# sygaldry: reusable software for making digital musical instruments

*Travis J. West*

Music Technology Area
Schulich School of Music
McGill University
Montreal, Canada

March 2025

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Philosophical Doctorate.



**2025/03/19**

## Abstract

Digital musical instruments are a family of instruments whose sound medium is digital audio. Their design and development has been the subject of research for over 30 years, and certain tools, techniques, and materials have begun to emerge as standard components for making digital musical instruments. In order to support the long-term maintenance and replication of these instruments, so as to enable long-term research and creative practice and foster the development of our collective digital musical instrument design savvy, we propose to develop a library of reusable digital musical instrument components. Existing libraries are unable to cover the wide range of components, hardware platforms, and software environments used in the making of digital musical instruments, due to issues of glue code, limited software interface, and runtime performance. Leveraging modern C++ design patterns based on compile-time reflection and metaprogramming, we address these issues. Our library, Sygaldry, provides immediate benefit to the replication, porting, and validation of digital musical instruments.



## Résumé

Les instruments de musique numériques sont une famille d'instruments dont le support de son est de l'audio numérique. Leur conception et leur développement font l'objet de recherches depuis plus de 30 ans, et certains outils, techniques et matériaux ont commencé à émerger en tant que composants standard pour la fabrication d'instruments de musique numériques. Afin de soutenir la maintenance et la reproduction à long terme de ces instruments, de permettre la recherche et la pratique créative à long terme et de favoriser le développement de notre savoir-faire collectif en matière de conception d'instruments de musique numériques, nous proposons de développer une bibliothèque de composants d'instruments de musique numériques réutilisables. Les bibliothèques existantes sont incapables de couvrir le large éventail de composants, de plates-formes matérielles et d'environnements logiciels utilisés dans la fabrication d'instruments de musique numériques, en raison de problèmes de code collé, d'interface logicielle limitée et de performances d'exécution. En nous appuyant sur des modèles de conception C++ modernes basés sur la réflexion à la compilation et la métaprogrammation, nous nous attaquons à ces problèmes. Notre bibliothèque, Sygaldry, apporte un bénéfice immédiat à la réplication, au portage et à la validation des instruments de musique numériques.

## Contributions of the Authors

All chapters of this thesis are written in whole by Travis West ("I" below and throughout the rest of the text always refers to Travis, while "we" refers to Travis and the imagined reader), under the supervision of Marcelo M. Wanderley and Stéphane Huot. The sections in chapter 3 pertaining to the mubone are adapted from the 2022 paper by Travis West and Kalun Leung [1] and are written in whole by Travis West and adapted with the permission of the authors. The rest of the document is originally written for this manuscript.

[2] T. J. West, S. Huot, and M. M. Wanderley, "Sygaldry: DMI components first and foremost," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2024

I wrote this paper in full based on the work presented in this thesis, supervised by Marcelo M. Wanderley and Stéphane Huot.

[1] T. J. West and K. Leung, "Early prototypes and artistic practice with the mubone," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2022

This paper was written in close collaboration with Kalun Leung, who contributed about half of the paper; roughly, the author wrote the sections pertaining to the design and implementation of the mubone, while Leung wrote the sections pertaining to its use and mappings.

[3] T. J. West, "Pitch fingering systems and the search for perfection," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2022

I wrote this paper in full based on a personal project.

[4] M. M. Wanderley, T. J. West, J. Rohs, *et al.*, "The IDMIL digital audio workbench: An interactive online application for teaching digital audio concepts," in *Proceedings of the International Audio Mostly Conference*, 2021

This paper was largely written by Marcelo M. Wanderley, based on the implementation of the IDMIL Digital Audio Workbench I developed in close collaboration with Josh Rohs.

[5] L. Turchet, T. J. West, and M. M. Wanderley, "Touching the audience: Musical haptic wearables for augmented and participatory live music performances," *Personal and Ubiquitous Computing*, 2021

Luca Turchet wrote this paper in full, based in part on results from a study I helped him to

conduct during my MA thesis research, supervised by Marcelo.

[6] T. J. West, B. Caramiaux, S. Huot, *et al.*, "Making mappings: Design criteria for live performance," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2021

I wrote this paper in full based on follow-up analysis of data collected during my MA thesis research, supervised by Marcelo and Baptiste Caramiaux.

## Acknowledgements

What a long and difficult journey! I think a PhD is normally challenging, without the added complications of a global pandemic and various complex developments in my personal life. I am indescribably grateful to so many people that helped me make my way through this unusually hard and educational period of my life, and who helped me get here in the first place. Thanks to my supervisors, Marcelo and Stéphane, for skillful guidance and support. Thanks to all the amazing teachers who I've had the privilege of learning from. Special thanks to Michelle McCartney, who first inspired a love of music in me, to Ross MacIntyre and Heidi Wood, whose encouragement inspired me to pursue that love, and Kevin Austin, whose guidance helped me pivot from music maker to musical instrument maker, and again to all of my teachers. I never yet learned a lesson that hasn't turned out to be more enriching and useful than I expected. Thanks to all my friends and colleagues at the labs for all the connection and conversation. Special thanks to Johnty for always keeping in touch, and to Kalun for our particularly fruitful collaboration on the mubone. Thanks to my whole family, to Josh for always calling, to Mum and Dad for so much support and for always believing in me. Thanks V for helping me get through the really rough times. Thanks Emil for being there for me for so long, for all the good times, and everything I learned from the hard times. I couldn't have done it without all of you, including those unnamed. You've all had an influence for which I am grateful.

This work was financially supported by grants from McGill University, l'Université de Lille, CIRMMT, SSHRC, and the FRQSC.

# Contents

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| ADC | Analog-digital converter |
| API | Application programming interface |
| DAC | Digital-analog converter |
| DAW | Digital audio workstation |
| DSP | Digital signal processing |
| DMI | Digital music instrument |
| FSR | Force-sensitive resistor |
| I2C | Inter-integrated circuit |
| MIDI | Musical instrument digital interface |
| MIMU | Magnetic-inertial measurement unit |
| MCU | Microcontroller unit |
| OSC | Open sound control |
| OTS | Off-the-shelf |
| Pd | Pure data |
| SBC | Single-board computer |
| SOC | System on chip |
| SPI | Serial peripheral interface |
| UDP | User datagram protocol |
| USB | Universal serial bus |

# Chapter 1

# Introduction

## 1.1 Digital Musical Instruments

In the organology of musical instruments, it is typical to describe different families of instruments by the acoustic phenomena by which they sound; aerophones make sound by exciting a resonant column of air, idiophones by exciting acoustically resonant solid volumes, cordophones by exciting strings, and so on. In these terms, the development of novel musical instruments since the 1900's has been dominated by what could be termed electrophones, or electroacoustic instruments, which operate by generating and processing an electrical signal that is analogous to an acoustic wave and used to drive the movement of a loudspeaker. Such instruments have a more flexible relationship with acoustic laws than purely acoustic instruments. The loudspeaker is essentially a general-purpose acoustic generator. Obviously each different speaker is subject to certain constraints on its frequency response, loudness, and so on, but for all practical intents and purposes, "good enough" speakers are almost always available that can produce all the necessary acoustic waves at more or less the required loudness; we can essentially take for granted that we can produce from the loudspeaker any sound that we want, provided that we can generate the appropriate electrical signal with which to drive the speaker. Many signals that we could drive a speaker with will produce no audible sound (e.g. a steady DC offset or a 2.4 GHz sinusoid). The task is then

to produce signals with properties analogous to audible sound (e.g. with energy in the frequency bandwidth from roughly 20 Hz to 20 kHz). We can term such electrical waves as electrical audio signals.

Over the course of the 1900s, concurrent with the development of electronic instruments of the sort described, computers began to find application to audio. Using a digital-to-analog converter (DAC), a sequence of binary numbers can be converted into an electrical audio signal that can in turn be used to drive a loudspeaker and produce sound. The DAC, like the loudspeaker, is essentially general-purpose and "good enough", so that we can usually assume that any sequence of numbers with audible qualities, what we can term as a digital audio signal, can be converted to an electrical audio signal and then to acoustic sound waves. Instruments that operate primarily with digital audio signals can be termed digital musical instruments (DMIs). Since the digital audio signal must be transduced to an electrical audio signal before being used to drive a loudspeaker, DMIs can be considered as a subset of electroacoustic musical instruments.

So with electronic instruments we have a situation where we no longer deal with acoustic laws, and instead work with and around electrical laws to produce circuits that make interesting audio signals. With DMIs, we no longer deal with acoustic laws, nor electrical laws, nor indeed with any practical constraints besides the limits of our collective knowledge. Any stream of numbers can be treated as a digital audio signal, and in principle we can produce any imaginable stream of numbers, so it follows that we are unburdened by constraints and instead laden with absolute freedom, and we can produce any sound that can be perceived by humans, if we only know a way to produce the appropriate stream of numbers. As one might expect, the limitation that we must know a good method for producing a certain sound to be able to make it turns out to be fairly severe. If a sound can be produced acoustically, it is easy enough to play a recording of it, but it is particularly challenging to produce interesting sounds from scratch, not based on analysis or playback of recorded signals, and it is further challenging to produce sounds that can be controlled in real-time so that they evolve in interesting ways. Digital audio nevertheless provides a high degree of freedom and offers many rich opportunities to make interesting sounds that can't easily

be produced acoustically or electrically.

Over time, the computer processor performance of widely available platforms began to reach the point where many sound synthesis routines could be run on general-purpose processor in real-time, enabling real-time interaction with these sounding algorithms. The power of consumer computing platforms continued to grow, and the cost continued to decrease. By now, it is common in many parts of the world that an individual should carry with them multiple computing devices capable of real-time audio synthesis and processing, such as a laptop, smartphone, wireless earphones, and fitness tracker. The real-time affordance of these devices enables the development and ubiquity of interactive DMIs and other interactive digital musical devices.

Since the 90s, DMIs have been a focus of research. Their design and development poses numerous challenges and opportunities. DMIs require a tight integration of sound synthesis, sensing, and interaction design, presenting opportunities to extend and integrate findings in all three of these fields of interest, among others. They represent a unique form of human-computer interaction with unusual constraints on real-time responsiveness, depth of engagement, and level of skill, making them an unusual and interesting area for human-computer interaction research. Because of their particular musical affordances and relative novelty, DMIs offer unique musical and sonic opportunities for creative exploration and research-creation. Further, music technology represents a large market where novel and well made DMIs have the potential to solve unique customer problems. DMIs are especially flexible and amenable to adaptation for players with diverse physical and cognitive capabilities, with the potential to make music making more accessible to a wider range of people. As well as these external justifications, it is likely many researchers, musicians, and creators are interested in DMIs for more idiomatic and personal reasons. As well as the rich external opportunities offered by studying this unique family of instruments, my own obsession with their design and performance is as much motivated by incoate child-like excitement, a well of exuberant curiosity that pours forth without any particular reason, fascination that eschews justification. DMIs bring together the puzzle-box problem solving of computer programming, the fine detailed tinkering and crafting of making electronic devices, the somatic engagement of acti-

vating a sounding object with your body in performance, and a whole world of wonderful sounds. There is much richness to enjoy for those who make these instruments, and those who play with them—inherent joys of the kind that can enrich an individual person's life.

The term "DMI", as defined, is relatively generic. It can reasonably be applied to commercial keyboard synthesizers, interactive audio-visual installations, and even video game consoles. In this thesis, we will closely examine DMIs of the sort commonly encountered in the research literature, especially those associated with the International Conference on New Interfaces for Musical Expression (NIME). These instruments are commonly viewed as having three main subsystems: an interface, a sound (or multimedia) synthesizer, and a mapping that associates data from the interface to control the parameters of the synthesizer (see fig. 2.2). These instruments often explore novel paradigms of music performance, frequently involving gestural performance. This research is especially oriented towards two instruments familiar to the author and used as case studies: the T-Stick and the mubone. Both of these instruments are presented in detail in chapter 3. The T-Stick is a cylindrical gestural controller that is designed to be reinterpreted by each performer, who newly invents the gestures that might be used to play the instrument. The mubone is an augmented trombone that links sound and space around the performer. Although the contributions presented have a special affinity for novel gestural DMIs as found in the NIME community, the results likely have good applicability to other sorts of DMIs.

## 1.2 Longevity of DMIs

Influenced by the tradition of human-computer interaction research, and the broader moral and philosophical proclivities of a western scientific perspective, our work developing DMIs is oriented towards exploring the affordances of these technogical devices and incrementally building a body of general knowledge about how they work, what they do, and how they are constructed. Among other broad considerations, we are interested in making long-lived research DMIs. This means learning how to make DMIs that are "good" according to some set of important criteria, so that our research and development efforts can have a positive impact on society, and it means learning

how to make DMIs that can be reused, replicated, and maintained over time; longevity allows our DMIs to be engaged with over long periods of time, enabling us to study longitudinal phenomena involved in creating and engaging with these devices, and allowing for the creation, use, and study of these instruments to cultivate long-term value and thus greater positive impact.

In 2014, Robert Godin of Godin Guitars presented his contributions and the history and development of Godin Guitars' manufacturing capabilities in a CIRMMT Distinguished Lecture [7]. During this presentation, Godin remarks on the longevity of an acoustic guitar.

> "Our instrument is going to live long, and you can fix them. We kept the highest standard, even on a low[-end] one. [...] We just saw a guy, he called us, and he saw on the internet on eBay, he saw an old Norman '74. He called the company and said, 'is it really a 74?' He gave us the serial number, and we said yes. He paid $2000, and this guitar in '74 was sold for about $200. [...] These instruments gain value. What gains value today? Nothing. You buy a computer, three years after, there's no parts, there's nothing. You call the company, they're gonna say it's too old! Three years! Gone. TVs are the same. Everything is disposable. But the way we make instruments, it's not. They increase in value."

Compare, during his presentation in 2014, when Perry Cook demonstrates a MIDI instrument made in the 1990's [8]:

> "My first sort of expressive instrument based on the kitchen was this coffee mug, which I built in 1996 and still works today, which is remarkable to me. Few of us in the computer-mediated space can say that. Unless it's an iPhone, and then it still probably doesn't work because we haven't upgraded to iOS 6 point whatever. This still works, because it just puts out MIDI."

Cook then almost immediately spends just over a minute awkwardly failing to coax a sound out of the instrument in a spectacular example of the demo effect, although he ultimately succeeds in showing that the instrument does indeed still function.

After 40 years, Godin's guitar increased in market value by an order of magnitude in a remarkable demonstration of the longevity of a well-manufactured and maintained acoustic instrument. After 17 years, Cook considered it remarkable that the coffee mug DMI was even still functional. And unfortunately this is in line with broader trends in the DMI research field. Our instruments do not generally accrue value over time. They just stop working. This is not to suggest that the market value of an instrument over time is a strong indicator of its existential or essential value, but the contrast between these examples serves nicely to illustrate that it is a challenge to make long-lived DMIs. It's likely that even highly prized electronic and digital instruments with high market value will tend to be difficult to maintain over time, becoming threatened by the extinction of essential electronic components, discontinuation by the manufacturer, and lack of open source details that might otherwise enable the instrument to be replicated.

In a survey of instruments presented at the International Conference on New Interfaces for Musical Expression (NIME) from 2010 to 2014 [9], Morreale and McPherson investigated the long-term use of DMIs presented at the conference. 70 instruments were surveyed, of which 19 (27%) were never intended to be used over a long period, being school assignments or research probes or otherwise expected to have short-term use. Among the remaining 51 instruments, all originally intended for long-term use, Morreale and McPherson found that just under half of the DMIs surveyed were not ready for performance. "Almost half of the DMIs have been played by fewer than 3 musicians." More than half were performed with fewer than 6 times.

Survey respondents gave numerous reasons why their instruments were not regularly performed with (table 3 in [9]). I suggest these could be grouped into three categories: external reasons, design reasons, and technical reasons. External reasons include "I don't have the opportunity at the moment", "It was a collaborative effort which has stopped", and "I no longer work with DMIs". These are reasons that have nothing to do with the instrument, but which nevertheless hinder its longevity, and account for 44% of responses. Design reasons include "I turned my attention to building other DMIs", "My musical interests no longer align with the capabilities of this DMI", "I was unsatisfied with the musical output", "I was unsatisfied with the playing experience", and

"I am working on an updated version of the DMI". These are reasons that have to do with the requirements placed on the instrument, and specifically with the instrument not continuing to meet these requirements. The designer has moved on to new efforts that better serve their needs. This accounts for 40% of responses. Finally, technical reasons include "the hardware needs too much maintenance" and "the software needs too much maintenance". These account for the remaining 16% of responses, and present purely technical maintenance challenges to ongoing use of the surveyed instruments.

Sullivan and Wanderley [10] surveyed musicians online about their use of DMIs. In this survey, about 45% of respondents cited technical reasons for discontinuing their use of an instrument. Most respondents cited ways in which the instrument failed to meet their requirements ("not useful for my needs", "disliked the interface", etc.).

These surveys highlight that many instruments are abandoned because they do not suit the ever-changing requirements of their performers, while many more are abandoned because they fall into technical disrepair. As well as external factors wholly outside of the realm of design and maintenance, these seem to be the main archetypes of instrument abandonment.

In the requirements case, the performer may move on to new instruments or other pursuits, and better maintenance of the abandoned instrument would likely not make a difference. If the goal is to promote the longevity of the instrument, the designer may ask "what requirements did the instrument fail to meet?", "how can we meet them?", and "why did they change?". In the technical case, better maintenance would directly enable better longevity. The questions for the designer are "what factors hindered maintenance?" and "how can the instrument be developed in a way that facilitates maintenance?".

In both cases, there are social and technical challenges involved. Technical abandonment may be informed by social pressures. Part of what makes maintenance difficult, especially in the research context, is the constant changing of the guard; an instrument designed by one grad student may be maintained by a new one the next year, and another the year after that (section 3.1.3). The transmission of technical knowledge becomes a challenge [11]. Performers' changing requirements

may be informed by technical developments. New technologies bring new possibilities, and what was once cutting edge may quickly seem outdated.

Similarly, in both cases there are social and technical approaches to improve longevity. Responding to shifts in requirements, proponents of an instrument could try to evolve the instrument to better serve new requirements (a technical approach), or try to shift the socio-musical ecology e.g. by encouraging adoption [12] or community development [13], so that performers' interests align with the affordances of the instrument (a social approach). Responding to maintenance challenges, proponents may try to improve the maintainability of the instrument (e.g. by making use of an embedded platform affording more fine grained control over the software environment and dependencies [14]), or make efforts to address the social barriers to maintenance by improving the documentation and replicability of the instrument [15], [16] and other means of propogating technical design prowess [11].

For our purposes, we will focus on the technical strategies for improving longevity. Arguably, these are prerequisite for social approaches. If the instrument is poorly engineered and difficult to maintain, no amount of documentation or technical pedagogy will stave off its inevitable obsolescence. Adequate maintainability is necessary for an instrument to be sufficiently stable and reliable to enable long-term engagement. Stability is also a necessary basis upon which an instrument can sustainably evolve and change to meet shifting requirements of performers (section 3.2.3). The same technical qualities that enable maintainability also tend to facilitate incremental change and adaptation. Maintainable design is maintainable precisely because it is amenable to incremental change. Without maintainability and stability, an instrument's only way of surviving is repeated replication. This is insufficient. Replication always entails a certain amount of re-invention [15], [16]. This precludes long-term compatibility and makes stability impossible.

## 1.3 Overview

In chapter 2 we will consider the structure of DMIs, solidifying a conceptual framework of endpoints, mappings, and components, existing in environments. We will consider the development

history of two DMIs in chapter 3, finding some challenges that arise in the long-term maintenance of these instruments. We will then review the environments used in the development of DMIs (section 4.1), and the most common components of DMIs (section 4.5). Here we will recognize that, although DMIs presented in the research literature are immensly diverse in terms of their design and goals, they are substantially similar in terms of their components and implementation. Considering the importance of evaluation for the development of a validated body of design knowledge, and reflecting on the practical challenges of DMI development, we will argue that in order to maximise the impact of DMI design research and manage the overwhelming burden of DMI maintenance, the research community must develop a shared pool of DMI components from which to assemble and maintain new designs. In section 4.7, we will develop requirements for a library of DMI components that could serve as a communal pool of resources, facilitating novel developments and consolidating the burden of evaluation and maintenance within and across DMI development projects. Considering prior work that addresses the need for such a library (section 4.5.5), we will uncover the challenges hindering the development of a communal library of DMI components (section 4.7), especially the burdens of portability, compatibility, glue-code, and API contract, which together hinder the sustainable development of reusable DMI components and prevent the development of a shared pool of DMI design resources.

The remainder of the discussion will describe Sygaldry, my attempt to address the technical barriers against developing a shared library of DMI components meeting our requirements. I will argue that modern C++ provides the best portability and compatibility, and enables glue-code and boilerplate to be automated, permitting the sustainable development of reusable DMI components. We will examine the design (section 5.1) and implementation (section 5.2) of Sygaldry, and its application to DMI replication, prototyping, maintenance, and evaluation (chapter 6). The library itself can be found online. The most recent commit as of the time of submission of this thesis is located at

`https://github.com/DocSunset/sygaldry/tree/8888ba2c4397cfe74aa1ba7d0c3767adbd04ea8b`

## 1.4 Summary

Digital musical instruments (DMIs) are musical instruments whose medium is digital information. We develop and study them because they provide fertile scope for research into human-computer interaction, novel product markets, and experimental opportunities for research-creation, and because they are exciting, interesting, and gratifying to construct and play. To maximise the impact of our efforts, we aim to develop instruments that are suitable for long-term use. There are numerous factors that make this difficult, especially shifting design requirements and the difficulty of developing maintainable technology. We will consider the components of DMIs, arguing that maintainable and adaptable DMI development will depend on developing a library of reusable components. We will develop a set of requirements for such a library, and describe our attempt to develop a library meeting these requirements.

# Chapter 2

# Dataflow in DMIs

In this chapter, we will develop our unified conceptual framework of digital musical instruments (DMIs) as a dataflow network of interconnected digital materials called endpoints, models, mappings, and components. A DMI is an assemblage of components involving these materials, implemented on one or more platforms. We will elaborate the way in which the dataflow model represents design knowledge and structure, and work towards a unified model of mappings and DMI design.

The term "mapping" is relatively generic, and can be readily applied to any stage that associates one stream of data or representation of information with another different stream or representation. The basic case that usually comes to mind is the mapping from the control parameters of the interface in a DMI to those of its synthesizer, but the term mapping can be applied just as readily to stages within the interface and synthesizer subsystems that associate one data domain to another. Examples might include sensor signal conditioning, gesture modelling, high-level sound parameter mappings, and even basic synthesis blocks such as filters and oscillators can be viewed as kinds of mappings. The framework developed in this chapter arises from the natural expansion of the concept of mappings, finding it in all parts of a DMI's design and applying the vocabulary and mental framework of mappings (e.g. inputs, outputs, source, destination, endpoints, connections, associations, etc.) to describe the structure of DMIs more generally. In order

to accomodate common design approaches that tend not to naturally fit into this mental model, especially systems trained on data, we recognize the relationship of these systems to dataflow mappings, using the notions of implicit associations, throughpoints, models, and cross-domain mappings to naturally integrate all DMI design strategies under one perspective while simultaneously introducing more distinct terminology allowing to better differentiate between different kinds of mappings commonly found in DMIs.

## 2.1 A Review of Mapping Authoring Tools

The conceptual framework developed in this chapter originates from a literature review of mapping authoring tools. We will find that, due to their central role in DMI design and the ambiguity about what constitutes a mapping anyways, this consideration of mappings proves adequate to understand DMI designs broadly, including the aspects of a DMI that are not necessarily always considered as part of the mapping.

An inital pool of literature was formed starting with the special issues on mapping from the journals Organised Sound (issue 2, volume 7) and Computer Music Journal (issue 3, volume 38) as well as the bibliography of the ISIDM [17]. The ISIDM bibliography does not include any papers after 2011, so I additionally included all papers published at the NIME conference from 2012 to 2022 (excluding papers I wrote) as a representative body of research since 2011. In addition, a few papers were included for analysis from my personal bibliography based on my prior familiarity with the literature.

The initial pool was scoured for papers presenting mapping design tools using a combination of keyword search and manual skimming. I identified 62 mapping tools based on 84 papers, including 49 papers presenting a design tool. I read each of these papers and annotated them considering the following questions:

- What are the objects that users are concerned with and manipulate?
- How do users interact with these objects of interest?

Much of the existing literature is occupied with details of implementation, comparing mappings in terms of whether they are explicitly or implicitly defined [18], [19], in terms of their functional mathematical properties [20], or in terms of their structure [21]. While important contributions, these prior works offer relatively little insight into how mappings are actually made, and provide no guidance on how to make mapping design choices to achieve certain aims. At best, they provide vocabulary for discussing, comparing, and understanding mapping designs, but these analytic tools are only a first step towards recognizing the properties of mappings, not determining which properties are useful and in which circumstances.

Van Nort and colleagues [22], [23] in addition to considering the structural and functional properties of mappings, also emphasized the importance of the perceptual qualities of mappings. In prior research by my colleagues and I [6], [24], our preliminary observations suggest that the perceptual view of mappings is most relevant to mapping designers. This emphasis on perceptual considerations is also seen in prior studies observing mapping designers: composers working with the Wekinator in a participatory study by Rebecca Fiebrink reported that they appreciated the way the tool allowed them to focus on gesture and sound [25]; and in a study by Dom Brown and colleagues, most participants described their mappings in terms of spatial and musical metaphor [26].

Based on these results and my own practical experience designing mappings, it is my view that shaping the perceptual influence of the mapping is the final goal of mapping design, and by extension DMI design. Consideration of algorithms, structure, and topology are only practical insofar as they help us to understand the perceptual influence of our mapping design choices. The ideal mapping design tool would allow users to directly specify the desired perceptual results of the mapping, and would reliably and faultlessly achieve their specification. This ideal is unfortunately not achievable today, and realistically never will be. This is mainly because "the desired perceptual results of the mapping", as well as being highly subjective and difficult to rigorously describe, may not be known in advance. Indeed, surprise may be an essential goal of the design, in which case a priori specification of the behavior of the mapping would be especially challenging. Nevertheless,

the end goals of any mapping design necessarily consider the perceptual effects of the mapping; everything else is instrumental (in the sense of "instrumental goals").

Since the perceptual effects of the mapping cannot be directly specified or manipulated, the mapping must be designed in terms of more immediate practical objects. Based on my review, mappings are most often designed in terms of dataflow and/or datasets; that is to say, the perceptual behavior of the mapping is influenced indirectly. Rather than directly specifying the perceptual behavior of the mapping, designers instead manipulate the flow of data and/or the curation of datasets. Design tools can be characterized in terms of their affinity for dataflow and datasets, among other features.

## 2.2 Dataflow Mapping Design Tools

It is no surprise that dataflow-oriented design tools feature prominently in my review, including 23 of 49 tools surveyed. Since the beginning, music technology has tended towards modular construction. From the modules of analog modular synthesizers to the opcodes of the earliest computer music environments, the dominant conceptual model of the field is deeply oriented towards the flow of signals between networks of devices. This carries through to contemporary music technology. The underlying conceptual model of every major computer music environment is still modular, networked, and oriented around the flow of (usually sound) signals through various interconnected functional components. We can term this general conceptual model as the dataflow model. In the dataflow model, a DMI can be viewed as a thing with inputs and outputs. Information goes into one end of the DMI-thing, such as in the form of gestures performed by the player, and information comes out the other end, such as in the form of sound (fig. 2.1).

The dataflow model is very general, and lends itself to various levels of granularity. Introducing slightly more granularity, another common perspective is the three-part view of a DMI as consisting of a controller part, a synthesizer part, and a mapping part that connects the other two [27] (fig. 2.2).

The arrows in this case can alternately be seen as passive associations that merely convey

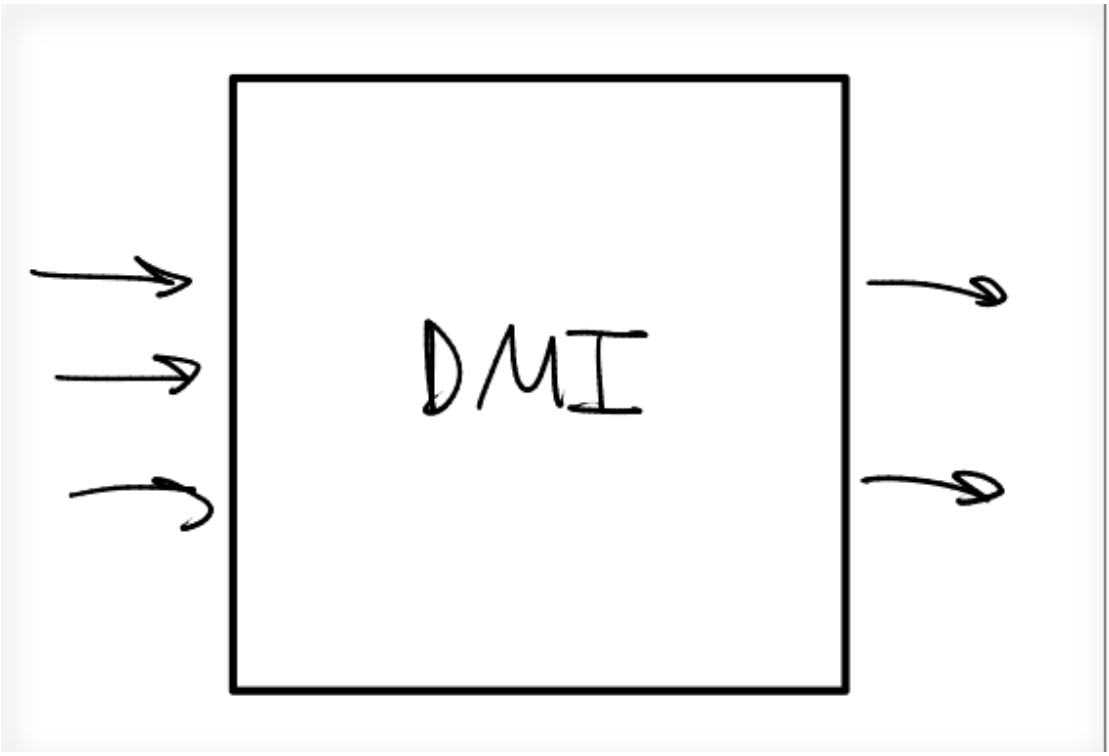


**Fig. 2.1** A DMI viewed as a single assembly: in essence, an intent-to-effect transducer.

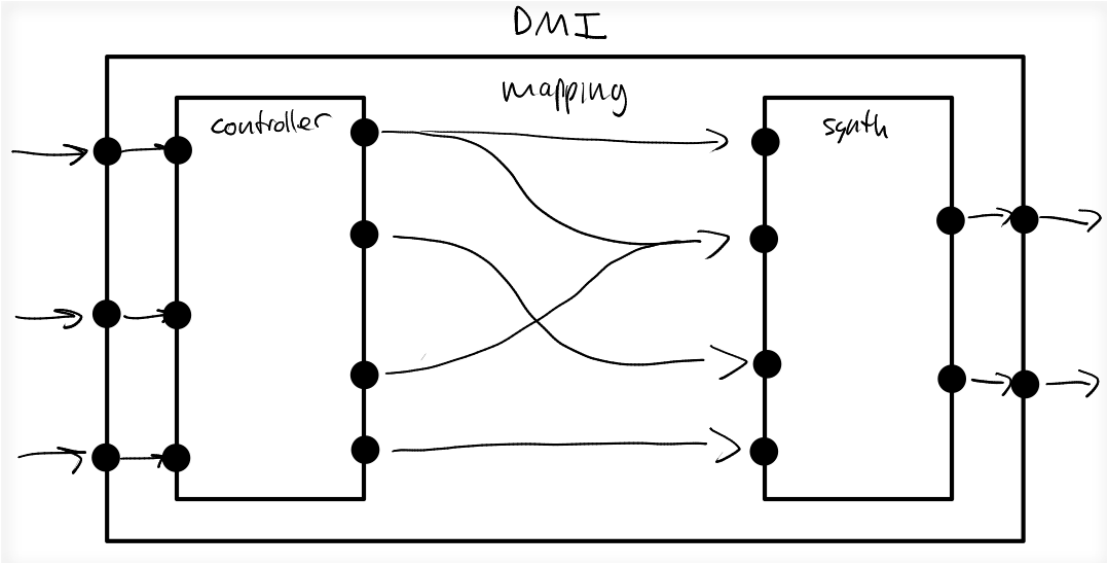


**Fig. 2.2** The common three-part view of a DMI, with a controller component, a synth component, and a mapping between the two ([27]).

data from one point to another, or as active transformations that adapt the data so that it moves from the domain of the point of origin to the range of the point of destination. A more classical dataflow model in the tradition of computer science (e.g. [28]) would tend to adopt the passive view of these connections, but both perspectives are common in the music technology context. For example, mappings in Libmapper correspond to the arrows in this model, and are able to execute generic scripted transformations including time-domain filtering and basic C-like mathematical transformations [29], and MIDI control change mappings in digital audio workstations usually include a linear transformation. On the other hand, cables in Max/MSP and Pure Data are purely passive connections, as are patch cords in most virtual modular synthesizers.

As the granularity of the perspective increases, a DMI can be seen as a more general network of functionally independent interconnected components [30]. In this view, the complexity of the overall system is more evident, and it is easier to conceptualise multiple sources of data besides the performer, feedback paths, and cross-cutting lines of influence, all of which may not fit neatly into the divisions of the three-part view of a DMI (e.g. fig. 2.3). This perspective facilitates examination of DMIs without the limitations of an acoustic causal mental model of an instrument or interactive system, implied by the three part view of a DMI.

At some point, the level of granularity reaches the point where the perspective is dominated by arrows that denote the passive flow of information between primitive operations such as addition, subtraction, multiplication, and so on. This approaches a more classical computer science dataflow model, and is less often discussed or considered in the context of DMI design, except when describing the implementation of algorithms, especially for synthesis and sound processing.

Multiple levels of granularity can be represented simultaneously by drawing boxes around chunks of the dataflow graph, grouping subgraphs by semantic association. If we obfuscate the contents of such a box, we reduce the level of granularity, or equivalently, we increase the level of abstraction.

Considering these features of the dataflow approach, we arrive at the following proposed terminology. In a dataflow model, *components* are the owners of *endpoints*. An endpoint may be

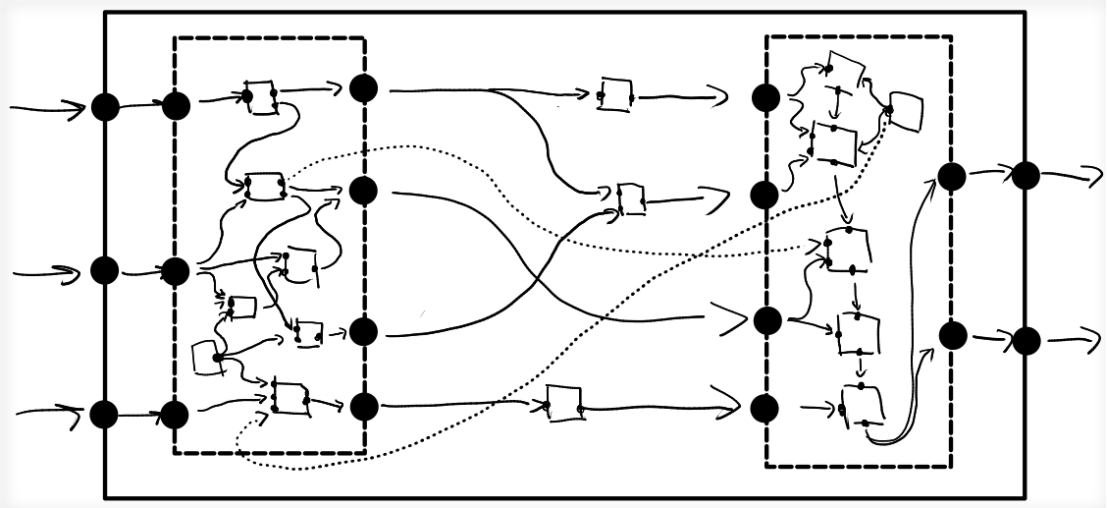

**Fig. 2.3** A representation showing multiple levels of abstraction, highlighting the possibility for independent data sources within high-level assemblies in the instrument, explicitly representing mapping components, and illustrating feedback paths within and across assemblies.

the *source* of a *signal*, or it may be the *destination* where a signal is sent to influence a *parameter* of the component. Equivalently, a source endpoint may be termed an *output*, and a destination endpoint may be termed an *input*, although these terms may introduce ambiguity where mappings are involved. When a signal is routed from a source to a destination, this constitutes a *connection* or an *association* between the two endpoints.

The term *component* refers to a certain chunk of the dataflow network that is grouped together for some reason, and to which certain endpoints naturally belong. We recognize that components are often built from numerous subcomponents that could be seperately identified as having their own meaningful ownership of the internal signals within their parent component. These could in turn be broken down further and futher until all that's left are primitive mathematical operators. When specifically talking about a component that is implemented in terms of subcomponents, we term it as an *assembly*.

As well as endpoints, components, and connections, dataflow mapping designers are often also interested in controlling the *range* of a signal related to an endpoint or the range of a connection, i.e. the minimum and maximum value of an endpoint. Designers often design associations from

the range of one endpoint to the range of another; this linear transformation is often viewed as a part of the connection, as in Webmapper [31] for example. More generally, the designer may be interested in the overall transfer function implemented by a connection. This view of connections as providing necessary transformations to allow one endpoint to be compatible with another is common.

## 2.3 Dataset Mapping Design Tools

Machine learning tools have been used in the design of mappings even before the importance of the design of mappings began to receive emphasis in the literature. An early example is Lee and colleagues' addition of artificial neural networks to an early version of Max (before MSP was added), presented in 1991 [32]. Since then, numerous different models and algorithms have been gainfully employed in the design of mappings, including 19 of the 49 tools surveyed.

In these tools, users main interactions with the mapping are achieved by creating or deriving data that implicitly encode the specification of the mapping. The data may take the form of uni-modal or multi-modal point-wise demonstrations, common in preset interpolators, or uni-modal recordings labelled with classifications, often generated by hand, and multi-modal data recordings, such as those generated by miming gesture along with sound sequences playing in real time [33]. Several projects have also proposed ways of generating datasets in real time without user intervention, leading to systems that partially or fully automatically learn the mapping without supervision [34]–[37].

The datasets, however they are derived, are used to train adaptive systems that model the relationships illustrated by the dataset. Running the models then allows output data in one domain to be generated by inputting data from a different domain, e.g. sound synthesis parameters can be determined from sensor signals.

At a fundamental level, the objects of interest when using these tools are points of data, streams of recorded or generated data, and associations linking data (points or streams) from different domains to form multi-modal datasets. The models themselves may be manually edited

to some extent, but this is done so that the model will more accurately learn the relationships reflected in the dataset.

## 2.4 Higher-order Mapping Design and Mixed Data-flow/-set Approaches

If a first-order mapping associates gesture signals to sound parameters, then a mapping which controls a first-order mapping may be considered a higher-order mapping. This kind of structure is usually called a *convergent* mapping, although this term has also been interpreted to be synonymous with a *many-to-one* or *many-to-many* structure, which is not necessarily higher-order in the sense described [38]. When dataflow- and dataset-oriented approaches are combined, this may also lead to higher-order mappings.

In many cases, the mappings achieved with dataflow tools are affine mappings, and can be represented by an augmented transformation matrix. In such cases, the mapping matrix itself can be treated as a component whose coefficients are parameter endpoints that can be modified in real time; a higher-order dataflow technique. This approach is suggested by de Campo [39] in the form of applying random offsets to matrix coefficients.

On the other hand, multiple matrices can be pre-set and interpolated between to dynamically alter the mapping. In this approach, the mappings themselves are treated as data. They can be associated with data points in a different domain, such as 2D spatial coordinates, and an algorithm such as a preset interpolator can be used to navigate the space of mappings by moving through the space of 2D coordinates. This kind of approach was found in three of the tools surveyed [39]–[41]. Alternatively, mappings themselves can be subjected to evolutionary techniques and subjective breeding, as demonstrated by Mandelis and Husbands [42].

Matrices are also interesting as there are efficient algorithms for learning a matrix transformation based on a dataset, but the matrix itself is also possible to understand for designers. As such, matrices are perhaps unique in their ability to be directly tweaked and specified by the designer or trained from data. Matrices are also conceptually simple and relatively easy to implement, which makes them particularly attractive as mapping primitives. This approach was found in one of the

tools surveyed [40].

Dataset-oriented interactions can be integrated into a dataflow framework by allowing trained models to exist as components in the dataflow graph. This approach was identified in three of the tools surveyed [40], [43], [44], but arguably can be found in more than those three, since many dataset-oriented tools have been implemented as modules for dataflow-oriented languages, notably Max/MSP and Pure Data, e.g. [33], [45].

Many algorithms used in dataset mapping design have free parameters, especially in preset interpolation algorithms. In a dataflow-oriented approach, these parameters could be treated as endpoints and signals could be connected to influence them in real time. In some cases, it may be practical to run a dataset-oriented model in real-time without training it in advance. In such cases, the dataset itself can be treated as a collection of parameters. Depending on the size of the dataset, it might be practical to individually route signals to each point in the dataset, controlling them in real time. This approach was not emphasized in any of the tools surveyed.

It is likely that any given dataset may be too large to treat in a dataflow-oriented approach. However, datasets may be treated as parameter vectors in a higher-order dataset-oriented approach. For example, Visi and colleagues [46] derive a topological representation of datasets recorded from multiple participants and use a common preset interpolation technique to interpolate between datasets, generating new intermediary datasets that are then used to train an adaptive model to follow gestures.

## 2.5 Datasets Viewed as Implicit Endpoints and Associations

In dataflow-oriented mappings, knowledge about the behavior and affordances of devices is encoded in their endpoint representations. The user specifies the mapping by making connections between endpoints. The design is encoded in the connections, and the knowledge implicit in the endpoints. When the endpoints encode a significant amount of knowledge, the connections may be relatively simple; for example, there is no need for a complicated mapping between a breath pressure sensor signal and a physical model of a flute with an air pressure parameter. In case the endpoints encode

very little information, the mapping may need more complexity, such as multiple processing layers, in order to extract, condition, and combine signals so that a meaningful set of endpoints is derived. The knowledge embedded in a dataflow-oriented mapping design is therefore strongly associated with the endpoints in the mapping. This is reflected by the way that endpoints can often be named and described in a dataflow network. That's the filter cutoff control, that's the shake gesture, and so on.

Dataset mapping design tools can be viewed as being at least somewhat dataflow-oriented, as once the dataset is used to train a model, the model is naturally treated as a component through which the user will route signals. But unlike typical components in a dataflow approach, a mapping model trained on data does not have its own set of meaningful endpoints. Its parameters are a reflection of the source domain, and its signals are a reflection of the destination domain. As such, the endpoints of a model trained on data do not encode any knowledge on their own. The trained model could even be viewed as having no endpoints; it is more akin to a single complex connection between the (often high-dimensional) domain and range represented by the signals coming into the model and the parameters it influences. Instead, *the dataset used to train the model* contains knowledge about the design. Presets, gesture recordings, and sound parameter automations have inherent meaning, especially when they are carefully designed by the user. Linking them together to form multi-modal datasets is analogous to making a connection between endpoints in a dataflow model. The data in a dataset-oriented model can thus be viewed as implicit representations of meaningful connections and endpoints that the learning models in these tools are tasked with implementing through their internal machinations.

If this is the case, we should expect that the datasets in dataset-oriented techniques would be associated with meaningful concepts, which is indeed what we observe. In all of the dataset-oriented tools surveyed, the datasets of interest were related to gestures and/or sound parameters that were known to be of interest in advance. The fact that datasets and endpoints encode specific meanings is also shown by their naming conventions. Like endpoints in a dataflow network, datasets can often be labelled in terms of what they are meant to represent, such as the names of

presets or gesture demonstrations.

In preset interpolators, the fact that the dataset is used to derive more meaningful endpoints from raw signals is explicit in the structure of the algorithm. The presets in control space represent meaningful states of the control space signals. These algorithms employ various geometric models of closeness to these control points to generate a vector of weight signals that are then used to mix between the presets in destination space. The vector of weights is literally the realization of a model of meaningful endpoints represented by the control space presets.

An exception to the rule that dataset-oriented models hide their meaningful endpoints internally is found with uni-modal tools such as gaussian mixture models [33]. In these tools, the model is trained to derive a set of meaningful parameters, such as gesture descriptors, from a set of raw inputs, and so is another case where the learning model is explicitly trained to represent meaningful endpoints.

Therefore, in both dataflow- and dataset-oriented mapping design approaches, endpoints (i.e. signals and parameters) may be considered as primary vessels of knowledge and intent in the mapping design, whether they are empirically represented or implicitly encoded in datasets and represented only in the internals of a learning model. Connections between endpoints, whether explicit connections or multi-modal associations between datasets, are also an essential component of the mapping design, since they specify how the potential behaviors and affordances encoded in the endpoints should be related. This provides one possible route by which dataflow- and dataset- oriented mapping design approaches may be unified under a dataflow-oriented conceptual framework.

However, higher-order mapping approaches blur the lines between endpoints and connections. For example, a transformation matrix may be thought of as a connection from $N$ signals to $M$ parameters. It may also be viewed as a collection of data modelling something that can be learned by a training model, i.e. as an implicit representation of endpoints and their connections. The same matrix could also be viewed as an endpoint unto itself, a single $N * M$ dimensional point to be modulated by some other process. The ambiguity here arises due to the fact that mappings

themselves can be viewed as having endpoints, or as being defined by datasets that implicitly model those endpoints.

From these considerations, it appears that if datasets are viewed as implicit models of endpoints and connections, then endpoints and connections are conceptually sufficient objects of interest that can fully model mapping designs, and thus a dataflow-oriented conceptual model is an appropriate framework under which to model DMIs. Notably, components in a dataflow approach seem to play a similar role to datasets; both components and datasets group together "endpoints", and implement hidden layers of processing to derive a set of outputs from their inputs. Components, like datasets, hide internal connections that associate inputs to outputs. Unlike datasets, the endpoints of components are generally semantically meaningful unto themselves, whereas datasets tend to hide some or all of their meaningful endpoints internally, and connect directly to externally meaningful endpoints. Datasets and components can thus be viewed as conceptually related, or even instances of the same conceptual object—datasets are a kind of component. They group together endpoints, and implicitly connect them. They are black boxes whose internal processes are inscrutable, or at a lower level of detail than is called for by the designer.

The main feature that distinguishes datasets from typical dataflow components is that a dataset alone is an incomplete component; it requires an algorithm to analyse or train on the dataset in order to implement an actual component through which data can flow. So a dataset, plus an algorithm that can learn from it, realizes an actual component in a dataflow network. This adds an additional layer of endpoints, those reflecting the input and/or output domain in the uni- or multi-modal dataset. This external layer of endpoints is semantically neutral. It simply reflects the domains present in the dataset.

The observation that models trained on datasets tend to have external endpoints that are semantically neutral, merely reflecting the semantics of the signals connected through them, is important. This quality is also true of many of the primitive mapping techniques commonly found in dataflow-oriented environments, such as linear transformations. We term these semantically neutral endpoints as *throughpoints*.

## 2.6 Kinds of Mappings

Consider the mappings that appear in and around a DMI. The concept of mapping is generic and flexible, arguably so much so that it fails to meaningfully signify anything in particular. Indeed, the entire framework developed in this chapter eventually comes to see everything as a kind of mapping. Components are mappings from their inputs to their outputs. Connections are mappings from their sources to their destinations. Datasets implicitly encode mappings from expected inputs to expected outputs. From this perspective, the entire framework merely provides terminology to disambiguate different types of mappings. In this section we introduce further terminology to distinguish mappings not only by the style of their implementation, but also in terms of their purpose and relationship with the designer of a DMI.

It is not unreasonable to consider the mappings e.g. from the performer's intention to their physical movement, from the physical movement to the actuation of sensors, from the actuation of sensors to the transduction of electrical signals, from the electrical signals to digital, and so on until digital signals are converted back to electrical, electrical to acoustic, acoustic to neurological, and from neurological signals another mapping to meaning in the mind of the listener. The connections also extend back from the audience to the performer, via visible light and other signals, which are mapped in the performer's brain to an interpretation of the audience's affective states, which may in turn influence the performer's intention. All of these mappings just described are outside of the purview and influence of the mapping designer. We could term them inherent mappings or external mappings. They are largely outside the scope of the designer's considerations, or at least outside the scope of their practical intervention, yet the flexibility of the term mapping enables it to reasonably apply to them. It is useful to be able to differentiate these external mappings from those that designers directly influence.

Within the designer's scope, there are also numerous mappings, in the broadest sense. Signals from the sensors must be conditioned, calibrated, fused, and analysed, perhaps producing gestural signals that can be further conditioned, calibrated, fused, and analysed to produce other signals

of interest. Rather than being mapped directly to synthesis parameters, there might be on the synthesizer's side further layers of conditioning, calibration, fusion, and analysis used to produce higher-level sound parameters. In the middle, a simpler layer of mappings might bridge the gap from high-level gesture signals to high-level synthesis parameters, crossing from the domain of movement to that of sound. This is the layered structure of mappings described by Hunt et al [47], proposed as a way of simplifying the design process, and potentially providing for portable mappings that could generalise to many instruments.

In order to disambiguate between all of these mappings, it is useful to distinguish the characteristics that may allow us to differentiate them, and to adopt terminology that enables us to speak more precisely about the kinds of mappings we are concerned with in any given moment. As well as distinguishing the external or inherent mappings outside of the influence of the digital luthier, we can recognize that the mappings giving rise to intermediate layers, in Hunt et al's discussion, are notably different in character from the innermost layer of mappings. Let us term the latter as *cross-domain mappings*, and the layers of conditioning, calibration, fusion, analysis, etc. as modelling mappings, or simply *models*, giving rise to the representation of novel meaningful signals and behaviors.

## 2.7 Models and Cross-Domain Mappings

Since models and cross domain mappings are part of the purview of the DMI designer, it is worthwhile to futher elaborate their properties and distinguishing characteristics.

Models are data processes that consider signals from one domain, and derive independently meaningful signals in a new (usually related) domain. Examples include algorithms such as sensor fusion, feature extraction, and high-level parameter mappings. Models tend to belong to the controller or the synthesizer, or to a particular combination of sensors or sound processors, and they have endpoints with strong semantic labels. For example, the sensor fusion model for a magnetic-inertial measurement unit is strongly associated with that particular group of sensors, and it has strict requirements for the format of the input data (e.g. accelerometer vector in meters

per second squared, gyroscope vector in radians per second, normalised magnetometer vector, and a timestamp) and outputs specific derived signals (e.g. orientation, acceleration due to motion, estimated sensor velocity, etc.).

As well as empirical algorithms such as sensor fusion, models can also be derived generatively from datasets, as in e.g. a neural network that classifies sensor data into known gestures based on demonstrations in a dataset. In this example, the outputs of the model have strong independent meaning (e.g. the probability estimate for each class of gesture), although the inputs to the model may be viewed as throughpoints reflecting the source domain. The defining feature of models is that they impart additional meaning on the data flowing through them; they encode external a-priori semantics and enrich the semantic vocabulary of the dataflow network with new independently meaningful endpoints.

Cross-domain mappings are data processes that take signals from one specific domain and adapt them to influence endpoints in another specific domain. Cross-domain mappings are distinct from models because they do not have independent endpoints. Instead, they merely transform data from the source so that it can be sent to the destination. Although this does enrich the dataflow network with additional meaning (i.e. "that source influences that destination in that manner"), they do not introduce new strongly labelled endpoints into the network. The independent source and destination domains that a cross-domain mapping connects have their own specific semantic characteristics and labels, but the cross-domain mapping itself does not. Examples include simple linear transformations (arguably the most common kind of mapping), preset-interpolation algorithms mapping directly from control signals to synthesis parameters, and other such connective algorithms. Generative cross-domain mappings may internally represent external a-priori semantics, but they do not generally expose these semantics to the overall network.

Models and cross-domain mappings are coupled with the components that they are associated with. Because models, by definition, have their own layer of semantically independent endpoints, they are only coupled with components on one side; for example, a magnetic-inertial sensor fusion model is only coupled with the magnetic-inertial sensors, and a high-level synthesis parameter

model is only coupled with the low-level synthesis components it influences. In contrast, cross-domain mappings are coupled with components on both sides. This is the main distinction between these two kinds of connective components, and the effect of this difference is that models tend to be more generalisable and portable across different DMI designs than cross-domain mappings, which tend to be highly specific to a particular design.

In the conceptual framework of meaningful endpoints with connection between them, models are characterized by giving rise to new meaningful endpoints, whereas cross-domain mappings merely associate existing endpoints. It could be said that models have one set of throughpoints, for either their sources *or* their destinations, whereas cross-domain mappings have throughpoints for their sources *and* their destinations.

Because of this, it is apparent that models exhibit greater portability than cross-domain mappings; whereas Hunt et al [47] emphasize the possiblity of developing portable cross-domain mappings in the middle of a layered structure, I argue that the modelling layers themselves are more readily and usefully portable. Models are coupled only to the specificities of their inputs *or* outputs, whereas cross-domain mappings are coupled to both their inputs *and* their outputs. Models are thus more likely to be useful in the context of multiple musician+instrument ecologies. For example, gesture models that derive new signals from MIMU data may be useful to any instrument with a MIMU, whereas cross-domain mappings from those gestures to synthesis parameters are only potentially useful if an instrument uses a MIMU *and* the same synthesizer, or one exposing comparable high-level synthesis parameters.

On the other hand, recognizing the diversity and complexity of the environment, I do not advocate that high-level signals produced by models should *replace* or otherwise reduce access to the least-conditioned raw data.

As described by Thorn and Sha [48], "signal processing affords continuous revision of instrument morphology without requiring termination in a conceptual schema demanded under the scientific desideratum of universal validity." They describe their approach to the design and construction of "instruments of articulation" emphasizing a closed loop of sensorimotor engagement with a system

leading to changes in the code that influence sensorimotor experience, a nomadic science where the development of the system responds to the conditions the system creates, through pure signal processing that doesn't necessarily encode a priori categories or semantic interpretations, but rather responds interactively to events, conditions those events, and produces results from which meaning emerges rather than enforcing assumed meaning through its design. Thorn and Sha's discussion constrasts vividly with my own emphasis and aspirations toward generality, universal usefulness, and interpretable networks marked with semantically laden endpoints. Let it be understood that these approaches are not mutually exclusive. In particular, so long as the inner mechanisms of an instrument system are exposed and accessible, Thorn and Sha's process of canalization remains possible, and within the conceptual framework we are exposing, there is no reason why the meaning inherent in an endpoint must necessarily submit itself to ready interpretation, categorization, or universal fixed understanding. Endpoints can also be semantically shallow, in the sense described by Thorn and Sha.

> "Through trial and error, salient features are designated within a code that constructs those features. Choices are made about what needs to be detected, this band of energy or that type of attack. The media itself is constructed by the actual mathematical choices in the analysis of the feature vectors. ... In building instruments that are radically open for players to rupture pre-conceived semantics through their improvised, alinguistic and articulating actions, we are advocating a view of intentionality that is semantically shallow and not based on the notion of intentional being connected to a goal known in advance."

## 2.8 Conceptual and Implementation Components

Up until now, we have discussed DMIs from a conceptual point of view, and the components we have described are conceptual components. The conceptual components model the functionality of the DMI design.

Actual DMIs as they exist in the world must be physically implemented, using various electronic parts, computing hardware, and software programming languages. We term the physical parts and software used to implement a DMI as its implementation components. This includes functional and environmental hardware and software components.

We term the physical parts used in a DMI's construction as its hardware components. Some hardware components, such as sensors, directly correspond to conceptual components. For example, a TDK InvenSense ICM20948 magnetic-inertial measurement unit is a physical hardware component that directly corresponds to a conceptual magnetic-inertial sensor component. We call these "functional hardware components", because they provide some of the main conceptual functionality of the instrument.

Other hardware components, such as microcontroller units, laptop computers, and system-on-chip processors, do not correspond to any conceptual component, but instead provide a computational context in which data processing algorithms can be executed. This context is termed as a software environment or software platform, and we call the hardware components on which the software environment is executed the "environmental hardware components" or the "hardware platform components". Different processor components provide different kinds of software environments, ranging from bare metal freestanding embedded C++ firmware to hosted dynamic scripting languages like Max/MSP and Pure Data. The conceptual components of a DMI access, generate, process, and output streams of data. These conceptual operations are physically realised by the execution of what we term "functional software components" in the context of a software environment; we call them "functional" in the sense that they provide the main conceptual functionality of the instrument, not because they are necessarily "functional" in the sense of "functional programming language".

In addition to functional software components associated with the conceptual components of a DMI, additional software components are required to interact with the software environment. Each environment has its own specific application programming interface (API), and hardware environments such as microcontrollers have unique hardware interfaces such as memory-mapped

peripherals that must be interacted with to access the functionality of the hardware. We term the software written to interact with the environment as "environmental software components" or "software platform components". Environmental software components are essential for the implementation of any DMI, and require as much consideration and maintenance as any other software component.

Functional software components are, in principle, related to conceptual DMI components and should be able to be implemented in a portable environment-independent manner, so that they can be used whenever their related hardware, if any, is available. In contrast, environmental software components by definition interact with environment-specific interfaces, and are thus inherently non-portable.

Taken together, we view the physical implementation of a DMI as consisting of a collection of connected hardware components, including environmental hardware components providing software environments and functional hardware components corresponding to conceptual components of the DMI. Along with hardware components, a DMI also requires functional software components that realise the conceptual dataflow of the DMI design and environmental software components that interact with the software environment to provide the necessary underlying functionality to implement the instrument.

Section 4.1 will review common implementation environments in more depth.

## 2.9 Summary

Based on a review of mapping authoring tools presented in the literature, we developed a robust terminology for describing the design of DMIs as dataflow networks. Considering the objects of interest manipulated by mapping designs in these tools, we segregate mapping design tools into those that adopt an approach based on dataflow, and those that adopt an approach based on datasets.

In the dataflow approach, we find DMIs as being composed of components that expose input and output endpoints that are connected in a network through which information moves and

is transformed. We note that components can be seen as assemblies of subcomponents, where each component or subcomponent usefully groups together its endpoints and abstracts away the internal processes that derive the component's outputs from its inputs.

In the dataset approach, we find mappings that are derived from pools of data, such as presets and data recordings. The data and their labels or multi-modal associations are the main objects of interest, and a variety of algorithms are trained on this data so that inputs can be associated to outputs.

I emphasize that dataset-oriented approaches can be integrated naturally into the dataflow conceptual framework in two ways. First, we can recognize the equivalent role played by dataflow components and datasets. Both encode strong semantic information, often with meaningful names or labels. Thus datasets can be viewed as implicitly encoding semantically strong endpoints, and abstracting away the details about how their endpoints are associated, just like dataflow components. Then, an appropriate algorithm trained on the dataset is used to implement these abstracted away details of association, so that the algorithm-dataset combination implements a component that realizes the implicit endpoints encoded in the dataset, and their possible association in the case of multi-modal datasets. We recognize that this implementation of a component by an algorithm-dataset combination creates layers of semantically neutral endpoints that merely reflect the input and/or output domains found in the dataset; we term these as throughpoints.

Building on prior conceptual models of mapping networks as having layers [47], we introduce a distinction between mappings that model particular signals and behaviors with distinct interpretation and semantics compared to those that merely connect between meaningful endpoints across different semantic domains. We propose to distinguish models and cross-domain mappings by their throughpoints. Models have one layer of throughpoints, either taking a certain reflected set of inputs and deriving a new layer of meaningful outputs (e.g. gesture feature modelling) or taking a new layer of meaningful inputs and connecting them to a certain reflected set of outputs (e.g. high-level synthesis parameters). Cross-domain mappings have only throughpoints, and do not introduce their own explicit meaningful endpoints. We highlight the important practical dif-

ference between models and cross-domain mappings: models have greater scope for portability, being coupled to only their inputs or their outputs, whereas cross-domain mappings are coupled to both, and are thus highly specific.

Finally, we highlight that as well as conceptual components that describe the meaningful functionality of a DMI, designers must also consider the components of implementation, the hardware and software platforms that are related to the conceptual components and enable their practical implementation. We propose to distinguish between functional hardware and software components, which are those that directly relate to the conceptual components of a DMI design, and the environmental hardware and software components, which are those that support the implementation and interaction with the functional components, such as operating system APIs and computing hardware.

In this chapter, I contribute extensions to the predominant dataflow conceptual model of DMI development, based on an extensive literature review, that bring together the two main approaches used in DMI design and implementation by recognizing the conceptual correspondance between datasets and dataflow components as endpoint encoders. I introduce the concepts of throughpoints, highlighting how this concept illustrates when mappings are most portable between instruments. I introduce terminology to bridge the gap between conceptual models of DMIs and their actual implementation.

# Chapter 3

# Case Studies in DMI Development and Maintenance

In this chapter, we will examine the development history of two instruments familiar to the author: the T-Stick, and the mubone. These instruments will serve as case studies for the long-term maintenance of DMIs, especially in the context of research, and as examples of the application of the conceptual framework described in chapter 2.

## 3.1 The T-Stick

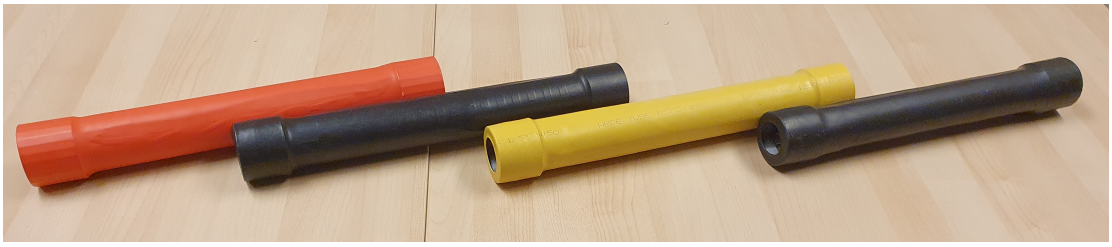

**Fig. 3.1** Several Sopranino T-Sticks. Image used under MIT license, copyright 2023 IDMIL.

The T-Stick [49], [50] was originally developed as part of the CIRMMT/McGill Digital Orchestra project [51] by Joseph Malloch and his collaborators. The design is conceived of as a

family or consort, with T-Sticks of various length ranging from the diminutive sopranino to the imposing bass. Regardless of their size, T-Sticks are characterised by having a multi-touch array of capacitive sensors on one side of the ABS structural substrate, a pressure sensor on the other side, and some combination of inertial motion sensors.

The T-Stick was designed to have low gestural specificity. That is, it was intentionally conceived so that it is not immediately obvious how the instrument should be played or what sound it should make, compared to for example a wind controller, MIDI keyboard, or drum pad. These interfaces have clear acoustic counterparts from which their gestural vocabulary is drawn. The T-Stick was originally conceived in terms of a similar acoustically-informed virtuosic performance paradigm, with an emphasis on long-term engagement, the development of performer skill, and a signal-level interaction semantic [52]. However, due to its ambiguous shape and flexible set of sensors, it's intentionally vague how the T-Stick should be played. It can be struck, brushed, shaken, swung—the gestural vocabulary of the T-Stick is always in development, being invented and rediscovered by each new player and composer for the instrument.

### 3.1.1 Dataflow of the T-Stick

The T-Stick is typically treated as an interface, meaning that there is no canonical sound or media subassembly that is characteristic of the instrument. As such, the dataflow of the T-Stick is well described by the three-part view of a DMI, with the mapping and synthesizer generally varying between, and sometimes within, pieces for the instrument. The implementation of the T-Stick interface as used in my prior research is shown in fig. 3.2 as an illustration of some of the characteristics common to most T-Sticks. With the addition of a mapping and sound synthesizer, the instrument would be complete; each T-Stick composer and performer typically invents their own mappings and brings their own synthesis approaches to the instrument.

The figure represents just the interface, with example gesture modelling. The two software platforms, the MCU and Max/MSP respectively, are grouped by black dashed rectangles. The subsequent layers of signal processing are grouped by grey dotted rectangles. Throughpoints are

represented by open circles, whereas semantically distinct endpoints are shown as filled in dots. The sequential coupling of each stage to its predecessor is characteristic of the layered structure of the dataflow.

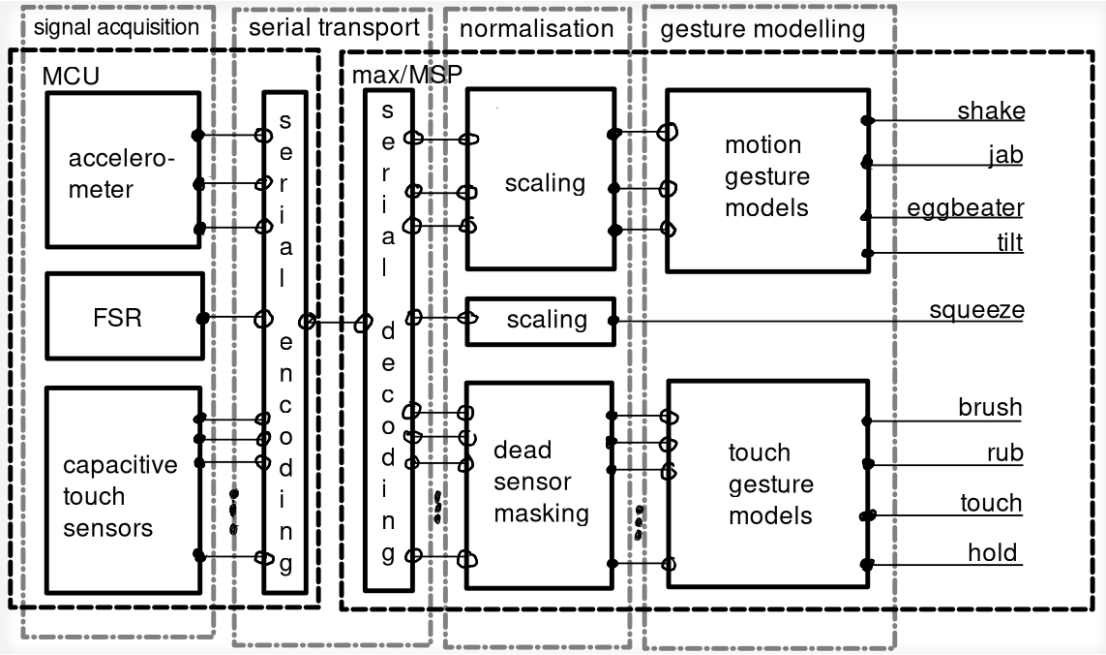


**Fig. 3.2** Block diagram of the T-Stick interface.

### 3.1.2 Maintaining the T-Stick

The earliest T-Sticks were constructed by Malloch and collaborators for the Digital Orchestra project in 2006. In the ensuing years, many more T-Sticks would be constructed, mainly by graduate students in music technology at McGill. These instruments were basically built as pedagogical probes, and so the goals of their construction were often not aligned with those of performance and long-term upkeep. They were generally not maintained, and so tended towards obsolescence and unusability. In his effort to refurbish some of these instruments [53], Alex Nieva encountered obsolescent firmware, electrical issues, and various unique arrangements of sensors, microcontrollers, and hardware connections. He refurbished numerous old T-Sticks, bringing them to a mutually compatible and functional state as part of his research, including updated Arduino firmware for

all of these instruments.

At this time, Nieva's working T-Sticks made use of AVR microcontrollers, and transmitted gestural data from the T-Stick to a PC over a proprietary USB serial protocol. Data were received by a Max/MSP program that performed a series of mappings to normalise the raw sensor data and extract various gestural features. Data were then exposed using libmapper protocol [54] for subsequent mapping to sound synthesis processes. Figure 3.2 above describes one of these T-Sticks.

Edu Meneses and others at IDMIL would take over maintenance of the T-Stick from Nieva. Their efforts would see the instrument ported from Arduino-brand microcontroller platforms to the wireless ESP8266 and later ESP32 microcontrollers, enabling wireless interaction with the instrument, updating the inertial sensors to include magnetic and angular-rate sensors, and introducing a web-based configuration system served by the microcontroller. Much of this work was consolidated by Meneses and his collaborators in the Puara library [55], and has been used in other projects as well as the T-Stick.

The port from AVR to ESP32 microcontrollers involved rewriting large portions of the T-Stick firmware, as well as introducing new software components. About 1/5 software components comprising the AVR firmware were maintained in the port to ESP32, for which 6 new software components were written.

The new features introduced by Meneses et al during this time [55] provide important practical and quality of life improvements to T-Stick users. However, updates during this time did not address the reliability of the manufacturing methods of the instrument, and the migration to a new microcontroller platform and newer sensors was undertaken without consideration for backwards compatibility or continued maintenance of older T-Sticks. Newer T-Sticks have continued to be subject to electrical issues in their fabrication. Most recent maintenance efforts, led by Albert-Ngabo Niyonsenga [56], [57] have worked to elevate the hardware maintainability of the instrument.

Niyonsenga's efforts have also included a port from ESP32 to ESP32s3. Unlike the earlier port from AVR, the port to ESP32s3 has maintained the vast majority of the firmware software components thanks to the strong upstream portability design of the ESP-IDF library that is used

to interact with hardware on ESP32 and related hardware platforms.

Unfortunately, the generation of T-Sticks refurbished by Nieva now languish unused and unmaintained. Although, in principle, these T-Sticks should still be usable with Nieva's Max/MSP drivers, they are generally never powered up. This may be attributed to a variety of factors. For example, the AVR-based sticks are wired, rather than wireless, and lack the newer sensors and other quality of life improvements afforded to newer T-Sticks. There are also numerous incompatibilities between the ESP- and AVR-based T-Sticks. The libmapper namespaces are different, the sampling rate of sensors is generally different, this in turn means that gesture feature models are not drop-in compatible, which has lead to two subtly different implementations of the T-Stick gestures.

In the current maintained T-Stick firmware, glue code is a significant problem. The firmware includes communication protocol bindings to OSC and libmapper, as well as a web-based interface for controlling several configuration parameters such as the destination address and port for OSC messages. In order to add a new endpoint, such as a novel gestural feature for example, it is necessary to manually add approximately 11 lines of code to the main implementation file. While this is not catastrophic, the lines are spread out drastically over the entire implementation file, and it is troublesome and error prone to remember all the places where glue needs to be added. Approximately 343 lines of code in the main implementation file (40% of the main file, and 24% of the whole implementation) is dedicated only to glue code. This represents a substantial maintenance burden. As we envisage adding additional control protocols such as MIDI2.0 and OSCQuery to future versions of the firmware, it is obvious that the current glue situation could become even more problematic.

### 3.1.3 Maintenance Challenges with the T-Stick

The maintenance of the T-Stick illustrates several challenges that may arise in the long-term maintenance of a DMI: we see that the purpose of fabricating an instrument is not always in line with long term maintenance, movement of personnel makes it difficult to maintain support for

instruments, hardware porting can incur significant firmware development costs, and glue code causes some maintenance difficulties.

Instruments are often implemented for reasons other than long-term maintainability. In the T-Stick's almost two decades, this has meant that many instruments have been fabricated that were not resilient to the basic requirements of performance. It is unknown how many T-Sticks have fallen in to such disrepair as to never be recovered, but it is certain that such cases have occurred. Unfortunately, I confess, I too have constructed at least one T-Stick that, after replacing the touch sensor with several different arrays testing various alternative fabrication strategies, now sits in pieces unusable on my desk. While this, and other lost T-Sticks, are sadly in no shape for music performance, this is a natural consequence of their origins. Many T-Sticks are made to explore other kinds of research. Although this inevitably leads to unrecoverable maintenance situations, such as the example given, it has implications for the design of the instrument's hardware and software components. The more modular and amenable to change the design is, the more likely any given instrument may adapt to various research probings without the instrument in question falling into disrepair.

Particularly in the context of a research lab, the departure of members of the maintenance team, such as when graduating from their degree, raises significant challenges for long-term maintenance. When Nieva graduated, the usability of T-Sticks refurbished by him dropped significantly for no other reason than because he was no longer readily accessible to offer support. Documentation can help to address this kind of issue, ideally enabling the specific expertise and knowledge of one maintainer to be transferred to the next. Pedagogical practices, such as apprenticeship, may also work to this effect. Regardless, the departure of an expert necessarily reduces ease of maintenance, if for no other reason than because there are now fewer experts from which to draw support. Furthermore, it has been my experience that documentation and apprenticeship can only go so far. Something is always lost when an expert leaves.

Hardware porting poses a significant challenge to firmware development. The port from AVR to ESP32 incurred rewriting of most of the T-Stick firmware, constituting a large repetition of

development effort, and also introducing a backwards incompatible break in the development history that has likely contributed to the lack of ongoing maintenance for AVR-based T-Sticks and their driver software and gesture analysis implementations. The later port to ESP32s3 was comparatively much more smooth, thanks to the strong upstream support for portability between ESP32-family microcontrollers, however, as the hardware ecosystem continues to develop, hardware porting remains a concern in case the instrument should ever need to be ported to a different family of microcontrollers.

Finally, glue code is currently a burden in the ongoing maintenance of the T-Stick firmware. Due to its necessity, and despite that it only makes the functionality of the instrument accessible over different communication protocols rather than adding meaningful functionality per se, glue code accounts for a significant amount of the firmware implementation. Because this glue is spread out over the implementation, and because it grows quadratically in the number of communication protocols and endpoints to be exposed over them, the long-term sustainability of continuing to manually implement glue is dubious. As is, glue code causes significant annoyance for T-Stick firmware maintainers.

## 3.2 The Mubone

The mubone [1] is an instrument family, design space, and artistic practice based on an orientation sensor and handheld controller used with a novel spatial gestural granular sound processor. I have developed the mubone with trombone player and movement artist Kalun Leung since 2019.

The trombone is a common platform for augmentation and electroacoustic exploration. Prior work has variously appropriated the trombone as a speaker resonator and MIDI slider [58], self-actuating feedback instrument [59], and augmented instrument [60], [61]. The mubone, as a form of augmented trombone, is most directly related to other trombone augmentations; trombone expertise is the foundation of performance with these instruments. Farwell [61] developed an ultrasonic slide position sensor, actuator, and loudspeaker mute for the trombone. Lemouton and colleagues [60] use an infrared laser to measure slide position. In our work, although we

**Fig. 3.3** The mubone in performance. Image from West and Leung [1].

have attempted to measure slide position, the main focus has been on using the orientation of the instrument rather than the position of the slide. We were also inspired by the handle of the sackbut (a predecessor of the trombone) and the double slide controller of Tomás Henriques [62] in the design of our handheld controllers.

Our granular synthesis environment, described below, draws on the established techniques of granular synthesis [63], especially standout implementations such as GrainTrain [64], Borderlands [65], and Nathan Weitzner's 3D granular program[1]. Although we did not initially consider them in our literature review, we realized later that our granular environment is also closely related to concatenative corpus-based synthesis [66], gesture following [67], and real-time adaptive approaches to mapping design [68]. The approach we have taken with the mubone is notable for the performative strategy used to build a corpus and "train" the synthesis environment about gesture and mapping. Sound, movement, and mapping are all recorded in real time as part of a performance.

Just as the concept of "trombone" has flexibility in terms of size (e.g. bass, tenor, soprano), material (brass, plastic, smartphone [69]), and interaction modality (slide, valves, touchscreen), "mubone" is meant to encompass a family of instruments or a space of possible instruments with a shared underlying identity somewhere in their midst. The conceptual approach of developing an instrument family rather than an individual instrument encourages experimentation and exploration, and recognizes the underlying goal of inquiry in the practice of developing mubones. This is key: we have aimed not to make *The Mubone*, but rather to make mubones, and to wonder what makes each one a mubone despite their differences. Nevertheless, we have also aimed to make instruments that are amenable to music performance. Leung has developed a handful of compositions that have been performed repeatedly, as well as an improvisatory practice with his mubone. Maintenance has thus been an essential underlying concern, necessary to facilitate this ongoing professional musical engagement with the instrument.

Over the course of its development, each mubone we've constructed has always had an orientation sensor and a handheld controller. The orientation sensor has remained fairly stable,

[1] `https://youtu.be/HyNhEc4VYmk`. Accessed 2024-09-14.

consisting of a MIMU sending data wirelessly to a main sound processing personal computer. The handheld controller, in contrast, has gone through numerous iterations, which will be described below. These two control devices are mapped to the parameters of the mubone granular synthesizer (mugranular), enabling the mubone's characteristic interaction paradigm where sounds are recorded into the granular corpus along with a directional vector reflecting where the orientation sensor is pointed at the time the sound is made so that, later, the current pointing direction is used to retrieve grains from the corpus, creating the effect that sounds are "placed" in a certain direction and then replayed by pointing there again. The orientation sensor is most often placed on a trombone, but Leung has collaborated with numerous other musicians and explored using the mubone system of augmentations with inputs ranging from the voice to flute to saxophone and so on. In these explorations, mubone is seen more as a technologically enabled artistic practice than a particular instrument or device. Images and videos of the the mubone in performance can be found online at `https://www.kalunleung.ca/projects/mubone`, and further documentation is provided in the 2022 paper on the instrument [1].

### 3.2.1 Dataflow of the Mubone

The structure of mubone system has remained fairly stable since 2022. A graphical depiction of dataflow through the system is shown, at a high level, in fig. 3.4 and fig. 3.5. Adaptations are often made to the mapping for individual compositions, particularly in the way that the parameters of the sound granulation process are varied over time. This graphical depiction of the instrument represents the most recent iteration of the controllers, which is used by Leung in performance.

#### The Orientation Sensor

The orientation sensor is used to estimate the pose of the mubone with respect to a global coordinate system. The data from the MIMU is read, calibrated, and fused by a microcontroller and streamed to a nearby laptop using Open Sound Control (OSC). We use the sensor fusion algorithm described by Sebastian Madgwick [70], based on the complementary filter introduced by Robert

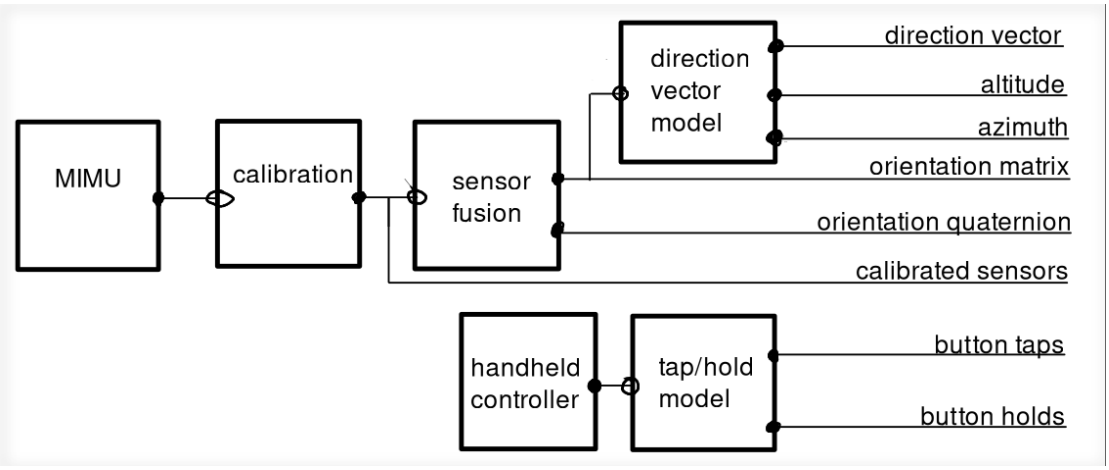


**Fig. 3.4** Dataflow of the mubone controllers.

Mahoney [71], to derive the orientation of the instrument from the sensor's measurements. The orientation is subsequently examined to derive a vector representing the direction in which the mubone is pointed. The rotation (i.e. orientation) of the instrument is given by the sensor fusion algorithm in terms of a unit quaternion:

$$q = a + bi + cj + dk$$
$$\text{with } ||q|| = \sqrt{a^2 + b^2 + c^2 + d^2} = 1$$
$$\text{where } i^2 = j^2 = k^2 = ijk = -1$$

This is not a convenient representation, so the quaternion is converted into a rotation matrix $R$:

$$R = \begin{bmatrix} a^2 + b^2 - c^2 - d^2 & 2bc - 2ad & 2bd + 2ac \\ 2bc + 2ad & a^2 - b^2 + c^2 - d^2 & 2cd - 2ab \\ 2bd - 2ac & 2cd + 2ab & a^2 - b^2 - c^2 + d^2 \end{bmatrix}$$

The rotation matrix represents the way the basis vectors of the global coordinate space are

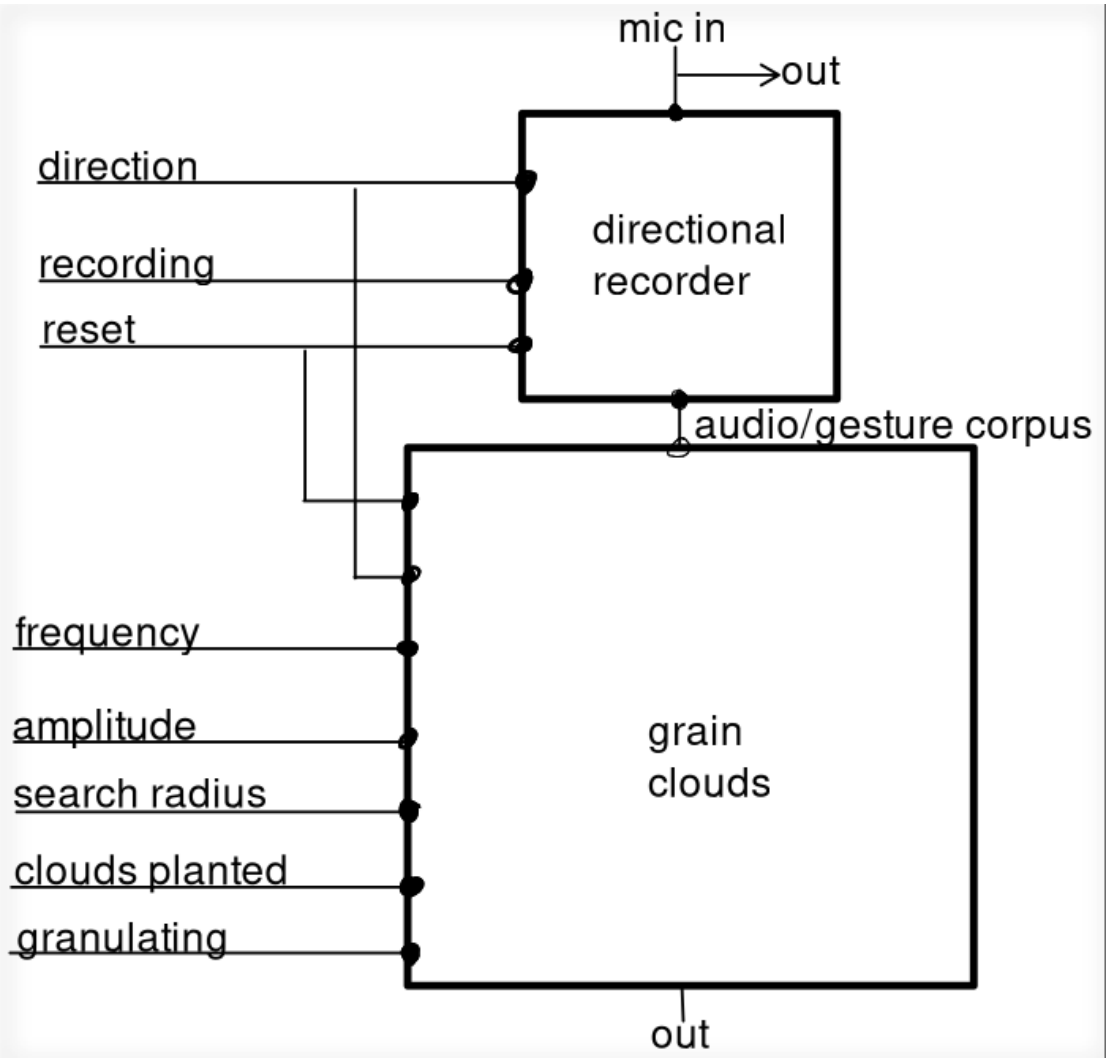


**Fig. 3.5** Dataflow of mugranular.

changed by the rotation. The first column corresponds to the x-axis, the second to the y-axis, and the third to the z-axis. If there is no rotation, the matrix is an identity matrix, and each column corresponds to the unit basis vector for that axis. When there is a rotation, each column represents the unit vector where the basis vectors are moved to by the rotation. Put another way, each column of the rotation matrix represents one of the basis vectors of the mubone's local coordinate space, as expressed in the global coordinate frame.

From the rotation matrix it is easy to determine the vector representing the direction in which the mubone is pointed. In our work we align the global coordinate axes so that positive x points to the right, positive y towards the audience, and positive z towards the sky. We consider the mubone to have zero rotation when held naturally, pointing towards the audience, with the slide parallel to the ground. This alignment implies that the y-axis of the mubone coordinate system is aligned with the slide, so the second column of the rotation matrix is a unit vector aligned with the slide, i.e. the direction the instrument is pointing.

We alternately represent the pointing direction of the instrument as a vector $v = (x, y, z) = (R_{1,2}, R_{2,2}, R_{3,2})$ (the second column of the rotation matrix) or as a pair of angles derived from the vector (i.e.latitude $= atan2(y, x)$ and longitude $= arcsin(z)$). Latitude and longitude are appropriate since the direction vector is a unit vector, and can thus be considered to represent a unique location on the surface of a sphere. Alternatively, latitude can be called altitude and longitude called azimuth, reflecting a horizontal coordinate system. The pointing direction vector is mapped directly in mugranular as the spatial index associating recorded sounds with a particular location in the space. The latitude and longitude are sometimes mapped to granulation parameters, depending on the piece being performed.

### The Handheld Controller

The handheld controller minimally consists of a few buttons on a wireless controller. These are used to arm the granular recording process, to switch granulation on and off, and to place grain clouds in the space. The latter operation is achieved internally in mugranular by taking a

snapshot of the direction vector at the moment a cloud is placed. An independent granulation process is then engaged, stuck at that location. This is used to build up layers of texture during performance. The mapping of buttons to these operations varies depending on the handheld controller implementation and piece. Where enough buttons are available, they are often used to switch between presets of granulation parameters, or between different granulation parameter mappings involving the latitude and longitude.

One notable mapping that is used most of the time with the handheld controller is a kind of button gesture model that distinguishes between a brief click of a button, where the button is pressed and then immediately released, and a button hold, where the button is pressed, held for some time, and then later released. Distinguishing between these two gestures enables one button to be alternately used to toggle or momentarily engage e.g. recording or granulation.

### Mugranular

Throughout our work with the mubone to date, we have used the same synthesizer as the sonic core of the instrument. A typical granulation-based synthesizer with a spatial twist, the mubone granular synthesizer (mugranular for short) uses the direction vector from the orientation sensor as an index into the granulation sound corpus rather than directly retrieving sound grains based on a time index, as in a typical granulator. The approach is very similar to corpus-based concatenative synthesis [66], but using a gestural feature vector instead of audio features.

In a typical scenario, the player records sound in real time, adding it to the corpus along with the concurrent direction vector. Later, previously recorded sounds are retrieved, in granulated form, by pointing the mubone in the same direction. The direction vector is relative to the global coordinate frame, and therefore does not account for the player's position in space, but as long as the player doesn't move around significantly from the position where they started recording, then the qualitative effect of mugranular's recording system is that sounds are recorded into and retrieved from the space around the performer. The actual sonic influence of a gesture is determined while playing; the result is a kind of mapping by demonstration [68], in real time, as

part of a performance.

Internally, mugranular provides a certain number of grain clouds, each with a certain number of potentially concurrent grains. In our present implementation, one cloud always follows the current direction vector from the orientation sensor, and is used to retrieve sounds in real time. The eight remaining clouds can be placed in space; this sets the direction vector and granulation parameters so that the cloud will granulate sounds in that direction until picked up. This allows the player to build up layers of concurrent granulation and build enormous sonic environments in real time.

The last important feature of mugranular is its modular construction. The parameters of the synthesizer are set by sending OSC messages to the synthesis program (implemented in C++ using the JUCE framework). The synthesizer has no internal knowledge about the mubone, and the mapping from mubone gestural signals to sound parameters is implemented in a separate program (in our experiments, typically Max/MSP or Pure Data). This has proven to be an advantageous design, allowing for rapid experimentation in the mapping layer without having to edit or compile native C++, which is used to implement the synthesizer. The design is functionally similar to implementing an external object in Max or Pure Data, but without tying the implementation to one or the other programming environment or having to individually support multiple environments. Both mugranular and the mapping program typically run on the same laptop computer, which receives sensor data from the orientation sensor and handheld controller.

The main parameters of mugranular are amplitude (alternately 0 to 1, or -127 to 0 in dB), grain rate (in Hz), grain duration (in ms), recording state and granulating state, and the spatial direction index used to look up grains in the corpus. Except for the last parameter, these all have the conventional meaning employed in granular synthesizers [63].

### 3.2.2 Maintaining the Mubone

There are roughly four parts of a mubone that have been developed in parallel: the orientation sensor, the handheld controller, the mapping, and mugranular. The mapping has most often been

implemented by Leung on a piece-by-piece basis, so I will not discuss it here. The maintenance of each of the other parts is addressed in more detail in the following discussion.

### The Orientation Sensor

Earlier implementations of the orientation sensor used the MPU-9250 magnetic-inertial measurement unit (MIMU) from InvenSense, read by an ESP8266 microcontroller. The MPU9250 was discontinued, so later implementations have made use of the ICM20948. Interestingly, although I have built new orientation sensors more or less regularly over the course of the mubone's development to keep pace with changes in the technological landscape, Leung most often uses the first orientation sensor constructed, running the original firmware. By avoiding updates, this now 5 year old piece of hardware has proven itself to be reliable and functional, and by continuing to use the same device without ever changing the firmware, software stability is guaranteed; subsequent implementations have incidentally incurred subtle changes e.g. in the OSC namespace and type of magnetometer calibration. These in turn have caused incompatibility with older mapping patches, requiring these mappings to be adapted in order to use the newer sensors. Important maintenance lessons, missed at the time of these subsequent orientation sensor implementations, are found here, and will be addressed further below.

### The Handheld Controller

We have built five different versions of a handheld controller in our different mubone prototypes; this part of the mubone has been a site of significant exploration and experimentation. In our earliest experiments, we used a controller from the Nintendo Wii gaming console, including 11 buttons on the top surface, one trigger button, and an accelerometer, pictured in fig. 3.6.

In later experiments, I developed a rough prototype interface using three buttons and an orientation sensor connected to an ESP8266 (picture not available).

Next, I developed two much more elaborate controllers that we called "yokes" in rough analogy with the flight yoke in an airplane cockpit. The yokes included a rigid attachment to the slide,

**Fig. 3.6** An early implementation of the mubone using a Wii remote.

potentiometers embedded to measure the rotation and deflection of the controller with respect to the slide, four thumb buttons, a thumb joystick, and in the latter version an analog trigger and three additional buttons on the grip. Both of these prototypes used Teensy 3.2 microcontrollers and streamed SLIP-encoded OSC data over a USB serial port to the laptop. The second iteration is pictured in fig. 3.7.

Finally, due to electrical and/or mechanical failure of both elaborate yokes, our most recent experiments have made use of the Joy-Con developed for the Nintendo Switch gaming console, pictured in fig. 3.3.

Although all of these versions of the handheld controller have included some form of orientation or movement sensing, we only briefly experimented with the orientation data from the first yoke controller in our mapping experiments. Instead, it's the discrete buttons that have been most productive.

Performance-ready reliability has been the most essential differentiator between these controllers. Our more elaborate yoke controllers never saw the light of the stage due to their more complicated design and DIY construction. Although it is more glamorous and exciting, and perhaps a better opportunity for novel research, to design a custom controller, the reliability and ease of replacement that comes with using an off-the-shelf controller is advantageous. Commercial products have also not noticeably slowed down the artistic and design research we have been conducting; on the contrary, there have been more delays to our schedule due to technical challenges in building the (now broken) custom controllers. The future may hold opportunities for more rigorous design and engineering of new elaborate yokes, but the use of off the shelf commercial products has so far proved more successful.

Implementing the elaborate yokes prompted a rewrite of the mubone firmware, since the different hardware platform necessitated replacing large parts of the firmware having to do with OSC communication. In both the original firmware and the rewrite, OSC glue code accounts for about 45% of the code. The original firmware included a substantial OSC binding subsystem aiming to streamline the declaration of OSC endpoints. The rewrite dispensed with this subsystem in a bid

**Fig. 3.7** The second of two mubone yoke controllers.

to reduce the overall size of the implementation, but in so doing accepted to include a greater verbosity of glue code when declaring OSC endpoints. Although the rewrite was achieved in fewer lines of code, unfortunately, the attempt to simplify the implementation was unsuccessful insofar as the overall proportion of glue remained the same.

Since switching to the Joy-Con, Leung has favored this device when performing with the mubone. As with the original orientation sensor, its unchanging interface and high reliability are essential motivators for its continuing use.

### Mugranular

Whereas the firmware for the orientation sensor and handheld controllers have been forced to change regularly due to changes in the available parts and hardware design of the controllers, mugranular has been fairly well insulated from changes in the environment, which are largely managed by the upstream JUCE framework. As such, the development of mugranular has been comparatively smooth and linear. The greatest source of complication in its ongoing maintenance has had to do with managing the interface between the synthesis implementation and OSC communications. A stable binding system has been developed, but it is fairly clunky and inconvenient.

The problem that arises is that whenever a new feature is added to the synthesizer, such as adding stereophonic spatialization, any associated parameter must be exposed over OSC, visualized in the graphical user interface, and synchronized between the non-real-time OSC network thread and the hard-real-time audio thread, including smoothing to adapt between the different processing rates of these two threads (i.e. asynchronous control rate and audio sampling rate).

In the simplest case, the parameters of the synthesis process might be specified in a struct, or as members of the synthesis class. Using the techniques that were known at the time mugranular was implemented, this causes a great headache. Because there was no way to generically iterate over the members of a struct or class, this approach would require manually repeating OSC glue code, GUI widget binding, thread synchronization, and smoothing code for each parameter added. As each of these interface are found in different implementation files, it creates a significant cognitive

burden for the developer and maintainer of the software.

Instead, the solution that was settled on at the time was to construct a compile-time list of parameters using `std::tuple`, along with generic functions to iterate over the list. This way, OSC bindings, GUI widgets, thread synchronization, and smoothing could be implemented generically by iterating over the parameter list, with compile-time type safety and static memory allocation. The main drawbacks of this approach, at the time, were that parameter access became indirect and somewhat ugly, requiring parameters to be extracted from the list by a call such as `get<parameter>(list)`, and the actual declaration of parameters remained fairly clunky and annoyingly repetitive; parameters had to be declared as unique types, along with their address, manually then assembled into lists, and manually initialised. For example, the declaration of the granular synthesis parameters is excerpted below:

```
#define ADD_SIGNAL(name, string, type, min, max)\
    constexpr char name##_address[] = string;\
    using name = Signal<name##_address, type, min, max>

namespace synthesis
{
    ADD_SIGNAL(direction, "/direction", Vector, 0, 1);
    ADD_SIGNAL(search_radius, "/search_angle", float, 1, 180);
    ADD_SIGNAL(frequency, "/frequency", float, 0, 10000);
    ADD_SIGNAL(duration, "/duration", float, 0, 60);
    ADD_SIGNAL(playback_rate, "/playback_rate", float, -4, 4);
    ADD_SIGNAL(amplitude, "/amplitude", float, 0, 1);

    ADD_SIGNAL(sound_recording, "/sound_recording", bool, 0, 1);
    ADD_SIGNAL(gesture_recording, "/gesture_recording", bool, 0, 1);
    ADD_SIGNAL(clouds_planted, "/clouds/planted", int, 0, 8);
```

```
ADD_SIGNAL(granulating, "/granulating", bool, 0, 1);
ADD_SIGNAL(reset_trigger, "/reset", bool, 0, 1);

using GrainDescription = List<direction, search_radius, frequency, duration,
      playback_rate, amplitude>;

using State = List<sound_recording, gesture_recording, clouds_planted,
      granulating, reset_trigger>;

void initialize(GrainDescription& g)
{
    get<direction>(g)            = Vector(1,  0,  0);
    get<search_radius>(g)        = 0.5;
    get<frequency>(g)            = 20;
    get<duration>(g)             = 0.05;
    get<playback_rate>(g)        = 1;
    get<amplitude>(g)            = 0.03125;
}

void initialize(State& s)
{
    get<sound_recording>(s)   = false;
    get<gesture_recording>(s) = false;
    get<clouds_planted>(s)       = false;
    get<granulating>(s)       = false;
    get<reset_trigger>(s)     = false;
}
```

```
} // namespace synthesis

#undef ADD_SIGNAL
```

Although this approach was not the most elegant, by adopting a generic programming approach, iterating over a list of parameters, this implementation greatly streamlined the implementation of numerous glue code layers in mugranular.

### 3.2.3 Maintenance Challenges with the Mubone

The maintenance story of the mubone strongly illustrates a few lessons. Glue code constitutes a large proportion and major challenge in the implementation of the mubone controllers and mugranular. Stability in the implementations, the behavior of controllers, and their OSC interfaces, appears as a fundamental concern in performance. Experimentation and long-term upkeep are both at odds with stability, resulting in notable concessions made by Leung in his choice of preferred implementations. Similar to the T-Stick, this suggests that a design favoring adaptation to change may be beneficial to enable experimentation without collapse, yet here it is more clearly shown that stability is also a strong opposing requirement alongside experimentation. In particular, technical exploration is often at odds with the stability needed for artistic exploration.

In both the mubone controllers and mugranular, glue code is a significant concern. Nearly half of all lines of code in the implementation of the controllers consists of glue code, and in mugranular where there are multiple layers of adaptation between OSC, different threads, the GUI, and different control rates, it is an even greater burden that necessitated an elaborate implementation strategy to mitigate. Similar to the case seen with the T-Stick, it is remarkable the amount of effort required to expose endpoints across different interfaces, despite how relatively little actual functionality this effort creates.

Presently, Leung favors the original implementation of the orientation sensor, and the Nintendo Joy-Con as handheld controller. These devices are highly stable, since their firmware essentially does not change. This makes their behavior predictable, and their interfaces consistent, which in

turn makes it easy to use them with old mappings, and give Leung a sense of confidence that they will behave as expected on stage. Over the course of its development, newer orientation sensors have been implemented. However, changes in the available hardware and careless backwards-incompatible breaks in the OSC namespace mean that to use these implementations requires adaptation to existing mappings. This has stymied adoption of the newer hardware. Similarly, as we have experimented with different versions of the handheld controller, each has posed a unique OSC namespace reflecting its unique interface design. Some of these experimental handheld controllers have suffered from poor physical robustness, falling into disrepair. Overall, a clear moral arises: an ongoing musical practice demands excellent stability and reliability. It's clear that disciplined maintenance is required to achieve this kind of stability, even as hardware changes. It's also clear that, for experimentation to have a long-term impact, it must take these strong requirements into account.

## 3.3 Summary

We reviewed the conceptual design and implementation of the T-Stick and the mubone using the framework developed in chapter 2. We examined their maintenance histories, highlighting the challenges and lessons that arise in these two instruments' ongoing development. In both cases, we note that glue code arises as a significant maintenance burden, accounting for a large proportion of code implementations, and in the case of mugranular requiring sophisticated strategies to mitigate. We also find that hardware porting challenges firmware maintenance, often triggering extensive rewrites of firmware. In both instruments, design motivations besides reliability and long-term robustness, such as pedagogy in the case of the T-Stick and experimentation in the case of both instruments, were found to lead to the implementation of fragile and ultimately unsustainable instances of these instruments, and development without considering compatibility was found to poorly affect the stability between implementations of these instruments. The movement of personnel away from the maintenance team was identified as a notable challenge facing maintainers of the T-Stick. We highlight the importance of long-term reliability and stability in the use of the

mubone.

Development efforts conducted without consideration for maintainability and stability, and the coming and going of personnel on the maintainance team of an instrument are difficult sociological issues that make maintenance more difficult. These will be considered as outside the scope of this research. The remaining challenges illustrated as facing DMI maintenance are summarised below:

1. burdensome glue code
2. poor hardware code portability
3. poor robustness against failure
4. instability and incompatibility of behavior and programming interfaces between implementations

# Chapter 4

# Towards a DMI Component Library

In this chapter, we will discuss the physical computing platforms and hardware and software components used in the development of DMIs. We will see that there is a great variety of hardware and software platforms used in these developments, and discuss the ramifications of this diversity for DMI developers. Later, we will find a similar variety of functional components used in DMIs. Respecting the variety of DMI designs, we will argue that there is significant overlap between different DMI developments, and discuss the potential for consolidation of effort in DMI development. This will lead to an enumeration of the challenges that have prevented the development of a standard library for DMI development, and a discussion of the requirements facing such a library.

## 4.1 DMI Platforms

DMIs generally have a physical interface with sensors that measure aspects of the performer's movement and gesture, which are then associated with the parameters of a synthesizer to produce sound and other responsive media. Sensor data is generally acquired by an embedded computer such as a microcontroller unit (MCU), or system on chip (SOC) processor, the latter often in the form of a single board computer (SBC). These embedded processors are often combined with a general-purpose personal computer (hereafter "a laptop", with the understanding that it could also be a personal computer with a different form factor). In this chapter, we will review the computing

environments and architectural arrangements employed in the design of DMIs, considering their respective strengths and weaknesses. Hardware programming is taken to mean programming deeply embedded systems such as MCUs and SBCs, whereas software programming is understood to mean programming applications for laptops and mobile devices like smartphones and tablets. We will consider the relative strengths and weaknesses of these different computing environments and various combinations of them, the issues raised by current development practices, as well as a proposal for how to achieve a hardware-software agnostic approach to development for DMIs.

An exhaustive review of all computing environments employed in the design of DMIs is outside the scope of this review. However, this chapter will provide some exploratory probing of the relative prevalance of different approaches at NIME, which we assume is a representative sample of DMI design practices in research. I reviewed the NIME proceedings from 2020 to 2022 and, based on the title and abstract, selected 76 papers for review that appeared to present one or more DMIs as part of their discussion. Several papers were excluded during detailed review, leaving 67 papers in the included sample set. The review was been updated to account for the 2023 proceedings as a time saving measure, and it is very likely that I missed papers in the considered period, since it is often ambiguous whether something should necessarily be considered a DMI (e.g. is an installation a DMI? What about a system that is completely analog with no computers?). The review is not meant to be comprehensive, but merely indicative. The selected papers were reviewed considering the computing platforms that were employed in the DMIs presented, and why the given platforms and combinations were chosen.

## 4.2 Hardware-Software Tradeoffs

Most of the papers surveyed do not explicitly mention the reasons for choosing the combination of computing devices used to implement the DMI presented. However, drawing on those papers where reasons are given, general considerations, and my own experience developing DMIs, several important factors arise including the following: computational resources; physical form factor; latency; cost; ease of use for rapid development, prototyping, modification, and collaboration; set

up time, and other phenomenological and embodied experiential impacts; and longevity.

### 4.2.1 Computational Resources

Deeply embedded processors such as MCUs are subject to stringent computational resource constraints. These devices generally have orders of magnitude less computational power, memory, and storage compared to a laptop or even a smartphone. SBCs such as Bela have significantly more resources than an MCU, but still much less than a laptop. Such constraints strictly limit these lower-resource platforms from being used to implement certain applications. Anything involving video synthesis or processing is generally not accessible. Many algorithms involving machine learning may be difficult or impossible to adapt to such environments. While MCUs and SBCs continue to grow in power and flexibility, their lower computational resources may motivate the choice of another platform for some parts of the DMI, such as [72], who opted to use Max/MSP on a laptop instead of the Teensy 4.0 Audio Library for sound synthesis.

### 4.2.2 Physical Form Factor

Part of the reason why MCUs and SBCs are necessarily resource-constrained when compared with a laptop is due to their significantly smaller size. In these embedded devices, processor, memory, storage, and a plethora of interfaces and peripherals are often all integrated into a single silicon die, drastically reducing the size and power consumption of these devices. This enables the construction of less intrusive instrument augmentations, handheld input devices, and wireless open air controllers, among other things. In many cases, it is not possible or appropriate to physically embed a laptop in the construction of a DMI, because the physical form of the input device needs to conform to a certain (generally physically smaller) specification. These considerations may also limit the applicability of an SBC such as Bela, which while much smaller than a laptop is still considerably larger than most MCU boards. In case the intended design also requires unconstrained computational resources, then the natural choice is to use an input device with a small MCU in combination with a laptop or more powerful SBC.

### 4.2.3 Latency

It has been demostrated [73] that latency is minimized when signal acquisition, mapping, and sound synthesis are all performed by an embedded real-time processor such as an MCU, SBC, or combination of the two, whereas latency is generally increased by using a laptop. This is due to a combination of factors, including the medium used to transmit data from input devices to the laptop, and non-deterministic non-real-time process latencies introduced by general-purpose operating systems. In case very low latency is an especially important consideration in the design of a particular DMI, this may motivate the choice of embedded platforms over a laptop or mobile device.

### 4.2.4 Cost

In general, MCUs are less expensive than SBCs, which are less expensive than laptops and mobile devices. While cost is likely always considered in the design of a DMI, in some contexts (e.g. [74]) cost may be an especially important factor that could motivate the choice of platform. The designers may take for granted that their users already possess a laptop, speakers, etc., in which case only the cost of the input device may be considered, as in Vieira et al [74] who build a low-cost MIDI controller and take for granted that the user must have something they could control with it. In this case, use of an MCU in combination with a laptop may be considered the most cost effective. If, on the other hand, the cost of the laptop is considered as part of the overall cost of a DMI, then a system using only embedded devices, without a laptop, may be considered as lower-cost.

### 4.2.5 Ease of Use

Certain platforms provide a lower barrier to entry for newcomers and inexperienced collaborators, which may motivate the choice to use them over other alternatives. Arduino and Arduino-compatible MCU development boards are pervasive at NIME, likely due to the increased ease of use these platforms provide compared to directly using the development toolchain supplied by the

MCU manufacturer. On laptops, graphical patcher languages and other high-level domain-specific audio programming environments such as Faust, Chuck, and Supercollider are very commonly used, again likely motivated by the accessibility of these platforms for inexperienced developers, such as Cavdir et al [75], who prioritize a modular design approach to allow collaborators to modify the system. Depending on the application, a certain computing environment might be chosen because it offers a library that already implements the required functionality (e.g. Robinson et al [76] who use Unity for its physics engine); this is part of what makes environments like Arduino and graphical patcher languages so accessible as well.

Another aspect of ease of use has to do with the developer's background experience. Renney et al [77] note that in addition to considering technical design requirements, DMI designers also tend to choose tools that they are already familiar with. For example, both of the DMIs in the survey that used ChucK have Ge Wang, one of the creators of Chuck, as a co-author, and all three DMIs in the survey that used Faust had co-authors associated with Stanford University, pehaps because major Faust proponents such as Romain Michon and Julius O. Smith III are affiliated with that institution. Similarly, 5/11 DMIs using Bela were presented by authors in the UK, where Bela's parent company is headquartered, and at least one author on all 11 of the papers presenting DMIs using Bela had an affiliation in Europe. This might be explained by geographically constrained social network effects resulting in a higher overall familiarity with Bela as a platform in Europe than in other geographic locales.

### 4.2.6 Set Up and Fully Integrated Feel

One strong motivator for using an SBC or MCU to implement a DMI, in combination with each other or alone, is that these combinations lead to fully self-contained instruments. This reduces set-up time compared to a system including a laptop, which was an important motivator for [78] to use Bela, for example. A fully integrated instrument also provides a different phenomenological impact for the player; as described by Ulfarsson et al [79] when describing their feedback-augmented double bass, a fully self-contained instrument "favours tacit musical engagement", and gives the

feeling of "a coherent instrument ... rather than spatially distributed discrete entities" so that "phenomenologically speaking, the feel of a physically singular instrument is carried over from the double bass." Franco et al [14] argue that "instruments that depend on general-purpose computers often fail to provide the sense of intimacy, appropriation and determinacy that traditional instruments offer, since they enforce non-musical activities and rely on fragile technological systems."

### 4.2.7 Longevity

Considering the "fragile technological systems" that Franco et al describe, self-contained DMIs using SBCs and/or MCUs alone may be considered to have better long-term maintenance characteristics compared to DMIs using a laptop or mobile device. Because these latter devices run general-purpose operating systems that receive regular manufacturer updates, sometimes regardless of the wishes of the device's user, software on these platforms tends to passively degrade unless constantly maintained, as the platform gradually drifts out of compatibility with the software. While this is mitigated somewhat by using a high-level environment such as Max/MSP or Pure Data, which handle OS compatibility so that the user should not have to, even these environments sometimes receive backwards incompatible updates to their internal objects, so that if the user updates the software to keep up with breakages caused by the operating system, they may become exposed to breakages caused by the environment meant to protect them from the operating system. In contrast, embedded devices can avoid some of these issues. When used without a laptop, these devices provide a fully self-contained instrument. This means that software updates are under the complete control of the maintainer of the instrument, so that breaking changes are admitted only at the deliberate behest of the maintainer. In general, a fully self-contained purpose-dedicated instrument left on a shelf for several years should still function when the dust is brushed off and the batteries are recharged. This may also be true of a modular instrument using a laptop, but only if the laptop, as well as the input device, is placed on the shelf to collect dust. Because of the higher cost of general purpose computing devices, this is generally not a practical possibility.

## 4.3 DMI Architectures: NIME 2020-2022

All papers reviewed were found to use some combination of a few broad kinds of computing devices: MCUs, SOCs, laptops (and other general purpose personal computers), mobile devices such as smartphones and tablets, and off-the-shelf (OTS) input devices such as the Myo armband, Gametrak, and MIDI keyboard controllers. It is generally understood that DMIs comprise input devices, synthesis processes, and mappings [27]. The papers reviewed generally admit this conceptual model, with all DMIs considered in the survey arguably having some form of input device mapped to a media synthesizer, usually audio. In a strong majority of the DMIs surveyed (67%, 46/67), the input device is physically implemented as a separate device from the rest of the system, using either an MCU with dedicated sensors (54%, 25/46), or an off the shelf (OTS) input device (39%, 18/46), sometimes (in two of the papers surveyed) both a custom controller using an MCU and an off the shelf input device, and in one of the papers surveyed the input device is a smartphone. In most of these cases (61%, 28/46), both devices require custom programming.

In contrast, in 13 of the papers surveyed (19%, 13/67) the entire DMI is implemented using a single embedded process: usually a lone SBC, sometimes a lone MCU. In these implementations, all signal acquisition, processing, and synthesis is performed in a single integrated hardware device. Also fully integrated, but without use of sensors other than audio microphones and those embedded in mobile devices, 7 of the papers surveyed (10%, 7/67) present DMIs that are implemented solely using a laptop (7%, 5/67) or mobile device (3%, 2/67). In the sole remaining paper, the computing architecture combination is not clear.

The breakdown of computing platforms used at NIME in the surveyed period is summarised in fig. 4.1.

The characterisation of these architectural patterns as involving hardware programming, software programming, or a combination of the two is not always clear. In general, programming MCUs may be considered to represent hardware programming unambiguously, and programming a laptop or mobile device to represent software programming unambiguously. However, SBCs

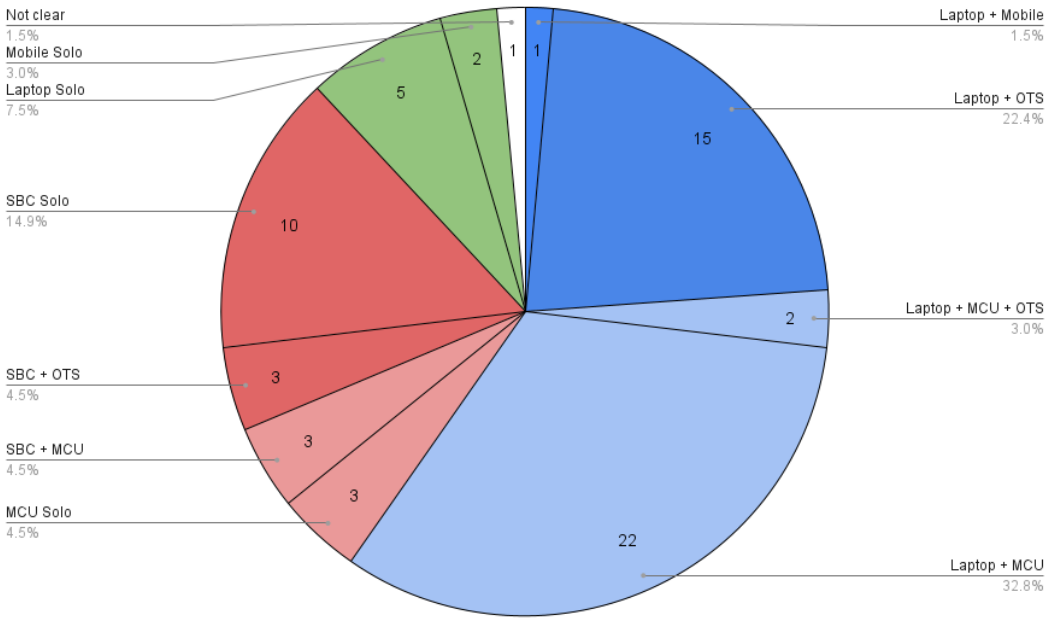


**Fig. 4.1** Computing platforms used in DMIs at NIME 2020-2022

often blur the line, presenting some of the strengths and weaknesses of both of the aforementioned extremes.

## 4.4 Issues with Combining Heterogeneous Computing Environments

In many of the DMIs surveyed (42%, 28/67), a combination of programmable devices is employed in the architecture of the DMI. These combinations introduce additional design challenges that are not as prevalent when a single device is used. These considerations mainly draw on general observations and my personal experience developing DMIs.

### 4.4.1 Readability

Writing computer code that is easy for programmers to read and understand is a general goal of most programmers. Various factors can influence code readability, ranging from variable naming to use of architectural patterns. Readability is important because it is essential for software

maintainers to be able to reason about a program in order for them to have any hope of finding and fixing issues, extending the functionality of the program, and generally performing the work necessary to maintain it over time. This is relevant even when a single person writes and maintains code, since maintenance regularly spans such long time frames that details which are clear while writing the code may become forgotten before maintenance is required. Readability is even more essential when the code may be maintained by other programmers than the original author, as is common in the research context.

For example, it is a common problem for the original author of a software library to eventually abandon the project for a variety of reasons (e.g. graduation, as in section 3.1.3). In order for the library to receive continued maintenance, others must be able to read, understand, and appropriate the code. In academia this is especially prevalent, since e.g. graduate researchers such as myself may exit academia to seek employment in industry upon completing their studies, causing them to abandon code developed during their degree due to lack of available time. This issue is further compounded by the incentives of academia, which can often bias researchers towards implementations that are sufficient for conducting their research, without immense care given to the readability of the code. Subsequent potential maintainers may then, upon experiencing difficulty in understanding existing implementations left by their predecessors, opt to develop new code instead of reusing existing libraries, restarting the cycle.

In the development of DMIs, the choice of computing environments has a large inherent impact on readability. Readability may be considered to require the developer to have a certain baseline familiarity with the programming language used, and where hardware programming is involved, also familiarity with the details of that hardware. Hardware programming and maintenance may thus be considered to be inherently more challenging than writing code for a software-oriented environment, due to the higher knowledge demanded of the developers reading and writing the hardware code. Similarly, code written in multiple programming languages may be considered as inherently less readable than code written in only one language. The predominant trend of using a combination of multiple computing devices in DMIs, such as a laptop programmed in Max/MSP

and an MCU programmed in C++, almost necessarily makes readability more difficult.

### 4.4.2 Code Reusability

Especially when different programming languages are employed across different parts of a DMI, it may be challenging or impossible to reuse code across a DMI's parts. For example, code written in Supercollider cannot be deployed on a microcontroller. Although numerous recent projects aim to address this issue, such as by cross-compiling code written in one language (e.g. Pure Data, Faust, gen~, rnbo, SOUL, Cmajor) to a more portable language (C++ in all of the examples just given), this is not without its own difficulties. It is not always obvious how to use the generated C++ code, which may assume certain run-time functionality, or may not provide a convenient interface. Even when the same language is used across different parts of the DMI, differences in the run-time framework and hardware environment can also make it challenging to reuse code. Poor conditions for code reuse eventually leads to wasteful repeated effort, and introduce incompatibilities between different versions of a DMI, complicating design and maintenance, when platforms inevitably change in response to changing maintenance requirements (e.g. a discontinued microcontroller or sensor), or when changes are deliberately introduced to allow the exploration of different design alternatives.

For example, recent work on the mubone has sought to port its granular synthesizer to run on Bela, in order to provide a fully self-contained instrument and reduce set up time. This unfortunately requires significant refactoring, since the original JUCE C++ code for the synthesizer depends on JUCE's runtime, which is not readily available on Bela, even though both use C++ as their main language. Another example comes from the development history of the T-Stick. Originally, the gesture modelling code for the T-Stick was implemented in Max/MSP, receiving data from the T-Stick encoded in an idiomatic USB serial protocol. Over time, these programs have recieved poor maintenance, and are no longer compatible with newer T-Sticks, which now use Open Sound Control to send data to the sound computer. To address the lack of available gesture models, many of the Max/MSP models were ported to C++ and now run directly in the

T-Stick firmware. In addition to requiring a cost in repeated development work, the new models have not been shown to perfectly replicate the behavior of the old ones, introducing possible incompatibilities between old T-Sticks and newer ones.

Furthermore, the design of a DMI is a complex endeavour, and the important factors in the design are not always obvious from the outset. For example, midway through the process of porting the mubone to use Bela, we had the opportunity to perform with the instrument at the new CIRMMT music multimedia room (MMR), with dozens of loudspeakers affording unique and exciting opportunities for sound spatialization. The limited compute power of the Bela board, and its limited number of output audio channels, necessitated the revival of the previous iteration of the mubone so that we could implement the performance in Max and make use of the MMR's multichannel affordances. Our current roadmap for the mubone involves maintaining both a laptop version and a Bela version as important alternatives that offer different tradeoffs.

This example illustrates how the choice of hardware-software combination is an ongoing process that can sometimes be subject to unexpected changes to requirements, and how these changes can lead to significant wasted effort as existing systems are refactored, re-implemented, revived, and reconsidered in unpredictable ways. The issue lies not with the conceptual design of the instruments, or even with the particular algorithms implementing that design, but with the changing and incompatible requirements of different computing platforms. If Max/MSP patches could be run on the ESP32, then the old T-Stick gesture extraction could have been used directly for the newer T-Sticks. If the previously implemented synthesis and sensor acquisition software for the mubone could have been used directly with Bela, then the effort of reimplementing these functionalities could have been avoided, and the sting of subsequently reverting to the previous iteration lessened.

### 4.4.3 Quadratic Glue

One way to work around issues of code reuse is to carefully partition code so that important subcomponents are platform and environment agnostic, allowing them to be more readily adapted

to work in different contexts. Although some parts of a DMI design are necessarily hardware specific, and thus challenging to make portable, many components can be successfully designed in a way that minimizes dependencies and requirements on the runtime environment. In terms of the framework developed in chapter 2, functional software components with semantically strong endpoints are good candidates for portable implementation. Unfortunately, even with careful design, different environments often require additional glue code to be introduced. This problem is well described by Celerier [80], in the context of writing media processors for different host environments such as Max/MSP, Pure Data, and other sound programming languages, as well as different plugin formats for digital audio workstations, such as VST, Audio Unit, and CLAP, and is discussed in further detail in section 4.7. Because each different host environment imposes a unique API to allow one's reusable code component to be bound to the environment, binding code grows quadratically $M * N$ in the number of components $M$ and host environments $N$.

In the context of DMI design and implementation, different computing devices may be viewed as further host environments; each MCU, SBC, and other processor to be targeted further multiplies binding code. Within these environments, communication protocols like MIDI, OSC, OSC query, and libmapper can be seen as additional host environments. This abundance of host environments further challenges code reuse in DMIs.

### 4.4.4 Comparing DMI Architectures

Considering the design factors outlined in the previous sections, the different platform combinations found in the surveyed papers can be characterised in terms of their benefits and tradeoffs. Such an analysis is presented in table 4.1. Notably, there is currently no platform that achieves the best possible performance on all considered factors. In particular, ease of use, general accessibility, and computational power tends to be potentially higher on a general purpose computing platform such as a laptop. This is likely to remain true over time, given that such platforms can in general draw on more power and space to provide higher compute performance, which can in turn be leveraged to provide improved ease of use. However, while sacrificing compute power and potentially

usability for inexperienced developers, using an SBC or a solo MCU can provide lower cost, smaller form factors, and allow the design of fully integrated, self-contained, purpose-dedicated devices, with the attendant benefits to latency, set-up, feel, and longevity. If compute power continues to decrease in physical size, and as their development toolchains continue to improve, simplify, and become more familiar to more users, it seems reasonable that embedded devices like SBCs and MCUs may eventually overtake combinations with a laptop. Until such time, however, DMI designers must choose what combination of hardware and software to use in order to achieve their design goals, considering the inherent tradeoffs of different computing platforms.

Alternatively, particularly as designs are iterated on and maintained over longer periods of time, designers may investigate many alternative combinations of computing platforms. Unfortunately, this raises severe issues for readability and code reuse, such as the quadratic growth of binding code. These issues will make it challenging to maintain DMIs over time unless adequately addressed. I will argue that a necessary step towards resolving these challenges will be the development of a cross-platform library of portable reusable DMI components. The remainder of this chapter will provide further motivation and requirements for such a library.

| Combo | compute | size | cost | ease of use | integrated |
|---|---|---|---|---|---|
| Laptop Plus | + + | big | 1000+* | + | modular, general-purpose |
| SBC (Solo or Plus) | + | small | 100+ | + | integrated, dedicated |
| MCU Solo | – | smaller | 10+ | – | integrated, dedicated |
| Laptop Solo | + + | big | 1000+* | + + | integrated, general-purpose |
| Mobile Solo | + | small | 100+* | + + | integrated, general-purpose |

*The cost associated with using a laptop or mobile device may be taken for granted, giving lower considered cost.

**Table 4.1** Comparison of computing platforms.

## 4.5 Components of DMIs

We will now consider the components employed in the development of DMIs, based on the framework developed in chapter 2. We will observe that there is significant commonality between different developments, and thus significant opportunity for consolidation of effort. These considerations

will lead to the development of a wishlist of DMI components that serves as a recommended pool of standard components that would enable the development and replication of a huge variety of different DMIs. The components centered in this section are the functional components a DMI (section 2.8) that provide for the functionality of the instrument. We are particularly concerned with the functional software components, and will survey existing reusable software component libraries in section 4.5.5.

### 4.5.1 Sensor Components of DMIs

The use of different sensors in the development of DMIs within the research community has been well characterised by previous comprehensive reviews [81], [82]. Although a more recent review is somewhat lacking, many heuristic observations can be addressed without controversy. Sensors that were common in previous reviews remain common. The use of combined magnetic and inertial measurement units (MIMUs) including magnetometer, accelerometer, and gyroscope sensors, appears increasingly common. The use of capacitive touch sensing, especially the Trill devices from Bela, appears increasingly common. Broadly speaking, both in prior reviews and more recently, the most common sensors used in research DMI developments tend to be those that are most accessible—sensors that are inexpensive, widely available, and reasonably easy to use appear to be favored.

In Medeiros' et al's 2014 review [81], building on Marshall's earlier review [82], all sensors used in over a decade of research at NIME are summarised in just 18 different sorts of sensors. There is a very high degree of technological homogeneity present in these developments. Many of these DMIs likely use the same exact parts, such as the ubiquitous force sensitive resistors (FSRs) manufactured by the Interlink company, the same small pool of MIMU integrated circuits, and the most common off-the-shelf consumer electronic devices such as Leap motion controllers, game controllers, and Kinect infrared trackers.

The various sensing technologies, in turn, tend to make use of highly similar interfaces and capture systems. A great variety of sensors are read by simply connecting their analog output

to an analog-digital converter (ADC). Still more are read directly from a digital general-purpose input output (GPIO). Another large group are interfaced over a serial protocol, typically I2C or SPI. The last main group are consumer electronics, which may be accessed using an API provided by the manufacturer, or over standard protocols such as USB HID.

The high level of homogeneity across sensor usage is notable, and presents an opportunity for significant consolidation of effort. If there were a portable library of inexpensive, available, and easy to use sensor software components, this could facilitate a very high level of reuse across DMI developments.

An illustrative summary of sensor components and their software coverage by existing libraries is provided in table 4.2. While many devices have commonly used libraries that facilitate accessing their data, such as the Arduino API, or an Arduino library for reading a particular sensor, the functionality of such libraries is almost always limited to accessing the sensor data; although there may be useful signal conditioning, fusion, and analysis that is common across different DMIs, these functionalities tend to be implemented anew in each DMI development. For example, both the mubone and the T-Stick have independent implementations of algorithms for differentiating a single tap from a double tap on a button. This issue will be revisited in section 4.5.5. There is a particular need for a DMI-specific library of sensor components that includes gesture models and signal conditioning commonly needed for DMI developments. It is argued, basically, that existing solutions do not come with "batteries included", resulting in repeated implementation of similar functionality after the sensor data is acquired. This common functionality becomes locked into instrument-specific firmware, despite that it could be useful across a variety of instruments making use of the same kinds of sensors.

### 4.5.2 Actuator Components of DMIs

DMIs regularly include actuators that enable them to exert physical influence over the environment. By far the most common actuator is the dynamic loudspeaker, which is the default actuator used by any DMI that produces sound. Video projectors and displays are similarly ubiquitous

| | |
|---|---|
| MIMUs | IC-specific Arduino libraries |
| FSRs | analogRead() |
| buttons | digitalRead() |
| potentiometers | analogRead() |
| video | OS and language-specific libraries |
| infrared | analogRead() |
| capacitive touch | IC-specific Arduino libraries, e.g. TrillCraft |
| biosensing | IC-specific Arduino libraries |
| piezoelectric disks | analog audio interface |
| microphones | analog audio interface |
| textiles | various |
| light | IC-specific Arduino libraries, analogRead() |
| bend | analogRead() |
| hall effect | analogRead() |
| ultrasound | digitalRead(), Arduino libraries |
| pressure/flow | analogRead() |
| GPIO | Arduino API |
| ADC | Arduino API |
| SPI | Arduino API |
| I2C | Arduino API |

**Table 4.2** Common sensors and software means of interacting with them. Available libraries are generally limited to merely accessing sensor data.

wherever a DMI produces a video output. RGB LEDs are also common, used for example as state indicators, signal visualisers, and for aesthetic purposes. Many new instruments involve acoustic feedback loops, often making use of typical loudspeaker woofers (e.g. [83], [84]), and surface transducers (e.g. [84]). Vibrotactile feedback is often of interest. Marshall [82] found that loudspeakers and vibrotactile motors were most commonly used for vibrotactile feedback. The type of loudspeaker may be dynamic loudspeakers [85], [86], surface transducers [87], or specialized vibrotactile voice coils [88]. Eccentric rotating mass (ERM) motors are especially common vibrotactile motors [89], [90]. Force feedback is less common, but is an important subject of research. In his review, Kirkegaard [91] found a great variety of specialized force feedback devices. In order to build a system that is affordable and relatively simple, he emphasises the usefulness of brushless direct-current (BLDC) motors [92], and stepper motors [93].

The actuators mentioned above very likely cover a large proportion of DMIs developed at NIME, although an extensive focused literature review would be necessary to say what proportion. At the time of writing, libraries that facilitate using these kinds of devices tend to be narrowly focused, often on a single device or type of device. In the case of more ubiquitous display devices, like loudspeakers and video displays, readily available consumer devices are most often used, and the lack of software support is less of an issue. Where software support is needed, it often falls to the DMI developer to implement most of the needed functionality, despite reasonably high commonality of actuators across DMI developments. As with sensors, there is a significant opportunity for consolidation of effort.

| | |
|---|---|
| loudspeakers | audio interface |
| surface transducers | audio interface |
| vibrotactile voice coils | audio interface |
| video projectors | PC display output |
| LEDs | analogWrite(), LED-specific libraries e.g. FastLED |
| ERMs | analogWrite() |
| BLDC motors | Arduino libraries; Mechaduino |
| stepper motors | As for BLDC motors |

**Table 4.3** Common actuators in DMIs and means by which to interact with them.

### 4.5.3 Audio Components of DMIs

The field of sound synthesis and audio processing is comparatively much more mature than the other areas that intersect DMI development. Highly sophisticated hardware and software devices with rich libraries of configurations are widespread and readily accessible. Digital audio workstations (DAWs) provide for a high degree of flexibility in crafting elaborate signal processing chains. Many DAWs include modular environments for developing customised sound synthesis systems. A generous variety of audio programming languages are also available, enabling low-level customisation starting from basic unit generators. Even in the bare-metal embedded context, there are numerous feature-complete audio libraries, such as the Teensy audio library[1] and the Lightweight Embedded Audio Framework[2], for example. We are spoiled with excellent options for sound synthesis, processing, and exploration.

Whereas the main challenge with sensors, actuators, and mappings is that one must build so much from scratch or assemble various disparate and heterogeneous component-specific libraries, with audio components the greater issue may be how to interoperate with the existing cornucopia of essentially feature-complete options. DAWs, modular environments, and audio programming languages all tend to define their own preferred API. Although MIDI is ubiquitously available across most systems, its limitations in terms of resolution and data rate may motivate a requirement to use a different method.

Although it may be worthwhile to develop a library of audio synthesis and processing components, because the existing landscape is already so rich it may be more benefitial for a library to focus on interoperating with DAWs and existing audio languages. This may, for example, take the form of allowing sensor, actuator, and mapping components to be used within other environments, or enabling existing environments to appear as components within the library, for example by embedding the external audio language (e.g. libpd), or using network protocols so that components in the external environment are controllable from library components (e.g. Mapper4Live [94]).

[1] `https://www.pjrc.com/teensy/td_libs_Audio.html`
[2] `https://github.com/spiricom/LEAF/`

### 4.5.4 Mapping Components of DMIs

| | |
|---|---|
| one-to-one linear transforms | basic math operations, language or library built-ins |
| filters | language built-ins, hand-rolled, Max/MSP DOT [95] |
| differentiators | hand rolled, DOT |
| integrators | hand rolled, DOT |
| matrix transforms | Eigen library, Max/MSP MnM library [40] |
| transfer functions | language built-ins, hand rolled |
| clipping | language built-ins, hand rolled, DOT |
| folding | as for clipping |
| wrapping | as for clipping |
| preset interpolators | specialized libraries (most for Max/MSP), hand rolled |
| neural networks | Wekinator [96], specialized libraries |
| gesture followers | C++ or Max/MSP library [97] |
| probabilistic models | Max/MSP XMM library [33] |
| autoencoders | specialized libraries, hand rolled |

**Table 4.4** Common and notable mapping techniques, and their most prominent software support.

The majority of mappings documented in the literature, and likely most of those found in artistic practice, consist of mere connections with scaling and offsets. Consider the control voltage input on a typical modular synthesizer; a knob provides a stable offset voltage $b$ that is added to an input control voltage $x$ that is adjusted in scale by an attenuator $m$ so that the final parameter value $y$ is given as $y = mx+b$. One-to-one connections of this sort are more than adequate for many applications, since in many cases the signals themselves already contain a high level of modelling; complex cross-domain mappings are regularly unnecessary. This was observed, for example, in the mappings produced by participants in my MA thesis research [38], which largely consisted of this kind of one-to-one linear mapping. Generalising this to allow many-to-many mappings, without increasing the complexity of the scaling and adjustment to signals flowing through the mapping stage, we can readily implement such mappings in terms of a matrix multiplication or an affine transformation [40], as discussed in section 2.4.

Nevertheless, numerous interesting mapping techniques have been presented in the literature, and more are introduced every year. Less common techniques such as preset interpolation (e.g. [98]–[100]), probabilistic models (e.g. [33]), gesture following ([97], [101]), neural networks (e.g. [96]), autoencoders (e.g. [37], [102]), as well as numerous other approaches borrowed from machine learning have all been reported and used gainfully. Use of these mapping techniques tends to be more experimental, in line with their greater novelty. As is seen in table 4.4, Max/MSP is especially well supported considering these more novel mapping strategies. Efforts to facilitate their availability and usability, especially in environments other than Max/MSP, seem likely to lead to more frequent use of these useful tools.

### 4.5.5 Prior Component Libraries

Consider the parts of DMIs outlined above. We summarise some notable existing libraries of DMI components according to this useful, if incomplete, division of DMI components in table 4.5. Rather than a comprehensive review, this sampling of prior work gives a broad sense of the available reusable sofware components that is roughly representative of the field at large.

| | |
|---|---|
| sensors | Arduino API implementations, sensor- and platform-specific libraries, Puara [55] |
| actuators | similar to sensors |
| mappings | Pd mappings library, the Digital Orchestra Toolbox [95], MnM [40], XMM [33], GVF [97], Max nodes object, Wekinator [96], libmapper [29], built in arithmetic objects of audio languages (see sound), built in mapping facilities of plugins and DAWs, application-specific models (e.g. xio Technologies Fusion library), machine-learning frameworks (e.g. tensor flow, MediaPipe, and many others) |
| sound | standard libraries of audio languages (Max/MSP, Pd, Supercollider, Faust, Chuck, etc.) and modular environments (VCV Rack, Cardinal, Bidule, Reaktor, etc.), DAW plugins, the STK, Teensy Audio Library, LEAF, Avendish [80], and many others |

**Table 4.5** Some existing libraries of DMI components.

There are several observations that seem notable considering prior work. There is no library that covers all aspects of DMI development. Sensor, actuator, and mapping libraries tend to focus on a specific device, technique, or model; there are few collections of components facing

these aspects of DMI development. Furthermore, these narrowly focused libraries tend to be implemented for a single platform; there are few cross-platform libraries facing these aspects of DMI development, which are typically either embedded-only or tied to a hosted interpreted language such as Max/MSP or Pure Data. As discussed in section 4.5.3, libraries and resources for sound synthesis and processing are comparably mature. However, even here components tend not to be widely compatible with different platforms; plugin formats are not compatible with embedded devices, and libraries built for any given sound and music computing language are generally incompatible with other environments. Faust and Avendish stand as a notable exception in this regard, providing (in principle) very good compatibility across platforms thanks to their unusual approaches that address the generation of glue code.

### 4.5.6 The Components of DMIs

The design and implementation of DMIs remains an advanced pursuit, marked by novelty and a high barrier to entry. The field is essentially not yet mature. The basic components are not yet well agreed-upon. There is no expected pool of components for a DMI development standard library in the way that there are expected unit generators in a standard library for audio synthesis and processing. However, the field is quickly reaching an important inflection point. As discussed in this chapter, we can observe a high degree of commonality among DMI developments. It is reasonable to identify a pool of components that are the most common, and which together would likely enable the development of many, if not most, DMIs as seen in the literature. The wishlist in table 4.6 is largely heuristic: a systematic review of the literature would allow to define a comprehensive list of components in each category ordered by prevalance, allowing to prioritize development efforts towards the most impactful components.

### 4.5.7 Summary

We reviewed the physical computing platforms used in the implementation of DMIs. We found that DMIs are commonly implemented using a combination of MCUs, laptops, mobile devices, and

| | |
|---|---|
| sensors ([81]) | MIMUs, FSRs, buttons, potentiometers, video, infrared, capacitive touch, biosensing, piezoelectric disks, microphones, textiles, light, bend, hall effect, ultrasound, pressure/flow, fibre optic, GPIO, DAC, SPI, I2C |
| actuators | loudspeakers, surface transducers, vibrotactile voice coils, video projectors, LEDs, LRAs, ERMs, BLDC motors, stepper motors |
| mappings | one-to-one linear transforms, filters, differentiators, integrators, matrix transforms, transfer functions, boundary conditions (clipping, folding, wrapping), preset interpolators, neural networks, gesture followers, probabilistic models, autoencoders |
| sound | bindings for audio languages (Max/MSP, Pd, Supercollider, Faust, Chuck, etc.), bindings for modular environments (VCV Rack, Cardinal, Bidule, Reaktor, etc.), bindings for DAW plugin formats, bindings for communications protocols (MIDI, OSC, libmapper, etc.), and perhaps also oscillators, filters, transfer functions, delay, feedback, and other audio unit generators |

**Table 4.6** Common DMI components.

SBCs, often including multiple programmable devices in one implementation, often using multiple heterogeneous programming languages, the choice of a platform or combination of platforms being motivated by considerations of computational resources, form factor, latency, cost, ease of use, set-up and feel, and expected longevity. We highlight how this landscape creates challenges in code readability and reuse, and causes a proliferation of glue code that arises when attempting to develop portable code for DMIs. Considering the factors that motivate DMI architectural decisions, we find that there is no single platform that meets all possible requirements. The problems of readability, reuse, and glue are thus expected to persist.

We then consider the functional components most commonly found in the implementation of DMIs. We review the most common sensor, actuator, mapping, and audio processing components found in DMIs, and the libraries of reusable code that enable their implementation. We note that, although there is a great variety of components, there is significant overlap and commonality between implementations of DMIs. We find that there are few libraries that are useful in both embedded and hosted environments, and that there is no library that attempts to cover the full range of functional components found in DMIs. We assemble a wishlist of components that such a library might aim to implement.

## 4.6 Why a DMI Component Library

In this section I will briefly consolidate the motivations for the development of a library of portable reusable DMI components that have been accumulating over the course of the discussion thus far, and provide additional arguments motivating for the valuable benefits that such a development may bring. As no such library has been around long enough to demonstrate that these benefits are attainable, this section stands as a position statement. I argue that a portable library of reusable DMI components would provide significant benefits to the maintainability, stability, and improvement over time of our DMI developments, and that a consolidated effort to build, maintain, and evaluate such a library would improve the scientific rigour and impact of DMI design research.

### 4.6.1 Portability and Reuse Favor Maintenance

We are concerned with the development of maintainable DMIs that can be engaged with over the long term. Unfortunately, as disussed in section 1.2, this is challenging to achieve. Requirements shift. The technical landscape changes. Considering the development history of the T-Stick and the mubone (chapter 3) we found challenges due to glue code, code portability, robustness against failure, and stability and compatibility over time. The diversity of the hardware and software systems used in the implementation of DMIs, discussed above, further complicates DMI development, introducing challenges of code readability, and contributing to issues of glue code and portability.

At the most simplistic level, considering the discussion to this point, it is obvious that in order to improve the maintenability of DMIs, from a technical standpoint, it would be beneficial to address these issues and attempt to develop code that is highly portable and reduces the requirement for glue. Improving the portability of our software components would allow us to reuse them more efficiently over the course of maintaining a DMI. This would improve the stability and compatibility of the instrument, since there would be minimal changes required to the code over time. The burden of maintenance would be reduced, since we would be able to reuse more

of our code more often. However, there are much greater impacts that could be realised from the development of such a library.

As we have described above, DMI developments within the research community exhibit a substantial amount of overlap in the components involved in their implementation. A highly portable software component for reading data from a MIMU, for example, could in principle therefore be reused in numerous DMIs. Rather than many DMIs each maintaining their own separate MIMU software, a great deal of wasteful repeated effort could be avoided with all of the maintenance focused on one shared component. So reusing a DMI software component allows the maintenance burden of the component to be consolidated. The more components, and the more widespread their use, the greater the impact. But the potential impact is even greater than just consolidating the effort of maintenance.

### 4.6.2 Portability and Reuse Enable Generalisable Evaluation

Evaluation of DMIs is deeply challenging. DMIs are complex systems composed of numerous modular components. Their evaluation is equally complex, highly nuanced, and heterogeneous. It is not well established, in general, how DMIs should be evaluated, what criteria should be used, whose perspectives and goals should be considered, or whether DMIs should even be subject to evaluation in the first place. As Dahl notes [103], "our work involves many interacting components, few clear guidelines exist, and the criteria for success are seldom explicit." Systematic reviews of DMI evaluation practice [104]–[106] suggest that there is little consensus about how DMIs should be evaluated. If we are interested, as designers, in developing a scientifically robust body of DMI design knowledge, in developing research that allows our instruments to improve over time [7] rather than shockingly not degrading [8], this situation is troubling. Yet as Rodger et al [107] note:

> "There are multiple possible stakeholders for a new musical instrument with different possible evaluative concerns (e.g. performer, audience, etc.), and different qualitative dimensions across which evaluative judgments might be mapped (e.g. robustness,

> playability, ...). If one takes seriously the complex multi-way interactions between the instrument's teeming affordances, the musician's repertoire of effective actions to engage these, her skill in coordinating these actions within the unfolding musical situation, the fit or tension between all these and the broader socio-cultural-historical setting, then the aim to perform any conclusive generalisable evaluation seems fanciful."

Expanding on Rodger et al, I would argue that the challenge of conclusive or generalisable evaluation of DMIs runs even deeper than the complexity of musician+instrument ecologies. A lone DMI by itself can be an incredibly complex system. DMIs are composed of numerous modular subsystems: sensing devices, signal acquisition systems, signal transmission systems and protocols, DSP algorithms, synthesizers, sound effects, and mappings. Any of these subsystems by itself can be taken as an entire field of study, in which numerous different evaluation methodologies are used. When these systems are combined, often in complex interweaving networks of influence [30], the resultant instrument becomes a complex ecology unto itself, with its own co-constituting interacting processes, teeming affordances and repetoires of effectivities. Why should we expect evaluation of one musical instrument to offer any insight for another, when they may utilize unrelated sensors, transmission media, mappings, and synthesizers?

Yet, consider the T-Stick and the mubone, two instruments with wildly different physical form, design intent, performer communities, etc., but both instruments use MIMUs to sense their orientation and dynamic movement. Thus, unlike an evaluation of a T-Stick or a mubone, research considering MIMUs is of immediate use to both of these otherwise dissimilar instruments. The specificities [107] of the instrument still need to be taken into account. For instance, a MIMU gesture modelling algorithm that is evaluated well in some given terms may or may not be useful for either of these instruments. But the probability that the results of such an evaluation are meaningful for these instruments, or any other instrument using a MIMU, is much higher than the probability that the results of an evaluation facing either the T-Stick or the mubone would be meaningful to any other instrument. Of course, evaluation of DMI components will not address all of the possible criteria that may need to be considered. Nevertheless, by evaluating DMI com-

ponents we can incrementally improve the quality of the materials we assemble to build complete DMIs. This can only provide benefit.

This is where the greatest possible impact of a portable library of reusable DMI components arises. The evaluation and validation of the components of such a library, unlike evaluation and validation of a whole instrument, would be expected to be generalisable—as generalisable as the components are reusable. If we are concerned with the development of validated DMI design knowledge, and with being able to efficiently mobilise this knowledge, then collaboration around a library of reusable components is a natural fit.

## 4.7 Requirements for a DMI Component Library

Considering the discussion above, we will now develop a set of requirements for a communal library of DMI components that can support the long-term validation and sustainable development of DMIs. This discussion will enumerate the underlying challenges that have prevented the development of an adequate libary of DMI components up until now.

### 4.7.1 Many Platforms

DMI development is an interdisciplinary activity, and one DMI very often involves multiple hardware and software platforms, as was discussed in section 4.1. This may be a natural and perhaps even necessary reflection of the design requirements of these devices. For example, the interface of a DMI often needs to be extremely light and unobtrusive, while at the same time the output of the instrument may involve computationally expensive mappings, audo signal processing, and even video processing requiring a GPU. Using a low-power low-resource microcontroller with a wireless transport to acquire sensor data and forward it to a more powerful hosted computing environment is an appropriate design decision given such requirements, and the use of multiple computing environments and APIs is, at the time of writing, the most efficient and convenient way to implement the software for such a system.

We can speculate about a few of the reasons it's most convenient to use e.g. Arduino for the

microcontroller, Max/MSP for the mapping, and a DAW for audio processing. The developer's experience and expertise is probably the primary determining factor; we will use the tools that we are most familiar with, and we will achieve our aims most efficiently with them because of our expertise in using them. Another considerable factor is the availability of ready to use components that provide the functionality we need. Each different computing platform, programming language, and media environment comes with its own standard library of components that will naturally make certain tasks trivial. There is no tool more convenient for sound synthesis than a ready-made sound synthesizer. There is no language better for custom making a sound synthesizer than a computer music language. There is no better way to prototype embedded firmware than with an embedded firmware prototyping environment.

Since there is no DMI development platform, it's only natural that we make our DMIs with a collection of other platforms, each suited to one of the parts of the work. While this is certainly an adequate solution to our problems, I argue that the requirements of DMI development are not merely the sum of the requirements of embedded firware development, computer music, and multimedia processing, nor the sum of any other specific collection of subdomains. What is lost by starting with these parts is that each DMI development effort begins working from a set of subcomponents that are not exactly made for the task at hand, and each effort leads to the creation, over and over, of the same common top-level functional components. This is a needless waste of effort and it also makes DMI development excessively difficult to learn. Not to suggest that DMI development should be easy—it necessarily requires some expertise in sound synthesis, interaction design, and embedded systems, among other things, all of which are substantial areas in their own right—but it may be more difficult than necessary when a novice developer has to learn two or three different programming languages, at least as many unique APIs, and find a way to put them all together.

### 4.7.2 Limitations of Interpreted Languages

Many of the most common platforms used in the development of DMIs are interpreted languages. Examples include Max/MSP, Pure Data, and Supercollider. These languages are well adapted to developing mappings and media synthesis systems, and are invaluable tools in DMI development. If we were to try to pick the thing closest to a DMI development platform today, we would most likely point to one of these kinds of languages. However, many DMIs require designs that cannot be implemented in these languages. Furthermore, because of their mutual incompatibility, component libraries written in any one of these languages are inherently unportable, limiting opportunities for widespread reuse and collaboration on such libraries.

Many DMIs make use of embedded systems for part of their design, especially for sensor data acquisition and sensor fusion, and for designs requiring especially low action-sound latency. Due to their lower memory and computational resources, and the relatively extreme variety of embedded systems, it is generally not possible to use interpreted languages that are commonly employed on hosted systems in an embedded context. Interpreted languages are generally not well optimised for tight memory resource constraints. The great variety of embedded systems makes it especially difficult to support an interpreted language since it must be ported to numerous different systems, often with idiosyncratic low-level quirks. As many embedded systems do not have a conventional operating system, assumptions about the available platform APIs made by most interpreted language implementations may not be met by embedded systems. The result is that languages such as Max/MSP and Pure Data can usually not be run on a microcontroller, making them poorly suited for sensor data acquisition that is essential in many DMI designs.

Furthermore, interpreted languages are, as a rule, not compatible with each other. Code written in Max/MSP is not usable in Pure Data or any other language. This means that libraries written in one such language are locked within that language. This limits the potential for collaboration and reuse of these libraries. Pure Data users have no ability to use libraries written in Max/MSP and therefore no incentive to collaborate on these libraries. This is unfortunate, because DMIs broadly share many design features. The lack of portability between interpreted

languages results in a situation where useful functionality must be implemented repeatedly in each language.

For these reasons, I argue that a compiled language is a more appropriate choice for the development of a portable library of DMI components. Compiled languages can be more adequately optimised for use in resource-constrained embedded systems, and through ubiquitous C/C++ APIs for extending interpreted languages, components written in an appropriate compiled language can be ported to run in a variety of other languages. The most obvious choice of language is C or C++, since these languages provide the most straightforward interoperability with the numerous interpreted languages and other platforms commonly used in DMI implementations. Other languages, notably Rust, may also be suitable, but we argue that C++ remains the most appropriate choice for a broadly portable and compatible library of DMI components, simply because C++ is already the standard low-level programming language used in DMI development: DMI firmware is most often implemented in Arduino, which is based on C++; audio plugins are most often implemented in C++; and many of the most common multimedia environments and languages used in DMIs are implemented in C++ or it's highly compatible predecessor C. C++ is the defacto standard language for real-time multimedia applications, and for this reason alone it is highly appropriate for the development of a DMI component library. As will be discussed below, the reflection and metaprogramming facilities available since C++20 make the language even more advantageous for this purpose.

This is not to suggest that everyone should immediately stop all work using interpreted languages; obviously each language brings its own advantages and disadvantages and it is far from my intention to suggest that C++ is the only appropriate language for DMI development. Rather, I argue that for the development of portable DMI components in particular, C++ is one of the strongest options. Eventually, any standard library for DMI development should aim for its components to be reusable in all common DMI development environments, including bare-metal firmware and graphical interpreted languages and everything in between. In this way, for example, users more comfortable in a graphical environment such as Max/MSP or Pure Data may also make

use of the components in the library, broadening its reuse and consequent impact. The use of an appropriate compiled language such as C++ is advantageous to enable this kind of portability.

### 4.7.3 Glue and Boilerplate

When writing a library, there are two common places where the library may be used: directly in user code, or in a binding to another language or environment. In the first case, the user must learn the interface of the library in order to invoke it appropriately in the user program. Often such invocations are trivial and repetitive across many cases where the library may be used, in which case we term the source code of the invocation as boilerplate. In the latter case, as well as interfacing with the library code, it is also necessary to interface with the target environment of the binding, such as the API for an interpreted language or plugin system. The source code of a manually written binding is termed glue code. In much the same way as the invocation of a library tends to be repetitive across different programs, glue code targeting the same environment also tends to exhibit repetitive patterns of implementation.

The purpose of code reuse is to reduce the amount of code that must be written and maintained in order to access a given software functionality. Unfortunately, glue code and boilerplate increase the amount of code that must be written. Although boilerplate is almost always worthwhile, because its cost is generally constant (i.e. a fixed number of invocations of the library) and negligible compared to reimplementing the needed functionality, glue code can be more problematic. Especially in a context where there are a large number of commonly used software environments (section 4.1), it may be necessary to write a substantial quantity of glue code.

For example, suppose you have implemented a harmonic distortion sound processor and a reverberator, and you wish to use both as VST and AU plugins in your DAW, as well as externals in Pure Data and Max/MSP. At a minimum, this would entail writing 8 bindings; the VST, AU, Pd, and Max/MSP bindings for the distortion, and the same four again for the reverberator. The overall number of bindings is given by the number of components (in the example, two) multiplied by the number of target platforms (in the example, four). Because computer music broadly

and DMI development especially involve a large number of potential components and platforms, the quadratic growth of glue code is a tyrannical force that essentially prohibits the creation of a broadly portable library of components. Hence why prior work exhibits an almost complete absence of cross-platform libraries (section 4.5.5); almost all prior libraries are compatible with only a narrow range of platforms.

### 4.7.4 Cross-platform APIs for Addressing Glue

The implementation of DMIs involves numerous heterogeneous computing environments. Because of conflicting motivations related to size, computational power, cost, and ease of use, it is likely that this heterogeneity will remain a feature of DMI development for the foreseeable future, or until such time as extremely computationally powerful, inexpensive, small, and easy to use platforms become widely available.

Because of this, and the huge variety of audio processing environments and languages that are commonly in use, it is extremely difficult to develop a library of portable software components for the implementation of DMIs. In the ideal, such a library would need to interoperate with a large number of environments, including various embedded hardware platforms, interpreted languages, and plugin architectures. Given a large number of environments and a large number of components, the amount of glue code that would need to be written and maintained would grow unsustainably large due to its quadratic growth in the number of environments and components.

In order to implement a portable library of DMI components, it's necessary to automate the creation of glue code. A prominent example of how this has been achieved in prior work is given by Faust, with its architectures system [108]. Faust architectures extend a set of base classes that effectively define an API expected by Faust signal processors. Derived classes that make up an architecture interact with this API and that of the target platform in a generic manner, independent of the particularities of the signal processor. This allows one architecture to act as glue for any number of signal processors. Other examples of portable libraries include JUCE for DAW plugins and Arduino for microcontrollers. Similar to Faust architectures, these libraries

effectively define an abstract API that components interact with directly, so that bindings for each target platform only have to be written once to match the abstract cross-platform API.

Cross-platform APIs are pragmatic, useful, and adequate. Their main limitation is that they are generally limited to one aspect of DMI development (such as DSP, DAW plugins, or microcontrollers). This is both a feature and consequence of their design. They do not aspire to cover all the functional needs of DMIs, so their APIs reflect the conceptual framework that most naturally suits their problem domain. Components that are outside of that domain are thus difficult to implement, as they do not fit the conceptual model of the API. It is difficult to implement a Faust DSP that reads from a sensor: the messy asynchronous realities of sensors do not fit into the pure functional synchronous world-view of Faust DSPs. In the reverse, it is difficult to implement sound processing in Arduino.

DMIs are complex systems, and their parts do not all resemble each other. Furthermore, the design of these systems remains an ongoing area of study. For these reasons, the components of a DMI are likely not to naturally fit into one overarching mental framework or implementation structure unless it is particularly flexible and amenable to adaptation. Having to conform the design of each component to the framework of a given API can pose a subtle challenge to the developer of a component, especially if the natural structure of the component does not fit the expected structure given by the API. But the biggest issue with the imposed worldview of a given API, at least if one's aim is to develop a library of DMI components that suits all kinds of DMI components, is that the API for one type of component (e.g. audio processing) does not, in general, cover all components of a DMI. Existing APIs tend to be focused on one group of platforms and the associated type of components, such as DAW plugins or graphical patcher languages. Their API worldview is predisposed to support components that fit into this world, and incompatible with those that do not.

To some extent, it is probably necessary for any cross-platform system to come with design expectations that must be imposed on the components developed under that system. Many cross-platform APIs make strong assumptions about the structure of components. An example of strong

assumptions is plugin frameworks like JUCE, which assume that the endpoints of a plugin will be exposed as GUI widgets. Weaker assumptions are made by dataflow languages like Max and Pd that require components to be cast as objects with inlets and outlets receiving floats, integers, lists, messages, and arguments. Even more flexibility is given by a system such as libmapper, which accepts all kinds of metadata associated with endpoints.

In the ideal of flexibility, a cross-platform system should impose nothing on the structure and design of components. Instead of requiring components to conform to the framework, such a system takes up the burden of adaptation, examining the structure of each component and appropriately conforming to the component's implied requirements without the component developer having to consider the assumptions of the cross-platform system. In this ideal of flexibility, the cross-platform system does not make assumptions. Each component can be implemented in the way that is most natural to its requirements. In the best case scenario, this lack of imposition should even extend to the physical implementation, in the sense that components should not have to draw in dependencies from the cross-platform system. They should be fully independent, implemented in the most natural terms without recourse to the requirements of the cross-platform system.

### 4.7.5 Requirements

In order to reduce the barriers to rapid development and maintenance of DMIs, it is necessary to develop a cross-platform library of DMI components. There are numerous challenges to such an effort. Because DMIs often span multiple platforms, such as embedded systems and interpreted languages, it shall be necessary to enable the portable use of components in the library across numerous environments. Use of a compiled language, most naturally C or C++ due to their prior ubiquitous use, provides the best opportunity for portability across these environments. However, the large number of environments commonly used results in an untenable explosion of glue code, unless steps are taken to automatically generate bindings for each component to each environment in which it may be used. Automating away glue requires components to be treated generically by the cross-platform library, meaning that information about their interface must be programatically

exposed to the library. The most common strategy to implement this is using a runtime API, where the component exposes its endpoints by calling into the API. As well as introducing performance and maintenance burdens on the development of components using such an API, this approach generally requires the component implementation to adapt to the imposed structure of the API.

Based on these considerations, we can derive the following requirements and recommendations for a cross-platform library of DMI components:

- Portability (Modern C++): the library should be implemented in a compiled language suitable for use in embedded systems as well as for interoperating with the most common interpreted languages used in DMI development. C++ is a natural choice due to its ubiquity, portability, interoperability, and prior status as the defacto standard for real-time multimedia applications.

- Automate Glue: the library should automate the generation of glue code, so that its components can be used in all of the environments in which DMIs are commonly implemented.

- Wide Contract: the library should adapt to the structure of components rather than imposing a particular structure, so that all of the components of a DMI may be included in the library.

### 4.7.6 Prior Work Consider the Requirements

In table 4.7 existing reusable libraries of DMI components are considered in view of the above requirements. The highest score possible is 3/3, if a library were to meet all requirements. The highest score achieved is 2/3; All existing libraries considered fail to meet at least one requirement, being specific to a particular interpreted language, requiring glue to use, or implementing an API that is specific to a particular domain of DMI development such as DSP.

Notably, Avendish [80] may merit a higher score, in principle. It makes use of an unusual approach, based on C++20 reflection features, so that its API contract can be widened at will. Avendish's main limitation is that it is focused on audio processors, but because of this flexible

API architecture, it is in principle possible that the library could be extended to address other domains of DMI development.

| Library | Portability | Glue | API contract | Score |
|---|---|---|---|---|
| Arduino APIs | embedded only | required | domain-specific | 0/3 |
| Arduino libraries | embedded only | required | domain-specific | 0/3 |
| Puara [55] | good | required | flexible | 2/3 |
| Pd mappings library | Pd only | minimal | flexible | 1/3 |
| DOT [95] | Max/MSP only | minimal | flexible | 1/3 |
| Eigen | strong | required | domain-specific | 1/3 |
| MnM [40] | Max/MSP only | required | domain-specific | 0/3 |
| XMM [33] | Max/MSP only | required | domain-specific | 0/3 |
| GVF [97] | Max/MSP only | required | domain-specific | 0/3 |
| Max nodes object | Max/MSP only | required | domain-specific | 0/3 |
| Wekinator [96] | good | required | domain-specific | 1/3 |
| libmapper [29] | good | required | flexible | 2/3 |
| application-specific models | often good | required | application-specific | 1/3 |
| machine-learning frameworks | often good | required | ML specific | 1/3 |
| DAW plugins | DAWs only | minimal | audio only | 1/3 |
| the STK | strong | required | audio-specific | 1/3 |
| Teensy Audio Library | Teensy only | required | audio-specific | 1/3 |
| LEAF | strong | required | audio-specific | 1/3 |
| Faust | strong | automated | DSP-specific | 2/3 |
| Avendish [80] | strong | automated | DSP-specific(?) | 2/3? |

**Table 4.7** Overview considering the extent to which existing DMI component libraries meet the stated requirements.

### 4.7.7 Summary

We reviewed the ideals and challenges facing the development of a cross-platform library of reusable DMI components. Based on these considerations, we developed a set of requirements for such a library. Considering a representative pool of prior work, we find that no existing library meets all our requirements. We note that the design pattern proposed by Celerier [80] in the context of audio plugins is especially promising prior work. Because of its implementation in plain C++,

using reflection to provide a wide API contract, it seems like an approach that could be readily generalised to cover the full range of DMI components, although this has not been attempted in prior work.

# Chapter 5

# Sygaldry

Sygaldry is a new C++20 library of DMI software components. This chapter will describe its design and implementation, which attempt to meet the requirements developed in section 4.7. The name of the library comes from the fantasy novel "The Name of the Wind" by Patrick Rothfuss ([109]), where "sygaldry" is a form of magical enchantment used to embue artifacts with unusual magical capabilities. The narrator describes the craft as follows (from Chapter 51 of the novel):

> Sygaldry, simply put, is a set of tools for channeling forces. Like sympathy made solid.
>
> For example, if you engraved one brick with the rune ule and another with the rune doch, the two runes would cause the bricks to cling to each other, as if mortared in place.
>
> But it's not as simple as that. What really happens is the two runes tear the bricks apart with the strength of their attraction. To prevent this you have to add the rune aru to each of the bricks. Aru is the rune for clay, and it makes the two pieces of clay cling to each other, solving your problem.
>
> Except that aru and doch don't fit together. They're the wrong shape. To get them to fit you have to add a few linking runes, gea and teh. Then, for balance, you have to add gea and teh to the other brick, too. Then the bricks cling to each other without

> breaking.
>
> But only if the bricks are made out of clay. Most bricks aren't. So, generally, it is a better idea to mix iron into the ceramic of the brick before it is fired. Of course, that means you have to use fehr instead of aru. Then you have to switch teh and gea so the ends come together properly...
>
> As you can see, mortar is a simpler and more reliable route for holding bricks together.

Rothfuss's artificing bears a strong practical resemblance to computer programming, probably recognizable to anyone who has implemented a program of any complexity. Analogous to his system of runes channeling forces, the Sygaldry C++ library aims to provide a set of reusable parts that can be combined to embue artifacts with unusual musical capabilities, channeling flows of data rather than magical forces.

## 5.1 Design of Sygaldry

In the following sections, we will describe the high-level design choices that informed the development of Sygaldry, including the scope of the library at the time of writing, its basic design elements, and important considerations that guide its implementation.

### 5.1.1 Scope

Our ideal DMI component library should implement components for sensors, mappings, and sound processing. These components should be highly portable, and readily reusable in as many environments as possible, without having to manually write glue code.

In order to limit the scope of its implementation, Sygaldry currently prioritises components for reading from sensors, and quickly exposing this data to external processors. It is essentially a library for writing DMI firmware, including functionality for reading from the most common types of sensors, and exposing this information over USB serial and Open Sound Control (OSC) protocol.

Fully functional reusable components for reading from sensors are not found in the most common software environments used for implementing DMIs. As such, providing coverage of this area of DMI implementation is most necessary. Furthermore, although the current implementation only covers reading sensors, the overall design is well suited to covering the other aspects of a DMI. The basic principles of the library's design were established by Celerier [80] in the context of audio plugins, demonstrating the applicability of the design patterns used to that domain, and this is also indicative of its suitability for controlling actuators beyond loudspeakers. The inclusion of sensor fusion and gesture modelling components for some of the sensors in Sygaldry provides evidence of the usability of the design patterns for mapping. Thus, by extending Celerier's work, Sygaldry provides proof of concept that the design patterns employed can be used to address all kinds of functional DMI components.

### 5.1.2 C++20 Functional Components

The fundamental unit of Sygaldry's implementation is a C++20 software component. These functional software components implement the functionality of the conceptual components of a DMI (chapter 2). Because these components are implemented in C++, there are no limitations on the type of functionality that can be implemented, from reading sensors to sound synthesis. They are implemented such that the endpoints of the component can be reflected over programmatically at compile time, enabling glue code to be generated automatically. This affords components in the library the best possible portability, with minimum maintenance burden. The details of the implementation design pattern are described in section 5.2.

### 5.1.3 Instruments

In order to eliminate boilerplate and facilitate reuse, Sygaldry extends the reflection over endpoints necessary to automatically generate glue code. Using the same C++20 reflection features, Sygaldry allows a DMI to be specified in a high-level declaration of its components, which are then iterated over to automaticallly generate boilerplate code needed to activate each component at

the appropriate time, and pass data between coupled components. This automatic generation of glue code and boilerplate is currently limited to the reasonably flexible Arduino-style setup/loop implementation structure. Other more elaborate execution structures are easily embedded within this top-level structure, which is also easily extended or ignored. Details of this feature of the library are described in section 5.2.8.

### 5.1.4 Environments

As well as functional software components, Sygaldry provides a collection of software platform components. These come in three types: bindings, platform abstractions, and build tools. Bindings provide automatic generation of glue code. They iterate over the endpoints of functional components, at compile time, and produce code that exposes those endpoints over a given protocol or transport such as Open Sound Control or USB serial. Platform abstractions provide access to APIs specific to a given platform, such as Arduino, or interact directly with hardware devices such as I2C and SPI controllers. They serve to provide a narrow path for platform dependencies to flow through, so that efforts to support more hardware and software platforms have minimal and explicit interfaces that need to be implemented (see section 6.2). Finally, build tools are software components responsible for configuring the code compilation toolchain, invoking the toolchain to produce executable code, and installing that code in its final runtime environment.

### 5.1.5 Dependency Management

Developing portable software components is essentially a problem of dependency management. Certain fundamental software components are portable: built-in language constructs, parts of the standard library, and any software components that depend only on built-in language and standard libraries. It's worth emphasizing that these things are also software components. It's usually taken for granted that the language itself is portable. In fact, C++ (and especially C++20) has exactly as much portability as its compilers have targets. There are two kinds of C++ targets: freestanding and hosted. Not all standard libraries are available in a freestanding environment.

The most portable possible C++ code consists only of built-in language constructs, and libraries available on a freestanding target. And this code is only portable to environments for which compiler targets have been written. C++20 in particular is not yet available on every platform on which Sygaldry might productively be used; this issue will be further discussed in section 5.4.2.

Beyond the highly-portable core features of the language, much of the necessary functionality of a DMI depends on interacting with specific hardware and particular software environments. Although it's usually possible to directly interact with the hardware of a microcontroller by reading and writing memory-mapped registers of the device, these kinds of interactions are almost always mediated by 3rd party software, such as the hardware manufacturer's hardware abstraction framework. For example, interacting with the hardware of an ESP32 microcontroller is most efficiently achieved by using the ESP-IDF library provided by the hardware manufacturer. As opposed to directly accessing the hardware, using the ESP-IDF improves the portability of a software component, since the framework is designed to work with numerous different ESP32-family microcontrollers. Whether the hardware is accessed via including and/or linking with the ESP-IDF, or by directly accessing memory-mapped peripherals, interacting with hardware means that a software component depends on external assumptions about the hardware (e.g. which memory address corresponds to a given peripheral), and is not portable to software environments where these assumptions are not met.

Carefully managing dependencies of this kind allows the explicit demarcation of portable code from non-portable code, which favors the development of portable code, and encourages careful management of dependencies to make them easier to manage when porting non-portable code to a different environment. To achieve this careful management, Sygaldry makes use of the software component architectural design patterns described by Lakos [110]. Software components are carefully demarcated using the filesystem, and dependencies are explicitly and prescriptively annotated in build-system metadata. Among other effects, this facilitates reuse of interface code such as header files when writing non-portable platform abstraction code, facilitating porting efforts. Functional and environmental software components are grouped into namespaces based on their

type and level of portability. Further detail is provided in section 5.2.2.

### 5.1.6 Literate Programming

One of the main goals of Sygaldry as a project is to support the development of a validated pool of design knowledge and readily reusable DMI components. There are numerous major technical barriers against reuse, such as glue code, boilerplate, and portability. These issues have to do with the logic of machines, code, and hardware, and they are best addressed by adaptations at these levels. In addition to these difficult issues, there are also numerous socio-economic barriers that make it difficult for developers to reuse existing software, challenges facing the human users and developers of design knowledge and reusable software components. Although these are not the main focus of this work, Sygaldry is developed in a literate programming style in hopes that this may address some of these human-level barriers to reuse. All of the source code is provided within the flow of a natural language exposition that is meant to be read and understood by human readers; machine readable code is extracted, and in principle is meant to serve a secondary role to the communication from human to human provided by the literate source code. More discussion of literate programming is provided in the literate source code of the parser used in the project to extract machine code from literate sources[1].

## 5.2 Implementation of Sygaldry

In the following sections, we will describe the physical implementation of Sygaldry at a high level. This serves to highlight some of the technically interesting properties of the library's implementation. The full literate source code of the library is located online[2] for readers interested in the unabridged and messy details.

[1] `https://github.com/DocSunset/lili/tree/6c87856a573a0a2001e2a06a15492a5bba87f33a`
[2] `https://github.com/DocSunset/sygaldry/tree/8888ba2c4397cfe74aa1ba7d0c3767adbd04ea8b`

### 5.2.1 Reflection in Modern C++

In order to achieve our requirements for a cross-platform library of DMI components, it's necessary to make use of several features of modern C++. Plain C does not provide the metaprogramming and reflection features necessary to fully automate glue code, so C++ must be used. Prior to C++20, the available tools for reflection at compile time were not sufficient to completely avoid the need for a runtime API to enable automatic generation of glue code. Thus C++20 or later is required in order to meet the requirements to avoid runtime binding code and fully automate the generation of glue code.

Jean-Michael Celerier describes the design patterns and language features that allow us to meet the above requirements [80]. In summary, according to Celerier, adequate reflection can be achieved by iterating over the members of an aggregate struct using Boost PFR, and using concepts and requires statements to query the existence of a certain member, and its type metadata. By reacting to the information accessible in this way, we can write bindings with a wide API contract. These reactions can be achieved using `if constexpr` branches, thus avoiding runtime polymorphism and similar structures with their associated performance overhead. Furthermore, metadata can be embedded using the type system and immediate (`consteval`) functions, further reducing runtime costs. Making use of these reflection and compile-time programming facilities, Celerier proposes a design pattern in which endpoints of a component are declared as members of an aggregate struct, allowing them to be iterated over automatically by binding code, while still achieving a reasonably elegant and readable implementation of components.

The details of Celerier's design pattern are further elaborated in his library Avendish (`https://github.com/celtera/avendish`), which is focused on audio processor components. An audio processor is modelled as a class with input and output endpoints and a main processing operation that provides the component's functionality. The key feature of the design pattern is that input and output endpoints are implemented in a reflectable manner: endpoints are grouped in aggregate structs, and metadata is exposed using the type system and immediate functions.

For each type of endpoint, such as inputs, there is an appropriately named `struct {/*...*/} inputs;`.

These structs of endpoints are required to be aggregates; the main limitation of this requirement is that these structs are not allowed to be derived classes. Each member of the struct is expected to be a single endpoint. Use of aggregate structs to implement endpoints enables the endpoints to be enumerated and iterated over using Boost PFR. This allows bindings to generically inspect components, as will be described in more detail below.

Endpoints, in turn, are designed using patterns that enable metadata to be readily extracted from them. Metadata such as the name, range, and initial value, are exposed using immediate functions. The sort of endpoint is exposed through the type system. Each endpoint has a data member named `value`; numbers, arrays of numbers, and messages can all be recognised by the type of this value data member. Further metadata tags can be applied using enumerations.

All of these patterns can be easily applied without including or invoking any library code, using pure and simple C++. For example, a valid endpoint implementation in plain C++20 may resemble the following, a floating-point endpoint with a range from 0 to 1, an initial value of 0.5, named "example endpoint" and tagged as session data:

```
struct example_endpoint {
    float value;
    static consteval auto name() {return "example endpoint";}
    static consteval auto min() {return 0.0f;}
    static consteval auto max() {return 1.0f;}
    static consteval auto init() {return 0.5f;}
    enum {session_data};
} example;
```

Certain conventions are implied, especially the naming of the input and output endpoint structs, the naming of the value data member of endpoints, and the naming of the expected metadata immediate functions. Essentially, binding code needs to know in advance to look for the appropriate data members and member functions in order to access the information encoded

by them. This is a limited form of API contract; bindings and component implementations need to agree on the expected names of things. This kind of contract is much weaker than typical runtime API contracts, primarily because it is relatively trivial to widen the contract to include additional expected names and shapes of things, and doing so does not imply any physical or logical requirement for existing components to change or adapt their implementation. A new component may introduce a new way of representing a certain concept; bindings may then be extended, usually as simply as by adding a compile-time branch, in order to accomodate the new representation. For example, another endpoint might name its value data member "state" instead of "value", and represent its range by returning a struct:

```
struct range_set {
    float min;
    float max;
    float init;
};

struct different_endpoint {
    float state;
    static consteval auto name() {return "different endpoint";}
    static consteval auto range() {return range_set{0.0f, 1.0f, 0.5f};}
    enum {session_data};
} different;
```

Binding code can be easily adapted to recognize these different kinds of implementation without any physical or logical adaptation required on the side of the component implementation. For example, binding code attempting to initialize the value of an endpoint might resemble the following:

```
auto& value_of(auto& ep) {
```

```
    if constexpr (requires {ep.value}) return ep.value;
    else if constexpr (requires {ep.state}) return ep.state;
}

void initialise_endpoint(auto& ep) {
    if constexpr (requires {ep.init()}) value_of(ep) = ep.init();
    else if constexpr (requires {ep.range().init}) value_of(ep) = ep.range().init;
}
```

Whereas APIs typically leak their expectations into component implementations, the use of compile-time reflection patterns means that expectations can flow the other way; bindings adapt to components, and changes to bindings needn't influence component implementations. Another branch could easily be added to allow bindings to accept a component whose value is returned by a method instead of accessed via a data member, for example.

Nevertheless, it is necessary to acknowledge that an API contract is still struck between components and bindings. Knowing that bindings are implemented to accept components implemented according to a certain design pattern naturally encourages the adoption of that design pattern. It quickly becomes convenient to develop abstractions for that design pattern, such as a helper library that makes it more convenient to declare endpoints with their metadata. Avendish provides a library of endpoints such as `slider_t` and `toggle_t` that enable endpoints to be declared without having to tediously repeat numerous immediate function definitions and other metadata representations for each endpoint. The helper library comes to define the expected shape of endpoints that bindings will expect, and use of other endpoint representations has the tendency to fall outside of the implicit API expectations of binding code.

So although the only requirements imposed by the design pattern on components is that their endpoints be exposed in a fashion that can be iterated over, and that endpoints' metadata and structure be exposed in a reflectable manner, there does seem to be a natural tendency for certain API expectations to arise, become codified concurrently with the development of a library of

endpoint helpers, and then be followed out of convenience by component implementations using the helpers. This story plays out below in section 5.2.3 and section 5.2.4.

### 5.2.2 Physical Design

Before delving into the logical implementation of the library, it is useful to provide a brief overview of the physical layout of the library to help the reader keep a sense of bearing as the presentation continues. Sygaldry is physically divided into software components according to the principles described by Lakos [110]. Software components are grouped thematically, and each group is given a package group identifier. The groups at the time of writing are summarised in table 5.1. Each software component is given a unique name prepended with its package group. The component is placed in a directory with this name, in which is located a literate source file bearing the component name, and a `CMakeLists.txt` file that declares build system metadata. Most of the literate sources generate corresponding C++ source and header files, also bearing the component name.

| | |
|---|---|
| sygac | concepts library |
| sygah | helpers library |
| sygbe | binding components using the ESP-IDF |
| sygbp | portable binding components |
| sygbr | binding components using the Raspberry Pi PicoSDK (RP2040) |
| syghe | hardware abstraction components for ESP32 devices |
| sygsa | sensing components using the Arduino API |
| sygse | sensing components using the ESP-IDF |
| sygsp | portable sensing components |
| sygsr | sensing components using the Raspberry Pi PicoSDK (RP2040) |
| sygup | portable utility software components |

**Table 5.1** Sygaldry package groups

Alongside the software components, the library also contains a collection of shell scripts that facilitate building and using the library under different platforms, nix scripts that facilitate setting up the development environment, documentation style files, dependencies, and instrument firmwares.

Sygaldry software components are developed with careful consideration towards dependencies. Each package group has explicitly allowed dependencies, and all other dependencies are disallowed. This ensures that platform-specific components are the only ones that are allowed to invoke platform-specific libraries, so that porting efforts are clearly delineated. Three kinds of dependency are distinguished: link-time dependency, compile-time dependency, and logical dependency. When one component depends on another, the former is termed the client and the latter the provider. A link-time dependency is found when one component links the compiled objects of another, typically including the header of the depended upon component. A compile-time dependency is found when one component makes use of the source code of another, as in the instantiation of templates in client code. A logical dependency exists when one component relies on another to have a certain shape, such as having a certain method with a certain name and characteristics, typically in the case of generic code that is required to adapt to the shape of their components.

Dependencies can introduce several troubles. When providers change their implementation, clients must either re-link or recompile. When providers change their interface, this introduces a more troubling situation where clients must change their source code to adapt to the new interface, so that as well as the provider and client recompiling, any transitive clients (i.e. the clients of the client) must also recompile. Finally, dependencies can introduce trouble when porting from one platform to another. Dependencies at the root of the dependency graph can make it especially difficult or impossible to port to a new platform in cases where the root dependency cannot be met on the new platform. For this reason, the fundamental component groups in Sygaldry have very strictly limited 3rd party dependencies (just a handful of pure C++ libraries). The main limitation of the library, in this respect, is its dependency on C++20, which remains unavailable on several important platforms, notably including many Arduino API implementations. This is addressed further below.

### 5.2.3 Endpoints

Sygaldry endpoints are based on the design pattern established by Avendish and described in section 5.2.1. Endpoints have a data member called `value` that holds their current value, and metadata are associated with endpoints using immediate functions and enumerations. The preferred method of exposing metadata is a static immediate function. This provides excellent runtime performance characteristics, and by associating expected metadata function names with classes, different metadata can be composed on an endpoint using inheritance. This allows users to define novel combinations of metadata declaratively while respecting the expectations of bindings as embodied in the available helper classes.

Textual metadata classes are defined in the metadata helper component[3], including classes `name_` and `description_` for associating a name and description with an endpoint (or component). Minimum, maximum, and initial values are associated with the class `range_`. These metadata classes merely define a single static immediate function that returns the metadata associated with the class; the metadata itself is in turn given by non-type template parameters. All of this means that there is minimum runtime cost associated with accessing the metadata, which is stored once no matter how many instances of a given endpoint exist.

For illustration, here is the implementation of the name class:

```
template<string_literal str>
struct name_
{
    static _consteval auto name() {return str.value;}
};
```

Inheriting from this class allows an endpoint to easily define its name.

Several tag helper classes are defined that merely introduce a particular enumeration definition into derived classes. The variadic template class `tagged_` is used to associate a list of tags with a

[3] https://github.com/DocSunset/sygaldry/blob/8888ba2c4397cfe74aa1ba7d0c3767adbd04ea8b/sygaldry/sygah-metadata/sygah-metadata.lili.md

derived class; it merely inherits from all of the template type arguments in the variadic argument list.

```
#define tag(TAG) struct tag_##TAG {enum {TAG}; }
tag(write_only);
tag(session_data);

template<typename ... Tags>
struct tagged_ : Tags... {};
```

Lastly, two broad sorts of values are defined that differ mainly by their temporal semantics: persistent values and occasional values. Persistent values are those that are expected to always have an active value that is updated more or less all the time. Occasional values have more message-like semantics; their value is only updated some of the time, and when updates take place there is usually work to do. Template classes are defined for both of these that include assignment and conversion operators so that instances of these classes behave transparently like values of their underlying type, given by the template type parameter to the class. As well as the underlying value, the occasional value class also stores a boolean flag that is set to true when the value has been updated.

Several compositions of metadata and value classes are provided that implement common kinds of endpoints that became needed during the development of functional components in the library. The roster of basic endpoints at the time of writing is summarised in table 5.2, and the implementation of these endpoint helpers, the two value types, and the tag metadata helpers are found in the endpoints helper component[4].

For illustration, the definition of the toggle helper follows below:

```
template< string_literal name_str
        , string_literal desc = ""
```

[4] https://github.com/DocSunset/sygaldry/blob/8888ba2c4397cfe74aa1ba7d0c3767adbd04ea8b/sygaldry/sygah-endpoints/sygah-endpoints.lili.md

```
        , char init = 0
        , typename ... Tags
        >
struct toggle
: persistent<char>
, name_<name_str>
, description_<desc>
, range_<0, 1, init>
, tagged_<Tags...>
{
    using persistent<char>::operator=;
};
```

An example using this helper is in section 5.2.4.

| | |
|---|---|
| toggle | Persistent `char` with a range of 0 or 1 |
| button | Occasional `char` with a range of 0 or 1 |
| text | Persistent `std::string` |
| text message | Occasional `std::string` |
| slider | Persistent numeric value |
| slider message | Occasional numeric value |
| array | Persistent array of (usually numeric) values |
| array message | Occasional array of (usually numeric) values |
| bng | A persistent bool tagged "bang" for impulsive activations |
| vec3 message | An array message with convenience x, y, z accessors for use as a 3D vector |

**Table 5.2** Sygaldry endpoint helper classes

As well as accepting inputs and producing outputs via endpoints, components may also accept inputs and references to output data locations via arguments to their main function. This enables the implementation of generic endpoints, termed throughpoints in the conceptual framework described in chapter 2. By implementing the component as a generic template, endpoints reflecting the strong semantic domain of external components can be implemented as template type arguments and treated generically by the component implementation. The runtime system described

in section 5.2.8 facilitates automatically ferrying around throughpoint data without the library user having to write boilerplate to do so.

### 5.2.4 Components

The central software components in the library are those that implement the useful functionality required by DMIs. These functional components, similar to Avendish, are expected to have aggregate reflectable structs of endpoints, one called `inputs` for input endpoints, and/or one called `outputs` for output endpoints. In a deviation from Avendish, functional components in Sygaldry are expected to have two member functions: init and main. The init member function is meant to set up the component for normal operation, and is expected to be called once during program startup. The main member function is expected to implement the principle functionality of the component, reading input endpoints, performing the expected processes, and possibly updating output endpoints.

Using the endpoints helpers, it is fairly trivial and highly declarative to enumerate the endpoints of a component. The init and main functions then contain the implementation of the functionality of the component. The init-main structure reflects the runtime operating model of Arduino sketches. It has not been found to pose any troublesome limitations yet, although it would not be excessively challenging to permit components to adopt different forms, such as those found in Avendish for audio processors, for example.

As an example, here is a complete implementation of a gesture model that debounces the raw readout from a button connected through a GPIO and provides some additional event outputs as the state changes. I would like to eventually include mappings for recognizing double taps, distinguishing taps from long presses and holds, and other gestures that can be performed with a single button, but the component in its current minimal state serves as a good demonstration of what a functional component roughly looks like in Sygaldry.

```
struct ButtonGestureModel
{
```

```
    struct inputs_t {
        toggle<"button state", "1 indicates button active, 0 inactive"> button_state;
    } inputs;

    struct outputs_t {
        toggle<"debounced state", "updates only when the button state has changed"
                                  "since the last tick"> debounced_state;
        bng<"any edge", "bangs on any change to debounced state"> any_edge;
        bng<"rising edge", "bangs when the button is activated"> rising_edge;
        bng<"falling edge", "bangs when the button is deactivated"> falling_edge;
    } outputs;

    void main()
    {
        if (outputs.debounced_state != inputs.button_state)
        {
            outputs.debounced_state = inputs.button_state;
            outputs.any_edge();
            if (inputs.button_state) outputs.rising_edge();
            else outputs.falling_edge();
        }
    }
};
```

Numerous functional components are implemented, providing facilities for reading from analog and digital sensors, performing sensor fusion, and basic gesture modelling. A full listing of components, also including instruments, bindings, and helpers, is maintained in the documentation

for the library[5].

### 5.2.5 Concepts

Although endpoints and components tend to follow the patterns described above, in principle one of the goals of the library is not to impose strong requirements on the structure of their implementation. Bindings should instead adapt to the structure of components and endpoints, so that the latter can be implemented in whatever fashion is most appropriate for their functionality.

In order to consolidate the effort of adapting to different component shapes, Sygaldry includes a group of software components in the sygac "concepts" group that implement concepts, metafunctions, and generic functions whose purpose is to funnel the, in principle vast, variety of components and endpoints down to a more manageable interface for bindings to interact with.

The concepts component group has no physical dependency on the endpoint helpers and more broadly has no physical connection with component implementations, and as such components are, in principle, free to ignore the concepts library, which bears the burden of a logical dependency on all components and endpoints. In practice, it has been more convenient to limit the variety of components and endpoints in order to lighten the load on the concepts group.

### 5.2.6 Endpoint Reflection

The first major part of the concepts group is the endpoints concept software component[6]. This component provides several generic functions that facilitate interacting with endpoints in a generic manner. The most important of these are a pair of value accessor functions, generic functions for accessing expected metadata, and a group of metafunctions and concepts for inspecting the underlying value type of endpoints and (in the case of numerical arrays) their elements.

The value accessors allow bindings to get and set the current value of an endpoint in a generic manner; they are more robust implementations of the `value_of()` example above. At the time of

[5] `https://github.com/DocSunset/sygaldry/blob/8888ba2c4397cfe74aa1ba7d0c3767adbd04ea8b/sygaldry/docs/implementation.md`

[6] `https://github.com/DocSunset/sygaldry/blob/8888ba2c4397cfe74aa1ba7d0c3767adbd04ea8b/sygaldry/sygac-endpoints/sygac-endpoints.lili.md`

writing, an endpoint `v` is expected to have a data member `value`, or a member function `value()`, or to define a dereference operator `*v` in the case of occasional values. These cases are handled by compile-time branches in the generic function `sygaldry::get_value`. Setting values with `sygaldry::set_value` is even simpler, relying on endpoints to define an appropriate assignment operator so that the generic code doesn't require any branches, except for the special case of setting all elements of an array to the same value, which requires a compile time branch to recognize when an endpoint is an array that can be filled by the argument to the generic setter. The value accessors provide insulation from the shape of endpoints. If new kinds of endpoints are introduced whose values are accessed in different ways, it would be trivial to introduce additional compile-time branches to handle these cases.

Metadata accessors are provided to inspect the range, name, description, tags, and other metadata of an endpoint. These accessors closely match the available metadata helpers, but in principle they provide the same insulation as the value accessors, being trivial to augment to handle alternative representations of the relevant metadata.

Finally, a group of metafunctions and concepts are provided that play an important role in binding code. The metafunctions facilitate accessing the type of the underlying value of an endpoint, and the type of the elements of an array-valued endpoint. A concept is provided to recognize string-like data, such as string literals and `std::string_view`. Similarly, a concept is provided to recognize array-like values, which at the time of writing are expected to more or less resemble a `std::array`. These facilities provide functionality that allows bindings to branch over different types of endpoints in order to handle each appropriately, often resembling the following illustrative psuedo-code:

```
template<typename T> void handle_endpoint(T& endpoint) {
    using V = value_t<T>;
    if constexpr (std::integral<V>) { /* handle int */ }
    else if constexpr (std::floating_point<V>) { /* handle float */ }
    else if constexpr (string_like<V>) { /* handle string */ }
```

```
    else if constexpr (array_like<V>) { /* handle array */ }
    // etc
}
```

### 5.2.7 Component Reflection

As well as the reflection facilities for endpoints, the concepts component group also includes reflection facilities for components. These permit to recognize a component, iterate over its endpoints, and call its expected member functions generically. These functionalities are further generalised, in a departure from prior work, to also support nested aggregate structures containing only components and aggregate structures of components. We term these as assemblies. Whereas bindings in Avendish are designed to accept a single component, bindings in Sygaldry expect one component or an assembly containing numerous components and possibly nested assemblies. Expecting these nested tree-like structures enables the development of useful runtime execution systems, allows for basic dataflow to be expressed through the type system, and naturally leads to a flexible view of bindings as just another sort of component. These ramifications of allowing nested structures of components are discussed in more detail below.

The component reflection software component[7] permits clients to easily iterate over an assembly of components, a component, or one of the endpoint containers of a component. The iteration can be requested to visit components, all endpoints, inputs only, or outputs only with a lambda called for each requested entity; for example, the function `for_each_input` iterates generically over all of the input endpoints of components in an assembly, passing each to a generic lambda for processing. Combined with the endpoint reflection facilities, this makes writing bindings reasonably comfortable.

[7] `https://github.com/DocSunset/sygaldry/blob/8888ba2c4397cfe74aa1ba7d0c3767adbd04ea8b/sygaldry/sygac-components/sygac-components.lili.md`

### 5.2.8 Throughpoints, and Runtime

A notable contribution of the library, Sygaldry provides a facility termed as a runtime. A runtime is a class that, combining the component reflection and function reflection facilities of the library, automatically activates the components in an assembly, eliminating the need for boilerplate. The implementation of this functionality is highly obscure, likely not optimal, and will not be described in detail. The brave reader, familiar with C++ metaprogramming, may review the literate source code for more detail[8]. Given that the performance of this system remains unvalidated, at this time, it serves mainly as a strong proof of concept, and demonstrates some previously untapped potential of the broad approach of the library based on reflection and generic programming.

Components may implement the notion of throughpoints by requiring arguments to their principle subroutines. For example, a MIMU sensor fusion component may require a reference to raw MIMU data as an argument to its main subroutine. Using the typesystem and reflection facilities described above, it is then possible to locate the raw MIMU data in the assembly of components in order to pass a reference to it when activating the sensor fusion component. This is a very simple but fairly powerful way to represent dataflow in the component assembly.

In this mental model, a binding can be naturally viewed as just another kind of component, with a throughpoint[9] that accepts an assembly of components to expose over the foreign API. This is a very small cognitive shift, but it has the significant effect of very naturally integrating bindings into the dataflow framework that is so ubiquitous and familiar in the music technology field. The main quirk of this shift in perspective is that it requires to accept that whole components and assemblies of components can be passed around as data in the dataflow network, which is somewhat at odds with the usual perspective where it is the endpoints of components from and through which data flows. This is easily enough resolved if we view passing a component to a binding as equivalent to connecting all of its endpoints to the binding.

[8] `https://github.com/DocSunset/sygaldry/blob/8888ba2c4397cfe74aa1ba7d0c3767adbd04ea8b/sygaldry/sygac-runtime/sygac-runtime.lili.md`

[9] In the literate source of the runtime component, the situation when one component accepts another as argument is termed as a "plugin"; let it be understood that the conceptual framework at the time of implementing the runtime was still in flux, and that at the time of writing the thesis these terms are seen as basically equivalent.

As well as inspecting the type signature of every component's subroutines in order to pass it the needed data, the runtime also calls all of the principle subroutines automatically—first the initialisation subroutines, then the main subroutines in an endless loop. This eliminates boilerplate completely, so that the implementation of a DMI firmware is refined to a mere listing of its components, as seen in section 6.1. In between main loops, the runtime also clears the flags of occasional value type endpoints, so that their updates can be readily recognised and acted upon by components during the main loop.

### 5.2.9 Bindings

As just established, bindings are treated in Sygaldry as regular components that just happen to inspect the endpoints of other components as part of their regular operation. As well as having these notable throughpoints, bindings are distinguished from other components in one other important fashion. As well as initialisation and main subroutines, Sygaldry introduces external sources and external destinations as additional stages in the main loop. Bindings may define a subroutine `external_sources()` that is used to receive data from sources outside of the main component assembly represented by the running program; this may include, notably, control data received over a protocol such as OSC or MIDI used to update input endpoints in the main assembly. Similarly, the `external_destinations()` subroutine can be defined and is expected to propagate data from the main assembly to destinations outside of the currently running program. External sources subroutines are called before main subroutines which are called before external destination subroutines. This way, main subroutines can react to changes triggered by external sources, and external destinations receive updates that took place in the most recent main loop. This gently extends the setup-loop execution model adopted from Arduino and nicely enables the integration of bindings in the Sygaldry runtime framework.

For an example of a binding component, the reader is referred to the literate source code of the liblo binding[10], or the description of the component below in section 5.3.5

[10] `https://github.com/DocSunset/sygaldry/blob/8888ba2c4397cfe74aa1ba7d0c3767adbd04ea8b/sygaldry/`

### 5.2.10 Platforms

In the work thus far using the library, there has not generally been occasion to make an instrument that used only a subset of the bindings available for a given platform. Recognizing this, platform classes extend the basic runtime so that all available bindings are available to clients without having to list them all out. This is achieved by what amounts to a partial instrument firmware implementation, where all of the available bindings are listed out with an additional template type placeholder for the final assembly to occupy when the runtime is instantiated.

For an example, the substantive implementation for the platform class for ESP32 is excerpted below. The `struct Instrument` is a template for an instrument assembly that includes components for I2C, WiFi, OSC bindings, and bindings to persistent data storage and a command line interface over USB serial. The user of the platform can then simply list the components of their instrument in a struct passed as template argument to `ESP32Instrument` and easily access all of this functionality, as well as a `void app_main()` function implementation.

```
template<typename InnerInstrument>
struct ESP32Instrument
{
    struct Instrument {
        struct Components {
            sygsa::TwoWire<syghe::I2C_MAIN_SDA,syghe::I2C_MAIN_SCL,400000> i2c;
            InnerInstrument instrument;
            sygbe::WiFi wifi;
            sygbp::LibloOsc<InnerInstrument> osc;
        };
        sygbe::SpiffsSessionStorage<Components> session_storage;
        Components components;
```

sygbp-liblo/sygbp-liblo.lili.md

```
        sygbp::CstdioCli<Components> cli;
    };

    static_assert(Assembly<Instrument>);

    static inline Instrument instrument{};

    void app_main()
    {
        constexpr auto runtime = Runtime{instrument};
        // give electrical conditions a moment to settle in
        vTaskDelay(pdMS_TO_TICKS(1000));
        printf("initializing\n");
        runtime.init();
        // give IDF processes time to finish up init business
        vTaskDelay(pdMS_TO_TICKS(100));
        printf("looping\n");
        while (true)
        {
            runtime.tick();
            vTaskDelay(pdMS_TO_TICKS(10));
        }
    }
};
```

### 5.2.11 Instruments

The culmination of combining all of the above elements, instruments are implemented by listing the required components in an aggregate struct that is passed to a platform runtime. Instruments are fairly trivial to implement in Sygaldry, because all of the hard work is consolidated into the library. More detail is provided in section 5.3.2 and section 6.1.

## 5.3 Using Sygaldry

With the aim of further clarifying the design and implementation of the library, this section provides several examples of how Sygaldry is employed in use. The purpose of this section is to provide additional detail to the presentation of the design and implementation of the library itself, rather than to motivate for its usefulness, which is addressed in chapter 6. This section is also not meant to provide a comprehensive up to date tutorial on using Sygaldry. Readers living in the distant future relative to the time of writing may wish to consult the project README for more up to date documentation on how to use the library.

### 5.3.1 Getting Started

Sygaldry is a C++ library distributed as source code. Strictly speaking, it is not necessary to install anything to use the library; the user may choose to download and integrate the source code in their build system manually. However, for the user's convenience, Sygaldry provides CMake integration and build tool software components that facilitate installing toolchains for supported platforms. In order to use these build tools, which consist of a number of shell scripts, the user may manually install the command line executables needed to run the scripts, such as CMake, and then proceed. Alternatively, for convenience, an appropriate shell environment with all needed tools can be installed automatically using the Nix package manager.

In order to download the library and all of its source code dependencies, the recommended method is to use git:

```
git clone --recurse-submodules https://github.com/docsunset/sygaldry
```

The upstream recommendation for installing Nix consists of the following one-liner:

```
sh <(curl -L https://nixos.org/nix/install) --daemon
```

Finally (after launching a fresh terminal), another line will install the build tool component dependencies and enable the build environment:

```
cd /path/to/repo/sygaldry ; nix-shell --pure
```

The user will then be presented a command prompt and can immediately make use of all of the build tool components in the library. Subsequent command line excerpts are assumed to run from the `nix-shell --pure` environment with the root of the Sygaldry repository as the working directory.

### 5.3.2 Instruments: Implementing the T-Stick

In the best case, when all of the components needed to implement an instrument's firmware are already present in the library, implementation is very straight forward. Using the runtime component for a given platform, the user need simply declare the required components in a struct, pass this to the runtime component as a template parameter, instantiate the runtime, and call its main loop routine. The full unabridged firmware for the T-Stick is presented below with comments demarcating these steps.

```
/* src/main.cpp *********************************/
// include necessary components

#include "sygse-button.hpp"
#include "sygse-adc.hpp"
#include "sygsa-two_wire.hpp"
```

```
#include "sygsa-trill_craft.hpp"
#include "sygsp-icm20948.hpp"
#include "sygsa-two_wire_serif.hpp"
#include "sygsp-complementary_mimu_fusion.hpp"
#include "sygbe-runtime.hpp"
#include "sygup-debug_printer.hpp"
#include "sygup-cstdio_logger.hpp"

// namespace declaration to save having to
// explicitly qualify "sygaldry::" on everything
using namespace sygaldry;

// declare a struct with all required components
struct TStick
{
    sygse::Button<GPIO_NUM_21> button;
    sygse::OneshotAdc<syghe::ADC1_CHANNEL_5> adc;
    sygsa::TrillCraft touch;
    sygsp::ICM20948< sygsa::TwoWireByteSerif<sygsp::ICM20948_I2C_ADDRESS_0>
                   , sygsa::TwoWireByteSerif<sygsp::AK09916_I2C_ADDRESS>
                   > mimu;
    sygsp::ComplementaryMimuFusion<decltype(mimu)> mimu_fusion;
};

// pass the declaration to the runtime and instantiate the runtime
sygbe::ESP32Instrument<TStick> tstick{};
```

```
// call main; ESP-IDF requires this to have C linkage
extern "C" void app_main(void) { tstick.app_main(); }
```

In order to be able to compile this, some platform-specific CMake boilerplate must also be supplied. In most cases, new instruments could copy the boilerplate from the T-Stick as a starting point:

```
# root CMakeLists.txt ###################################
cmake_minimum_required(VERSION 3.16)
# ESP-IDF project include;
# the IDF_PATH environment variable is set automatically
# by the build tool component for ESP-IDF
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(t-stick)

# src/CMakeLists.txt ####################################
# declare an ESP-IDF component for the firmware
idf_component_register(SRCS "t_stick.cpp"
        )

# set some configuration variables specific to ESP-IDF
set(SYGALDRY_I2C_PINS 1)
set(SYGALDRY_SDA 12)
set(SYGALDRY_SCL 11)

# make Sygaldry itself available as a library
add_subdirectory(../../../ sygbuild)

# some compile options that could be freely customized
```

```
target_compile_options(${COMPONENT_LIB} PRIVATE
        "-Wfatal-errors"
        "-Wno-error=unused-but-set-parameter"
        "-ftemplate-backtrace-limit=0"
        )

# link to Sygaldry
target_link_libraries(${COMPONENT_LIB} PRIVATE sygaldry)
```

The CMake boilerplate tends to be extremely basic and invariant across instruments, essentially reflecting the minimum working example provided by the manufacturer of the hardware for a given platform. It remains as future work to automate this step, such as by providing template CMake files from which the final boilerplate can be generated for each specific instrument. In any case, although the CMake boilerplate introduces an annoying and unsightly extra step in the build process compared to e.g. simply pushing "compile" in Arduino, the long term maintenance burden is relatively small since this boilerplate rarely changes.

Individual platforms may have additional configuration steps or peculiarities. In the case of ESP-IDF, it's necessary to provide a file `sdkconfig` as well as a partition table. These are included in the full literate source code for the T-Stick firmware, but will be glossed over here.

Once the source code and build automation is written, compiling and uploading the firmware is done using the standard toolchain for a given platform. To facilitate installing and using these, build tool components are provided for each platform. For example, to compile and upload the ESP32 T-Stick firmware, the following command invoked from the Nix shell will suffice, installing the toolchain the first time it is run, before building and flashing the firmware found in the directory `sygaldry-instruments/t_stick_esp32s3`:

```
idf.sh t_stick_esp32s3 build flash
```

### 5.3.3 Functional Components: Implementing a Keyboard Scanner

In case the user wishes to implement an instrument requiring a functional component that does not already exist in the library, it will be necessary to implement a new component. The shell script `new_component.sh` can be run to generate the required boilerplate; this script will prompt the user for the name, identifier, and description of the new component, along with other useful metadata. Alternatively, the required information can be provided using command line arguments, as shown below. Supposing a new component should be created for scanning an array of infrared LEDs connected to phototransistors in a continuous keyboard sensing mechanism ([111]), the user might invoke the following command[11] from the Nix shell:

```
new_component.sh -p sygsp \ # package group ID
                 -c continuouskeys \ # package ID
                 -t "Continuous Keyboard Scanner" \ # Component title
                 -n "Continuous Key Scanner" \ # Component name (used in OSC addresses)
                 -s "KeyScanner" \ # C++ type name of the component struct
                 -d "Scanning logic for a continuous keyboard sensor" \ # description
                 -a "Travis J. West" \ # author
                 -l MIT \ # license
```

This generates a directory `sygaldry/sygsp-continuouskeys` containing literate and machine source files pre-populated with boilerplate for the component structure, unit tests, and CMake build automation, including license boilerplate comments and basic Doxygen documentation comments.

At this point, the user has an empty component that looks something like the following (with the comments removed for readability):

```
// @#'sygsp-continuous-key-scanner.hpp'

#pragma once
```

[11] minus the comments, which are strictly speaking not valid shell syntax

```
#include "sygah-metadata.hpp"
#include "sygah-endpoints.hpp"

namespace sygaldry { namespace sygsp {
struct KeyScanner
: name_<"Continuous Key Scanner">
, description_<"Scanning logic for a continuous keyboard sensor">
, author_<"Travis J. West">
, copyright_<"Copyright 2025 Sygaldry Contributors">
, license_<"SPDX-License-Identifier: MIT">
, version_<"0.0.0">
{
    struct inputs_t {
    } inputs;

    struct outputs_t {
    } outputs;

    void init()
    {
    }

    void main()
    {
    }
};
} }
```

All that remains is to implement the behavior of the component. In this example, an array of infrared sensors is connected to a keyboard in order to measure the movement of the keys continuously ([111]). Each infrared sensor consists of an infrared LED that emits light and a phototransistor that measures the reflected light from the LED. The closer the key is to the sensor, the higher the amount of light reflected and thus the higher the output voltage. All of the sensors are connected to one ADC input, since only one sensor is ever engaged at a time by controlling the LEDs so that only one is ever lit.

The component should switch on one infrared LED, read the voltage output from the associated phototransistor, store that value, then move to the next LED. We assume that the LEDs are configured as a matrix, so that each LED can be addressed by a 2-dimensional coordinate. We will output this coordinate from the scanner component and defer to another component for the hardware-specific task of controlling the GPIO of the microcontroller to engage the LEDs. In the struct of outputs, we declare an array to store the readings and another to output the coordinate of the LED that should be lit. The number of rows and columns is determined by template parameters, as shown in more detail below.

```
struct outputs_t {
    array_message<"keys", ROWS*COLUMNS, "sensor output state", float> keys;
    array_message<"LED coordinate", 2
                , "coordinate of the lit LED in the matrix", char> leds;
} outputs;
```

It's convenient to read the phototransistor values using the existing ADC component. Along with the LED driver component, this establishes a modular design where the key scanning logic is completely platform independent, since all of the interaction with hardware is handled by the other components. A consequence of this design is that the scanner's main subroutine operates inside of the main loop. In each loop, the ADC runs, then the scanner, then the LED driver. This means the scanner needs to keep track of which LED is lit and which key of the keyboard that

LED is associated with. This is accomplished with a loop variable `current_key` that is used both to index into the array of key values and to set the LED coordinates:

```
std::size_t current_key{};


// output the coordinate in the matrix for the LED
// that should be lit based on the value of current_key
void set_led() {
    outputs.leds = { static_cast<char>(current_key % COLUMNS)
                   , static_cast<char>(current_key / COLUMNS)
                   };
}
```

Before the main loop starts, the initialisation subroutine is called. We use this to reset the current key and illuminate the LED at coordinate $(0, 0)$:

```
void init()
{
    current_key = 0;
    set_led();
}
```

Then, in each call to the main loop subroutine, we grab the most recent value from the ADC, store it in the keys array, then switch to the next LED so that it's associated phototransistor will be read by the ADC on the next loop.

```
void main(const adc_reading_t& adc_reading)
{
    // store the current ADC state, associated with the current key
    outputs.keys[current_key] = adc_reading;
```

```
    // switch to the next key,
    // restarting the scan loop after scanning the last key
    if (++current_key >= ROWS*COLUMNS) current_key = 0;

    // output the current LED coordinate
    set_led();
}
```

In order to access the ADC reading, a throughpoint is implemented (section 2.5). The scanner component is thus a model that integrates the raw readings from the ADC with the knowledge of which key is currently activated in order to produce the resulting array of key values.

To implement the throughpoint, a template parameter is added to the scanner component to receive the type signature of the ADC reading. An argument is added to the main subroutine using this type. The runtime component will detect this automatically at compile time and pass the associated value to the subroutine at runtime. Assuming the infrared LED/phototransistor pairs are built in a matrix configuration, we also pass the number of rows and columns in the matrix as template parameters, using these to set the size of the array where the ADC values are stored and to recognize when we have reached the end of the scan loop, as seen above.

This leads to the final implementation, collected below from the snippets above:

```
// the component user configures the scanner with the type signature of the ADC reading
// and the number of rows and columns
template<typename adc_reading_t, std::size_t ROWS, std::size_t COLUMNS>
struct KeyScanner
// ... metadata ...
{
    struct inputs_t {} inputs; // no runtime inputs
```

```
    struct outputs_t {
        array_message<"keys", ROWS*COLUMNS, "sensor output state", float> keys;
        array_message<"LED coordinate", 2, "coordinate of the lit LED in the matrix", char>
    } outputs;

    std::size_t current_key{};

    void set_led() {
        outputs.leds = { static_cast<char>(current_key % COLUMNS)
                       , static_cast<char>(current_key / COLUMNS)
                       };
    }

    void init()
    {
        current_key = 0;
        set_led();
    }

    void main(const adc_reading_t& adc_reading)
    {
        outputs.keys[current_key] = adc_reading;
        if (++current_key >= ROWS*COLUMNS) current_key = 0;
        set_led();
    }
};
```

This makes a nearly complete, if basic, implementation of a continuous keyboard scanner firmware. Here the direct interaction with the LED matrix is handled by another component; this is reasonable so that the scanning logic can be implemented completely independent from the hardware. The implementation of the LED driver component is left to the reader as an exercise.

The new instrument firmware can then be declared as follows:

```
unsigned int row_pins[] = {5, 4, 3, 2, 1, 0};
unsigned int col_pins[] = {8, 7, 6, 11, 10, 9};

struct ContinuousKeyboard {
    sygse::OneshotAdc<syghe::ADC1_CHANNEL_5> adc;
    sygsp::KeyScanner< decltype(adc.outputs.raw)
                     , std::size(row_pins), std::size(col_pins)
                     > scanner;
    sygsr::LedMatrixDrive< decltype(scanner.outputs.leds)
                         , std::size(row_pins), std::size(col_pins)
                         , row_pins, col_pins
                         > pin_driver;
};

sygbe::ESP32Instrument<ContinuousKeyboard> runtime{};
int main(){runtime.main();}
```

Given a real implementation of the LED driver, along with build automation boilerplate copied from above, this makes a functional implementation of a continuous keyboard sensor firmware.

### 5.3.4 Functional Components: Using Existing Libraries

Compared to writing a bespoke firmware from scratch, e.g. using Arduino, the main complication Sygaldry introduces is that, in order to make good use of the existing components in the library, it's often necessary to adapt the design of the new component to fit naturally into the dataflow-oriented runtime loop implemented by Sygaldry. The advantage, if this cognitive adaptation is adopted, is that the resulting new component that the user implements could be readily adopted and reused by other users of the library. In case the user did not wish to take on the challenge of learning to think in the framework Sygaldry is predisposed to, however, it would be no more difficult to implement a monolithic hardware-dependent version of the component in Sygaldry than it would be to write a new firmware from scratch. For example, the keyboard scanner component above could just as well have been implemented with its own inner loop that directly reads the ADC, drives the LEDs, and so on all in the main subroutine. The cognitive adaptation to Sygaldry's mental model is entirely optional.

The situation is more fraught for Sygaldry in case a library already exists in another environment for using a certain functionality. For example, the Trill touch sensors from Bela are well supported by an Arduino library. In this case, using Sygaldry may impose a bit of additional effort compared to directly using the Arduino library in a bespoke Arduino firmware written from scratch. In the worst case, as is seen in Sygaldry's MIMU component, the user might have to write their own driver for interacting with a given sensor. In the best case, as seen in Sygaldry's component for the Trill sensor, the existing library can be used to implement the component in Sygaldry style. The particularities of these situations vary substantially depending on the implementation of the existing 3rd party library. The reader is encouraged to examine the implementation of the mentioned components for more information on these cases. There is further discussion of this issue in section 5.4.2.

### 5.3.5 Bindings: Implementing an OSC Binding

Sygaldry currently supports sending data from the firmware to a hosted application processor such as a laptop using a handful of protocols. An ad hoc USB serial command line interface is provided by the `sygbp-cli` component. The `sygbp-liblo` component provides OSC protocol bindings using the liblo library. On ESP-IDF based microcontrollers, the `sygbe-spiffs` component provides persistent storage of endpoint state across boot cycles. It remains as future work to implement bindings for MIDI, and eventually other protocols and platforms. As an example of how bindings may be implemented, we will examine the implementation of the liblo OSC binding at a high level.

The initial process for implementing a binding is the same as for a functional component. The new component script can be used to generate the boilerplate of a new component. Right away, a few additions will be necessary, however. Bindings work by reflecting over other components. In order to access these other components, a throughpoint is used. A template parameter `typename Components` is added to receive the type signature of what is termed as an assembly (section 2.2): this can be an individual component, or an aggregate struct of nested assemblies. This gives the following high-level skeleton of the binding:

```
template<typename Components>
struct LibloOsc
: name_<"Liblo OSC">
, author_<"Travis J. West">
, copyright_<"Copyright 2023 Travis J. West">
, license_<"SPDX-License-Identifier: LGPL-2.1-or-later">
, version_<"0.0.0">
, description_<"Open Sound Control bindings using the liblo library">
{
    struct inputs_t {
        // ...
```

```
} inputs;

struct outputs_t {
    // ...
} outputs;

// ... data members and subroutines ...

void init(Components& components)
{
    // ...
}

void external_sources()
{
    // ...
}

void main(Components& components)
{
    // ...
}

void external_destinations(Components& components)
{
    // ...
}
```

```
};
```

Compared to a typical functional component, bindings have two extra subroutines, `external_sources` and `external_destinations`. These are used to receive input from the the external protocol and to send outputs to it, respectively.

### Setting Things Up

The liblo component has input endpoints for the destination port and IP, where messages are sent to, and for the incoming port, where messages are received. As well, output endpoints reflecting the state of the server, i.e. whether it is currently able to send or receive data based on the configuration of the input endpoints, are provided. These endpoints can be configured and inspected using e.g. the USB serial command line interface, or another binding that reflects over the liblo component.

A large portion of the implementation of the component has to do with reacting to changes to the input endpoints, reconfiguring the server as needed. These details will be glossed over here in order to focus on the more interesting use of the reflection features of the libary; the reader may refer to the literate source code for all the details[12].To help simplify the discussion, we will assume that the source port as well as the destination port and address are all reasonably configured. We will also not delve into too much detail on the use of liblo itself. The bare minimum of context will be provided, in hopes that readers unfamiliar with liblo will not be completely lost, but we will largely gloss over the why and how of using liblo, as well as regularly omitting error checking code that complicates readability.

```
struct inputs_t {

    text_message< "source port"
                , "The UDP port on which to receive incoming messages."
                , tag_session_data
```

[12] https://github.com/DocSunset/sygaldry/blob/8888ba2c4397cfe74aa1ba7d0c3767adbd04ea8b/sygaldry/sygbp-liblo/sygbp-liblo.lili.md

```
                > src_port;
    text_message< "destination port"
                , "The UDP port on which to send outgoing messages."
                , tag_session_data
                > dst_port;
    text_message< "destination address"
                , "The IP address to send outgoing messages to."
                , tag_session_data
                > dst_addr;
} inputs;

struct outputs_t {
    toggle<"server running"> server_running;
    toggle<"output running"> output_running;
} outputs;
```

In the initialisation subroutine, we call subroutines to set up the `lo_server` used for receiving OSC and the `lo_address` used for sending OSC. The same subroutines are called in the main subroutine in order to respond to changes to the input endpoints.

```
lo_server server{}; // this is used to receive OSC from the network
lo_address dst{}; // this is used to send OSC to the network

void init(Components& components)
{
    outputs.server_running = 0; // so that set_server will try to set up "server"
    set_server(components);
    outputs.output_running = 0; // so that set_dst will try to set up "dst"
```

```
    set_dst();
}

void main(Components& components)
{
    set_server(components);
    set_dst();
}
```

The `lo_address` simply holds the port and IP address to which OSC messages should be sent. We'll skip over the implementation of `set_dst()`, which simply checks if the destination port or address has changed, and if so makes sure they're valid before updating `dst`, reporting any errors that arise.

The `lo_server` plays a similar role, holding the port on which to listen for incoming OSC messages. In addition, this object also keeps track of call back functions associated with different OSC addresses. Setting up the server is thus the first instance of reflection in the binding, when the callbacks are registered. This uses the Sygaldry reflection function `for_each_input` which takes as arguments a reference to an assembly and a generic lambda that will be called for each input endpoint of the components in the assembly.

In setting up the liblo server, the `for_each_input` lambda body adds a liblo callback method for each input endpoint. The OSC path and type tag string are derived from the endpoint's metadata, and a reference to the endpoint itself is passed in the form of a `void *` so that the liblo callback can later access the endpoint.

```
// add a method for each input endpoint in the assembly "components"
// "T" is the type signature of the endpoint, and "in" is the endpoint itself
for_each_input(components, [&]<typename T>(T& in)
{
```

```
    // ... define the callback function and put a function pointer in "handler" ...

    lo_server_add_method( server
                        , osc_path_v<T, Components>, osc_type_string_v<T>+1
                        , handler, (void*)&in
                        );
});
```

The OSC path and type tag string of the endpoint are determined using compile-time metafunctions that reflect over the metadata of the endpoint (i.e. its name and type, and the names of its parents in the `Components` assembly). The details of these metafunctions are quite interesting, but a bit too involved, and not relevant to implementing a binding unless for the OSC protocol; the interested reader is referred to the `sygbp-osc_string_constants` component's literate source code[13].

The definition of the handler involves a bit of unusual syntax. Liblo needs to receive a pointer to a plain old C-style function. Each endpoint needs its own callback function. Here we use a non-capturing lambda to define the callback function, since such a lambda can be converted to a C-style function pointer. We prefix the lambda expression with a unary-plus "`+`" to cause it to be converted to a function pointer. That the unary plus has this effect is at first surprising. Basically, the unary plus requires an argument to which it can apply. Unary plus of a non-capturing lambda is not valid, but unary plus of a function pointer is. This forces the lambda to implicitly convert into a function pointer, but is otherwise a no-op. We gratefully use `auto` to avoid having to spell out the type signature of the function pointer:

```
// "handler" has type void(*)( /* args */ )
constexpr auto handler = +[]( /* ... arguments ... */ )
{
```

[13] https://github.com/DocSunset/sygaldry/blob/8888ba2c4397cfe74aa1ba7d0c3767adbd04ea8b/sygaldry/sygbp-osc_string_constants/sygbp-osc_string_constants.lili.md

```
    // ...
};
```

The handler itself simply extracts the endpoint from the user data pointer and then defers to a static method of the liblo Sygaldry component that sets the input based on the OSC message's arguments:

```
constexpr auto handler = +[]( const char *path, const char *types
                            , lo_arg **argv, int argc, lo_message msg
                            , void *user_data
                            )
{
    #ifndef NDEBUG
        fprintf(stdout, "liblo: got message %s", path);
        lo_message_pp(msg);
    #endif
    T& in = *(T*)user_data;
    LibloOsc::set_input(path, types, argv, argc, msg, in);
    return 0;
};
```

The overall effect of setting up the server is that whenever a message is received with an OSC address pattern that corresponds to one of the input endpoints of the assembly, the liblo server invokes a callback such that the value of the endpoint is updated based on the arguments of the OSC message.

### External Sources

The external sources subroutine simply invokes the liblo server so that any incoming messages will be dispatched to the appropriate callback handlers. The non-blocking receive method ("`lo_server_recv_noblo`

is used with 0 wait time so that if no messages have arrived we won't waste time waiting for them.

```
void external_sources()
{
    if (outputs.server_running) lo_server_recv_noblock(server, 0);
}
```

The server will in turn invoke the callbacks registered during set up, causing the `set_input` method to be invoked. This method provides a very clear and detailed example of the kind of implementation involved in all bindings. The method is a template function parameterised on the type signature of the endpoint that should be updated:

```
template<typename T>
static void set_input( const char *path, const char *types
                     , lo_arg **argv, int argc, lo_message msg
                     , T& in
                     )
{
    // ...
};
```

The body essentially consists of a sequence of compile time branches. Each branch reflects over the type signature of the endpoint to try to determine how the OSC arguments should be interpreted. As soon as enough information has been determined through this process to interpret the OSC arguments, they can be verified appropriately and used to set the value of the endpoint using the generic `set_value` function that takes as arguments an endpoint and a value to which it should be set.

In pseudocode, the body amounts to the following:

```
if the endpoint is an impulse, trigger it;
else if the endpoint is a float,
```

```
    check if the OSC argument is a float and if so set the endpoint
else if the endpoint is an integer,
    check if the OSC argument is an integer and if so set the endpoint
else if the endpoint is a string,
    check if the OSC argument is a string and if so set the endpoint
else if the endpoint is an array,
    iterate over its elements and try to set each of them,
    then mark the array as updated if it has an "updated" flag
```

The actual code basically looks as follows. I've removed the error checking for clarity; in reality, if for some reason the type of the OSC argument does not match the type of the endpoint, we simply report it and return from the callback.

```
if constexpr (Bang<T>) in = true;
else if constexpr (std::integral<value_t<T>>)       set_value(in, argv[0]->i);
else if constexpr (std::floating_point<value_t<T>>) set_value(in, argv[0]->f);
else if constexpr (string_like<value_t<T>>)         set_value(in, &argv[0]->s);
else if constexpr (array_like<value_t<T>>)
{
    for (std::size_t i = 0; i < size<value_t<T>>(); ++i)
    {
        auto& element = value_of(in)[i];
        if constexpr (std::integral<element_t<T>>)            element = argv[i]->i;
        else if constexpr (std::floating_point<element_t<T>>) element = argv[i]->f;
        else if constexpr (string_like<element_t<T>>)         element = &argv[i]->s;
    }
    if constexpr (UpdatedFlag<T>) in.updated = true;
}
```

This kind of tree of compile-time branches is at the heart of all binding components. The fact that the number of branches grows in the number of kinds of endpoints is a notable concern for the long-term maintainability of the library, and is further discussed in section 5.4.3.

### External Destinations

The external destinations subroutine is the mirror image of the external sources subroutine. Here we iterate over all of the output endpoints in the assembly and add an OSC message to a bundle for each endpoint that has updated before sending the bundle to the configured destination.

```
void external_destinations(Components& components)
{
    if (outputs.output_running)
    {
        lo_bundle bundle = lo_bundle_new(LO_TT_IMMEDIATE);
        for_each_output(components, [&]<typename T>(T& output)
        {
            // ... output OSC messages for each output that has changed ...
        });
        int ret = lo_send_bundle(dst, bundle);
        lo_bundle_free_recursive(bundle);
    }
}
```

The loop body is somewhat simpler than for inputs. We'll gloss over the code that checks if the endpoint has been updated, as well as the error checking, assuming that the endpoint was updated and there are no errors. We then allocate an OSC message and populate it with arguments, if the endpoint has any values, before adding the message to the bundle.

```
// .. check if the endpoint has been updated and return if not
```

```
lo_message message = lo_message_new();

if constexpr (has_value<T> && not Bang<T>)
{
    // ... populate OSC arguments ...
}

lo_bundle_add_message(bundle, osc_path_v<T, Components>, message);
return;
```

To add arguments to the message (once again ignoring the need for error checking), we reflect over the endpoint's type to determine the OSC type tag and the appropriate liblo method to call. This is less involved than for inputs since we can use liblo's `lo_message_add` method in most cases, which can take numeric arguments generically by referring to the type tag string at runtime.

```
constexpr auto type = std::integral<element_t<T>> ? "i"
                    : std::floating_point<element_t<T>> ? "f"
                    : string_like<element_t<T>> ? "s" : "" ;

if constexpr (string_like<value_t<T>>)
    lo_message_add_string(message, value_of(output).c_str());
else if constexpr (array_like<value_t<T>>)
{
    for (auto& element : value_of(output))
        lo_message_add(message, type, element);
}
else lo_message_add(message, type, value_of(output));
```

#### Summary

The liblo OSC component enables all of the endpoints in the instrument firmware to be externally influenced over OSC protocol, as well as forwardng output endpoint data over the network whenever the endpoints are updated. Most bindings will share many characteristics in common with this implementation. In particular, most bindings will have somewhere in their implementation a call to `for_each_input`, and another to `for_each_output`, both containing compile time branches based on inspecting the type signature and metadata of the endpoints. The particularities of the implementation will depend on the exact protocol or environment to which the assembly should be glued, but this component provides a reasonably good example of what bindings tend to look like.

### 5.3.6 Platforms: Supporting PicoSDK

In the worst case scenario, a Sygaldry user may want to implement their instrument using a platform that is not already supported by the library. This case represents the biggest barrier to adopting Sygaldry, since it remains fairly involved to add a new platform. This section will provide some very high level consideration of the steps involved, without delving excessively into the details, by describing the process of adding support for PicoSDK to the library. Further discussion of the issues involved in this process can be found in section 5.4.2.

As the initial minimum viable version of the library was developed to replicate the T-Stick, which at the time used the ESP32 microcontroller in its implementation, the first platform supported by Sygaldry was ESP-IDF. The library components that make up support for this platform include the build tool components for the framework, and all of the platform-specific software components. This includes, notably, support for the SPIFFs flash memory file system, WiFi, ADC, GPIO, and a bare-bones implementation of the Arduino API used in the implementation of the Trill touch sensor and MIMU sensor components. These components are difficult to implement without depending on hardware-specific APIs, and constitute a non-portable layer in the library

with which other more portable components can interact.

In the port of the T-Stick to PicoSDK, it was necessary to implement PicoSDK-specific versions of the Arduino implementation, ADC, and GPIO components. As the behavior of the PicoSDK's `cstdio` implementation differed from that of the ESP-IDF, it was also necessary to write a small component adapting the portable command line interface component to work with PicoSDK's `cstdio`. Of these efforts, the barebones Arduino implementation constituted the most significant portion of the effort, since the functionality of the other components are all quite trivial. Better Arduino compatibility is an important target for future work (see section 5.4.2).

It is not strictly necessary to implement all platform-specific functionalities for every platform. For example, the PicoSDK support does not include a flash memory component, since the PicoSDK microcontrollers do not natively possess hardware dedicated to user data storage in flash (although it is possible to store data in the flash memory where program machine code is stored). Thus, the process of adding support for a new platform essentially consists of implementing the hardware-specific components required for the new application.

There is more said about the issue of supporting additional platforms in section 5.4.2. Notably, because Sygaldry does not at this time introduce its own hardware abstraction API, this process is relatively involved compared to writing a new component or binding. Future work should provide better support for existing hardware abstraction layers, such as Arduino or one or more of the various real-time operating systems that provide hardware abstraction, rather than directly targeting bare-metal hardware APIs as is currently done. Such hardware abstraction APIs would be considered as software platforms in Sygaldry's mental model, and would more efficiently enable support for multiple hardware platforms.

## 5.4 Limitations and Future Work

I believe the potential benefit to be had from developing a library of reusable portable DMI components is enormous, and that the techniques described, using C++, provide an efficient route to achieve these benefits. However, the present work has notable limitations, and there remains

significant future work to realise the full expected benefits of such a library. This section enumerates some of the prominent areas where improvements may be made, including the extensibility, compatibility, and scope of the library, its operational and dataflow models, and consideration for the long-term evaluation and repercussions of the library.

### 5.4.1 Extending Components of the Library

Software is more maintainable when software components can be extended without physical modification [110]. Although it is quite possible to use one Sygaldry component in the implementation of another, there are limitations to how Sygaldry components can be extended, due to the architecture of components and constraints of the reflection methods that are available in C++ at the time of writing. Specifically, because it is not allowed to iterate over the data members of a derived class, there is no convenient way for one component to derive its endpoints from another component. For example, suppose there is a component for mapping button gestures such as the one shown in section 5.2.4. In the implementation of a button component for a given hardware platform, it would be convenient to start with the functionality of this gesture model, perhaps extending it with some additional endpoints that are specific to the hardware involved on that platform. Ideally, we might write something like this:

```
struct HardwareButton
{
    struct inputs_t : public ButtonGestureModel::inputs_t {
        // additional inputs
    } inputs;

    // similarly for outputs

    void main() {
        // additional hardware specific main loop procedures
```

```
        ButtonGestureModel::main(inputs, outputs);
        // additional hardware specific main loop procedures
    }
};
```

While it is easy enough to modify `ButtonGestureModel` in this example to accept inputs and outputs that are derived from its own endpoint structures, such as by defining a static template main function, because it is not possible to reflect over the data members of a derived class, there is no convenient way for something like `HardwareButton` to extend the endpoints of `ButtonGestureModel`. It is expected that by C++26, the reflection functionality of the language should have developed to the point where this issue will sort itself out, but in the meantime this limits the extensibility of Sygaldry components and is a likely cause of maintenance issues.

One possible workaround that was explored during the development of the library is to allow nested structures of endpoints. In this case, the hypothetical above might be realised as follows:

```
struct HardwareButton
{
    struct inputs_t {
        ButtonGestureModel::inputs_t parent_inputs;
        // additional inputs
    } inputs;

    // similarly for outputs

    void main() {
        // additional hardware specific main loop procedures
        ButtonGestureModel::main(inputs.parent_inputs, outputs.parent_outputs);
        // additional hardware specific main loop procedures
```

```
    }
};
```

While somewhat less convenient in implementation, this approach, favoring composition to enable extension, is very likely workable. However, it was found that allowing nested structures of endpoints gives rise to some tricky implementation challenges and design choices. It is sometimes difficult to distinguish between an endpoint and a structure of endpoints. Because inheritance is used extensively by the endpoint helper package, it is not allowed to reflect over an endpoint in Sygaldry and attempting to do so will trigger a difficult compiler error. Further, allowing nested structures of endpoints begs the questions: how should this tree-like structure of endpoints be handled when e.g. generating OSC addresses for each endpoint? In the interest of time, these challenges were set aside as future work during the present research.

### 5.4.2 Compatibility

#### Arduino Compatibility

The current mainline T-Stick firmware (`https://github.com/IDMIL/T-Stick`) makes extensive use of the Arduino API to avoid including platform-specific dependencies. This is a common portability strategy, and means that in principle most of the firmware should be portable to any Arduino microcontroller. In practice, this portability is undercut by the use of libmapper-arduino (`https://github.com/libmapper/libmapper-arduino`), and Puara (`https://github.com/Puara/puara-module`) libraries, which both depend directly on ESP-IDF components (at least at the time of writing). Nevertheless, one of Sygaldry's main limitations is its reliance on modern C++20; many Arduino implementations are incompatible with this C++ version, which precludes the use of Sygaldry on those platforms. Because of this limitation, in many cases Sygaldry may be effectively less portable than a more typical Arduino-based firmware implementation; as well as having to port any hardware-specific components, it may also be necessary to write a minimal Arduino API implementation. It is hoped that over time this limitation will reduce as third-

party Arduino implementers update their implementations to support newer C++ standards. For example, the ESP-IDF Arduino implementation was not C++20 compatible when Sygaldry was first implemented, but has been updated so that it is now possible to use C++20 with this Arduino implementation.

One unfortunate manifestation of the lack of Arduino support in Sygaldry is seen in the implementation of components that interact with common hardware, such as the ADC components for ESP32 and PicoSDK, as well as the button components. These components are almost identical across these two platforms, differing only in the use of the platform-specific vendor API used to interact with the ADC or GPIO pins. At some point, as more platforms become supported, it seems warranted that these kinds of components should target a portable hardware-abstract API instead of directly using the vendor-supplied API for interacting with this kind of very common microcontroller hardware. This is exactly where the Arduino API shines; as the defacto standard API for abstracting away common microcontroller hardware, it provides for more portable software components that interact with the abstracted hardware addressed by the API. The last thing that we want to happen is for Sygaldry to become a new competing standard for hardware abstraction. Although this issue does not strongly impact on the main contributions demonstrated by Sygaldry, it is another very strong reason why Arduino compatibility must be addressed in future work on the library.

Ultimately, it's likely that many Arduino implementations will never update to more modern compilers supporting C++20. Future work will thus be forced address whether to support these platforms, and if so, how. Since many of the key advantages of the approach demonstrated in Sygaldry rely on reflection functionalities not available in earier C++ standards, it is most likely that only well-maintained Arduino implementations updated to use newer compilers will be supported by the library. The only alternative would be to adopt an approach to reflection not based in C++20. For example, component metadata could be declared in some other structured format from which glue code and boilerplate could be generated. There are, however, significant advantages in terms of type safety, parsimony, and performance, that come from direct representation of

components in reflectable C++. Regardless, Arduino compatibility must be addressed eventually in order to allow Sygaldry to make use of the widest variety of platforms, as well as the extensive ecosystem of reusable software components that already exist targeting Arduino.

These issues highlight that hardware portability remains a challenging issue. Although Sygaldry provides advantages for portability through its runtime system, throughpoints, and binding components, all enabled by its use of reflection, these strategies do not trivially or obviously lend themselves to the development of portable functional components that must directly interact with hardware. Besides improving support for existing hardware abstraction platforms, and with great caution against accidentally inventing a competing standard for hardware abstraction, future work may also consider whether reflection can be used to implement portable hardare-dependent software components more efficiently than existing approaches, such as hardware abstraction APIs like Arduino.

#### Other Compatibility Concerns

Another similar limitation of Sygaldry comes from its extensive use of CMake for build automation. Although, thanks to the careful partitioning of the library, it is very likely that it should be trivial to build Sygaldry in most cases even if CMake is not available, this has not been attempted at the time of writing since CMake is available on all the platforms currently targeted by the library. It may be that reliance on CMake could pose a challenge when eventually porting/extending the library to support a platform on which CMake is not available or is not centrally involved in build automation, such as when using the Arduino IDE.

Finally, as Sygaldry has so far been developed exclusively on Linux and MacOS machines, and relies on Nix to set up the development environment, Windows developers face large barriers to adoption when considering the use of Sygaldry. It remains as future work to support this operating system for development.

### 5.4.3 Maintainability Depending on Quantity of Types of Endpoints

One of the main advantages of Sygaldry is that its use of reflection and treatment of bindings as components with generic throughpoints eliminates glue code for users of the library. Since glue code, if not automated, grows quadratically depending on the number of bindings and components, it poses a major maintenance burden for a library of cross-platform components.

Glue code is completely eliminated for firmware developers using Sygaldry (as long as all required bindings have already been implemented) since all glue is consolidated in the library, and the quadratic glue problem is resolved within the library by use of generic binding implementations. However, there is still a problem of quadratic code that must be carefully approached in future work. Most bindings have a similar structure that essentially resembles the following pseudocode:

```
For each component:
    For each endpoint:
        If the endpoint is a float, do one thing
        If the endpoint is an int, do another
        If the endpoint is an array, do another
        And so on for each different type of endpoint supported in the binding
```

The innermost loop of each binding consists of a similar sequence of branches, one branch for each different type of endpoint. Thus there remains a quadratic code problem, since such an innermost loop must be written for each binding and for each type of endpoint. Given $N$ bindings and $M$ kinds of endpoints, code grows as $N * M$. Although this is a strict improvement over quadratic glue with components, since the number of types of endpoints should be expected to be much smaller than the number of types of components, it seems likely to cause maintenance issues if not handled carefully. Future work must find strategies to reduce this quadratic code problem as much as possible. This may involve consolidating similar patterns of binding implementations in a shared reusable bindings helper library, finding ways to automate this code away as was done with binding-component glue, or merely limiting the quantity of types of endpoints to a manageable

level.

### 5.4.4 Dataflow Based on Types

Sygaldry's basic dataflow mechanism leverages the type system and compile-time reflection to enable the runtime system to pass data from one component to another, enabling the realisation of throughpoints in a declarative manner and eliminating boilerplate code for users of the library. This provides a strong proof of concept, but the implementation has limitations. Fundamentally, the system works based on the assumption that a component or one of its endpoints can be recognised unambiguously and uniquely addressed by its type signature alone. This assumption has proved valid in all uses of the library so far (chapter 6), but this is purely a lucky fluke. Because endpoints are annotated with metadata using template arguments, they tend to have a unique type signature owing to their unique metadata. However, this is a flimsy means of unique identification, and breaks down immediately if there are two instances of the same component within one instrument, such as two buttons. Similarly, components cannot be uniquely identified by their type if there are multiple instances of the same component. As such, the whole dataflow system requires significant rethinking in order to achieve a robust implementation that can handle multiple instances of components. Additional reflection features expected in C++26 should enable a more robust implementation. In the meantime, it may be necessary to make fundamental changes to the library to better support the unique identification of components and endpoints. Until such changes are realised, the dataflow features of Sygaldry are only usable when the data to be passed around has a unique type signature.

Furthermore, although convenient enough for situations where there is just a bit of dataflow between components, as seen in the instrument firmwares implemented to date, the current approach to dataflow in Sygaldry is likely not adequately convenient for more involved dataflow graphs. As future work, it would be useful to devise a more flexible and ergonomic approach to defining dataflow between Sygaldry components. This could, for example, take the form of more systems for composing Sygaldry dataflows, such as using generic programming to implement the

Faust dataflow operators, or perhaps through the development of a graphical environment allowing the composition of dataflow networks and resulting in the generation of Sygaldry C++ code.

### 5.4.5 Library Scope

The relatively small pool of functional components means that, at the time of writing, most DMI designs cannot be implemented without also implementing new components. This makes the cost of adopting Sygaldry relatively high compared to existing ecosystems, such as Arduino, where it is more likely that a 3rd-party library already exists for any given functionality. Those who are willing to accept the additional development effort of porting existing libraries or writing new functional component implementations will in so doing contribute to the development of a broader pool of functional components. In fact, in most cases the additional effort related to using Sygaldry should be relatively small, mainly limited to adapting to Sygaldry's component-oriented dataflow model, since implementing a new component largely consists of writing the same boilerplate and other code that would otherwise be required in the same design implemented using another approach. As reward, subsequent users will be able to trivially make use of the newly implemented functionality in as few as two lines of code (an include directive and an invocation of the component). Nevertheless, future work should actively extend the pool of functional components, as this lowers barriers to adoption and increases the overall utility of the library.

Similarly, future work should extend Sygaldry to support the most common platforms and languages used in the implementation of DMIs. As the ultimate goal of developing such a library is to enable the reuse of code and design knowledge, the more environments in which the code (and associated design knowledge) can be leveraged, the better. This should ideally include the most common embedded systems, interpreted languages, and digital audio workstations. Extending Sygaldry to enable as many of its components as possible to be used in the most common graphical environments, notably Pure Data and Max/MSP, would significantly broaden the impact and usefulness of the library, by making its highly-validated DMI-specific library of portable software components available to many more users, especially those that might be challenged by having to

compile C++ code to otherwise use the library. Although many components in the library are hardware specific, and would not be able to be deployed in this kind of environment, as the scope of the library grows to include more sound synthesis and processing, mapping, and modelling components, many of these will eventually be useful to have available in environments besides embedded firmware.

One major limitation of Sygaldry's design, particularly the runtime class system, is the imposition of a strong operational model consisting of a one-time setup phase followed by an indefinite loop. Although the runtime system is optional, and can easily be ignored by a Sygaldry user, its use provides significant improvements to the composability of the components of the library, so we are wary of the way it may bias designs implemented in the library. The setup-loop operational model is also found in Arduino, where it hasn't been shown to be a significant issue. However, we recognize that this model predisposes users towards certain design decisions that may not always serve the intended application, and that it may make it more difficult or unfamiliar to adopt other structures for program execution, such as interrupt service routines and other forms of concurrency. Although Sygaldry functional components are developed using a dataflow mental model, the library's actual ability to capitalise on this is still limited, partially due to the setup-loop execution model (this is discussed more above in section 5.4.4). Future work should examine how other models for runtime execution can be supported, in addition or alternatively to the current setup-loop system. This should enable, for example, the inclusion of audio processing, interrupt handlers, and multi-thread concurrency within Sygaldry programs.

### 5.4.6 Evaluation

Because reusable components can be used across multiple DMI implementations, validation, testing, and evaluation of components provides information and design improvements that are equally as reusable. To date, relatively little work to this effect has been conducted (section 6.4). Future work should include evaluation of different versions of similar components, such as different MIMU sensor fusion algorithms, or different gesture models. It is expected that such efforts, when applied

to a library of reusable components, should have generalisable benefit for all instruments using the components under study.

### 5.4.7 Long-term Benefits

Many of the expected benefits of the design and use of the library are beyond the scope of the present dissertation to examine. In particular, the extent to which the reusability and portability of the component-oriented architecture will influence the maintainability and replicability of instruments developed using the library will take time to become fully apparent. It is expected that these important qualities will be improved by use of a component-oriented library.

## 5.5 Summary

Sygaldry is implemented as a collection of interdependent software components. Most of these provide a functional component of a DMI, and can be assembled in instruments using a highly declarative implementation. Leveraging C++20 reflection techniques, the endpoints of these components can be inspected at compile time, enabling the generation of generic bindings, runtime systems, and simple dataflow networks with minimum runtime cost.

Recall the requirements for a DMI component library from section 4.7:

- Modern C++: the library should be implemented in a compiled language suitable for use in embedded systems as well as for interoperating with the most common interpreted languages used in DMI development. C++ is a natural choice due to its ubiquity, portability, interoperability, and prior status as the defacto standard for real-time multimedia applications.
- Automate Glue: the library should automate the generation of glue code, so that its components can be used in all of the environments in which DMIs are commonly implemented.
- Wide Contract: the library should adapt to the structure of components rather than imposing a particular structure, so that all of the components of a DMI may be included in the library.

By extending prior work [80], Sygaldry demonstrates how the C++20 reflection functionalities and design patterns used can gainfully implement a wide API contract that can adapt to functional components addressing all of the kinds of components found in a DMI, including sensors, actuators, mappings, and audio processors, across embedded and hosted environments. The use of reflection *completely eliminates manual glue code*. Further, the novel Sygaldry runtime system also *eliminates the vast majority of boilerplate*, enabling very succinct declarative implementation of DMIs. The runtime system also enables the implementation of generic components with inputs and outputs that merely reflect the data of other components, i.e. throughpoints. The immediate practical ramifications of these design features are explored in the next chapter.

We considered limitations and future work facing Sygaldry. The main limitations are the difficulty of extending the functionality of a component without physically modifying it, the difficulty of compatibility with prior work due to the ongoing adoption of C++20, the poor maintainability of binding code with respect to the number of kinds of endpoints, weaknesses in the dataflow mechanism owing to overly strong assumptions made about the uniqueness of endpoint types, and the high barrier to entry imposed by C++. Future work should address these limitations, as well as increasing the range of components, platforms, and operational models supported by the library, and continuing to validate and evaluate components within the library. Finally, it remains to be seen the extent to which the expected benefits of the present work will be realised. The long term benefits to maintainability, replicability, and the development and transmission of design knowledge and capability remain untested due to limitations on the duration of doctoral research.

# Chapter 6

# Applications of Sygaldry

This chapter highlights some applications of Sygaldry that serve as preliminary validation that the library is able to achieve some of its stated aims. These include replications of the T-Stick and mubone controllers, a port of the library between two dissimilar hardware platforms, and a demonstration of the generalisability of validation testing facing components of the library.

Olsen [112] argues that the fundamental question, considering the evaluation of a complex toolkit, is whether or not important progress has been made. We will compare applications of Sygaldry with prior work achieving the same aims. Our purpose is to demonstrate that the design of the library does in practice address these applications (replication, hardware porting, and validation testing), and that the library provides a high level of expressive leverage and power in combination [112] compared to conventional approaches, leading to solutions that are considerably more succinct, with much higher incidence of code reuse. This is made possible by Sygaldry's runtime system, which makes it trivial to combine DMI components into an executable firmware, as will be seen.

These preliminary results are subject to notable limitations. Future work, especially longitudinal evaluation of the library, user-facing studies, and runtime performance evaluation, will be required to begin to fully characterise the repercussions of Sygaldry's design and use. Nevertheless, the applications described below provide important evidence of immediate practical benefits

provided by the library, demonstrating some merits of the library's design and offering empirical motivation for future work.

## 6.1 Replicability

Replicating an existing instrument using Sygaldry serves to demonstrate that the library is capable of implementing DMI firmware. By comparing the effort involved in the replication with that of the original, we can make observations about the relative strengths and weaknesses of using Sygaldry compared to previous implementation strategies.

### 6.1.1 Replicating the T-Stick

In order to provide a clear target for a minimum viable DMI component library, one of the first applications of Sygaldry was to replicate the T-Stick. This provided an initial pool of requirements for functional and binding components.

The development history of the T-Stick is discussed in more detail in section 3.1. The T-Stick is a controller initially designed by Joe Malloch and collaborators in the 2000s [49], [50] and that has received ongoing maintenance since (e.g. [56]). The device consists of a length of ABS pipe upon which are affixed several sensors. Recent iterations of the controller include a linear array of capacitive touch sensing strips, a piezo-resistive pressure sensor strip, a button in one endcap, and a MIMU. Sensor data are acquired and processed on the microcontroller, which must transmit the data over WiFi in OSC protocol format using UDP.

The substantive code implementing the T-Stick firmware in Sygaldry is as follows. Each line of the struct is merely a declaration of one of the functional components of the instrument—a button, an analog sensor (the FSR), the capacitive touch sensor, a fuel gauge for monitoring battery state, the MIMU, and sensor fusion for the latter. The template arguments to the MIMU give an implementation of the serial interface for reading from the sensor. The template argument to the sensor fusion algorithm allows the sensor fusion to accept the MIMU's data as an argument to its main subroutine (implementing a throughpoint) without having a physical dependency on the

implementation of the MIMU. The final two lines pass the instrument declaration to the runtime specialization for the ESP32 hardware platform and then invoke the generated main loop.

```
struct TStick
{
    sygse::Button<GPIO_NUM_15> button;
    sygse::OneshotAdc<syghe::ADC1_CHANNEL_5> adc;
    sygsa::TrillCraft touch;
    sygsa::MAX17055 fuelgauge;
    sygsp::ICM20948< sygsa::TwoWireByteSerif<sygsp::ICM20948_I2C_ADDRESS_1>
                   , sygsa::TwoWireByteSerif<sygsp::AK09916_I2C_ADDRESS>
                   > mimu;
    sygsp::ComplementaryMimuFusion<decltype(mimu)> mimu_fusion;
};

sygbe::ESP32Instrument<TStick> tstick{};
extern "C" void app_main(void) { tstick.app_main(); }
```

The implementation consists of a high-level declaration of the components of the device, which is then passed to a hardware platform wrapper (shown in section 5.2.10) that generates most of the boilerplate needed to set up and activate the functional components in a loop (section 5.2.8). Other than this succinct high-level declaration of the T-Stick's sensors, the entire implementation consists of reused library code. This is a significant departure from previous implementations, which are entirely T-Stick-specific code. For comparison, the substantive implementation of the T-Stick in Sygaldry uses about 15 lines of code that are specific to the T-Stick (seen above), whereas the main T-Stick firmware is 1052 lines of code.

Getting the library to the point where the T-Stick firmware was able to be implemented in this way constituted a significant amount of development, as the whole design of the library, all

of the involved components including the platform and build tools, and a hardware replication of the T-Stick were all necessary. However, according to the purpose of the library, all of this work should be reusable.

The mubone is an augmented trombone developed by myself and Kalun Leung, described in more detail in section 3.2. It is made with two control devices: an orientation sensor consisting of just a MIMU, and a handheld controller that in most stable form merely consists of several buttons. These controllers collect the sensor data and forward it to a personal computer for further interpretation, mapping, and control. Since all of the functional components of a mubone controller are also found in the T-Stick, once the effort of developing the latter was complete it was trivial to replicate the mubone in Sygaldry. Here is the implementation of the orientation sensor:

```
struct Orientor
{
    sygse::Button<GPIO_NUM_15> button;
    sygsp::ICM20948< sygsa::TwoWireByteSerif<sygsp::ICM20948_I2C_ADDRESS_1>
                   , sygsa::TwoWireByteSerif<sygsp::AK09916_I2C_ADDRESS>
                   > mimu;
    sygsp::ComplementaryMimuFusion<decltype(mimu)> mimu_fusion;
};

sygbe::ESP32Instrument<Orientor> orientor{};
extern "C" void app_main(void) { orientor.app_main(); }
```

Similar to the T-Stick, the implementation of the mubone controller consists of a high-level declaration of its components. The mubone-specific aspects of the implementation are succinct and limited. The majority of the implementation draws on library code. The implementation is about 11 lines of code, compared to 816 and 336 lines of code in the two prior implementations.

As the replication of the T-Stick and mubone in Sygaldry can be compared directly with prior implementations, we can immediately observe some of the practical effects of the library. The level of reuse exhibited here is *drastic and immediate*. Previous implementations of the T-Stick and mubone made use of shared Arduino and ESP-IDF libraries, but none of the implementation of these instruments specifically related to their function as DMIs was shared, in both cases consisting of hundreds of lines of code. In the implementations using Sygaldry, the vast majority of the implementation is shared. Each instrument has about a dozen lines of boilerplate that are unique to that instrument, and the rest of the implementation is consolidated in the shared library. The improved efficiency results from the emphasis of the library; whereas Arduino is not geared towards any particular application, Sygaldry is designed specifically for developing DMIs.

Furthermore, as a result of the application of modern C++ reflection features, Sygaldry provides a particularly beneficial improvement by avoiding the need for significant amounts of glue code and boilerplate. The problematic glue code required in previous T-Stick implementations involved over 300 lines of code of the following form, spread out all over the main implementation file:

```
// declaration of the signal value at global scope
T signal;
// declaration of the libmapper signal and metadata at global scope
mpr_sig lmsignal = 0;
float sigMax = 1.0f;
float sigMin = 0.0f;
// instantiation of the libmapper signal in the setup function
lmsignal = mpr_sig_new( dev, MPR_DIR_OUT, "sig/address", 1, MPR_FLT, "un"
                      , &sigMin, &sigMax, 0,0,0);
// reading the signal value
signal = get_signal();
// sending to the libmapper network
```

```
mpr_sig_set_value(lmsignal, 0, 1, MPR_FLT, &signal);
// sending over open sound control to the first destination address
oscNamespace.replace( oscNamespace.begin()+baseNamespace.size(),oscNamespace.end()
                    , "sig/address");
lo_send(osc1, oscNamespace.c_str(), "f", signal);
// sending over open sound control to the second destination address
oscNamespace.replace( oscNamespace.begin()+baseNamespace.size(),oscNamespace.end()
                    , "sig/address");
lo_send(osc2, oscNamespace.c_str(), "f", signal);
```

Or the similar glue found in the original mubone implementation, which simply enumerates all of the exposed OSC endpoints manually one after the other using the "dispatcher" class:

```
// Outputs
#ifdef HARDWARE_COUNTERWEIGHT
dispatcher.addOutputSignal("/raw", reading.data, 10);
dispatcher.addOutputSignal("/accl", reading.accl.data(), 3);
dispatcher.addOutputSignal("/gyro", reading.gyro.data(), 3);
dispatcher.addOutputSignal("/magn", reading.magn.data(), 3);
dispatcher.addOutputSignal("/quat", quat.coeffs().data(), 4);
dispatcher.addOutputSignal("/motion", motion.data(), 3);
dispatcher.addOutputSignal("/period", &period, 1);
#else // assume HARDWARE_YOKE
dispatcher.addOutputSignal("/raw", reading.data, 10);
dispatcher.addOutputSignal("/yoke_quat", quat.coeffs().data(), 4);
dispatcher.addOutputSignal("/buttons", &buttonmask, 1);
#endif
```

```
#ifdef HARDWARE_HASDISTANCESENSOR
dispatcher.addOutputSignal("/slide", &distance, 1);
#endif

// Inputs
dispatcher.addTrigger("/state/initialize", &state, initialize);
dispatcher.addTrigger("/state/align", &state, align);
dispatcher.addTrigger("/state/normal", &state, normal);
dispatcher.addTrigger("/state/calibrate", &state, calibrate);
dispatcher.addTrigger("/reset_flash", &state, initialize, reset_flash);
dispatcher.addInputSignal("/calibration/accl/matrix", calibr.cc.acclcalibration.data(), 9, u
dispatcher.addInputSignal("/calibration/gyro/matrix", calibr.cc.gyrocalibration.data(), 9, u
dispatcher.addInputSignal("/calibration/magn/matrix", calibr.cc.magncalibration.data(), 9, u
dispatcher.addInputSignal("/calibration/accl/vector", calibr.cc.abias.data(), 3, update_and_
dispatcher.addInputSignal("/calibration/gyro/vector", calibr.cc.gbias.data(), 3, update_and_
dispatcher.addInputSignal("/calibration/magn/vector", calibr.cc.mbias.data(), 3, update_and_
#ifdef HARDWARE_HASDISTANCESENSOR
dispatcher.addInputSignal("/calibration/slide/first_position", &(distnc.cc.firstposition), 1
dispatcher.addInputSignal("/calibration/slide/third_position", &(distnc.cc.thirdposition), 1
#endif
dispatcher.addInputSignal("/filter_coefficients/k_P", &(filter.fc.k_P), 1, store_flash);
dispatcher.addInputSignal("/filter_coefficients/k_I", &(filter.fc.k_I), 1, store_flash);
dispatcher.addInputSignal("/filter_coefficients/k_a", &(filter.fc.k_a), 1, store_flash);
dispatcher.addInputSignal("/filter_coefficients/k_m", &(filter.fc.k_m), 1, store_flash);
```

This kind of code is completely eliminated in Sygaldry, being automated by bindings that reflect over components' endpoints at compile time.

Similarly, both the T-Stick and mubone firmwares include significant amounts of boilerplate,

where subroutines of components are manually invoked and data shuffled around so that it can be conveniently exposed to the network.

```
// example T-Stick boilerplate ***************************

mpr_dev_poll(lm_dev, 0);
button.readButton();
fsr.readFsr();
// Read Touch
#ifdef touch_TRILL
    touch.readTouch();
    touch.cookData();
    gestures.updateTouchArray(touch.touch,touch.touchSize);
#endif
#ifdef touch_CAPSENSE
    capsense.readCapsense();
    gestures.updateTouchArray(capsense.data,capsense.touchStripsSize);
#endif

// read battery
if (millis() - battery.interval > battery.timer) {
  battery.timer = millis();
  readBattery();
  batteryFilter();
}

// read IMU and update puara-gestures
    if (imu.accelAvailable()) {
```

```
    imu.readAccel();
    // In g's
    gestures.setAccelerometerValues(imu.calcAccel(imu.ax),
                                    imu.calcAccel(imu.ay),
                                    imu.calcAccel(imu.az));
}
if (imu.gyroAvailable()) {
    imu.readGyro();
    // In degrees/sec
    gestures.setGyroscopeValues(imu.calcGyro(imu.gx),
                                imu.calcGyro(imu.gy),
                                imu.calcGyro(imu.gz));
}
if (imu.magAvailable()) {
    imu.readMag();
    // In Gauss
    gestures.setMagnetometerValues(imu.calcMag(imu.mx),
                                   imu.calcMag(imu.my),
                                   imu.calcMag(imu.mz));
}


gestures.updateInertialGestures();
gestures.updateTrigButton(button.getButton());


// go to deep sleep if double press button
if (gestures.getButtonDTap()){
    std::cout << "\nEntering deep sleep.\n\nGoodbye!\n" << std::endl;
```

```
    delay(1000);
    esp_deep_sleep_start();
}



// example mubone boilerplate ****************************
indicator.toggle();

if (mimu.readInto(reading))
    calibr.calibrate(reading);
quat = filter.fuse(reading.gyro, reading.accl, reading.magn);
motion = filter.getZeroGravityAccl();
period = filter.getPeriod();

#ifdef HARDWARE_YOKE
buttns.read();
buttonmask = buttns.getButtonMask();
#endif

#ifdef HARDWARE_HASDISTANCESENSOR
distance = distnc.getDistance();
#endif
reading.updateBuffer();
dispatcher.dispatch();
```

All code of this kind is also eliminated in Sygaldry. Instead of having to manually invoke subcomponents in a hand-written main loop, this mechanistic work is performed automatically based on compile-time reflection over the components in the instrument. This elimination of

boilerplate is a unique feature of Sygaldry, not found in prior work, and is a consequence of Sygaldry's compile-time-generated runtime system (section 5.2.8).

These results suggest the long-term potential of the library. As the pool of components grows to encompass more functionality and software environments, it will become increasingly trivial to implement more and more possible designs in the library. In the case of the implementation of the mubone, the entire implementation was reduced to trivial reuse, resulting in a reduction of new lines of code by an order of magnitude compared to the previous implementation. In cases where new components may need to be implemented, users of the library may still expect benefits, thanks to the elimination of glue code and the significant reduction of boilerplate enabled by the binding and runtime systems of the library.

## 6.2 Hardware Porting

Over the lifetime of a well-maintained instrument, it is eventually always necessary to port the device to new hardware. For example, over the lifetime of the T-Stick, the instrument has been implemented using Arduino Pro Micro, ESP8266, ESP32, and ESP32s3. In the main T-Stick firmware development tree, all hardware ports except the one from ESP32 to ESP32s3 have triggered ground up rewrites of the firmware. Similarly, the mubone handheld controller has been implemented using ESP8266 and Teensy 3.2. As with the T-Stick, the port to new hardware triggered a ground up rewrite of the firmware.

In order to test the portability and reusability of the Sygaldry components used to replicate the T-Stick, a second replication was undertaken targeting a new hardware platform: RP2040. This was a minimum port, missing certain functionality described below. The final implementation is very reminescent of the one above:

```
struct TStick {
    sygsa::TwoWire<0,1,400000> i2c;
    sygsr::Button<26> button;
```

```
    sygsr::OneshotAdc<ADC_CHANNEL_1> adc;
    sygsa::TrillCraft trill;
    sygsp::ICM20948< sygsa::TwoWireByteSerif<sygsp::ICM20948_I2C_ADDRESS_1>
                   , sygsa::TwoWireByteSerif<sygsp::AK09916_I2C_ADDRESS>
                   > mimu;
    sygsp::ComplementaryMimuFusion<decltype(mimu)> mimu_fusion;
};

sygaldry::sygbr::PicoSDKInstrument<TStick> runtime{};
int main(){runtime.main();}
```

Many of the top-level components of the RP2040 version of the firmware are the exact same as those used in the ESP32 version. The main differences are the I2C component, the ADC component, and the button component. In the ESP32 version of the firmware, the I2C component is implicit in the ESP32 runtime class, whereas in the RP2040 version the I2C component is stated explicitly. This was a design choice motivated by the fact that I2C on the RP2040 can make use of a great variety of possible hardware pins. In this context, making the I2C component top-level is appropriate since the choice of pins may vary with each hardware implementation. The button component is necessarily hardware dependent, since reading a button requires interacting with GPIO, for which every hardware platform has its own interface. However, both button component implementations derive from a common portable button gesture model component that implements some feature analysis of the button data, seen in section 4.5. Similarly for the ADC.

The greater part of the effort required to port the T-Stick to this new hardware consisted of implementing platform abstraction, binding, and build tool components. About 7 substantive lines of code / 87 lines of literate code were required to port the USB serial CLI binding component, about 6 substantive lines of code / 90 lines of literate code for the button, about 4 substantive lines of code / 156 lines of literate code for the ADC, and most substantially 194 lines of literate code (most of it substantive) for the Arduino platform abstraction.

For a more feature complete replication, it would be necessary to at least implement WiFi, OSC, and persistent storage on the RP2040 platform. However, as the purpose of the port was to stress test the portability of Sygaldry components rather than to create a feature complete replication of the T-Stick, these features were deliberately omitted in the interest of saving time. Even if all of the skipped over components had to be fully reimplemented for the RP2040 platform, it is not expected that this would detract meaningfully from the result.

Considering all software components transitively involved in their implementations, the ESP32 and RP2040 T-Stick implementations share 29 software components. The ESP32 implementation additionally requires 7 platform-dependent components. The RP2040 implementation requires 5 platform-dependent software components. Therefore, the ESP32 implementation involves 81% portable reusable components, and the RP2040 implementation involves 85%. This is a drastically greater level of software reuse than has been achieved in any previous port of the T-Stick from one microcontroller platform to another. As discussed in section 3.1, during the port from Arduino Pro Micro to ESP8266, 1/5 implementation files from the Pro Micro firmware were used in the ESP8266 firmware, for which 6 new implementation files were written. Heuristically, this consitutes 14% portable component use. Although the comparison here is somewhat dubious due to the very different quantity of implementation components and the very broad differences in the implementation architecture, it is still strongly suggestive of a much higher incidence of code reuse in the port using Sygaldry.

As well as an incomplete port to RP2040, a complete port of the ESP32 T-Stick firmware to ESP32s3 was also conducted. In this case, the excellent upstream portability design of the ESP-IDF meant that all of the platform-specific components of the ESP32 firmware were able to be used directly with very little modification. The mainstream T-Stick firmware also benefitted from the ESP-IDF's strong portability, maintaining most of its code in the ports from ESP8266 to ESP32 and ESP32 to ESP32s3. In the long-term, as hardware manufacturers improve the portability of their libraries, and as hardware abstraction platforms like Arduino eventually catch up to more modern C++ standards, this level of portability should become more common. It will, however,

remain beneficial to be able to port parts of the library to support different software environments (e.g. ESP-IDF to Raspberry Pi Pico SDK), which the design of Sygaldry facilitates through its strict and explicit management of platform-specific dependencies.

## 6.3 Rapid Prototyping

In this section, implementation of a novel unnamed wind controller is described. This work serves to explore the practical benefits of the library in the context of rapid prototyping.

### 6.3.1 Hardware Design of the Wind Controller

The controller consists of an air pressure sensor, twelve capacitive touch pads, and an orientation sensor that communicate their sensor data via WiFi to a nearby laptop which implements mapping and sound synthesis. The MP3V5050GC6U pressure sensor from NPX Inc., Trill Craft capacitive touch sensor from Bela, and ICM20948 MIMU from Invensense respectively are chosen as sensors, and the TinyS3 ESP32s3 development board from Unexpected Maker is used for microprocessor. The pressure sensor outputs an analog voltage that is easily read using the ADC of the microcontroller via the existing Sygaldry component for that purpose. The other two sensors communicate over I2C, and were chosen since Sygaldry components were already available for these sensors following the replication of the T-Stick and Mubone. The touch pads are arranged according to the novel pitch fingering system described in my 2022 NIME paper on the subject [3]. The mouthpiece was modelled in OpenSCAD and 3D printed in PLA. The PCB, which also serves as the main body of the instrument, was designed in KiCAD and manufactured by a prototyping-oriented consumer-facing PCB fabrication company. The final electronics assembly was done by hand, including point-to-point soldering of the connections between the sensors on the PCB and the microcontroller and motion sensor on a seperate perforated prototyping PCB; the use of the latter, in combination with a custom fabricated PCB, was chosen to facilitate rapid iteration on the electronics design.

The firmware implementation for the wind controller is essentially the same as the T-Stick.

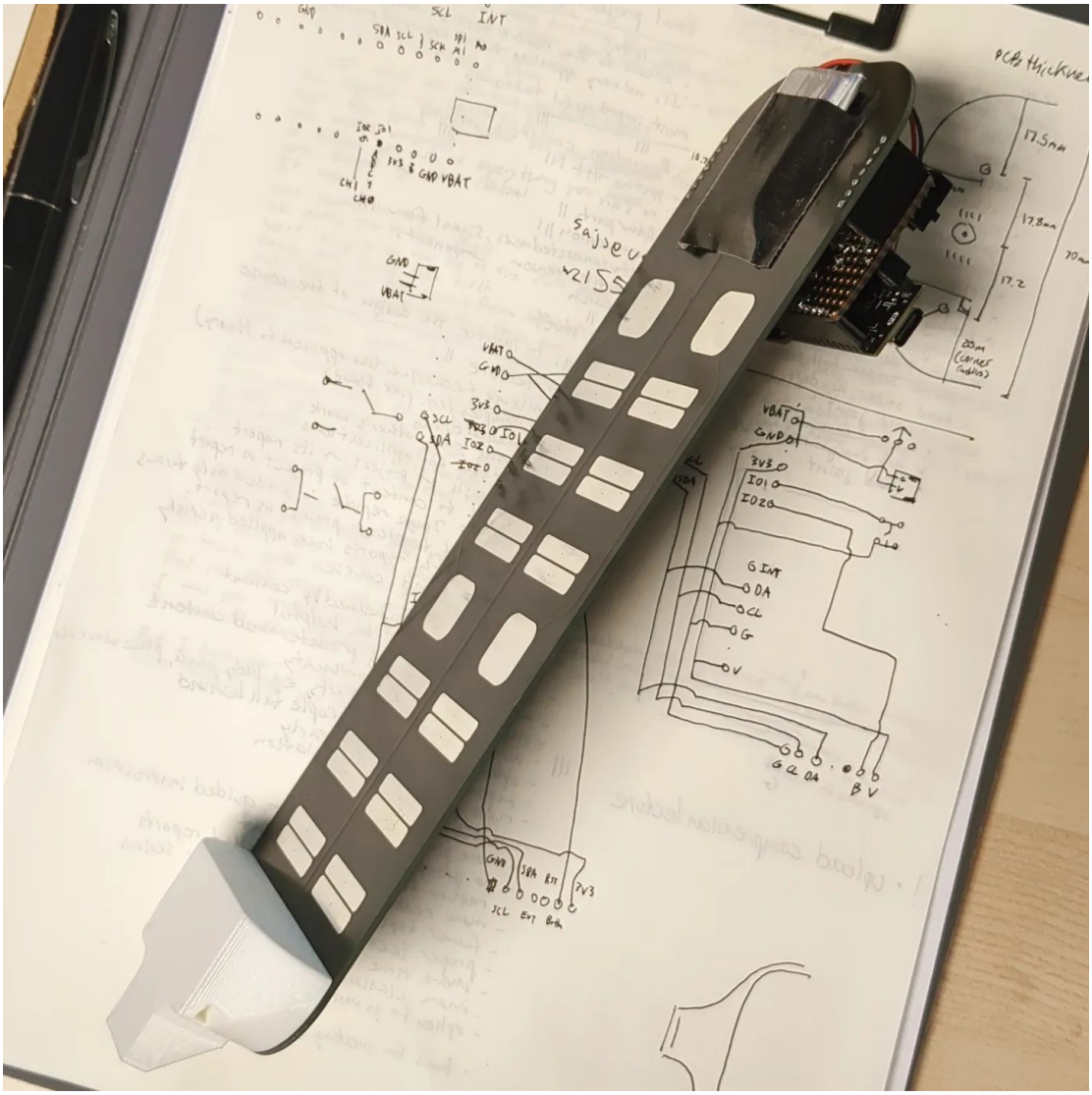


**Fig. 6.1** A novel unnamed wind controller.

The hardware design is entirely different, providing for a completely different interface with unrelated gestural and interactive affordances, but the sensors are all the same, and so the software components can be reused unmodified. Firmware implementation for the new interface thus took a matter of minutes.

Although the library doesn't provide any benefit to hardware design and development at this time, this small and tentative initial exploration of its application to rapid prototyping suggests potential. The exploration is admittedly very limited, and should be seen as a best-case scenario; the library user has intimate familiarity with the library (as its sole architect-implementer), already has the development toolchain installed, and implements an instrument that just so happens to have all its components already provided by the library. Under such good conditions, the results are not surprising, and it should not be assumed that they would generalise. A more ecologically valid study would likely reveal usability issues and missing functionality that were not encountered in this preliminary exploration.

Nevertheless, by implementing a novel interface using the library, we see that in a case where the functionality needed for the design is already available in the library, and when usability is ruled out as an issue, requirements that seem reasonable to hope may be met in the long term should work continue on the library, use of Sygaldry reduces firmware development to a very trivial listing of components needed by the design, once again facilitated by the runtime system enabling such a high-level declarative implementation. This is a suggestive proof of concept for the long term possibilities for such a library, though it is obviously too soon to make strong claims.

## 6.4 Component Validation

Sygaldry components each provide a small and specific functionality to the library. Because the purpose of each component is relatively narrow, it is relatively straightforward to determine useful measures by which to assess the correctness and performance of these components, especially compared to a complete musical instrument, which is very difficult to evaluate in any meaningful and general way.

Unit testing is a practice in software development that helps to prevent accidentally introducing runtime functional errors into code by ensuring empirically that the software passes the requirements imposed by unit tests. The practice is extremely common in industry, but essentially never observed in the development of DMI firmware. Although it is beyond the scope of this research to assess why unit testing is not more common in academic DMI development, I speculate that there are three main factors. Firstly, unit testing is boring, unpleasant, time consuming, sometimes difficult, and adds no functionality. Under tight academic deadlines, and in an environment where the only results of activity that are explicitly and directly incentivized are research results, it is natural that DMI developers in academia are not inclined to rigorously unit test their work. Secondly, given the experimental nature of development in academia, there is always a significant possibility that any code you are writing now will no longer be relevant in the near future; under these uncertain conditions, effort made to improve the long-term maintainability of software is potentially effort wasted, which doesn't favor the expenditure of this effort. Lastly, it is especially difficult to unit test code targeting embedded systems and interacting with hardware. Since the correct functionality of this code can only be observed when certain external actions in the physical world are enacted, it is very tricky to develop automated tests for this code.

Out of 56 Sygaldry software components, 21 include unit tests. While certainly incomplete, this is a significant improvement over e.g. the main-line T-Stick firmware, which currently has no unit tests at all. Tests are currently concentrated on the portable software components in the library, with the concepts and endpoints packages being especially well tested. This is because it is more difficult to test hardware-specific software components, since it requires specific hardware to execute this code, and might even require manipulation of physical hardware to properly assess the correctness of some components. It's possible some components, for this reason, may never be fully unit tested. Nevertheless, testing of portable and hardware-abstract components can still improve the overall robustness of the library.

In principle, testing should improve the long-term maintanability of the Sygaldry T-Stick firmware and other instruments developed using the library. The main point, however, is recogniz-

ing that the component architecture of the library facilitates and encourages this kind of testing, especially for portable components that can be executed and tested on a typical developer's laptop. The high level of portability and reusability of components favors their long term use, which provides stronger incentive to conduct this kind of validation testing. The agglomeration of this functionality in a library encourages collaboration towards this work.

## 6.5 Summary

Recall the maintenance challenges described in chapter 3:

1. burdensome glue code
2. poor hardware code portability
3. poor robustness against failure
4. instability and incompatibility of behavior and programming interfaces between implementations

Sygaldry's high-level declarative component-oriented architecture permits rapid development and replication of DMI firmware through its runtime system, which eliminates the majority of boilerplate, and its bindings, which eliminate manual glue code. The explicit delineation of dependencies facilitated by the component-oriented design improves the portability of the components, and thus the DMIs developed using the library. The components are more easily validated and evaluated than whole instruments, and results of these efforts benefit all designs using the validated components, providing better robustness against software errors.

The validation efforts presented in this chapter are early results, and there remains significant future work to do. The usability of the library for new users is untested, so there is most probably significant room for improvement, and the extent to which the greater reuse facilitated by the library may lead to improvements in long-term maintenance, stability, and compatibility for DMIs implemented using the library remains to be demonstrated. Despite these limitations, it is clear

that the approach implemented in Sygaldry can provide immediate benefits to DMI development and replication, and it is reasonable to claim that the long term effects of developing and using such a library are promising. Further investigation is warranted to explore the long-term benefits that may come from this approach.

# Chapter 7

# Conclusion

## 7.1 Summary of contributions

We are interested in making DMIs that can be maintained over a long period of time, to foster and sustain long-term engagement with these instruments, and enable long-term research and improvement of our design knowledge and capabilities. With these goals in mind, we have discussed a conceptual framework of DMIs as networks of components with endpoints and throughpoints connected through models and cross-domain mappings. Leveraging this framework, we explored the challenges that have arisen in the maintenance of the T-Stick and the mubone. In light of these challenges, we reviewed the hardware platforms, functional components, and existing libraries of reusable components commonly employed in DMI development within the research community. The prior work reviewed exhibited several main limitations: most existing libraries are not suited for cross-platform use due to the issues of glue code and boilerplate; they are not suited for implementing all kinds of functional components of DMIs due to issues of API contract and worldview; these issues are compounded by the limitations of interpreted languages and runtime APIs, which impose performance limitations that preclude components implemented using these techniques from being used in certain common (especially resource-constrained embedded) environments. We examined the design and implementation of Sygaldry, which uses modern C++

to automate the generation of glue and boilerplate, providing a wide API contract suitable for all kinds of DMI components, and enabling the maximum of cross-platform portability at a minimum of runtime cost. In practical applications, Sygaldry provides benefits to replication, porting, testing, and rapid prototyping of DMI components, providing a promising direction that may provide the necessary foundation for long-term maintainable development of these instruments.

### 7.1.1 Novel Conceptual Framework

The framework presented in chapter 2 proposes novel terminology that aids in the conception of a feature-complete flexible library of reusable and portable DMI components. We view components as self-similar collections of subcomponents with endpoints between which data flows. We introduce the concept of a throughpoint, which is distinguished from a regular endpoint by its semantic character. Whereas endpoints are semantically strong, usually having a name and a definite interpretation within the network, throughpoints reflect the dominant semantic character of the endpoints they are connected to. Throughpoints are generic and adaptible. Regular endpoints are specific. The notion of throughpoints leads naturally to distinguishing between cross-domain mappings, which are (usually instrument-specific) mappings that bridge one semantically strong set of endpoints to another, and models, which are (usually generic reusable) mappings that derive a new set of semantically strong endpoints from another. We emphasize that functional components with strong endpoints encode the majority of the design knowledge of a DMI dataflow network, and that these components are in principle likely to be portable to different DMIs. In parallel, highly generic components with only non-specific throughpoints, such as generic mapping techniques like preset interpolation, also have good portability. The remaining cross-domain mappings are the main instrumentally-specific components of a design, and are not expected to be generally portable.

### 7.1.2 Maintenance Case Studies

We reviewed the maintenance histories of the T-Stick and the mubone. Despite their significant difference, we found notable overlap in the challenges that have arisen in the maintenance and development of these two instruments. In both instruments, we find burdensome glue code, difficulty with hardware ports, robustness against failure, and long-term compatibility between different implementations.

### 7.1.3 Noting the Opportunity for Reuse

Our conceptual framework implies that the majority of most DMI implementations may consist of portable components, and that a reusable library of DMI components could have a significant positive impact on the way our designs are implemented. Our heuristic review of the platforms and components of DMIs provides further motivation for the development of such a library. There is significant commonality in DMI implementations in the research context, and it seems likely that a relatively small pool of components could provide the majority of the functionality of many instruments found in the literature. There is thus a large opportunity to consolidate effort and reduce the overall burden of maintenance, validation, and development, and improve the replicability of this research.

### 7.1.4 Requirements for a DMI Component Library

Unfortunately, existing libraries towards this benefit are limited in important ways. DMIs are implemented on a large variety of platforms, including both freestanding resource constrained microcontrollers and powerful hosted personal computers, often both within the same instrument. They involve a variety of different kinds of components, spanning sensors, actuators, gesture modelling, mapping, and sound, and may span several physical transports, communication protocols, and programming languages. In contrast with this incredible diversity, prior libraries tend to only work in a limited number of programming languages and hardware environments, and only cover a limited range of the kinds of components involved in making a DMI. These limitations are

attributable to the reliance on runtime APIs, interpreted languages, and narrow APIs, which all contribute to the need for unmanageable glue code and performance restrictions that make it impractical to develop a library that covers all aspects of a DMI implementation and with adequate performance and portability for all of the environments it may span.

### 7.1.5 Sygaldry

These limitations can be overcome using modern C++. Using compile-time reflection, metaprogramming, and generic programming techniques available as of C++20, it is now possible to develop a library with a wide API, best-possible runtime performance, and portability to any environment or language that may be involved in a DMI implementation. These techniques have already been shown by Celerier to be highly applicable to sound synthesis and processing in hosted environments and interpreted languages [80]. Using these techniques, Sygaldry provides proof of concept that they are equally applicable to sensing and gesture modelling, communications protocol binding, and embedded environments. Our implementation using these techniques further extends Celerier's work, demonstrating how the same techniques can be used to eliminate boilerplate code and implement basic dataflow networks in a high-level declarative style, implement throughpoints using generic programming, and implement a message-like semantics. This implementation of the concept of throughpoints also demonstrates how bindings can be viewed and implemented as a kind of generic cross-domain mapping component. This work was found to have immediate benefits when applied to replicating existing instruments, porting them to a new microcontroller platform, validating the correctness of their implementation, and rapid prototyping of a novel wind controller.

## 7.2 Closing Remarks

Research has been ongoing into the design and development of DMIs for over 30 years. Over this period, numerous tools and technologies have emerged as common members of the digital luthier's kit. Despite that DMIs often have more in common than they don't, in terms of their sensors,

actuators, and even their mappings, at this time there is no consensus about what materials, tools, and techniques are "standard" in the making of DMIs, because there has been no discussion or effort towards defining or implementing such a standard toolbox. This is in contrast to the widespread availability of environments for sound synthesis and processing, specifically, that demonstrate the high maturity of digital audio as a discipline. These environments have standards. Unfortunately, the environments most often employed in the implementation of DMIs are not geared specifically towards this application area, and consequently there is nothing remotely approaching consensus around what DMI components are commonly needed and used, outside of the digital audio domain. There is no compelling reason for this to continue. The design and development of DMIs has reached a high level of maturity. It is clear in even a perfunctory review of the literature that certain components are standard. There is no good reason why DMI developers should continue to implement software for dealing with these components, every time from scratch. Now is the time for our digital lutherie workstations to include digital lutherie libraries. By building these libraries, we will consolidate the basic effort of building the standard tools and materials, lower the barrier to developing DMIs, and enable the next generation of research into the design of these instruments to reach as-yet unseen potential.

# Bibliography


[1] T. J. West and K. Leung, "Early prototypes and artistic practice with the mubone," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2022.

[2] T. J. West, S. Huot, and M. M. Wanderley, "Sygaldry: DMI components first and foremost," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2024.

[3] T. J. West, "Pitch fingering systems and the search for perfection," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2022.

[4] M. M. Wanderley, T. J. West, J. Rohs, E. A. L. Meneses, and C. Frisson, "The IDMIL digital audio workbench: An interactive online application for teaching digital audio concepts," in *Proceedings of the International Audio Mostly Conference*, 2021.

[5] L. Turchet, T. J. West, and M. M. Wanderley, "Touching the audience: Musical haptic wearables for augmented and participatory live music performances," *Personal and Ubiquitous Computing*, 2021.

[6] T. J. West, B. Caramiaux, S. Huot, and M. M. Wanderley, "Making mappings: Design criteria for live performance," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2021.

[7] R. Godin. "Growing a global business from a small factory - CIRMMT distinguished lectures in the science and technology of music." (2014), [Online]. Available: `https://www.cirmmt.org/en/events/distinguished-lectures/godin`.

[8] P. Cook. "A (not so) brief history of laptop orchestras and ensembles - CIRMMT student symposium keynote." (2014), [Online]. Available: `https://youtu.be/RAdZd4NOGfg?si=94600ihJGduLzy8q`.

[9] F. Morreale and A. P. McPherson, "Design for longevity: Ongoing use of instruments from NIME 2010-14," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2017.

[10] J. Sullivan and M. M. Wanderley, "Stability, reliability, compatibility: Reviewing 40 years of DMI design," in *Proceedings of the International Conference on Sound and Music Computing (SMC)*, 2018.

[11] L. Zayas-Garin, J. Harrison, R. Jack, and A. McPherson, "DMI apprenticeship: Sharing and replicating musical artefacts," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2021.

[12] A. P. McPherson and Y. E. Kim, "The problem of the second performer: Building a community around an augmented piano," *Computer Music Journal*, vol. 36, no. 4, pp. 10–27, 2012.

[13] T. Fukuda, E. Meneses, T. West, and M. M. Wanderley, "The T-Stick music creation project: An approach to building a creative community around a DMI," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2021.

[14] I. Franco and M. M. Wanderley, "Prynth: A framework for self-contained digital music instruments," in *International Symposium on Computer Music Multidisciplinary Research (CMMR)*, 2016.

[15] F. Calegario, J. Tragtenberg, C. Frisson, *et al.*, "Documentation and replicability in the NIME community," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2021.

[16] A. Tom, H. Venkatesan, I. Franco, and M. M. Wanderley, "Rebuilding and reinterpreting a digital musical instrument - the sponge," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2019.

[17] "ISIDM mapping bibliography," *SensorWiki.org*, https://sensorwiki.org/isidm/mapping/bibliography 2018.

[18] A. Hunt, M. M. Wanderley, and R. Kirk, "Towards a model for instrumental mapping in expert musical interaction," in *Proceedings of the International Computer Music Conference (ICMC)*, 2000.

[19] D. Arfib, J. M. Couturier, L. Kessous, and V. Verfaille, "Strategies of mapping between gesture data and synthesis model parameters using perceptual spaces," *Organised Sound*, vol. 7, no. 2, pp. 127–144, 2002.

[20] D. van Nort, M. M. Wanderley, and P. Depalle, "On the choice of mappings based on geometric properties," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2004.

[21] A. Momeni and C. Henry, "Dynamic independent mapping layers for concurrent control of audio and video synthesis," *Computer Music Journal*, vol. 30, no. 1, pp. 49–66, 2006.

[22] D. van Nort, "Modular and adaptive control of sound processing," Ph.D. dissertation, McGill University, 2010.

[23] D. van Nort, M. M. Wanderley, and P. Depalle, "Mapping control structures for sound synthesis: Functional and topological perspectives," *Computer Music Journal*, vol. 38, no. 3, 2014.

[24] T. J. West, B. Caramiaux, and M. M. Wanderley, "Making mappings: Examining the design process," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2020.

[25] R. A. Fiebrink, "Real-time human interaction with supervised learning algorithms for music composition and performance," Ph.D. dissertation, Princeton University, 2011.

[26] D. Brown, C. Nash, and T. Mitchell, "Understanding user-defined mapping design in mid-air musical performance," in *Proceedings of the International Conference on Movement and Computing (MOCO)*, 2018.

[27] J. B. Rovan, M. M. Wanderley, S. Dubnov, and P. Depalle, "Instrumental gestural mapping strategies as expressivity determinants in computer music performance," *Kansei-The Technology of Emotion Workshop. Proceedings of the AIMI International Workshop*, pp. 68–73, 1997.

[28] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM computing surveys (CSUR)*, vol. 36, no. 1, pp. 1–34, 2004.

[29] J. Malloch, "A framework and tools for mapping of digital musical instruments," Ph.D. dissertation, McGill University, 2013.

[30] J. Chadabe, "The limitations of mapping as a structural descriptive in electronic instruments," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2002, pp. 1–5.

[31] J. Wang, J. Malloch, S. Sinclair, J. Wilansky, and M. M. Wanderley, "Webmapper: A tool for visualizing and manipulating mappings in digital musical instruments," *Proceedings of the International Symposium on Computer Music Multidisciplinary Research (CMMR)*, 2019.

[32] M. Lee, A. Freed, and D. Wessel, "Real-time neural network processing of gestural and acoustic signals," in *Proceedings of the International Computer Music Conference (ICMC)*, 1991.

[33] J. Françoise, N. Schnell, R. Borghesi, and F. Bevilacqua, "Probabilistic models for designing motion and sound relationships," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2014.

[34] B. D. Smith and G. E. Garnett, "The self-supervising machine," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2011.

[35] H. Scurto, F. Bevilacqua, and J. Françoise, "Shaping and exploring interactive motion-sound mappings using online clustering techniques," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2017.

[36] V. de las Pozas, "Semi-automated mappings for object-manipulating gestural control of electronic music," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2020.

[37] T. Murray-Browne and P. Tigas, "Latent mappings: Generating open-ended expressive mappings using variational autoencoders," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2021.

[38] T. J. West, "Making mappings: Examining the design process with libmapper and the T-Stick," M.A. thesis, McGill University, 2020.

[39] A. de Campo, "Lose control, gain influence - concepts for metacontrol," in *Proceedings of the International Computer Music Conference (ICMC)*, 2014.

[40] F. Bevilacqua, R. Müller, and N. Schnell, "MnM: A Max/MSP mapping toolbox," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2005.

[41] Ø. Brandtsegg, S. Saue, and T. Johansen, "A modulation matrix for complex parameter sets," *Proceedings of the International Conference on New Interfaces for Musical Expression*, no. 7491, pp. 316–319, 2011.

[42] J. Mandelis and P. Husbands, "Don't just play it, grow it! : Breeding sound synthesis and performance mappings," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2004.

[43] N. Gillian, R. B. Knapp, and S. O'Modhrain, "A machine learning toolbox for musician computer interaction," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2011.

[44] J. Bullock and A. Momeni, "ml.lib: Robust, cross-platform, open-source machine learning for Max and Pure Data," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2015.

[45] C. Kiefer, "Musical instrument mapping design with echo state networks," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2014.

[46] F. Visi, B. Caramiaux, and M. Mcloughlin, "A knowledge-based, data-driven method for action-sound mapping," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2017.

[47] A. Hunt, M. M. Wanderley, and M. Paradis, "The importance of parameter mapping in electronic instrument design," *Journal of New Music Research*, vol. 31, no. 1, pp. 1–12, 2002.

[48] S. D. Thorn and S. X. Wei, "Instruments of articulation: Signal processing in live performance," in *Proceedings of the International Conference on Movement and Computing (MOCO)*, 2019.

[49] J. Malloch and M. M. Wanderley, "The T-Stick: From musical interface to musical instrument," *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2007.

[50] J. Malloch, "A consort of gestural musical controllers: Design, construction, and performance," M.A. thesis, McGill University, 2008.

[51] X. Pestova, E. Donald, H. Hindman, *et al.*, "The CIRMMT/McGill digital orchestra project," in *Proceedings of the International Computer Music Conference (ICMC)*, 2009.

[52] J. Malloch, D. Birnbaum, E. Sinyor, and M. M. Wanderley, "Towards a new conceptual framework for digital musical instruments," in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, 2006.

[53] A. Nieva, J. Wang, J. Malloch, and M. M. Wanderley, "The T-Stick: Maintaining a 12 year-old digital musical instrument," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2018.

[54] J. Malloch, "A framework and tools for mapping of digital musical instruments," Ph.D. dissertation, McGill University, 2013.

[55] E. A. L. Meneses, T. Piquet, J. Noble, and M. M. Wanderley, "The Puara framework: Hiding complexity and modularity for reproducibility and usability in NIMEs," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2023.

[56] A.-N. Niyonsenga and M. M. Wanderley, "Tools and techniques for the maintenance and support of digital musical instruments," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2023.

[57] A.-N. Niyonsenga and M. M. Wanderley, "Take five: Improving maintainability and reliability of the T-Stick," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2024.

[58] N. Collins, "Low brass: The evolution of trombone-propelled electronics," *Leonardo Music Journal*, vol. 1, pp. 41–44, 1991.

[59] J. Snyder, M. Mulshine, and P. Art, "The feedback trombone: Controlling feedback in brass instruments," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2018.

[60] S. Lemouton, M. Stroppa, and B. Sluchin, "Using the augmented trombone in "I will not kiss your f.ing flag"," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2006.

[61] N. Farwell, "Adapting the trombone: A suite of electro-acoustic interventions for the piece Rouse," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2006.

[62] T. Henriques, "Double slide controller," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2009.

[63] C. Roads, *Microsound*. MIT Press, 2001.

[64] A. Çamcı, "GrainTrain: A hand-drawn multi-touch interface for granular synthesis," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2018.

[65] C. Carlson and G. Wang, "Borderlands: An audiovisual interface for granular synthesis," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2012.

[66] D. Schwarz, G. Beller, B. Verbrugghe, and S. Britton, "Real-time corpus-based concatenative synthesis with CataRT," in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, 2006.

[67] B. Caramiaux, J. Françoise, N. Schnell, and F. Bevilacqua, "Mapping Through Listening," *Computer Music Journal*, vol. 38, no. 3, pp. 34–48, 2014.

[68] J. Françoise, "Gesture-sound mapping by demonstration in interactive music systems," in *Proceedings of the 21st ACM international conference on Multimedia (MM'13)*, 2013.

[69] G. Wang, J. Oh, S. Salazar, and R. Hamilton, "World Stage: A crowdsourcing paradigm for social / mobile music," in *Proceedings of the International Computer Music Conference (ICMC)*, 2011.

[70] S. O. H. Madgwick, "AHRS algorithms and calibration solutions to facilitate new applications using low-cost MEMS," Ph.D. dissertation, University of Bristol, 2014, p. 273.

[71] R. Mahony, T. Hamel, and J.-M. Pflimlin, "Nonlinear complementary filters on the special orthogonal group," *IEEE Transactions on Automatic Control*, vol. 53, no. 5, pp. 1203–1218, 2008.

[72] G. B. Fernandez and K. Larson, "Tune field," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2021.

[73] A. P. McPherson, R. H. Jack, and G. Moro, "Action-sound latency: Are our tools fast enough?" In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2016.

[74] R. Vieira and F. Schiavoni, "Fliperama: An affordable Arduino based MIDI controller," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2020.

[75] D. Cavdir and G. Wang, "Felt sound: A shared musical experience for the deaf and hard of hearing," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2020.

[76] F. A. Robinson, "Debris: A playful interface for direct manipulation of audio waveforms," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2021.

[77] N. Renney, H. Renney, T. J. Mitchell, and B. R. Gaster, "Studying how digital luthiers choose their tools," in *CHI Conference on Human Factors in Computing Systems (CHI '22)*, 2022.

[78] W. J. Slager, "Designing and performing with Pandora's box: Transforming feedback physically and with algorithms," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2021.

[79] H. Ulfarsson and A. P. Melbye, "Sculpting the behaviour of the feedback-actuated augmented bass: Design strategies for subtle manipulations of string feedback using simple adaptive algorithms," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2020.

[80] J.-M. Celerier, "Rage against the glue: Beyond run-time media frameworks with modern C++," in *Proceedings of the International Computer Music Conference (ICMC)*, 2022.

[81] C. B. Medeiros and M. M. Wanderley, "A comprehensive review of sensors and instrumentation methods in devices for musical expression," *Sensors*, vol. 14, pp. 13 556–13 591, 2014.

[82] M. T. Marshall, "Physical interface design for digital musical instruments," Ph.D. dissertation, McGill University, 2008.

[83] H. Úlfarsson, "The halldorophone: The ongoing innovation of a cello-like drone instrument," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2018.

[84] A. Eldridge and C. Kiefer, "The self-resonating feedback cello: Interfacing gestural and generative processes in improvised performance," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2017.

[85] M. T. Marshall and M. M. Wanderley, "Vibrotactile feedback in digital musical instruments," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2006.

[86] S. Thorn and B. Lahey, "Decolonizing the violin with active shoulder rests (ASRs)," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2022.

[87] S. Papetti, M. Civolani, and F. Fontana, "Rhythm'n'shoes: A wearable foot tapping interface with audio-tactile feedback," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2011.

[88] M. Giordano, S. Sinclair, and M. M. Wanderley, "Bowing a vibration-enhanced force feedback device," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2012.

[89] I. Hattwick, "The creation of hardware systems for professional artistic productions," Ph.D. dissertation, McGill University, 2017.

[90] T. J. West, A. Bachmayer, S. Bhagwati, J. Berzowska, and M. M. Wanderley, "The design of the Body:Suit:Score, a full-body vibrotactile musical score," in *Human Interface and the Management of Information. Information in Intelligent Systems*, S. Yamamoto and H. Mori, Eds., Cham: Springer International Publishing, 2019, pp. 70–89.

[91] M. Kirkegaard, "Integrating 1-DoF force feedback interactions in self-contained DMIs," M.A. thesis, McGill University, 2021.

[92] C. Frisson, M. Kirkegaard, T. Pietrzak, and M. M. Wanderley, "ForceHost: An open-source toolchain for generating firmware embedding the authoring and rendering of audio and force-feedback haptics," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2022.

[93] M. Kirkegaard, M. Bredholt, C. Frisson, and M. M. Wanderley, "TorqueTuner: A self contained module for designing rotary haptic force feedback for digital musical instruments," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2020.

[94] B. Boettcher, J. Malloch, J. Wang, and M. M. Wanderley, "Mapper4Live: Using control structures to embed complex mapping tools into Ableton Live," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2022.

[95] J. Malloch, M. Schumacher, and S. Sinclair, "The Digital Orchestra Toolbox for Max," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2018.

[96] R. Fiebrink, D. Trueman, and P. R. Cook, "A meta-instrument for interactive, on-the-fly machine learning," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2009.

[97] B. Caramiaux, N. Montecchio, A. Tanaka, and F. Bevilacqua, "Adaptive gesture recognition with variation estimation for interactive systems," *ACM Transactions on Interactive Intelligent Systems*, vol. 4, no. 4, 2014.

[98] M. Marier, "Designing mappings for musical interfaces using preset interpolation," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2012.

[99] T. Todoroff and L. Reboursière, "1-D, 2-D and 3-D interpolation tools for Max/MSP/Jitter," in *Proceedings of the International Computer Music Conference (ICMC)*, 2009.

[100] R. Bencina, "The Metasurface – applying natural neighbour interpolation to two-to-many mapping," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2005.

[101] B. Caramiaux, J. Françoise, N. Schnell, and F. Bevilacqua, "Mapping through listening," *Computer Music Journal*, vol. 38, no. 3, 2014.

[102] R. F. Ohad Fried, "Cross-modal sound mapping using deep learning," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2013.

[103] L. Dahl, "Designing new musical interfaces as research: What's the problem?" *Leonardo*, vol. 49, no. 1, pp. 76–77, 2016.

[104] J. Babosa, J. Malloch, M. M. Wanderley, and S. Huot, "What does "evaluation" mean for the NIME community?" In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2015.

[105] D. Brown, C. Nash, and T. Mitchell, "A user experience review of music interaction evaluations," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2017.

[106] P. J. C. Reimer and M. M. Wanderley, "Embracing less common evaluation strategies for studying user experience in nime," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2021.

[107] M. Rodger, P. Stapleton, M. van Walstijn, M. Ortiz, and L. Pardue, "What makes a good musical instrument? A matter of processes, ecologies and specificities," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2020.

[108] D. Fober, Y. Orlarey, and S. Letz, "FAUST architectures design and OSC support," in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, 2011.

[109] P. Rothfuss, *The Name of the Wind*. DAW Books, 2007.

[110] J. Lakos, *Large-Scale C++: Process and Architecture*. Addison-Wesley Professional, 2019, vol. 1.

[111] A. P. McPherson, "Portable measurement and mapping of continuous piano gesture," in *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, 2013.

[112] D. R. Olsen Jr, "Evaluating user interface systems research," 2007.