Technische Universität Berlin
Fakultät I
Institut für Sprache und Kommunikation
Fachgebiet Audiokommunikation und -technologie

Development of a Digital Musical Instrument with Embedded Sound Synthesis

Masterarbeit

für die Prüfung zum Master of Science im Studiengang
Audiokommunikation und -technologie

vorgelegt von

Pascal Staudt
[redacted]

Erstgutachter:	Prof. Dr. Stefan Weinzierl
Zweitgutachter:	Dominik Hildebrand Marques Lopes (Universität der Künste Berlin)
eingereicht am:	11.10.2016

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt gegenüber der Fakultät I der Technischen Universität Berlin, dass die vorliegende, dieser Erklärung angefügte Arbeit selbstständig und nur unter Zuhilfenahme der im Literaturverzeichnis genannten Quellen und Hilfsmittel angefertigt wurde. Alle Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind kenntlich gemacht. Ich reiche die Arbeit erstmals als Prüfungsleistung ein. Ich versichere, dass diese Arbeit oder wesentliche Teile dieser Arbeit nicht bereits dem Leistungserwerb in einer anderen Lehrveranstaltung zugrunde lagen.

Mit meiner Unterschrift bestätige ich, dass ich über fachübliche Zitierregeln unterrichtet worden bin und diese verstanden habe. Die im betroffenen Fachgebiet üblichen Zitiervorschriften sind eingehalten worden. Eine Überprüfung der Arbeit auf Plagiate mithilfe elektronischer Hilfsmittel darf vorgenommen werden.

Berlin, den	Unterschrift

ABSTRACT

Performances with new *digital musical instruments* (DMIs) are commonplace nowadays. In most cases these instruments are gestural controllers connected to a laptop that runs the sound synthesis. However, a general purpose computer is not primarily intended to be a musical instrument and when it comes to performing on stage, it can cause issues both technically and artistically. Furthermore the short lifespan of new DMIs is often linked to the use of personal computers, which threaten the functionality of the instrument. This work examines the development of a DMI with embedded sound synthesis – the *PushPull embedded* ($\mathrm{PP}_{\mathrm{EMB}}$) – an instrument that overcomes these issues by embedding the computing hardware. Therefore, requirements concerning the hard- and software are set, before different microcontroller and microprocessor devices are considered and compared carefully. The iterative and incremental development of a prototype is laid down and three instrument patches are presented to validate the set requirements and demonstrate the capabilities of the $\mathrm{PP}_{\mathrm{EMB}}$ in terms of parameter mapping and sound synthesis. The performance evaluation shows that a reliable DMI has been developed, with enough processing resources for the realisation of complex sound synthesis. With a sample rate of $48kHz$ and a buffer size of 16 samples, up to 80 oscillators can be used without dropouts or clicks in the sound output and changing between patches stored on the instrument happens within one second. Taken as a whole, the thesis shows that recent microcontroller technology has potential to replace general-purpose computers in DMI designs and consequently enhance the portability and longevity of these instruments.

ZUSAMMENFASSUNG

Performances mit digitalen Musikinstrumenten (DMI) sind heutzutage keine Ausnahme mehr. Die Instrumente, welche hierbei verwendet werden, sind zumeist eine Kombination aus Controller und Laptop. Während der Controller für die Erfassung von Gesten zuständig ist, dient der Laptop zur Klangsynthese. Die Nutzung eines gewöhnlichen Computers als Bestandteil eines Musikinstruments, der nicht speziell als solches konzipiert ist, weist jedoch Probleme auf. Die Nachteile betreffen nicht nur die Beständigkeit des Instruments sondern auch die technische und künstlerische Aufführungsaspekte. Gegenstand der Arbeit ist die Entwicklung eines DMIs mit eingebetteter Klangsynthese – das *PushPull embedded* – ein Instrument, das die eben genannten Defizite überwindet, indem ein Computer zur Klangerzeugung in den Instrumentencorpus integriert wird. Hierfür werden die Anforderungen an Hard- und Software bestimmt, verschiedene Mikrokontroller und Mikroprozessoren in Betracht gezogen sowie deren Vor- und Nachteile verglichen. Nach begründeter Wahl eines geeigneten Geräts wird die iterative und inkrementelle Entwicklung des Gesamtsystems aus Hard- und Software dargelegt. Drei Patches werden vorgestellt, welche verschiedenste Ausführungen des entwickelten Instrumentenprototyps in Hinblick auf Parameter Mapping und Klangsynthese demonstrieren. Die Beispiele dienen außerdem dazu, die gesetzten Anforderungen und Ziele zu überprüfen. Eine Evaluation der Leistungsfähigkeit des Gesamtsystems zeigt, dass ein zuverlässiges und zudem leicht zu programmierendes DMI entwickelt wurde, welches genug Rechenkapazitäten für komplexe Klangsynthese bietet. Bei einer Puffergröße von 16 Samples und einer Samplingrate von $48kHz$ können bis zu 80 Oszillatoren verwendet werden, ohne dass wahrnehmbare Unterbrechungen in der Klangsynthese auftreten. Das Wechseln zwischen auf dem Instrument gespeicherten Patches geschieht innerhalb einer Sekunde. Insgesamt zeigen die Ergebnisse der Arbeit, dass moderne Mikrocontroller das Potential haben, gewöhnliche Computer in DMI Designs zu ersetzten, wodurch die Autonomie und somit die Portabilität und Beständigkeit der Instrumente verbessert wird.

CONTENTS

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

The invention and rapid evolution of computers gave rise to the *Digital Revolution* in the last century. The impact of emerging digital technologies on our life is steadily growing since then. Computers changed not only our every day lives in many regards but also our creative processes.

The availability of powerful and inexpensive computing hardware, which expands the possibilities of software based sound synthesis, is one of the reasons why performances with new *digital musical instruments* (DMIs) are commonplace nowadays (Cook, 2001). Visual audio programming environments, such as *Max/MSP*[1] and *Pure Data*[2], furthermore contribute to the fact that more and more musicians and researchers interested in musical innovation implement their own DMIs for the use in their musical compositions and performance practices (Miranda and Wanderley, 2006). Numerous new DMIs have been developed in the last decade, however, none of them have managed to become widespread in the performance scene and "The need to work out why previous attempts have not had commercial success is critical to the long-term viability of this field of enquiry" (Paine, 2013, p. 84). Indeed, new DMIs are brought to the market by the major audio hardware sellers, but still it seems that they stick to the traditional paradigm of keyboard controlled synthesis. Numerous innovative DMIs are presented at the *International Conference on New*

[1] `https://cycling74.com/products/max/`
[2] `https://www.puredata.info`

Interfaces for Musical Expression[3] every year. Even though many of these interfaces use digital *sound synthesis*, most of the effort goes into the development of new gestural controllers. These interfaces then become part of a DMI, which consists of a gestural controller and a synthesis unit – usually a laptop or desktop computer. Undoubtedly, a lot of musicians exploit the advantages of software synthesizers, but a desktop computer or laptop is not primarily intended to be a musical instrument (Lyons and Fels, 2012) and Paine (2013, p. 79) goes as far as saying "the general public will always be uncertain about the authenticity of a computer music performance." Anyhow, when it comes to performing on stage, the use of a general purpose computer as a musical instrument can cause issues both technically and artistically. According to Berdahl and Ju (2011), the short lifespan of new DMIs is often linked to the use of personal computers, which threaten the functionality of the instrument, for example when a change to the operating system is applied. The result is a lack of dissemination and a limited quality of the musical output, if musicians cannot become experts on their instruments in the short life span of the instrument (Berdahl and Ju, 2011). One method to overcome these issues is to embed the sound synthesis unit, i.e. the computing hardware, into the instrument itself. Such a *self-contained* instrument does not rely on an external computer and offers several advantages concerning reliability and longevity (Berdahl, 2014). The aim of this thesis is to explore the development of a self-contained DMI: the *PushPull embedded* ($\mathrm{PP_{EMB}}$).

The *PushPull* (PP) is a DMI that has been designed and developed within the 3DMIN project[4]. In autumn 2015 the *PushPull student edition* ($\mathrm{PP_{SE}}$) was built according to the development stage at the time. Various sound synthesis implementations have been programmed to examine the possibilities and constraints of the instrument. The goal and subject of this project has been set to overcome the main drawback on the initial design – the need of an external computer – by creating a self-contained prototype of the instrument.

Thereby, the following questions are examined:

[3]`http://www.nime.org/`

[4]Design, Development and Dissemination of New Musical Instruments: `http://www.3dmin.org/`

- What are the technical requirements for the development of a DMI with embedded sound synthesis?
- To what extent is recent *microcontroller/microprocessor* technology suitable for this purpose?
- What possibilities and problems occurred during the development process?

The thesis is structured as follows: the next chapter will give an introduction into DMI design basics and provide the theoretical framework for the thesis. An overview of related work is presented in chapter 3. Chapter 4 will provide insight into the PP_{SE}, as it was the starting point of my work and determined the requirements, which will be outlined in chapter 5 together with the taken approach of an *iterative and incremental development*. Chapters 6 and 7 lay down the implementation of the hardware and the software. In chapter 8, the technical requirements will be validated and the results of my work, as well as the development process itself, will be evaluated. Finally, a conclusion and an outlook is given.

CHAPTER 2 DIGITAL MUSICAL INSTRUMENT DESIGN

From the initial idea to a ready-to-play DMI a lot of different design decisions have to be made. The choice of the sensors, the computing hardware and the synthesis software affect all stages of the design process right up to performance aspects. In this chapter, the basics of DMI design are explained. Therefore, two different models are presented and the term *self-contained instrument* is introduced. Furthermore a short overview of different sensor and feedback technologies is given and the principles of parameter mapping and sound synthesis are laid down. To start with, a definition of the term "digital musical instrument" is given.

Even though there is no clear definition of the term "digital musical instrument", the use of digital technologies is implied. While acoustic instruments produce sounds that can be sensed by the humans auditory system through the resonating of physical entities that cause the surrounding air to vibrate, DMIs use digital technologies to generate electronic signals that can be only heard when amplified and converted into sound waves by an electroacoustic transducer such as a loudspeaker. Miranda and Wanderley (2006, p. 1) state:

> An instrument that uses computer-generated sound can be called a digital musical instrument (DMI) and consists of a control surface or gestural controller, which drives the musical parameters of a sound synthesizer in real time.

This sentence boils down the substance of what is meant when speaking of a DMI in the context of this work.

2.1. DMI Models

In order to conceptualise and visualise the general principle of a DMI, many different models have been published in the last decades (e.g. Lee and Wessel, 1992; Bongers, 2000; Wessel and Wright, 2001; Miranda and Wanderley, 2006; Paine, 2013, etc.). In the following, two existing models are presented: the frequently cited and rather basic DMI model by Miranda and Wanderley and a more detailed model by Marshall (2008).

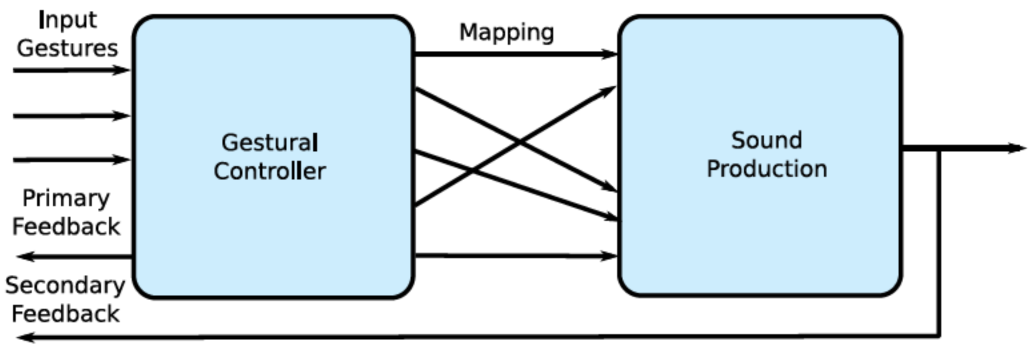


Figure 2.1.: DMI model after Miranda and Wanderley (2006).

In keeping with their definition, Miranda and Wanderley divide a DMI into two logical units: a control surface unit – to which they also refer to as gestural or performance controller – and a sound production unit (see Figure 2.1). While the gestural controller provides the input to the instrument and the *primary feedback* to the performer, the sound production unit is the part of the instrument which includes the sound synthesis and produces the *secondary feedback* – the sonic output. The mapping layer represents the linkage between these two units. Mapping gestural controls to sound synthesis parameters is an essential part in DMI design and will be discussed in section 2.5.

A more detailed DMI model is presented by Marshall (2008) and incorporates elements from the models of Bongers (2000), Miranda and Wanderley (2006) and Birnbaum (2007). Here, the gestural controller or *physical interface* is subdivided

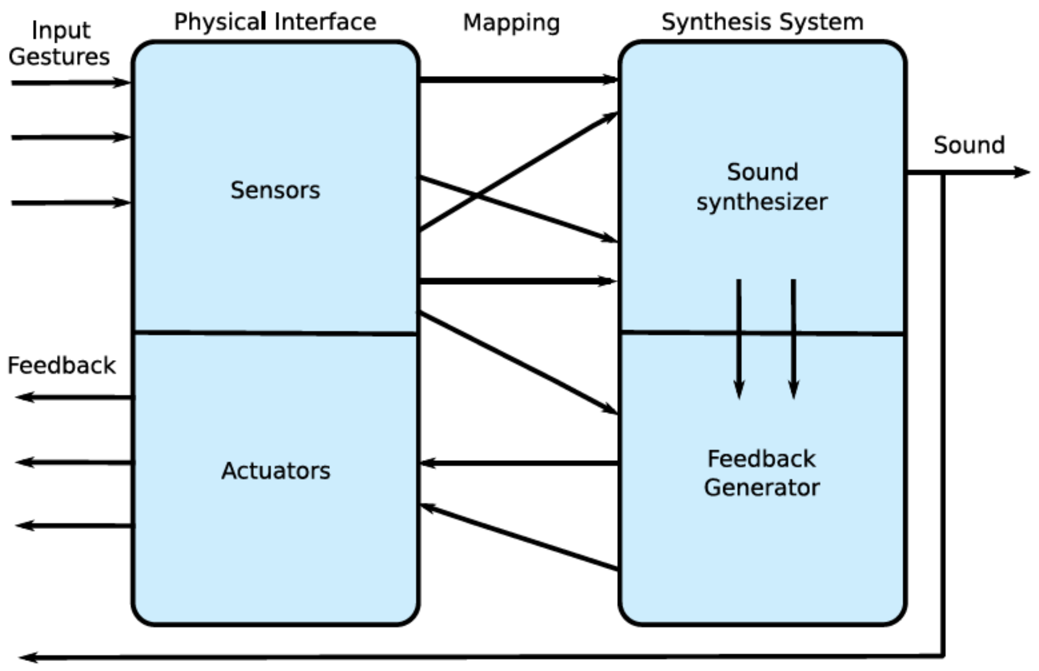


Figure 2.2.: DMI model after Marshall (2008).

into *sensors* and *actuators*. Furthermore, the sound generation unit is included in a *synthesis system* that contains a *sound synthesizer* and a *feedback generator*. This takes into account instruments which include actuators that provide additional feedback. The feedback may be haptic, e.g. vibrotactile feedback generated by a motor or loudspeaker, visual, e.g. light sources, or a combination of both. The bidirectional character of the mapping layer reflects the fact that these actuators can be controlled by the synthesis system. An example for such a connection are *light-emitting diodes* (LEDs) whose brightness is mapped to the amplitude envelope of a synthesis signal.

2.2. Self-contained Instruments

Marshall (2008, p. 53) presents a survey on types of DMIs presented at the *International Conference on New Interfaces for Musical Expression* (NIME) from 2001 to 2008. The following categories are used for the classification of DMIs:

- *Instrument-like controller*
- *Instrument-inspired controller*

- *Extended instrument*
- *Alternate controller*

The fact that 229 of the 266 instruments that were presented are classified as controllers underline that most of the effort went into the development of new gestural controllers. Although no specification is made whether the controllers use embedded sound synthesis, the literature research of NIME papers revealed only few related attempts (see chapter 3).

To distinguish DMIs with embedded sound synthesis from gestural controllers that make use of an external sound synthesis unit, terms such as *self-contained musical instrument* (Smith and Michon, 2015) or *embedded musical instrument* (Berdahl, 2015) are used. Throughout this work the term self-contained instrument will be used for DMIs that combine the gestural controller and the sound synthesis unit in a single instrument body.

Self-contained DMIs offer several advantages concerning portability, autonomy and longevity (Berdahl and Ju, 2011). According to Berdahl and Ju, the use of personal computers as the critical engine of a DMI is the weak link in terms of longevity and robustness of a DMI. General purpose computers that are also used to write emails and theses have the potential to be rendered unusable with every necessary system upgrade or major software update. On the contrary embedded computers are solely used for their intended purpose (Berdahl and Ju, 2011).

The better portability of self-contained DMIs is directly related to their autonomy from an external synthesizer unit and is self-explanatory. The advantages of portability and autonomy manifest themselves most notably in jam and performance situations. Who wants to bring his or her laptop (and an audio interface) to the next jam session and hassle with the set up of the computer, rather than just bringing a compact instrument that can be plugged into an amplifier, just like an electric piano or an electric guitar that can be played right away.

2.3. Sensors

"A sensor is a device that receives a stimulus and responds with an electrical signal" (Fraden, 2010, p. 2). Sensors play an important role in DMIs. They are the first item in the chain when translating performance gestures into sonic output. The translation happens in two stages: first, the gesture has to be translated into an electrical signal – this can be done with a wide range of different sensors; second, after converting this signal into the digital domain – which is most often done by a microcontroller, the signal is mapped to one or more parameters of the synthesis system.

Many different sensors can be used in DMI design. Example categories to classify the different sensor types according to Miranda and Wanderley are:

- Absolute vs. relative
- Contact vs. non contact
- Analog vs. digital

Examples of sensors used in DMIs are: force-sensitive resistors (FSRs), linear and rotary potentiometers, piezoelectric sensors, capacitive sensors and accelerometers – just to name a few. Furthermore, Miranda and Wanderley give an extensive overview and description of sensors commonly used in DMI design. A detailed description of the sensors that are used in the PP is given in chapter 4.

2.4. Feedback

The feedback a DMI provides effects the interaction with the performer and furthermore the expressiveness of the performance. As already mentioned in section 2.1, feedback can be primary or secondary. Primary feedback involves visual, haptic and/or tactile feedback while the sound output of the instrument is the secondary feedback. Bongers (2000) furthermore classifies the different feedback types into *passive* vs. *active*. Active feedback is primarily given by the sound output but can also involve actuators such as tactile stimulators or light sources. The passive feedback is determined by the physical characteristics of the instrument and impacts the control gestures of the performer.

While some feedback types are only accessible by the performer, others can also be witnessed by the spectators directly – e.g sound and light, or indirectly through the performed gestures – e.g haptic feedback. Therefore feedback plays an important role not only for the playability of the instrument but also for its expressiveness (Arfib et al., 2005). Both primary and secondary feedback may be mainly or partly influenced by the parameter mapping of the instrument (see Figure 2.2).

2.5. Parameter Mapping

The choice of parameter mapping is a central element in DMI design. As the previous sections already pointed out, it manifests the linkage between input gestures and sound output but can also effect the primary and secondary feedback. In DMIs this linkage is flexible in contrast to traditional acoustic instruments, where the gestural input and the sound generating entity are combined in the same physical object. For example, the string of a guitar acts both as gestural input unit and sound generation source. This separation in DMIs leads to the fact that there is no necessary fixed causality between a gestural input and the resulting sound of the instrument and therefore the very same input gesture can lead to entirely different sound outputs (Miranda and Wanderley, 2006).

Furthermore the mapping layer is also a determining factor for the constraints and the dimensionality of a DMI. As Magnusson (2010) lays down, "The main bulk of the time spent in learning [an] instrument involves building a habituated mental model of its constraints". User studies on dimensionality (Zappi and McPherson, 2014b) indicate that constraints can even be beneficial for the musical creativity. Since dimensionality determines how the performer is playing the instrument, it also effects the audiences perception of expressiveness (Arfib et al., 2005). Multidimensionality often leads to the fact that it is hard to grasp the link between the sound output of a DMI and the related performance gesture (Gurevich and von Muehlen, 2001).

This can be a problem for both the performer and the audience which expects a observable primary causation for the the sound that is heard.

Different mapping strategies can be applied to DMIs. Rovan et al. (1997) classify these strategies into three categories:

- One-to-One-Mapping, where each independent gestural output is linked with one musical parameter
- Divergent Mapping, where one gestural output is simultaneously linked to more than one musical parameter
- Convergent Mapping, where one musical parameter is assigned to multiple gestures

The choice of mapping is usually done by the instrument designer but may also stay flexible (as in the case of the PP) and thus accessible to the performer. In the latter case the instrumentalist can modify the mapping according to his or her own needs and may even switch between different mappings during one performance.

Another way of achieving parameter mappings is to use a software algorithm which randomly or semi randomly determines the connections. De Campo (2014) presents *Influx*, an automated system that helps to find non-obvious mapping strategies. This concept is described as "Lose Control, Gain Influence".

2.6. Sound Synthesis

Since a gestural controller alone is not able to produce sound, DMIs rely on a sound synthesis unit. This can be a hardware synthesizer or hardware that runs software synthesis. Whereas in the past external MIDI synthesizers commonly where used to run the sound synthesis, today most DMIs use software synthesis run on a laptop computer. Programming environments like Pure Data, Max/MSP, *SuperCollider* et cetera offer a variety of different synthesis techniques from classical additive/-subtractive and FM synthesis over sample based synthesis to physical modelling techniques and granular/wavelet synthesis. Explaining all of these techniques would go beyond the scope of this thesis.

The choice of the synthesis method(s) depends on the capabilities of the employed hardware and the preferences of the instrument designer. It is important to mention that the inherent separation of gestural input and synthesis unit in DMIs means that the applied synthesis strategy does not necessarily have to be fixed but is flexible and thus can be accessed by both the instrument designer and the performer. However, as Cook (2001) points out in his DMI design principles, beside the advantages this flexibility offers, programmability can also be a "curse".

CHAPTER 3 RELATED WORK

A lot of research on the design of DMIs was published in the last decades. Since 2001, a large variety of DMIs have been presented at the annual International Conference on New Interfaces for Musical Expression (NIME Steering Committee, 2016). Although artistic and scientific research in this field dates back earlier – the first *International Computer Music Conference* (ICMC) took place in 1974 (International Computer Music Association, 2016) – the interest in new DMIs has grown due to the availability of high computing power and sensors of all type, which made performances with DMIs commonplace (Cook, 2001).

While some work deals with the principles of DMI design in general (Cook, 2001; Magnusson, 2010; Lyons and Fels, 2012; Paine, 2013), other work focuses on specific aspects such as parameter mapping (Hunt et al., 2003), expressiveness (Arfib et al., 2005) or dimensionality (Zappi and McPherson, 2014b).

Miranda and Wanderley devote a whole book to "Control and Interaction Beyond the Keyboard" but the subject of self-contained instruments stays untouched (Miranda and Wanderley, 2006). Little research is published that covers the development of DMIs with embedded sound synthesis. In the following an overview of published research is given that has been found during the literature research. The focus was on projects that use a microcontroller or microprocessor board for embedded sound synthesis.

3.1. Satellite CCRMA

The *Satellite CCRMA* is a system that was developed to replace laptop and desktop computers in DMI designs with "a compact and integrated control, computation and sound generation platform" (Berdahl and Ju, 2011, p. 1).

Berdahl and Ju state that many new musical instruments and installations subsist only for a very short time since a lot of effort has to be made to keep them working over time for real life performance situations. According to the authors this fact introduces two issues: first, musicians cannot become experts on the instrument for the simple reason that the lifetime of the instrument is too short and consequently the quality of the yielded music is limited; second, the lack of dissemination of new DMIs limits the possibility of sharing knowledge and playing expertise. Berdahl and Ju blame, among other things, the use of general purpose computers for these problems.

Technically, the Satellite CCRMA platform is build upon a microprocessor board which runs an embedded linux system. In the initial version, a *BeagleBone Rev 4*[1] is used while later versions support the *BeagleBoard xM*, the *Raspberry Pi Model B*[2] and the *Raspberry Pi 2* (Berdahl et al., 2013; Berdahl, 2015). The employed microprocessor board is connected to an *Arduino nano*[3] clone sitting on a breadboard. The Arduino microcontroller manages the sensor input, while the breadboard provides rapid prototyping.

Several facts make the Satellite CRRMA very convenient for the use in DMI designs: first, it can run Pure Data which makes the implementation of sound synthesis very accessible; second, it can be programmed and controlled remotely via ethernet and third, all of the employed microprocessors support native floating point computation. Above all, the latter is an important feature for computing sound synthesis algorithms.

[1]http://beagleboard.org/bone
[2]https://www.raspberrypi.org/
[3]https://www.arduino.cc

3.2. JamBerry

The *JamBerry* is a standalone device for distributed network performances (Berdahl and Ju, 2011, p. 1). It is an all-in-one device that "is supposed to be usable in realistic environments" and is supposed to "be a compact system that integrates all important features for easy to setup jamming sessions" (Meier et al., 2014, p. 2). It's purpose is to give musicians the possibility to jam together in a virtual acoustic space connected via the internet.

The JamBerry uses a *Raspberry Pi* (RPI) as sound processing unit. Since this microprocessor board has no audio input and lacks high quality audio outputs (see section 6.1), a codec board is attached to the *integrated interchip sound* (I^2S) pin headers of the microprocessor board in order to provide AD/DA conversion with sample rates up to $192kHz$ and a maximum bit-rate of $24bits$ per sample. The software implementation is based on an embedded linux system in conjunction with ALSA[4]. For the integration of the codec board an appropriate kernel driver had to be written to provide I^2S support. Furthermore a amplifier board is utilised to allow the connection of different sources to the *XLR/TRS* connectors and to provide line-level and headphone output. The audio input to audio output latency of the JamBerry is below $5ms$ and shows that the RPI can be used for DMI designs when extending it with external AD/DA-conversion. However, the effort to achieve this is quite high. Furthermore the JamBerry does not host sound synthesis processes, so no conclusions can be made about the use of the RPI for embedded sound synthesis.

3.3. Vega and Gómez (2012)

A similar approach to the Satellite CCRMA is presented by Vega and Gómez (2012). Their goal was to "build a stable platform for processing and synthesizing audio signals using an open source, low cost, portable computer" (Vega and Gómez, 2012, p. 1).

Vega and Gómez implemented a platform based on a lightweight linux distribution which runs on a BeagleBone microprocessor. JACK and ALSA were used in order to

[4]Advanced Linux Sound Architecture: `http://www.alsa-project.org`

process audio with a sample rate of $48kHz$ and a buffer size of 256 samples. As with the Satellite CCRMA, Pure Data was used for the sound synthesis. The developers managed to run synthesis patches including additive/subtractive synthesis and FM modulation but loading more complex Pure Data objects caused problems.

3.4. D-Box

The *D-Box* is "a novel DMI based on embedded technologies and specifically designed to be appropriated and repurposed in unexpected ways" (Zappi and McPherson, 2014a, 2015; McPherson and Zappi, 2015).

The instrument uses circuits on a breadboard to create hardware-software feedback loops that can modified by the performer. Capacitive touch sensors and piezo pickups are used for the gestural input. The implemented software system runs on a *BeagleBone Black* (BBB)[5] microprocessor with an extension cape that provides eight audio in- and outputs. For this, an audio environment based on a Debian[6] linux system and the Xenomai[7] *application programming interface* (API) was developed. The system operates with a buffer size of four audio samples at $22.05kHz$ and is capable of running up to 700 oscillators using an oscillator bank.

User case studies by Zappi and McPherson (2014b) showed that the possibilities of the instrument are not determined by the instrument designer but the creativity of the performers which used the instrument in a way that went beyond the suggestions of the original designers .

3.5. Others

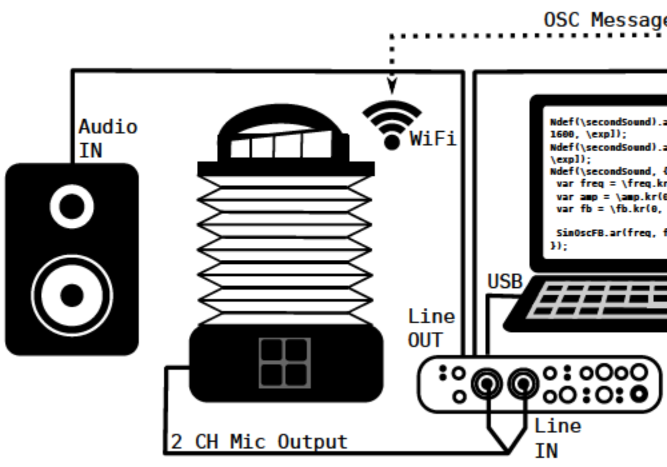
Examples with a physical design that is related to the PP are the *SqueezeVox* (Cook and Leider, 2000), an accordion like controller for models of the human voice, and the *Accordiatron* (Gurevich and von Muehlen, 2001), an accordion inspired instrument that uses a programmable microprocessor to output MIDI to a computer running

[5]`http://beagleboard.org/black`
[6]`https://www.debian.org/`
[7]`https://xenomai.org/`

Max/MSP. Both of the instruments share similarities with the PP due to their related gestural input method, but their development was focused on building new gestural controllers and rather than developing a self-contained DMI.

CHAPTER 4

THE PUSHPULL



```
Ndef(\secondSound).addSpec(\freq, [50,
1600, \exp]);
Ndef(\secondSound).addSpec(\fb, [0, 1,
\exp]);
Ndef(\secondSound, {
 var freq = \freq.kr(440, 0.02);
 var amp = \amp.kr(0, 0.05);
 var fb = \fb.kr(0, 0.02);

 SinOscFB.ar(freq, fb) * amp!2
});
```

Figure 4.1.: Minimal PushPull setup with a laptop that runs the sound synthesis, an audio interface to convert the microphone signals into the digital domain and active loudspeakers for monitoring the sonic output.

The development of the PP was started in 2014 by Dominik Hildebrand Marques Lopes, Amelie Hinrichsen and Till Bovermann within the *design, development and dissemination of new musical instruments* (3DMIN) project at the UdK Berlin (Hinrichsen et al., 2014; Bovermann et al., 2014). Since then, several versions of the instrument have been developed. In 2015 the $\mathrm{PP}_{\mathrm{SE}}$ was built, the version that was the basis for the development of the $\mathrm{PP}_{\mathrm{EMB}}$. Before describing the process of developing the $\mathrm{PP}_{\mathrm{EMB}}$, the overall characteristics of the initial instrument version as well as the hardware and software implementation are laid down. This is the basis on what the requirements for the embedded hardware and the software implementation of the $\mathrm{PP}_{\mathrm{EMB}}$ have been set.

The chapter is divided into three main parts. The first part will deal with the instrument corpus, which determines not only the exterior appearance of the instrument but also its haptic properties and constraints. The second part will give an overview of the technologies that are employed and which provide the technical functionality of the $\mathrm{PP}_{\mathrm{SE}}$. This includes sensor technologies, embedded computing hardware and further technologies that play an important role in the development of the $\mathrm{PP}_{\mathrm{EMB}}$. The last part briefly deals with the sound synthesis of the $\mathrm{PP}_{\mathrm{SE}}$

Figure 4.1 shows an exemplary minimal setup for playing the $\mathrm{PP}_{\mathrm{SE}}$. The two channel microphone output of the instrument is connected with two line inputs of an audio interface attached to a computer. The gestural input provided by the sensors of the PP is sent over WiFi via *open sound control* (OSC) messages to an external computer which runs the SuperCollider programming environment and is responsible for the sound synthesis. The resulting digital audio signal is converted by the audio interface into the analog domain and monitored with active speakers.

4.1. Instrument Corpus

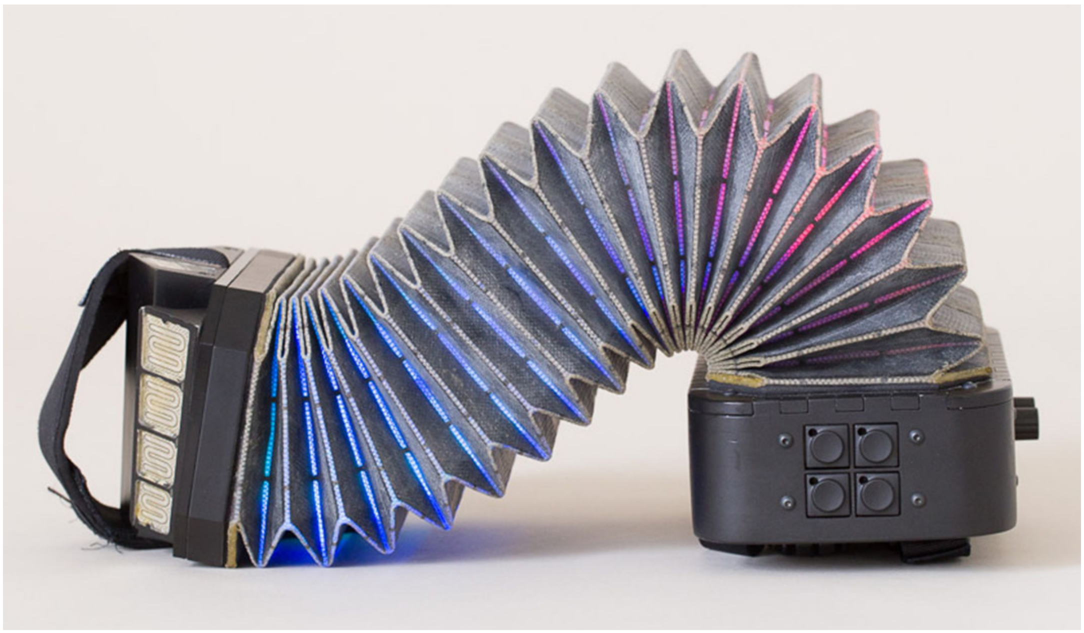


Figure 4.2.: The PushPull. © 3DMIN project, adapted by permission.

Bellows

The bellows is the main element of the PP. Beside its crucial influence on the instruments exterior appearance its physical characteristics strongly determine the gestures of the performer. Its visual appearance is reminiscent of traditional squeezeboxes; however, in contrast to the bellows of a squeezebox it can also be rotated to some degree.

When the bellows is moved an air flow into and out of the valves is produced and picked up by two microphones. This breathing of the bellows is used as a sound source and/or as a control for synthesis parameters.

The skeleton of the bellows is made out of perforated cardboard with a mirror film glued to the inside. Its translucent character allows light to shine through the kinks of the folded bellows which are coated with latex. Figure 4.3 shows pictures of the construction.

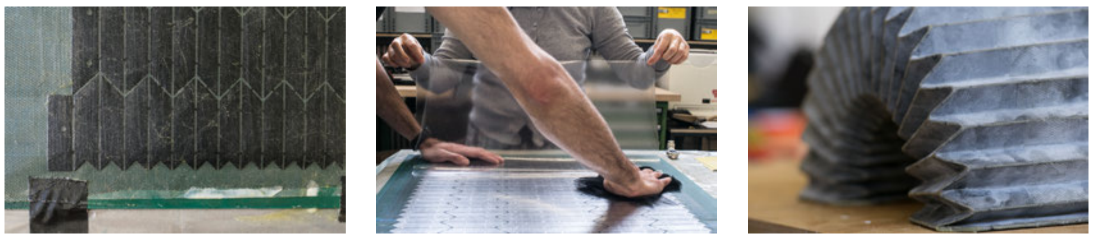


Figure 4.3.: Construction of the bellows. © 3DMIN project, adapted by permission.

Box

The function of the PP_{SE}'s wooden box is to enclose the components that sit in the bottom of the instrument: the valves and the microphones, a microphone preamplifier, four buttons, two rotary encoders and an *ATmega328-PU* microcontroller. The box is made out of laser-cut plywood. The different wooden pieces provide notches for parts that are accessible from the outside (e.g. the push buttons and rotary encoders). Figure 4.4 shows pictures of the box assembly.

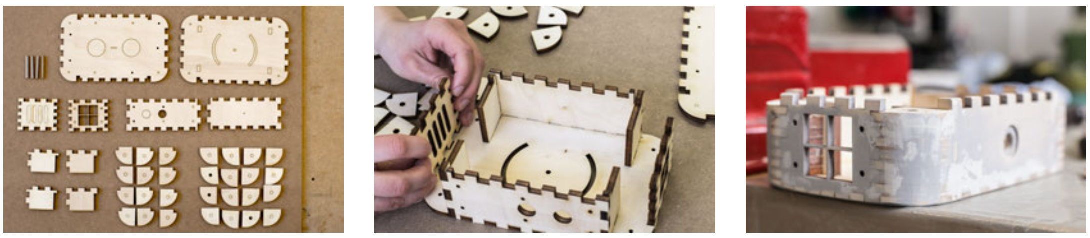


Figure 4.4.: Construction of the box. © 3DMIN project, adapted by permission.

Hand Piece

The hand piece forms the upper logical unit of the PP. It's most important function is to provide the performer with the possibility to grasp the instrument – via a neoprene hand strap – and thus to transmit the motion of the hand to the bellows. Furthermore it includes six capacitive touch sensors (see section 4.3), the mainboard and the power supply of the $\mathrm{PP}_{\mathrm{SE}}$.

The position of the hand block can be adjusted, via two metal rails that are mounted on the base plate, to match the various hand shapes of different players. Adjustments can also be made to the position and angle of the part that mounts the capacitive sensors which can be touched by the index, middle, ring and little finger.

4.2. Overview of the Signal Flow

Figure 4.5 shows a *Composite Structure Diagram* of the signal flow inside the $\mathrm{PP}_{\mathrm{SE}}$. The light blue boxes represent the three different body parts of the instrument: the box, the bellows and the hand piece. Each of these parts embeds several electronic components (dark-blue).

The six capacitive sensors (see section 4.3) are connected to analog inputs of a *Programmable System-on-Chip* (PSoC 4) microcontroller (Cypress Semiconductor Corporation, 2016), which sits on the mainboard in the frame of the hand piece. The microcontroller samples the sensor data and transmits it asynchronously via a serial data line to a *x-OSC* board[1].

The x-OSC has several functions in the $\mathrm{PP}_{\mathrm{SE}}$. First, it has a built in gyroscope and accelerometer. The data from these sensors, which capture the position and

[1] `http://x-io.co.uk/x-osc/`

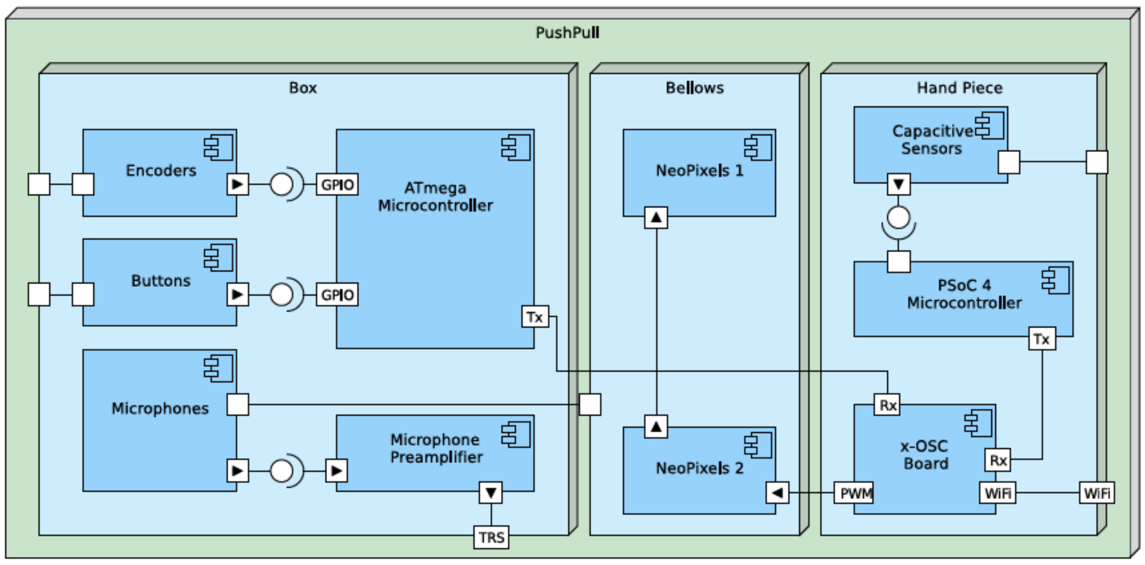


Figure 4.5.: The illustration is an overview of the signal flow inside the PushPull student edition (UML composite structure) and shows the three main parts of the instrument body (light-blue): box, bellows and hand piece. These parts contain the embedded hardware parts (dark-blue) which are connected through wires. White rectangles represent hardware interfaces such as *general purpose input/output lines* (GPIOs), serial (RX, TX) or *pulse-width modulation* (PWM). White rectangles on instrument body parts imply that an interface to the outside of the instrument is provided such as push buttons or audio output (TRS)

.

movement of the hand piece, is sent via OSC messages to an external computer. Furthermore, it receives sensor data from the PSoC 4 and button/encoder data from the ATmega328-PU microcontroller. This information is also sent to the external computer through OSC messages. From within SuperCollider, commands can be sent to the x-OSC to control eight *NeoPixels* LEDs inside the bellows. This is done through a PWM signal that goes through the first four LEDs in the hand piece frame (facing towards the inside of the bellows) before it is transmitted further downwards to the four LEDs in the bottom part of the bellows.

The *push buttons* and *rotary encoders* on the box are connected to the digital pins of a ATmega328-PU microcontroller, which decodes and transmits the sampled signals via a serial data line that goes from the bottom of the instrument through the inside of the bellows to the x-OSC board in the top of the instrument. What makes the ATmega328-PU chip so convenient for the use in the PP$_{\mathrm{SE}}$ is the fact that it is also used on the Arduino UNO[2] and therefore can be programmed via the Arduino *integrated development environment* (IDE). Furthermore an extensive software library is provided which enables developers to accomplish complex tasks like serial communication with minimal effort. With just a few additional components, soldered to a *printed circuit board* (PCB), the ATmega328-PU handles the sampling and decoding of the buttons and rotary encoders.

The *power supply*, which is in the top part of the instrument, is not shown in the diagram. The PP_{SE} is powered by a lithium-ion battery which provides the supply voltage for all electronic components. A power regulator transforms the voltage of the battery to $3.7V$ and controls the charging of the battery when the PP_{SE} is connected to an external power supply through a mini-USB connector.

4.3. Capacitive Touch Sensors

This section explains the method of *capacitive touch sensing* and is based on the technical reference for designing touch sensors for the PSoC 4 microcontroller (see Cypress Semiconductor Corporation, 2016). The six capacitive touch sensors of the PP are implemented on PCBs as shown in Figure 4.6.

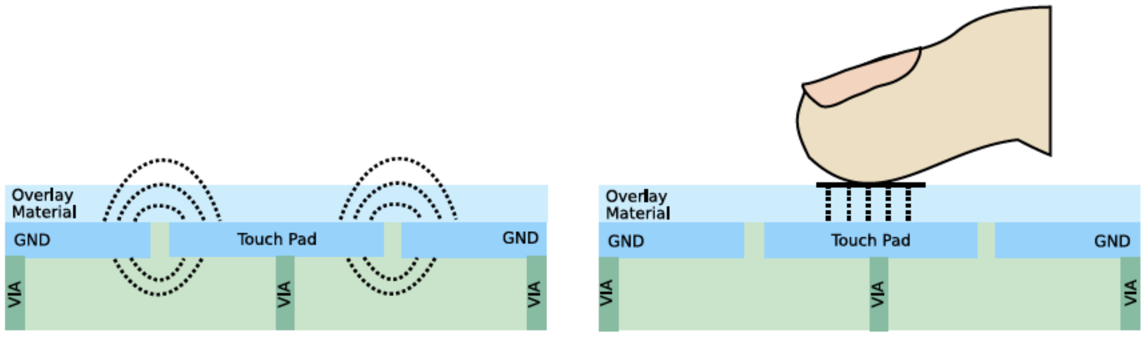


Figure 4.6.: Capacitive touch sensors implemented on a PCB. The dotted lines imply the electric field lines when the sensor is untouched (left) or touched (right).

The top layer of each PCB has two isolated conductive surfaces. While one of the surfaces acts as an electrical ground, the other surface is the sensor pad and is connected to an input pin of the PSoC 4. When the sensor is untouched the

[2]`https://www.arduino.cc`

circuit acts as a *parasitic capacitance*. Once the sensor gets touched the finger acts as a grounded conductive plane parallel to the sensor pad. This results from the conductive characteristic of the human skin and the large mass of the body. The finger consequently acts as an electrical ground. In simplified terms the finger and the sensor pad act as a plate capacitor with capacitance:

$$C_F = \frac{\epsilon_0 \epsilon_r A}{d} \tag{4.1}$$

Where:

- d is the thickness of the overlay material
- ϵ_0 is the free space permittivity
- ϵ_r is the relative permittivity of the overlay
- A is the overlapping area of the finger and the sensor pad

Since the parasitic capacitance of the circuit and the finger capacitance are parallel between the sensor pin and ground, the total sensed capacitance C_S results to:

$$C_S = C_P + C_F \tag{4.2}$$

Where:

- C_F is the finger capacitance from formula 4.3.
- C_P is the parasitic capacitance which results from all the conductors on the PCB (sensor pad, traces, vias and ground planes), any metal in the enclosure and the internal capacitances of the PSoC 4.

When the sensor pad is untouched only the parasitic capacitance is sensed and therefore C_S equals C_P. The PSoC 4 chip converts the capacitance C_S into equivalent digital counts, which can then be further processed.

4.4. Microphones and Mirophone Pre-Amplification

The airflow into and out of the bellows is picked up by two electret microphones located inside the valves. Electret microphones are condenser microphones which can be implemented with a minimal amount of space. A supply voltage and pre-amplification is needed in order to gain a suitable signal level with electret microphones (see Schneider, 2008). The supply voltage is provided by a microphone preamplifier that was developed and assembled for the $\mathrm{PP_{SE}}$. Since the microphone pre-amplifier is obsolete in the $\mathrm{PP_{EMB}}$, no further details about its implementation are given here.

4.5. NeoPixel LEDs

A primary feedback method of the PP is the lighting from inside of the instrument (see Figure 4.7). The $\mathrm{PP_{SE}}$ uses this visual feedback to indicate which instrument is loaded.

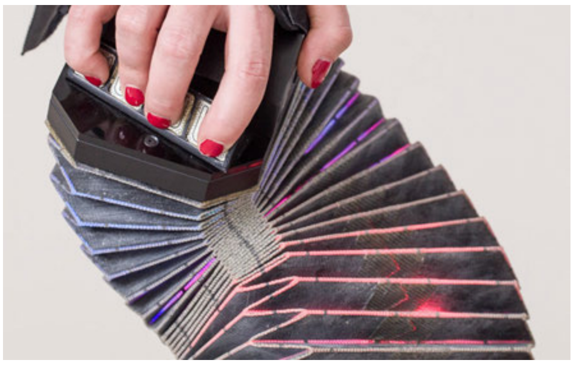


Figure 4.7.: Illuminated bellows. © 3DMIN project, adapted by permission.

Technically this feature is enabled by eight NeoPixel LEDs, integrated inside the bellows. NeoPixel is *Adafruit*'s[3] brand for *RGB* colour pixels based on the WS2812 LED drivers (Worldsemi Corporation, n.d.). The pixels can be cascaded and therefore controlled via one single data line. With the help of a microcontroller that

[3]`https://www.adafruit.com/products/1312`

features PWM and an implementation of the protocol the colour of each individual LED can be controlled.

4.6. Sound Synthesis

One of the PP_{SE}'s main features is its flexibility in terms of parameter mapping and sound synthesis. This flexibility results from the fact that the parameter mapping and sound synthesis are implemented and accessible through SuperCollider, which runs on an external computer. Different sound synthesis and parameter strategies can be implemented. These instrument patches share the same gestural control inputs methods of the PP but lead to different sonic output. This is similar to keyboard controlled synthesizers which are used to implement a wide variety of different sounds with the same gestural input methods. The possible synthesis techniques that can be implemented for the PP_{SE} are theoretically only limited by the processing power of the utilised computer and the features of the SuperCollider environment.

CHAPTER 5 REQUIREMENTS AND APPROACH

Before the iterative and incremental development of the hardware and software was started, requirements were set for the PP_{EMB}. The requirements were largely determined by the characteristics and the PP_{SE}'s range of functions together with the goal to overcome the need for an external computer as a synthesis unit. They were divided into general requirements and requirements that concern only the sound synthesis.

5.1. General Requirements

The most fundamental requirement for the PP_{EMB} was it's independence from an external computer during a performance. This means that the employed computing hardware had to be small enough to be embedded into the instrument corpus. Requirements that were set based on the characteristics of the PP_{SE} are:

- Incorporation of the sensor and actuator technologies described in chapter 4
- Programmability, i.e. the sound synthesis and parameter mapping has to be flexible and modifiable
- Possibility of changing between different instrument patches during performance

- High quality audio processing and AD-conversion

Furthermore the following requirements were set, which are important for every DMI:

- Low audio and control latency
- Stability and reliability
- Durability and Maintainability

These points impact not only the choice of hardware but also the software design and integration. Maintainability is heavily influenced by a clean and comprehensible programming style. This includes modular programming as well as a detailed documentation of the implemented software parts.

5.2. Requirements for the Synthesis Environment

The possible synthesis techniques that can be implemented for the PP_{SE} are theoretically only limited by the processing power of the utilised computer and the features of the SuperCollider environment. It is obvious that PP_{EMB}'s embedded hardware cannot compete with a laptop or desktop computer in terms of processing power. Just as it's software implementation cannot offer the same range of functions as the SuperCollider environment, which is being developed since two decades. The goal was not to provide the same theoretical possibilities but the same essential functionality.

For the PP_{EMB}'s sound synthesis capabilities, the following range of functions was set as a requirement:

- Additive and subtractive synthesis with different wave forms and multiple oscillators
- Processing samples, i.e. storing, loading and triggering samples
- Polyphony, i.e. playing more than one voice at the same time

- Envelope following
- Basic effects such as reverb, distortion and delay
- Dynamic range control such as compression, expansion and limiting
- Flexible parameter mapping and modulation of the oscillators, filters and effects

A basic prerequisite was that the embedded hardware has enough processing power to compute the *digital signal processing* (DSP) algorithms in order to perform these tasks. Furthermore these synthesis methods have to be implemented for the chosen hardware.

5.3. Iterative and Incremental Development

Iterative and incremental development combines methods of two different development approaches. Incremental development describes a process in which different parts of a system are developed in subsequent stages and are integrated into the whole product in the end, while in iterative development the system is implemented in iterative development cycles, ensuring that reworks of individual parts can be incorporated quickly (see Cockburn, 2008).

After the hardware for the sound synthesis unit was chosen according to the set requirements, a simple prototype with limited functionality was developed in the first iterative development cycle. The compatibility of the hardware was examined and validated. Therefore, test programs with debugging routines were implemented which allowed to test individual parts of the software and hardware.

After validating the prototype and examining the requirements for the individual parts, different models for the software and hardware design were developed. The *unified modeling language* (UML) was used to visualise the structure and behaviour of the system. *Composite structure* and *class diagrams* were used for the static structure of the hardware and software while the behaviour of the designed software was furthermore modelled with the help of *activity* and *sequence diagrams* (see

Object Management Group, 2015). A detailed description of the UML is beyond the scope of this thesis and, for the sake of clarity, simplified models are presented here, showing only the most important details. The models were implemented and refined in several iterative cycles before the final prototype was evaluated according to the requirements that were set before.

CHAPTER 6
HARDWARE IMPLEMENTATION

Before describing the hardware implementation, an overview is given of the hardware that was considered for embedding the sound synthesis. This includes a comparison of devices that were reviewed more closely and an explanation for the final decision that was made.

6.1. Reviewed Hardware Devices

Microcontrollers have become more convenient since the rise of the Arduino. While they were hard to program in the past and therefore reserved for experts, they are now accessible for a considerable number of persons with basic programming skills (Smith and Michon, 2015). On the one hand their user-friendliness and affordability make them very popular for use in art installations and the control of sensor input in DMIs, on the other hand they usually lack fast digital-to-analog conversion (DAC) and do not offer enough processing power for dedicated sound synthesis. In the field of instrument design they are mainly used for processing sensor data. Since the requirements that were set for the sound synthesis environment in section 5.2 cannot be met by conventional microcontrollers, devices such as the Arduino were not considered for embedding the sound synthesis.

Promising hardware to meet the requirements were so called microprocessor boards, which bring full featured minicomputers at the size of a credit card that cost less than 100€. Various products with different specifications are sold in this category but mainly two brands got more attention from the DMI developer scene recently (see chapter 3): the RPI and the *BeagleBone*. Different derivates of these two products exist of which two were considered for the development of the $\mathrm{PP}_{\mathrm{EMB}}$, namely the *Raspberry Pi 2* (RPI2) and the BBB.

Numerous embedded linux systems are available for both devices and they provide various interfaces for sensor input and the communication with other devices. Furthermore the RPI2 and the BBB both offer hardware floating point support, which makes them suitable for dedicated audio processing. A main drawback of these two boards is that they neither provide audio inputs by default, nor offer suitable audio outputs. Indeed, the RPI2 ships with a consumer quality stereo mini jack output and the BBB provides audio output via its HDMI connector, but in order to be able to use convenient analog-to-digital (AD) and digital-to-analog (DA) conversion further extensions are needed. One can use either a small audio interface connected to one of the USB ports, or an audio extension cape like the *Bela*[1] for the BBB or the *Cirrus Logic Audio Card*[2] for the RPI2.

A relatively new hardware device is the *Axoloti core* (AXOC)[3], a powerful microcontroller board which is graphically programmable via a software that is similar to Max/MSP and Pure Data. The most important difference to the above mentioned devices is the fact that the board provides integrated $24bit/96kHz$ capable stereo AD/DA conversion, as well as MIDI in- and outputs. Furthermore, unlike the other hardware devices, it is specially designed for audio applications, and does not come with a lot of interfaces that aren't necessarily needed for DMI design (e.g. Ethernet or *high-definition multimedia interface* (HDMI)). A critical factor is that the AXOC was developed by a single person, however it has a growing community of users[4] (Taelman, 2016a).

[1]`http://www.bela.io`
[2]`https://www.element14.com/community/community/raspberry-pi/raspberry-pi-accessories/cirrus_logic_audio_card`
[3]`http://www.axoloti.com/`
[4]`http://community.axoloti.com/`

	Device		
Specifications	**RPI2 (Model B)**	**BBB (Rev. C)**	**AXOC**
CPU	900MHz ARM Cortex-A7	1GHz ARM Cortex-A8	168MHz ARM Cortex-M4
RAM	1 GB SDRAM	512MB DDR3 RAM	8 MB SDRAM
Flash	microSD	4 GB on board, microSD	microSD
GPIOs	40	69	16
Audio IN	-	-	2 (6.35mm TRS & connection pads)
Audio OUT	2 (3.5mm TRS)	-	2 (6.35mm TRS & 3.5mm headphone TRS)
Floating-Point	yes	yes	yes
Extras	4 x USB, Ethernet, HDMI	2 x USB, Ethernet, HDMI	USB, microUSB, MIDI-IN, MIDI-OUT

Table 6.1.: Comparison of the three contemplated devices for the sound synthesis unit. The table shows the relevant specifications for the *Raspberry Pi 2* (RPI2), the *BeagleBone Black* (BBB) and the *Axoloti core* (AXOC), including employed CPU, *random-access memory* (RAM), *Flash* memory, number of GPIOs, audio input and output channels, capability of floating-point calculation and additional interfaces.

6.2. Comparison and Decision

Table 6.1 shows the relevant hardware specifications for the three devices that where reviewed more closely for comparison. While the BBB offers the most powerful CPU, the RPI2 has more RAM. Since the AXOC is a microcontroller, which is not intended to run a general purpose operating system, it comes with far less computing power and memory. Nonetheless, the fact that its core supports DSP instructions and has a floating-point unit makes it powerful enough for advanced DSP processing. All three devices have a micro SD-card slot for storage, the BBB additionally has $4GB$ on-board flash memory. Furthermore, each device has a sufficient number of GPIOs for the use in the PP_{EMB}. In terms of audio connectivity the AXOC clearly outperforms the other devices since it offers two input and two output channels. The input channels can be DC-biased for the use of electret-microphones and the output channels are also available via a headphone connector. The AXOC also offers the most useful extra features such as *musical instrument digital interface* (MIDI) input and output.

Three facts particularly made the AXOC the first choice for the use as a synthesis unit:

- Its out-of-the box audio connectivity, i.e. no extensions are needed in contrast to the other two devices
- It offers bias microphone inputs, i.e. the employed microphone amplifier in the $\mathrm{PP}_{\mathrm{SE}}$ becomes obsolete
- A software framework is provided and maintained that already offers much of the required functionality

Admittedly, the other two devices theoretically provide way more processing power, but as the work presented in chapter 3 shows, it takes a lot of effort to set up a running system that is suitable for dedicated audio processing to make use of these capabilities. Furthermore these general purpose devices are shipped with a lot of extra functionality such as Ethernet and HDMI connectors which were not attempted to be used in the $\mathrm{PP}_{\mathrm{EMB}}$.

6.3. Implementation

As described in chapter 5, in a first iterative development cycle a simple prototype of the $\mathrm{PP}_{\mathrm{EMB}}$ with limited functionality was built to validate the compatibility of the AXOC with the other hardware parts. Tests with a basic software framework gained promising results and consequently the Axoloti was embedded into the instrument.

Figure 6.1 shows a *Composite Structure Diagram* (UML) which reflects the changes to the embedded hardware that were made and the resulting signal flow inside the $\mathrm{PP}_{\mathrm{EMB}}$.

The following changes have been made to the embedded hardware:

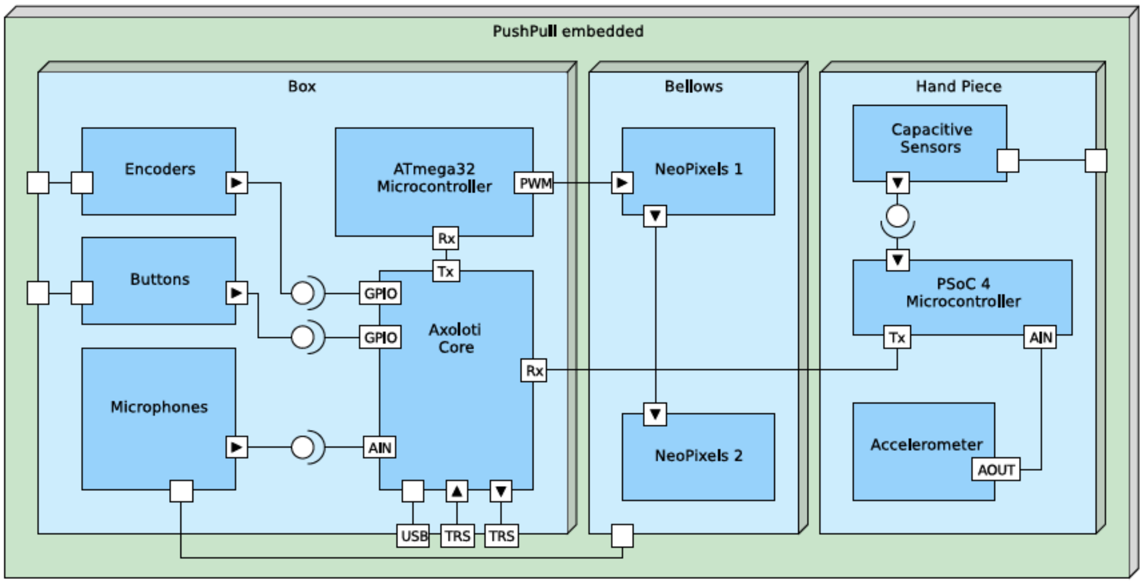


Figure 6.1.: Overview of the signal flow inside the PP_{EMB} (UML composite structure). The illustration shows the three main parts of the instrument body (light-blue): box, bellows and hand piece. These parts contain the embedded hardware parts (dark-blue) which are connected through multiple wires. White rectangles represent hardware interfaces such as GPIOs, serial communication (Rx, Tx) or PWM. White rectangles on instrument body parts imply that an interface to the outside of the instrument is provided, such as the push buttons or the audio output (TRS).

Removal of the microphone pre-amplifier and embedding the AXOC The AXOC was installed at the very bottom of the instrument inside the box, with its connectors accessible from the outside. The electret microphones were now connected to the biased analog inputs of the AXOC. This made the microphone amplifier board obsolete and was very beneficial since space was gained inside the box which could be used for the AXOC. With the help of a laser cutter, the box could be modified in way that access to the connectors of the Axoloti board was provided.

Changing the power supply The power supply was removed from the top part of the instrument. There are two reasons for this. First, the main computing device in the PP_{EMB} is the AXOC, which is located in the bottom of the instrument. Therefore, it made sense to move the power source to the bottom. Second, the AXOC can be either powered via its mini-USB connector, its DC connector or a regulated power source connected to the solder pads. Since the power supply that was used in the PP_{SE} neither provides enough voltage, nor switching power modes, unlike the power regulator of the AXOC, the decision was made to forgo the battery and power the PP_{EMB} via the usb connector of the AXOC. In a future version, the power regulator of the AXOC will be coupled with a suitable *lithium polymer battery* (LiPo) to provide battery based power supply. Since this was not a main priority for the PP_{EMB} and would have required major modifications to the box design, this measure was postponed.

Substituting the x-OSC with an accelerometer module The x-OSC board is the most expensive element in the PP_{SE}, however it's WiFi functionality was not needed anymore and therefore the x-OSC was replaced with an inexpensive accelerometer module (MMA7361, Apex Electrix LLC, 2013).

Extending the function of the PSoC 4 In the PP_{EMB} the PSoC 4 microcontroller does not only read the capacitive sensors, but also samples the analog accelerometer signals. For this purpose the analog outputs of the accelerometer were connected to the analog inputs of the PSoC 4. The data gathered from the capacitive sensors and the accelerometer in the hand piece is now transmitted via a serial connection that was implemented from the serial output (Tx) of the PSoC 4 to the serial input of the AXOC (Rx) in the bottom of the instrument.

Sampling of the encoders and buttons with the AXOC The sampling of the rotary encoders and buttons in the PP_{EMB} is done by the AXOC. Therefore these parts were connected to the digital inputs of the board. Consequently, the ATmega328-PU microcontroller has lost its function. It is now used to drive the NeoPixel LEDs.

Controlling of the NeoPixel LEDs with the ATmega328-PU The AXOC sends commands for the NeoPixels via a serial data line to the ATmega328-PU, which controls the LEDs via its PWM output. Theoretically the NeoPixel LEDs could also be driven by the AXOC itself. However, a first iterative development step showed that an implementation of the NeoPixel protocol for the AXOC would have required great effort. Tests with Adafruit's NeoPixel library for the ATmega328-PU revealed that the NeoPixels were easier controlled with the help of this microcontroller.

Changing the wiring of the NeoPixels The order in which the NeoPixels receive the control signal had to be changed. Whereas initially the signal cascade started in the top of the instrument, now it starts at the bottom – the location of the ATmega328-PU. What used to be the last LED in the cascade before has become the first one and vice versa.

More detailed illustrations of the wiring before and after embedding the sound synthesis can be found in appendix B.

CHAPTER 7 SOFTWARE IMPLEMENTATION

The implementation of the software will be laid down in four sections. The first section deals with the serial communication protocol that has been implemented for all three microcontrollers. The following three sections cover the software implementation for each microcontroller individually. Figure 7.1 gives an overview of the implemented software components and the associated files.

7.1. Serial Communication Protocol

One of the main efforts in developing the self-contained version of the PP was implementing the internal communication of the instrument, i.e. the exchange of information between the three microcontrollers.

Serial communication is widely used for developing embedded systems as it requires only a minimum of wired connections between the peripherals. These connections are called buses and are either *synchronous* or *asynchronous*. Synchronous buses used by protocols such as *inter-integrated circuit* (I^2C) or *serial peripheral interface* (SPI) require an additional wire for synchronising the transmission via a clock signal, whereas in asynchronous serial communication a single wire between the transmitter and the receiver is sufficient to transmit data in one direction (see Barr and Massa, 2006).

The minimum number of required wires and the fact that all three microcon-

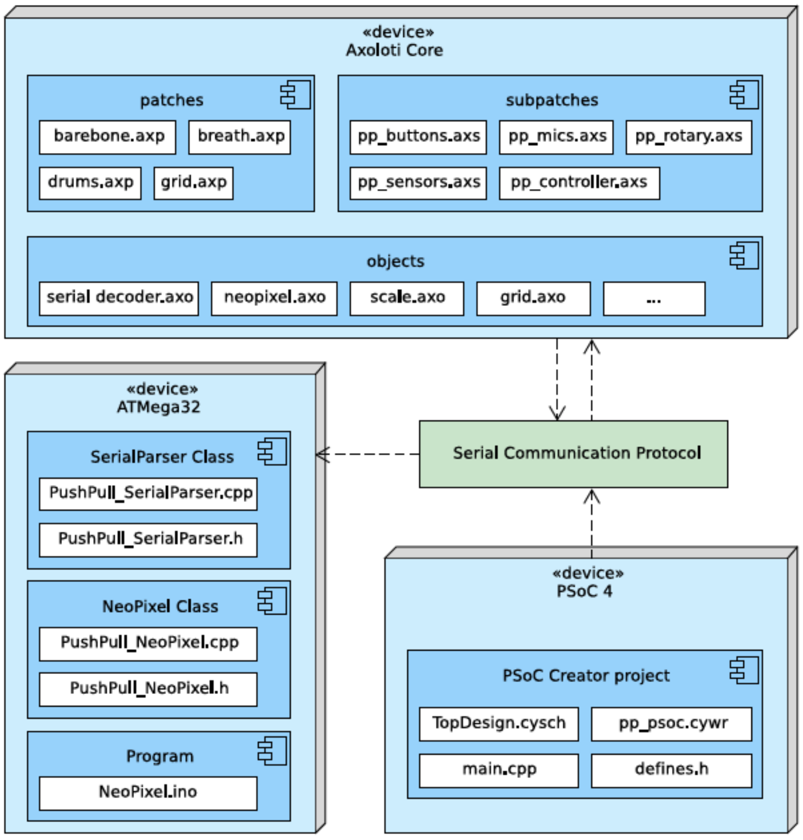


Figure 7.1.: Overview of the implemented software for the three microcontrollers Axoloti Core, ATMega32 and PSoC 4 (light-blue). The illustration shows the software components (dark-blue) with the associated files (white) for each device. The serial communication protocol (green) is used to exchange data between the devices and is implemented for each device.

trollers provide serial interfaces made asynchronous serial communication the first choice for the exchange of data between the devices. Furthermore debugging asynchronous serial communication via a serial terminal is very straightforward and thus a beneficial feature in the development process.

In asynchronous serial communication information is sent bitwise over a single data line. Bytes are framed with a start bit and a stop bit to indicate the start and the end of the transmission. More rarely an additional stop bit or parity bit is used. The transmission rate is called baud rate and has to be set to the same value for both devices in order to sync the transmission. In digital systems one baud (Bd) equals one bit per second. Ten bits are needed to transfer one byte of information, thus a typical baud rate of $115200Bd$ allows the transmission of 11520 data bytes per second.

For the PP_{EMB} a consistent communication protocol has been developed to send messages between the devices. The encoding and decoding of the serial messages incorporates elements of the *high-level data link control* (HDLC) protocol (Davis, 2012). Figure 7.2 illustrates the structure of a serial message according to the implemented protocol.

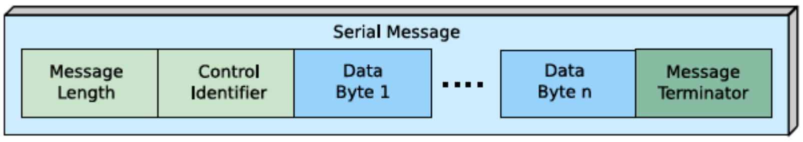


Figure 7.2.: Serial message encoding.

Each block in the illustration represents one byte of data. The message starts with the information of how many bytes will follow. Since the number of data bytes in the control messages vary, this information is used to perform some simple error checking at the receiver. In an early development stage many errors appeared while decoding messages received by the AXOC. Bytes where missed especially at higher baud rates. Consequently an error checking method was introduced in order to allow higher baud rates to lower the control latency. In a later development stage it turned out that the errors were caused by interrupts in the serial decoding thread instead of transmission errors (see serial decoder description in subsection 7.2.4).

The message length byte is followed by a *control identifier* byte. This information is used for parsing the message. The control identifier indicates what kind of data is sent (e.g value of capacitive sensor one, acceleration on the x-axis, NeoPixel command type one etc.). Next, a variable number of bytes is sent containing the control values or commands. Finally the message is closed with a terminator flag that indicates the end of the message.

Since the terminator byte can also appear in the control data, control-octet transparency was applied as suggested by Simpson (1994). Once the message terminator (0x7E) or a control escape (0x7D) appears in the data, a control escape is sent, followed by the original data byte with the fifth bit inverted. This *byte stuffing* is reverted by the receiver. When an escape byte is received, the following data byte is recovered by flipping the fifth bit again.

7.2. Axoloti Software

In order to understand the software components that were implemented for the AXOC, first the basics of the *axoloti software environment* (ASE) are introduced.

7.2.1. Basics

Firmware

The AXOC is a stand-alone microcontroller board that runs an embedded *real-time operating system* (RTOS), which provides the drivers for the hardware interfaces. The RTOS and all software that enables the base functionality is part of the firmware, which is preinstalled on the microcontroller and implements the connectivity with an external computer. Firmware upgrades are uploaded via the Axoloti patcher software.

Patcher

The Axoloti patcher provides a *graphical user interface* (GUI) where sound synthesis patches can be arranged similar to Max/MSP or Pure Data. The GUI is written in Java and translates the patch into C++ code. Figure Figure 7.3 shows a simple patch arrangement with the Axoloti patcher software. By clicking the *live* button, a binary executable is created and uploaded to the microcontroller. Subsequently, the GUI is locked and the patch is executed on the microcontroller, whereas control parameters of the patch can still be accessed and modified in the patcher window (see Taelman, 2016b).

Objects

A patch contains different *objects* that perform specific tasks. The modular objects share data when wires are connected to their inputs and outputs. Five different main types of in- and outputs exist: *audio buffer*, *integer*, *fractional*, *boolean* and *string*. These types are indicated by different colours and are further subdivided (e.g. positive and bipolar for integer and fractional values). Furthermore, objects can include parameters, attributes and displays. Parameters can be modified at run

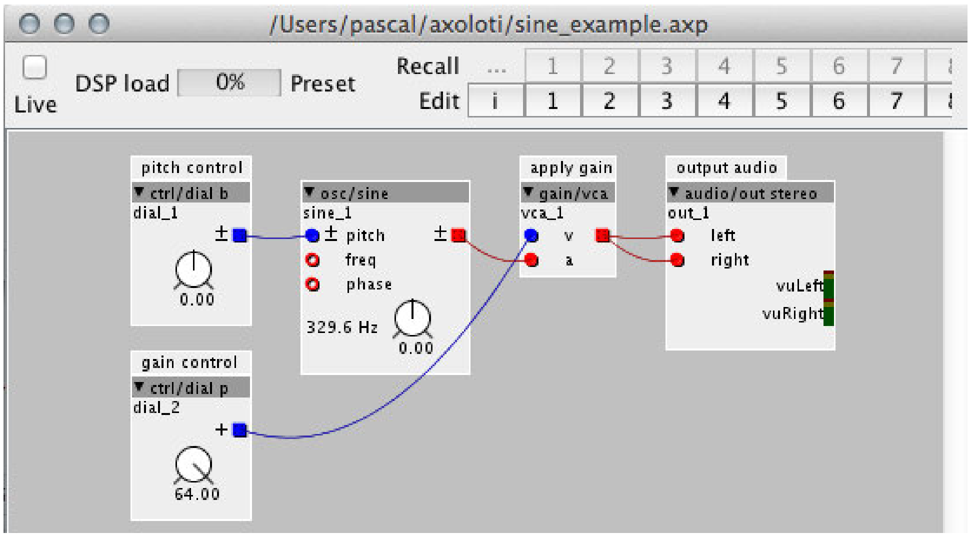


Figure 7.3.: Screenshot of the Axoloti patcher software. The patch includes a sine oscillator object with frequency control and objets to control the gain before the audio is output.

time while attributes are set in the editing process. Displays provide the possibility to show data in the patcher window when the patch is running.

The Axoloti object library offers numerous elements such as oscillators, envelope controls, filters, effects and many more. Additionally further objects can be implemented. This is done by using either the *extensible markup language* (XML) and a standard text editor or by using the provided *object editor*.

An object includes code sections for *srate* and *krate* C/C++ code. While srate code is executed at the defined sample rate of $48kHz$, krate code is executed 3000 times per second, which results from the buffer size of 16 samples (48000/16 = 3000).

The normal range for inputs and outputs (audio, integer and fractional) is from -64 to 64 or 0 to 64 (integer and fractional) units. In which way these general units relate to real world units – e.g. the frequency in Hz of an oscillators pitch input – depends on the implementation of the object. This follows the principle of modular synthesizers which use a defined control voltage range (-5V to 5V) but can be confusing for Max/MSP or SuperCollider users who expect to be able to input frequency in Hz.

Subpatches

Patches can also contain *subpatches*. These are patches inside patches, which usually combine a block of logic that performs a specific task. They are especially useful when the same logic is used more than once in the parent patch or in different patches. Parameters of subpatches can be made visible on the parent patch. Sharing values between the parent patch and the subpatch is possible via defined inlets and outlets of the subpatch.

Uploading Patches

Once a patch is finished it can be uploaded to the internal flash memory of the AXOC as a startup patch or to a *secure digital* (SD) card. The startup patch is loaded once the AXOC is powered and the firmware has booted. Only one patch can run at a time and consequently loading a patch from the SD card has to be done from within another running patch (this can also be the startup patch). The *patch bank editor* provides a tool that can be used to upload an index file to the SD card with a list of patches. This file helps to load different patches from the SD card via their index in the created patch bank without having to store the filenames of all patches in every single patch.

7.2.2. Instrument Patch Model

This section gives insight into the implementation of an instrument patch in the ASE. Each patch contains a specific parameter mapping and sound synthesis implementation for the PP_{EMB}. The different instrument patches all share the same gestural input but lead to different styles of performing and sonic output. Figure 7.4 illustrates how each instrument patch is implemented in the ASE. Specific implementations are described in chapter 8.

Each instrument patch includes several subpatches (dark-blue rectangles) that provide the gestural input of the PP_{EMB} and a sound synthesis implementation which is connected to the input controls via a parameter mapping. Furthermore it contains a *neopixel* object for controlling the NeoPixel LEDs and a controller

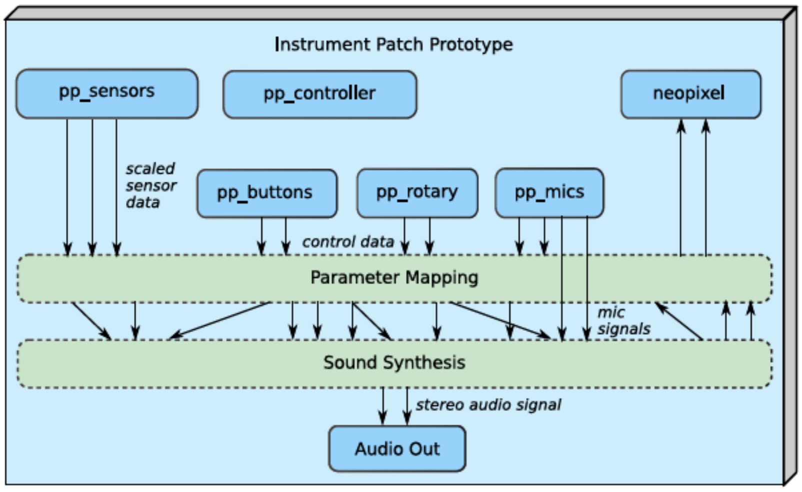


Figure 7.4.: PP$_{\text{EMB}}$ instrument patch model. Dark blue rectangles are Axoloti subpatches or objects, green rectangles represent a block of logic in the Axoloti patcher and arrows represent the signal flow between the elements.

subpatch which provides the logic to change between different instrument patches. Green rectangles represent blocks of logic in the patch. These are not single elements but an abstract of a specific function complex in the instrument patch, containing several objects and/or subpatches. Arrows represent the connections between objects and subpatches. The fact that the pp_mics subpatch in Figure 7.4 has arrows both to the *Parameter Mapping* and to the *Sound Synthesis* logic unit implies that the microphone signals can be used as a sound source and/or for controlling parameters of the sound synthesis. For the latter one has to apply an envelope follower first, in order to obtain a usable control signal. For reasons of simplicity the model does not include every single connection nor does it contain all the elements that are actually used, only the ones which are characteristic for the implementation of an PP$_{\text{EMB}}$ instrument patch.

7.2.3. Subpatches

This section lays down the implementation of the subpatches that have been implemented to provide an interface for the gestural control input of the PP$_{\text{EMB}}$.

pp_sensors subpatch

The *pp_ sensors* subpatch provides the sensor data of the capacitive *touch pads* and the accelerometer. Figure 7.5 shows a model of the patch. This simplified illustration helps to understand the quite complex patch arrangement. The serial *decoder object*

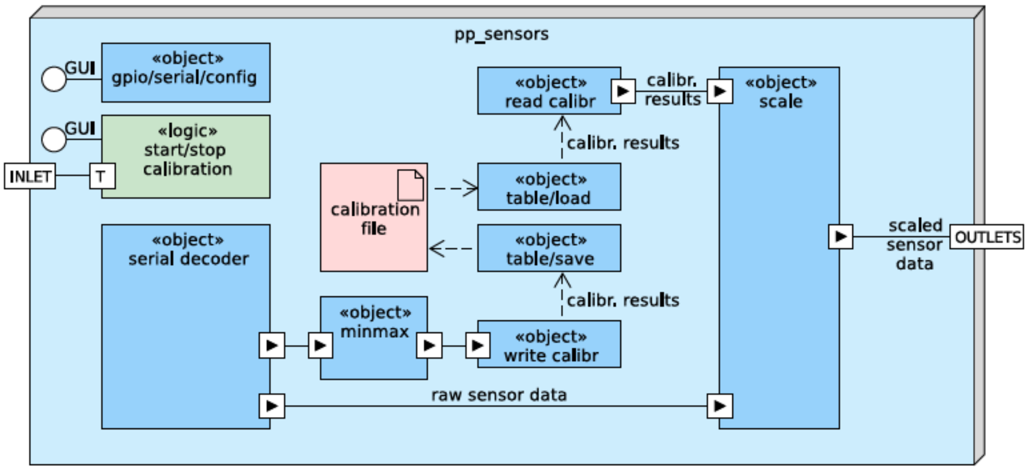


Figure 7.5.: Model of the pp_sensors subpatch (UML composite structure) with Axoloti objects (dark blue), logic blocks (green), GUI elements (circles), inlets/outlets of the patch and artefacts (red). Wires represent patcher connections, while dotted wires are internal object references.

in the bottom left decodes the serial messages from the PSoC 4 and outputs the received sensor data. In order to be able to receive serial data, the serial interface has to be initialised with the *gpio/serial/config* object.

Since the received sensor data does not fit the Axoloti parameter range, it has to be mapped to this range first. This is referred to as calibration and is done with the help of the two objects *minmax 9 i* and *scale i*. The minmax object determines the minimum and maximum sensor values and passes the results to the scale object, which maps the data to the desired range based on these values. Therefore, when in calibration mode, one has to touch each capacitive sensor once and tilt the hand piece left/right and front/back in order to obtain the maximum and minimum sensor values.

Due to the fact that the parameter range of the sensor data does not change without modifications in the hardware or severe temperature shifts, calibration does not need to be redone every time. The results of the calibration process are saved once the calibration is finished and are automatically recalled when the instrument patch is loaded the next time.

For this purpose the objects *write calibr* and *read calibr* were implemented. With the help of these objects together with the *table/load* and *table/save* objects, the calibration results are saved to and loaded from the internal SD card. The objects are linked through an object reference that is given to all of the four objects. The process of calibration requires a specific execution order which is triggered via the GUI or an inlet. This logic includes many different elements which are not shown in the model and is represented by the green rectangle. A detailed description of all implemented axoloti objects is given in subsection 7.2.4 and in appendix A.

pp_controller subpatch

The *pp_ controller* subpatch implements the functionality to switch between different instruments. Due to the fact that the Axoloti can only run single patches, the logic to switch instruments has to be included in every instrument. The ASE offers a handy feature for automatically adding a controller subpatch to every instrument that is uploaded to the SD card. Once a controller object has been implemented, a reference to the object can be set in the preferences and, as a result, every patch that will be uploaded to the SD card with the patch bank tool includes the referenced subpatch. Figure 7.6 shows the arrangement of the pp_controller patch that was implemented for the PP$_{\mathrm{EMB}}$.

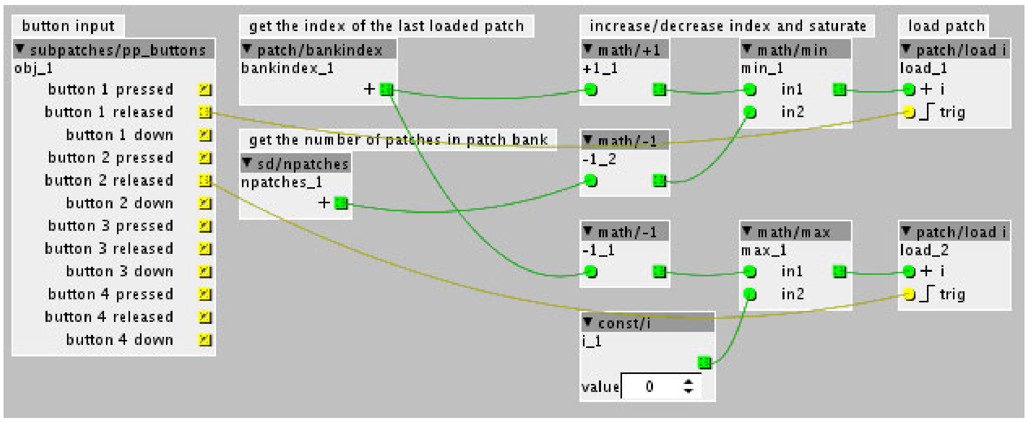


Figure 7.6.: Implementation of the pp_controller subpatch.

The *patch/bankindex* object is used to get the index of the last loaded patch. The index of an instrument depends on the order in which the instruments appear in the axoloti patch bank. To trigger the instrument change, the *pp_ buttons* object is used. Instead of the pp_buttons subpatch one could also use the pp_rotary subpatch to change instruments via one of the two rotary encoders. This complies with the concept of keeping the instrument implementation flexible and easy to modify.

Once one of the defined buttons is released, the pp_controller subpatch loads a new patch with the given index via the *patch/load i* object. The index is calculated by incrementing or decrementing the current instrument index. Furthermore the logic prevents loading patches with indices that don't exist. Therefore the *sd/n-patches* was implemented, which reads the number of patches in the patch bank.

pp_buttons subpatch

The *pp_ buttons* subpatch provides information about the state of the four push buttons. To do this, it initialises the digital input pins (sets the pin number and pin mode) and *debounces* the sampled button signals. Three boolean outputs exist for each of the four buttons: *button pressed*, *button released* and *button down*. While the first two outputs provide trigger signals on momentary changes, the third one indicates the current state of the button. Figure 7.7 shows an excerpt of the

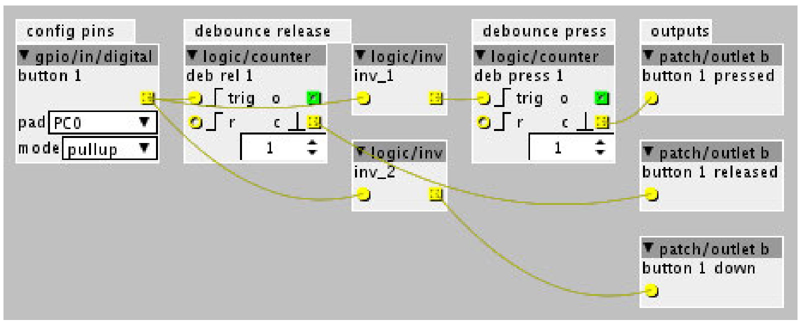


Figure 7.7.: Excerpt from the pp_buttons subpatch.

pp_buttons subpatch with the elements that are used for one button. The input pin, to which the button is connected, is configured with the *gpio/in/digital* object. In order to provide single trigger pulses on button state changes the *logic/counter* object is used for *debouncing* the signal. This logic is applied twice, one time with the unmodified signal for the release of the button and another time with the inverted signal for detecting when the button is pressed. The inverted signal is also used to determine the current state of the button.

pp_rotary subpatch

The *pp_rotary* subpatch handles the configuration and decoding of the two rotary encoders. The configuration is done the same way as within the pp_buttons subpatch except that this time two inputs are sampled per encoder, since the decoding of the rotary encoders is based on two signals. Figure 7.8 shows an excerpt of the subpatch including the logic for debouncing and decoding one rotary encoder.

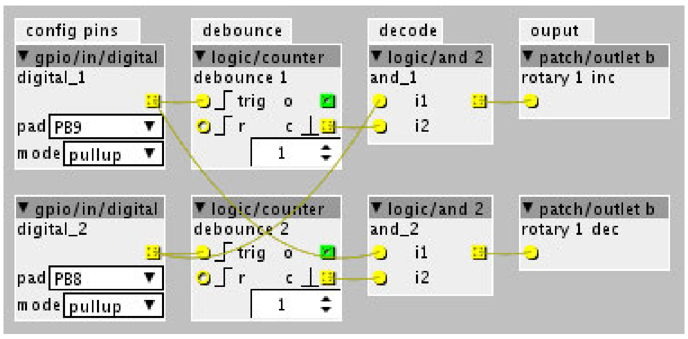


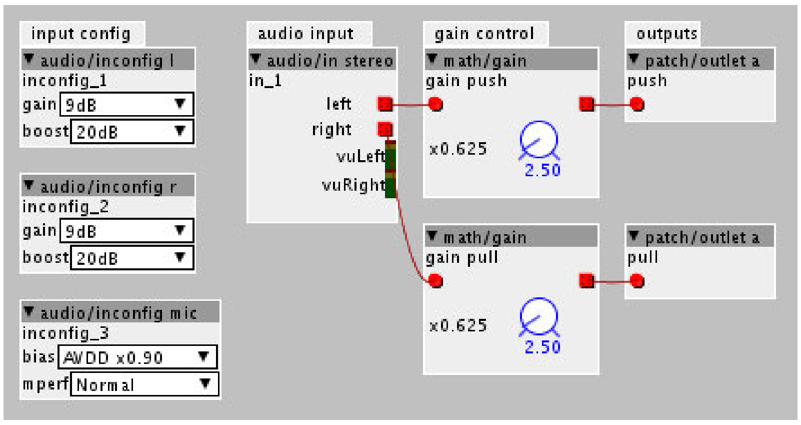
Figure 7.8.: Excerpt from the pp_rotary subpatch.

The decoding is done with the *logic/and 2* object which performs a logic *AND* operation on the debounced signal A from one of the pins and the unmodified signal B of the second pin. When the encoder is rotated one step clockwise the logical signals change from $(A, B) = (0, 1)$ to $(A, B) = (1, 1)$ and the logical AND operation becomes true. The same logic is applied a second time – with the two signals interchanged – to detect a rotation step counter clockwise.

pp_mics subpatch

The *pp_mics* subpatch handles the microphone input configuration and provides control over the microphone pre-amplification. Figure 7.9 shows the underlying patch arrangement.

The input configuration is done via the *audio/inconfig l* and the *audio/inconfig r* objects. Furthermore the *audio/inconfig mic* object is used to set a voltage bias on the inputs (see section 4.4). Gain control from within the parent patch is provided



inconfig_1
gain 9dB
boost 20dB

inconfig_2
gain 9dB
boost 20dB

inconfig_3
bias AVDD x0.90
mperf Normal

x0.625 2.50

x0.625 2.50

Figure 7.9.: Implementation of the pp_mics subpatch.

by the *math/gain* objects. The dials of the objects can be set to be visible on the parent patch. This is indicated by their blue colour.

7.2.4. Objects

This subsection lays down the implementation of the most important axoloti objects that have been coded to provide the required framework for the PP$_{\text{EMB}}$ instrument patches. A complete overview of all implemented objects is given in appendix A.

serial decoder object

The implementation of the *serial decoder* object was one of the most challenging steps in the development of the software framework. Therefore a detailed insight will be given, including all the crucial steps that had to be taken to render the communication between the axoloti and the PSoC 4 possible. To start with, Figure 7.10 shows an activity digram (UML), which models the program flow that implements the serial communication protocol in the serial decoder object.

The central element in the program flow is the *read serial* loop which is implemented as a thread that listens to events which are thrown by the serial interface. It is important to mention that contrary to most other Axoloti objects, for the serial decoding a thread has to be invoked which runs separately from the control rate code and with higher priority. Hence, it is guaranteed that the receive buffer of the serial interface does not overrun at higher baud rates. Since the receive buffer is set

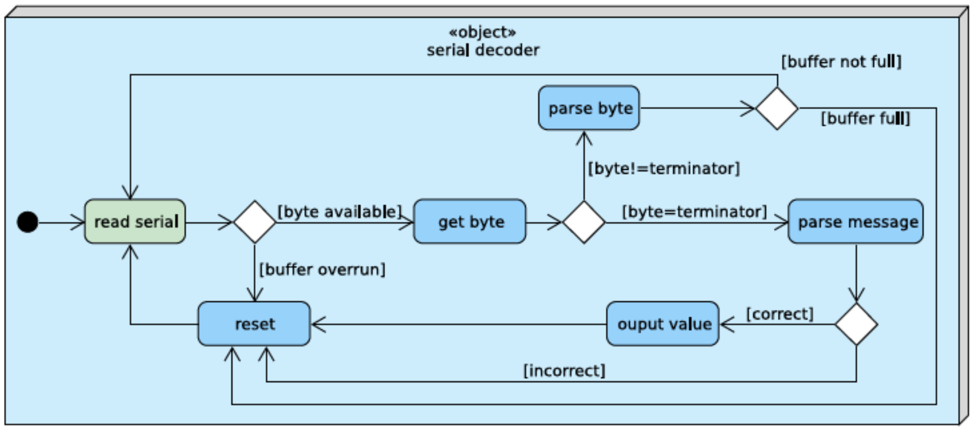


Figure 7.10.: UML activity diagram for the serial decoder object. Rounded rectangles represent actions while diamonds represent decisions and can lead to two or more control branches based on the condition (text in square brackets). The black circle is the entry point of the program.

to only 32 bytes in the Axoloti firmware, the reading of the bytes has to happen fast enough and without interrupts, otherwise bytes will be lost and the message can not be parsed.

The following code shows the registration of the event listener for events that are invoked by the serial driver *SD2*:

```
EventListener s2EventListener;
chEvtRegisterMask((EventSource *)chnGetEventSource(&SD2), &s2EventListener,
 ↪   EVENT_MASK(1));
```

After the event has been registered, a loop is called which waits until an event is thrown:

```
while (!chThdShouldTerminate()) {
    chEvtWaitOneTimeout(EVENT_MASK(1),MS2ST(10));
```

When an event has happened, such as the receiving of a new byte, the flag mask of the event is copied into *flags*, and a bitwise *AND* operation of the flag mask with the *CHN_INPUT_AVAILABLE* identifier will result true. This indicates that one or more bytes are available in the serial receive buffer:

```
    chEvtWaitOneTimeout(EVENT_MASK(1),MS2ST(10));
    flags = chEvtGetAndClearFlags(&s4EventListener);
    if (flags & CHN_INPUT_AVAILABLE) {
```

If so, the last byte in the receive buffer is read with *sdGet()* which removes the byte from the buffer after calling it:

```
            in_byte = sdGet(&SD2);
```

The rest of the code in the loop checks if the byte was the predefined terminator and if that is the case, the *parse_msg()* function is called and the control flow is reset with the *reset()* function. If not, the *parse_byte()* function is called, which flips the fifth bit of the byte if necessary:

```
            in_byte ^= U_MASK;
```

This is done by a *U_MASK* logical *XOR* operation on the byte and the predefined byte mask (0x20). The parse_msg() function will return true, if the message had the right format and consequently the control value is output with the output_value() function. In both cases the reset() function is called to reset the program to its initial state. Another important step inside the loop is too check if a buffer overrun has happened:

```
        if(flags & SD_OVERRUN_ERROR){
            for(uint8_t i = SERIAL_BUFFERS_SIZE; i != 0; --i)
            {
                sdGet(&SD2);
            }
            overrun_errors++;
            reset();
        }
    }
```

If an overrun happened, all bytes from the receive buffer are read in order to empty the buffer and to be able to receive new bytes. This will set the program to its initial state. If this was not done, the program would get stuck in a buffer overrun state in which bytes are permanently missed and messages cannot be parsed anymore.

The output_value() function casts the control value and writes it to the proper variable, based on the received control identifier:

```
void output_value(){
    int32_t value = (int)control_value; /* cast value */
    switch(control_id){
        case CAP1: cap1_val = value;
        break;
        case CAP2: cap2_val = value;
```

These values are then written to the object outputs in the krate code section of the axoloti object:

```
outlet_cap1 = this->cap1_val;
outlet_cap2 = this->cap2_val;
```

The complete source code of the serial decoder object is in appendix C.1.

neopixel object

The *neopixel* object provides an interface for controlling the NeoPixel LEDs. It has several inputs to define a command and send it to the ATmega328-PU microcontroller. The commands are based on the interface definition in the ATmega328-PU software (see section 7.4) and are packed into encoded messages according to the serial communication protocol described in section 7.1. The implementation of the message encoding is the same as in the PSoC 4 software and is described in the following section.

7.3. PSoC 4 Software

In order to sample the analog accelerometer signals and send the sensor data to the AXOC, the PSoC 4 microcontroller had to be re-programmed. This can be done with the *PSoC Creator* IDE and a programmer/debugger device such as the

MiniProg3 programmer, connected to the debug pins on the mainboard of the PP. A PSoC Creator project includes several different files:

- The *TopDesign.cysch* file, which provides a GUI for adding and modifying predefined objects and making connections between them
- The *<projectname>.cydwr* file, in which the input and output pins of the chip are mapped to the inputs and outputs of the objects and where general settings can be made
- The source code section, in which customised program code can be implemented

The PSoC Creator IDE provides many different library objects that can be added to the TopDesign.cysch GUI for achieving standard tasks like AD/DA-conversion, setting up interfaces et cetera. The source code section adds the possibility to implement customised logic in the C programming language. Before the implementation in the source code section is explained, a short overview of what has been arranged in the TopDesign.cysch file is given. Four different objects are employed to provide the required functionality:

- The *UART* and *EZI2C* objects for setting up the serial communication
- The *CapSenseCSD* object for *sampling* and processing capacitive touch pads
- The *ADC SAR Seq* object for sampling and processing the three analog accelerometer signals

Each of the objects offers plenty of settings that can be made. The configuration of the input and output pins is done in the pp_psoc.cydwr file. The chosen settings can be examined in the project files which are provided on the attached CD.

The reading and sending of the sensor values via encoded serial messages is implemented in the *main.c* file. Figure 7.11 shows an activity diagram (UML), modelling the program flow of the software.

The program is divided into two sections: the setup part and the main loop, which is entered after the setup. The setup part initialises the sampling and processing of

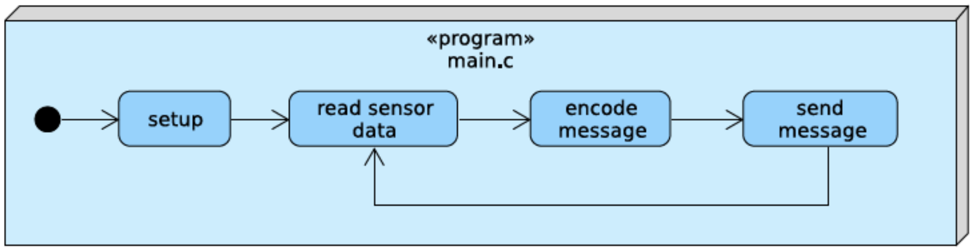


Figure 7.11.: Activity diagram (UML) for reading sensor data and sending encoded messages in the main program of the PSoC 4 software.

the sensor data before the program loop is entered. In every loop cycle the sensor values are read, packed into encoded messages and sent via the serial interface.

The encoding and sending of the message is done by the functions *encode_msg* and *send_msg* (see source code in appendix section C.2). In the following, the implementation of the serial message encoding is laid down. Figure Figure 7.12 shows an activity diagram for encoding serial messages in the *encode_message* function (see source code in appendix C.2).

When the function is called a pointer to the command buffer, which holds the *control identifier* and the *control value* of the last read sensor, is passed to it. The command buffer is an array of single bytes. Since the control values are encoded as unsigned 16 bit integers, each value is stored in two bytes. Theoretically the values could be down-sampled to 8 bit in order to transmit them as single bytes, but this would decrease the resolution from 4096 steps to 256 steps (the sensor data is sampled with 12 bit). Due to the fact that the serial communication happens fast enough (see section 8.2), it is worth sending two bytes without losing accuracy.

The encoding function iterates through each byte in the buffer and checks if a control byte appears in the data (see section 7.1). If so, the fifth bit of the data byte is flipped with a logical *XOR* operation on the byte and a stuffing mask:

```
encoded_byte = byte^S_MASK /* S_MASK = 0x20 or 00010000 */;
```

An escape byte is added to the message, followed by the encoded data byte. The bytes are copied to a message buffer. When the iteration is finished the number of bytes is added to the beginning of the message and the message is finally finished with the terminator byte.

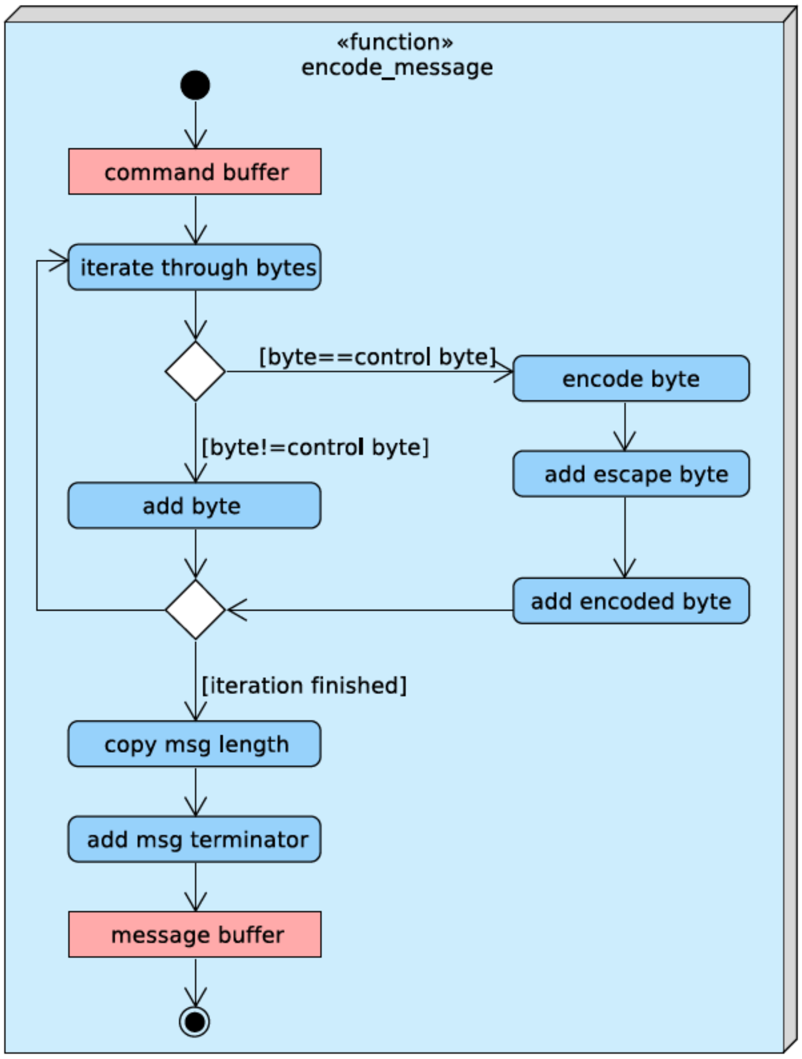


Figure 7.12.: Activity diagram (UML) for encoding serial messages.

7.4. ATmega32 Software

The implemented software for the ATmega328-PU microcontroller provides an interface to control the NeoPixel LEDs. Several functional requirements were set and implemented in order to equip the PP_{EMB} with visual feedback:

- Set the colour of a individual LED
- Set the colour of all LEDs at once
- Provide predefined colours
- Provide animations such as blinking LEDs or automated colour fades

Furthermore, the requirement of maintainability is met by following *object-oriented programming* (OOP) concepts that enhance the reusability and modularity of the code.

The ATmega328-PU chip was programmed with the help of an Arduino microcontroller and the Arduino software library. Additionally, the *PlatformIO IDE*[1] has been used to simplify the development. The Arduino library is written in the C/C++ programming language and consequently is the source code which is attached in appendix C.3.

The software contains two modules, namely *PushPull_SerialParser* and *PushPull_NeoPixel*, and the main program *NeoPixel.ino*, which makes use of these modules. Figure 7.13 shows a class diagram (UML) for the *pp_atmega* software including the implemented classes and their public methods.

The *PushPull_NeoPixel* class inherits from the *Adafruit_NeoPixel* class, provided by the Adafruit NeoPixel library, and adds/overrides methods to provide the specific functionality for the NeoPixel interface as described above. Furthermore, figure 7.14 models the implementation of the software in a sequence diagram (UML).

The implemented logic in the *NeoPixel.ino* program is divided into two parts: the *setup* and the *loop* that is entered after the setup. The setup part creates instances of the *PushPull_SerialParser* and the *PushPull_NeoPixel* classes and passes a reference of the NeoPixel object to the Parser. Furthermore it initialises the NeoPixels by calling the *initPixels()* method of the NeoPixel object.

In the subsequent loop received serial messages are parsed and the containing NeoPixel commands are executed. Since a consistent serial protocol has been developed, decoding serial messages follows the same logic as implemented in the serial decoder object of the Axoloti software (see subsection 7.2.4). The only difference is that the program flow is controlled from within the loop function of the main program here. No separate thread is needed for reading the serial buffer, since no interruptions happen and consequently the loop is executed fast enough in order to read the serial buffer in time.

Reading the serial buffer is done via the method *readSerial()* provided by the

[1]`http://platformio.org/platformio-ide`

SerialParser class. Once a message is received the parsing method *parseMsg()* is called. For checking the format of the received message, the parser object passes the command identifier to the *getCmdLength(id)* method, which returns the expected command length. When the parsing of the message was successful, the command is copied to the command buffer of the main program and the command is executed. Therefore, the colour and current mode of the LEDs are updated with the *setPixelColor(color)* and *setMode(mode)* methods.

Finally the parser is reset to its initial state and the *update()* method of the NeoPixel object is called. The parsing only takes place when a message has been received, whereas the update method is called every loop cycle in order to update the LEDs. The periodic updating is needed to perform the animations.

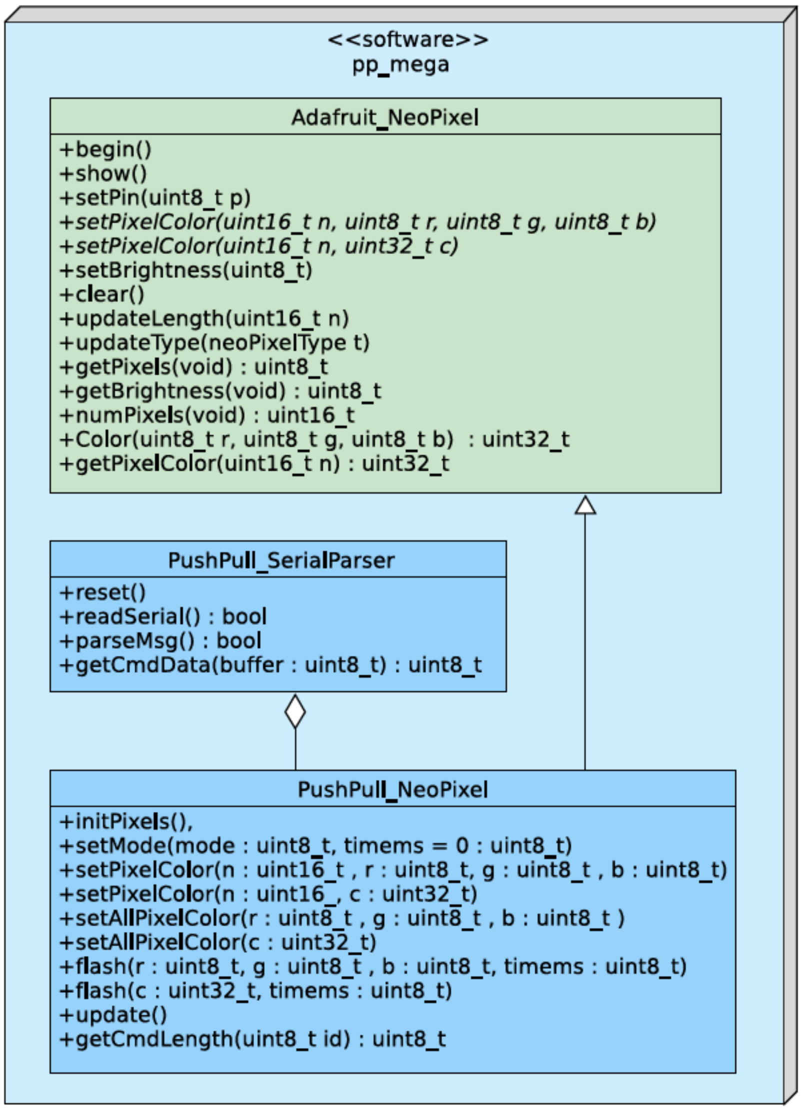

<<software>>
pp_mega
Adafruit_NeoPixel
+begin()
+show()
+setPin(uint8_t p)
+setPixelColor(uint16_t n, uint8_t r, uint8_t g, uint8_t b)
+setPixelColor(uint16_t n, uint32_t c)
+setBrightness(uint8_t)
+clear()
+updateLength(uint16_t n)
+updateType(neoPixelType t)
+getPixels(void) : uint8_t
+getBrightness(void) : uint8_t
+numPixels(void) : uint16_t
+Color(uint8_t r, uint8_t g, uint8_t b) : uint32_t
+getPixelColor(uint16_t n) : uint32_t
PushPull_SerialParser
+reset()
+readSerial() : bool
+parseMsg() : bool
+getCmdData(buffer : uint8_t) : uint8_t
PushPull_NeoPixel
+initPixels(),
+setMode(mode : uint8_t, timems = 0 : uint8_t)
+setPixelColor(n : uint16_t , r : uint8_t, g : uint8_t , b : uint8_t)
+setPixelColor(n : uint16_, c : uint32_t)
+setAllPixelColor(r : uint8_t , g : uint8_t , b : uint8_t)
+setAllPixelColor(c : uint32_t)
+flash(r : uint8_t, g : uint8_t , b : uint8_t, timems : uint8_t)
+flash(c : uint32_t, timems : uint8_t)
+update()
+getCmdLength(uint8_t id) : uint8_t

Figure 7.13.: Class diagram UML of the pp_atmega software with the *PushPull_SerialParser* class and the *PushPull_NeoPixel* class that inherits the methods from the *Adafruit_NeoPixel* and adds the required functionality for the NeoPixel interface.

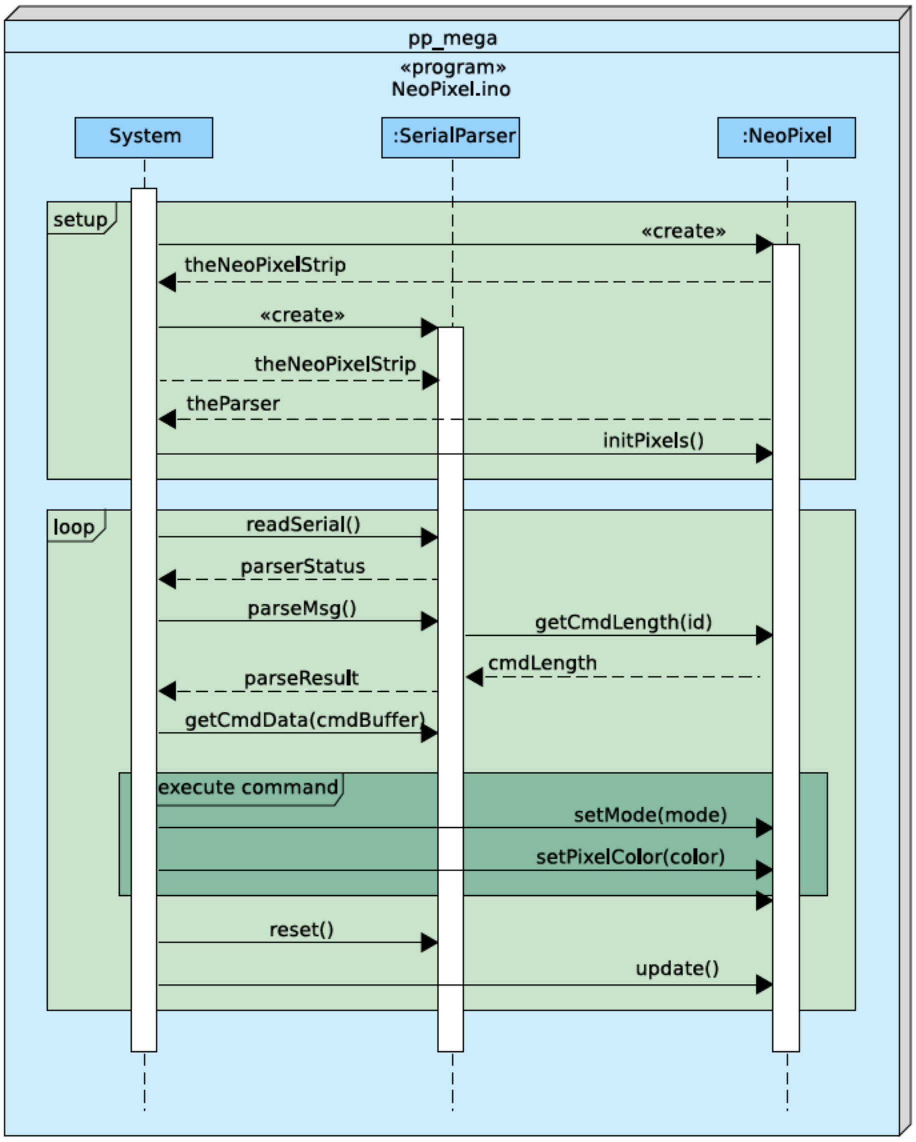


Figure 7.14.: Sequence diagram UML of the pp_mega software. When the system is started, instances of the SerialParser and NeoPixel classes are created in the setup before the program enters a loop where serial messages – containing NeoPixel commands – are parsed in order to control the LEDs.

CHAPTER 8 EVALUATION

In this chapter the result of the hardware and software implementation is evaluated. Different aspects are considered on the basis of the requirements that were set (see chapter 5). In the first section three different instrument patches are presented, which were implemented to test the overall implementation as well as the sound synthesis requirements. An estimation of the overall latency from sensor input to audio output is given in the second section. The following parts lay down performance aspects of the $\mathrm{PP}_{\mathrm{EMB}}$ in terms of processing power and stability and finally the development process itself is discussed.

8.1. Instrument Patches

Figure 8.1 shows simplified composite structures for three instrument patches that were implemented. A short description of each patch will give insight into their diverse sound synthesis and parameter mapping implementations.

First Patch

The most distinctive feature of the first patch (*breath*) is it's use of the microphone signals as a synthesis source. These two audio signals are fed into a network of different filter types. Touching one or more capacitive touch sensors activates individual

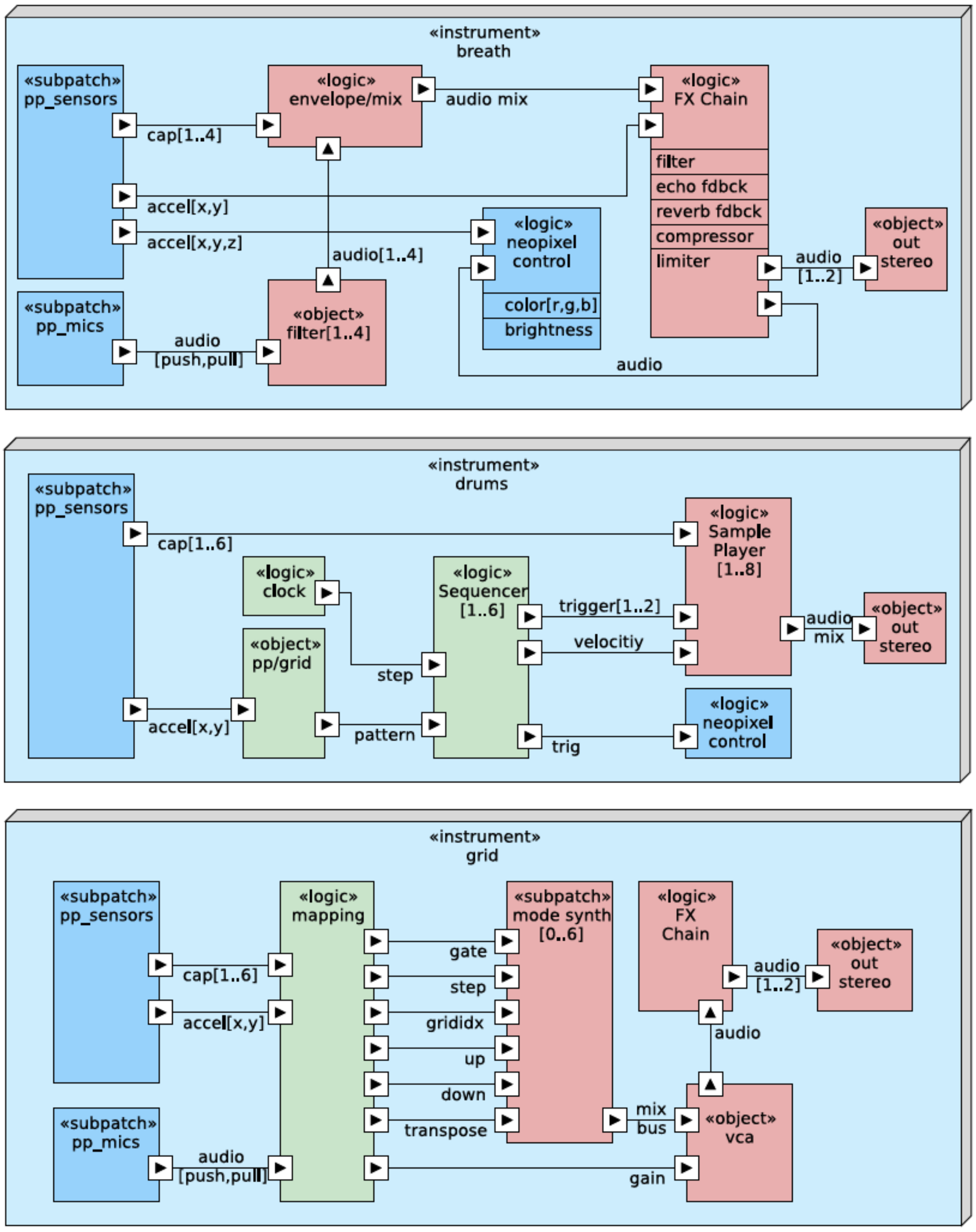


Figure 8.1.: Simplified composite structure for the three implemented instrument patches: *breath*, *drums* and grid. The objects which provide an interface to the PP_{EMB} hardware are coloured dark-blue, control objects/logic green and audio objects/logic red.

filters, whose outputs are mixed and further processed in an effect chain. Tilting the hand piece to the left↔right controls the centre frequency of a bandpass filter in the effect chain. Additionally, echo and delay effects – both provided with feedback control – are used. The feedback parameter of the echo is controlled by tilting the hand piece to the front↔back.

When monitoring the instrument a feedback loop is created, due to the fact that the output of the speakers is captured by the microphones inside the bellows. Although this can be used artistically, the feedback has to be constrained in order to prevent clipping of the audio signal. To accomplish this, the dynamic range of the audio signal is controlled by a compressor and limiter at the end of the effect chain. To provide visual feedback, the envelope of the output signal is mapped to the brightness of the NeoPixel LEDs with the movements of the hand piece controlling their colour.

The patch demonstrates the use of the microphone signals as synthesis source together with several effects and dynamic range control. It is capable of producing various sonic textures like drone sounds, rhythmic pulsations and high pitch feedbacks.

Second Patch

The second patch (*drums*) was developed to explore the abilities of the $\mathrm{PP}_{\mathrm{EMB}}$ as a *sampler* instrument. The design is based on a patch implemented by Dominik Hildebrand Marques Lopes for the $\mathrm{PP}_{\mathrm{SE}}$, a drum sequencer with pre-programmed drum hit patterns that are controlled with the movements of the hand piece in combination with touching the capacitive sensors.

For this patch, the *pp/grid* object was implemented. It determines – based on the x-axis and y-axis acceleration – the position of the hand piece in a virtual orientation grid with nine different fields (see Figure 8.2). The fields result from combinations of tilting the hand piece front↔back and left↔right. The four set acceleration thresholds – two for each acceleration axis – define nine fields with indices $0 - 8$. Given the current x-axis and y-axis acceleration, the *grid* object outputs the index indicating the position of the hand piece in the virtual grid. The thresholds are set as parameters of the grid object.

In the step sequencer section of the patch different note and velocity patterns are programmed, which trigger the playback of different samples. Related samples are grouped into five buses. The samples are stored on the SD card as raw header-

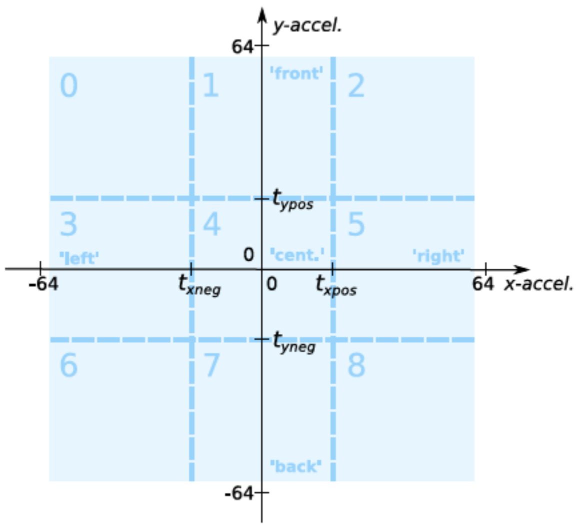


Figure 8.2.: Determination of the hand piece's position in a virtual orientation grid, based on the x-axis and y-axis acceleration. The four set thresholds t_{xneg}, t_{xpos}, t_{yneg} and t_{ypos} define nine fields with indices $0-8$. Given the current x-axis and y-axis acceleration, the *grid* object outputs the index indicating the position of the hand piece in the virtual grid.

less PCM files and are loaded into the *synchronous dynamic random-access memory* (SDRAM) of the axoloti when the patch is started. Touching the capacitive sensors activates individual buses. Which pattern is played for each sample depends on the orientation of the hand piece provided by the grid object. Therefore it is possible to produce a wide variety of different drum rhythms.

Additionally the *pp_neopixel* object is connected to the sequencer and changes the colour of the NeoPixel LEDs on individual drum hits.

Third Patch

The main characteristic of the third patch is its polyphonic pitch control. The absolute pitch of up to five voices is modulated by the orientation of the hand piece. Notes are triggered by five capacitive touch sensors. Each touch pad is associated with one voice. Furthermore, the envelopes of the microphone signals are used to modify the amplitude of the mixed signal. The mapping between pitch and orientation follows a complex system based on modern western modes (Ionian, Dorian, Phrygian, Lydian, Mixolydian, Aeolian and Locrian).

When the hand piece is in its initial position (centre), touching the capacitive sensors triggers *degrees* of the mixolydian, respectively ionian scale. Each sensor is mapped to one degree of the ionian mode on C (I, III, V, VII and VIII), consequently touching the first three sensors results in a C-major chord (C-E-G). When the hand piece is tilted to the front, each degree is raised one step, when tilting to the back the degrees are lowered.

The step modulates the base note either by one semitone or one tone, depending on the mode that is associated with the field in the orientation grid. For example, the two orientations centre/centre and centre/front map the capacitive sensors to the notes from the ionian scale while the fields centre/centre and centre/back map to the mixolydian scale. Consequently hitting the same three sensors and tilting to the front results in a D-minor chord (D-F-A) while tilting back results in a Bb-major chord (Bb-D-F). When adding the fourth voice, major seventh chords are played.

Always two fields represent one of the eight scales, e.g. left/centre and left/front represent the aeolian scale, left/centre and left/back the phrygian scale etcetera. With this system a range of 16 semitones can be controlled, either playing melodies with individual notes or different triad and tetrad chords (e.g. minor, major or diminished).

Conclusion

The three instrument patches showed that the software implementation is suitable for deploying different mapping and sound synthesis strategies. The Axoloti object library provides numerous elements such as oscillators, filters, effects and many more. The parameters of these objects can be mapped to the gestural input of the PP_{EMB} trough the implemented subpatches and objects. Extending the patcher software with your own objects is straightforward. Objects created by the community of Axoloti users further extend the mapping and sound synthesis possibilities of the PP_{EMB}. Thus, all set synthesis requirements could be met.

The first instrument patch demonstrates the use of effects and dynamic range control. The effect objects are easy to use, whereas objects to control the dynamics take some time to get used to; the input parameters of their controls are in the axoloti parameter range ([0, 64[) and consequently one may not know to which values they refer to (e.g. threshold in dB, compression ratio, etc.). Furthermore, gain staging turned out to be challenging. The absence of a proper level meter makes it difficult to avoid digital clipping.

The drum sequencer patch proved that sample based instrument patches are possible, however, the limits of the embedded hardware were reached when arranging this patch. A lot of objects were needed to ensure control over the step sequencers and consequently, the Axoloti ran out of *static random-access memory* (SRAM) at some point. The limited amount of SRAM ($256kb$) made it impossible to add more objects to the patch. When using a lot of objects or subpatches that contain multiple objects the available memory is consumed. Consequently the number of possible sequencer units was limited.

The third instrument patch showed that complex mapping strategies can be achieved and that the PP_{EMB} is able to process polyphonic sound synthesis. This meets a further requirement that has been set.

8.2. Latency Evaluation

As McPherson et al. (2016, p. 1) state:

> "Latency is a fundamental issue affecting digital systems. The delay between a user's action and the corresponding reaction (be it auditory, visual or tactile) can present problems both obvious and subtle."

And although, "Few practitioners of live performance computer music would deny that low latency is essential" (Wessel and Wright, 2001, p. 2), there is no clear answer to the question "how fast is 'fast enough'?"; thresholds may vary for different musical contexts (e.g. percussive instruments require a much lower latency then instruments with continuous gestural input, Lago and Kon (2004)).

Wessel and Wright set the acceptable threshold for the systems audible reaction to a gesture at $10ms$. According to McPherson et al. (2016, p. 1), this value "is perhaps the most common one still used in the community".

Latency in audio systems is usually measured as the time delay between a signal excitation at the audio input and the response of the system at its audio output

(McPherson et al., 2016). The systems input/output latency depends on various elements in the audio chain, such as AD/DA conversion, audio buffering and digital signal processing. In the case of a DMI, one has to consider the latency between sensor input and audio output. McPherson et al. point out that the primary latency factor in usual DMI setups is the communication link between the microcontroller and the computer which runs the audio synthesis. The following subsections discuss the internal audio latency, the audio input to output latency and the serial communication latency of the PP_{EMB} in order to estimate and evaluate the overall sensor input to audio output latency.

Internal Audio Latency

The internal audio latency t_{int} of the PP_{EMB} is derived from the audio buffer size n and *sampling rate* f_s:

$$t_{int} = \frac{n}{f_s} \tag{8.1}$$

The AXOC processes audio in blocks of 16 samples at a sampling rate of $48kHz$, with a resulting audio latency of $t_{int} \approx 0.33ms$. Theoretically the board is capable of sampling at $96kHz$ but the patcher software and the firmware only support $48kHz$ at this time.

Audio Input to Audio Output Latency

Measurements of the audio latency from the analog inputs to the analog outputs of the Axoloti revealed a latency of $t_{io} \approx 2.04ms$ (Taelman, 2016c). Compared to most laptop with audio interface setups this value is outstanding. Furthermore, one should consider that the latency introduced by the performers distance to the speaker is about $3ms$ per meter. Consequently in a real world setup where the distance between performer and nearest monitor is $1.5m$, the sound propagation through the air would add more than twice as much to the overall latency of the setup.

Serial Communication Latency

In order to test the latency of the internal serial communication between the PSoC 4 microcontroller and the AXOC a test patch was implemented with a modified version of the *serial decoder* object. The patch measures the time difference between two subsequent updates of the same sensor data in the patcher. Since the sensor values are transmitted sequentially this is the worst case latency from a change of sensor input to the arrival of the new sensor value at the Axoloti.

The test patch revealed a minimum latency of $5.00ms$ and a maximum latency of $5.33ms$, while the average latency (calculated for 100 values) is $5.20ms$. The theoretical transmission latency is calculated with the given baud rate and the information about the message encoding. With the employed baud rate ($115200Bd$), 11520 data bytes are transmitted per second. The sensor values are encoded with two bytes, consequently it needs a minimum of five bytes to transmit a sensor value via a serial message if no escape bytes are added (see section 7.1). With nine sensor values that are sent in total, a minimum of 45 bytes has to be sent until one sensor value is updated. The resulting theoretical minimum latency is $3.90ms$.

The serial communication latency could be reduced if all sensor values were sent in one message. Consequently a minimum of 21 bytes[1] would be required for one message and the theoretical minimum latency between the arrival of two subsequent messages would be $1.82ms$. But this would not comply with the protocol, which ensures that a serial message contains only one control value. A change of the serial protocol is considered for future versions.

Sensor Input to Audio Output Latency

An exact determination of the overall latency from sensor input to audio output cannot be done with the calculated and measured values from the previous subsections, therefore an estimation is made.

Assuming that the audio output latency of the system is half the measured input to output latency of $2.04ms$, subtracting the internal latency of $0.33ms$, the overall

[1]A maximum of 39 bytes is sent in the very unlikely case that all data bytes are control bytes and have to be escaped and therefore one byte is added for each data byte.

latency is estimated as follows:

$$t = t_{ser} + t_{int} + t_o \tag{8.2}$$

Where $t_{ser} = 5.2ms$ is the average time it takes for a sensor value to be updated, $t_{int} = 0.33ms$ is the calculated latency for the processing of one audio buffer and $t_o = 0.85ms$ is the output latency of the system. With these values, the estimated overall latency is $6.38ms$. This value is not derived from an elaborate test setup measuring the latency from sensor input to audio output and considering jitter as done by McPherson et al. (2016), however, the estimation is considered reasonable and complies with the upper limit of $10ms$ set by Wessel and Wright (2001).

8.3. Performance Evaluation

A minimal patch including all objects and subpatches that are needed to make use of the sensors, microphone signals, buttons and rotary encoders of the PP_{EMB} causes 6-8% DSP load. This leaves enough processing power for the sound synthesis. A test patch revealed that more than 80 sine oscillators can be used without any dropouts or clicks in the sound output. The overall system performance is sufficient to process complex sound synthesis algorithms.

Countless different instrument patches can be implemented and stored on the internal SD card. Changing between these patches happens in less than one second. The patches run stable and without any crashes, even after hours of runtime.

These facts make the PP_{EMB} a promising instrument for real life performances and comply with the requirements set in chapter 5. Furthermore it shows that the employed hardware is suitable for dedicated DMI designs.

8.4. Evaluation of the Development Process

A development up to this stage would not have been possible within the frame of this thesis without the software framework provided by the Axoloti developers. Furthermore the Axoloti community forum has been an important source of information.

This underlines the fact that using open source hardware and software which is used by a broad community is beneficial for DMI development.

The absence of a detailed documentation of the firmware and the patcher was problematic. The only way of learning how to implement my own objects was to look at implementations of the Axoloti library objects. Therefore, especially the implementation of the serial communication for the Axoloti turned out to be challenging. It required an elaborate examination of the underlying real time operation system and its drivers.

The modular character of the developed software components made it possible to iterate through different prototype stages and make changes and refinements in each stage. It allowed to test individual parts of the software and to modify them without having to change all the software parts that were programmed afterwards. The integration of debugging routines was essential for developing and testing of the individual software components.

CHAPTER 9

CONCLUSIONS AND OUTLOOK

Conclusions

Within the framework of this thesis a self-contained version of the PushPull was developed – the PushPull embedded. The embedding of the synthesis unit offers several advantages compared to DMIs which rely on an external computer. This autonomy concerns many aspects such as reliability in live performances and longevity of the instrument.

The starting point of the development was the PushPull student edition. Based on this initial design, several requirements were set for embedding the sound synthesis and overcoming the need for an external computer. Different hardware solutions were considered and compared carefully. The decision was made to choose the Axoloti core, due to the fact that it features out-of-the box audio connectivity and consequently no further extensions were needed. Furthermore the availability of a maintained software framework for the AXOC offered much of the required functionality and enabled the development of dedicated instrument patches.

The software and hardware was implemented in an iterative and incremental development process. Following the concepts of this method, changes and refinements to the implementation could be made at each development stage. This was especially facilitated by the modular character of the developed Axoloti objects and subpatches together with the object oriented design of all software components.

The most challenging part in the development process was the implementation of the internal communication. A serial protocol was developed in order to exchange data between the three microcontrollers inside the PP_{EMB}.

Three diverse instrument patches were implemented for the evaluation of the hardware and the developed software. The patches showed that complex and flexible parameter mappings can be applied and that all set requirements for the sound synthesis unit are met. The limits of the hardware were reached while arranging a drum sequencer patch. Surprisingly the limiting factor was not the DSP load but the small amount of SRAM.

The estimated overall latency, from sensor input to audio output is about $6.5ms$ in average and complies with the upper limit of $10ms$ set by Wessel and Wright (2001). Performance tests showed that the PP_{EMB} offers enough processing resources for the realisation of complex sound synthesis applications.

Taken as a whole, the thesis showed that recent microcontroller technology is well suited for the design and development of innovative DMIs with embedded sound synthesis. It has potential to replace general-purpose computers in DMI designs and consequently enhance the portability and longevity of these instruments.

Outlook

For the future development of the PP_{EMB} the integration of a battery power supply and the Axoloti MIDI interface is planned. Therefore a new box design has to be made. Concerning the software implementation, the latency introduced by the serial communication can be further reduced and the implementation of a proper level meter would simplify digital gain staging. Finally, the instrument patches will be fine tuned in terms of parameter mapping and sonic output. This is one of the essential steps in every DMI design. Last but not least, I am looking forward to perform with the PP_{EMB}, learning and mastering its possibilities and constraints and exposing it in jams and performances with other musicians.

ACRONYMS

3DMIN	design, development and dissemination of new musical instruments.
API	application programming interface.
ASE	axoloti software environment.
AXOC	Axoloti core.
BBB	BeagleBone Black.
DMI	digital musical instrument.
DSP	digital signal processing.
GPIO	general purpose input/output line.
GUI	graphical user interface.
HDLC	high-level data link control.
HDMI	high-definition multimedia interface.
I^2C	inter-integrated circuit.
I^2S	integrated interchip sound.
IDE	integrated development environment.
LED	light-emitting diode.
LiPo	lithium polymer battery.
MIDI	musical instrument digital interface.

OOP	object-oriented programming.
OSC	open sound control.
PCB	printed circuit board.
PP	PushPull.
$\mathbf{PP_{EMB}}$	PushPull embedded.
$\mathbf{PP_{SE}}$	PushPull student edition.
PSoC 4	Programmable System-on-Chip.
PWM	pulse-width modulation.
RPI	Raspberry Pi.
RPI2	Raspberry Pi 2.
RTOS	real-time operating system.
SD	secure digital.
SDRAM	synchronous dynamic random-access memory.
SPI	serial peripheral interface.
SRAM	static random-access memory.
UML	unified modeling language.
XML	extensible markup language.

GLOSSARY

Additive synthesis

Sound synthesis method based on the addition of harmonic oscillations.

ATmega328-PU

Atmel 8-bit AVR microcontroller.

Capacitive touch sensing

Technology that measures the capacitance of conductive surfaces in order to detect touches.

Debouncing

(Software) method that ensures to detect only a single pulse when sampling an electrical contact switch.

FM synthesis

Sound synthesis method based on the modulation of oscillator frequencies.

Integrated Interchip Sound

Serial bus interface standard to communicate audio data between different electronic devices.

Iterative and incremental development

Combines methods of iterative and incremental development (see section 5.3).

Max/MSP

Visual programming language for processing and generating sound and graphics.

Microcontroller

Small electronic device that incorporates a microprocessor and peripheral devices for embedded applications.

Microprocessor

Computer processor which runs computer programs on an integrated circuit. Often used for embedded applications.

NeoPixel

Adafruit's brand for RGB colour pixels based on the WS2812 LED drivers (see section 4.5).

Open Sound Control

Network protocol for the communication between sound and/or multimedia devices.

Parasitic capacitance

Capacitance that unavoidable appears between all parts of an electronic component or circuit. Hence this capacitance is usually unwanted, it is called parasitic.

Pure Data

Open source visual programming language for processing and generating sound and graphics.

RGB

Refers to the RGB colour model in which light of the three primary colours red, green and blue is composed to archive many different colours.

Sampler

Musical instrument with the ability to store, modulate and playback audio 'samples', in order to compose new sounds or rhythms based on the employed audio material.

Sampling

Conversion of a continuous (analog) signal into a discrete-time signal for digital signal processing.

Sampling rate

Measures the frequency, usually in samples per second, of sampling a continuos time signal when converting it into a discrete-time signal.

Sound synthesis

General term for different methods of sound generation with electronic devices.

Subtractive synthesis

Sound synthesis method where a usually overtone rich signal is sculpted by 'subtractive' methods such as filtering or applying envelopes.

SuperCollider

Programming language for real-time audio synthesis and algorithmic composition.

Touch pad

Sensor surface that is used to detect touches.

TRS

3.5 mm stereo connector also known as stereo jack, widely used for analog stereo signals in the audio domain.

XLR

Name of a connector mainly used in professional audio, video, and lighting applications.

X-OSC

Input/output board that can send data gathered from its on-board sensors (gyroscope, accelerometer and magnetometer) and its GPIO pins via OSC messages over WiFi.

BIBLIOGRAPHY

Apex Electrix LLC (2013): *MMA7361 3-Axis Accelerometer Module.* Online. URL `http://eecs.oregonstate.edu/education/docs/accelerometer/MMA7361_module.pdf`. Access 05.07.2016.

Arfib, Daniel; Jean-Michel Couturier; and Loïc Kessous (2005): "Expressiveness and Digital Musical Instrument Design." In: *Journal of New Music Research*, **34**(1), pp. 125–136.

Barr, M. and A. Massa (2006): *Programming Embedded Systems: With C and GNU Development Tools.* O'Reilly Media.

Berdahl, Edgar (2014): "How to Make Embedded Acoustic Instruments." In: *Proceedings of the International Conference on New Interfaces for Musical Expression.* London, United Kingdom: Goldsmiths, University of London, pp. 140–143.

Berdahl, Edgar (2015): *Embedded Musical Instruments with the Raspberry Pi 2.* Online. URL `http://edgarberdahl.com/tools/2015/09/22/Satellite-CCRMA-on-RPi-2.html`. Access 16.05.2016.

Berdahl, Edgar and Wendy Ju (2011): "Satellite CCRMA: A Musical Interaction and Sound Synthesis Platform." In: *Proceedings of the International Conference on New Interfaces for Musical Expression, NIME 2011.* Oslo, Norway, pp. 173–178.

Berdahl, Edgar; Spencer Salazar; and Myles Borins (2013): "Embedded Networking and Hardware-Accelerated Graphics with Satellite CCRMA." In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. Daejeon, Republic of Korea: Graduate School of Culture Technology, KAIST, pp. 325–330.

Birnbaum, David M. (2007): *Musical vibrotactile feedback*. Master's thesis, McGill University, Department of Music Research, Schulich School of Music, Montreal, Canada.

Bongers, Bert (2000): "Physical Interfaces in the Electronic Arts. Interaction Theory and Interfacing Techniques for Real-time Performance." In: Marc Battier and Marcelo M. Wanderley (Eds.) *Trends in Gestural Control of Music*. Paris: IRCAM, Centre Georges Pompidou, pp. 124–164.

Bovermann, Till; et al. (2014): "3DMIN - Challenges and Interventions in Design, Development and Dissemination of New Musical Instruments." In: *Proceedings of the ICMC/SMC*. Athens, Greeces: National and Kapodistrian University of Athens.

Cockburn, Alistair (2008): "Using Both Incremental and Iterative Development." In: *CrossTalk Journal*, **21**(5), pp. 27–30.

Cook, P and C N Leider (2000): "SqueezeVox: A New Controller for Vocal Synthesis Models." In: *In Proc. of the International Computer Music Conference*. pp. 5–8.

Cook, Perry (2001): "Principles for designing computer music controllers." In: *Proceedings of the International Conference on New Interfaces for Musical Expression, NIME 2001*. Seattle, pp. 3–6.

Cypress Semiconductor Corporation (2016): *AN85951 - PSoC® 4 and PSoC Analog Coprocessor CapSense® Design Guide*. Online. URL `http://www.cypress.com/file/46081/download`. Access 4.8.2016.

Davis, Larry (2012): *HDLC Protocol HDLC Protocol*. Online. URL `http://www.interfacebus.com/HDLC_Protocol_Description.html`. Access 01.07.2016.

Fraden, J. (2010): *Handbook of Modern Sensors: Physics, Designs, and Applications*. Springer New York.

Gurevich, Michael and Stephan von Muehlen (2001): "The Accordiatron: A MIDI Controller For Interactive Music." In: *Proceedings of the International Conference on New Interfaces for Musical Expression, NIME 2001*. Seattle, pp. 27–29.

Hinrichsen, Amelie; Dominik Hildebrand MarquesLopes; Sarah-Indriyati Hardjowirogo; and Till Bovermann (2014): "PushPull: Reflections on Building a Musical Istrument Prototype." In: *ICLI 2014 - INTER-FACE: International Conference on Live Interfaces*. Lisbon, Portugal, pp. 196–207.

Hunt, Andy; Marcelo M. Wanderley; and Matthew Paradis (2003): "The Importance of Parameter Mapping in Electronic Instrument Design." In: *Journal of New Music Research*, **32**(4), pp. 429–440.

International Computer Music Association (2016): *ICMC*. Online. URL `http://www.computermusic.org/page/23/`. Access 20.05.16.

Lago, Nelson and Fabio Kon (2004): "The Quest for Low Latency." In: *Proceedings of the 2004 International Computer Music Conference, ICMC 2004*. Miami, Florida, USA.

Lee, Michael A. and David Wessel (1992): "Connectionist Models for Real-Time Control of Synthesis and Compositional Algorithms." In: *Proceedings of the 1992 International Computer Music Conference, ICMC 1992*. San Jose, California, USA.

Lyons, Michael and Sidney Fels (2012): "Advances in New Interfaces for Musical Expression." In: *Procdeedings of the International Conference and Exhibition on Computer Graphics and Interactive Techniques (SIGGRAPH)*. Singapore, pp. 1–159.

Magnusson, T. (2010): "Designing Constraints: Composing and Performing with Digital Musical Systems." In: *Computer Music Journal*, **34**(4), pp. 62–73.

Marshall, Mark T. (2008): *Physical Interface Design for Digital Musical Instruments*. Ph.D. thesis, McGill University, Department of Music Research, Schulich School of Music, Montreal, Canada.

McPherson, Andrew and Victor Zappi (2015): "Exposing the Scaffolding of Digital Instruments with Hardware-Software Feedback Loops." In: Edgar Berdahl and Jesse Allison (Eds.) *Proceedings of the International Conference on New Interfaces for Musical Expression*. Louisiana, USA: Louisiana State University, pp. 162–167.

McPherson, Andrew P.; Robert H. Jack; and Giulio Moro (2016): "Action-Sound Latency: Are Our Tools Fast Enough?" In: *Proceedings of the International Conference on New Interfaces for Musical Expression, NIME 2016*. Brisbane, Australia: Griffith University, pp. 20–25.

Meier, Florian; Marco Fink; and Udo Zölzer (2014): "The JamBerry - A Stand-Alone Device for Networked Music Performance Based on the Raspberry Pi." In: *Linux Audio Conference 2014*. Karlsruhe, Germany: Zentrum für Kunst und Medientechnologie.

Miranda, E.R. and M.M. Wanderley (2006): *New Digital Musical Instruments: Control and Interaction Beyond the Keyboard*. Computer Music and Digital Audio Series. A-R Editions.

NIME Steering Committee (2016): *Archive of NIME Proceedings*. Online. URL `http://www.nime.org/archives/`. Access 20.05.16.

Object Management Group (2015): *Unified Modeling Language*. Online. URL `http://www.omg.org/spec/UML/2.5/PDF/`. Access 20.09.16.

Paine, Garth (2013): "New Musical Instrument Design Considerations." In: *IEEE MultiMedia*, **20**(4), pp. 76–84.

Rovan, Joseph Butch; Marcelo M Wanderley; Shlomo Dubnov; and Philippe Depalle (1997): "Instrumental gestural mapping strategies as expressivity determinants in computer music performance." In: *Kansei, The Technology of Emotion. Proceedings of the AIMI International Workshop*. Genoa: Associazione di Informatica Musicale Italiana, October, pp. 68–73.

Schneider, Martin (2008): "Mikrofone." In: Stefan Weinzierl (Ed.) *Handbuch der Audiotechnik*, chap. 7. Berlin Heidelberg: Springer, pp. 313–413.

Simpson, W. (1994): *RFC1662: PPP in HDLC-like Framing*. Online. URL `https://www.ietf.org/rfc/rfc1662.txt`. Access 05.04.2016.

Smith, Julius and Romain Michon (2015): "Emerging Technologies for Musical Audio Synthesis and Effects." In: *Linux Audio Conference 2015*. Mainz.

Taelman, Johannes (2016a): *Axoloti*. Online. URL `http://www.axoloti.com/`. Access 20.05.16.

Taelman, Johannes (2016b): *Axoloti Patcher*. Online. URL `http://www.axoloti.com/axoloti-patcher/`. Access 17.08.16.

Taelman, Johannes (2016c): *Latency*. Online. URL `http://www.axoloti.com/more-info/latency/`. Access 17.08.16.

Vega, Rafael and Daniel Gómez (2012): "Using the BeagleBoard as hardware to process sound." In: *Linux Audio Conference 2012*. California, USA: Stanford University: Center for Computer Research in Music and Acoustics, pp. 189–193.

Wessel, David and Matthew Wright (2001): "Problems and Prospects for Intimate Musical Control of Computers." In: *ACM Computer-Human Interaction Workshop on New Interfaces for Musical Expression*. Seattle, USA: Association for Computing Machinery's Special Interest Group on Computer-Human Interaction.

Worldsemi Corporation (n.d.): *WS2812: Intelligent control LED integrated light source*. Online. URL `https://cdn-shop.adafruit.com/datasheets/WS2812.pdf`. Access 6.08.16.

Zappi, Victor and Andrew McPherson (2014a): “Design and Use of a Hackable Digital Instrument.” In: *ICLI 2014 - INTER-FACE: International Conference on Live Interfaces*. Lisbon, Portugal, pp. 208–219.

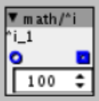
Zappi, Victor and Andrew McPherson (2014b): “Dimensionality and Appropriation in Digital Musical Instrument Design.” In: *Proceedings of the International Conference on New Interfaces for Musical Expression, NIME 2014*. London, United Kingdom, pp. 455–460.

Zappi, Victor and Andrew McPherson (2015): “The D-Box: How to Rethink a Digital Musical Instrument.” In: *Proceedings of the 21st International Symposium on Electronic Art*.

Appendices

APPENDIX A

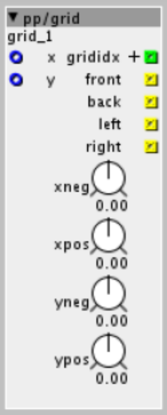
OVERVIEW OF IMPLEMENTED AXOLOTI OBJECTS



***i**

Multiplies input with the given integer.
Used in: drum patch, grid patch, breath patch et cœtera.

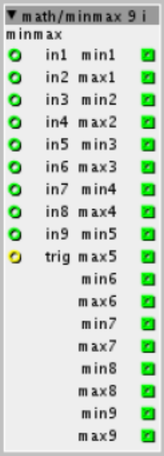
	Name	**Description**
Inlets	in (frac32)	input value
Outlets	out (frac32)	result
Controls	c (int32)	multiplier



grid

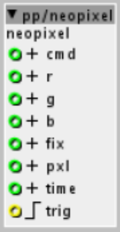
Outputs an index [0 − 8] for the orientation of the hand piece, based on set thresholds for the x and y acceleration.
Used in: drums patch and grid patch.

	Name	**Description**
Inlets	x (frac32)	x acceleration
	y (frac32)	y acceleration
Outlets	grididx (int32.positive)	the index in the virtual grid
	front (bool32)	hand piece tilted to front
	back (bool32)	hand piece tilted to back
	left (bool32)	hand piece tilted to left
	right (bool32)	hand piece tilted to right
Controls	xneg (frac.32.s.map)	x threshold left
	xpos (frac.32.s.map)	x threshold right
	yneg (frac.32.s.map)	x threshold back
	ypos (frac.32.s.map)	x threshold front



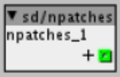
minmax 9 i
Determines the maximum and minimum values of the inputs when triggered.
Used in: pp_sensors.

	Name	Description
Inlets	in1 (int32)	input1
	in2 (int32)	input2
	...	...
Outlets	min1 (int32)	minimum at input 1
	max1 (int32)	maximum at input 1
	min2 (int32)	minimum at input 2
	max2 (int32)	maximum at input 2
	min3 (int32)	minimum at input 3
	max3 (int32)	maximum at input 3
	...	...
Controls	trig (bool32)	trigger, reset



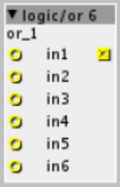
neopixel
Sends encoded serial messages with commands for the NeoPixel LEDs to the ATmega32.
Used in: all instrument patches.

	Name	Description
Inlets	cmd (int32.positive)	command identifier
	r (int32.positive)	the red value (0–127)
	g (int32.positive)	the green value (0–127)
	b (int32.positive)	the blue value (0–127)
	fix (int32.positive)	predefined color identifier
	time (int32.positive)	time value for animations (0–127)
	trig (bool32.rising)	sends the command



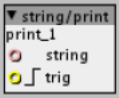
npatches
Reads the number of patches in the patch bank file on the sd card. *Note*: implementation based on an answer by *DrJustice* in the Axoloti community forum
Used in: controller subpatch.

	Name	Description
Outlets	out (int32.positive)	Output



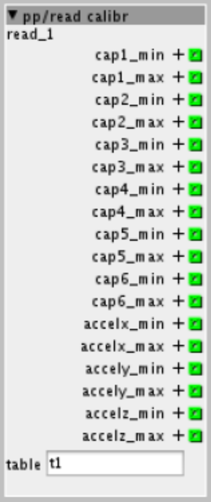
or 6
Logic OR with 6 inputs. *Note*: based on the factory library object *or 2*, which is published under the BSD license.
Used in: breath patch.

	Name	Description
Inlets	in1 (bool32)	input 1
	in2 (bool32)	input 2
	in3 (bool32)	input 3
	in4 (bool32)	input 4
	in5 (bool32)	input 5
	in6 (bool32)	input 6
Outlets	out (bool32)	result



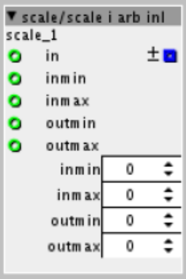
print
Prints the given string to the debug window.
Used in: debugging.

	Name	Description
Inlets	string (charptr32)	string input
	trigger (bool23.rising)	trigger



read calibr
Reads the calibration values from the referenced table.
Used in: pp_sensors subpatch.

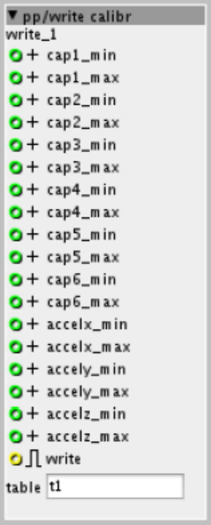
	Name	**Description**
Outlets	cap1_min (int32.postitive)	minimum of capacitive sensor 1
	cap1_max (int32.postitive)	minimum of capacitive sensor 1
	cap2_min (int32.postitive)	minimum of capacitive sensor 2
	cap2_max (int32.postitive)	minimum of capacitive sensor 2
	cap3_min (int32.postitive)	minimum of capacitive sensor 3
	cap3_max (int32.postitive)	minimum of capacitive sensor 3
	cap4_min (int32.postitive)	minimum of capacitive sensor 4
	cap4_max (int32.postitive)	minimum of capacitive sensor 4
	cap5_min (int32.postitive)	minimum of capacitive sensor 5
	cap5_max (int32.postitive)	minimum of capacitive sensor 5
	cap6_min (int32.postitive)	minimum of capacitive sensor 6
	cap6_max (int32.postitive)	minimum of capacitive sensor 6
	accelx_min (int32.postitive)	minimum x-axis acceleration
	accelx_max (int32.postitive)	maximum x-axis acceleration
	accely_min (int32.postitive)	minimum y-axis acceleration
	accely_max (int32.postitive)	maximum y-axis acceleration
	accelz_min (int32.postitive)	minimum z-axis acceleration
	accelz_max (int32.postitive)	maximum z-axis acceleration
Controls	table (objref)	The referenced table



scale i arb inl

Scales the set integer input range to the desired output range.
Used in: pp_sensors subpatch.

	Name	**Description**
Inlets	in (int32)	input
	inmin (int32)	sets the input minimum
	inmax (int32)	sets the input maximum
	outmin (int32)	sets the output minimum
	outmax (int32)	sets the output maximum
Outlets	out (frac32.bipolar)	the scaled input
Controls	inmin (int32)	input minimum
	inmax (int32)	input maximum
	outmin (int32)	output minimum
	outmax (int32)	output maximum



write calibr
Writes the calibration values to the referenced table.
Used in: pp_sensors subpatch.

	Name	**Description**
Inlets	cap1_min (int32.postitive)	minimum of capacitive sensor 1
	cap1_max (int32.postitive)	minimum of capacitive sensor 1
	cap2_min (int32.postitive)	minimum of capacitive sensor 2
	cap2_max (int32.postitive)	minimum of capacitive sensor 2
	cap3_min (int32.postitive)	minimum of capacitive sensor 3
	cap3_max (int32.postitive)	minimum of capacitive sensor 3
	cap4_min (int32.postitive)	minimum of capacitive sensor 4
	cap4_max (int32.postitive)	minimum of capacitive sensor 4
	cap5_min (int32.postitive)	minimum of capacitive sensor 5
	cap5_max (int32.postitive)	minimum of capacitive sensor 5
	cap6_min (int32.postitive)	minimum of capacitive sensor 6
	cap6_max (int32.postitive)	minimum of capacitive sensor 6
	accelx_min (int32.postitive)	minimum x-axis acceleration
	accelx_max (int32.postitive)	maximum x-axis acceleration
	accely_min (int32.postitive)	minimum y-axis acceleration
	accely_max (int32.postitive)	maximum y-axis acceleration
	accelz_min (int32.postitive)	minimum z-axis acceleration
	accelz_max (int32.postitive)	maximum z-axis acceleration
Controls	table (objref)	The referenced table

APPENDIX B

WIRING DIAGRAMS

Figures B.1 and B.2 show wiring diagrams of the PP_{SE} and the PP_{EMB}. The green boxes indicate the different units of the instrument body as introduced in section 4.1. Light-blue boxes are PCBs, while individual electronic components are shown in dark-blue. The white boxes on top of the PCBs are connectors. Not all signal connections are drawn and most lines represent traces on PCBs, except these between the different PCBs. The illustrations do not contain all electronic parts and connections, but only the most important ones in order to indicate changes that were made to the hardware. Furthermore, the pin labels don't necessarily match the labels of the real parts and the arrangement in the overview is set to allow a lucid illustration rather than matching the real physical arrangement of the parts.

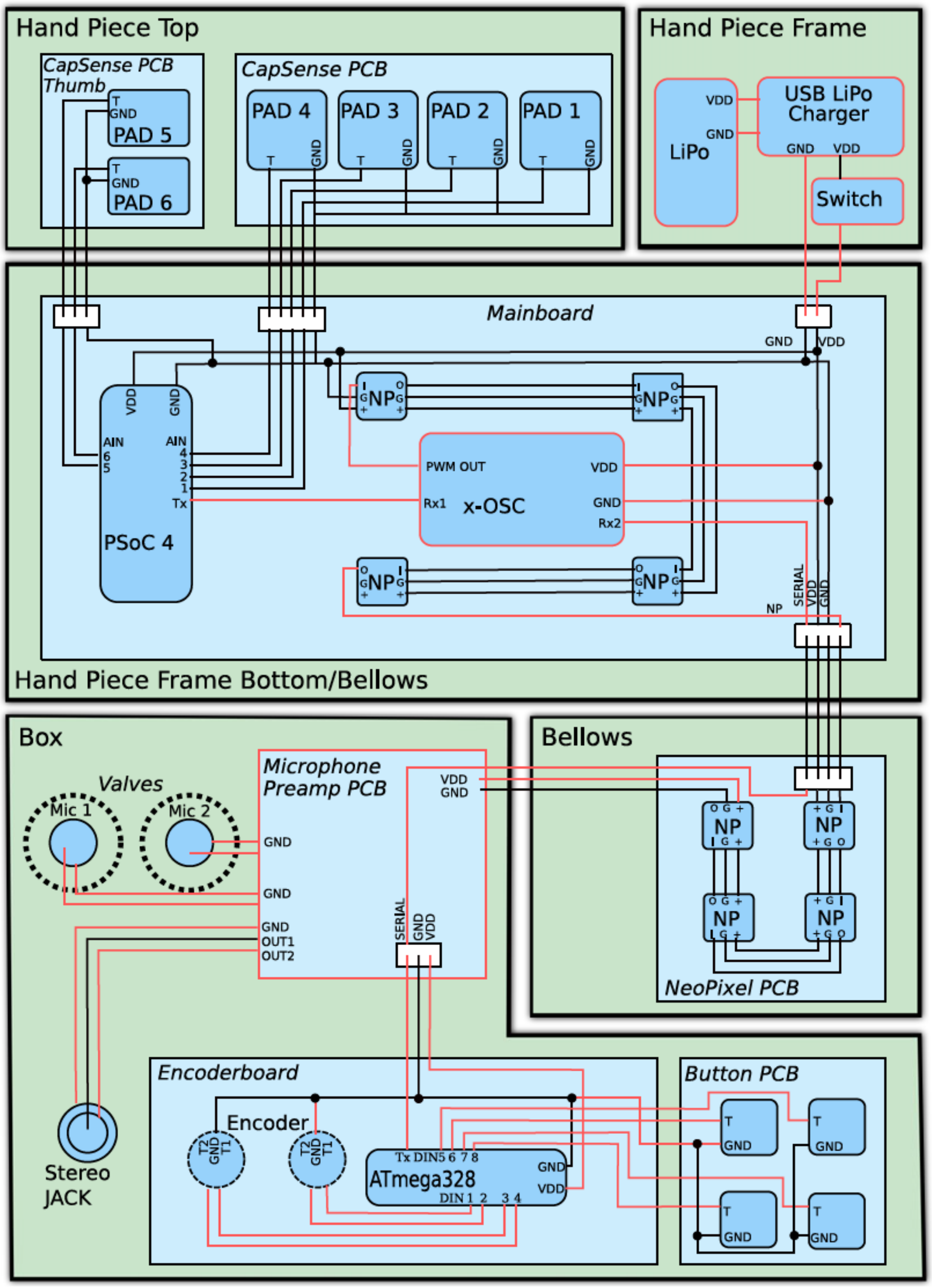


Figure B.1.: Simplified wiring diagram of the PP_{SE}. Changes are highlighted in red.

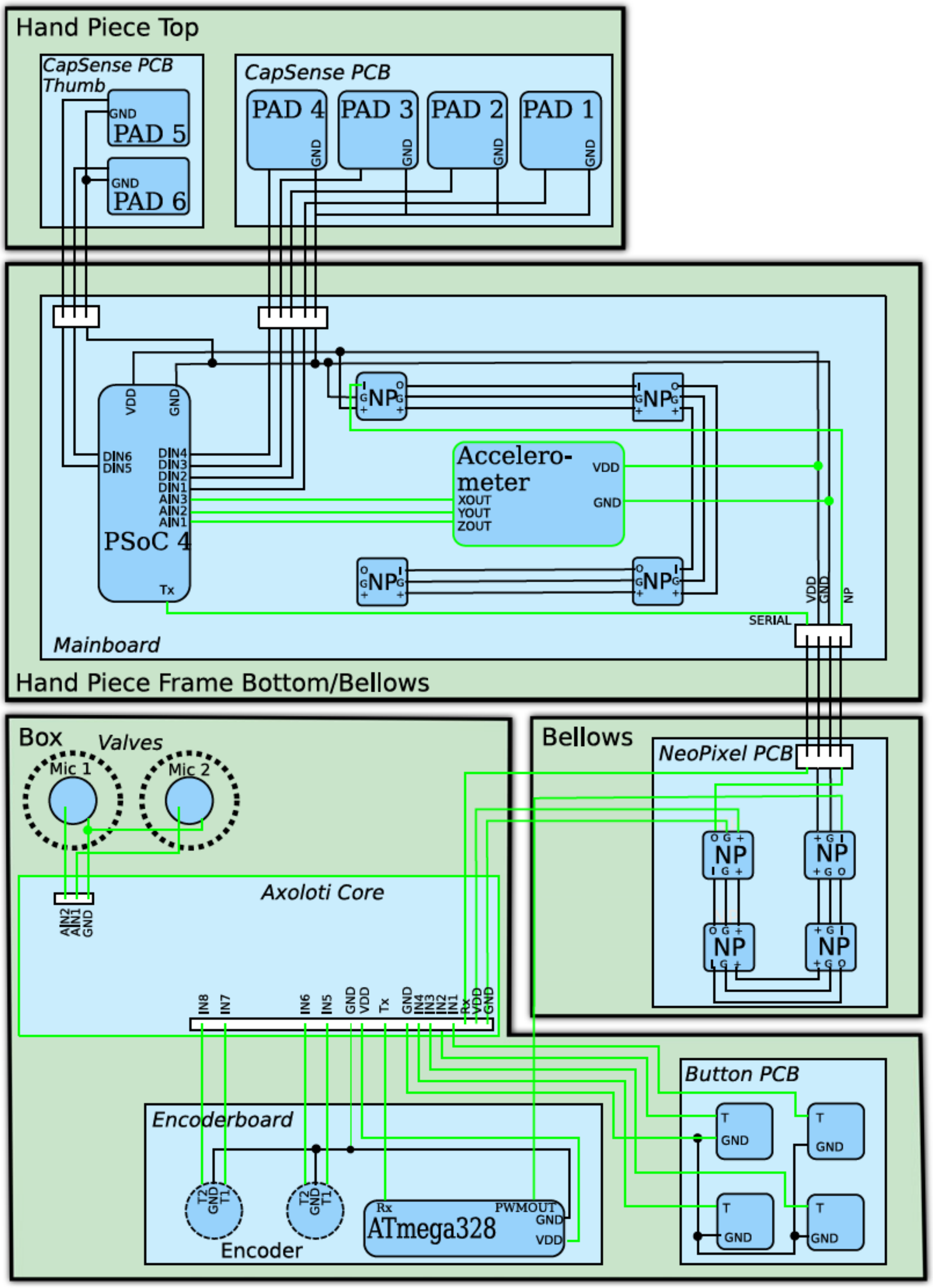


Figure B.2.: Simplified wiring diagram of the PP$_{\text{EMB}}$. Changes that have been made are highlighted in green.

APPENDIX C SOURCE CODE

C.1. Axoloti Source Code

Source Code C.1.: pp_axo/objects/pp/serial decoder.axo

```
<objdefs appVersion="1.0.10">
   <obj.normal id="serial decoder" uuid="de99af16-c883-446a-bf4d-319bda784d26">
      <sDescription>Decodes the serial messages received from the PSOC 4
       ↪  microcontroller and outputs the raw sensor data.</sDescription>
      <author>Pascal Staudt</author>
      <license>GPL</license>
      <inlets/>
      <outlets>
         <int32 name="cap1" description="Sensor data from capacitive sensor 1"/>
         <int32 name="cap2" description="Sensor data from capacitive sensor 2"/>
         <int32 name="cap3" description="Sensor data from capacitive sensor 3"/>
         <int32 name="cap4" description="Sensor data from capacitive sensor 4"/>
         <int32 name="cap5" description="Sensor data from capacitive sensor 5"/>
         <int32 name="cap6" description="Sensor data from capacitive sensor 6"/>
         <int32 name="accelx" description="X-axis acceleration"/>
         <int32 name="accely" description="Y-axis acceleration"/>
         <int32 name="accelz" description="Z-axis acceleration"/>
         <int32 name="perr" description="Debug output. Number of parsing
          ↪  errors"/>
         <int32 name="oerr" description="Debug output. Number of buffer
          ↪  overflows"/>
      </outlets>
      <displays/>
      <params/>
      <attribs/>
      <depends>
         <depend>SD2</depend>
      </depends>
      <code.declaration><![CDATA[
/* Define debug and logging macros. Set defines to 1 for debug or logging out-
   put to the Axoloti window. Be careful, too many outputs in a row will crash
   the Axoloti application. When defines are set to 0, tsmart compiler will
```

```c
   remove the debug code */
#define DEBUG 0
#define debug_print(...) do { if (DEBUG) LogTextMessage(__VA_ARGS__); } while
↪  (0)
#define LOG 0
#define log_print(...) do { if (LOG) LogTextMessage(__VA_ARGS__); } while (0)

#define MAX_MSG_LEN 8 /* The maximum length of the incoming message */

/* Define control identifiers */
#define CAP1 1
#define CAP2 2
#define CAP3 3
#define CAP4 4
#define CAP5 5
#define CAP6 6
#define ACCELX 7
#define ACCELY 8
#define ACCELZ 9

const uint8_t ESC_OCTET  = 0x7D; /* Escape byte */
const uint8_t U_MASK     = 0x20; /* Mask for restoring the escaped bytes */
const uint8_t TERMINATOR = 0x7E; /* Message terminator */

/* Define variables to hold the output values */
int32_t cap1_val,cap2_val,cap3_val,cap4_val,cap5_val,cap6_val;
int32_t accelx_val, accely_val, accelz_val;
int32_t parse_errors,overrun_errors;

uint8_t msg_buffer[MAX_MSG_LEN]; /* Message buffer for storing the read bytes
                                    before parsing the msg */
uint8_t msg_length; /* Number of bytes expected to be received (without
                       framing byte) */
uint8_t in_byte; /* The incomming serial byte */
uint8_t control_id; /* The control identifier */
uint16_t control_value; /* Variable to store the control value */

uint8_t received_bytes = 0; /* The number of received bytes */
uint8_t escaped_bytes  = 0; /* The number of escaped bytes */

/* Flags for program flow control */
int esc_flag   = 0; /* Stuff next byte */
int buffer_idx = 0; /* Current position in the message buffer */

/* Definition of the Thread for Reading the bytes from the serial buffer */
msg_t ReadSerial() {
    if (DEBUG) log_print("Debugging on!");
    if (LOG) log_print("Logging on!");
    if (LOG) log_print("Waiting for incoming bytes!");
    reset(); /* Initialize */

    flagsmask_t flags; /* Mask for the status flags of the serial driver */
    /* Define event listener for the serial status */
    EventListener s4EventListener;
    /* Register the event */
    chEvtRegisterMask((EventSource *)chnGetEventSource(&SD2), &s4EventListener,
    EVENT_MASK(1));

    debug_print("Started serial read thread");

    /* Loop for checking the serial buffer status and read pending bytes */
    while (!chThdShouldTerminate())
    {
        /* Wait for an event */
        chEvtWaitOneTimeout(EVENT_MASK(1),MS2ST(10));
        flags = chEvtGetAndClearFlags(&s4EventListener);

        /* Check if bytes have been received */
        if (flags & CHN_INPUT_AVAILABLE) {

            /* Read one byte from the serial receive buffer */
            in_byte = sdGet(&SD2);
            received_bytes++;

            debug_print("Parse Byte: %x", in_byte);

            /* Check if byte was a terminator */
            if (in_byte != TERMINATOR){
                /* Check if bytes have to be escaped and write byte to
                   the message buffer */
                if (parse_byte()) escaped_bytes++;
            } else {
                /* Parse the message, output the value and reset the
                   control flow */
                if (parse_msg()){
                    output_value();
                    reset();
                } else {
                    parse_errors++;
                    debug_print("Buffer: ");
                    for (int i=0; i < MAX_MSG_LEN; i++){
                        debug_print("%x", msg_buffer[i]);
                    }
                    debug_print("last Byte %x: ", in_byte);
                    reset();
                }
            }
        }

        /* Check for serial receive errors */
        if(flags & SD_PARITY_ERROR)   debug_print("SD PARITY ERROR");
        if(flags & SD_FRAMING_ERROR)  debug_print("SD FRAMING ERROR");
        if(flags & SD_NOISE_ERROR)    debug_print("SD NOISE ERROR");
        if(flags & SD_BREAK_DETECTED) debug_print("SD BREAK DETECTED");
        if(flags & SD_OVERRUN_ERROR){
            debug_print("SD OVERRUN ERROR");
            /* If the receive buffer is full, it has to be cleared,
               in order to be able to receive new bytes */
            for(uint8_t i = SERIAL_BUFFERS_SIZE; i != 0; --i)
            {
                sdGet(&SD2);
            }
            overrun_errors++;
            reset();
        }
    }
    chEvtUnregister(chnGetEventSource(&SD2), &s4EventListener);
    chThdExit((msg_t)0);
}

/* Define static helper thread with working area in RAM */
static msg_t ThreadX(void *arg) {
((attr_parent *)arg)->ReadSerial();
}
WORKING_AREA(waThreadX, 512);
Thread *Thd;

/*
* Function parse_byte()
* ---------------------
* Parses in_byte and restores the byte if the escape flag is set.
*
* returns: 1 if byte has been decoded, 0 if not.
*/
int parse_byte(){
    /* Check if byte is an escape flag */
    if (in_byte == ESC_OCTET){
        esc_flag = 1; /* Set the escape flag */
        return 0;
    }

    /* Prevent buffer overflow */
    if (buffer_idx < MAX_MSG_LEN){
        /* If escape byte was received restore the data and reset flag */
        if (esc_flag){
            in_byte ^= U_MASK;
            msg_buffer[buffer_idx++] = in_byte;
            esc_flag = 0;
            return 1;
        }
        /* Finally copy the byte to the message buffer */
        msg_buffer[buffer_idx++] = in_byte;
        return 0;
    } else {
        log_print("Message buffer full! Resetting...");
        reset();
        return 0;
    }
}

/*
* Function parse_msg
* ------------------
* Parses the msg_buffer according to the defined message protocol (see
* definition) If the message has the wrong length incomplete it will be dropped
```

```
*
* returns: 1 if the message could be parsed correctly, 0 if the message had the
* wrong length
*
*/
int parse_msg(){
    /* First check if the message and the payload had the right length */
    msg_length = msg_buffer[0];
    if (received_bytes == msg_length
        && (received_bytes - escaped_bytes - 2 == 3)) {
        /* Copy the control identifier and the control value */
        control_id = msg_buffer[1];
        memcpy(&control_value, &msg_buffer[2], 2);
        return 1;
    } else {
        debug_print("Wrong number of data bytes received: %i", received_bytes);
        debug_print("Expected %x bytes", msg_length);
        debug_print("Bufer idx: %x", buffer_idx);
        debug_print("Escaped %x bytes.", escaped_bytes);
        return 0;
    }
}

/*
* Function reset
* --------------
* Resets the program flow to its initial state
*
*/
void reset(){
    memset(msg_buffer, 0 , sizeof(msg_buffer));
    buffer_idx = 0;
    received_bytes = 0;
    escaped_bytes  = 0;
}

/*
* Function output_value
* ---------------------
* Casts the control value to int32_t and writes it to the output based on the
* control identifier
*
*/
void output_value(){
    debug_print("Control value (HEX) %x", control_value);
    int32_t value = (int)control_value; /* cast value */
    debug_print("Writing value %u to control output %i", value, control_id);
    /* Output value to the output with the matching identifier */
    switch(control_id){
        case CAP1: cap1_val = value;
        break;
        case CAP2: cap2_val = value;
        break;
        case CAP3: cap3_val = value;
        break;
        case CAP4: cap4_val = value;
        break;
        case CAP5: cap5_val = value;
        break;
        case CAP6: cap6_val = value;
        break;
        case ACCELX: accelx_val = value;
        break;
        case ACCELY: accely_val = value;
        break;
        case ACCELZ: accelz_val = value;
        break;
        /* Undefined control */
        default:
        debug_print("Undefined control: %i", control_id);
        break;
    }
}]]></code.declaration>
      <code.init><![CDATA[cap1_val       = 0;
cap2_val       = 0;
cap3_val       = 0;
cap4_val       = 0;
cap5_val       = 0;
cap6_val       = 0;
accelx_val     = 0;
accely_val     = 0;
accelz_val     = 0;
parse_errors   = 0;
overrun_errors = 0;

/* Initialize static ReadSerial Thread with high priority */
Thd = chThdCreateStatic(waThreadX, sizeof(waThreadX),
                    HIGHPRIO+5, ThreadX, (void *)this);]]></code.init>
      <code.dispose><![CDATA[/* Terminate the Thread */
chThdTerminate(Thd);
chThdWait(Thd);]]></code.dispose>
      <code.krate><![CDATA[outlet_cap1 = this->cap1_val;
outlet_cap2 = this->cap2_val;
outlet_cap3 = this->cap3_val;
outlet_cap4 = this->cap4_val;
outlet_cap5 = this->cap5_val;
outlet_cap6 = this->cap6_val;
outlet_accelx = this->accelx_val;
outlet_accely = this->accely_val;
outlet_accelz = this->accelz_val;
outlet_perr = this->parse_errors;
outlet_oerr = this->overrun_errors;]]></code.krate>
   </obj.normal>
</objdefs>
```

Source Code C.2.: pp_axo/objects/pp/neopixel.axo

```
<objdefs appVersion="1.0.10">
   <obj.normal id="neopixel" uuid="9c21e23f-a42e-42e1-ac5a-80b086bbc2de">
      <sDescription>Sends encoded serial messages with commands for the NeoPixel
       ↪  LEDs to the ATmega32.</sDescription>
      <author>Pascal Staudt</author>
      <license>BSD</license>
      <inlets>
         <int32.positive name="cmd" description="command identifier"/>
         <int32.positive name="r" description="red value (0-127)"/>
         <int32.positive name="g" description="green value (0-127)"/>
         <int32.positive name="b" description="blue value (0-127)"/>
         <int32.positive name="fix" description="predefined colors"/>
         <int32.positive name="pxl" description="LED index in strip (0-7)"/>
         <int32.positive name="time" description="time value for animations"/>
         <bool32.rising name="trig" description="trigger"/>
      </inlets>
      <outlets/>
      <displays/>
      <params/>
      <attribs/>
      <depends>
         <depend>SD2</depend>
      </depends>
      <code.declaration><![CDATA[int ntrig; /* Flag to prevent retriggering */

/* Define the NeoPixel Interface. Has to match the implementation of
   PushPull_Neopixel Class in ATmega32 Software */
#define MAX_MSG_LEN 8
#define MAX_CMD_LEN 6

/* Animation mode */
#define NEOM_OFF      0
#define NEOM_STATIC   1
#define NEOM_BLINK    2
#define NEOM_FLASH    3
#define NEOM_RAINBOW  4

/* Command identifiers */
#define NEOCMD_SETOFF     0
#define NEOCMD_SET        1
#define NEOCMD_SETFIX     2
#define NEOCMD_SETALL     3
#define NEOCMD_SETALLFIX  4
#define NEOCMD_SETANIM    5
#define NEOCMD_SETFLASH   6

/* Color indentifiers */
#define NEOC_RED      0
#define NEOC_GREEN    1
#define NEOC_BLUE     2
#define NEOC_MAGENTA  3

/* Message and command buffers */
uint8_t msg_buffer[MAX_MSG_LEN];
uint8_t cmd_buffer[MAX_CMD_LEN];
```

```

const uint8_t TERMINATOR = 0x7E; /* Message terminator */
const uint8_t S_MASK = 0x20; /* Stuffing Mask for flipping 5th bit */
const uint8_t ESCAPE = 0x7D; /* Escape octet */

/**
* Function send_msg
* --------------------
* Sends bytes via the serial interface
*
* ch: Pointer to the data
* length: Number of data bytes
*
*/
void send_msg(uint8_t *msg, uint8_t l){
    if(!chThdShouldTerminate()){
    sdWrite(&SD2, msg, l);
    }
}

/**
* Function encode_msg
* --------------------
* Encodes the message according to the serial communication protocol of
* PushPull embedded.
*
* data: Pointer to the data buffer
* size: Number of bytes in buffer
*
* returns: Resulting message length
*/
uint8_t encode_msg(uint8_t *data, uint8_t size){
    uint8_t msg_length = 1;
    if (size <= MAX_CMD_LEN){
        uint8_t byte;
        int i;
        for (i = 0; i < size; i++){
            byte = *data++;
            if ((byte == TERMINATOR) || (byte == ESCAPE)){
                msg_buffer[msg_length++] = ESCAPE;
                msg_buffer[msg_length++] = byte^S_MASK;
            } else {
                msg_buffer[msg_length++] = byte;
            }
        }
        msg_buffer[msg_length++] = TERMINATOR;
        msg_buffer[0] = msg_length;
        return msg_length;
    } else {
        return 0;
    }
}]]></code.declaration>
      <code.init><![CDATA[ntrig = 1;]]></code.init>
      <code.krate><![CDATA[/* Encode and send message whe triggered */
if ((inlet_trig>0) && !ntrig) {
    uint8_t msg_len = 0;
    uint8_t cmd_len = 0;
    /* Pack and encode message according to NeoPixel interface definition */
    switch (inlet_cmd) {
        case NEOCMD_SETOFF:
        cmd_buffer[cmd_len++] = inlet_cmd;
        break;

        case NEOCMD_SET:
        cmd_buffer[cmd_len++] = inlet_cmd;
        cmd_buffer[cmd_len++] = inlet_pxl;
        cmd_buffer[cmd_len++] = inlet_r;
        cmd_buffer[cmd_len++] = inlet_g;
        cmd_buffer[cmd_len++] = inlet_b;
        cmd_buffer[cmd_len++] = inlet_time;
        break;

        case NEOCMD_SETFIX:
        cmd_buffer[cmd_len++] = inlet_cmd;
        cmd_buffer[cmd_len++] = inlet_pxl;
        cmd_buffer[cmd_len++] = inlet_fix;
        cmd_buffer[cmd_len++] = inlet_time;
        break;

        case NEOCMD_SETALL:
        cmd_buffer[cmd_len++] = inlet_cmd;
        cmd_buffer[cmd_len++] = inlet_r;
        cmd_buffer[cmd_len++] = inlet_g;
        cmd_buffer[cmd_len++] = inlet_b;
        cmd_buffer[cmd_len++] = inlet_time;
        break;

        case NEOCMD_SETALLFIX:
        cmd_buffer[cmd_len++] = inlet_cmd;
        cmd_buffer[cmd_len++] = inlet_fix;
        cmd_buffer[cmd_len++] = inlet_time;
        break;

        case NEOCMD_SETANIM:
        cmd_buffer[cmd_len++] = inlet_cmd;
        cmd_buffer[cmd_len++] = inlet_fix;
        cmd_buffer[cmd_len++] = inlet_time;
        break;

        case NEOCMD_SETFLASH:
        cmd_buffer[cmd_len++] = inlet_cmd;
        cmd_buffer[cmd_len++] = inlet_fix;
        cmd_buffer[cmd_len++] = inlet_time;
        break;

        default:
        /* Undefined */
        break;
    }
    msg_len = encode_msg(cmd_buffer, cmd_len);
    send_msg(msg_buffer, msg_len);
    ntrig=1;
} else if (!(inlet_trig>0)) {
    ntrig=0;
    }
]]></code.krate>
   </obj.normal>
</objdefs>
```

C.2. PSoC 4 Source Code

Source Code C.3.: pp_psoc/main.c

```
#include "defines.h"

uint8 msg_buffer[MAX_MES_LEN]; /* The message buffer */
uint8 cmd_buffer[MAX_CMD_LEN]; /* The command buffer */

int main()
{   CyGlobalIntEnable;
    /* Start SCB UART TX+RX operation */
    UART_1_Start();
    CapSense_Start();
    ACCEL_Start();
    ACCEL_StartConvert();

    CapSense_InitializeAllBaselines();

    /* The main loop */
    for(;;)
    {
        /* Sample the capacitive sensor data */
        CapSense_ScanEnabledWidgets();

        /* Active wait until cap sense reading is finished*/
        while(CapSense_IsBusy() != 0){
            ;
        }

        /* Read and send the cap values */
        uint8 cap_idx;
        for(cap_idx = 0; cap_idx < NUM_CAPS; cap_idx++){
            uint16 cap_val = CapSense_ReadSensorRaw(cap_idx);
            uint8 msg_len;
            cmd_buffer[0] = cap_idx + 1;    /* Add the control ientifier */
            cmd_buffer[1] = cap_val & 0xff; /* Add the first value byte */
            cmd_buffer[2] = cap_val >> 8;   /* Add the second value byte */
            msg_len = encode_msg(cmd_buffer, 3);
            send_msg(msg_buffer, msg_len);
        }

        /* Read and send the accelerometer values */
        uint8 accel_idx;
        for(accel_idx = 0; accel_idx < 3; accel_idx++){
            uint16 accel_val = ACCEL_GetResult16(accel_idx);
            uint8 msg_len;
            cmd_buffer[0] = accel_idx + 7;      /* Add the control ientifier */
            cmd_buffer[1] = accel_val & 0xff;   /* Add the first value byte */
            cmd_buffer[2] = accel_val >> 8;     /* Add the second value byte */
            msg_len = encode_msg(cmd_buffer, 3);
            send_msg(msg_buffer, msg_len);
        }
    }
}

/**
* Function encode_msg
* --------------------
* Encodes the msg. If an terminator or escape byte appears in the data, it
* encodes the data byte and adds an escape byte before the data byte. Adds the
* msg length to the beginning of the msg and a terminator byte at the end.
*
* data: Pointer to the data buffer
* size: Number of bytes in the buffer
*
* returns: The length of the encoded message
*/
uint8 encode_msg(uint8 *data, size_t size){
    uint8 msg_length = 1;
    if (size <= MAX_CMD_LEN){
        uint8 byte;
        uint8 i;
        /* Iterate through the buffer */
        for (i = 0; i < size; i++){
            byte = *data++;
            /* Check if data byte is a control byte */
            if ((byte == TERMINATOR) || (byte == ESCAPE)){
                msg_buffer[msg_length++] = ESCAPE; /* Add escape byte */
                msg_buffer[msg_length++] = byte^S_MASK; /* Flip the fifth bit
                 ↪  */
            } else {
                msg_buffer[msg_length++] = byte;
            }
        }
        msg_buffer[msg_length++] = TERMINATOR; /* Add message terminator */
        msg_buffer[0] = msg_length; /* Add msg length to the beginning */
        return msg_length;
    } else {
        return 0;
    }
}

/**
* Function send_msg
* --------------------
* Sends bytes via the serial interface
*
* data: Pointer to the data buffer
* size: Number of data bytes
*
*/
void send_msg(uint8 *data, size_t size){
    uint8 i;
    for (i=0; i < size; i++){
            UART_1_UartPutChar(*data);
            data++;
    }
}
```

C.3. ATmega32 Source Code

Source Code C.4.: pp_mega/PushPull_NeoPixel.cpp

```
#include "PushPull_NeoPixel.h"

/* Constructor */
PushPull_NeoPixel::PushPull_NeoPixel()
{
  /* Define colors */
  red     = this->Color(255, 0, 0);
  green   = this->Color(0, 255, 0);
  blue    = this->Color(0, 0, 255);
  magenta = this->Color(255, 0, 255);
  ;
}

/* Destructor */
PushPull_NeoPixel::~PushPull_NeoPixel() {
  ;
}

/**
* Function initPixels
* -------------------
* Initializes the NeoPixels and sets the pixel type, the number of LEDs, and
 ↪  the
* pixel brightness to the predefined values
*/
void PushPull_NeoPixel::initPixels(){
  begin();
  updateLength(NUM_LEDS);
  setPin(PIN);
  updateType(NEO_GRB + NEO_KHZ800);
  setBrightness(BRIGHTNESS);
  lastUpdate = millis();
  show();
}

/**
* Function update
* ----------------
* Updates the NeoPixels. Has to be called once per loop cycle, to keep the
* animations running
*
*/
void PushPull_NeoPixel::update()
{
  /* Check if the time since the last update is greater than the update
  interval. If so, call the update function for the current
  pixel mode */
  if((millis() - lastUpdate) > animationInterval)
  {
    lastUpdate = millis();
    switch(activeMode)
    {
      /* Just need to handle modes that need a update */
      case NEOM_BLINK:
      blinkUpdate();
      break;
      case NEOM_RAINBOW:
      rainbowUpdate();
      break;
      case NEOM_FLASH:
      flashUpdate();
      break;
      default:
      /* Do nothing */
      break;
    }
  }
}

/**
* Function blinkUpdate
* ----------------------
* Update function for the rainbow animation
*/
void PushPull_NeoPixel::blinkUpdate() {
  static int toggle = 1;
  switch (toggle) {
    case 1:
    setBrightness(0);
    toggle = 0;
    break;
    case 0:
    setBrightness(255);
    restorePixels(); /* Since setBrightness is destructive, the pixel color has
                        to be restored */
    toggle = 1;
    break;
  }
  show();
}

/**
* Function rainbowUpdate
* ----------------------
* Update function for the rainbow animation
*/
void PushPull_NeoPixel::rainbowUpdate() {
  uint16_t i;
  static uint16_t j=0;
  if(j<256) {
    for(i=0; i<numPixels(); i++) {
      setPixelColor(i, Wheel((i+j) & 255));
    }
    show();
    j++;
  } else {
    j = 0;
  }
}

/**
* Function flashUpdate
* ----------------------
* Update function for the flash animation
*/
void PushPull_NeoPixel::flashUpdate() {
    static bool flash = false;
    if (flash) {
      setBrightness(0);
      flash = false;
      activeMode = NEOM_OFF;
    } else {
      setBrightness(255);
      restorePixels();
      flash = true;
    }
    show();
}

/**
* Function setMode
* ----------------------
* Sets the current pixel mode
*
* mode: the pixel mode
* time: the time paramter in ms for animations
*/
void PushPull_NeoPixel::setMode(uint8_t mode, uint8_t timems) {
  if (mode != NEOM_OFF){
    /* Reset the brightness */
    setBrightness(255);
    restorePixels();
  }
  activeMode = mode;
  animationInterval = (unsigned long)timems*10;
  show();
}

/**
* Function setPixelColor
* -------------------------
* Sets individual pixel to 32bit RGB color
*
* i: index of the pixel in the strip
* c: 32bit RGB color
*
*/
void PushPull_NeoPixel::setPixelColor(uint16_t i, uint32_t c) {
    Adafruit_NeoPixel::setPixelColor(i, c);
    currentColor[i] = c;
}

/**
* Function setPixelColor (overloaded)
* -------------------------
* Sets the color of an individual pixel
*
* i: index of the pixel in the strip
* r: the red value
* g: the green value
* b: the blue value
*
*/
void PushPull_NeoPixel::setPixelColor(uint16_t i, uint8_t r, uint8_t g, uint8_t
 ↪  b) {
    Adafruit_NeoPixel::setPixelColor(i, r, g, b);
    currentColor[i] = this->Color(r, g, b);
}

/**
* Function setAllPixelColor
* -------------------------
* Sets all pixels to 32bit RGB color
*
* c: The 32bit RGB color
*/
void PushPull_NeoPixel::setAllPixelColor(uint32_t c) {
  for(uint16_t i=0; i<numPixels(); i++) {
    setPixelColor(i, c);
    currentColor[i] = c;
  }
  show();
}

/**
* Function setAllPixelColor (overloaded)
* -------------------------
* Set all pixels to color:
*
* r: The red value
* g: The green value
* b: The blue value
*/
void PushPull_NeoPixel::setAllPixelColor(uint8_t r, uint8_t g, uint8_t b) {
  for(uint16_t i=0; i<numPixels(); i++) {
    Adafruit_NeoPixel::setPixelColor(i, r, g, b);
    currentColor[i] = this->Color(r, g, b);
  }
  show();
}

/**
* Function flash
* -------------------------
* Sets all pixels to 32bit color for given time interval:
*
* c: 32bit RGB color
* time: the time intercal in milliseconds
*/
void PushPull_NeoPixel::flash(uint32_t c, uint8_t timems) {
  activeMode = NEOM_FLASH;
  animationInterval = (unsigned long)timems;
  setAllPixelColor(c);
}

/**
* Function flash (overloaded)
* -------------------------
* Sets all pixels to color for given time interval:
*
* r: The red value
* g: The green value
* b: The blue value
* time: the time intercal in milliseconds
*/
void PushPull_NeoPixel::flash(uint8_t r, uint8_t g, uint8_t b, uint8_t timems) {
  setBrightness(255);
  activeMode = NEOM_FLASH;
  animationInterval = timems;
  for(uint16_t i=0; i<numPixels(); i++) {
    setPixelColor(i, r, g, b);
  }
  show();
}

/**
* Function restorePixels
* -------------------------
* Restores the color of all pixels:
*
*/
void PushPull_NeoPixel::restorePixels() {
  for(uint16_t i=0; i<numPixels(); i++) {
    Adafruit_NeoPixel::setPixelColor(i, currentColor[i]);
  }
  show();
}

/**
* Function Wheel
* -------------------------
* Implements a color wheel that returns a color according to its position in
↪  the
* wheel. The colors are a transition from red to green to blue and back to red.
* Taken from Adafruit NeoPixel Library example puplished under the GPL license:
* https://github.com/adafruit/Adafruit_NeoPixel
*
* WheelPos: The wheel postion (0-255)
*
* returns: The color according to the wheel position
*/
uint32_t PushPull_NeoPixel::Wheel(byte WheelPos) {
  WheelPos = 255 - WheelPos;
  if(WheelPos < 85) {
    return Color(255 - WheelPos * 3, 0, WheelPos * 3,0);
  }
  if(WheelPos < 170) {
    WheelPos -= 85;
    return Color(0, WheelPos * 3, 255 - WheelPos * 3,0);
  }
  WheelPos -= 170;
  return Color(WheelPos * 3, 255 - WheelPos * 3, 0,0);
}

/**
* Function getCmdLength
* -------------------------
* Returns the expected command length for the given command identifier
*
* id: the command identifier
* returns: the command length
*
*/
uint8_t PushPull_NeoPixel::getCmdLength(uint8_t id){
  switch (id) {
    case NEOCMD_SETOFF:
    return 1;
    break;
    case NEOCMD_SET:
    return 6;
    break;
    case NEOCMD_SETFIX:
    return 4;
    break;
    case NEOCMD_SETALL:
    return 5;
    break;
    case NEOCMD_SETALLFIX:
    return 3;
    break;
    case NEOCMD_SETANIM:
    return 39;
    break;
    case NEOCMD_SETFLASH:
    return 3;
    break;
    default:
    /* Undefined */
    return 0;
  }
}
```

Source Code C.5.: pp_mega/PushPull_NeoPixel.h

```
#ifndef PUSHPULL_NEOPIXEL_H
#define PUSHPULL_NEOPIXEL_H

#include <Arduino.h>
#include "Adafruit_NeoPixel.h"

#define PIN         9   /* Define the pin for the Neo Pixel data line */
#define NUM_LEDS    8   /* Define the number of Neo Pixels */
#define BRIGHTNESS  255 /* Define brightness of Neo Pixels */

/* NeoPixel Interface definition */

/* Pixel mode */
#define NEOM_OFF      0
#define NEOM_STATIC   1
#define NEOM_BLINK    2
#define NEOM_FLASH    3
#define NEOM_RAINBOW  4

/* Animations */
#define NEOA_RAINBOW 0

/* Command identifiers */
#define NEOCMD_SETOFF     0
#define NEOCMD_SET        1
#define NEOCMD_SETFIX     2
#define NEOCMD_SETALL     3
#define NEOCMD_SETALLFIX  4
#define NEOCMD_SETANIM    5
#define NEOCMD_SETFLASH   6

/* Color indentifiers */
#define NEOC_RED      0
#define NEOC_GREEN    1
#define NEOC_BLUE     2
```

```
#define NEOC_MAGENTA  3

class PushPull_NeoPixel : public Adafruit_NeoPixel{

public:

  PushPull_NeoPixel();
  ~PushPull_NeoPixel();

  void
  initPixels(void),
  setMode(uint8_t mode, uint8_t timems = 0),
  setPixelColor(uint16_t n, uint8_t r, uint8_t g, uint8_t b),
  setPixelColor(uint16_t n, uint32_t c),
  setAllPixelColor(uint8_t r, uint8_t g, uint8_t b),
  setAllPixelColor(uint32_t c),
  flash(uint8_t r, uint8_t g, uint8_t b, uint8_t timems),
  flash(uint32_t c, uint8_t timems),
  update(void);

  uint8_t getCmdLength(uint8_t id);

  /* Colors are predefined when class is initialized */
  uint32_t
  red,
  green,
  blue,
  magenta;

private:

  uint8_t activeMode; /* The current pixel mode */
  unsigned long animationInterval;   /* Milliseconds between two updates */
  unsigned long lastUpdate;          /* Time of last update */

  uint32_t Wheel(byte WheelPos);
  uint32_t currentColor[NUM_LEDS];

  void
  rainbowUpdate(void),
  flashUpdate(void),
  restorePixels(void),
  blinkUpdate(void);
};

#endif /* PUSHPULL_NEOPIXEL */
```

Source Code C.6.: pp_mega/PushPull_SerialParser.cpp

```
#include "PushPull_SerialParser.h"

/**
```

```
* Constructor PushPull_SerialParser
* ---------------------------------
* baudRate: The baudrate the serial interface will be set up with
*/
PushPull_SerialParser::PushPull_SerialParser(uint32_t baudRate,
 ↪  PushPull_NeoPixel *neopixel) :
isParsing(false), baudRate(baudRate), msgBufferIdx(0)
{
  neopixel = neopixel;
  reset();
}
/**
* Destructor
*/
PushPull_SerialParser::~PushPull_SerialParser() {
  if(msgBuffer) free(msgBuffer);
}

/**
* Function initSerial -> code could go to constructor
* ---------------------------------
* Initializes the serial interface
*/
void PushPull_SerialParser::initSerial(void) {
  Serial.begin(baudRate);
}

/**
* Function readSerial
* -------------------
* Reads all pending bytes from the serial receive buffer and parses them into
* the command buffer.
*
* returns: true if message was received completely, false otherwise
*/
bool PushPull_SerialParser::readSerial(){
  /* While bytes are available read them from the reveive buffer */
  while(Serial.available()>0){
    inByte = Serial.read();
    dprint("Received byte: ");
    dprintln(inByte, HEX);
    receivedBytes++;

    /* Check if byte was a terminator */
    if (inByte != TERMINATOR){
      if (parseByte()) escapedBytes++;
    } else {
      return true;
    }
  }
  return false;
}

/**
* Function parseByte
* --------------------
* Parses the received byte. Unstuffs the byte if the escape flag is set and
* copies the byte to the message buffer.
*
* returns: true if byte has been unstuffed, false otherwise
*/
bool PushPull_SerialParser::parseByte() {
  /* Check if byte is an escape flag */
  if (inByte == ESC_OCTET) {
    escFlag = true;
    return false;
  }

  /* Prevent buffer overflow */
  if (msgBufferIdx < MAX_MSG_LENGTH){
    /* If esape flag was received before unstuff the byte and reset the flag */
    if(escFlag){
        inByte ^= U_MASK;
        msgBuffer[msgBufferIdx++] = inByte;
        escFlag = false;
        return true;
    }

  /* Finaly copy the byte to the message buffer */
  msgBuffer[msgBufferIdx++] = inByte;
  return false;
  } else {
    dprintln("Message buffer full! Resetting...");
    reset();
    return false;
  };
}

/**
* Function parseMsg
* --------------------
* Parses the message.
*
* returns: true if message and payload had the right length, false otherwise.
*/
bool PushPull_SerialParser::parseMsg() {
  /* First check if the message had the right length */
  uint8_t msgLength = msgBuffer[0];
  uint8_t payloadLength = (receivedBytes - escapedBytes - 2);
  uint8_t cmdId = msgBuffer[1];
  cmdLength = neopixel->getCmdLength(cmdId);
  if ((receivedBytes == msgLength) && (payloadLength == cmdLength)){
    if (cmdLength <= MAX_CMD_LENGTH) {
    /* Copy the command identifier and the data into the command buffer */
      memcpy(&cmdBuffer[0], &msgBuffer[1], 1);
      memcpy(&cmdBuffer[1], &msgBuffer[2], cmdLength);
      return true;
    } else {
      dprint("Command exeeded maximum command length: ");
      dprintln(payloadLength);
      return false;
    }
  }/* If message had the wrong length return false */
  else {
    dprintln("Wrong message format!");
    dprint("Total received bytes: ");
    dprintln(receivedBytes, DEC);
    dprint("Expected bytes: ");
    dprintln(msgLength, DEC);
    dprint("Payload length: ");
    dprintln(payloadLength, DEC);
    dprint("Expected payload length: ");
    dprintln(cmdLength, DEC);
  return false;
  }
}

/**
* Function reset
* --------------------
* Resets the program flow to its initial state
*
*/
void PushPull_SerialParser::reset(void) {
  memset(msgBuffer, 0, sizeof(msgBuffer));
  memset(cmdBuffer, 0, sizeof(cmdBuffer));
  msgBufferIdx = 0;
  receivedBytes = 0;
  escapedBytes = 0;
}

/**
* Function getCmdData
* --------------------
* Copies the command to the given buffer;
*
* buffer: pointer to the command buffer
* returns: length of the command
*
*/
uint8_t PushPull_SerialParser::getCmdData(uint8_t *buffer){
  memcpy(buffer, cmdBuffer, cmdLength);
  return cmdLength;
}
```

Source Code C.7.: pp_mega/PushPull_SerialParser.h

```
#ifndef PUSHPULL_SERIALPARSER_H
#define PUSHPULL_SERIALPARSER_H

#include <Arduino.h>
#include "PushPull_NeoPixel.h"
```

```

/* Define debugging macros */
#define DEBUG 1 // -> replace with ifdef
#define dprint(...) do { if (DEBUG) Serial.print(__VA_ARGS__); } while (0)
#define dprintln(...) do { if (DEBUG) Serial.println(__VA_ARGS__); } while (0)

#define  MAX_CMD_LENGTH 6 /* The maximum command length */

class PushPull_SerialParser {

public:

  PushPull_SerialParser(uint32_t baudRate, PushPull_NeoPixel *neopixel);
  ~PushPull_SerialParser();

  void
  initSerial(void),
  initPixels(void),
  reset(void);

  bool
  readSerial(),
  parseMsg();

  uint8_t getCmdData(uint8_t *buffer);

private:

  PushPull_NeoPixel *neopixel;

  static const uint8_t // -> better with #define
  MAX_MSG_LENGTH  = 8,  /* The maximum length of the incoming message in
   ↪  bytes*/
  TERMINATOR     = 0x7E, /* The terminator flag */
  ESC_OCTET      = 0x7D, /* The escape octet */
  U_MASK         = 0x20; /* The mask for unstuffing escaped bytes */

  bool
  escFlag,
  isParsing;

  uint8_t
  inByte, /* The incomming serial byte */
  msgBuffer[MAX_MSG_LENGTH], /* Message buffer for incoming message */
  cmdBuffer[MAX_CMD_LENGTH], /* Command buffer for incomming command */
  receivedBytes,             /* The number of received bytes */
  escapedBytes;              /* The number of escaped bytes */

  size_t cmdLength; /* The length of the incoming command in bytes */

  uint32_t const baudRate;

  int msgBufferIdx; /* Current postion in the buffer */

  bool parseByte();
};

#endif /* PUSHPULL_SERIALPARSER */
```

Source Code C.8.: pp_mega/examples/NeoPixel.ino

```
/* Define debugging macros
   If DEBUG is defined, Serial.print and Serial.println is used to debug
   IF not, the smart compiler will remove the debug code */
#define DEBUG 0
#define dprint(...) do { if (DEBUG) Serial.print(__VA_ARGS__); } while (0)
#define dprintln(...) do { if (DEBUG) Serial.println(__VA_ARGS__); } while (0)

#include <Arduino.h>
#include <PushPull_NeoPixel.h>
#include <PushPull_SerialParser.h>

uint8_t cmdBuffer[MAX_CMD_LENGTH]; /* Buffer for NeoPixel commands */
uint8_t cmdLength;    /* Length of the NeoPixel command */
uint32_t baudRate = 115200; /* The baud rate for the serial interface */

/* Instance of the NeoPixel class */
PushPull_NeoPixel neopixel = PushPull_NeoPixel();
/* Instance of the SerialParser class */
PushPull_SerialParser parser = PushPull_SerialParser(baudRate, &neopixel);

/* Init the program */
void setup() {
  /* Initialize the hardware serial interface */
  parser.initSerial();
  dprint("Serial Port is set up with baudrate: ");
  dprintln(baudRate, DEC);

  /* Initialize the NeoPixels */
  neopixel.initPixels();
  /* Set pixel color to red */
  neopixel.setAllPixelColor(neopixel.red);
  delay(1000);
  neopixel.flash(neopixel.green, 100);
  dprintln("NeoPixels ready for commands!");
}

/**
* Program loop
* ------------
* Every loop cycle all received bytes are parsed and if a complete control
* message arrived, the associated NeoPixel command is executed and the NeoPixel
* strip is being updated
*
*/
void loop() {
      if (parser.readSerial()){
        /* Parse the message, execute the command and reset the parser */
        if (parser.parseMsg()){
          cmdLength = parser.getCmdData(cmdBuffer);
          dprint("Received command: (");
          for (int i = 0; i < (int)cmdLength-1; i++){
            dprint(cmdBuffer[i], DEC);
            dprint(", ");
          }
          dprint(cmdBuffer[cmdLength-1], DEC);
          dprintln(")");
          executeCommand();
          parser.reset();
        } else {
          dprintln("Parsing error. Discarding message!");
          parser.reset();
        }
      }
      neopixel.update();
}

/**
* Function executeCommand
* -----------------------
* Execute the received NeoPixel command
*
*/
void executeCommand(){
  dprint("Executing Command with type: ");
  dprintln(cmdBuffer[0], DEC);
  switch (cmdBuffer[0]) {
    case NEOCMD_SETOFF:
      neopixel.setAllPixelColor(0);
      neopixel.setMode(NEOM_OFF);
    break;

    case NEOCMD_SET:
    neopixel.setPixelColor(cmdBuffer[1],  cmdBuffer[2], cmdBuffer[3],
      cmdBuffer[4]);
    if (cmdBuffer[5] == 0){
      neopixel.setMode(NEOM_STATIC);
    } else {
      neopixel.setMode(NEOM_BLINK, cmdBuffer[5]);
    }
    break;

    case NEOCMD_SETFIX:
    switch (cmdBuffer[2]) {
      case NEOC_GREEN:
      neopixel.setPixelColor(cmdBuffer[1], neopixel.green);
      break;
      case NEOC_RED:
      neopixel.setPixelColor(cmdBuffer[1], neopixel.red);
      break;
      case NEOC_BLUE:
      neopixel.setPixelColor(cmdBuffer[1], neopixel.blue);
      break;
      case NEOC_MAGENTA:
      neopixel.setPixelColor(cmdBuffer[1], neopixel.magenta);
      /* Undefined */
      default:
      dprint("Undefined color id: ");
      dprintln(cmdBuffer[2], DEC);
      break;
    }
    if (cmdBuffer[3] == 0){
      neopixel.setMode(NEOM_STATIC);
    } else {
      neopixel.setMode(NEOM_BLINK, cmdBuffer[3]);
    }
    break;

    case NEOCMD_SETALL:
    neopixel.setAllPixelColor(cmdBuffer[1],  cmdBuffer[2], cmdBuffer[3]);
    if (cmdBuffer[4] == 0){
      neopixel.setMode(NEOM_STATIC);
    } else {
      neopixel.setMode(NEOM_BLINK, cmdBuffer[4]);
    }
    break;

    case NEOCMD_SETALLFIX:
    switch (cmdBuffer[1]) {
      case NEOC_GREEN:
      neopixel.setAllPixelColor(neopixel.green);
      break;
      case NEOC_RED:
      neopixel.setAllPixelColor(neopixel.red);
      break;
      case NEOC_BLUE:
      neopixel.setAllPixelColor(neopixel.blue);
      break;
      case NEOC_MAGENTA:
      neopixel.setAllPixelColor(neopixel.magenta);
      /* Undefined */
      default:
      dprint("Undefined color id: ");
      dprintln(cmdBuffer[1], DEC);
      break;
    }
    if (cmdBuffer[2] == 0){
      neopixel.setMode(NEOM_STATIC);
    } else {
      neopixel.setMode(NEOM_BLINK, cmdBuffer[2]);
    }
    break;
    case NEOCMD_SETANIM:
    switch (cmdBuffer[1]) {
      case NEOA_RAINBOW:
      neopixel.setMode(NEOM_RAINBOW, cmdBuffer[2]);
      default:
```

```
      dprint("Undefined Animation: ");
      dprintln(cmdBuffer[1], DEC);
      break;
    }
    break;

    case NEOCMD_SETFLASH:
    switch (cmdBuffer[1]) {
      case NEOC_GREEN:
      neopixel.flash(neopixel.green, cmdBuffer[2]);
      break;
      case NEOC_RED:
      neopixel.flash(neopixel.red, cmdBuffer[2]);
      break;
      case NEOC_BLUE:
      neopixel.flash(neopixel.blue, cmdBuffer[2]);
      break;
      case NEOC_MAGENTA:
      neopixel.flash(neopixel.magenta, cmdBuffer[2]);
      /* Undefined */
      default:
      dprint("Undefined color id: ");
      dprintln(cmdBuffer[1], DEC);
      break;
    }
    break;
    default:
    dprint("Undefined Command: ");
    break;
  }
}
```

APPENDIX D CD CONTENT

Pruned directory tree of the attached CD:

```
/
├─literature ........................................ cited literature and bibtex file
│   └─01-bibliography.bib
├─software
│   ├─pp_axo ........................................ software for the Axoloti
│   │   ├─copyright.txt
│   │   ├─license.txt
│   │   ├─banks
│   │   │   └─instruments.axb
│   │   ├─instrument patches
│   │   │   ├─barebone.axp
│   │   │   ├─breath.axp
│   │   │   ├─drums.axp
│   │   │   └─grid.axp
│   │   ├─objects
│   │   │   ├─...
│   │   │   ├─controller
│   │   │   │   └─pp_controller.axs
│   │   │   └─subpatches
│   │   │       ├─pp_buttons.axs
│   │   │       ├─pp_mics.axs
│   │   │       ├─pp_modesynth.axs
│   │   │       ├─pp_rotary.axs
│   │   │       ├─pp_sensors.axs
│   │   │       └─...
│   │   ├─samples
│   │   └─test patches
│   ├─pp_mega ....................... software for the ATmega32 microcontroller
│   │   ├─copyright.txt
│   │   ├─license.txt
│   │   ├─PushPull_NeoPixel.cpp
│   │   ├─PushPull_NeoPixel.h
│   │   ├─PushPull_SerialParser.cpp
│   │   ├─PushPull_SerialParser.h
│   │   └─examples
│   │       └─NeoPixel
│   │           └─NeoPixel.ino
│   └─pp_psoc ........................ software for the PSoC 4 microcontroller
│       ├─copyright.txt
│       ├─license.txt
│       └─pp_psoc.cydsn
│           ├─defines.h
│           ├─main.c
│           ├─pp_psoc.cydwr
│           ├─pp_psoc.cyprj
│           └─TopDesign
│               └─TopDesign.cysch
└─thesis ................................................ full text pdf of the thesis
```