program, in order to capture and display all relevant debugging data.
– The RequestReceived event is raised whenever a request has been received on the bus. This functionality, in combination with the ‘respond’ method, has been included at the time that the PC program was used to emulate a ‘virtual’ relay card, and thus had to respond to in-

coming requests. Under normal operating conditions, however, it should not be used, and can be regarded as deprecated functionality.

- The ResponseReceived event is raised every time a valid response has been received, and returns a RelayMessage object with the correct data as a sender. Every object that holds a reference to the RelayHandler instance can subscribe to this event in order to be able to listen for a response that concerns the object.
- The TimeOutEvent is raised whenever a response was expected, but not received within an appropriate timeframe (default 200ms).

5.5 Designing an intuitive user interface

With our core functional library completed, we could begin to focus on the creation of an attractive visual interface to display on the touchscreens. Since neither of us is a graphical designer, we had to think carefully, and have a few brainstorm sessions to come to a conclusion.

We ultimately decided that we would embrace a new graphical ‘language’ called Metro, instead of the more traditional layouts of text-filled buttons, frames and screens to switch to. The main reason for this choice is because of the effectiveness of the message: with Metro, it should be clear instantly where you have to press in order to access a specific function.

5.5.1 The Metro language

So, where does this so-called Metro language come from? For starters, it has been developed by Microsoft, and first used hesitantly in the operating system of their series of MP3-players. When they noticed that it gained traction with the wide audience, it was further perfected and implemented in their mobile OS revamp, Windows Phone 7, and is due to be implemented in the next version of Windows: Windows 8.

The inspiration for Metro stems from the clear signage at public locations: airports, train and metro/subway (hence the name) stations, parking lots, etcetera. These signs have been developed to guide the people as efficiently as possible to their right destination, and they do so through the use of icons where possible.

Should an extra guiding text be necessary, it is written in a clear, singular typeface, reducing the clutter on the signs. You will not find Comic Sans on a sign! The typeface used for Microsoft’s implementation of Metro, is Segoe UI.

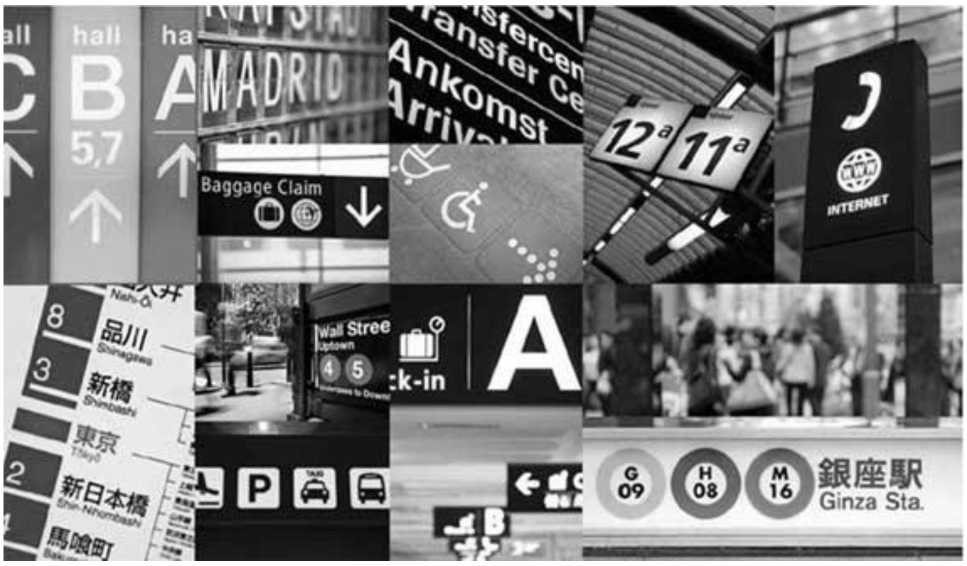


Figure 36 Inspiration for the Metro design language

One of the main eye-catchers of the Metro language interpretation is the use of square blocks called 'tiles'. These are filled with either a couple of words of text, or an icon and a small title, and are arranged in a tight grid.

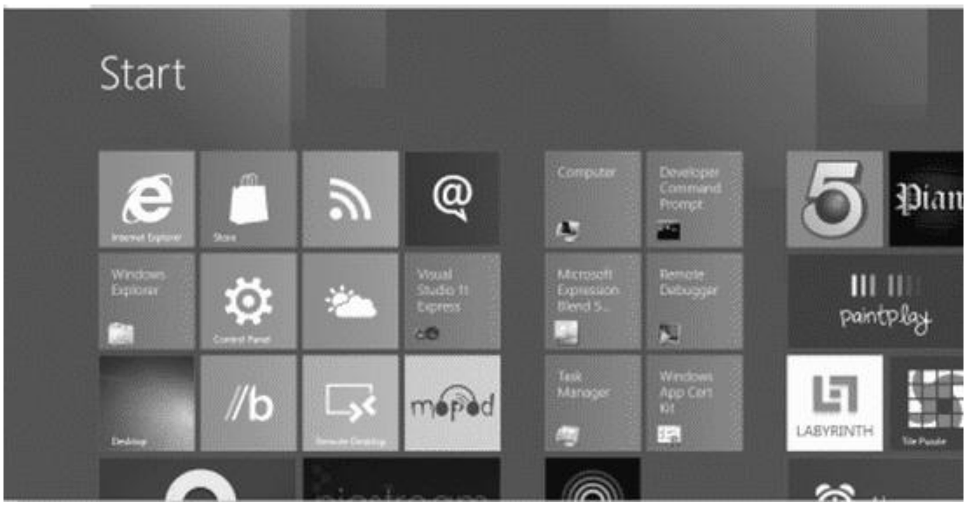


Figure 37 Metro tiles on a home screen

Another aspect is simplicity: There is one, and only one, contrasting colour used throughout the application. The background can be set to either light or dark, depending on the needs of the user and the circumstances, for example ambient light. When this happens, all UI black-and-white colours invert, to keep the text legible, but the one contrasting colour remains.

A final principle is called 'Content, not chrome', which is an extension of the 'fierce reduction' principle. It means that the application should place its content in the spotlight in a clear and prominent fashion, and reduce the impact of other visual elements to a minimum, or hide them altogether.

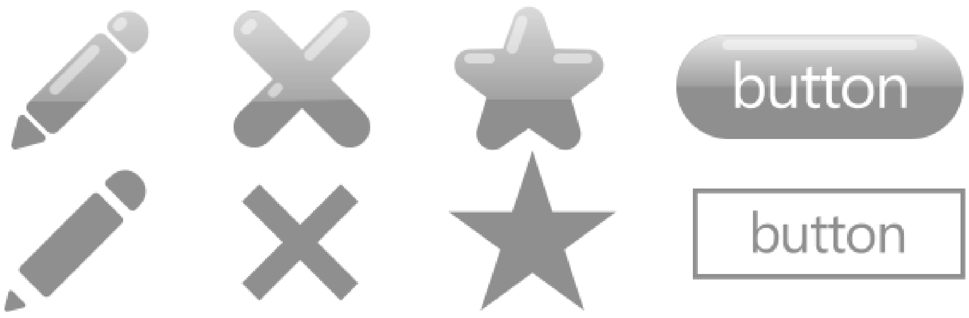


Figure 38 Difference between Glass (top) and Metro (bottom) icons

5.5.2 First sketches

But, how would we design our own application? We decided to go for a grid layout to further optimize the conveying of information and the fluidness and 'belong together' trait of the UI. This means that every object on-screen is rendered in proportion to its encapsulating or neighbouring elements.

We also decided not to have multiple screens which the user should navigate between by using forward buttons and back buttons. Instead, there would be a tabbed control to select either the main interface or the detailed control elements, and so switch between the front end and back end of the application.

Furthermore, a quick access bar to the right of the screen should give access to the most quickly needed functions: turn the lights on/off, and adjust the volume, in case the audio source is outputting a high volume.
A permanent bar under the screen should hold the tiles pertaining to the program itself, and not the functionality: exit, debug, and language selection features.

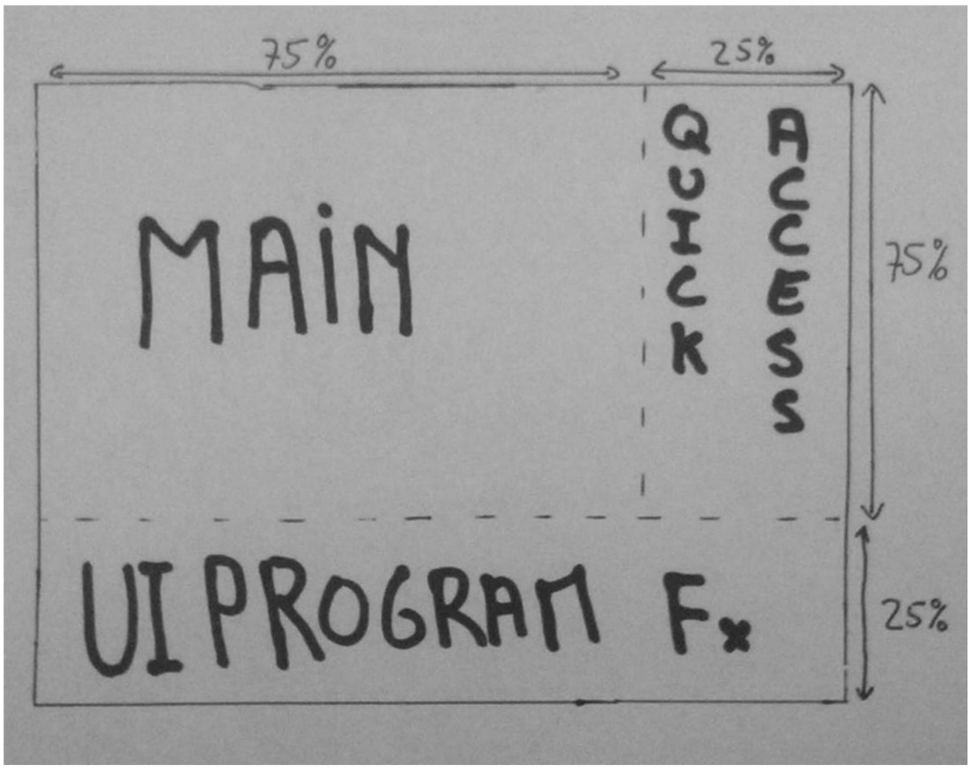


Figure 39 Sketch of the base view layout

The main view area can be filled with tiles, each holding a particular preset to put the room directly in the desired condition. The quick access bar is filled with chromeless buttons, because it would have been a waste of space to fill it with the same tiles as the main screen.

As for the tile, it consists of a uniform background color, an icon in either black or white depending on the background, and some small text title to clarify the icon a bit more.

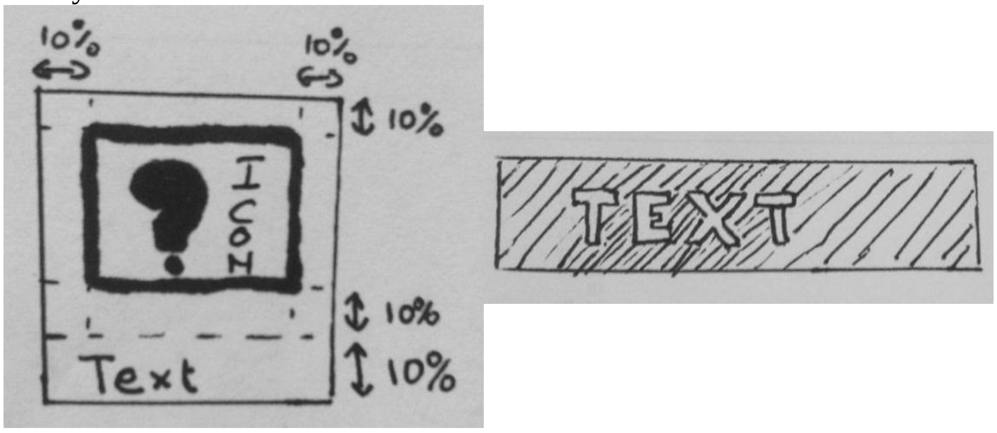


Figure 40 Tile and chromeless button sketches

As for the main field, on start-up, it is designed to hold three rows of tiles, either 1 or 2 units wide, for preset selection. As can be seen, it is shaped in a grid, to enhance the visual ‘flow’.

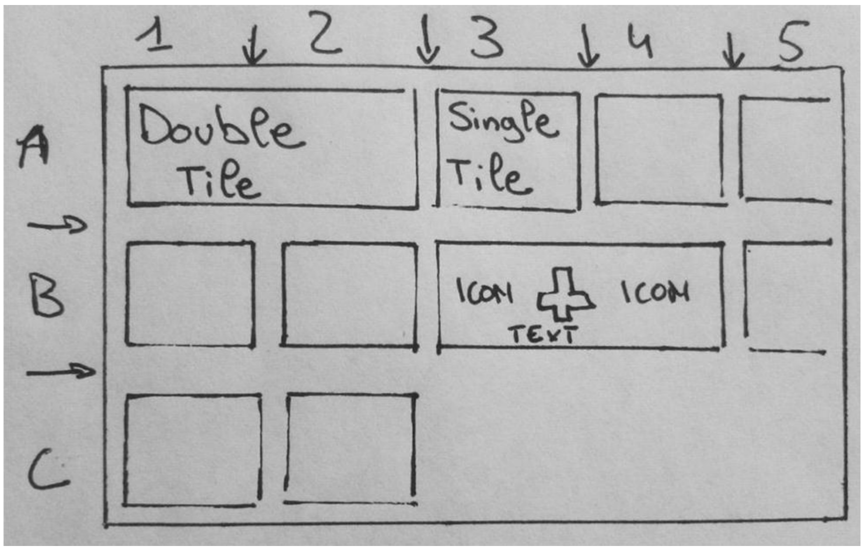


Figure 41 Organizing tiles on the main screen

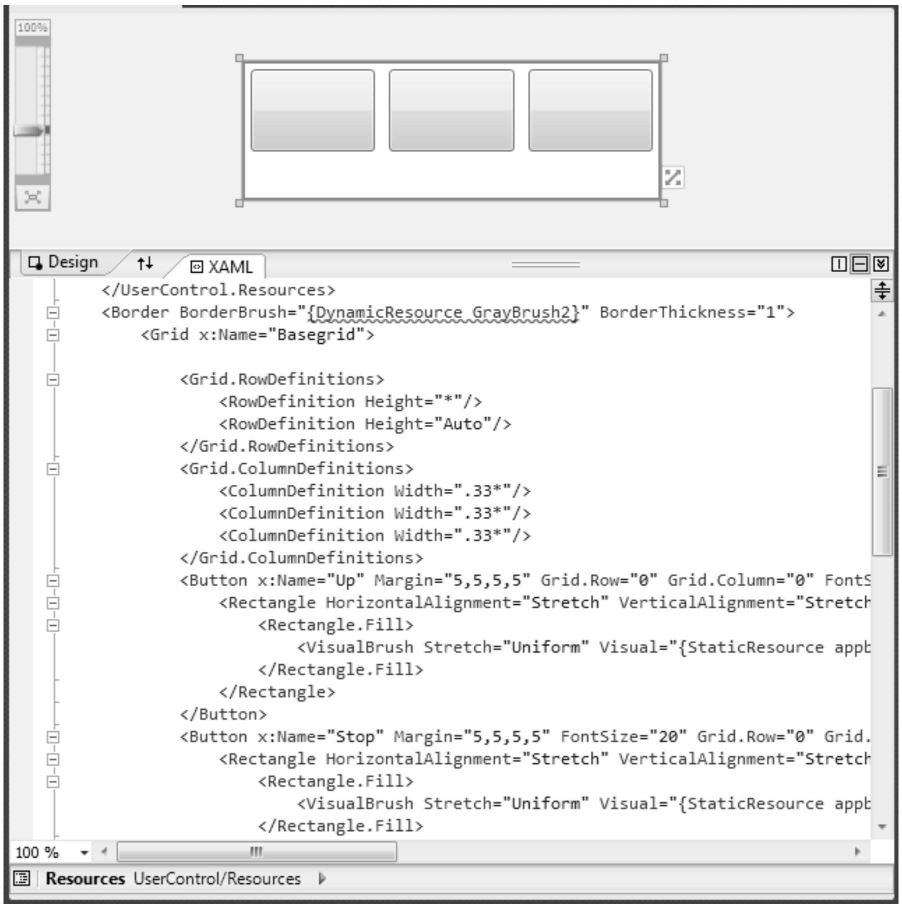
5.6 Creating reusable components

One of the main advantages of using WPF as a graphical framework is that UI elements can be regarded as small, stand-alone programs with their own graphical layout and interaction logic, therefore eliminating much of the interaction logic otherwise cluttering up the main program.

The UI object is, for the visual part, designed using a mark-up language called XAML[24]. This language is similar to XML[25], in that it uses the same style of 'tags' to define a visual tree of base objects.

For example, to define a grid, one simply uses the `<Grid>` and `</Grid>` tags, the former representing the start of the grid, and the latter the end of it. Every other object that then is placed between those tags, is a visual child of the grid object, and is thus enclosed in it.

It also provides a way to hard-code properties of the object within the XAML file: for example, within the start tag `<Grid>`, you can define a property that defines the background colour: you just use `<Grid Back-ground='White'>`. That way, all of the Grid's properties can be set at the time of design by the designer.



```
    </UserControl.Resources>
    <Border BorderBrush="{DynamicResource GrayBrush2}" BorderThickness="1">
        <Grid x:Name="Basegrid">

            <Grid.RowDefinitions>
                <RowDefinition Height="*"/>
                <RowDefinition Height="Auto"/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width=".33*"/>
                <ColumnDefinition Width=".33*"/>
                <ColumnDefinition Width=".33*"/>
            </Grid.ColumnDefinitions>
            <Button x:Name="Up" Margin="5,5,5,5" Grid.Row="0" Grid.Column="0" FontS
                <Rectangle HorizontalAlignment="Stretch" VerticalAlignment="Stretch
                    <Rectangle.Fill>
                        <VisualBrush Stretch="Uniform" Visual="{StaticResource appb
                    </Rectangle.Fill>
                </Rectangle>
            </Button>
            <Button x:Name="Stop" Margin="5,5,5,5" FontSize="20" Grid.Row="0" Grid.
                <Rectangle HorizontalAlignment="Stretch" VerticalAlignment="Stretch
                    <Rectangle.Fill>
                        <VisualBrush Stretch="Uniform" Visual="{StaticResource appb
                    </Rectangle.Fill>
```

Figure 42 Example XAML file and its rendered design

[24] XAML: eXtensible Application Markup Language, pronounce 'zammel'.
[25] XML: eXtensible Markup Language

The code part of the object is enclosed in a file called the ‘code-behind’. This is a C# (or C++) file, with at least one method in it, named after the control itself. This method gets executed on creation of the object. Of course, you are free to write and add other methods to extend the functionality of the object.

Elements defined in the XAML, can be accessed from the code-behind either by looping through all elements and selecting the one you need, or by name. The property ‘x:Name’ of a XAML object defines a static name with which your code-behind file can access it.

As becomes clear from this explanation, the XAML file is used for the static layout of a UI element, while the code-behind is used to change parameters, views, or update the layout dynamically by changing properties of the in the XAML defined objects. This way, a designer can create a good-looking object without worrying about the interaction logic, and a coder can code the interaction logic without having to worry about the design.

Another advantage over WinForms, is that you can easily design multiple custom UI elements, and then have your main program instantiate possibly multiple instances of them. Even though they are all coded the same, they won’t conflict with each other. So, if you create a UI element with 2 buttons, called button1 and button2, this element can be presented multiple times, without having to rename the buttons of the second element to prevent naming conflicts. It is a good way to create reusable code.

5.6.1 Tile

The first UI control (in WPF, a custom UI control is called a ‘UserControl’) that was created, was the Tile. A tile, in fact, is just a normal button, which is why we start with subclassing the Button object. Subclassing is using another class as a base for your class, and then extending on it.

The normal button style, however, does obviously not fit our purposes. Therefore, we make some changes to the default button style that only get applied to our Tile object: We remove the border, create a squared layout, add a property for the title and the icon, apply the contrast colour to the background and the text colour to the foreground, etcetera. In the end, the result looks like this:

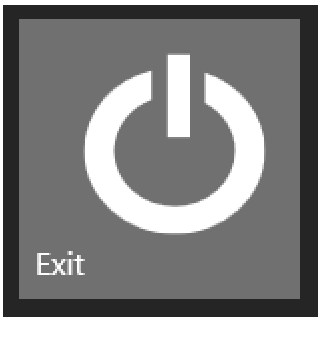


Figure 43 Example custom styled Metro tile

5.6.2 Preset tile

Our custom tile now does what it should do, but for our preset tiles, we need some more functionality. Seeing that the preset tiles should launch a number of commands when clicked, we need the control to hold a list of commands to be executed on the click.

Therefore, we create, again, a new class, subclassing our Tile object, called CTile. We add a property Commands that holds a list of AuditoriumCommand objects, which can be accessed by the program when the tile is clicked.

Also, since we are dealing with a multilingual interface, a dictionary had to be added that contains the translations of the tile's title, and a property to set the language. When the language is set from another portion of the code, external to the CTile object, the CTile object should automatically update the displayed title to the correct text for the set language. This is done through the use of a principle called 'data binding'.

Data binding is a very handy feature. You can bind a property in XAML to a code-behind variable using the syntax {Binding Path=…} with the name of the variable replacing the three dots. Then, when the variable is updated in the code, the layout gets notified, and updates the view to display the new value of the variable. To make this happen, the code-behind should raise the PropertyChanged event with the name of the variable that has been changed, after its value has been changed, in order to notify the layout of the change

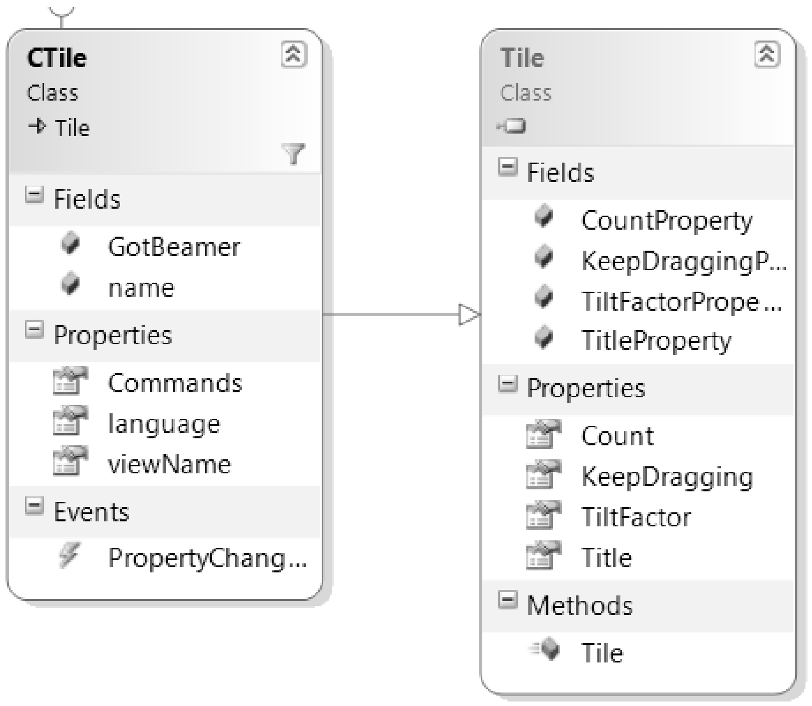


Figure 44 Class diagram of the Preset tile.

5.6.3 Light control

The light control is used to control a single light, which is either on or off. To do this, it uses two buttons, of which one at a time is enabled, depending on the status of the light. Obviously, if the light is on, the off button is enabled, and vice-versa.

To check on the status of the lights, it needs to hold a reference to the appropriate RelayHandler, so that it can request the right relay's status. It also needs to hold a variable with the address of the card and relay number on which the light contact is situated.

As with all the controls, every piece of text should have an attached dictionary to contain the translations, so that when the language is updated, the proper text can be displayed.

On creation of the LightControl, it initially does nothing. However, when the reference to the RelayHandler is updated, it subscribes to the ResponseReceived event of the new handler (because it can always receive an unsolicited response when the light has been switched via the keypad), and requests the status of the relay that is indicated by the user control's address and channel variables. When that request gets a response, the buttons are updated to reflect this change.

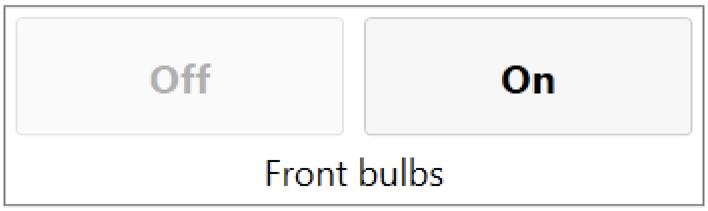


Figure 45 A finished LightControl UI object

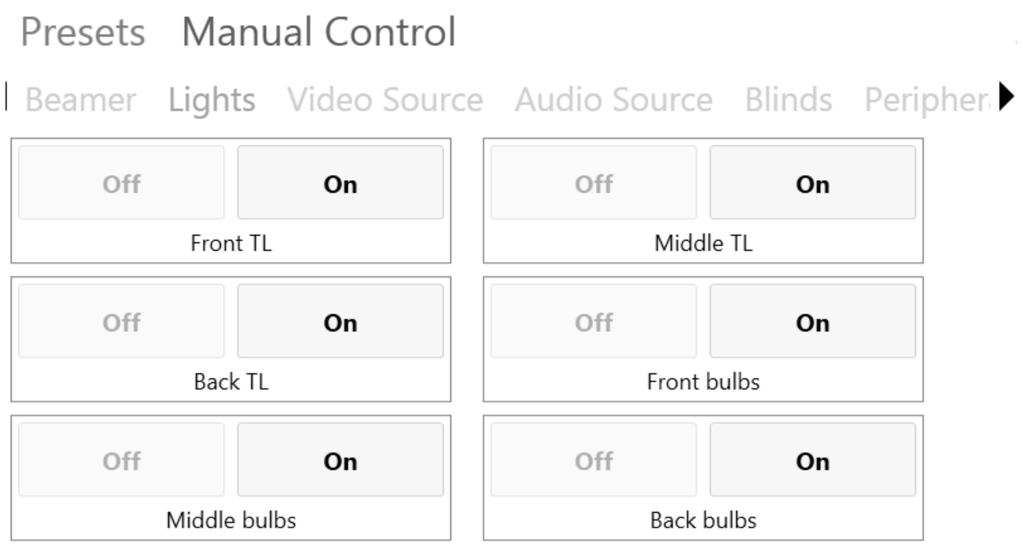


Figure 46 All of the lights in a room arranged in the program

5.6.4 Blinds control

Like the LightControl, the BlindsControl needs to hold a reference to the appropriate RelayHandler, so that it can set the right relays and request their status. It also needs to hold a dictionary for all translations.

The BlindsControl has three buttons: up, down and stop. When the blind is going up or down, only the stop button can be used. Likewise, when the blind is not in motion, only the up or down button can be used.

When a motion is started, a timer is initialized, and set to expire when the maximum time of travel has elapsed. This is to protect the motor in the event that the built-in limit switches should fail. When the timer elapses, it resets the relay that had been set, so that the motion is stopped, and the supply is cut off from the blind's motors.

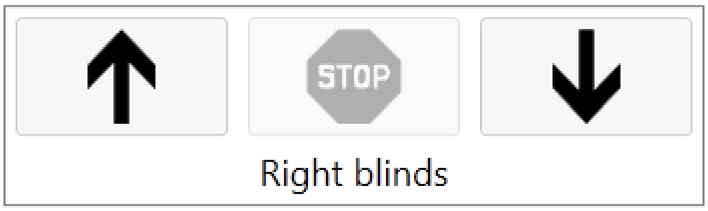


Figure 47 Finished BlindsControl

Except for control through the buttons, the BlindsControl also has a public method, by which external coded objects can initiate a movement. This is important, because the commands attached to preset tiles can include a command to move the blinds. Those commands get interpreted by the main program, which should then look up a reference to the right Blind-sControl, and execute that method to have the blinds move to the desired position.

As a safeguard, the BlindsControl also keeps the status of the blinds in memory. For example, if the blind has just finished its full movement up-wards, there will be no action when the 'up' button is clicked again, or when the 'up' command is issued through the Move method.

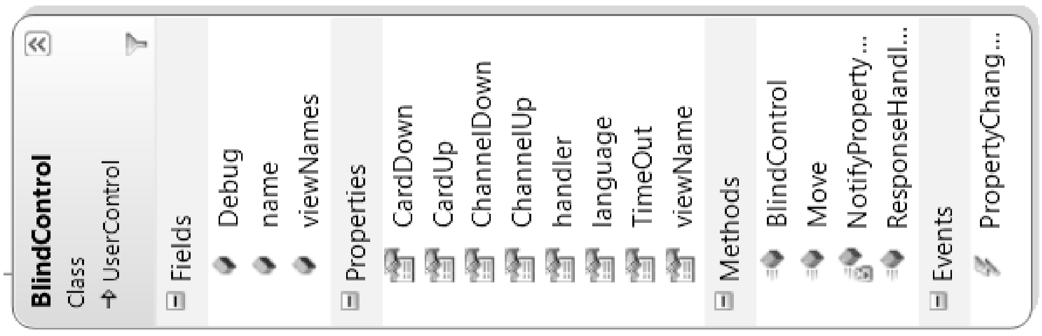


Figure 48 Class diagram of the BlindsControl

5.6.5 Volume control

Like the two previous UserControls, the VolumeControl also needs to hold a reference to the appropriate RelayHandler to be able to set and request the volume level of the given attenuator. For this, it needs the address of the relay card on which the volume expansion card is installed, and an indication whether it is coupled to the first or the second attenuator on that expansion card. It also needs a dictionary to translate the name of the volume control into the appropriate language as requested by the main program.

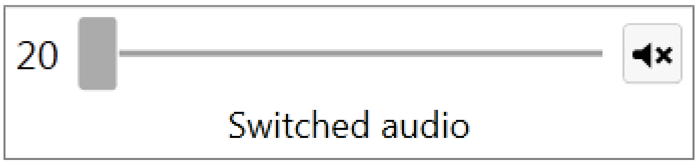


Figure 49 Sample view of a VolumeControl

The volume can be controlled by dragging the slider over the track. It can also be muted by using the mute button, which will then disable the slider. Once the mute function is turned off again, the volume level will be restored to its previous value, and the slider will be enabled again. The status of the mute function is indicated by colouring the background of the mute button with the contrast colour when the function is active.

In an effort to reduce the bus traffic, the volume not updated with every value the slider passes as the user slides it over its track. Instead, the volume is only updated when the difference between the previous value and the instantaneous value is bigger than 5 (out of 64 possible steps). When the slider is released, the volume is then updated to the exact level where the slider has been released.

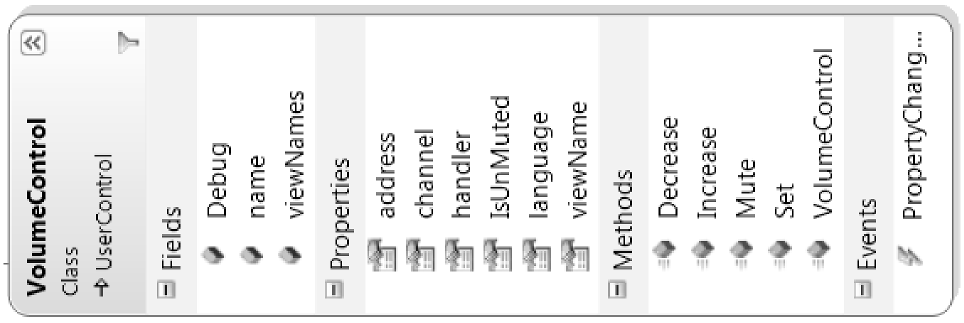


Figure 50 Class diagram of the VolumeControl

The VolumeControl also contains methods to enable the main program to issue commands to it, like increasing and decreasing the volume, setting it to a specific level, and controlling the status of the mute function.

5.6.6 Switcher control

Because we had to deal with multiple types of switchers, there was a need to create multiple Switcher controls. Because the A auditorium was tackled first, the controls pertaining to that one have been created.

Kramer 12-port double bus switch
This switch has two controllable busses, whereby one is used to switch the audio, and one is used for the video. Because the audio and video switching controls will end up in different tabs in the main program, the serial port interpreter and the user controls had to be separated from one another.

The serial protocol used for the Kramer switch regrettably doesn't have a function to request the currently selected input from the device, so that we have to buffer this ourselves, and hope the command will be received by the switch. This, however, makes the handler easy to write: it's just a matter of sending the right two bytes to the switch. For full documentation on the protocol, please consult the datasheet on the attached CD.

The user control for interaction consists of 12 buttons, each representing an input on the switch, and each user control representing a bus. That means that two user controls can be coupled to one protocol handler. At this moment there is nothing that prevents more than two user controls being coupled to the same handler, but that situation would be non-critical, just annoying.

Furthermore, the user control should hold a dictionary with the translation of the in- and output names, because each input and output can be assigned a name, which is then displayed vertically in order to fit in the layout. That, however, means care should be taken not to give too long a name to an input, because the user control would then clip the name because of its maximum height.

When a button is pressed, the serial handler gets a command to have the switch select the input corresponding to that button, on the output defined by the variable in the user control. The pressed button's background is then coloured with the contrast colour to indicate that the command has been sent, and that that input should now have been selected.

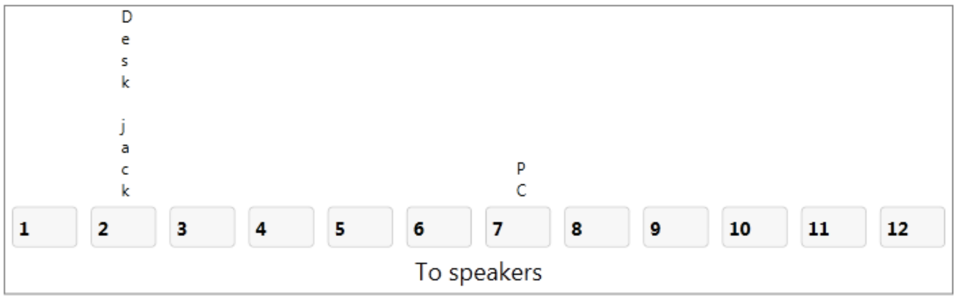


Figure 51 User control for a 12-input Kramer switch

Extron 4-port VGA switch

This switch has four VGA inputs, and one output. It is controlled through a serial port, using a protocol designed by the manufacturer. For more details on this protocol, please review the protocol manual on the attached CD.

Seeing that the Extron switch is not able to be operated on a bus, but only by direct serial communication, the serial port handler has been built into the user control. This means the computer needs an available serial port for each Extron switcher. The serial port interfacing logic, being included in the user control, means less source code files, and thus less clutter in the programming environment.

An advantage of the Extron switch over the Kramer switch is that the Extron can return the currently selected input when it is requested. So, when the user control is initialized, it requests the current status from the switcher, and colours the background of the input button corresponding with the actual selected input.

Another nice feature of this switch is that it automatically sends a message to the PC when the inputs' signal statuses change. So, we have added a visual indicator to the user control that colours green when the signal on a specific input is present and grey when there is no signal. This status message is sent unsolicited, so the indicator's colour will change the moment the corresponding status changes.

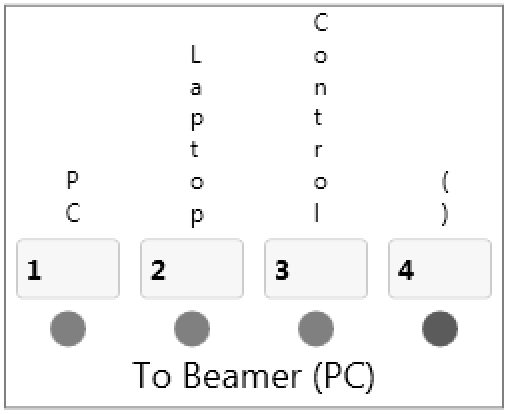


Figure 52 User control for an Extron 4-port switch

5.6.7 Beamer control

The beamer user control is a special one, in that it is not reusable per se. It is tailored to control one type of beamer only, but the general layout can be recycled eventually for other types of beamers, albeit with another code-behind file.

That said, the beamer user control which was developed first, was developed in function of the A auditorium, and so coded with the Eiki protocol for the LC-X71 data projector.

At first sight, this should have been a straightforward exercise. A document from Eiki exists (and is included on the attached CD) which details all the protocol specifications and interactions. This meant that all that had to be done was simply to send the right serial commands and update the layout to reflect the status of the beamer.

However, after completing the interaction logic, the beamer would not respond. Seeing that the serial port communications are handled within the user control because there can be only one single projector attached to a serial port, and that the user control already had a debug handler, we started examining both the input and output of the serial channel.

As it turns out, the projector does not comply with the specified protocol. Sometimes it returns the right values, luckily, but other times it returns nothing, or just plain bogus characters. Now, how are we to create a fluent interface when the beamer will not even properly acknowledge a given command reliably?

For example: if the program issues the ‘menu on’ command, the beamer would respond with something like 0x16 0x25 0x36 0x32, which makes absolutely no sense, instead of the 0x06 0x0A (ASCII[26] for acknowledge, linefeed) that it is supposed to return on successful reception of the command. We know, however, that the command was successfully received, because the menu had been activated.

Because of this, the user control of the beamer has been redesigned to implement a ‘fire and forget’ method for all non-critical functions, and to repeat the command for all critical functions until a valid response has been received (can sometimes take up to ten retries). The critical functions are the power-on and power-off functions, because they have an inherent lag to them during which no commands can be sent to the beamer until it returns to a ready state, and the mirroring function, because it persists through a power cycle of the projector.

As a side effect, all preset tiles are blocked from operating until the beamer is in a ready state, to avoid conflicting operations. In addition, the whole beamer user control is blocked when there is an operation taking place that requires an acknowledgement.

[26] ASCII: American Standard Code for Information Interchange: a character set encoding that translates characters and control codes to a 7-bit binary code.

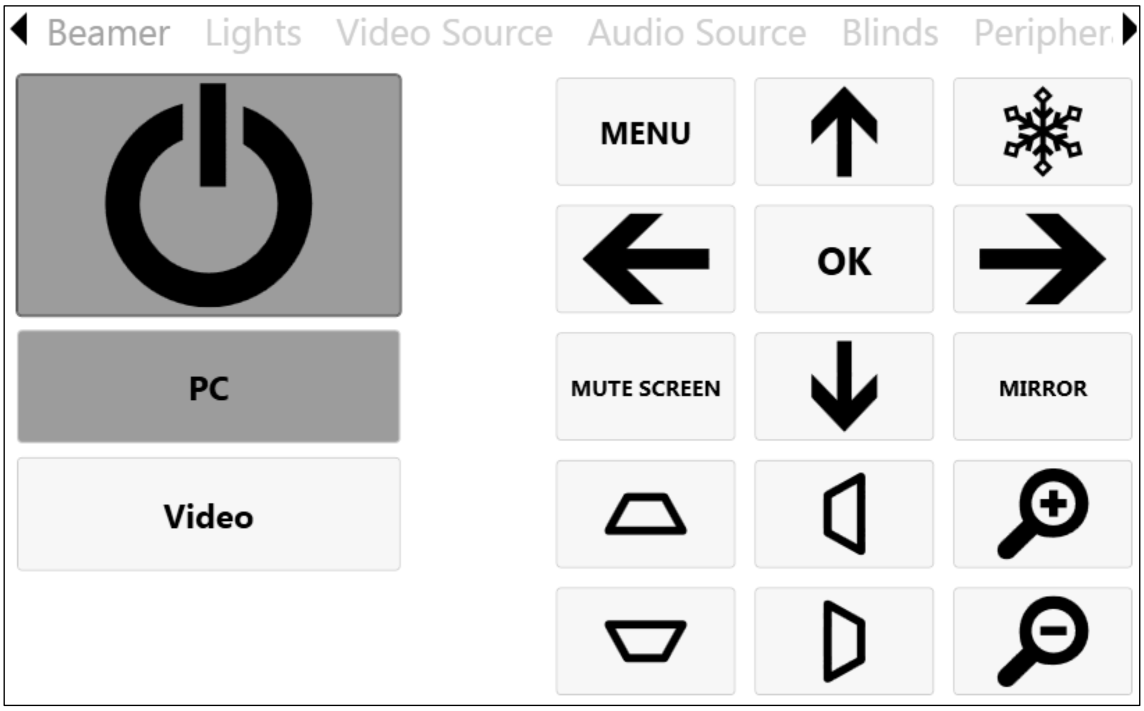
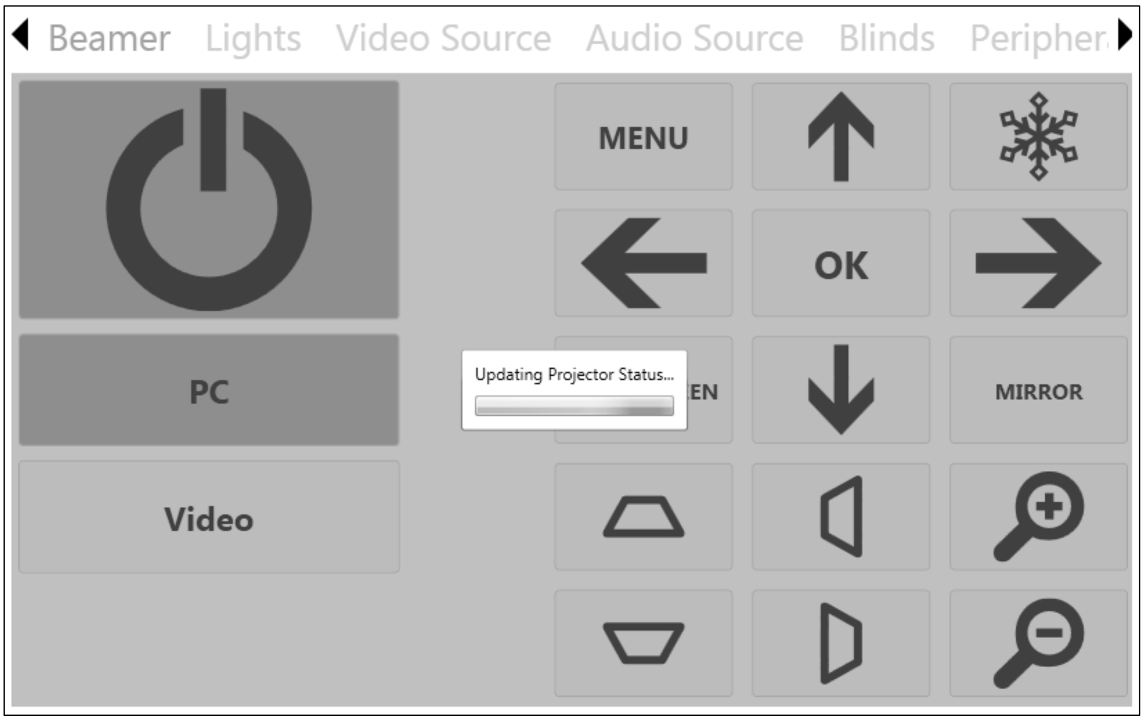


Figure 53 Data projector user control



Updating Projector Status...

Figure 54 Data projector user control in a busy state

5.7 Practical implementation

With all of the user controls now made, a final layout could be made. To do so, the home screen was divided in a grid, as illustrated in the design sketches previously shown.

Practically, that meant creating a 2x2 grid in XAML, and combining the bottom two cells into one. Margins were added to leave some room between the visual elements.

To display the tiles, a wrappanel has been used. This control automatically places its children elements next to each other, starting a new line if necessary. The advantage is that, when the child elements are sized uniformly, and their margins are set the same, they become laid out in a nice and orderly grid. Both the preset tiles and the UI tiles at the bottom of the screen are enclosed in such a wrappanel, albeit in two different ones.

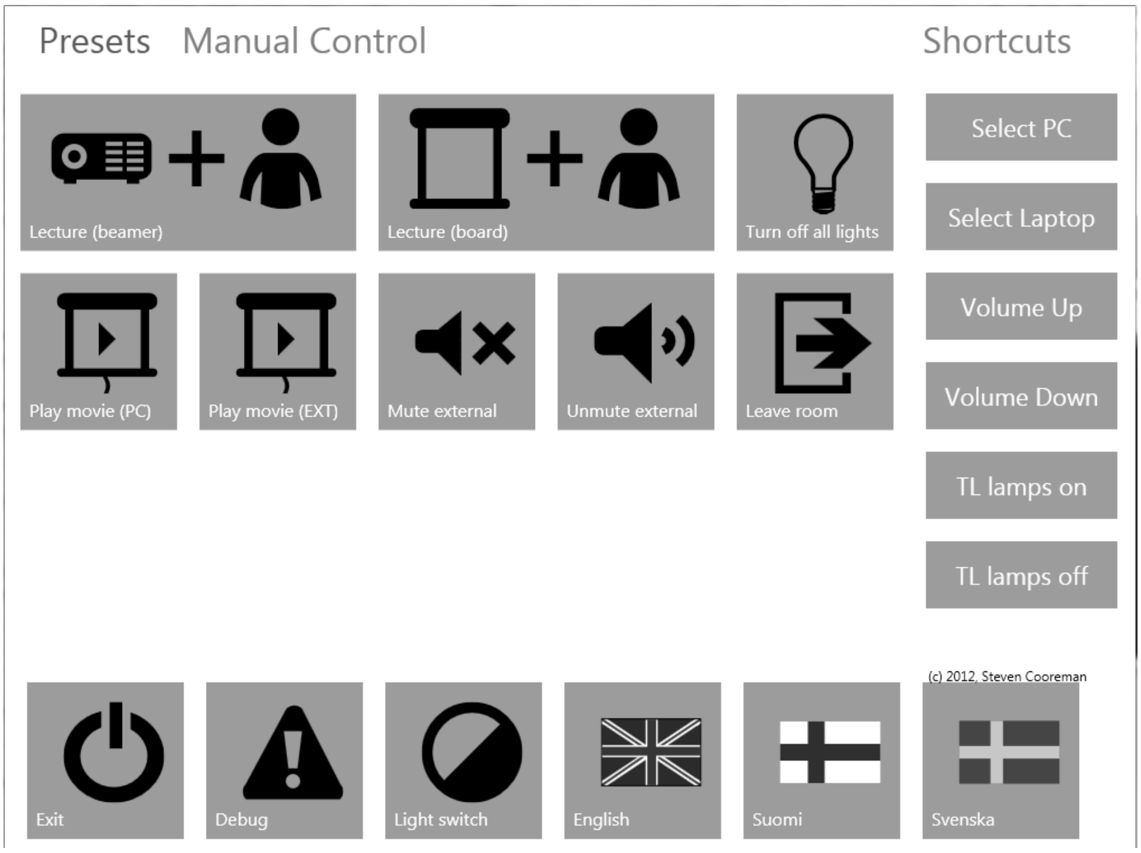


Figure 55 Final layout of the home screen

The shortcuts are simple chromeless buttons in a listbox, which automatically orders its elements in a list. The advantage is that its collection of elements can be accessed as an enumerable object in the code-behind, so the code does not have to keep a reference to each button as it is created. Instead, it just looks up the button in the collection of the listbox, and only keeps a reference to said listbox.

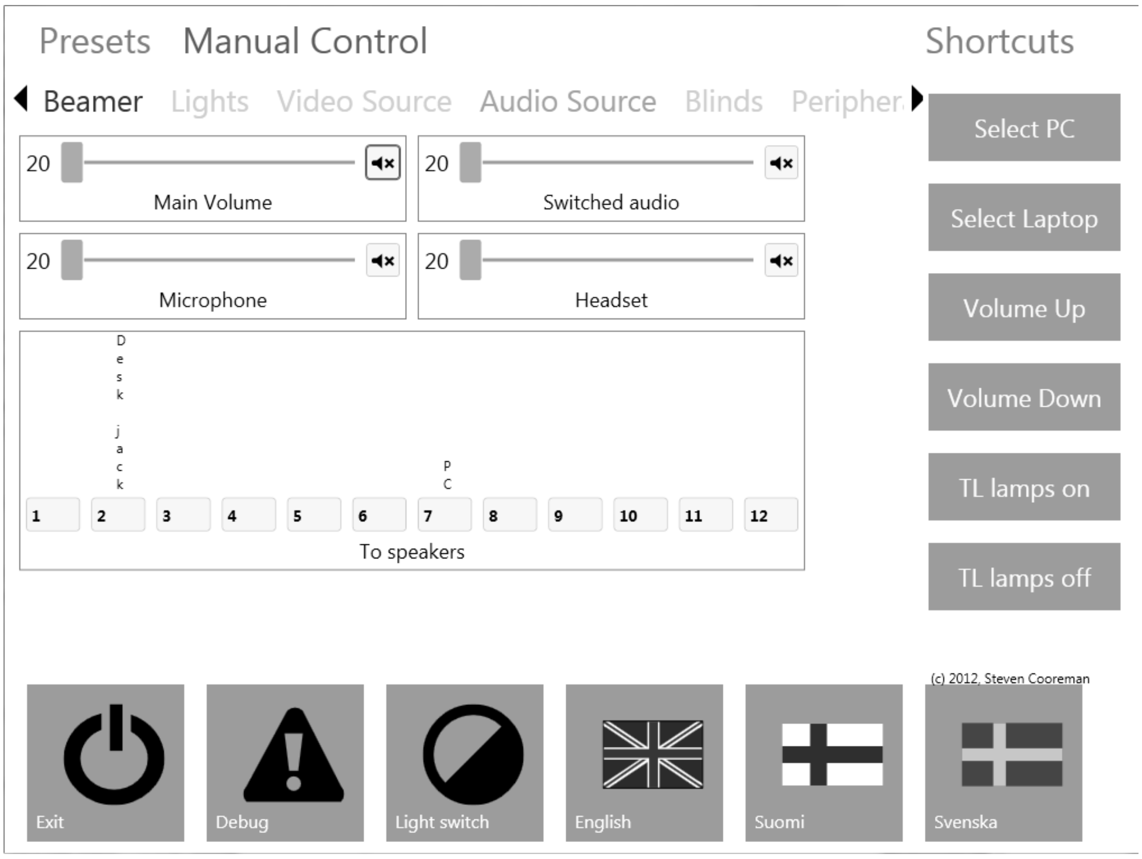


Figure 56 Manual control screen

The manual controls are all enclosed in their respective tab panels. The beamer control is enclosed in its own panel because it is just one user control instance. The other controls are also arranged in their tab using a wrappanel, just like the tiles on the home screen. And, like the listbox, the wrappanel keeps track of its children elements' references to reduce the use of hard-coded objects in the main program.

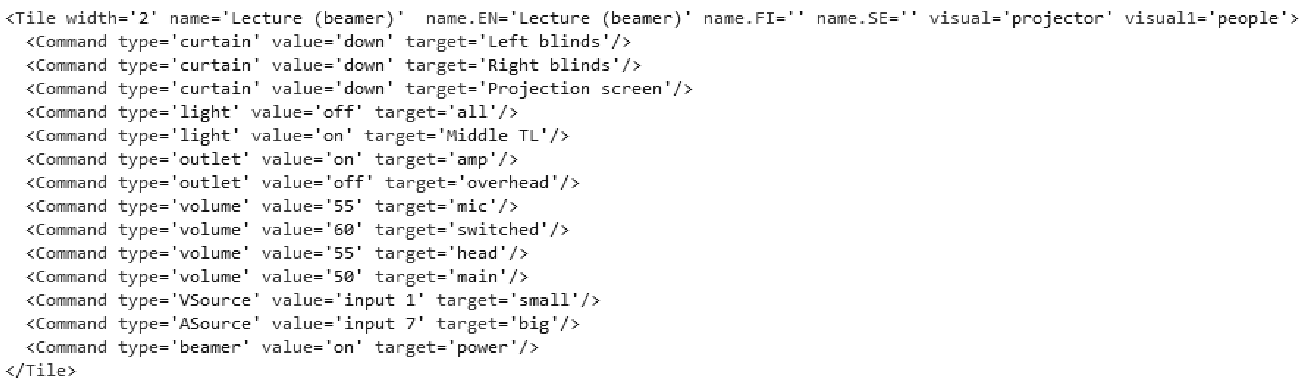
As illustrated by Figure 56, there can be multiple different UI elements enclosed in the same wrappanel, as long as due care is taken with regards to the width and margin of each element. In order not to break the visual structure, the elements should line up nicely next to, and above and under each other.

Even if multiple different elements exist in the same parent element, the program can still differentiate between them when it cycles through the child elements in order to get a reference to a specific object (for example, to execute a command that was bound to a preset tile). When it requests all child elements of the parent object, the program simply includes a specifier to indicate which type of child elements it is expecting, in order to avoid casting errors and subsequent crashes.

5.8 Abstraction of the room parameters

After all the user controls are made up, they should be placed within the program in a dynamic fashion, taking the room's requirements and equipment into account. This is a major point of functionality, because it allows the program to be reused in a different environment with the same peripherals, simply by changing a configuration file.

XML was chosen as the mark-up language for the configuration file, because it is fairly intuitive. In XML, the document represents a tree structure of objects, each object represented by an opening and a closing tag with its name. The opening tag contains any properties that the object might have, and between the opening and closing tags are the child elements of the object. This way, a tree structure can be illustrated by drawing the objects' relation to each other.



```
<Tile width='2' name='Lecture (beamer)'  name.EN='Lecture (beamer)' name.FI='' name.SE='' visual='projector' visual1='people'>
  <Command type='curtain' value='down' target='Left blinds'/>
  <Command type='curtain' value='down' target='Right blinds'/>
  <Command type='curtain' value='down' target='Projection screen'/>
  <Command type='light' value='off' target='all'/>
  <Command type='light' value='on' target='Middle TL'/>
  <Command type='outlet' value='on' target='amp'/>
  <Command type='outlet' value='off' target='overhead'/>
  <Command type='volume' value='55' target='mic'/>
  <Command type='volume' value='60' target='switched'/>
  <Command type='volume' value='55' target='head'/>
  <Command type='volume' value='50' target='main'/>
  <Command type='VSource' value='input 1' target='small'/>
  <Command type='ASource' value='input 7' target='big'/>
  <Command type='beamer' value='on' target='power'/>
</Tile>
```

Figure 57 Extract from a configuration file

By interpreting this configuration file when the program starts up, all relevant objects can be loaded dynamically, and placed in their right boxes. Of course, when there is a mistake in the configuration file, the program will malfunction too. Subsequently, care has to be taken when creating the configuration file.

<u>Basic layout of the configuration file</u>

The basic layout of the configuration file is as follows:

- The whole configuration file is enclosed within a 'Settings' tag.
- The global variables are enclosed in a 'Preferences' tag, and have to be defined:
    - The PIN code for exiting the program and activating the debug mode, is defined as the property 'pin' of a tag 'PIN'.
    - The contrasting colour, and the default start-up theme are defined in the tag 'Interface', as properties 'accent' and 'theme' respectively. Acceptable values for 'theme' are either 'Light' or 'Dark', and the colour can be chosen from 'Blue', 'Green', 'Orange', 'Purple' and 'Red'.
- The available serial ports of the PC have to be defined in a 'SerialPorts' tag.
    - Each serial port is a 'Port' tag, with the properties:
        - 'name': the system name, for example 'COM2'

        - ‘alias’: the name that we will use in the configuration file to reference this port
        - ‘speed’: the baudrate on which the port should be initialized
        - ‘databits’ the amount of databits per transmission
        - ‘stopbits’: the amount of stopbits per transmission
        - ‘parity’: whether parity should be calculated. Possible values are ‘none’, ‘even’ and ‘uneven’.
- The relay cards that are attached to our own bus system are defined within a `<ExtensionCards>` tag.
    - Each relay card has its own `<Card>` tag, with the properties:
        - ‘number’: the number by which we will reference this card in the configuration file.
        - ‘type’: the type of card. At this moment, only the type ‘relay’ can be used.
        - ‘ports’: how many relays there are on the card. Can be used for possible future cards with more or less relays per card.
        - ‘address’: the address on the bus of the relay card
        - ‘interface’: the alias of the serial port on which the card is located.
    - Each relay card can also have an extension card, which is added within the Card tag with an `<Extension>` tag. The Extension tag has the following properties:
        - ‘type’: the type of extension. Currently, only ‘audio’ exists.
        - ‘name’: the name with which the extension functionality will be referenced.
        - ‘channel’: only for audio extensions. Indicates the audio channel of the card (0 or 1).
        - ‘name.EN’, ‘name.FI’ and ‘name.SE’: The name of the extension that should be displayed to the user, in each language.
- The additional, controllable equipment is defined in the ‘Peripherals’ tag. Possibilities are:
    - `<Beamer>`, with the properties:
        - ‘type’: type of the beamer
        - ‘interface’: the alias of the serial port on which the beamer is located.
        - Currently, only one beamer can be defined, and only the type ‘LCX71’ is implemented.
    - `<MatrixSwitch>`, with the properties:
        - ‘type’: type of the switcher. Currently, only ‘VS1202’ and ‘VGArs4’ are implemented.
        - ‘interface’: the alias of the serial port on which the switcher is located.
        - ‘name’: the name with which the switcher shall be referenced in the configuration file.
    - Each MatrixSwitch has multiple inputs and outputs, defined by `<Input>` and `<Output>` tags with the properties:

    - ‘name’: name with which the input or output will be referenced.
    - ‘pos’: the number of the input on the switch
    - ‘bus’: the bus on which the input or output is located (only for VS1202)
    - ‘type’: if the input or output is used for ‘audio’ or ‘video’, so that it can be displayed in the right panel (only for VS1202)
    - ‘name.EN’, ‘name.FI’ and ‘name.SE’: The name of the input or output that should be displayed to the user, in each language.
  - `<SwitchedOutlet>`, with the properties:
    - ‘name’: name with which the outlet will be referenced
    - ‘card’: the address of the relay card on which the controlling relay is located
    - ‘port’: the zero-based number of the relay, on the given relay card, that controls the outlet
    - ‘name.EN’, ‘name.FI’ and ‘name.SE’: The name of the outlet that should be displayed to the user, in each language.
- Lights and blinds, the basic functionality of this program, are located between `<AuditoriumDefinition>` tags.
  - Lights are defined in a `<Lights>` tag, with each light control represented by a `<Light>` tag with the properties:
    - ‘type’: currently, only ‘basic’ is supported (on or off type of light control)
    - ‘name’: name with which the light will be referenced
    - ‘name.EN’, ‘name.FI’ and ‘name.SE’: The name of the light that should be displayed to the user, in each language.
  - Blinds are defined in a `<Curtains>` tag, with each blind control represented by a `<Curtain>` tag with the properties:
    - ‘type’: ‘blind’ or ‘projection’
    - ‘name’: name with which the blind will be referenced
    - ‘timeout’: for the blind type, number of milliseconds that the blind needs for a full opening or closing.
    - ‘timeout.up’ and ‘timeout.down’: for the projection type, number of milliseconds the screen needs for, respectively, a full upwards or downwards motion.
    - ‘name.EN’, ‘name.FI’ and ‘name.SE’: The name of the blind that should be displayed to the user, in each language.
    - The location of the relays that control the upwards and downwards motion, are defined with `<Control>` tags within the `<Curtain>` tag, and with the properties:

        - ‘type’: ‘up’ or ‘down’
        - ‘card’: address of the relay card on which the relay is located
        - ‘port’: zero-based number of the relay on the relay card.

- Tiles and shortcuts for the program are defined within a `<Start-Buttons>` tag.
- Each tile on the home screen is a `<Tile>` tag, and can contain zero, one, or multiple commands that will be executed on clicking it. The `<Tile>` tag itself has the following properties:
    - ‘name’: name with which the tile is referenced
    - ‘name.EN’, ‘name.FI’ and ‘name.SE’: The title of the tile that should be displayed to the user, in each language.
    - ‘width’: can be either 1 or 2, and equals the width of the tile in units. 1 is a standard tile, 2 is a double tile.
    - ‘visual’: the name of the primary icon. For a list of names, please consult the file ‘Icons.xaml’ in the project directory.
    - ‘visual1’: the name of the icon that will appear next to the one defined in ‘visual’, in the case of a double tile. Both icons will be separated by a ‘plus’ sign.
- Each shortcut in the shortcut list is a `<Shortcut>` tag, and, like the tiles, can contain an arbitrary number of commands that will be executed when the shortcut is activated. The `<Shortcut>` tag contains the following properties:
    - ‘name’: name with which the shortcut is referenced
    - ‘name.EN’, ‘name.FI’ and ‘name.SE’: The text on the shortcut that should appear to the user, in each language.

The commands that can be attached to the defined tiles and shortcuts, are defined with a `<Command>` tag. Which commands can be assigned, depends on the available equipment, and the defined interactions within the program. Each command has a ‘type’, ‘value’ and ‘target’ property, of which the meaning should be pretty self-explanatory. At the moment of writing, the following command types are available:

- ‘light’: to control a light. Possible values are ‘on’ or ‘off’, and the target is the name of the light that needs to be controlled. Target can also be ‘all’, with which the specified action is applied to all defined lights.
- ‘curtain’: to control a blind. Possible values are ‘up’ and ‘down’, and the target is the name of the blind that needs to be controlled.
- ‘volume’: is used to control the volume of a volume extension card. The value is the value of the volume that is requested, between 0 and 64, and the target is the name of the volume channel, as defined in the Extension section of the relay cards.
- ‘mute’: is used to control the mute function of a volume control. Value is either ‘on’ or ‘off’, and the target is the name of the volume channel.
- ‘VSource’: to select an input on a video switcher. Value is the name of the input that needs to be selected, target is the name of the switcher on which the input is located.

- ‘ASource’: same as ‘VSource’, but for the audio part of the switcher.
- ‘beamer’: is used to control the beamer. If the target is ‘power’, possible values are ‘on’ or ‘off’ to switch the beamer to the respective poser state. Target ‘input’ can be used with the values 1, 2 and 3 to have the beamer select either input 1, input 2, or input 3.

With this information, it should be possible to create arbitrary configurations. When in doubt, consult the sample configuration file supplied on the attached CD.

5.9 Future extensions

The application was designed with expandability in mind. In the list below are the guidelines which should be followed when an extension needs to be created and implemented:

- If there is a need for a new user control, create that user control and all necessary interaction logic, and put it in the ‘Controls’ folder. In particular, take care to add translation functionality to the control. When in doubt, take a look at the provided controls for reference.
- Please also include a Debug event in the user control that raises each time a relevant action is taken, and includes a human-interpretable string as sender, detailing what happened.
- In the file ‘MainWindow.xaml.cs’, all main logic for the application is programmed.
    - The configuration file is parsed in the method ‘ConfigurationRead’. All new objects are created there, their parameters are set at the moment of creation, and they are added to the respective tab panel. The DebugHandler is also added to the Debug event of all user controls. Please see the examples in that method for practical information about how to implement a new control.
    - The commands are parsed with the method ‘ExecuteCommand’. To define a new command, please insert the handling logic there, as well as the lookup of the corresponding object in the tab panels. For examples, please consult the code that has already been written.
    - The language function is the method ‘Change-Language_Click’. For a new user control, please insert a line to change that control’s language property to the ‘currentLanguage’ variable, and take care to update the display language of the control at the moment the language is updated. The language is a two-character string: “EN” for English, “FI” for Finnish, and “SE” for Swedish.

6 INSTALLATION IN AUDITORIUM A

The new control system is not only designed by us, but also installed. Firstly, we started updating auditorium A. This is the oldest auditorium, and is the most in need of an update

6.1 Key panel

The first step we took was to replace the keypad controller with a controller of our own design. This was made possible by the already present three-core cable, of which two cores are used for bidirectional data transfer, and one for the common reference.

At first we tried to connect the keypad controller to the cable at one end, and a relay card at the other end of the cable for testing. The temporary relay card was still powered by a laboratory power supply. For the permanent installation, the data cable, arriving in the switching cabinet, was connected to a shielded twisted pair cable running to the A/V-rack. This twisted pair was the signal cable previously used to control the Helvar dimmers from the transfer module connected to the Crestron system.

6.2 Beamer

The beamer used in this auditorium is suspended above the audience at a height of three meters above a seated person. The ceiling at this point is 4.20 meters above the ground. When looking more closely at beamer, the missing serial cable connection was noticed.

When following the other data cables, such as the VGA and video cables, a multi-core cable in the ceiling was found. On one of its cores was a standard D-SUB9 male connector attached, which is the de facto standard for an RS232 connection. The other end of this cable ran back to the 19” rack where it was patched into the Crestron system.
.
A new pigtail cable had to be made because the serial control port of the beamer in auditorium A uses a mini-DIN 8 male connector. Discovering the serial cable was already available, and running from the rack to the beamer, was a great help due to the height of the ceiling. We were not sure if we would be able to reach the ceiling all the way to the beamer, in order to install a new cable.

After making this pigtail cable from D-SUB 9 to mini-DIN 8, we started testing the beamer commands. The connection between beamer and pc was very poor. The beamer doesn’t always respond in an orderly fashion to the commands given by the controller.

To improve the connection between the pc and the beamer we opted to use another twisted pair inside the multi-core cable. This seemingly improved the communication a bit, but some responses were still dodgy at best. Eventually we just gave up, and used the communications cable as-is.

6.3 Video switches.

The next devices of which the communications were checked, are the VGA-switch at the lecturer's desk, and the 2x12 A/V-switch matrix in the technical room. During the testing of these protocols, the only problem encountered was the wiring of the VGA-switch. The datasheet of the VGA-switch stated a crossed[27] cable should be used. Eventually a straight[28] cable was proven to be needed in order to communicate with this switch.

6.4 Wiring

In a project like this, it is important to keep all wiring structured. This is to keep an overview during the installation. It is also to keep things easy to understand with regards to future service by others. Some wires were kept for their original function; others got reused for other functions.

The backbone of our system are the serial ports on the touchscreen pc. Connecting these ports with terminals in the technical room at the back of the auditorium proved to be a big challenge. At least three separate communication channels had to be available. During the installation, we also had to think aesthetical, so a proper serial connection point was mounted in the lecturer's desk, providing the right type of connectors: D-SUB 9.

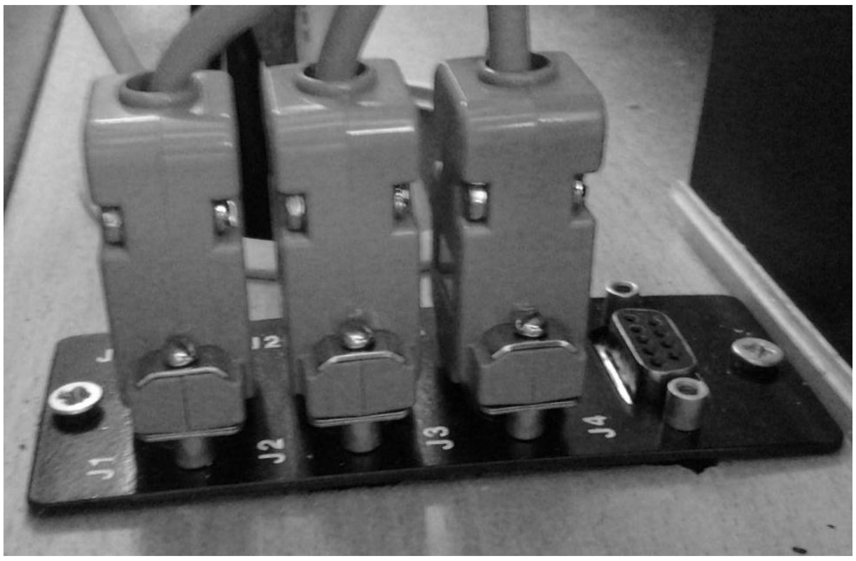


Figure 58 RS232 connectors on the lecturers desk

The cables used for this connection are straight male-female cables. This way they are used as an extension cable to extend the serial port connection of the computer to the outlet now mounted in the desk.

The connection from this point on is made with a shielded Cat 5e cable which was already installed, and was unused. This cable runs all the way to the technical room, has shielded RJ45 female connectors attached at

[27] The RX and TX pins are switched on the other end of the cable
[28] The RX and TX pins of the cable stay on the same place

both ends, and is ideally suited to contain the three communication lines. Three twisted pairs of the cable are used for the data lines, and the fourth twisted pair is used to connect the reference voltage. Each twisted pair represents a serial port of the pc. The full-coloured wire is connected to pin two of the D-SUB 9 connector, and the striped wire is connected to the third pin of the connector. These wires are connected in the 19” rack to screw terminals, to ease the local patching work. The wire pairs are labelled with the numbers of their connection point in the desk, and mounted in the same order. The full-coloured wire is always connected first. If a crossed cable is needed to control the specific device, the crossing of the cable happens after these terminals. This is to avoid double crossing, and as such end up with a straight cable.

These screw terminals also are the patch point to connect our relay cards to the mains switching cabinet.

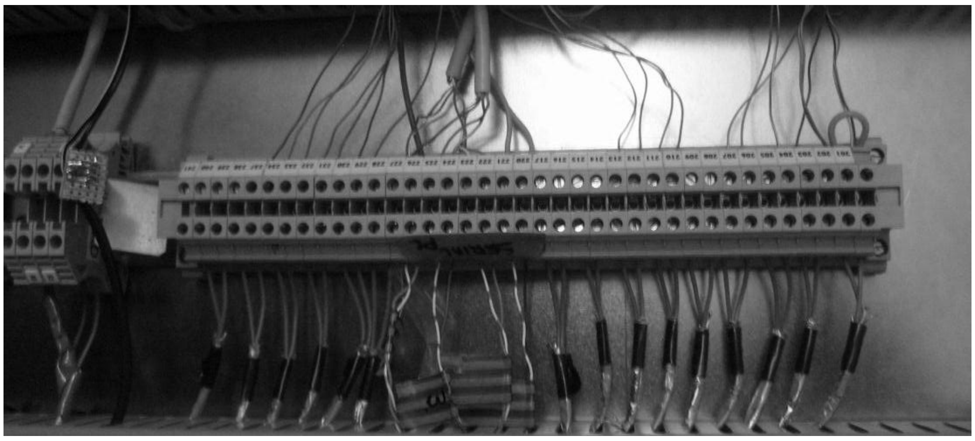


Figure 59 Screw terminals with serial communications and relay cables

The connections on the bottom row of the photograph are the cables which run to and from the 19” rack. The cables on the top row are the patch cables within the rack. On the left side of the figure, an extra screw terminal is visible. This is the connection running to the keypad at the door. Across the keypad terminal, a piece of breadboard PCB is visible. The LM7805 voltage regulator which regulates the voltage to the audio cards is soldered on this piece of circuit board.

On the main terminal block (Figure 60), the connections are, from left to right: six light relays, the A/V-switch’s serial communications line, the relay cards’ serial communications line, the beamer’s communications line, the left blinds’ up and down contacts, the right blinds’ up and down contacts, the projection screen’s up and down contacts, and last but not least the connection to the switchable power outlets’ relay contacts. All the relays have a common connection; this common connection is the rightmost screw terminal on the photograph.

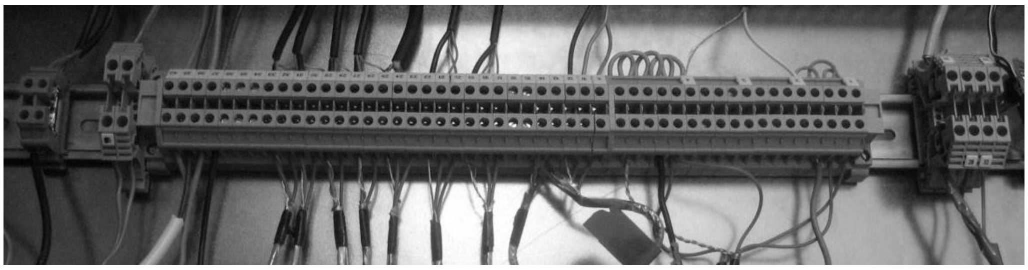


Figure 60 Screw terminals for A/V signals and power

To the left of the A/V-signal connections, two power connections are installed. The leftmost one is the +5VDC output of the LM7805. It is used as a power supply for the volume control cards. Immediately to the right of it is the power supply connection for the relay cards. The power supply for the relay cards isn't as ripple-free as it needs to be for the audio. The reason for this was explained previously in this document.

Next to the power supply connections, is the connection between the amplifier and speakers. To the right of that one are the audio lines from the PC and laptop connection point at the desk. They are both stereo signals, and each channel has its own ground reference. After these four audio-feeds, three mono feeds are connected. Two of these stem from unconnected points under the desk, and are plugged straight into the mixer. The third microphone is situated at the lecturer's desk, and connected to a volume control unit before it is plugged into the mixer.

To the right of the mic feeds, the beamer's serial communication signal is connected, leaving the 19" rack and going to the beamer. The other end is internally patched through to the connection point going to the PC, as mentioned earlier.

Next to the serial communication lines going to the beamer, several screw terminals are present in order to distribute the +24V supply rail which drives the relays on both the relay cards and in the fuse cabinet.
This rail has so many connections, because CresNet was previously also connected on these terminals.

6.5 Installation of the relay cards

The back of the 19" rack is made out of a hardened metal. To mount something on this plate, you would need to drill a hole and tap a screw thread for screws or bolts.

The Crestron system previously had its own external relay cards which were mounted on the plate, also having the size of a eurocard. The solution to mount our relay cards was to reuse the existing holes to mount all relay cards.

A black polycarbonate plate was used to premount all the relay cards. They were mounted on the plate using nylon screws and spacers. Once this procedure was finished, the complete setup was mounted to the back of the rack using the fixing points already available from the Crestron expansion cards.

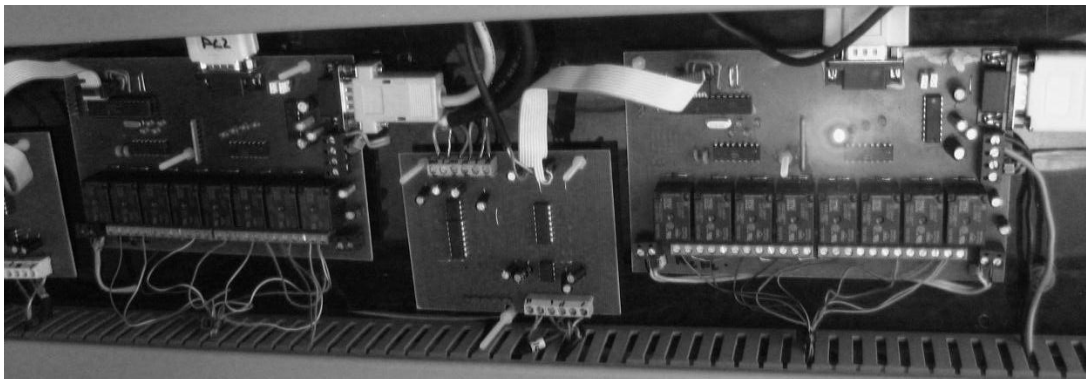


Figure 61 Our relay cards mounted in the 19” rack

Both relay cards have the expansion card for volume control and extra digital I/O ports. However, in this situation the I/O port expander is not in use.

The power link cables of the +5V and +24V supply rail are pre-mounted on the backside of the mounting panel. The power supply connections themselves, however, do not run underneath the panel. They run through the cable guides to their respective screw terminals.

6.6 Touchscreen-PC enclosure

When all wiring was done, and all extra hardware was successfully installed, the housing unit for the touchscreen-PC arrived. The design was done by our friend and colleague Riemert Viaene, and it was built by the vocational school in Valkeakoski. When test-fitting the touchscreen for a trial run, the need to have ventilation holes surfaced.

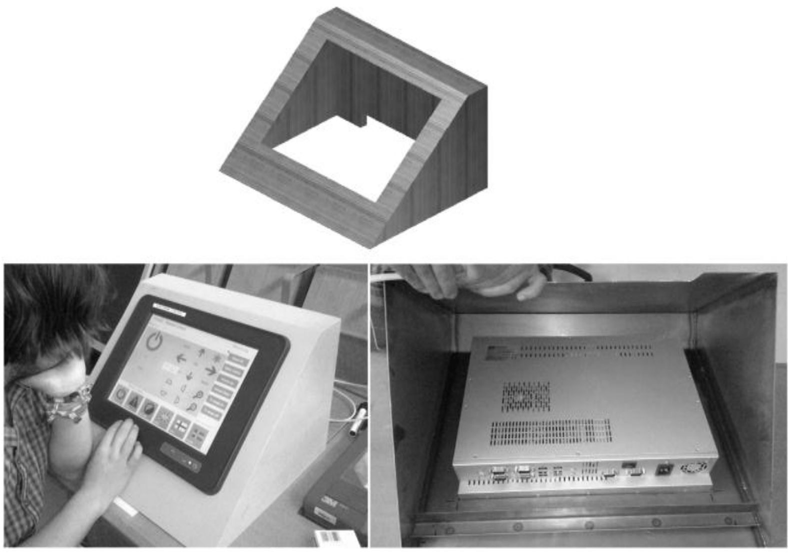


Figure 62 Testing the touchscreen computer in its custom housing

These ventilation holes were drilled in the sides of the housing, in a hexagonal pattern. After the addition of the ventilation holes, the housing was spray painted with a matte black paint.

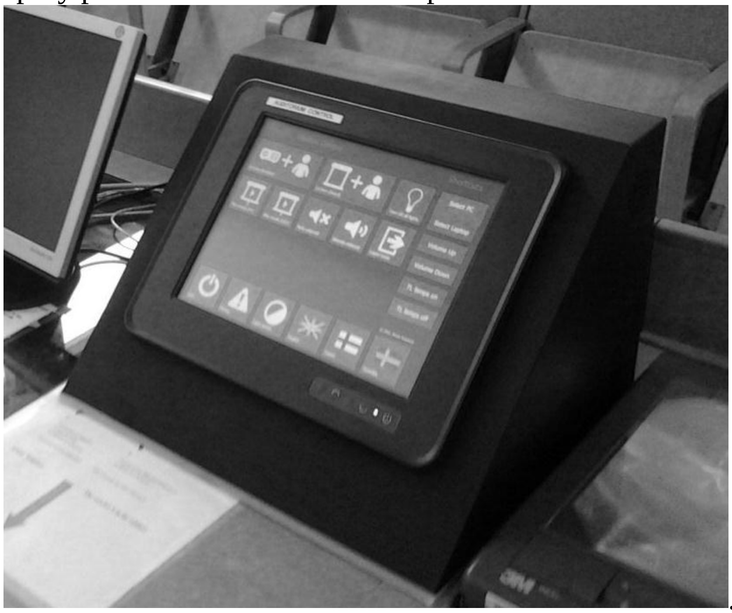

.

Figure 63 Completed touchscreen enclosure

6.7 Block diagram

After all modifications were made, a block diagram of the current situation with regards to the control system was created.

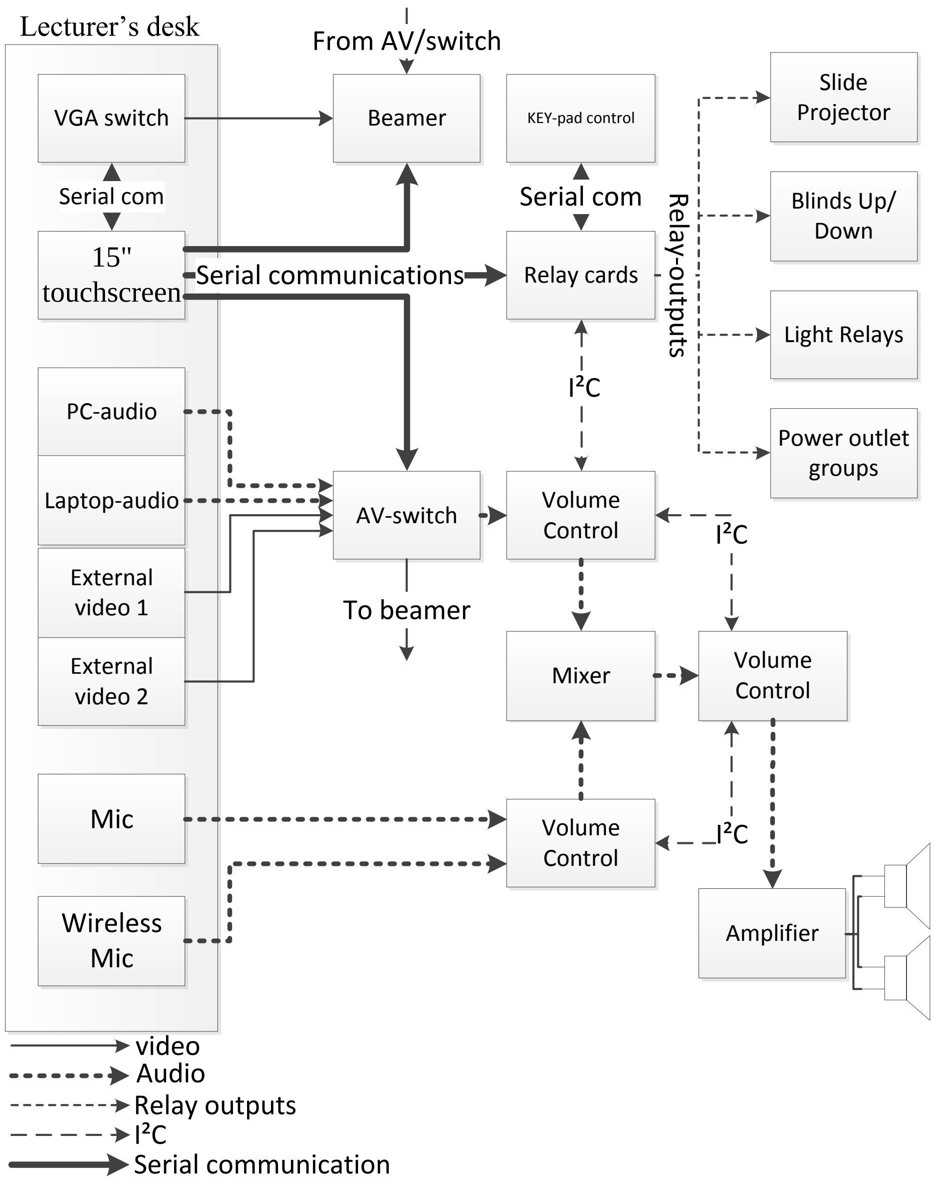


Figure 64 Auditorium A: new situation (block diagram)

If this diagram is compared to the previous state of the auditorium, the main additions are the multiple volume control units, the bus controlled relay cards, and a working beamer interface. The Crestron control unit is now taken out of service, and the whole auditorium is running on our own custom developed automation system.

7 CONCLUSIONS

The installation of the new control unit for the auditorium resulted in a user-friendly control interface for this multi-purpose room. The touchscreen displays big multi-lingual and colourful buttons to create an intuitive control environment.

The control software has both an easy and intuitive preset window for the occasional user, and a manual control window where every function of the system can be adjusted to satisfy the more demanding users.

The complete control unit is mounted at a 45° angle in a clearly visible black housing located on the lecturer’s desk. Part of the design was to draw the attention of someone who wants to control the system for the first time.

Four serial communication channels are leaving the control unit. One remains at the lecturer’s desk in order to control the VGA switch. The other three are connected to numbered connection ports mounted in the desk. The communication channels run up to the technical room in order to control the beamer, the 2x12 A/V-switch and the custom-designed relay and expansion cards.

The custom relay cards are located in the 19” rack, and mounted on its back panel. Both cards are equipped with an extension card to control the volume of the microphones, the external audio and the main volume.

Both the touchscreen control unit and the keypad next to the entrance door control these relay cards. If one of them changes the state of a light, it always updates the other one to keep synchronisation.

By replacing the control system with our own, both functionality and user-friendliness have been improved. Adaptations to the system are now more easily implemented. And, last but not least, all technical files and details are available and up-to-date to enable any interested party to make changes to the system.

SOURCES

BSS. (n.d.). *BSS Audio sw9088iis*. Retrieved February- May 2012, from BSSaudio: http://www.bssaudio.com/discont_productpg.php?product_id=31

CadSoft. (n.d.). *Downloads*. Retrieved February 2012, from CadSoftUSA: http://www.cadsoftusa.com/download-eagle/?language=en

Crestron programmers group. (n.d.). Retrieved February 2012, from tech.groups.yahoo: http://tech.groups.yahoo.com/group/Crestron/

Helvar. (n.d.). *discontinued products*. Retrieved April 2012, from helvar: http://www.helvar.com/default.asp?path=3386,3660&article=5331&lan=EN&search=true&index=X&page=1

Howie, J. (2008, December 03). *NEC Infrared Transmission*. Retrieved April 2012, from altium: http://wiki.altium.com/display/ADOH/NEC+Infrared+Transmission+Protocol

Maxim. (2002, June 27). *Using a DS1802 Push-Button Digital Potentiometer*. Retrieved April 2012, from maxim-ic: http://www.maxim-ic.com/app-notes/index.mvp/id/0161

microchip. (n.d.). *PIC18F24K22*. Retrieved February 2012, from Microchip Technology Inc: http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en547749

Microsoft. (n.d.). *.NET Common Language Overview*. Haettu March through May 2012 osoitteesta Microsoft Developers Network: http://msdn.microsoft.com/en-us/library/ddk909ch.aspx

MikroElektronika. (n.d.). *mikroC pro for pic Language reference*. Retrieved February 2012, from mikroelektronika: http://www.mikroe.com/download/eng/documents/compilers/mikroc/pro/pic/help/mikroC_PRO_language_reference.htm

Moser, C. (n.d.). *A collection of WPF tutorials*. Retrieved March through April 2012 from WPF tutorials: http://www.wpftutorials.net

SB projects. (2011, May 29). *IR Remote Control, JVC Protocol*. Retrieved April 2012, from SBprojects: http://www.sbprojects.com/knowledge/ir/jvc.php

SB-projects. (2011, May 29). *IR Remote Control, NEC Protocol*. Retrieved April 2012, from sbprojects: http://www.sbprojects.com/knowledge/ir/nec.php

Stephens, R. (2010). *WPF Programmer's Reference: Windows Presentation Foundation with C# 2010 and .NET 4*. USA: Wrox.

Unknown. (n.d.). *MahApps.Metro: a metro-styled WPF theme*. Retrieved March 2012 from MahApps: http://mahapps.com/MahApps.Metro/

WinLIRC. (n.d.). *index of remotes*. Retrieved April 2012, from winlirc: http://lirc.sourceforge.net/remotes/

APPENDIX 1: Original work plan

Goal	description	deadline
Analysis of the current system	Analysing and checking the plans and wire-schematics of the existing system	25.02.2012
Implementation of beamer protocols	Implementation of the communication protocol used by the beamers.	28.02.2012
Design relay-card	Designing the schematics and PCB-layout used to control the relay-switches currently used. Component selection.	27.02.2012
Design firmware relay-card	Programming a custom designed protocol to control the relay-cards.	10.03.2012
Implement relay-card	Etching and soldering the pcbs.	19.03.2012
Test relay-cards	Controlling a chain of 4 relay-cards by means of the designed protocol	19.03.2012
Design expansion-card	Designing the schematics an pcb-layout for the expansion-card used as volume-control and port-expansion. Component selection.	31.03.2012
Implement expansion-card	Etching and soldering the pcbs.	14.04.2012
Test expansion-card	Adjusting volume and using extra i/o-ports by means of the touchscreen computer	14.04.2012
Design GUI	Designing a user-friendly interface to control the whole system.	20.04.2012
Design IR-card	Designing the schematics and pcb-layout for de IR-card. This card will be used to control devices that use IR. Component selection.	28.04.2012
Light control	Analyse the existing HELVAR-system and control it by the port expanders on the expansion-card.	1.05.2012

Replace the system in A-auditorium	Installing a complete system in the A-auditorium and testing	5.05.2012
Implement IR-card	Etching and soldering the pcbs.	12.05.2012
Designing firmware IR-card	Designing protocol and firmware of the IR-card, so it can control a specific device.	20.05.2012
Test IR-card	Control all the devices by use of the IR-card	20.05.2012
Reverse engineer dsp-protocol	Research for the control-protocol to control the digital signal processor used in the B-auditorium	20.05.2012
Replace the system in B-auditorium	Installing a complete system in the B-auditorium and testing	1.06.2012
Thesis document	The thesis document has to be finished.	1.06.2012
Thesis presentation	Defending the thesis	

APPENDIX 2: Reversing the SoundWeb protocol

The DSP used in auditorium B is a BSS SoundWeb-series processor. This is a networked DSP and can be linked to other units. It is also controllable trough a serial communications link.

The SoundWeb is a programmable DSP which can be programmed to the specific needs of the end user. The ability to customise the system makes it very hard to determine the commands needed to control an existing configuration.

To be able to control and reuse the DSP we needed to ‘sniff’ the serial communications. BSS provided us with an application to transmit commands to the DSP, and a PDF file containing the protocol definitions. By sniffing the commands sent by the Crestron system, and matching the data to the BSS specification, an insight in the protocol was achieved.

The SoundWeb protocol has various similarities with our own protocol conceived to control the relay cards.

<message> = <STX> <Body> <Checksum byte> <ETX>

Structure of a SoundWeb message

A command consists out of 4 parts. A start byte starts the message. The body contains the actual command. The checksum is used to verify the command, and lastly a stop byte is used to mark the end of the message. When a command is received, the receiving device should answer with an ACK[29] response, within one second after receiving the command. When nothing or a NAK[30] is received by the transmitter, the command is reissued.

The protocol has some reserved values. These are the start and stop byte, the ACK and NAK bytes, and the escape character.

–	0x02	STX
–	0x03	ETX
–	0x06	ACK
–	0x15	NAK
–	0x1B	escape

When one of these values needs to be used in the body of the message, they are replaced by the following values:

- 0x02 → 0x1B 0x82
- 0x03 → 0x1B 0x83
- 0x06 → 0x1B 0x86

[29] Acknowledge
[30] Not Acknowledged

- 0x15 → 0x1B 0x95
- 0x1B → 0x1B 0x9B

The body starts with a byte which indicates what will be done. This means to set or request a value of the DSP. These bytes are respectively 0x80 and 0x82. This control byte defines the meaning of the other bytes in the message's body.

<**SET_VALUE**> |<***group***> <***id***> <***value Hi***> <***value Lo***>

SET_VALUE structure of the BSS protocol.

When SET_VALUE is selected, at least[31] four extra bytes are needed.
The first byte defines which value will be changed. For example, when the audio will be adjusted, this byte will be 0x05.

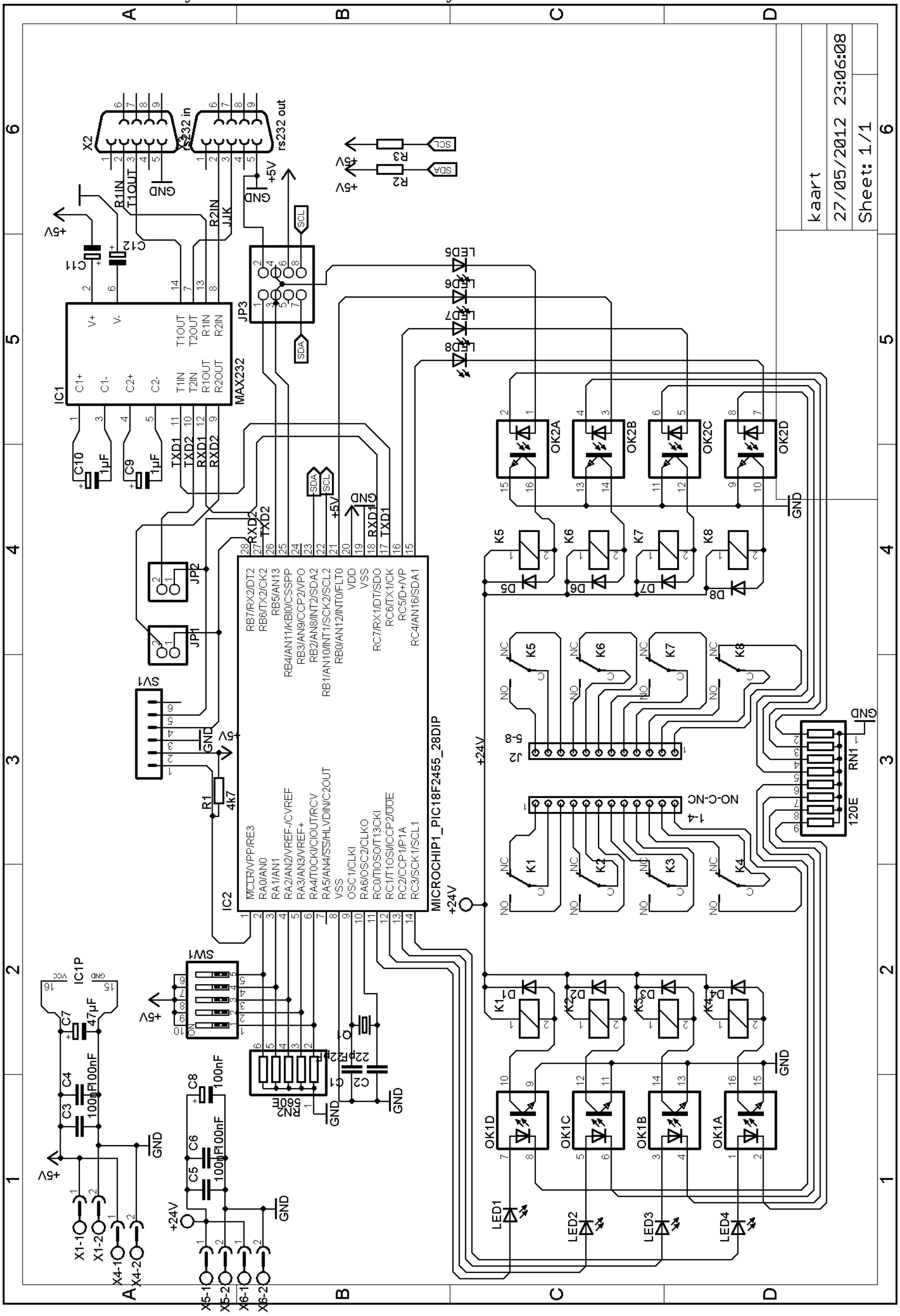
The second byte is the identifier byte. This byte specifies which control of the group, defined by the previous byte, will be adjusted. For example, in auditorium B, this ID will be 0x1B83. As mentioned before, this is because the actual value 0x03 is a reserved character.

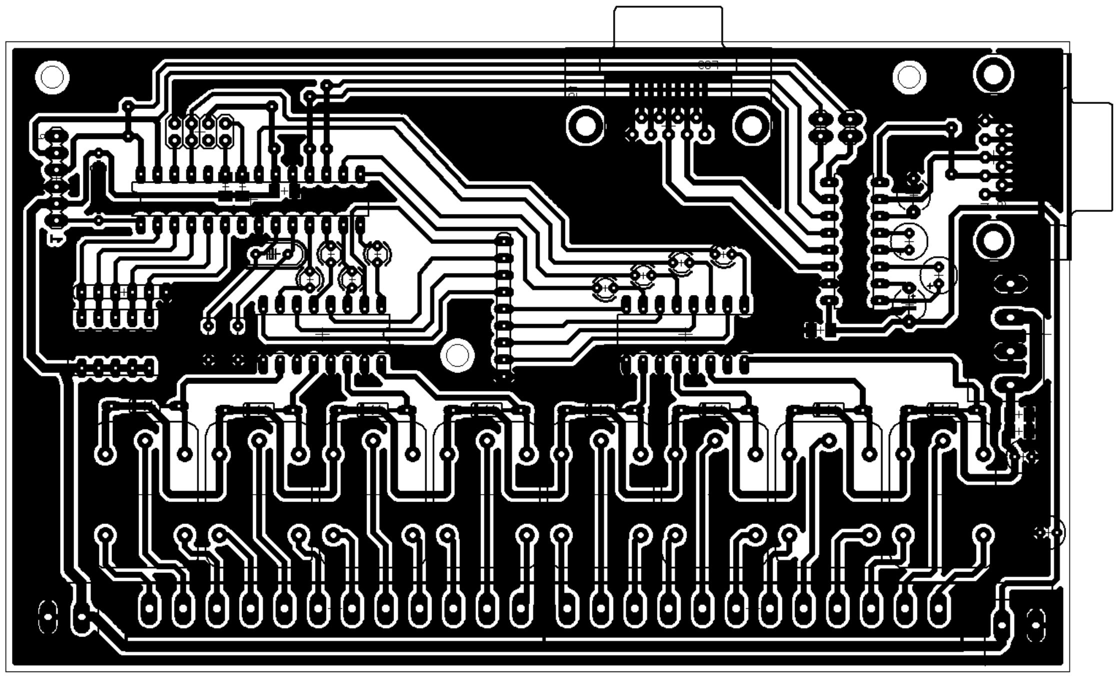
The two following bytes, which are the last bytes of the body, are the 16-bit value that can be given to the DSP parameter. When the level is changed like in the example given here, the high byte remains 0x00. This means the audio level can be adjusted on an 8bit scale, so values can change between 0x00 and 0xFF, 0x00 being mute.

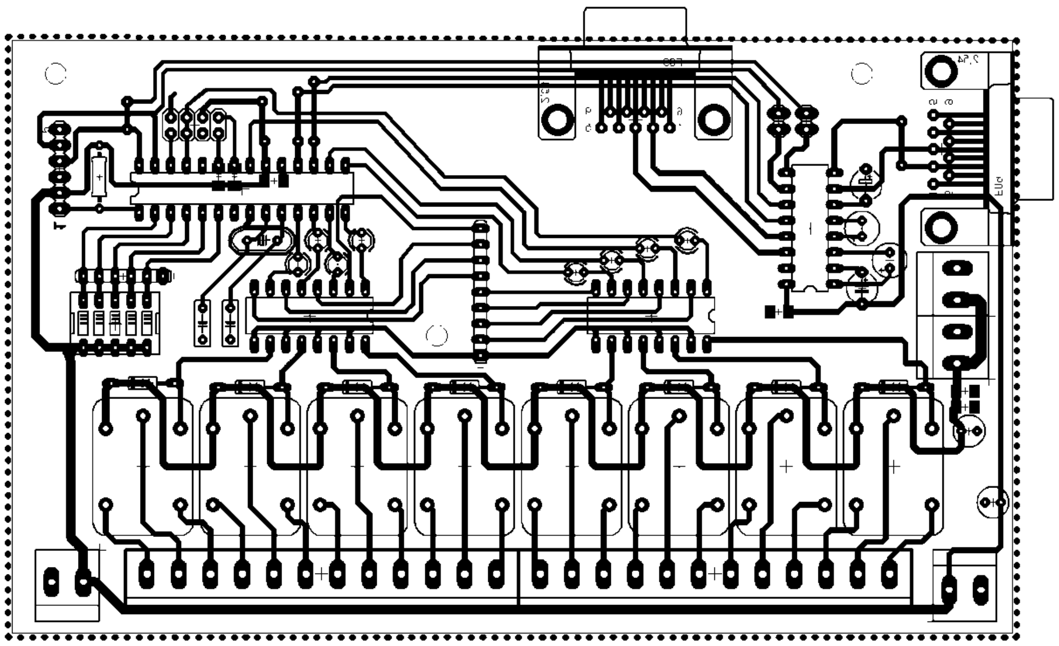
After the body, a checksum is added. This checksum is used to verify whether the data has been received correctly or not. The checksum is an XOR function of all the bytes from the body. In case a reserved byte is used in the body, the original value will be used to calculate the checksum and then is changed to a special byte. It is important to note that when the checksum is one of the reserved values, it is replaced by the alternative, just as if it were a body byte.

[31] When a special value is used as data byte, it has to be replaced by two bytes

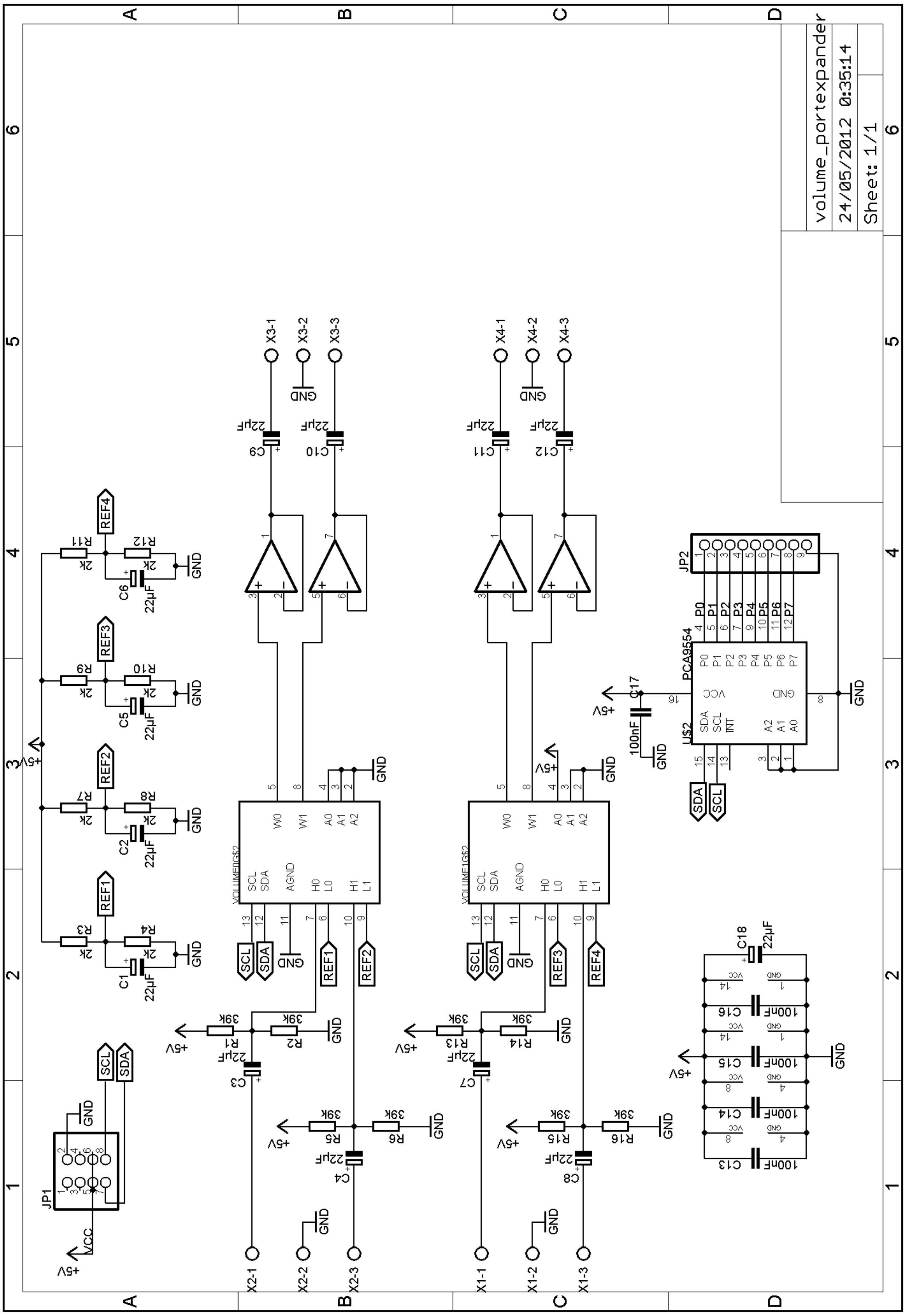
APPENDIX 3: Relay card schematic and PCB layout

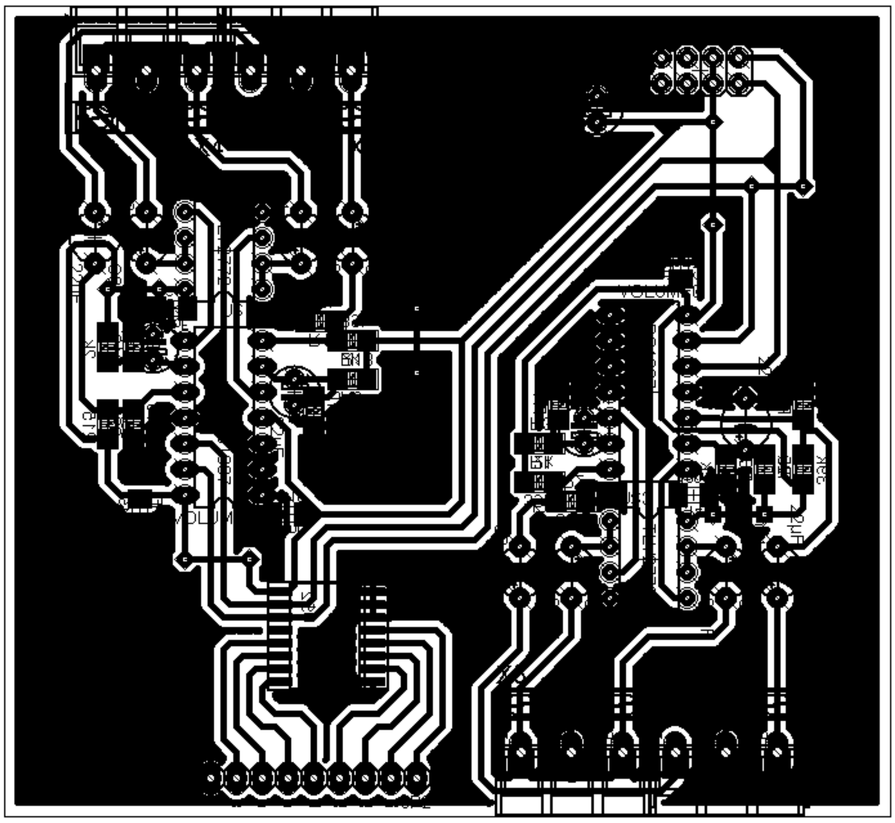


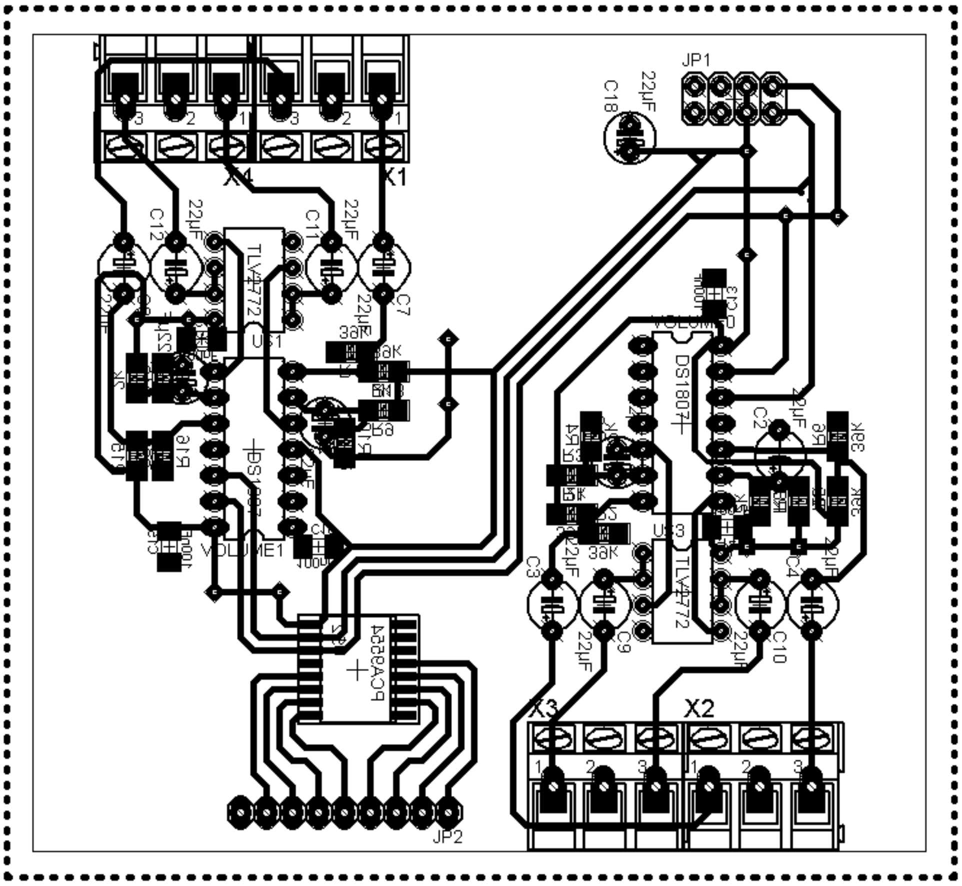




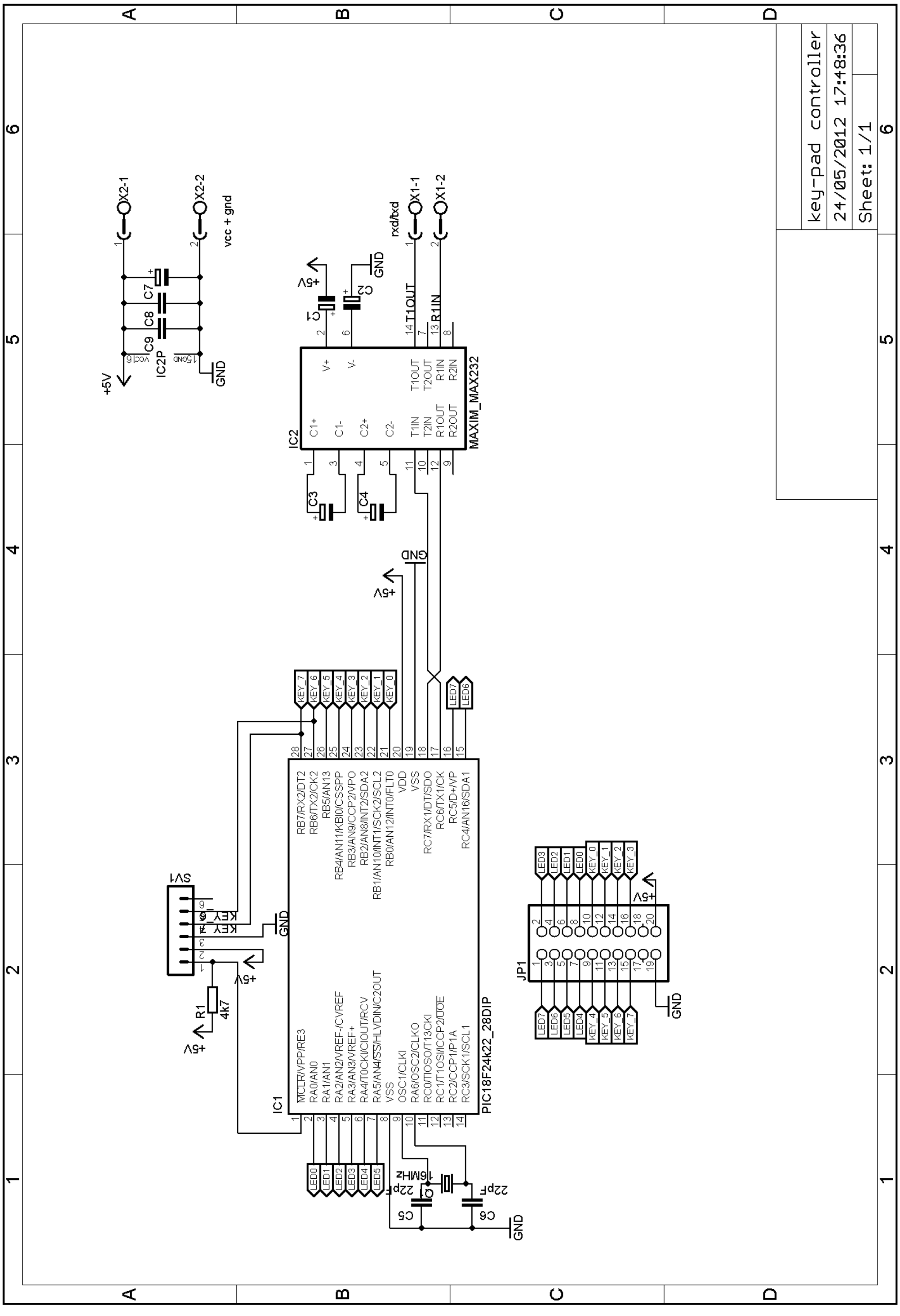
APPENDIX 4: volume control and I/O port extension schematic and PCB layout






JP1
C18
22µF
X4
X1
C12
C11
22µF
TLV4772
C7
IC1
DS1807
VOLUME1
PCA9564
C3
C4
C9
C10
US3
X3
X2
JP2
38K
36K

APPENDIX 5: Keypad schematic and circuit implementation


key-pad controller
24/05/2012 17:48:36
Sheet: 1/1
IC1
PIC18F24K22_28DIP
IC2
MAXIM_MAX232
SV1
JP1
R1
4k7
Q1
16MHz
C5
22pF
C6
22pF
C1
C2
C3
C4
C7
C8
C9
IC2P
X1-1
X1-2
X2-1
X2-2
rxd/txd
vcc + gnd
T1OUT
R1IN
+5V
GND
MCLR/VPP/RE3
RA0/AN0
RA1/AN1
RA2/AN2/VREF-/CVREF
RA3/AN3/VREF+
RA4/T0CKI/C1OUT/RCV
RA5/AN4/SS/HLVDIN/C2OUT
VSS
OSC1/CLKI
RA6/OSC2/CLKO
RC0/T1OSO/T13CKI
RC1/T1OSI/CCP2/UOE
RC2/CCP1/P1A
RC3/SCK1/SCL1
RB7/RX2/DT2
RB6/TX2/CK2
RB5/AN13
RB4/AN11/KBI0/CSSPP
RB3/AN9/CCP2/VPO
RB2/AN8/INT2/SDA2
RB1/AN10/INT1/SCK2/SCL2
RB0/AN12/INT0/FLT0
VDD
VSS
RC7/RX1/DT/SDO
RC6/TX1/CK
RC5/D+/VP
RC4/AN16/SDA1
C1+
C1-
C2+
C2-
T1IN
T2IN
R1OUT
R2OUT
V+
V-
T1OUT
T2OUT
R1IN
R2IN
KEY_0
KEY_1
KEY_2
KEY_3
KEY_4
KEY_5
KEY_6
KEY_7
LED0
LED1
LED2
LED3
LED4
LED5
LED6
LED7