Live-looping of distributed gesture-to-sound mappings

Mathias Bredholt

Department of Music Research
Input Devices and Music Interaction Laboratory
McGill University
Montreal, Canada

April 2021

An exam submitted to McGill University in partial fulfillment of the requirements for the degree of Master's of Arts in Music Technology.

2021/04/13

Abstract

This thesis presents the development of a live-looping system for gesture-to-sound mappings built on a connectivity infrastructure for wireless embedded musical instruments using a distributed mapping and synchronization ecosystem. Following the recent trend of Internet of Musical Things (IoMusT), I ported my ecosystem to an embedded platform and evaluated in the context of the real-time constraints of music performance such as low latency and low jitter. On top of the infrastructure, I developed a live-looping system through three iterations with example applications: 1) a wireless Digital Musical Instrument (DMI) with live-looping and flexible mapping capabilities, 2) an embedded loop synthesizer, and 3) a software harp synthesizer/looper with a graphical user interface. My final iteration is based on a novel approach to mapping, extrapolating from using Finite and Infinite Impulse Response filters (FIR and IIR) on gestural data to using delay-lines as part of the mapping of DMI's. The system features rhythmic time quantization and a flexible loop manipulation system for creative musical exploration. I release my tools as open-source libraries for building new DMI's, musical interactions, and interactive multimedia installations.

Résumé

Cette thèse présente le développement d'un système de bouclage en direct de mappages entre gestes et sons. Ce système est construit sur une infrastructure de connectivité pour des instruments de musique embarqués sans fil utilisant un écosystème de mappage et de synchronisation distribuée. J'ai porté mon écosystème s'inscrivant dans la tendance récente de l'Internet des objets musicaux (IoMusT) vers une plateforme embarquée et je l'ai évalué avec des métriques de contraintes temps réel de la performance musicale telles que la faible latence et la haute stabilité. Sur base de cette infrastructure, j'ai développé un système de bouclage en direct à travers trois exemples d'applications: 1) un instrument de musique numérique sans fil (DMI) avec des capacités de lecture en boucle et de mappage flexibles, 2) un synthétiseur de boucle intégré et 3) un synthétiseur/boucleur de harpe logiciel avec une interface utilisateur graphique. Ma dernière itération est basée sur une nouvelle approche du mappage, extrapolant l'utilisation de filtres à réponse impulsionnelle finie et infinie (FIR et IIR) sur des données gestuelles par l'utilisation de lignes à retard dans le cadre du mappage des DMI. Le système a une fonctionnalité de quantification rythmique du temps pour la manipulation flexible de boucles pour l'exploration musicale créative. J'ai publié mes outils sous forme de bibliothèques à code source ouvert pour la construction de nouveaux DMI, d'interactions musicales et d'installations multimédia interactives.

Acknowledgements

This project has come to life with the help of many inspiring and loving people around me. I want to send thanks to Joseph Malloch for much-needed help on understanding libmapper, guidance on my project, and helping with the development of MapLooper. Also, many thanks to Florian Goltz for helping with porting Ableton Link. Many thanks to Christian Frisson, who edited the forthcoming paper about MapLooper, contributed to porting libmapper, and pointed me towards exciting projects. Thanks to Romain Michon, who reviewed this thesis. Thanks to Eduardo Meneses for our collaboration on integrating libmapper in the T-Stick and guidance on my project. Thanks to Filipe Calegario, who contributed with examples for the libmapper Arduino library. Many thanks to my supervisor Marcelo M. Wanderley, who guided me and brought me to knowledge about an inspiring world of gestural controllers. Thanks to Simon Littauer for helping with references and ideas for gesture looping. I send a special thanks to Mathias Kirkegaard, who has been an immense support and source of inspiration and my go-to partner when I was lost in frustration. Finally, I want to send thanks to my family and friends, who have brought me much inspiration and care throughout my life.

Contents

List of Figures

List of Tables

List of Acronyms

ADC	Analog to Digital Converter
API	Application Programming Interface
BPM	Beats Per Minute
CV	Control Voltage
DMI	Digital Musical Instrument
DoF	Degrees of Freedom
DSP	Digital Signal Processing
ESP32	32-bit Espressif Systems Processor
FIR	Finite Impulse Response
GCC	GNU Compiler Collection
GUI	Graphical User Interface
MIDI	Musical Instrument Digital Interface
IIR	Infinite Impulse Response
IMU	Inertial Measurement Unit
IoMusT	Internet of Musical Things
NIME	New Interfaces for Musical Expression
NTP	Network Time Protocol
OSC	Open Sound Control
PPQN	Pulses Per Quarter Note
SDK	Software Development Kit
STL	C++ Standard Template Library

Chapter 1

Introduction

1.1 Repetition as an aesthetic

In his paper (Reinecke, 2009), David Reinecke tells how the German music group Kraftwerk, dissatisfied with the lack of repeatable control of the timbre and volume of acoustic drums, started to use music sequencers to trigger drum sounds for their 1978 album, *Die Mensch Maschine*. This music-making method became popularized through various genres within electronic dance music, such as house and techno. These genres are driven by electronic drums triggered by fixed-rate clocks accompanied by short repeated phrases that are either synthesized or sampled (Wright, 2017). The timing accuracy and precision is aesthetically desirable, as Mark Butler writes (Butler, 2014):

> [..] Mechanically precise repetition in this style is not a technologically induced aftereffect but rather a deliberately cultivated aesthetic strategy.

At the San Francisco Tape Music Center, composers Pauline Oliveros and Terry Riley also explored technology-driven repetition in music. In the 1950s, they did pioneering experiments with tape loop techniques and tape delay/feedback systems (Peters, 1996). These systems worked by stringing tape between two tape recorders and feeding the signal from the second machine back to the first, mixing incoming sound with the tape's previously recorded sound. Riley called this

system the *Time Lag Accumulator* and used it in extended solo improvisations on the saxophone to create layers of short loops. Each layer would slowly fade away as the sound was attenuated before being re-recorded onto the tape. Later, digital looping devices reimplemented this concept. Digital memory replaced magnetic tape, and digital loopers are now available in much smaller form factors than magnetic tape recorders. These devices have become popular to form one-human-bands, i.e., a single musician takes on the role of a full band with drums, guitar, and vocals.

1.2 Digital Musical Instruments

In the academic world of music, the concept of Digital Musical Instruments (DMI's) has been studied extensively Holland, Simon et al., 2019. A DMI consists of a gestural interface and a sound generation unit. The gestural interface and sound generator are separate units, related by mapping strategies. The concept of mapping plays a vital role in instrument design. As Hunt et al. demonstrated (A. D. Hunt et al., 2002), different mappings can completely change an instrument's behavior. Mapping can be seen as a function that maps an n-dimensional control space to an m-dimensional sound parameter space. Van Nort et al. reference this perspective as a systems point of view on mapping (van Nort et al., 2014). Rovan et al. (Rovan et al., 1997) categorizes mappings as

- One-to-one, a single control parameter is mapped to a single sound parameter.
- One-to-many (divergent), a single control parameter is mapped to multiple sound parameters.
- Many-to-one (convergent), multiple control parameters are mapped to a single sound parameter.

The mapping may be performed explicitly, where an instrument designer decides which control parameters should be mapped to which sound parameters. It may also be created with a generative mechanism such as a neural network that performs the mapping (Wanderley & Depalle, 2004). A mapping may consist of multiple layers (A. Hunt & Wanderley, 2002), in which processing or cross-coupling of the control parameters is happening.

1.3 Mapping and loop-based music

Mapping has been used in the context of synthesis engines (A. Hunt & Wanderley, 2002), between physical models (Wanderley & Depalle, 2004), or audio effects (Verfaille et al., 2006). In these contexts, mappings are mostly focused on facilitating what Malloch et al. categorize as skill-based performance (Malloch et al., 2006). Skill-based performance is characterized by rapid, coordinated movements in response to continuous signals (Malloch et al., 2006). This type of performance often involves instruments with a high level of mapping *transparency* i.e., the link between a performer's gesture and the resulting sound is clear to both the audience and the performer. As Fels states (Fels et al., 2002):

> The more transparent a mapping is, the more expressive an instrument can be.

For musicians seeking the aesthetics of accurate and precise timing, this performance type requires a high skill level. On the other hand, existing tools for creating loop-based music such as music sequencers, samplers, and loopers offer beginners a low entry fee (Wessel & Wright, 2002). However, the control mapping of these tools is often opaque and challenging for the audience to infer. In this thesis, I will explore mapping in the context of loop-based music performance with the aspirational goal of creating instruments with a low entry fee and high mapping transparency.

1.4 Internet of Musical Things

The Internet of Musical Things (IoMusT) is an emerging research field bridging existing fields such as the internet of things, new interfaces for musical expression, and human-computer interaction (Turchet, Fischione, et al., 2018). An IoMusT ecosystem consists of three core components: 1) Musical Things, 2) Connectivity and 3) Applications and services. The term 'Musical Thing' refers to a computing device on a network capable of sensing, acquiring, processing, actuating, and exchanging data serving a musical purpose (Turchet, Fischione, et al., 2018). The computing device may be in the form of a smart instrument, i.e., an instrument with embedded computational intelligence, wireless connectivity, embedded sound generation and system for feedback to the

player (Turchet et al., 2017), a musical haptic wearable, or any other device serving a musical purpose. Connectivity refers to the wireless communication infrastructure, which should meet music performance constraints such as low-latency, high reliability, and tight synchronization between devices. Applications and services refer to applications that can be built on top of the connectivity infrastructure. The applications may be targeted at both performers and the audience and maybe both interactive or non-interactive. In their article, Turchet et al. present several scenarios this ecosystem could support, such as augmented and immersive concert experiences. The audience can experience multi-modal concerts through devices such as vibrotactile wearables and synthesizers with loudspeakers embedded in clothes. Smart instruments can capture ancillary gestures of performers (Turchet, McPherson, et al., 2018), and these can be mapped in real-time to deliver tactile stimuli to audience members. The opportunities that arise from these technologies are manifold, and this thesis seeks to follow the trend by developing tools suited for IoMusT applications (Vieira et al., 2020) to open for new explorations in collaborative and participatory music interaction.

1.5 Structure of this thesis

This thesis has five chapters:

1. **Introduction**.
2. **Literature review of gestural looping**, where I review several looping tools and list the project's design requirements.
3. **Design overview and connectivity infrastructure**, where I describe the porting of a mapping and synchronization platform for an embedded device.
4. **Implementation of gesture-to-sound live-looping**, where I describe three iterations of implementing a looper application.

5. **Conclusions and future work**, where I discuss the perspectives on the work presented in this thesis.

Parts of this thesis are also presented in the forthcoming paper *MapLooper: Live-looping of distributed gesture-to-sound mappings* (Frisson et al., 2021).

Chapter 2

Literature review of gesture-to-sound live-looping

In this chapter, several looping tools involving gesture-to-sound mappings are reviewed. The tools fall into two main categories: a) audio stream loopers with a mapping interface for controlling loop parameters and b) control data stream loopers where the looper itself can be considered a part of the mapping. The referenced tools exemplify different mapping strategies and gestural interfaces. A set of design requirements based on the review is listed at the end of the chapter.

2.1 Audio stream loopers

Audio stream loopers have become popular in the form of commercial live-looping pedals. These devices usually have an interface of buttons and knobs for controlling recording and playback state, loop length, and volume of loop layers. Loop controls can also be controlled gesturally, giving the performer the possibility to perform with gestures and body movements. Two projects that employ different mapping strategies for gestural control of audio stream live-looping are reviewed in the following. In this section, two audio stream loopers are reviewed, SoundCatcher (Vigliensoni & Wanderley, 2010) and SoundGrasp (Mitchell & Heap, 2011).

2.1.1 SoundCatcher

SoundCatcher, is a live-looping system with a mid-air gestural control interface. The control interface is attached to a microphone stand and consists of ultrasonic sensors that measure the distance to the performer's hands. The distance is mapped to the loop-points, i.e., the loop's length can be controlled by the performer's hands. Two actuators provide vibrotactile feedback for the performer to sense the loop-points' control space (see fig. 2.1). A pedal footswitch controls the recording state. The loop-points can be synchronized with the MIDI clock to facilitate interoperability when performing with other tools and musicians. Additionally, a time-freezing audio effect can be applied to the recorded buffer. SoundCatcher is an example of the usage of an explicit mapping strategy for the control of live-looping.

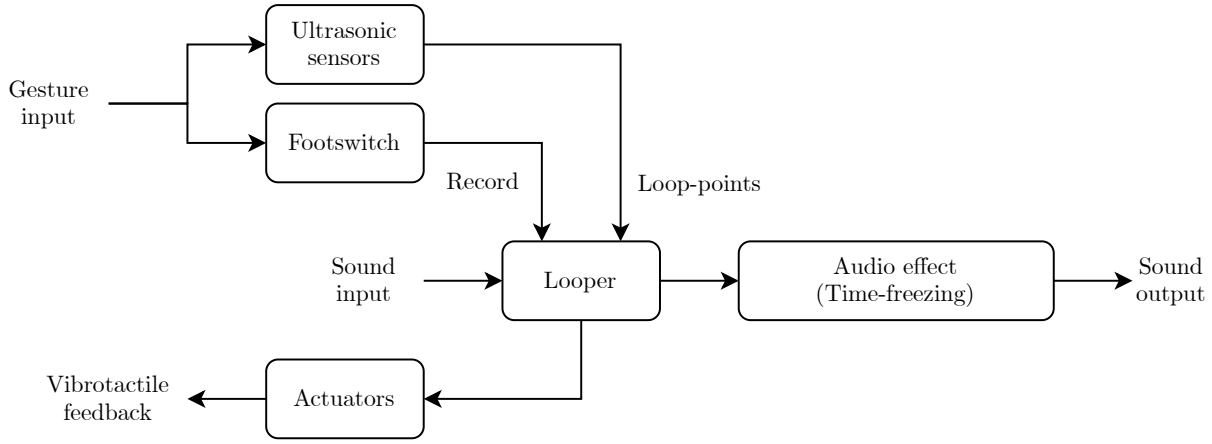

Fig. 2.1 Gesture-to-sound interface of SoundCatcher (Vigliensoni & Wanderley, 2010). The performer is holding the actuators, and the ultrasonic sensors are mounted to a microphone stand.

2.1.2 SoundGrasp

SoundGrasp, is a live-looping system, also with a mid-air gestural control interface. The interface consists of a glove, which controls the recording/playback state and parameters for two audio effects, reverb and echo. A posture identification system is implemented with an artificial neural network,

which classifies the postures into a vocabulary of control commands such as record/play/stop (see fig. 2.2). For controlling audio effects parameters, the sensor data stream is used, similar to SoundCatcher. SoundGrasp is an example of using machine learning as a mapping strategy for the control of live-looping.

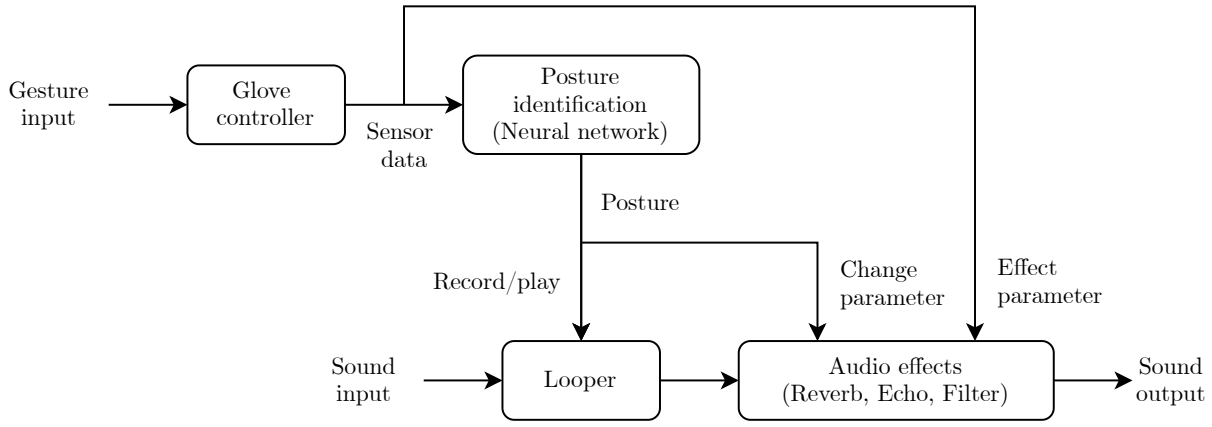

Fig. 2.2 Gesture-to-sound interface of SoundGrasp (Mitchell & Heap, 2011). Gestures are recognized using a neural network. The identified postures are used as commands for controlling the looper. Sensor data is also mapped directly to audio effect parameters.

2.2 Control stream loopers

Looping devices based on control data streams are inserted between a control interface and a sound generator. From a mapping perspective, control stream loopers can therefore be seen as a mapping layer. Like audio stream loopers, control data is recorded into a buffer and played back in a loop. The control data could be in the form of MIDI or Open Sound Control (OSC) messages or analog control voltages (CV). Control stream loopers, being part of the mapping, have the advantage over audio stream loopers, that mappings can be changed post-recording, giving the possibility to route the control data to different synthesis processes. In this section, a number of control stream loopers is reviewed: the open-source project MidiREX (Kvitek, 2014), the commercial product Midilooper (Instruments, 2020), the mobile app Ribn (Petrovic, 2018),

the Eurorack module Tetrapad (Intellijel, 2018), and the virtual-reality system Drile (Berthaut et al., 2010).

2.2.1 MidiREX and Midilooper

MidiREX and *Midilooper* both take their inspiration from digital loop pedals both in appearance and functionality. The devices record incoming MIDI messages into a buffer in either overwrite and overdub modes. MIDI Polyphonic Expression (MPE) (The MIDI Manufacturers Association, 2018) messages are supported on both devices, allowing to use gestural controllers such as the XT Synth (Oliveira da Silveira, 2018) and the Linnstrument (Linn, n.d.) as inputs (see fig. 2.3). The interface on both devices has buttons for switching recording and playback on/off, selecting loop layer, configuring loop length, and selecting MIDI channel. Both devices have options for manipulating the timing and pitch of messages. For timing, incoming MIDI messages can be quantized to a grid. On the Midilooper, the buffer can be time-stretched to double/half its length, and a shuffle function can apply time shifts to achieve a swung feel to the rhythm of the loop. For pitch, recorded MIDI notes can be transposed by discrete semitones. Finally, the Midilooper can modulate MIDI velocity either randomly or using a control voltage input as a modulation source. Random modulation has become an increasingly popular feature of music sequencers as a tool for "humanization" (Rodgers, 2003). Cascone characterizes this trend as the era of "post-digital" music defined by the aesthetics of failure where musicians, as a means of expression, insert audible glitches into their music (Cascone, 2000). On the Midilooper, the random velocity feature, labeled "human velocity", can add dynamic variation to the recorded loops. Both the MidiRex and Midilooper have MIDI clock synchronization, and the Midilooper additionally features analog clock synchronization.

2.2.2 Ribn and Tetrapad

The two other looping tools, *Ribn* and *Tetrapad* work differently as the interface for control input is contained within the system. Both devices have a touch interface (see fig. 2.4) to record horizontal

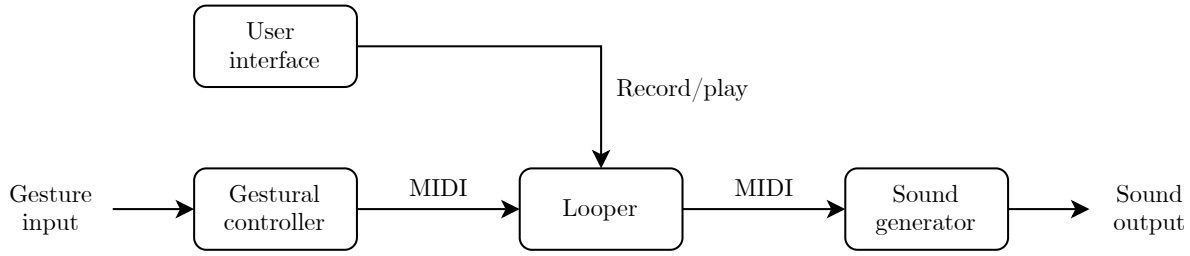

Fig. 2.3 Gesture-to-sound interface of MidiREX by Peter Kvitek (Kvitek, 2014) and Midilooper by Bastl Instruments (Instruments, 2020). The MIDI protocol allows using any MPE-compatible gestural controller.

or vertical gestures. Ribn runs on iOS and uses the touch screen on the mobile devices as its gestural interface. Up to eight sliders can be added to the screen simultaneously, and each slider sends a single MIDI control change message. The recording is started when the slider is touched and ends when the slider is released. The gesture's playback starts immediately after recording, and loop lengths cannot be changed after recording.

Tetrapad is a Eurorack module and has four dedicated touch interfaces that sense both position and pressure, allowing for two-dimensional gesture recordings. Loops can be synchronized externally by an analog clock signal. Tetrapad has eight control voltage outputs that can be patched to any parameter within a eurorack system. With the Tête expander module, recorded sequences can be quantized in both time and value, with the possibility of quantizing control voltage outputs to a selection of musical scales.

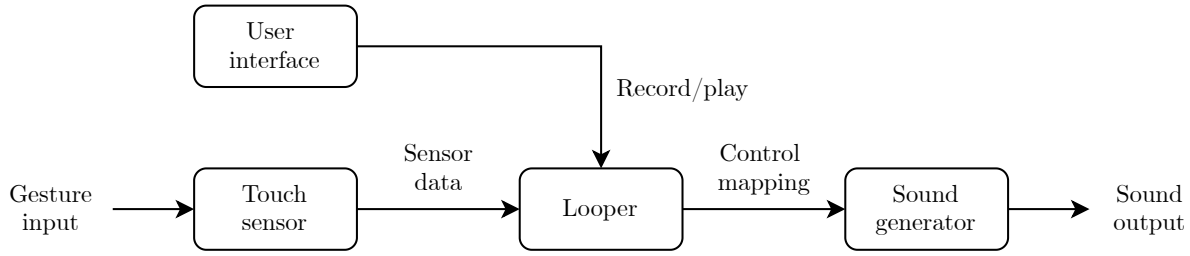

Fig. 2.4 Gesture-to-sound interface of Ribn by Nobjsa Petrovic (Petrovic, 2018) and Tetrapad + Tête by Intellijel (Intellijel, 2018). Touch sensor is embedded in the interface.

2.2.3 Drile

Drile is a virtual reality-based live-looping system. A bi-manual 6-DoF controller is used to create loops and control audio effects in a 3D space. Like SoundCatcher, the controls are explicitly mapped. Unlike the other looping tools, Drile has both audio and control streams. The looping system is built on the concept of hierarchical live-looping, an alternative to traditional live-looping. Hierarchical live-looping is based on a hierarchical tree structure for grouping loop layers, in contrast to traditional live-looping, where loop layers are arranged in a flat structure (see fig. 2.5). This structure has the advantage that hierarchical musical structures can be represented and controlled in useful ways. For instance, a loop node can have children nodes for a bass-loop, a drum-loop, and a keyboard-loop. Triggering the root node causes all children nodes to start playback. Children nodes can be played independently to "solo" a particular node. Multiple trees of nodes can be created to represent different sections of a piece.

2.3 Summary

The reviewed projects share several common properties. A table showing a comparison of the projects can be seen in table 2.1. A few conclusions can be drawn from this comparison. Most of the tools reviewed contain a gestural interface as part of the tool; only MidiRex and Midilooper

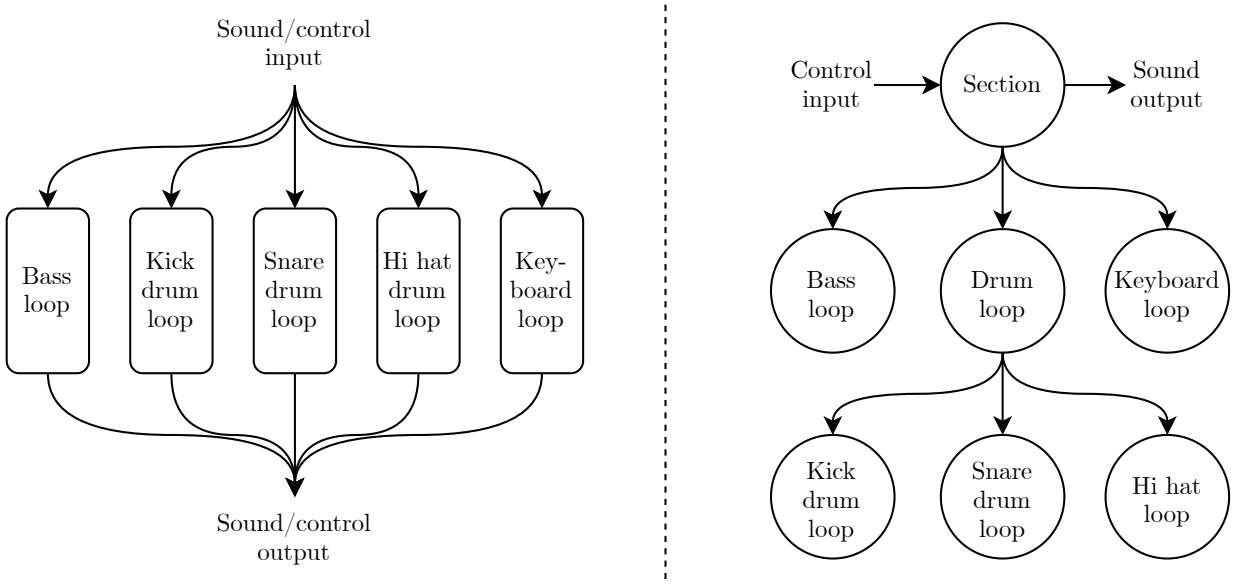

Fig. 2.5 Traditional and hierarchical live-looping structures. Drile by Berthaut et al. (Berthaut et al., 2010).

can use external gestural interfaces. However, on these tools, the recording and playback state can only be controlled through a button. All of the reviewed tools feature either time quantization, external synchronization, or loop manipulation. Most of the tools' mapping strategies are explicit, except for SoundGrasp, which employs mapping using machine learning.

Project	Stream	Interface	Rec/play	Quantization	Ext. sync	Manipulation	Mapping
SoundCatcher	Audio	Ultra-sonic	Footswitch	No	Yes	Audio FX	Explicit
SoundGrasp	Audio	Glove	Posture	No	No	Audio FX	Machine learning
MidiRex	Control	*(MPE)	Button	Yes	Yes	No	Explicit
Midilooper	Control	*(MPE)	Button	Yes	Yes	Random/CV	Explicit
Ribn	Control	Touch	Touch	No	No	No	Explicit
Tetrapad	Control	Touch	Touch	Yes	Yes	CV	Explicit
Drile	Both	6-DoF	6-DoF	Yes	No	No	Explicit

Table 2.1 Comparison of loopers involving gestural mapping. Interface refers to the gestural interface. *(x) means all devices supported by x. Rec/play refers to the interface for switching recording and playback state. Quantization refers to time quantization. Manipulation refers to any real-time processing of the recorded loops.

2.4 Design requirements

The review above acted as a guide in determining the design requirements of this project. I chose to design a control stream looper to address the limitation of SoundGrasp and SoundCatcher, where the sound source cannot be changed post-recording. Inspired by the MidiRex and Midilooper, I decided that the gestural interface should be open-ended, such that the looper could work with various gestural controllers. The loop controls interface should be open-ended, such that these could be controlled gesturally, similar to SoundGrasp and SoundCatcher. To meet the needs of techno and house music genres, I decided to support time quantization to achieve accurate and precise timing. To encourage collaboration when using the looper, it should include an external synchronization feature too. For manipulation, I decided to add a random modulation feature inspired by the Midilooper to the requirements. Further, the looper should support both machine learning and explicit mapping strategies and traditional and hierarchical live-looping

taking inspiration from all the reviewed loopers and addressing some of their limitations. Finally, I decided to design a system that could run on a wireless embedded device to facilitate future support for scalable distributed performances, multimedia mapping, haptic feedback, and other potential uses for IoMusT devices. To encourage the creation of new instruments by the community, I decided that the tools should be released as open-source.

To summarize, the looper should:

- be able to record and loop control stream data from different gestural controllers.
- have an open-ended gestural interface for the loop controls.
- support time quantization.
- support external synchronization.
- be able to manipulate loops by random modulation.
- support both explicit and machine learning mapping strategies.
- support both traditional and hierarchical live-looping.
- be able to run on a wireless embedded device.
- be open-source.

Chapter 3

Design overview and connectivity infrastructure

To build applications for live-looping satisfying the design requirements that I elicited in section 2.4, I first need to develop a connectivity infrastructure for wireless mapping and synchronization. In this chapter, I describe the process of porting existing libraries for mapping and synchronization to a wireless embedded platform. Before diving into the details of the infrastructure implementations, I give an overview of the final instrument's design.

3.1 Design overview

The instrument designed in this project consists of several modules which can be combined in different hardware and software configurations. The modules are

- **Sensor** Acquires gesture input.
- **MapLooper** Module for making mappings between sensor data and synthesis parameters (mapping). The module can playback earlier sensor data readings (looping). The playback is synchronized with other sequencers/loopers (synchronization).
- **Synthesizer** Produces sound output using a synthesis engine with input parameters.

The modules can be embedded within a single hardware device as seen in fig. 3.1. This is the stand-alone configuration of the instrument where no external systems are required to produce sound. The synthesis engine can be moved to a desktop computer for more powerful audio processing. Additionally, the mapping module can be moved to the computer, allowing for using any sensor hardware as the instrument's controller. The external synthesis and mapping configuration is seen in fig. 3.2. Finally, several instances of the instrument may be used together. In this configuration, data (sensor data, sound parameters, and synchronization) is shared through the mapping module allowing for a modular approach to instrument building, where an instrument instance represents a subsystem of a complete instrument. The multi-instance configuration is illustrated in fig. 3.3. In all of these configurations, a transport to communicate data between modules is needed. The transport can be either wired or wireless. In this project, I chose a wireless transport to accommodate compatibility with the wireless DMI, T-Stick Sopranino (Nieva et al., 2018), and laptops without ethernet ports.

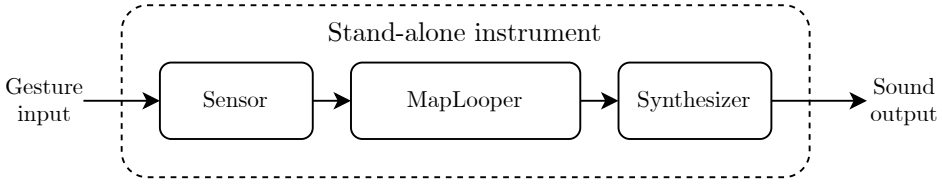

Fig. 3.1 Stand-alone configuration of the instrument.

3.2 Mapping framework

To build a looper with advanced mapping capabilities, I chose the mapping software libmapper (Malloch et al., 2015) as the main building block. Among other software candidates, OSSIA (Celerier et al., 2015) also has advanced mapping capabilities, but OSSIA has more dependencies making it more challenging to implement on an embedded device.

libmapper is an open-source, cross-platform software library for making connections between data signals on a shared network. libmapper builds on a distributed approach to mapping,

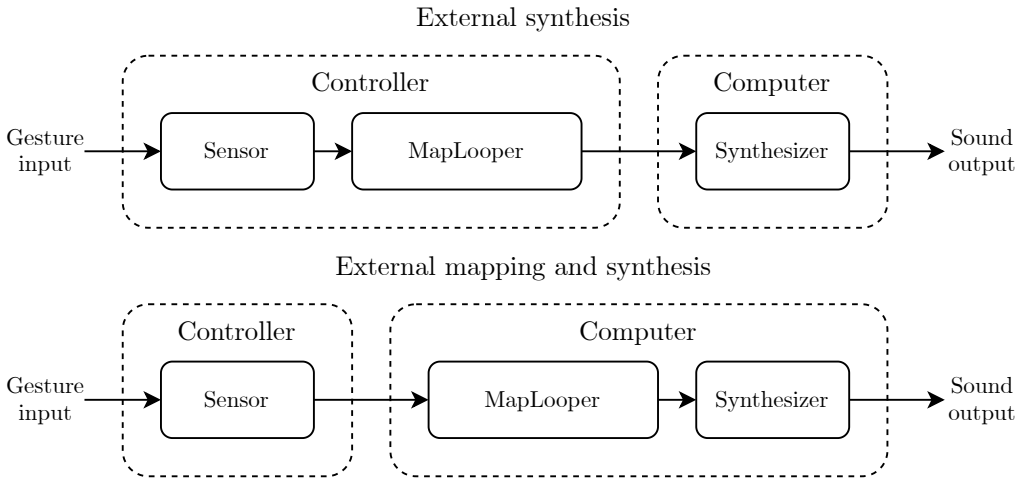

Fig. 3.2 Controller/computer configuration. Doing synthesis on a computer gives access to more powerful audio processing. Moving the mapping/looping/synchronization module to the compute allows for using any type of sensor module as controller for the instrument.

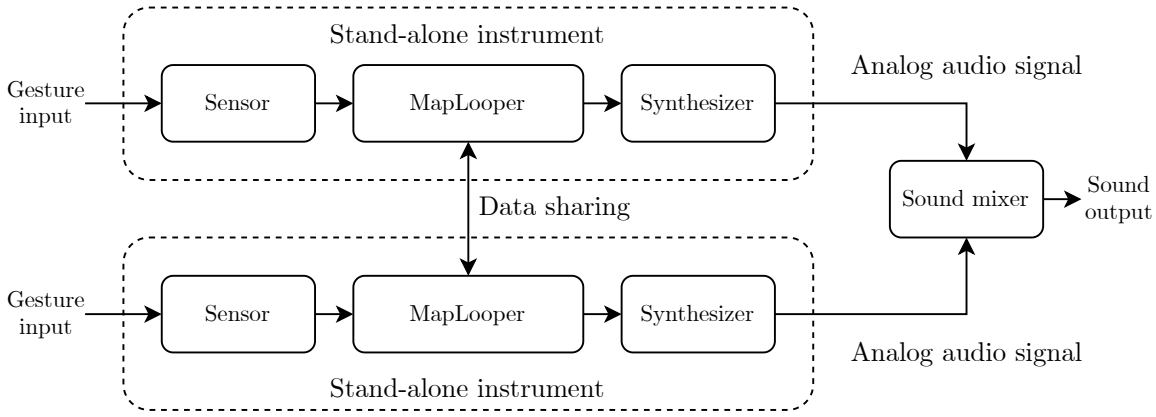

Fig. 3.3 Multiple stand-alone instruments. Sensor data and sound parameters are shared between instruments and loop playback is synchronized. The sound produced by the instruments is mixed through a sound mixer.

and hence there is no centralized server handling the communication. Instead, libmapper uses peer-to-peer messaging to promote shared access to data between peers. Devices and software running libmapper announce themselves through multicast UDP messages on a local network, either wirelessly through WiFi or wired through an Ethernet cable.

Devices in libmapper have *signals*, an abstraction representing data streams such as sensor data or sound generator parameters. Using OSC as the communication protocol, supported data types for signals are 32/64-bit floating-point and integer numbers. Signals can be multi-dimensional (called signal vectors) and have multiple instances describing polyphonic synthesizer voices or multi-touch screens. The *map* abstraction creates mappings between signals. Instantiating a map forms a network connection between devices managing the transmission and reception of signal data. Maps can have *map expressions*, a mini-language for expressing signal processing of source signals through mathematical expressions, supporting familiar math operators and processing techniques such as FIR and IIR filtering.

Mappings can be created explicitly, either through the libmapper API or one of the libmapper front-end GUI's, e.g., WebMapper (Wang et al., 2019), a web-based mapping visualization tool, that polls the network for maps and signals and display these in several different visualizations. Mappings can also be created by adding an intermediate device that implements a machine learning algorithm. By creating several "snapshots" of gestural data and synthesizer configurations, the intermediate device creates a mapping based on an algorithm such as an artificial neural network. Additionally, hierarchical mappings structures can easily be created and visualized through WebMapper's node visualization mode seen in fig. 3.4.

3.3 Synchronization framework

For tight synchronization on wireless networks, I chose to use Ableton Link (Goltz, 2018). Ableton Link is an open-source C++ library for synchronizing tempo, beat, phase, and start/stop commands on wireless and wired networks. Like libmapper, Ableton Link is a peer-to-peer technology and uses multicast for the discovery of peers. The distributed approach allows everyone to change

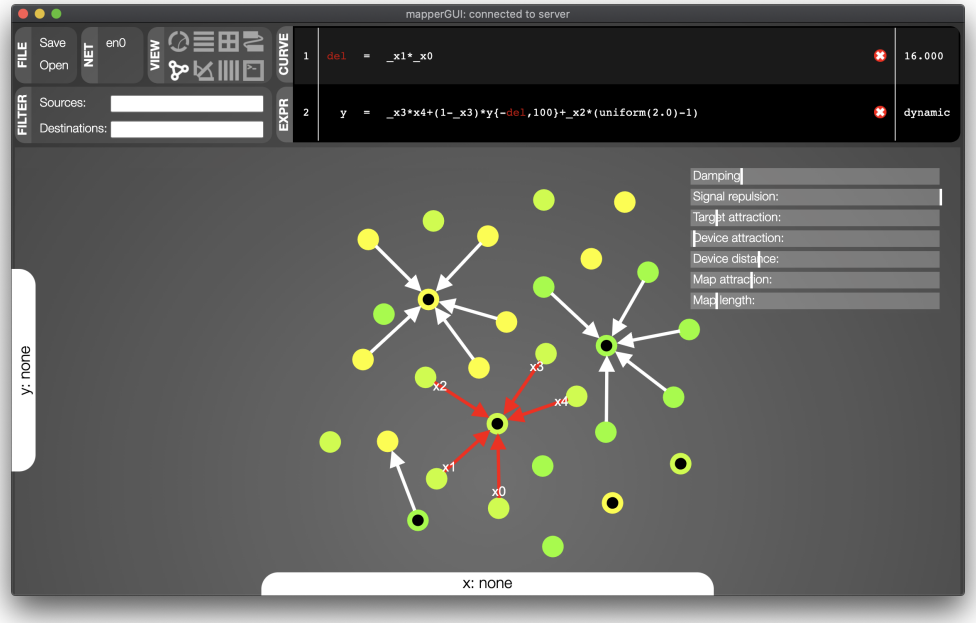

Fig. 3.4 Node visualization mode of WebMapper.

the tempo in a session, removing the main/secondary configuration step of technologies such as MIDI beat clock or OSC sync (Madgwick et al., 2015). Additionally, having phase synchronization makes Ableton Link more capable than the MIDI beat clock protocol, which only synchronizes tempo and sends start/stop messages. Finally, Turchet et al. mention Ableton Link as a candidate for becoming a standard for music synchronization for IoMusT devices (Turchet, Fischione, et al., 2018). Ableton Link is supported on Windows, macOS, and Linux.

3.4 Embedded platform

For the wireless embedded platform, I chose the microcontroller ESP32. Another viable platform, Raspberry Pi Zero W, could also have been used, but as the ESP32 is small, cheap, and powerful enough for digital signal processing (Michon et al., 2020), this was the chosen platform. Other projects, from and with close collaborators, such as the DMI T-Stick Sopranino (Nieva et al., 2018), the 1-DOF rotary force-feedback device for DMI's TorqueTuner (Kirkegaard, Bredholt,

Frisson, et al., 2020), the algorithmic sequencer T-1, (Kirkegaard, Bredholt, & Wanderley, 2020), also employ an ESP32, making these projects compatible with the mapping and synchronization infrastructure I developed here. Additionally, many libraries for interfacing with sensors, actuators, and LED lights exist for the ESP32, making future infrastructure applications such as multimedia installations possible.

ESP32 is a system on a chip (SoC) developed by Espressif Systems and released in 2016. ESP32 incorporates a 32-bit 240MHz dual-core Tensilica Xtensa LX6 microprocessor and has WiFi and Bluetooth connectivity. For developing applications, Espressif provides an SDK, *Espressif IoT Development Framework* (ESP-IDF). The SDK is based on the FreeRTOS real-time operating system ("FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions", n.d.), which allows pre-emptive multitasking on the ESP32, facilitating the development of real-time applications. Additionally, the *arduino-esp32* package provides support for the Arduino environment, giving access to a large number of community-developed Arduino libraries. The toolkit *arduino-esp32* is based on ESP-IDF, allowing the user to mix high-level Arduino code with more advanced and lower-level ESP-IDF subroutines. ESP32 is available in several different hardware modules. In this project, I used the ESP32 WROVER, as it has an external 8 MB RAM chip, useful for digital signal processing.

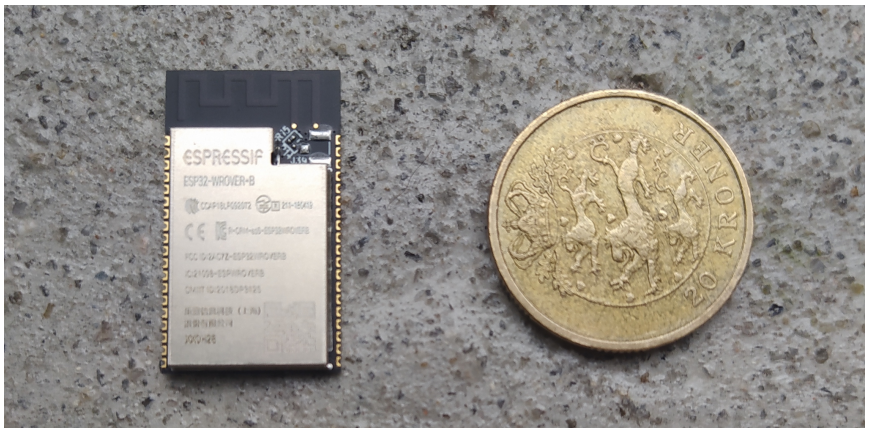

Fig. 3.5 ESP32 WROVER module.

3.5 Porting libraries

3.5.1 libmapper

To compile libmapper for ESP32, its dependencies needed to be resolved first. libmapper had two dependencies, the library *liblo*, for OSC communication, and *zlib*, for data compression. liblo was not supported on the ESP32, so this library needed to be ported first. The zlib library had no dependencies, and it was possible to compile it as is, with no modifications.

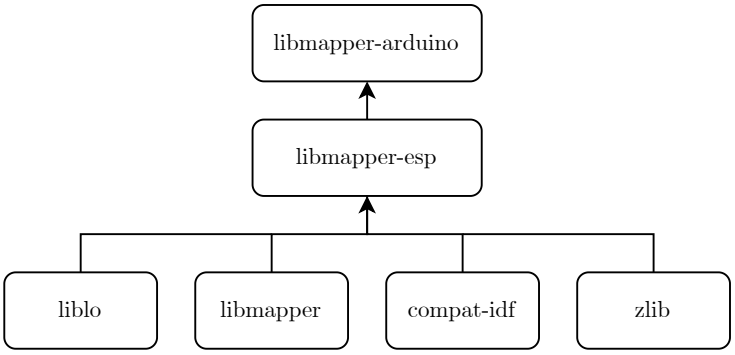

Fig. 3.6 Structure of the libraries ported to ESP32 for libmapper support.

liblo

The liblo library relies on POSIX sockets and threads (pthreads) for creating UDP/TCP sockets and servers. The ESP-IDF includes the library lwip, a lightweight TCP/IP stack. The API of lwip is made to replicate POSIX sockets' behavior, such that the library can serve as a compatibility layer for using POSIX sockets on embedded systems. With FreeRTOS, the ESP-IDF supports multitasking with threads (called *tasks* in FreeRTOS). Conveniently, ESP-IDF contains a pthread library that translates the FreeRTOS API into the POSIX threads API, allowing software depending on pthreads to run on the ESP32. With these tools, a few changes were still needed before liblo could run on the ESP32. Most importantly, several implementations of POSIX functions (see table 3.1) were missing. The functions were implemented and packaged as an ESP-IDF component, *compat-idf*, available on GitHub at https://github.com/mathiasbredholt/compat-idf. With these

four components, liblo, libmapper, compat-idf, and zlib, I successfully compiled and ran libmapper on ESP32. I packaged the full port as an ESP-IDF component, libmapper-esp, available on GitHub at https://github.com/mathiasbredholt/libmapper-esp.

Function	Description
`getnameinfo`	Translates a socket address to a string and an integer representing the IP address and port number. Returns an integer representing the error code.
`gai_strerror`	Returns a string describing the error code returned by `getnameinfo`.
`gethostname`	Gets the hostname of the device.
`getifaddrs`	Creates a Ableton Linked list containing the information about all network interfaces on the device.
`freeifaddrs`	Deallocates the memory allocated by `getifaddrs`.

Table 3.1 Description of the compatibility functions implemented for libmapper and liblo support on ESP32.

libmapper-arduino

To add libmapper support for the T-Sticks Sopranino DMI, I implemented an Arduino version of the libmapper library. Arduino libraries are distributed as source and header files and a text file containing metadata for the library. The Arduino library specification ("Library Specification - Arduino CLI", 2020) has strict directives on structuring the library's source files. The source file structure of libmapper and liblo was not compatible with these directives. According to the specification, a library could alternatively be distributed as a precompiled static library. This solution was more flexible than the specification's directives on source code structuring, so a build system was written that generated the Arduino library as a collection of header files and a precompiled archive file. A few examples were included in the library. I added a description of the examples in table 3.2. The library, libmapper-arduino, is available on GitHub at https://github.com/mathiasbredholt/libmapper-arduino.

Example	Description
ESP32	Sends an increasing floating test signal. Receives an input test signal.
M5StickC	An example for the M5StickC ESP32 board. Sends an increasing floating test signal. Receives an input test signal and displays it to the LCD screen on the M5StickC.
AnalogRead	Reads an analog voltage from the ADC and makes it available on the libmapper network.

Table 3.2 Example Arduino sketches for using the libmapper-arduino library.

Testing

After the porting was completed, I measured round-trip latency, jitter, and package loss of signals sent using the library. The test setup consisted of an ESP32 WROVER KIT development board (Espressif, 2020a) running the libmapper-esp library. In the firmware on the ESP32, an input and an output signal were created. The input signal handler was set to forward incoming data to the output signal. A test software was made that ran on a MacBook Pro (16-inch, 2019) running macOS 10.15. The software sent a 100 Hz signal to the ESP32, and the time between sending and receiving the data was measured. The ESP32 was running in access-point mode, i.e., it created an access point, and the computer was connected to this access-point through WiFi. When measuring WiFi applications' performance, the results can be highly affected by the environment (Grigorik, 2013). WiFi provides no bandwidth and latency guarantees, and the activity of nearby WiFi traffic can have a high impact on the performance (Grigorik, 2013). I did the test in a low-density rural area (Gudmindrup strand, Denmark) so that minimum disturbance could be expected. The low level of traffic should be taken into account when evaluating my results. The results can be seen in fig. 3.7.

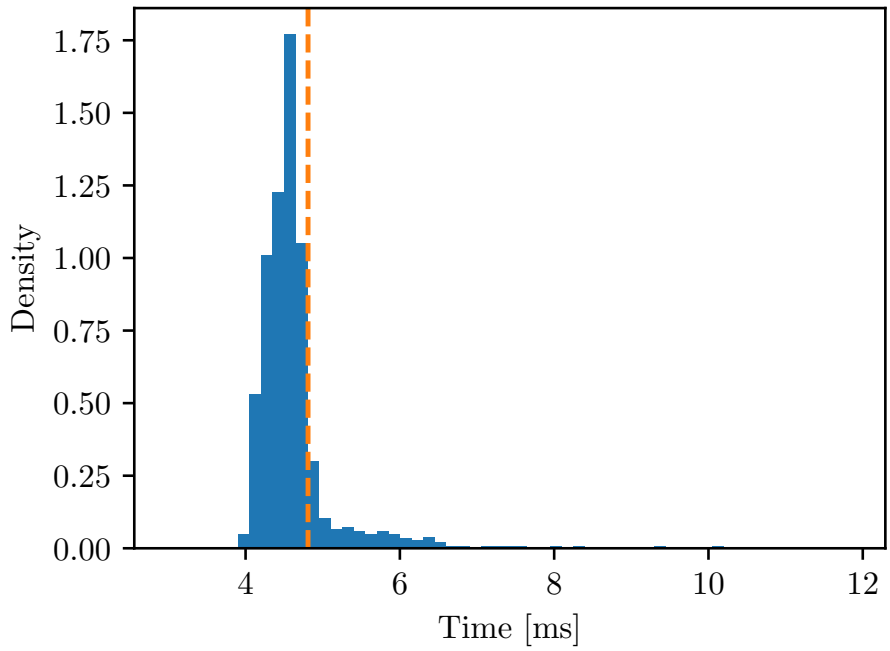

Fig. 3.7 Histogram of round-trip latency measurement with power saving feature disabled. A 100 Hz is signal measured over a period of 100 seconds. The dashed line shows the mean round-trip latency of 4.81 ms.

The mean of the round-trip latency with power-saving disabled[1] was 4.81 ms. According to my results, in a one-way communication situation, where the ESP32 is only transmitting data, an average end-to-end latency L_m

$$L_m = \frac{4.81 \text{ ms}}{2} = 2.41 \text{ ms}$$

can be expected.

Three more measurements were taken at increasing rates for testing latency, jitter, and packet loss P_L at different signal rates. A histogram of the results is seen in fig. 3.8. I found that the system has a significant packet loss for signals at 500 Hz. For signals at 1000 Hz, the packet loss is substantial, with 55% of packets being dropped. The jitter also increases with frequency, which

[1] I found that the ESP32 has a WiFi power-saving feature enabled by default. Disabling this feature had a significant impact on low latency performance. I did measurements with power-saving, both enabled and disabled. The mean of the round-trip latency with power saving enabled was 406 ms.

can be observed in the increase of the standard deviation of the latency listed in table 3.3. There is no significant change in the mean latency for signals at 100 Hz and 200 Hz; for signals at 500 Hz, the latency increases by a factor of 3. Signals at 500 Hz and 1000 Hz had similar performance in terms of latency and jitter, but the packet loss increases from 7.8% at 500 Hz to 55% at 1000 Hz (cf. Table table 3.3).

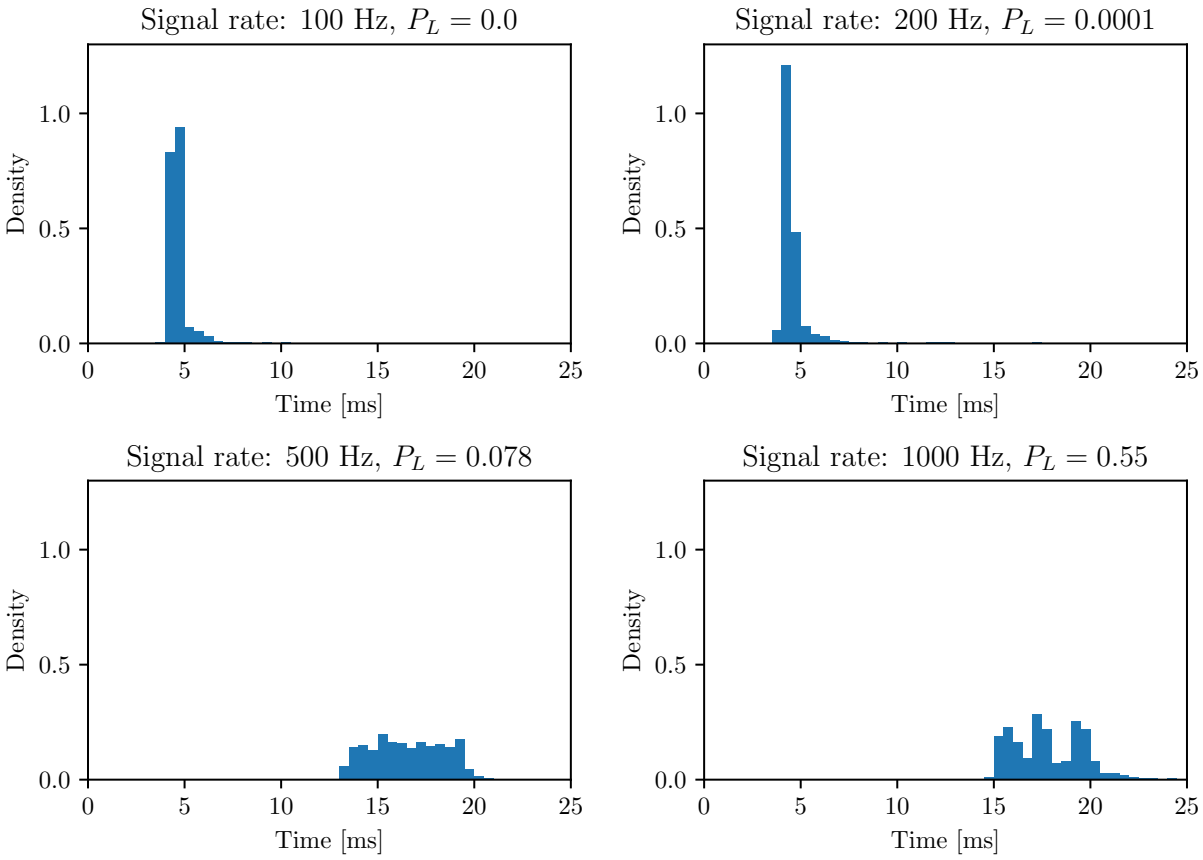

Fig. 3.8 Histogram of round-trip latency of test signals at 100 Hz, 200 Hz, 500 Hz, and 1000 Hz, with 10,000 test points recorded for each signal. Packet loss P_L and frequency is listed in the title of each histogram.

Signal rate [Hz]	Mean latency [ms]	Std. dev. of latency [ms]	Packet loss P_L
100	4.81	1.56	0.0
200	4.78	1.86	0.0001
500	16.6	1.92	0.078
1000	17.9	1.98	0.55

Table 3.3 Results from latency measurements. Mean and standard deviation of round-trip latency and packet loss for the four signal rates.

Result evaluation

This project's results for the embedded libmapper implementation were slightly better than previous studies by Wang et al. (Wang et al., 2020), who conducted tests of latency and jitter with OSC communication over WiFi using ESP32. The study measured a mean 6.62 ms round-trip latency, which is slightly higher than the 4.81 ms measured in this project. The difference can be attributed to the computer being connected through an access point created by the ESP32.

Wang et al. found that there are several issues with using WiFi for DMI performances. First of all, the mean end-to-end latency increases as the number of devices increases. For instance, it was found that for 12 devices sending messages at 100 Hz, the latency was 16.45 ms compared to 6.62 ms for a single device. For a single device, though, this is well below the upper bound of 10 milliseconds on the computer's audible reaction to gesture, proposed by Wessel and Wright (Wessel & Wright, 2002). Additionally, they found that the latency varies significantly with the amount of WiFi traffic in the environment.

They conclude that WiFi has been demonstrated to work well under certain conditions, but for timing-critical applications such as DMI's usage, wired connections are preferred to WiFi. This issue is further discussed in section 5.1.

3.5.2 Ableton Link

This section describes the Ableton Link library's technical details and the components of the Ableton Link port for ESP32. Ableton Link works by establishing a global host time, which works

as a shared reference between all peers in a session. The global host time works as a shared timeline from which the number of elapsed beats can be calculated. Peers joining a session uses ping-pong messaging to calculate their offset to the global host time. Throughout a session, peers measure their offset periodically to stay synchronized.

Ableton Link API

The main component of the Ableton Link API is the *SessionState*. This component holds the global Ableton Link session state, i.e., the global timeline, the tempo, and the playback state. When using Ableton Link within the DSP loop of music applications, real-time safety must be guaranteed to avoid audio buffer underruns. However, the system API calls through which Ableton Link connects to the network are not real-time safe. Therefore, a copy of the session state is saved such that the audio thread can access Ableton Link without waiting for the system calls. Furthermore, the session state copy cannot be protected by locks as this would also compromise real-time safety. Therefore, Ableton Link has two methods for capturing the session state, one for the audio thread and another for the remaining threads. These methods are listed in table 3.4. The methods used to retrieve the tempo and timeline are listed in table 3.5.

Function	Real-time safe	Thread safe
`captureAudioSessionState()`	Yes	No
`captureAppSessionState()`	No	Yes

Table 3.4 Two methods for capturing the session state in Ableton Link.

Function	Description
`setTempo(bpm, atTime)`	At the given time, Ableton Link will attempt to set the tempo in beats per minute.
`beatAtTime(time, quantum)`	Returns the beat value at time with the given quantum.

Table 3.5 Methods for retrieving the timeline and tempo.

3.5.3 Platform modules

Ableton Link relies on the C++ Standard Template Library (STL) and Asio C++ Library (Kohlhoff, 2020) for cross-platform networking. ESP-IDF supports the STL, and Asio has been ported to ESP32 by Espressif (Cermak, 2020). Ableton Link contains a platform component for each supported platform. Each platform component contains platform-dependent implementations of the modules seen in fig. 3.9. To compile and run Ableton Link on ESP32, I needed to create a new platform component for ESP32 and implement these modules. In the following, I describe the implementation of each of these modules.

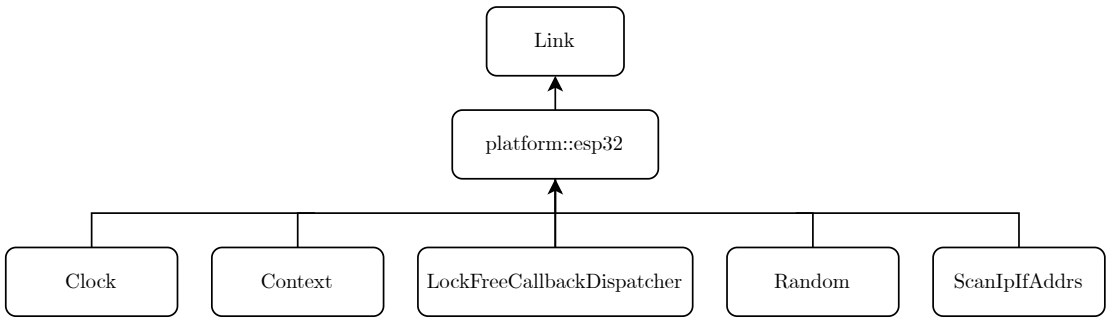

Fig. 3.9 An overview of the ESP32 platform module.

Clock

The Clock module implements a simple timer with microseconds resolution. ESP-IDF contains the `esp_timer` module, an API for a high resolution 64-bit hardware timer. The `esp32::Clock` module casts the value of `esp_timer_get_time()` as a `std::chrono::microseconds` value. This is the time value used by the methods in table 3.5.

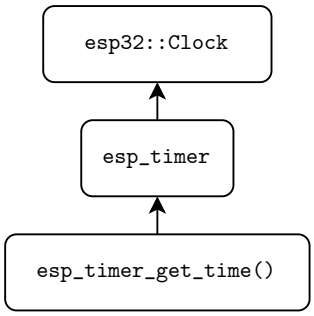

Fig. 3.10 The ESP32 implementation of the Clock module.

Context

The Context module allows for the asynchronous operation of Ableton Link. The module was implemented based on FreeRTOS tasks. The Context module implements a task that repeatedly calls the `poll()` function of an Asio `io_service`, which handles the network communication. Other modules can call the `async()` function of the `io_service`, giving a function handle as the argument. The referenced function will be executed within the task of the Context module.

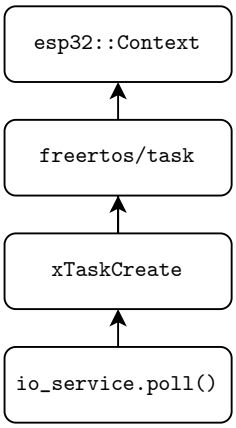

Fig. 3.11 The ESP32 implementation of the Context module.

LockFreeCallbackDispatcher

The LockFreeCallbackDispatcher implements the real-time safety of the session state. When the SessionState needs to be changed from the audio thread, race conditions must be avoided, so other threads must not simultaneously access the SessionState. Ableton Link uses the LockFreeCallback-Dispatcher module to handle this issue. The module implements a task that waits for a notification signal to run a callback. The notification is given every time the audio thread wants to change the SessionState. The callback calls the Context to update the state. This module was ported to use a FreeRTOS task for executing the callback.

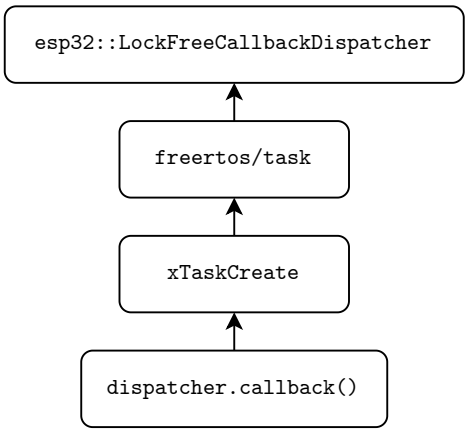

Fig. 3.12 The ESP32 implementation of the LockFreeCallbackDispatcher module.

Random

The Random module generates a random identification string for the peer. The ESP-IDF provides the system API function `esp_random()` that samples noise from the WiFi radio to generate a truly random number. As the identification string needs to be unique for each reboot of the ESP32, this function provides a suitable implementation.

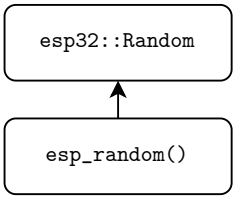

Fig. 3.13 The ESP32 implementation of the Random module.

ScanIpIfAddrs

The ScanIpIfAddrs module retrieves information about the available network interfaces on the system. The ESP-IDF provides the ESP-NETIF API, an abstraction layer for network interfaces on ESP32. This abstraction layer allows Ableton Link to work with WiFi, Ethernet, or a custom interface implementation (e.g., serial over USB) through the ESP-NETIF Custom I/O Driver. The

implementation of ScanIpIfAddrs iterates through all available interfaces, checks if the interface is enabled, retrieves the interface's IP address, and stores the address in a `std::vector`. In some cases, which seemed to be when the WiFi connection is weak, the retrieved IP address was "0.0.0.0", which caused the system to crash. I implemented a check such that this address was not accepted.

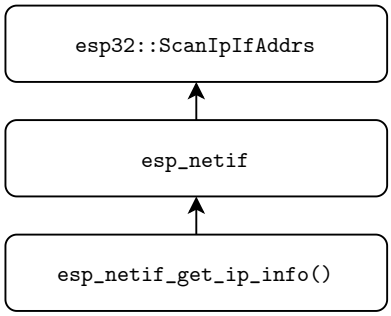

Fig. 3.14 The ESP32 implementation of the ScanIpIfAddrs module.

With these modules implemented, Ableton Link was successfully compiled and ran on ESP32. The library was packaged as an ESP-IDF component, available on GitHub at https://github.com/mathiasbredholt/link-esp.

Testing

To test the port of Ableton Link, I created a test setup for measuring the delay between peers. The test setup consisted of two computers (MacBook Pro (16-inch, 2019) and MacBook Pro (15-inch, 2018), both running macOS 10.15) and an ESP32, all connected to a RIGOL DS1054 oscilloscope. Two probes were connected to an audio jack from the headphone output of each of the computers. The probe for the ESP32 was connected to a GPIO. The computers synthesized a pulse signal through Ableton Live (Ableton, 2020). The ESP32 ran a test software outputting a pulse on a GPIO. All devices were connected through an Ableton Link session and outputted a periodic pulse on every quarter note at 120 BPM. A plot of the measurements is seen in fig. 3.15. I found that the ESP32 performs similarly to the two computers in terms of inter-onset delay. The minimum,

maximum, and average delay between the ESP32 and Computer 1 is seen in table 3.6. The test lasted 10 minutes, and the average delay between Computer 1 and ESP32 was 3.03 ms.

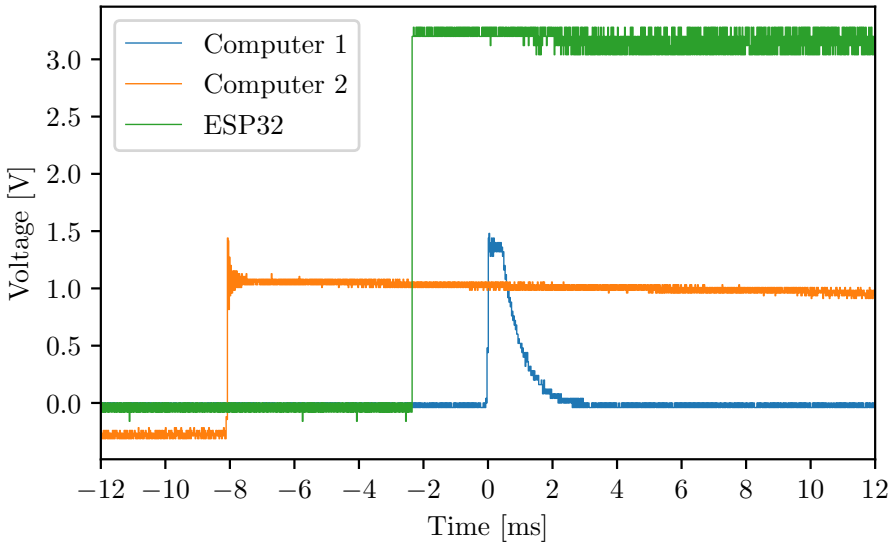

Fig. 3.15 Oscilloscope measurement of a Ableton Link session consisting of two computers and an ESP32. All peers output a pulse signal at every quarter note at 120 BPM.

Min [ms]	Max [ms]	Average [ms]
-6.62	0.02	-3.03

Table 3.6 Results from Ableton Link test. Inter-onset delay from Computer 1 to ESP32.

3.6 Summary

This chapter described the porting of libmapper, a library for mapping, and Ableton Link, a library for synchronization, to the wireless microcontroller ESP32. Both libraries were tested on the microcontroller for performance, focusing on relevant parameters for real-time music performance. For libmapper, a round-trip latency of 4.81 ms for signals at 100 Hz was measured. For Ableton Link, an average inter-onset delay of 3.03 ms was measured. The round-trip latency is well below

the upper bound of 10 milliseconds on the computer's audible reaction to gesture, proposed by Wessel and Wright (Wessel & Wright, 2002). The inter-onset delay was comparable to devices using the existing Ableton Link implementation for desktop computers. The libraries formed the connectivity infrastructure for building the application described in the next chapter.

Chapter 4

Implementation of gesture-to-sound live-looping

This chapter describes the implementation of our gesture-to-sound looper using the previous chapter's connectivity infrastructure. The looper was implemented through three iterations. Two early prototypes are described in the first section: T-Stick looper and Hash table sequencer. The second section describes the final implementation, MapLooper, with delay-lines. The looper was employed in several example applications. All the implementations were written in C++ using primitives from the Standard Template Library (STL).

4.1 Early prototypes

This section describes the implementation of two early prototypes. The initial prototype extended the T-Stick DMI by integrating an MPE-based looping tool into the firmware. The looping tool was extended in the second prototype to handle arbitrary signal types.

4.1.1 T-Stick looper

The T-Stick (Malloch & Wanderley, 2007) is a family of DMI's consisting of a long plastic tube incorporating sensors for motion and touch. Specifically, a T-Stick has copper touch strips for capacitive touch sensing, an inertial measurement sensor (an accelerometer or, more recently, a 9-DOF IMU), a long force-sensing resistor, and a piezoelectric sensor. The T-Stick Sopranino is seen in fig. 4.1

Fig. 4.1 The T-Stick Sopranino. The push button (bottom left picture) was used for switching recording on/off in the performance fig. 4.3.

The T-Stick looper is a self-contained MIDI controller, which allows the recording and playback of sensor data acquired from the sensors within the T-Stick. Similar to MidiRex and Midilooper (see section 2.2.1), the T-Stick looper records MPE control data. Each sensor output can be mapped freely to 3 loop layers. The layers represent the three dimensions of control defined by the MPE specification (The MIDI Manufacturers Association, 2018). By using the MPE specification, I could use existing MPE-compatible VST plugins with the looper. The VST compatibility made it straightforward to prototype mappings for a music performance with the looper (see fig. 4.3). MPE specifies the three dimensions of control as the following types of MIDI messages:

- Pitch bend
- Channel pressure
- Timbre (CC#74)

By assigning a MIDI channel to each Note on/off message, these messages can be sent on the same MIDI channel, thereby achieving polyphonic control of pitch and timbre of individual notes. The T-Stick looper sends MIDI messages to the VST-host over Bluetooth using the MIDI over Bluetooth Low Energy (BLE-MIDI) specification (The MIDI Manufacturers Association, 2015). The BLE-MIDI implementation allowed the T-Stick looper to be used with both desktop and mobile platforms.

Implementation

The implementation of the T-Stick looper involved several steps. First, I ported libmapper to the ESP32 as described in section 3.5.1. The port allowed the integration of libmapper into the firmware for the T-Stick Sopranino (Nieva et al., 2018). The original Sopranino firmware contained an OSC module that sent all the raw data acquired from sensors. With the integration of libmapper, a new mode of operation was added, where signal data only gets transmitted when mappings are created to the signal. Using this mode optimized the bandwidth usage considerably.

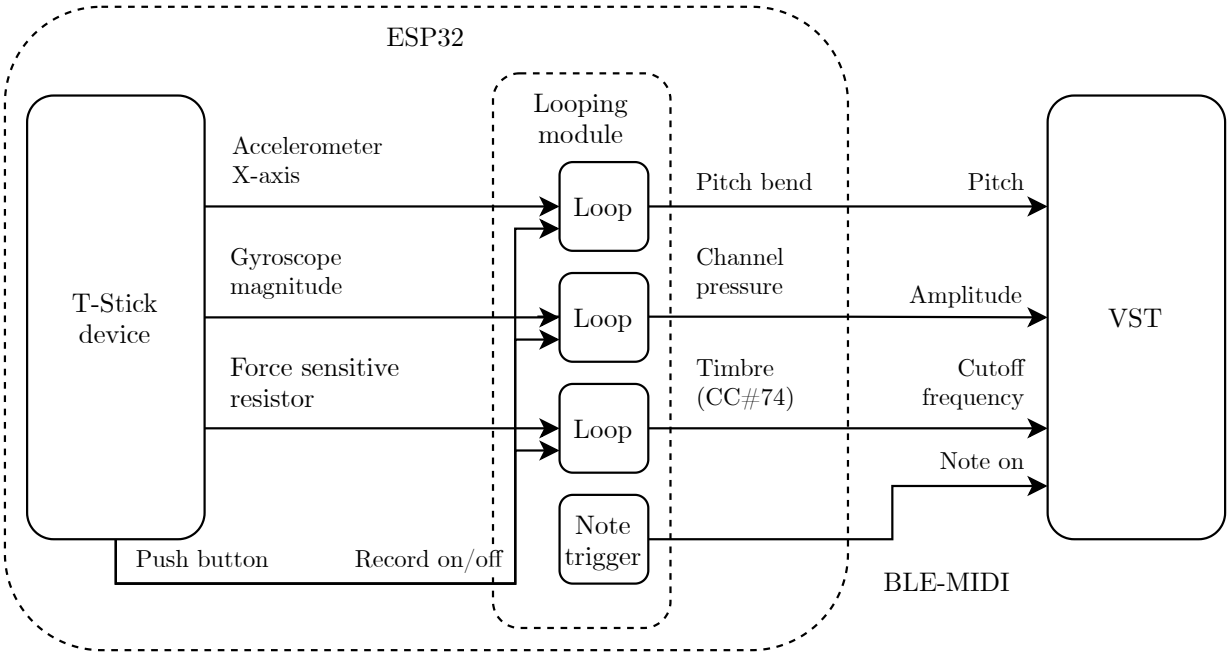

Fig. 4.2 Block diagram of the T-Stick looper including the mapping made for the the music performance.

Second, I implemented a looping module consisting of 3 loop layers, each with a 2-bar long buffer. I integrated libmapper into the module and added an input signal for each layer. The input signals were sampled at a rate synchronized through Link. In the music performance, Link was used to synchronizing the looping module with a pre-recorded drum sample played back on a computer. As the VST synthesizer that I used only created sound after receiving a MIDI Note on message, I added a note trigger to the module. The note trigger sent a note on message at the beginning of each loop, re-triggering the synthesizer.

Performance mapping

For the performance fig. 4.3, I created a mapping from the T-Stick looper to the VST synthesizer. Taking inspiration from Hunt et al. (A. D. Hunt et al., 2002) I wanted to create a mapping that coupled the energy from movements to the amplitude of the produced sound. I implemented this idea by mapping the magnitude of the three axes of the gyroscope to the loop layer controlling

the amplitude of the VST. As the gyroscope measured the angular velocity, the amplitude was non-zero only when the T-Stick was moved. The accelerometer's x-axis, which gave an estimate of the T-Stick elevation, was mapped to the layer controlling the pitch (tone height). This mapping gave an intuitive understanding of high and low pitch, which was obtained by pointing the T-Stick upwards and downward. I mapped the force-sensing resistor's output to the layer controlling the cutoff frequency of a low-pass filter. This mapping allowed me to control the sound's brightness through a 'squeezing' gesture on the T-Stick. A block diagram displaying the mapping can be seen in fig. 4.2.

Fig. 4.3 Still images from video recording of music performance with T-Stick Sopranino using the T-Stick looper at live@CIRMMT event, Café Resonance, Montreal (CA), March 11th 2020. The video is recorded by Mathias Kirkegaard and used with permission.

Advantages and limitations

The T-Stick looper worked well as a quick prototype system for testing different mappings to VST plugins. However, the MIDI resolution of the 7-bit message was limiting for musical expression, and the BLE-MIDI interface only allowed the output to be routed to a single device. Also, the

looper was limited to 3 layers due to the adherence to the MPE standard. Several workarounds such as adding more voices or adding control change layers were considered, but it was deemed that a more general solution would integrate better with libmapper.

4.1.2 Hash table sequencer

To overcome the limitations of the T-Stick looper, I created a second prototype. This prototype was able to record any number of layers and supported 32-bit floating-point resolution. Additionally, the sequencer used a more open-ended data representation, inspired by the data translation approach of libmapper (Malloch et al., 2015). The core of the implementation was the *hash table sequencer*. A hash table is a data structure that is used to store key/value pairs. It uses a hash function to find the array index of the value associated with the key. Hash tables are available in the STL through the `std::unordered_map` data structure.

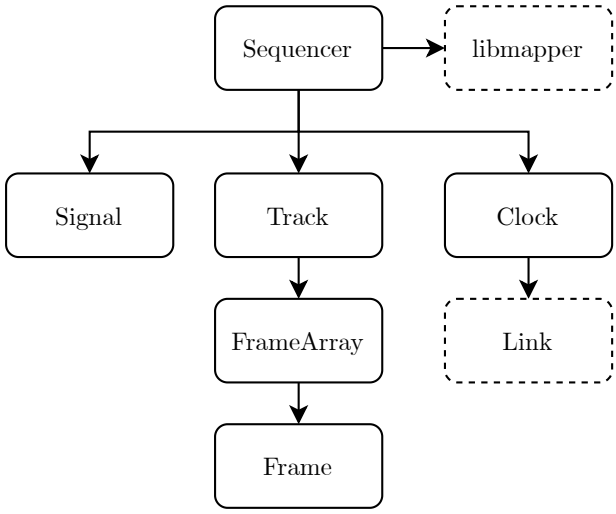

Fig. 4.4 Block diagram of hash table sequencer implementation. External dependencies are marked with a dashed outline.

In the hash table sequencer, every track had an array of hash tables. This array was called *FrameArray*. Each index in the array represented a *Frame* in the sequence. The hash table's keys were control parameters, and the value was a floating-point number representing the value of the control parameter. An illustration visualizing a FrameArray can be seen in fig. 4.5.

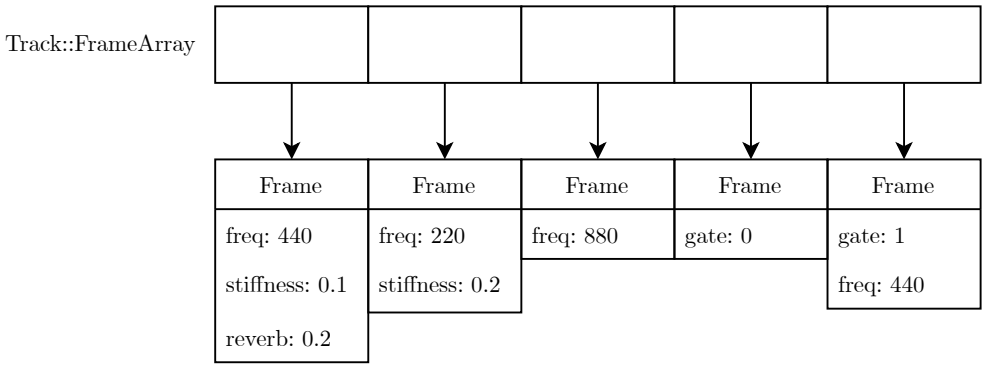

Fig. 4.5 A visualization of a FrameArray with recorded signals *freq*, *stiffness*, *reverb*, and *gate*. Each entry in FrameArray holds a Frame with signal/value pairs.

A Signal class was created as a simple wrapper around libmapper signals to be used as a reference during playback. When the sequencer encountered a Frame with a value that needed to be updated, the string was used to look up the right signal for updating its value through libmapper. As with the T-Stick looper, the input signal values were sampled at a rate synchronized with Link. In the block diagram fig. 4.4, this is represented by the Clock class.

Advantages and limitations

The hash table sequencer added a more flexible data representation and better resolution than the T-Stick looper. The prototype's main limitation was that it did not support signal vectors and signal instances. Support for signal vectors could readily be added, but support for signal instances would require a new implementation of the functionality already existing in libmapper.

4.2 Final implementation

After developing the first two prototypes, I found that more complete integration with libmapper was possible by implementing the loop mechanism within the map expression system of libmapper. The theory behind the final implementation, which I called *MapLooper*, is explained in the following sections. The MapLooper software is available on GitHub at https://github.com/mathiasbredholt/MapLooper.

4.2.1 Looping with a delay-line

As mentioned in section 3.2 libmapper has support for FIR and IIR filtering of signals. Discrete-time systems can be implemented as part of mappings by entering the difference equations into the map expression, allowing filtering techniques such as low- and high-pass. A digital delay-line is a special case of IIR filtering. By adding feedback to the delay-line, a digital looper can be built. Extrapolating from the existing libmapper support for FIR and IIR filtering to delay-lines with feedback is the core idea behind the final implementation.

The discrete-time system implementing a digital looper can be expressed in terms of a linear interpolation between an input $x[n]$, and a delayed output term $y[n - D]$, with the linear interpolation factor representing a record signal $r[n]$

$$y[n] = r[n] \cdot x[n] + (1 - r[n]) \cdot y[n - D]$$

A block diagram of this system is seen in fig. 4.6. For most live-looping devices, the record/playback state is binary, and the signal $r[n]$ is an integer, that is, either 0 or 1. When $r[n] = 0$, only the delay-line output is passed to the system's output. When $r[n] = 1$, the input is passed directly to the output and into the delay-line, thereby being recorded. For $0 < r[n] < 1$, a sort of overdub feature can be achieved as the input is mixed with the delayed input.

4.2.2 Synchronization and time quantization

For a loop to be synchronized to a meter, the length D of the delay-line should be specified in terms of tempo [bpm] T and duration in beats B

$$D = \frac{B \cdot 60}{T}$$

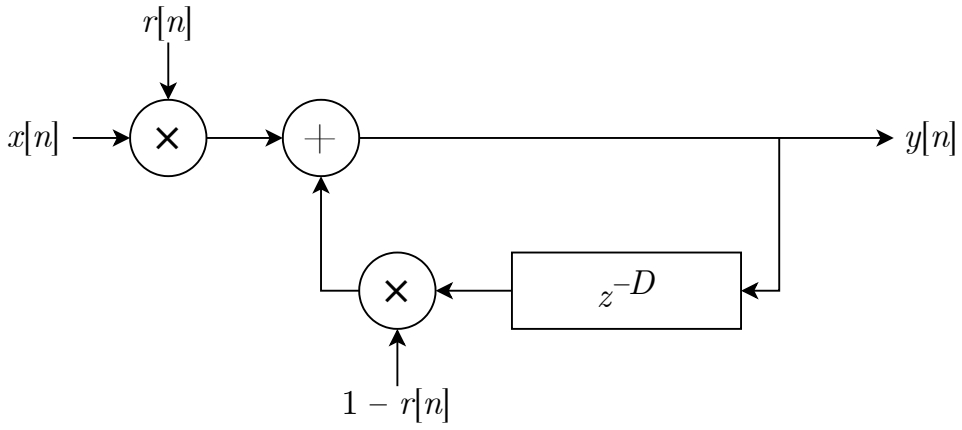

Fig. 4.6 Block diagram of basic looping system implemented as a delay-line with feedback.

For a 1-bar loop with a tempo of 140 bpm and time signature 4/4, this results in

$$D = \frac{4 \text{ beats} \cdot 60 \text{ s}}{140 \text{ beats/min}} = 1.7142857143 \text{ s}$$

However, at the time of implementation, delay-lines in libmapper were non-interpolating in terms of delay-length, and hence the delay-length was integer only. I solved this issue by sampling the input at a rate given by an integer subdivision of the tempo, ensuring that delay-lengths were always an integer multiple of the loop-length in beats. A sample-and-hold structure was added to the system to implement tempo-synchronized sampling. A clock signal $c[n]$ synchronized with the tempo triggers a sampling of the input signal $x[n]$. The rate of the clock determines the quantization. This rate is commonly given for analog synchronization systems in the unit *pulses per quarter note* (PPQN). In the final implementation, the clock is synchronized with Ableton Link. A block diagram of this system can be seen in fig. 4.7.

4.2.3 Loop manipulation

As specified by the design requirements (section 2.4), a system for manipulating recorded sequences was added. A simple modulation system based on the sample-and-hold structure was implemented.

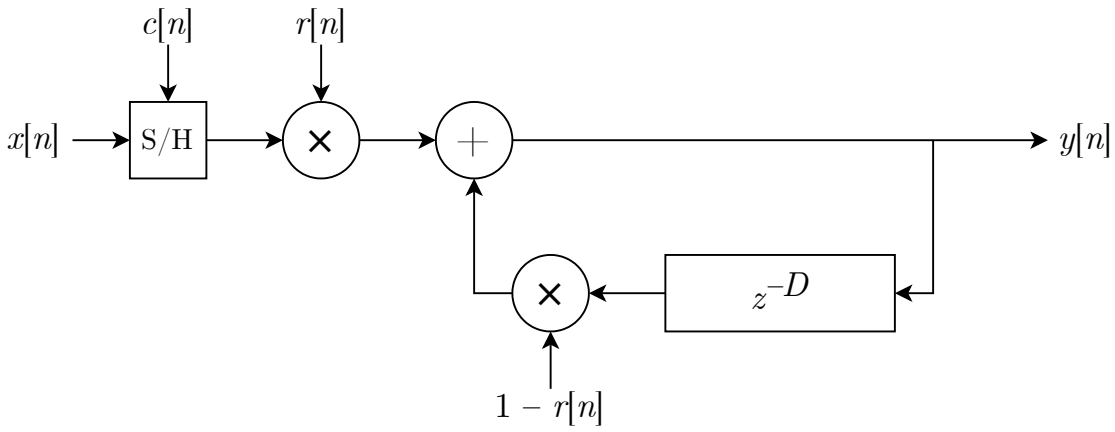

Fig. 4.7 Block diagram of looping system with time quantization.

The modulation source was a uniform noise generator, sampled at the same rate as the input. The modulation signal was added to the system within the feedback path, with the result that an input sequence could be recorded, after which modulation could be applied to make the sequence slowly 'evolve' over time. A block diagram of the system can be seen in fig. 4.8. The uniform noise generator created a noise signal with a range between [-1, 1] multiplied by the signal $m[n]$, controlling the modulation amount. For small amounts of modulation, the original contour of a recorded sequence was retained on a macro timing level with an increasing variation on the micro timing level. An example can be seen in fig. 4.17 in section 4.2.8.

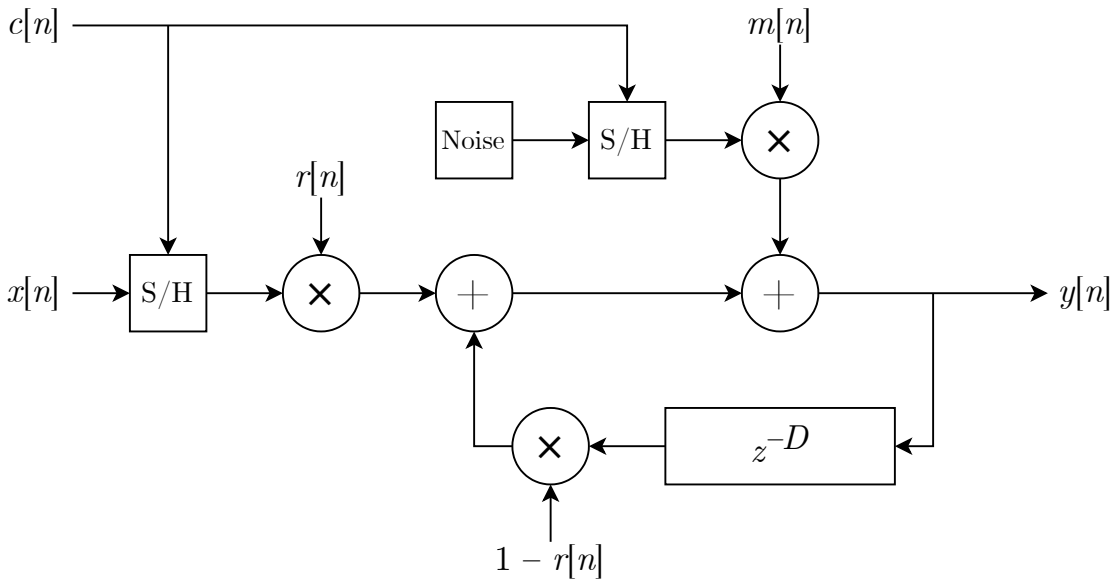

Fig. 4.8 Block diagram of loop manipulation system. The loop is modulated by noise through a sample-and-hold structure. The modulation is within the feedback path.

4.2.4 Implementation details

The final implementation contained two classes *MapLooper* and *Loop* as seen on fig. 4.9. The MapLooper class creates a libmapper device and initiates a Link session. Also, MapLooper holds a `std::vector` of the class Loop. MapLooper has a two public methods, `createLoop` and `update`. The `createLoop` method creates an instance of the Loop class and adds it to the vector. The `udpate` method polls the libmapper device, retrieves the current Link timeline, and calls an update method on all the Loop instances in the vector.

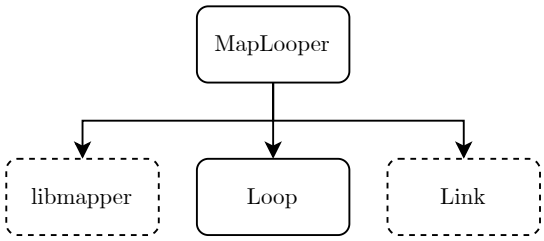

Fig. 4.9 Class overview of final implementation. External dependencies are marked with a dashed outline.

The Loop class represents a single loop layer. When initiated, all the libmapper signals and maps seen in fig. 4.10 are created for this layer. The control interface consists of five signals: *record*, *length*, *division*, *modulation*, and *mute*. A description of these signals is seen in table 4.1.

Signal	Description	Unit	Min	Max
record	Controls whether the looper is recording	-	0	1
length	Length of the loop	beats	1	100
division	Controls the time quantization	PPQN	1	100
modulation	Amount of modulation	-	0	1
mute	Controls whether the looper is outputting a signal	-	0	1

Table 4.1 Control interface of loop layer exposed as libmapper signals. The length is limited by the current maximum of 100 samples of delay in libmapper.

The *record* signal represents the $r[t]$ signal in fig. 4.6. The *length* and *division* signals determines the length D of the delay-line in fig. 4.6 by the relation

$$D = \text{length} \cdot \text{division}$$

The *modulation* signal represents the $m[t]$ signal in fig. 4.8. The *mute* signal was added to control whether the output from *local/recv* propagates to the *output* signal. When the Loop instance is initiated, a convergent map is created between the control signals, the *local/send*, and the *local/recv*

signal. A map expression is created for the map, describing the system in fig. 4.8. In the Loop class update method, the input is sampled at a rate synchronized with Link. The sampled value is sent to the *local/send* signal, and the map expression is evaluated. Finally, if the Loop instance is not muted, the value of the *local/recv* signal is copied to the output signal. By mapping a gestural controller to the input signal and a sound generator to the output signal, a DMI with looping capabilities can be implemented.

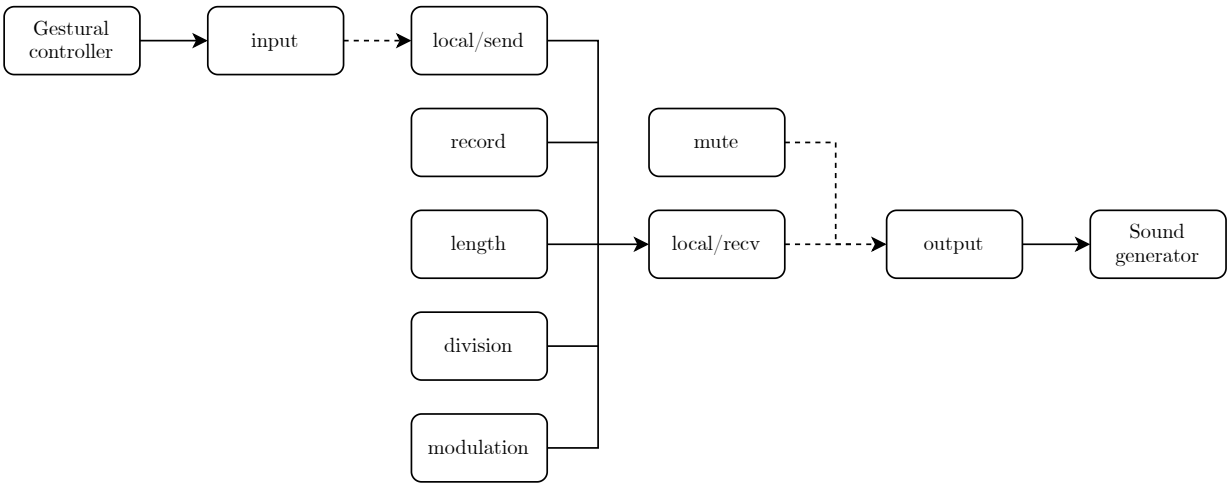

Fig. 4.10 Block diagram of mapping configuration. Each block represents a signal. The solid lines represent the convergent mapping between the control signals and the local send and receive signals. The dashed lines are internal mappings done outside of libmapper.

Local signals

The local signals' purpose is to control the sampling, propagation, and life-time of the map. In libmapper, maps are updated when an input signal is updated. Without a local signal, the gestural controller would be responsible for controlling the sampling rate, which is undesirable as the looper should control the time quantization. Additionally, the local signals allow the Loop class to control whether the signal propagates to the output. Finally, in libmapper, maps are destroyed when one of their signals is removed, removing the map's buffer. The local signals prevent the map from being destroyed when devices go online and offline throughout a session.

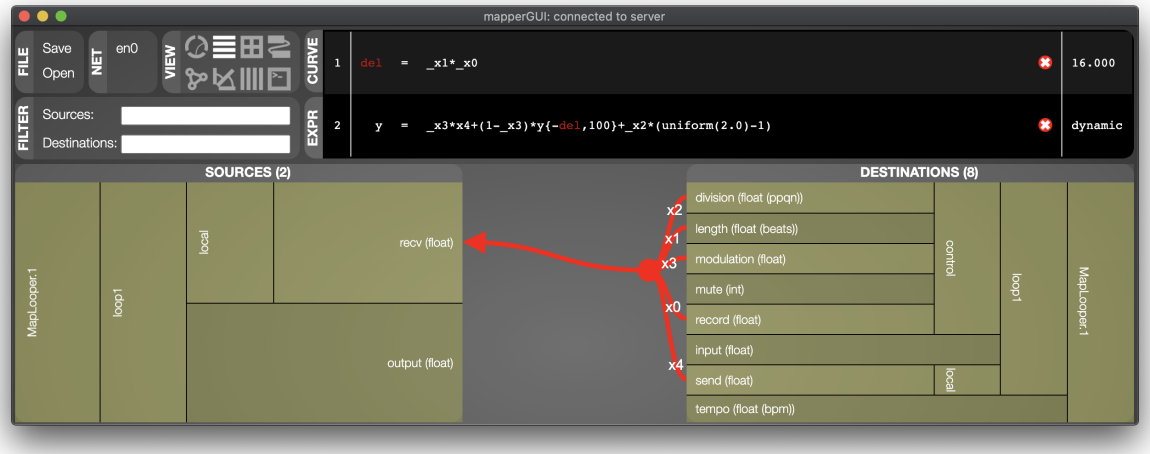

Fig. 4.11 Visualization of the mapping in fig. 4.10 in the mapping visualization tool Webmapper.

Memory requirements

The libmapper API supports 3 data types for signals. These data types are seen table 4.2. As all map buffer samples are tagged with a 64-bit NTP timestamp, these bits should be added when calculating the loop's memory requirements. For a 100 Hz sampling rate, at 120 BPM, the division is

$$\text{division} = \frac{100\ \text{Hz} \cdot 60\ \text{s}}{120\ \text{beats/min}} = 50\ \text{PPQN}$$

Using table 4.2, the required memory for a 4-bar loop with a division of 50 PPQN and the float data type can be calculated.

$$D = 16\ \text{beats} \cdot 50\ \text{PPQN} = 800\ \text{samples}$$

$$800\ \text{samples} \cdot 96\ \text{bits/sample} = 76800\ \text{bits} = 9600\ \text{bytes}$$

Using only the 8MB external memory of the ESP32 WROVER module, the number of 4-bar loops that can be stored simultaneously is

$$\text{Number of 4-bar loops @ 100 Hz} = \frac{8388608 \text{ bytes}}{9600 \text{ bytes}} = 873.8$$

which is much more than needed for most musical applications (depending on the BPM and performance duration). The main limitation here is CPU and WiFi through-put. In its current state, libmapper has a limitation of 100 samples per buffer. This limit should be increased in the future for creating longer sequences.

Data type	Value	Timestamp	Total
int	32*	64	96
float	32*	64	96
double	64*	64	128

Table 4.2 Memory requirements for different data types in bits per sample. *Value bits should be multiplied by the signal vector length. Timestamp applies to entire vector.

4.2.5 Auto-mapping

During the development, it was found useful to create default mappings for the example applications automatically. A simple auto-mapping system was created for setups, where the mapping between signals on multiple physical devices was needed. For each input and output signal of Loop, a function taking a string as an argument was created. The function creates a subscriber to the libmapper graph that looks for newly found signals. When a signal is found, its name is compared to a string provided as the argument. If the strings are equal, a map is created. This functionality makes it possible to create setups with default mappings to other libmapper-enabled devices.

4.2.6 Graphical user interface

For testing the implementation with a user interface, a cross-platform GUI application for desktop was created. The GUI is based on the JUCE framework ("JUCE: class index", n.d.) and contains six sliders and a button as seen on fig. 4.12. When launching the GUI, a Loop instance is created, and sliders are initialized to the control signals' default values. The first slider, labeled *input*, sends its value to the loop's input. When interacting with the slider with a pointer, the recorded signal is toggled. This interaction implements the same touch interface as Ribn and Tetrapad (see section 2.2.2) - the value of the slider is only recorded when the slider is pressed. The next slider displays the output of the loop and is not editable. The following four sliders control the length in beats, the amount of noise modulation, division in pulses per quarter note, and tempo in beats per minute. Finally, a toggle at the bottom controls whether the local loop map's output propagates to the loop's output. The GUI is available on GitHub at https://github.com/mathiasbredholt/MapLooper-gui.

4.2.7 Sound synthesis examples

SuperCollider

To test the GUI with synthesis on the computer, I wanted to use SuperCollider. However, the libmapper bindings for SuperCollider are outdated and not very user-friendly as the bindings require recompilation of SuperCollider. Therefore, I decided to implement a new SuperCollider extension for using libmapper. As extensions for the SuperCollider server, called *UGens*, can be dynamically linked at run-time, these extensions do not require recompilation and can be downloaded by the user as precompiled binaries. I created a UGen called MapperUGen, which is available on GitHub at https://github.com/mathiasbredholt/MapperUGen. The extension has two classes, MapIn and MapOut, for creating input and output signals. The signal name and range can be specified as arguments for the constructor. When synths are created and destroyed in SuperCollider, the UGens are erased from memory, which caused maps to SuperCollider to

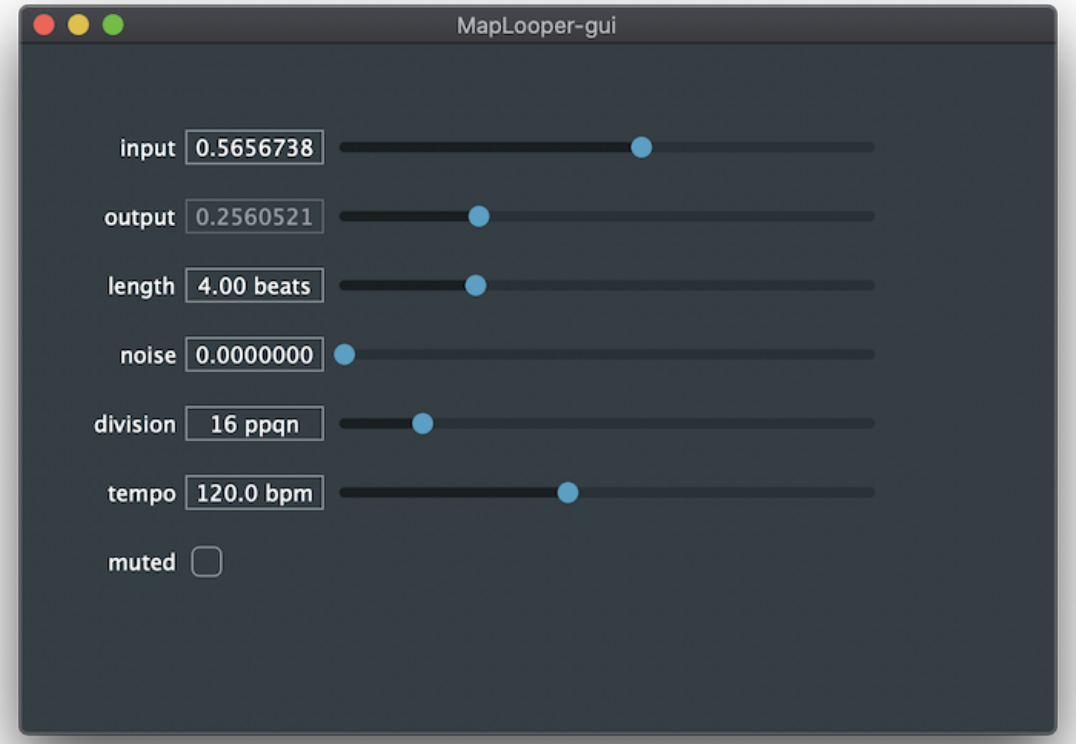

Fig. 4.12 Screenshot of JUCE-based GUI.

be destroyed. A system for persistent maps was implemented by saving libmapper signals in a global variable. When MapIn and MapOut are instantiated, the classes automatically bind to existing signals with the signal name given as an argument. This solution optimized the workflow considerably when prototyping mappings.

```
fork {
  Mapper.enable;
  // Wait 2 seconds for libmapper initialization
  2.wait;
  {
    var index, scale, freqCtl, freq, amp, src, trig;
    // Create buffer with pentatonic minor scale
    scale = 36.collect{ |i|
      Scale.minorPentatonic.degreeToFreq(i, 50, 0);
    }.as(LocalBuf);
    // libmapper input signals
    freq = MapIn.kr(name: \freq, min: 50, max: 2000);
    amp = MapIn.kr(name: \amp, min: 0, max: 1);
    // Quantize frequency to pitch
    freq = Index.kr(bufnum: scale, in: IndexInBetween.kr(scale, freq));
    // Trigger the string on change
    trig = Changed.kr(freq);
    // Karplus–Strong string model
    src = Pluck.ar(in: PinkNoise.ar, trig: K2A.ar(trig), delaytime: 1 / freq);
    src * 0.5;
  }.play;
}
```

Fig. 4.13 SuperCollider code for harp demo.

A musical demo of using GUI was made by mapping the output signal to a harp synthesizer implemented in SuperCollider. The harp synthesizer was based on a Karplus-Strong string model. It had two input signals, *frequency* and *amplitude*. These controlled the frequency and amplitude of the string model. The frequency input was quantized to a melodic scale within the synthesizer, and a slope detector on the quantized frequency triggered the string excitation. The result was that when moving the input slider, melodic notes were triggered along with the range of the slider. The interaction had a similar feel as when sliding fingers over the strings of a harp. The SuperCollider code for the demo is seen in fig. 4.13.

Embedded sound synthesis

I also created a proof-of-concept demo of using the looper with embedded sound synthesis for implementing the stand-alone configuration as described in section 3.1. The demo was based on the ESP32 LyraT board (Espressif, 2020b), which contains an ESP32 WROVER module and an audio codec chip along with 1/8 inch TRS connectors for headphones and auxiliary audio input. An image of the board can be seen in fig. 4.14. The demo project, which is available at https://github.com/mathiasbredholt/MapLooper-faust uses the Faust (Orlarey et al., 2009) library for compiling a DSP program to the LyraT board, which is supported by the Faust compiler (Michon et al., 2020). The DSP program used in the demo is listed in fig. 4.15.

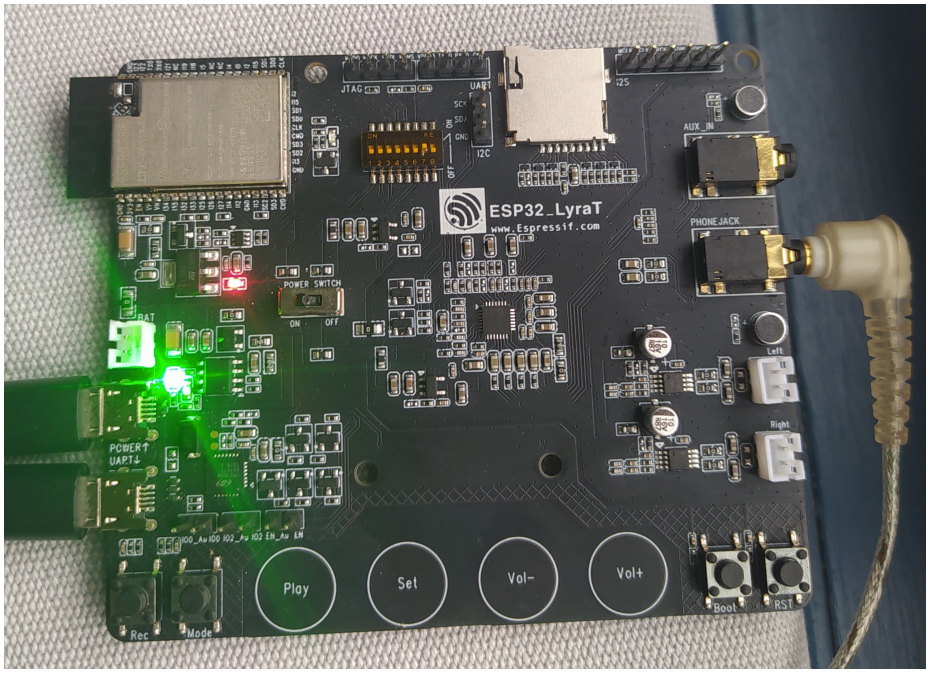

Fig. 4.14 The LyraT board.

The program generated pink noise and passed it through a Moog-style voltage-controlled filter emulation. The program had three parameters, cutoff frequency, and resonance of the filter, and gain of the output. A Loop was created for each parameter mapped to a libmapper signal that

updated the parameter when receiving a value. A random number generator sent an input signal to each of the loop layers. The recorded signal was 1.0 when the program started and was set to 0.0 after 10 seconds. The program continued indefinitely, repeating the same 1 bar sequence. A block diagram of the demo program is seen in fig. 4.16.

```
import("stdfaust.lib");
ctFreq = hslider("cutoffFrequency", 500, 50, 3000, 0.01);
res = hslider("resonance", 0.5, 0, 1, 0.1);
gain = hslider("gain", 1, 0, 1, 0.01);
process = no.pink_noise : ve.moog_vcf(res, ctFreq) * gain;
```

Fig. 4.15 DSP code for embedded synthesis example.

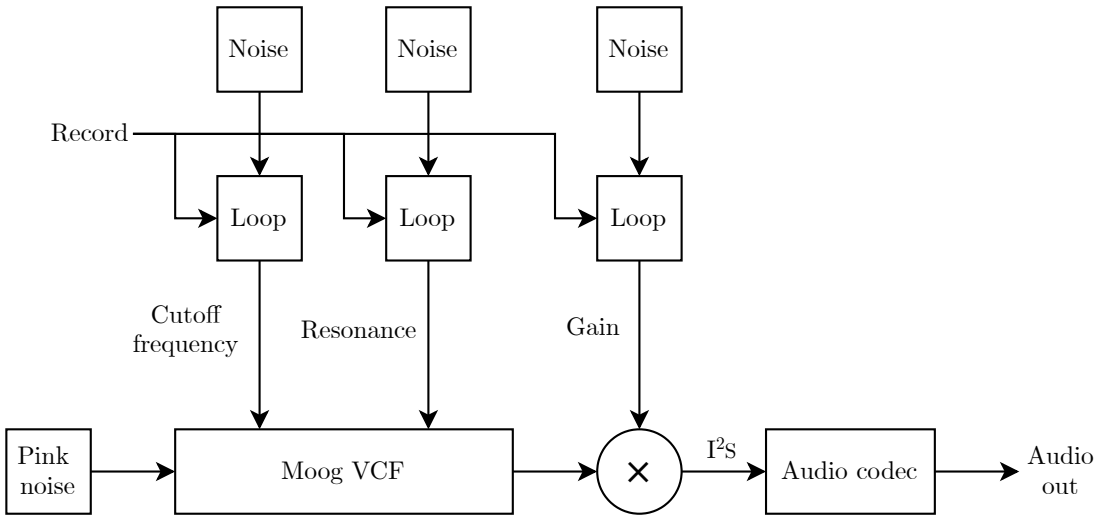

Fig. 4.16 Block diagram of embedded sound synthesis example.

4.2.8 Testing

For verifying and testing the final implementation, I created a testing software for controlling and logging signals. The software created an instance of Loop with a loop-length of 2 beats. A test signal, a single ramp, was passed to the input of the loop. The *record* signal was set to 1.0 for the duration of the loop. At the beginning of the new cycle, the *record* signal was set to 0.0. After the next cycle, the modulation signal was to 0.1, and the noise modulated the ramp. After two cycles,

the modulation signal was set to 0.0, and the cycle repeated itself unchanged until the program ended. The test was run with two different time quantization levels, 16 PPQN, and 2 PPQN. All signals were logged and saved to a file. Plots of the tests can be seen in fig. 4.17.

4.2.9 Advantages and limitations

The final implementation had several advantages over the early prototypes. First, the solution was more scalable, as the implementation through map expressions added support for signal vectors and signal instances. All libmapper signal types could also be used for control stream looping, allowing for looping integer sequences. Also, the map expression interface allowed flexible mapping configurations for loop manipulation. The random modulation implementation could be changed to use any modulation signal by merely changing the map expression.

One limitation compared to the hash table sequencer is that the delay-line based model only allows for continuous signals. The hash table sequencer updated the output signal when a recorded key/value pair was encountered; the delay-line model updated the output at every time quantization step. Continually updating the output can be an issue in scenarios where event-based signal updates are needed. Additionally, the delay-line model causes issues when changing loop-length known from echo effects as zipper-noise. The noise is caused by discontinuities in the signal when moving the read pointer of the circular buffer. For interpolating delay-lines, the zipper-noise is replaced by Doppler-shifts. This effect has been used creatively as an audio effect, but it might not be what the user expects for control data streams. The issue could be solved by cross-fading multiple read pointers when the loop-length is changed. Also, I encountered an issue when the libmapper thread was taking too long to finish, which resulted in phase shifts of the loop sequence. As the map expression delay references the previous sample, it needs to be updated once for every time quantization step. If an update is skipped, the phase is skewed relative to the timeline of Link. The phase skew could be solved by driving the read pointer with Link, but this is not currently possible with the libmapper API. Finally, using local maps is a workaround that might confuse users and cause cluttering in mapping visualizations. A more elegant solution would be possible if

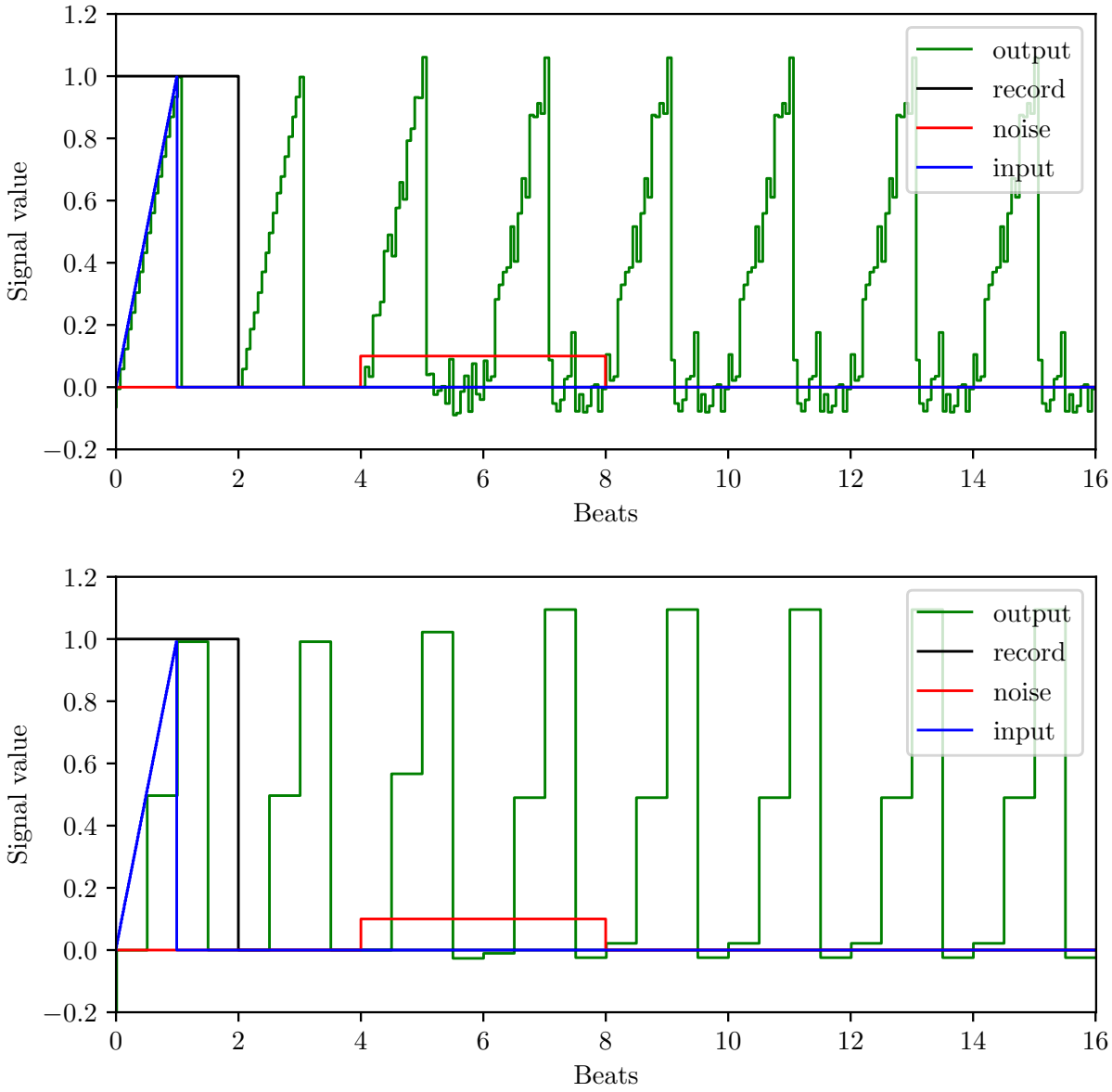

Fig. 4.17 Plot of signals from test and verification software. The y-axis represents the unit-less signal value. The plot at the top has a time quantization of 16 PPQN. The plot at the bottom has a time quantization of 2 PPQN. After 4 beats, the modulation signal is set to 0.1, and the output signal is modulated by uniform noise. After 8 beats, the modulation signal is set to zero, and the output signal is repeated.

all map updates could be controlled by a clock source such that the update happens at regular intervals. If maps could persist when their signals are removed, rerouting loops' outputs to new synthesis processes would be possible.

4.3 Summary

In this chapter, I have described the implementation of a gesture-to-sound looper built on the infrastructure presented in chapter 3. Two early prototypes were built, the first based on MPE message streams, and the second based on hash tables, before settling on a final implementation based on a delay-line model using libmapper map expressions. I have discussed the advantages and limitations of each iteration and verified the final implementation. Finally, I have presented several musical applications built with the tool: 1) A looper integrated into the T-Stick DMI, 2) an embedded synthesizer generating looping sequences, and 3) a SuperCollider harp synthesizer/looper with a graphical user interface.

Chapter 5

Conclusions and future work

I have presented the development of a live-looping system for gesture-to-sound mappings built on a connectivity infrastructure for wireless embedded musical instruments using a distributed mapping and synchronization ecosystem. I ported my ecosystem to an embedded platform and evaluated in the context of the real-time constraints of music performance such as low latency and jitter.

On top of the infrastructure, I developed a live-looping system through three iterations with example applications.

This chapter will discuss perspectives on the work described in this thesis and comment on what could further improve the project.

5.1 Scalability of WiFi for music interaction

I have implemented a connectivity infrastructure and a gesturee-to-sound looper application for wireless embedded devices. With these tools, DMI's and other musical applications for loop-based music can be created for expressive collaborative performances. The scalability of these applications depends on the scalability of WiFi for real-time musical applications.

5.1.1 Compensating latency

In the case of this project, some factors remedy latency issues. As gestures are recorded through libmapper, all samples are time-tagged, which means that latency could be subtracted during playback to achieve accurate timing during playback. Such a system would require peers to continuously measure the latency between them, which could be implemented by periodically sending a heartbeat signal between peers and keeping a record of each peer's round-trip latency. This idea is similar to how host time offsets are handled with Link (see section 3.5.2).

In section 3.5.1, it was found that the libmapper implementation on the ESP32 can maintain a signal rate at 200 Hz, which Wanderley and Depalle describe as a typical gestural acquisition sampling frequency (Wanderley & Depalle, 2004). At frequencies above 500 Hz, the implementation had significant reliability issues. However, when using rhythmic time quantization as an aesthetic strategy, the need for high signal rates diminishes. Besides, each peer can acquire the gestural data at a higher sampling rate for local usage while only sending quantized data to the network. Finally, for applications where these constraints are too limiting, a wired version of the mapping framework can be realized using ESP32 boards with Ethernet connection such as ESP32-POE(Olimex, n.d.). These boards also provide power through Ethernet (POE), removing the need for a battery.

5.2 Visual and haptic feedback

When recording sequences in a looper, the instantaneous feedback is lost when the recording is finished, as the auditory feedback no longer corresponds to the physical gesture currently being held. For recordings of a single loop layer, the GUI implemented in section 4.2.6 provides visual feedback through a slider that displays the looper's current output value. For more complex mappings, such as the one made for the T-Stick looper in section 4.1.1, where three layers are being recorded simultaneously, the current system provides no feedback on what has been recorded.

For short loops, with a duration ranging from 200 milliseconds to six seconds, feedback may be of less importance, as the precognitive sensory information is maintained in our echoic memory,

which for audio and visual stimuli, appears to last in this range (Brower, 1993). For longer loops, feedback could potentially serve as a guide for the performer in expressing a musical idea. This feedback could be in the form of visualization on a screen, displaying multiple recorded sequences simultaneously. With the distributed design of libmapper, such a system could be implemented by mapping the output of loop devices on the network to a computer running a visualization software. This idea is similar to the workings of the mapping visualization tool WebMapper (see section 3.2). The loop visualization tool could even be developed as an extension of WebMapper, taking advantage of existing work, and contributing to the development.

Additionally, feedback could be given in the form of haptic feedback. The TorqueTuner project, which I have contributed to, (Kirkegaard, Bredholt, Frisson, et al., 2020), uses libmapper for changing meta-parameters of haptic effects. A loop could be mapped to display a force related to a recorded sequence.

Also, the Vibropixels project (Hattwick et al., 2017), a wearable wireless vibrotactile display system, could be used to display a recorded sequence or to give cues on the loop-points similar to SoundCatcher (see section 2.1.1).

5.3 Improvements

Several things could be done to improve the project further and explore gesture-to-sound mapping and looping.

5.3.1 Mapping strategies

First, only explicit mappings strategies were explored in the example applications. Interesting loopers could be built using mapping strategies with artificial neural networks and other machine learning algorithms. Also, hierarchical live-looping, as mentioned in the review of Drile section 2.2.3, could be interesting to implement for scenarios with many loop layers. Here, a hierarchical mapping structure could help with controlling many parts simultaneously. Hierarchical live-looping would be straightforward to implement using the WebMapper node visualization seen in fig. 3.4.

5.3.2 Multiple read pointers

Other improvements to the looper, such as multiple variable-speed read pointers, could be implemented to explore new looping techniques inspired by multi-tap delays and granular time-stretching audio effects. Here, the support for signal instances could be used, such that each signal instance represents a read pointer. A single loop layer could control several voices by mapping the instances to voices on a polyphonic synthesizer, adding variations on a micro time-scale. Also, non-constant time quantization could add the shuffle effect popular on many drum machines and featured on the Midilooper (see section 2.2.1).

5.3.3 Run-time implementation

Several opportunities would arise if the suggested API changes to libmapper for allowing persistent maps and clock synchronized signal updates were implemented. For example, the looper implemented here could be implemented with libmapper during run-time through map expressions. Implementation during run-time would increase the availability to users, as loopers could be created from different libmapper front-ends such as WebMapper. For users of the computer music systems, Max/MSP, Pd, and SuperCollider, custom sequencers and loopers for digital laptop orchestras could be built using the existing libmapper bindings (and the SuperCollider extension developed in this project) to create distributed loops shared among orchestra members. For synchronization, Link could be added as a libmapper device and used as a clock source for loopers.

5.3.4 Availability

Another way to increase availability would be to package MapLooper and Link as an Arduino library for ESP32 similar to the libmapper Arduino library developed in this project. Tapping into Arduino's ecosystem would make the tool available to the maker community, allowing more people to use it.

5.3.5 Embedded platform advancements

Announced in 2019, Espressif Systems has released a new SoC, ESP32-S2 (Espressif, n.d.), which adds several new features to the ESP32 platform, some of which are relevant for this project. The ESP32-S2 has USB connectivity, which would allow wired applications of libmapper using Ethernet over USB. Within recent years, computer manufacturers have removed Ethernet ports in favor of USB ports, and therefore in situations where the constraints of WiFi are too limiting, having a USB solution for embedded libmapper would be an advantage for users. The chip manufacturer claims better WiFi performance and stability for the ESP32-S2 compared to the ESP32. This project's measurements should be repeated for the ESP32-S2 to verify these claims. Additionally, the ESP32-S2 has a feature for measuring the time of flight of WiFi packets. This technology can be used for indoor geolocation, which would be interesting to explore for mapping with dance performances, interactive installations, and participatory art.

Bibliography

Ableton. (2020). *Music Production with Live and Push | Ableton*. Retrieved November 17, 2020, from https://www.ableton.com/en/

Berthaut, F., Desainte-Catherine, M., & Hachet, M. (2010). DRILE: An Immersive Environment for Hierarchical Live-Looping, In *Proceedings of the International Conference on New Interfaces for Musical Expression*. https://doi.org/10.5281/zenodo.1177721

Brower, C. (1993). Memory and the Perception of Rhythm. *Music Theory Spectrum*, *15*(1), 19–35. https://doi.org/10.2307/745907

Butler, M. J. (2014). *Playing with Something That Runs: Technology, Improvisation, and Composition in DJ and Laptop Performance* (1st edition). New York, Oxford University Press.

Cascone, K. (2000). The Aesthetics of Failure: "Post-Digital" Tendencies in Contemporary Computer Music. *Computer Music Journal*, *24*(4), 12–18. https://doi.org/10.1162/014892600559489

Celerier, J.-M., Baltazar, P., Bossut, C., Vuaille, N., Couturier, J.-M., & Desainte-Catherine, M. (2015). OSSIA: Towards a Unified Interface for Scoring Time and Interaction (M. Battier, J. Bresson, P. Couprie, C. Davy-Rigaux, D. Fober, Y. Geslin, H. Genevois, F. Picard, & A. Tacaille, Eds.). In M. Battier, J. Bresson, P. Couprie, C. Davy-Rigaux, D. Fober, Y. Geslin, H. Genevois, F. Picard, & A. Tacaille (Eds.), *Proceedings of the First International Conference on Technologies for Music Notation and Representation – Tenor'15*.

Cermak, D. (2020). Espressif/asio. Espressif Systems. Retrieved May 19, 2020, from https://github.com/espressif/asio

Espressif. (n.d.). ESP32-S2 Wi-Fi MCU. Retrieved December 1, 2020, from https://www.espressif.com/en/products/socs/esp32-s2

Espressif. (2020a). ESP32 Modules and Boards - ESP32 - — ESP-IDF Programming Guide Latest Documentation. Retrieved November 11, 2020, from https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/modules-and-boards.html#esp32-wrover-series

Espressif. (2020b). ESP32-Lyrat V4.3 Getting Started Guide — Audio Development Framework Documentation. Retrieved November 2, 2020, from https://docs.espressif.com/projects/esp-adf/en/latest/get-started/get-started-esp32-lyrat.html

Fels, S., Gadd, A., & Mulder, A. (2002). Mapping Transparency Through Metaphor: Towards More Expressive Musical Instruments. *Organised Sound*, *7*(2), 109–126. https://doi.org/10.1017/S1355771802002042

FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions. (n.d.). Retrieved December 1, 2020, from https://www.freertos.org/

Frisson, C., Bredholt, M., Malloch, J., & Wanderley, M. M. (2021). Maplooper: Live-looping of distributed gesture-to-sound mappings, In *Proceedings of the international conference on new interfaces for musical expression*. https://nime.pubpub.org/pub/2pqbusk7/

Goltz, F. (2018). Ableton Link: A Technology to Synchronize Music Software, In *Proceedings of the Linux Audio Conference*. http://dx.doi.org/10.14279/depositonce-7046

Grigorik, I. (2013). *High Performance Browser Networking: What Every Web Developer Should Know About Networking and Web Performance*. "O'Reilly Media, Inc."

Hattwick, I., Franco, I., & Wanderley, M. M. (2017). The Vibropixels: A Scalable Wireless Tactile Display System (S. Yamamoto, Ed.). In S. Yamamoto (Ed.), *Human Interface and the Management of Information: Information, Knowledge and Interaction Design*, Springer International Publishing. https://doi.org/10.1007/978-3-319-58521-5_40

Holland, Simon, Mudd, Tom, Wilkie-McKenna, K., McPherson, A., & Wanderley, M. M. (2019). *New Directions in Music and Human-Computer Interaction* [https://doi-org.proxy3.library.mcgill.ca/10.1007/978-3-319-92069-6]. Springer, Cham.

Hunt, A. D., Wanderley, M. M., & Paradis, M. (2002). The Importance of Parameter Mapping in Electronic Instrument Design, In *Proceedings of the International Conference on New Interfaces for Musical Expression*. https://doi.org/10.5281/zenodo.1176424

Hunt, A., & Wanderley, M. M. (2002). Mapping Performer Parameters to Synthesis Engines. *Organised Sound*, *7*(2), 97–108. https://doi.org/10.1017/S1355771802002030

Instruments, B. (2020). Midilooper. Retrieved November 19, 2020, from https://bastl-instruments.com/instruments/midilooper

Intellijel. (2018). Tetrapad. Retrieved November 20, 2020, from https://intellijel.com/shop/eurorack/tetrapad/

JUCE: Class index. (n.d.). Retrieved November 2, 2020, from https://docs.juce.com/master/index.html

Kirkegaard, M., Bredholt, M., Frisson, C., & Wanderley, M. (2020). TorqueTuner: A Self Contained Module for Designing Rotary Haptic Force Feedback for Digital Musical Instruments (R. Michon & F. Schroeder, Eds.). In R. Michon & F. Schroeder (Eds.), *Proceedings of the International Conference on New Interfaces for Musical Expression*, Birmingham City University. https://www.nime.org/proceedings/2020/nime2020_paper52.pdf

Kirkegaard, M., Bredholt, M., & Wanderley, M. M. (2020). An Intermediate Mapping Layer for Interactive Sequencing (S. Yamamoto & H. Mori, Eds.). In S. Yamamoto & H. Mori (Eds.), *Human Interface and the Management of Information. Interacting with Information*, Springer International Publishing.

Kohlhoff, C. M. (2020). Asio C++ Library. Retrieved May 19, 2020, from http://think-async.com/Asio/

Kvitek, P. (2014). Midirex. Retrieved April 18, 2020, from https://midisizer.com/midirex/

Library Specification - Arduino CLI. (2020). Retrieved May 20, 2020, from https://arduino.github.io/arduino-cli/library-specification/

Linn, R. (n.d.). Linnstrument. Retrieved November 28, 2020, from https://www.rogerlinndesign.com/linnstrument

Madgwick, S. O. H., Mitchell, T. J., Barreto, C., & Freed, A. (2015). Simple Synchronisation for Open Sound Control.

Malloch, J., Birnbaum, D., Sinyor, E., & Wanderley, M. M. (2006). Towards a New Conceptual Framework for Digital Musical Instruments, In *Proceedings of the 9th International Conference on Digital Audio Effects (DAFx-06)*.

Malloch, J., Sinclair, S., & Wanderley, M. M. (2015). Distributed Tools for Interactive Design of Heterogeneous Signal Networks. *Multimedia Tools and Applications*, *74*(15), 5683–5707. https://doi.org/10.1007/s11042-014-1878-5

Malloch, J., & Wanderley, M. M. (2007). The T-Stick: From Musical Interface to Musical Instrument, In *Proceedings of the International Conference on New Interfaces for Musical Expression*. https://doi.org/10.5281/zenodo.1177175

Michon, R., Overholt, D., Letz, S., Orlarey, Y., Fober, D., & Dumitrascu, C. (2020). A Faust Architecture for the ESP32 Microcontroller, In *Sound and Music Computing Conference (SMC-20)*. https://hal.archives-ouvertes.fr/hal-02988312

Mitchell, T., & Heap, I. (2011). Soundgrasp: A Gestural Interface for the Performance of Live Music, In *Proceedings of the International Conference on New Interfaces for Musical Expression*. https://doi.org/10.5281/zenodo.1178111

Nieva, A., Wang, J., Malloch, J., & Wanderley, M. (2018). The T-Stick: Maintaining a 12 Year-Old Digital Musical Instrument (T. M. Luke Dahl Douglas Bowman, Ed.). In T. M. Luke Dahl Douglas Bowman (Ed.), *Proceedings of the International Conference on New Interfaces for Musical Expression*, Virginia Tech. http://www.nime.org/proceedings/2018/nime2018_paper0042.pdf

Olimex. (n.d.). ESP32-POE - Open Source Hardware Board. Retrieved November 26, 2020, from https://www.olimex.com/Products/IoT/ESP32/ESP32-POE/open-source-hardware

Oliveira da Silveira, G. (2018). The XT Synth: A New Controller for String Players (T. M. Luke Dahl Douglas Bowman, Ed.). In T. M. Luke Dahl Douglas Bowman (Ed.), *Proceedings of the International Conference on New Interfaces for Musical Expression*, Virginia Tech. https://doi.org/10.5281/zenodo.1302673

Orlarey, Y., Fober, D., & Letz, S. (2009). FAUST: An Efficient Functional Approach to Dsp Programming (E. D. France, Ed.). In E. D. France (Ed.), *New Computational Paradigms for Computer Music*. https://hal.archives-ouvertes.fr/hal-02159014

Peters, M. (1996). Michael Peters: The Birth of Loop (1996-). Retrieved September 28, 2020, from http://www.livelooping.org/history_concepts/theory/the-birth-of-loop/

Petrovic, N. (2018). Ribn. Retrieved November 20, 2020, from https://apps.apple.com/us/app/ribn/id1413777040

Reinecke, D. (2009). 'When I Count to Four. . . ': James Brown, Kraftwerk, and the Practice of Musical Time Keeping Before Techno. *Popular Music and Society*, *32*, 607–616. https://doi.org/10.1080/03007760903251425

Rodgers, T. (2003). On the Process and Aesthetics of Sampling in Electronic Music Production. *Organised Sound*, *8*(3), 313–320. https://doi.org/10.1017/S1355771803000293

Rovan, J. B., Wanderley, M. M., Dubnov, S., & Depalle, P. (1997). Instrumental Gestural Mapping Strategies as Expressivity Determinants in Computer Music Performance, In *In Proceedings of Kansei - The Technology of Emotion Workshop*.

The MIDI Manufacturers Association. (2015). Specification for MIDI over Bluetooth Low Energy (BLE-MIDI). https://www.midi.org/specifications/item/bluetooth-le-midi

The MIDI Manufacturers Association. (2018). MIDI Polyphonic Expression Version 1.0. https://www.midi.org/articles-old/midi-polyphonic-expression-mpe

Turchet, L., Benincaso, M., & Fischione, C. (2017). Examples of Use Cases with Smart Instruments, In *Proceedings of the 12th international audio mostly conference on augmented and participatory sound and music experiences*, Association for Computing Machinery. https://doi.org/10.1145/3123514.3123553

Turchet, L., Fischione, C., Essl, G., Keller, D., & Barthet, M. (2018). Internet of Musical Things: Vision and Challenges. *IEEE Access*, *6*, 61994–62017. https://doi.org/10.1109/ACCESS.2018.2872625

Turchet, L., McPherson, A., & Barthet, M. (2018). Co-design of a Smart Cajón. *Journal of the Audio Engineering Society*, *66*(4), 220–230. https://doi.org/10.17743/jaes.2018.0007

van Nort, D., Wanderley, M. M., & Depalle, P. (2014). Mapping Control Structures for Sound Synthesis: Functional and Topological Perspectives. *Computer Music Journal*, *38*, 6–22. https://doi.org/10.1162/COMJ_a_00253

Verfaille, V., Wanderley, M., & Depalle, P. (2006). Mapping Strategies for Gestural and Adaptive Control of Digital Audio Effects. *Journal of New Music Research*, *35*, 71–93. https://doi.org/10.1080/09298210600696881

Vieira, R., Barthet, M., & Schiavoni, F. L. (2020, November). Everyday Use of the Internet of Musical Things: Intersections with Ubiquitous Music, In *Proceedings of the Workshop on Ubiquitous Music 2020*. https://doi.org/10.5281/zenodo.4247759

Vigliensoni, G., & Wanderley, M. M. (2010). Soundcatcher: Explorations in Audio-Looping and Time-Freezing Using an Open-Air Gestural Controller, In *Proceedings of the International Computer Music Conference*. http://hdl.handle.net/2027/spo.bbp2372.2010.020

Wanderley, M. M., & Depalle, P. (2004). Gestural Control of Sound Synthesis. *Proceedings of the IEEE*, *92*(4), 632–644. https://doi.org/10.1109/JPROC.2004.825882

Wang, J., Meneses, E., & Wanderley, M. (2020). The Scalability of WiFi for Mobile Embedded Sensor Interfaces (R. Michon & F. Schroeder, Eds.). In R. Michon & F. Schroeder (Eds.), *Proceedings of the International Conference on New Interfaces for Musical Expression*, Birmingham City University. https://www.nime.org/proceedings/2020/nime2020_paper14.pdf

Wang, J., Malloch, J., Sinclair, S., Wilansky, J., Krajeski, A., & Wanderley, M. M. (2019). Webmapper: A Tool for Visualizing and Manipulating Mappings in Digital Musical Instruments, In *Proceedings of the 14th International Conference on Computer Music Multidisciplinary Research (CMMR)*.

Wessel, D., & Wright, M. (2002). Problems and Prospects for Intimate Musical Control of Computers. *Computer Music Journal*, *26*(3), 11–22. https://doi.org/10.1162/014892602320582945

Wright, E. (2017). *Making Hammers with Art: The Producer of House and Techno* (PhD Thesis).