University of California
Santa Barbara

Stride: A Language for Sound Synthesis, Processing, and Interaction Design

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Media Arts and Technology

by

Joseph Tilbian

December 2018

The Dissertation of **Joseph Tilbian** is approved.

Theodore Kim

Matthew Wright

Andrés Cabrera

Curtis Roads, Committee Chair

June 2018

Stride: A Language for Sound Synthesis, Processing, and Interaction Design

For my parents:

Haig Tilbian

Mary Movsesian

Acknowledgments

This dissertation is one of the outcomes of a close collaboration that started in late 2013 between Dr. Andrés Cabrera and me. The drive behind this collaboration was a mutual interest to design microcontroller-based electronic platforms for sound synthesis, processing, and interaction design to supplement some of the existing popular electronic platforms which were designed for physical computing and general-purpose computation. The novelties and contributions of this dissertation are the result of hundreds of hours of conversations and discussions to design something powerful yet simple and elegant. I would like to express my sincere gratitude to Andrés Cabrera for being an indispensable partner throughout the ongoing collaboration.

I would like to thank my advisor and the chair of my dissertation committee Professor Curtis Roads and the members of the committee Professor Theodore Kim, Dr. Matthew Wright, and Dr. Andrés Cabrera for the encouragement and invaluable feedback they provided throughout the process.

I would like to thank Professor JoAnn Kuchera-Morin for having me as part of the AlloSphere Research Group as a Graduate Student Researcher and supporting this research work.

I would like to thank the Robert W. Deutsch Foundation for the generous graduate fellowship grant made possible through the AlloSphere Research Group at UCSB.

I would like to thank the UCSB Center for Research in Electronic Art Technology (CREATE) for a generous grant to support the development of Stride and the Systemics Lab for letting us borrow their electronic test equipment.

I would like to thank the Media Arts and Technology Program and Dr. Andrés Cabrera for the teaching assistant positions for the Digital Audio Programming course series.

I would like to thank Steffen Martin and Łukasz Olczyk for their help with the design, development, and testing of the Saturn M7 Audio Development Board, which became a target platform for Stride.

I would like to thank my colleagues at the Media Arts and Technology Program in particular Sahar Sajadieh, Gustavo Rincon, Şölen Kıratlı, Owen Campbell, and Hanna Wolfe for their friendship and the great memories.

I finally want to thank my family for their unconditional love and support.

Curriculum Vitæ

Joseph Tilbian

Education

2018	Ph.D. in Media Arts and Technology University of California, Santa Barbara
2006	M.Sc. in Mechatronics University of Applied Sciences, Aachen - Germany
2002	B.E. in Mechanical Engineering American University of Beirut, Beirut - Lebanon

Publications

J. Tilbian and A. Cabrera, "**Stride: A Declarative and Reactive Language for Sound Synthesis and Beyond,**" in Proceedings of the 2016 International Computer Music Conference, Utrecht, 2016, pp. 472-478.

J. Tilbian and A. Cabrera, "**Stride for Interactive Musical Instrument Design,**" in Proceedings of the 2017 International Conference on New Interfaces for Musical Expression, Copenhagen, 2017, pp. 446-449.

J. Tilbian, A. Cabrera, S. Martin, and Ł. Olczyk, "**Stride on Saturn M7 for Interactive Musical Instrument Design,**" in Proceedings of the 2017 International Conference on New Interfaces for Musical Expression, Copenhagen, 2017, pp. 503-504.

Abstract

Stride: A Language for Sound Synthesis, Processing, and Interaction Design

by

Joseph Tilbian

This dissertation presents Stride, a language for sound synthesis, processing, and interaction design. With a novel and unique approach for handling sampling rates as well as clocking and computation domains, Stride prompts the generation of highly optimized target code. Optimization is achieved by giving the user of Stride control over the Stride code generator through its syntax. The optimizations render Stride an ideal language to target resource-constrained devices such as microcontrollers. Stride is a declarative language and adopts features from dataflow languages. With only two syntactic constructs, Stride is easy to learn. Through resource abstraction and separation of semantics from implementation, a wide range of computation devices could be targeted such as microcontrollers, system-on-chips, general-purpose computers, and heterogeneous systems. Users of Stride can write code once and deploy on any supported hardware.

After presenting the challenges of targeting resource-constrained microcontrollers with popular music programming languages in use today for sound synthesis and processing, a new programming language and its syntax are introduced to address these challenges. This is followed by demonstrating how the language enables its user to control the code generation process to yield efficient and optimized target code. Next, the semantics of the language and some of its core building blocks are presented in detail followed by the user-controlled concurrency model built into the language. Designing interaction using some of the core blocks is then presented through a set of examples followed by some of the advanced building blocks of the language. Finally, the language is presented as part of an encompassing development environment and all of its components including the integrated development environment and the compiler.

Contents

Curriculum Vitæ **vii**

Abstract **viii**

1 Introduction **1**

1.1 Scope 2
1.2 Problem Statements 4
1.3 Research Questions 5
1.4 Contributions 8
1.4.1 A New Syntax 8
1.4.2 Signals with Rates and Domains 9
1.4.3 Code Generation 9
1.4.4 Concurrency 10
1.5 Dissertation Structure 10
1.6 Permissions and Attributions 12
1.7 Additional Notes 12

2 Survey of Music Programming Languages **13**

2.1 Music Programming Languages 14
2.1.1 Csound (1985) 14
2.1.2 Pd - Pure Data (1996) 15
2.1.3 SuperCollider (1996) 17
2.1.4 Faust (2002) 18
2.1.5 ChucK (2003) 19
2.1.6 Discussion 20
2.2 Concurrent Research 20
2.2.1 Kronos 20
2.2.2 WaveCore 21

3 **Faust and Targeting Microcontrollers** **22**
3.1 A Simple Faust Example 22
3.2 Computing Constants 27
3.3 Processing Loads and Relative Rates 28
3.4 Concurrency 33
3.5 Vector Processing 36
3.6 Libraries and APIs 37
3.7 Summary 38

4 **Improvements with Stride** **39**
4.1 An Oscillator with Frequency Control 39
4.2 A New Language 41
4.2.1 Block Declarations 41
4.2.2 Stream Expressions 43
4.3 Code Generation for an Embedded Platform 44
4.3.1 Oscillator with Frequency Control 45
4.3.2 Oscillator's Frequency Control at Reduced Rate 49
4.3.3 Reactive Control of the Oscillator's Frequency 51
4.3.4 Audio Callback Optimization 54
4.4 Discussion 56
4.5 Summary 58

5 **Signals, Rates, Domains, and Modules** **59**
5.1 Behavior of a Signal 60
5.1.1 Rates 60
5.1.2 Domains 63
5.2 A Sine Oscillator Module in Stride 66
5.2.1 Sine Oscillator Stream Expressions 66
5.2.2 Sine Oscillator Module 72
5.2.3 Code Generation for the Sine Oscillator Module 78
5.3 Using Modules in Stride 80
5.3.1 Level Module 80
5.3.2 Synchronous Frequency Modulation 83
5.3.3 Asynchronous Frequency Modulation 86
5.4 Summary 89

6 **Domains and Concurrency** **90**
6.0.1 Domain Execution Order 91
6.0.2 Concurrency Declaration 92
6.1 Concurrency and Stateless C++ Template Classes 93
6.1.1 Asynchronous Frequency Modulation with Concurrency 95
6.2 Discussion 97

6.3 Summary 98

7 Interaction Design with Triggers and Reactions **99**
7.1 The Switch Block 100
7.2 The Trigger Block 101
7.2.1 Single Domain Trigger Example 102
7.2.2 Multiple Domain Trigger Example 104
7.3 Reactions 117
7.4 Attack/Decay Envelope in Stride 120
7.5 Summary 128

8 Advanced Blocks in Stride **129**
8.1 The Buffer Block 129
8.1.1 Buffer Block as Delay Line 130
8.1.2 Buffers and Hardware IO Abstraction 133
8.1.3 Buffers and Vector Operations 136
8.2 The Loop Block 142
8.3 The Group Block 148
8.4 Summary 154

9 Stride **155**
9.1 Language Features 156
9.2 Stride Environment 159
9.2.1 Stride Systems 159
9.2.2 Stride Compiler 160
9.3 Stride IDE 163
9.4 Stride Syntax 164
9.4.1 Basic Blocks 165
9.4.2 Block Bundles 172
9.4.3 Advanced Blocks 173
9.4.4 Stream Expressions 178
9.5 Summary 191

10 Conclusion **192**
10.1 Summary 192
10.2 Discussion 195
10.3 Future Work 197

A Faust DSP and Generated Code **199**
A.1 Resonant Low Pass with Constant Arguments 199
A.1.1 Faust DSP Code 200
A.1.2 C++ Generated Code 200

A.2 Resonant Low Pass with Variable Arguments 203
A.2.1 Faust DSP Code 203
A.2.2 C++ Generated Code 204
A.3 Bela Template Code for Faust 207

B Relative Computation Cost of Floating-Point Operations **217**
B.1 Relative Computation Cost Measurement 217
B.1.1 Results 218
B.1.2 Source Code 218

C Frequency Modulation in Stride **223**
C.1 Synchronous and Asynchronous Modulation 224
C.1.1 Synchronous Frequency Modulation 224
C.1.2 Asynchronous Frequency Modulation 231
C.1.3 Asynchronous Frequency Modulation with Concurrency 239

D Stride Helper Classes **248**
D.1 Synchronization 248
D.2 Signals 249
D.3 Trigger Observers 252
D.4 Triggers 252

E Stride Lexeme and Grammar **256**
E.1 Stride Lexeme 256
E.2 Stride Grammar 260

Bibliography **275**

Terms and Abbreviations **279**

Glossary **280**

List of Figures **281**

List of Tables **284**

List of Codes **288**

Chapter 1

Introduction

Over the past two decades single-board computers with microcontrollers and system-on-chips as their main processor have become popular among artists, hobbyists, and "do-it-yourself" enthusiasts. Their popularity can be attributed to making the programming of these small computers easier, thus making them more accessible to people who lack the technical expertise required otherwise.

Most single-board computers have been designed with physical computing, general-purpose computing, or graphical applications in mind. Supplemental boards and hardware are usually required with these boards to generate good quality sound and enable the control of synthesis and processing parameters in real time.

Single-board computers designed for low latency, high fidelity, and high-resolution audio applications such as sound capture, reproduction, synthesis, and processing with real-time response capabilities are rare. One of the reasons appears to be the lack of a modern language for sound synthesis and processing to target the computers that power such boards.

1.1 Scope

This dissertation covers the design of a new programming language for sound synthesis, processing, and interaction design to target resource-constrained single-board computers specifically and any computer or computer system generally.

Popular music programming languages for sound synthesis and processing, running on modern general-purpose computers characterized by their computational power and the abundance of memory, achieve desired qualities such as real-time performance, high sound resolution (bit depth and sampling rate), and precision (double precision floating-point). However, microcontrollers generally only possess a small fraction of the computation power those machines offer. Many synthesis techniques also require ample memory which is also a scarce resource on microcontrollers.

Historically, dedicated Digital Signal Processor (DSP) Integrated Circuits (IC) with dedicated external memory have been used to achieve those qualities, enabled through on-board circuits to perform general-purpose and specific signal processing tasks, such as single instruction multiply-add operations or Fast Fourier Transforms.

In recent years, with the introduction of microcontrollers designed to target multimedia applications, such as those designed around an Arm® Cortex®-M core, the line between DSPs and microcontrollers has become blurred.

These modern microcontrollers have Central Processing Units (CPU) clocked at three-digit MHz speeds, come with dedicated single precision or double precision Floating-Point Units (FPU), and are capable of performing multiply add operations as well as operate on multiple data with a single CPU instruction in a single clock cycle. These are features common to DSPs. These microcontrollers also feature peripherals supporting electrical serial bus interface standards for digital audio communication among ICs like I^2S (Inter-IC Sound) or S/PDIF (Sony/Philips Digital Interface) among others.

Another stark difference between general-purpose computers and microcontrollers appears in the need for an operating system. Almost all general-purpose computers today run an operating system that hosts applications designed for it. Microcontrollers on the other hand either run a real-time operating system or run bare metal[1]. Running code on a bare metal microcontroller reduces overhead introduced by an operating system. This latter case is an important consideration when designing a language to target microcontrollers.

Therefore, designing a language and a code generator that could target bare metal microcontrollers and produce code with the smallest possible footprint and least overhead is one, if not the most important, criterion to consider.

Because microcontrollers have limited resources, it is important to give the user the ability to control how often computations happen and in which context these computations happen. Another consideration is giving the user control over the code generator through the language syntax, rather than through passing compilation flags to the compiler, to generate efficient and optimized code that meets the computational

[1] A computer system that does not contain an operating system.

or aesthetic needs of the user.

Prior to embarking on the design of a new language it is only reasonable to assess whether current music programming languages would be up to the task of supporting resource-constrained systems by introducing modifications to their syntax and/or internal processes.

From this point on we will refer to single-board computers as microcontroller-based embedded systems[2].

1.2 Problem Statements

Popular computer music languages in use today (Csound, Pd, SuperCollider, Faust, and ChucK) for sound synthesis and processing are designed for general-purpose computers or embedded systems running an operating system. Although some produce highly efficient code, they are not designed to run on or generate code for resource-constrained microcontroller-based embedded systems.

To reduce the overhead introduced by running an operating system on a resource-constrained system it is paramount to run bare metal. Programming bare metal systems is not trivial and requires expert knowledge of the target device making them inaccessible to artists, hobbyists, and "do-it-yourself" enthusiasts.

Microcontroller cores designed to perform digital signal processing tasks come with

[2]An embedded system is a dedicated computer system designed and embedded in a device that includes various electrical and mechanical components.

dedicated digital signal processing libraries that are optimized for the core. Abstracting these libraries and giving the user the ability to utilize them during code generation is of utmost importance to take full advantage of the device's capabilities and optimize for it.

Microcontrollers communicate with the outside world through peripherals. Software running on a microcontroller, usually referred to as firmware, controls and communicates with these peripherals through drivers[3]. Different microcontroller manufacturers have different hardware implementations and usually provide drivers for them. This renders code generated for one target useless for another target. Separating semantics from implementation and abstracting hardware and drivers in a uniform way across manufacturers is one way to enable moving code from one device to another.

Modern microcontrollers can have more than one core or be part of a heterogenous system. Concurrency and data integrity become immediate concerns that need to be addressed especially when memory is shared between cores or computations are distributed across components of a heterogenous system.

1.3 Research Questions

The following are the research questions that arise from the problem statements described above as well as ones related to designing a programming language for sound synthesis, processing, and interaction design.

[3]A piece of software that abstracts hardware and enables an operating system or other software to communicate with the hardware.

Q1 *Can a language for sound synthesis, processing, and interaction be designed with only a few syntactic constructs that meet the following specifications?*

- *Simplify or unify the interfacing between entities*
- *Enable parallel expansion of entities and interfaces*
- *Abstract the static and dynamic allocation of entities*
- *Perform computations on a per sample basis, on real and complex numbers, in both time and frequency domains*
- *Handle synchronous and asynchronous data and events*
- *Abstract threading and thread synchronization*
- *Enable seamless interfacing of entities running at different rates and in different threads*

If one were to design a modern language for sound synthesis, processing, and interaction design, the language should meet most of, if not all, the specifications put forward by this question. It is also important to incorporate most of the features from existing music programming languages that have made them popular and successful among their users. Given the processing capabilities of host computers today, it is possible to design complex interpreters that not only parse, process, and interpret user code in fractions of a second but also analyze the code and recommend potential optimizations to the user.

Q2 *How can the unit generator / processor approach be adapted to resource-constrained systems to enable optimized code generation with the smallest memory footprint?*

How and to what extent can a user control the optimization? How would the units behave in a multi-threaded or heterogenous environment?

The unit generator / processor design approach has been incorporated into almost all music programming languages since its inception as part of MUSIC III in the sixties by Max Mathews. In modern computer music languages, designed around the object-oriented programming paradigm, unit generators and processors are abstracted as classes from which instances of these units are created. Depending on the type of the unit, the class that represents it might incorporate states. In a multi-core or heterogenous system, where control and signal computations can be distributed across various threads, this abstraction of unit generators might not meet the optimization goals required by resource-constrained targets because it would result in the need for a class to accommodate various concurrency scenarios.

Q3 *Could various hardware components (inputs, outputs, clocks, cores, etc.) and software architectures (application programming interfaces, real-time operating systems, etc.) be abstracted in a unified way?*

To make the user code portable from one target to another the underlying hardware and software architectures need to be abstracted. This can be achieved by separating semantics from implementation.

Q4 *Can various types of interactions with the system be abstracted in a unified way?*

Interactions with a microcontroller-based embedded system can come in various forms and from multiple sources. These interactions could be used to control the sound synthesis and processing parameters on the system. Interactions could come from a knob or switch, a sensor, a periodic or aperiodic impulse train, a message over a serial bus peripheral or over a network (following message protocols like MIDI[1] or Open Sound Control (OSC)[2]). Abstracting these interactions in a unified way and incorporating it into the language would allow the user to seamlessly switch from one interaction type to another without having to modify the core synthesis and processing blocks.

1.4 Contributions

This dissertation makes the following contributions to the field of computer music in general and to sound synthesis and processing in particular.

1.4.1 A New Syntax

Stride presents a new syntax to design signal processing algorithms as well as to implement and develop sound synthesis techniques with real-time control. The syntax is made up of only two constructs. The syntax is mostly declarative which allows for expanding the capabilities of the language by adding new "blocks". Entities in the language are connected to each other with a single operator. Unlike regular dataflow languages where only data is exchanged between connected entities, in Stride information provided by the user in the code is also exchanged between entities, such as rates and domains[4].

[4]A context in Stride where code is executed. The concept is discussed in detail in this dissertation.

1.4.2 Signals with Rates and Domains

In Stride, data exchange between entities is abstracted through signals. Stride takes a novel approach by allowing the user to specify the rate and the domain of every signal, giving the user control over how often an expression where the signal appears is evaluated and in which thread and computation device this evaluation takes place. The rate and domain of signals specified by the user propagate throughout the code to replace placeholder aliases of other signals that are embedded within modules. This approach puts the user in full control of generating code optimized for a given target.

1.4.3 Code Generation

Modules in Stride, which abstract unit generators and processors, translate to stateless C++ template classes through its code generator and helper classes. The generated code significantly differs from ones that usually appear in music synthesis and processing libraries or ones generated by other music programming languages. This approach results in classes with independent processes mapped to domains which in turn can be distributed across threads and devices, thus breaking down and distributing computations defined inside a unit generator or processor.

1.4.4 Concurrency

Stride incorporates a flexible concurrency model that is controlled by the user by defining policies controlling shared memory between domains. This model allows for the segmentation and distribution of processes across threads and devices while maintaining data integrity and avoiding race conditions and priority inversions.

1.5 Dissertation Structure

This dissertation is presented through the following chapters:

In chapter 2, "Survey of Music Programming Languages", we present a set of popular music programming languages in use today and discuss their limitations or incompatibility with targeting resource-constrained systems. We also present concurrent research related to the problem statements and research questions posed and addressed by this dissertation.

In chapter 3, "Faust and Targeting Microcontrollers", we discuss the limitations of Faust when it comes to generating optimized code for a microcontroller-based embedded system. We also identify a few optimization schemes that could result in efficient code for such systems.

In chapter 4, "Improvements with Stride", we introduce a new programming language and its syntax through a simple sine oscillator example, we demonstrate how the user of this language could generate efficient and optimized code.

In chapter 5, "Signals, Rates, Domains, and Modules", we present the core building blocks of Stride and discuss their semantics and behaviors. We demonstrate the use of these blocks through examples of synchronous and asynchronous frequency modulation.

In chapter 6, "Domains and Concurrency", we present the user-controlled concurrency scheme built into the language and discuss how it affects the code generation process.

In chapter 7, "Interaction Design with Triggers and Reactions", we demonstrate how interaction is modeled and designed in Stride by presenting additional core building blocks of the language. We also demonstrate how a state machine is created in Stride.

In chapter 8, "Advanced Blocks in Stride", we present additional building blocks of the language, which bring advanced features to the language and simplify the user code.

In chapter 9, "Stride", we present the features of Stride as a programming language. We also present the language as a component of the Stride Environment which also comprises a compiler and an integrated development environment. We also present the formal definitions of the core and advanced building blocks of the language and expand on the semantics that control expressions in the language.

In chapter 10, "Conclusion", we summarize the research carried out to produce this dissertation and address its research questions. We also discuss related future work.

1.6 Permissions and Attributions

The syntax diagrams for the grammar were generated using Railroad Diagram Generator by Gunther Rademacher, URL: `http://www.bottlecaps.de/rr/ui` [accessed November 7, 2018].

All other figures, charts, and diagrams appearing in this dissertation have been created by the author for the purpose of this document.

1.7 Additional Notes

All Stride code examples included in this dissertation are shown in their expanded form. All block properties and their default values are explicitly stated, which is not generally required.

Chapter 2

Survey of Music Programming Languages

Targeting microprocessors and DSP boards with music programming languages to achieve real-time control in sound synthesis and processing has a long history dating back to the late seventies and early eighties. In this chapter we will briefly touch on a few of these languages and focus on ones that are still in use today or were introduced later for general-purpose computers. We will also mention some concurrent research and projects.

2.1 Music Programming Languages

One of the early examples of a music programming language targeting a DSP board is the 4CED language[3] designed to target the 4C machine[4] at Institut de recherche et coordination acoustique/musique (IRCAM), hosted on a PDP-11 computer, for real-time sound synthesis.

Max[5][6] (currently sold as a commercial product, Max/MSP, by Cycling '74 and Ableton) was developed to run on a NeXT machine as part of the IRCAM Music Workstation to target signal processing boards based on the intel i860 microprocessor[7].

A comprehensive list of music programming languages to target DSP boards along with their host machines can be found in The Computer Music Tutorial[8, chapter 17].

2.1.1 Csound (1985)

Csound[9] designed and developed by Barry Vercoe and introduced in 1985 followed the MUSIC-N model and was a translation of MUSIC11 into the C programming language (C), making it host independent.

In 1989 Csound was used to target Inmos transputers[10] to considerably enhance its speed of execution.

In 1990 a new version of Csound[11] was introduced with real-time capabilities which could run on a MacII host to target a real-time DSP system based on a Motorola DSP56000.

The real-time capabilities of Csound with greatly expanded with the introduction of Csound Extended[12] where support for the SHARC 21060 DSP by Analog Devices was introduced.

Comments on Csound

Although still popular and in use today, the syntax of Csound is a markup language for defining instruments and a score, which seems outdated when compared to most modern programming languages since its roots are in the MUSIC-N family of languages.

To introduce new unit generators or algorithms that run efficiently the user has to write them in C and introduce corresponding opcodes into Csound using an Application Programming Interface (API). This presents a particular challenge when it comes to microcontrollers, specifically when it comes to using their optimized DSP libraries. A new opcode that performs the same task is required to target a different core.

Csound is not designed for single sample processing[1] and it only supports two rates: control rate and audio rate.

2.1.2 Pd - Pure Data (1996)

Pd[13] was introduced in 1996 by Miller Puckette based on his earlier work on Max and FTS[14] at IRCAM. Like Max, Pd is a graphical programming language. Unlike Max, Pd

[1] The control rate could be temporarily set to one (setksmps 1). However, this is not efficient.

was designed to perform all control and audio processing on the host’s CPU rather than target a DSP system.

Like Csound, Pd can be extended through an API to enable users to add their own control and audio processing code written in C.

Comments on Pd

Targeting microcontrollers with Pd is not ideal since Pd is designed to dynamically invoke objects’ methods at runtime. For non-audio signals these methods are invoked based on events. Invoking methods dynamically adds an overhead which sometimes taxes the system more than the actual process the method accomplishes. One way to overcome this would be to take the signal graph from Pd and generate static target code from it. This approach has been successfully implemented by Enzien Audio with their Heavy compiler[15]. The compiler supports a limited list of Pd objects.

Since Pd follows the dataflow programming paradigm, it suffers from limitations when it comes to object-oriented programming concepts. Creating parallel processes or managing a large list of objects is not possible due to the lack of constructors and destructors.

Single sample processing in Pd is not possible (Max/MSP introduced Gen to achieve single sample processing). Pd runs at two rates. The first is the audio sampling rate and the second is the control rate where data is processed once per 64 samples of audio.

2.1.3 SuperCollider (1996)

In 1996, James McCartney introduced SuperCollider[16], an environment for real time synthesis. It featured a programming language designed on the object-oriented programming paradigm. It supported closures and had a garbage collector.

Later versions of SuperCollider featured two applications, a client and a server, which communicated over a modified version of OSC[17]. The server ran the synthesis engine and the client ran on top of the language engine. Multiple clients could connect to a single server and perform in real-time.

SuperCollider has a synthesis class library which generates C++ code that can be loaded on its synthesis engine running on the server. The synthesis engine also has a C linkage API which allows users to write instruments in C and load them on the server.

Instead of supporting a single control rate, unit generators can be written to run at any power of two division of the audio clock rate. Values are linearly interpolated when connecting to unit generators running at different rates.

Comments on SuperCollider

SuperCollider's language syntax and architecture was a departure from the markup and graphical languages for music programming. Due to its architecture, SuperCollider is not suited for microcontroller-based embedded systems. However, it offers many valuable solutions that one needs to consider when designing a new language.

SuperCollider can achieve single sample processing if the audio buffer size on the synthesis engine running on the server is set to one. However, this requires a machine capable of handling the load introduced by this change.

2.1.4 Faust (2002)

Designed at Grame (Centre National de Création Musicale) by by Yann Orlarey et al., Faust[18] was first introduced in 2002. It is a purely functional programming language with an algebraic block diagram syntax.

Faust compiles its block diagram syntax to highly efficient C++ code which operates at the sample level. Operating at the sample level makes it possible to create recursions (sample feedback) and create low-level signal processing functions. These functions can then be brought together using high-level composition operators to create more complex signal processing functions.

Faust does not rely on any external modules or libraries to generate code and is self-contained. The generated static C++ code could be compiled and used on any target as long as the target has a C++ compiler.

Comments on Faust

Faust would be the ideal candidate among the languages presented in this section to target microcontrollers. The following chapter is dedicated to discussing the capabilities and limitations of Faust when it comes to targeting embedded systems.

2.1.5 ChucK (2003)

ChucK, designed and developed by Ge Wang et al., was introduced in 2003. ChucK is a concurrent and strongly timed language for real-time sound synthesis, composition, and performance.

The syntax of Chuck is C-like and designed with the object-oriented paradigm. The ChucK operator (=>) is used to connect entities together. Because of its strong unified timing mechanism, it is capable of multi-rate events and control processing. ChucK code is dynamically compiled to ChucK virtual machine bytecode that runs on the Chuck Virtual Machine. This architecture allows for on-the-fly programming in ChucK. A "Shred" in ChucK abstracts threads and fits into the concurrency model built into ChucK. Single sample processing is supported in ChucK since the user is responsible for "advancing time" and can do so by the duration of a single sample.

Comments on ChucK

Because of its architecture and reliance on a virtual machine, ChucK is not suited for microcontroller-based embedded systems for sound synthesis and processing. However, many features of ChucK are worth considering when designing a new language, specifically the ChucK Operator which seamlessly enables the connection of entities running at different rates.

2.1.6 Discussion

Having reviewed some of the popular music programming languages in use today, Faust seems to be the best candidate to consider for targeting microcontrollers. Faust meets many of the desired specifications to target a resource constrained system, specifically its ability to generate efficient C++ code and define low-level signal processing functions.

Many features from the other languages are worth considering if one were to design a modern language, particularly ones that result in the new language meeting the specifications set forward in the introduction of this dissertation.

2.2 Concurrent Research

The following research works are closely related to that presented in this dissertation.

2.2.1 Kronos

Kronos[19] is a functional high-level language and a just-in-time compiler[20]. It is well suited to build digital signal processing solutions due to its capability to generate high performance code. The language implements the functional reactive paradigm and can handle multi-rate processing.

2.2.2 WaveCore

WaveCore[21] is a coarse-grained reconfigurable processor (CGRP) architecture, based on the dataflow paradigm. It is designed to target any Field-Programmable Gate Array (FPGA) because it is designed in VHDL[2], which is a target independent language. The WaveCore programming model is based on explicitly describing a dataflow graph in a declarative manner.

Prior to WaveCore, finite difference physical models of musical instruments were implemented on FPGAs that can be configured, modified, and played in real time[22]. However, WaveCore abstracts the implementation with a scalable and interconnected cluster of Processing Units, where each unit embodies a small floating-point RISC processor.

An experimental compiler has been designed to target the WaveCore Processor with Faust code. Kronos was also used to target the WaveCore Processor to design a low-latency parallel graphic equalizer[23].

[2]VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

Chapter 3

Faust and Targeting Microcontrollers

Faust has its limitations when it comes to generating optimized code for a resource-constrained microcontroller-based embedded system. This chapter will present some of the optimizations Faust performs and discuss the shortcoming of these optimizations when targeting microcontrollers. Some improvements will also be proposed to make the generated code better suited for a microcontroller.

3.1 A Simple Faust Example

Faust generates code in various languages including C, C++, Java, Web Assembly, LLVM IR, etc. In the following discussion, we will focus on the C++ version of the generated code. Almost all modern compilers targeting embedded systems compile target code from C and/or C++. Among the many programming languages, C and C++

are considered the de facto programming languages for modern embedded systems development[24, section 2.3.2].

By analyzing the code generated by Faust, the following optimizations are identified:

- Computing expressions that result in constant values throughout the execution of the program only once.
- Computing slow changing control variables once per audio rendering callback.
- Performing all remaining computations on a per sample basis in the audio rendering callback.

In the following sections we will look at a simple Faust example, a resonant low pass filter, to highlight Faust optimizations and analyze their impact on microcontroller-based embedded systems. We will also propose some improvements in order to achieve further optimization.

The Faust code in Code 3.1 represents processing a signal through a resonant low pass filter with constant arguments.

```
import("stdfaust.lib");

// Cutoff Frequency
ctFreq = 500;
// Q Factor
q = 5;
// Gain
gain = 1;

// Resonant Low Pass
process = fi.resonlp(ctFreq,q,gain);
```

Code 3.1: Faust resonant low pass filter with constant arguments.

With C++ set as the target language, the Faust compiler generates a single class called `mydsp` (default compiler option) with multiple methods[1]. Out of these methods, two are relevant to this discussion.

The first method, called `instanceConstant`, is where values that remain constant throughout the execution of the program are computed. For the case of the resonant low pass filter with constant arguments, the method is shown in Code 3.2.

```
virtual void instanceConstants(int samplingFreq) {
    fSamplingFreq = samplingFreq;
    fConst0 = tanf((1570.79639f / min(192000.0f, max(1000.0f, float(
    fSamplingFreq)))));
    fConst1 = (1.0f / fConst0);
    fConst2 = (1.0f / (((fConst1 + 0.200000003f) / fConst0) + 1.0f));
    fConst3 = (((fConst1 + -0.200000003f) / fConst0) + 1.0f);
    fConst4 = (2.0f * (1.0f - (1.0f / mydsp_faustpower2_f(fConst0))))
    ;
}
```

Code 3.2: Faust generated `instanceConstant` method for a resonant low pass filter with constant arguments.

The second method is `compute`. This method is responsible for computing the audio samples to fill the audio buffer. The constant values computed in the first method are used in this one to compute the audio samples. For the case of the resonant low pass filter with constant arguments the `compute` method is shown in Code 3.3.

In the case where the arguments of the resonant low pass filter are constant, the `compute` method is efficient and optimized for a microcontroller. Only computations necessary to compute the audio samples are included in it. When the arguments of the resonant low pass filter are replaced with variables, the generated `compute` function is no longer efficient or optimized.

[1]The generated code in its entirety is available in Appendix A

```
virtual void compute(int count, FAUSTFLOAT** inputs, FAUSTFLOAT**
outputs) {
    FAUSTFLOAT* input0 = inputs[0];
    FAUSTFLOAT* output0 = outputs[0];
    for (int i = 0; (i < count); i = (i + 1)) {
        fRec0[0] = (float(input0[i]) - (fConst2 * ((fConst3 * fRec0
        [2]) + (fConst4 * fRec0[1]))));
        output0[i] = FAUSTFLOAT((fConst2 * (fRec0[2] + (fRec0[0] +
        (2.0f * fRec0[1])))));
        fRec0[2] = fRec0[1];
        fRec0[1] = fRec0[0];
    }
}
```

Code 3.3: Faust generated `compute` method for a resonant low pass filter with constant arguments.

Code 3.4 replaces the constant arguments in Code 3.1 with variable arguments. These variable arguments are controlled by horizontal sliders appearing on a Graphical User Interface (GUI).

```
import("stdfaust.lib");

// Cutoff Frequency Horizontal Slider
ctfreq = hslider("cutoffFrequency",500,50,10000,0.01);
// Q Factor Horizontal Slider
q = hslider("q",5,1,30,0.1);
// Gain Horizontal Slider
gain = hslider("gain",1,0,1,0.01);

// Resonant Low Pass
process = fi.resonlp(ctFreq,q,gain);
```

Code 3.4: Faust resonant low pass filter with variable arguments.

The `instanceConstant` and `compute` methods generated after replacing the constant arguments with variable ones are shown in Code 3.5.

As a result of these changes, variables that are evaluated in the `compute` method can be divided into two sets. The first set of variables are those designated by `fSlow`. Every time `compute` is called, these variables get evaluated only once before the code appearing in the for-loop is evaluated. The second set of variables are those evaluated inside the for-loop. For the rest of this discussion we will refer to the first set of variables as control variables and the second set as audio variables.

Audio variables are associated with computing the audio samples, while control variables are associated with the arguments passed to the resonant low pass filter in the Faust code. Generally, control signals in Faust relate to GUI elements, MIDI messages, OSC messages, or physical sensors.

```
virtual void instanceConstants(int samplingFreq) {
    fSamplingFreq = samplingFreq;
    fConst0 = (3.14159274f / min(192000.0f, max(1000.0f, float(
    fSamplingFreq))));
}
virtual void compute(int count, FAUSTFLOAT** inputs, FAUSTFLOAT**
outputs) {
    FAUSTFLOAT* input0 = inputs[0];
    FAUSTFLOAT* output0 = outputs[0];
    float fSlow0 = (1.0f / float(fHslider1));
    float fSlow1 = tanf((fConst0 * float(fHslider2)));
    float fSlow2 = (1.0f / fSlow1);
    float fSlow3 = (((fSlow0 + fSlow2) / fSlow1) + 1.0f);
    float fSlow4 = (float(fHslider0) / fSlow3);
    float fSlow5 = (1.0f / fSlow3);
    float fSlow6 = (((fSlow2 - fSlow0) / fSlow1) + 1.0f);
    float fSlow7 = (2.0f * (1.0f - (1.0f / mydsp_faustpower2_f(fSlow1
    ))));
    for (int i = 0; (i < count); i = (i + 1)) {
        fRec0[0] = (float(input0[i]) - (fSlow5 * ((fSlow6 * fRec0[2])
         + (fSlow7 * fRec0[1]))));
        output0[i] = FAUSTFLOAT((fSlow4 * (fRec0[2] + (fRec0[0] +
        (2.0f * fRec0[1])))));
        fRec0[2] = fRec0[1];
        fRec0[1] = fRec0[0];
    }
}
```

Code 3.5: Faust generated `instanceConstant` and `compute` methods for a resonant low pass filter with variable arguments.

3.2 Computing Constants

When targeting microcontrollers, one of the optimization goals is keeping the size of the compiled executable binary file small. The executable file is usually loaded and stored in flash memory[2]. Microcontrollers have limited onboard flash memory and this limitation should be taken into consideration.

In Code 3.5, the only argument passed to the `instanceConstant` method is the sampling rate. Some of the constants are computed based on this sampling rate. This approach is generally ideal, since the class generated by Faust gets incorporated into a platform specific target application where the sampling rate is usually passed at runtime. An audio plugin used in a Digital Audio Workstation (DAW) is one example, where the sampling rate needs to match that of the DAW when the plugin is instantiated.

However, if the sampling rate is predetermined at compile time, computing the constant values during code generation would result in a smaller binary file and faster startup time.

The code generation in Faust could be tailored to such use cases by adding a compiler option and passing the sampling rate at compile time, thus making the compiler generate more suitable code for a microcontroller-based embedded system.

[2]A solid-state non-volatile computer storage medium that can be electrically erased and reprogrammed.

3.3 Processing Loads and Relative Rates

Minimizing the amount of computations on a microcontroller is another optimization goal. Unnecessary computations result in additional power consumption and have a direct effect on the responsiveness of a microcontroller-based embedded system. Setting the size of the audio buffer to a single sample to achieve glitch-free real-time performance (single sample latency) is possible when running on a bare metal microcontroller. However, to realize this the audio rendering callback needs to run as efficiently as possible.

In Faust the relative computation time spent on computing control variables and audio variables in the `compute` method is dependent on the buffer size of the audio rendering callback.

As the buffer size of the audio rendering callback increases, the ratio of CPU cycles required to compute control variables to those required to computing audio variables decreases. Regardless of the audio buffer size, the amount of computation required to compute the control variables stays constant, while the amount required to compute the audio variables proportionally increases with the size of the audio buffer.

For the resonant low pass filter with variable arguments this relationship is shown in Figure 3.1. If we only consider the arithmetic and trigonometric operations in the `compute` method, 44.5 CPU cycles are needed to compute the control variables per audio rendering callback and 9 CPU cycles to compute each audio sample[3].

For large buffer sizes this is not an issue. However, as the buffer size decreases, the

[3]The CPU cycles are based on an analysis in Appendix B.

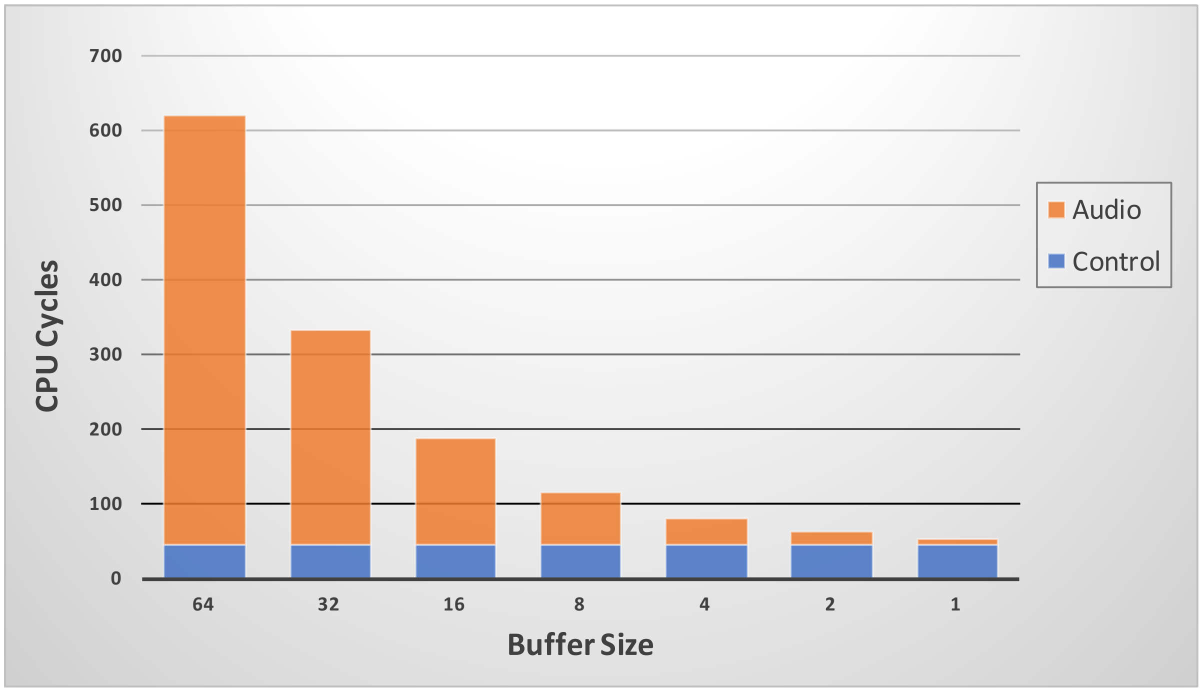


Figure 3.1: CPU cycles required per audio rendering callback for various buffer sizes.

effort spent on computing control variables per audio rendering callback becomes significant. With the audio buffer size set to 64 samples, 7.17% of the CPU cycles required to render the audio samples in the audio buffer are for computing the control variables. If the buffer size is reduced to a single sample, 83.18% of the CPU cycles are for computing the control variables. Reducing the buffer size results in a significant reduction in the computation efficiency, given the control signals will not change at the audio sampling rate. The reduction in computation efficiency is shown in Figure 3.2.

Spending 83.18% of computation time per audio rendering callback to compute control variables that might never change or change at a very slow rate relative to the audio sampling rate is far from efficient.

There are multiple improvements that can be made to achieve an efficient audio ren-

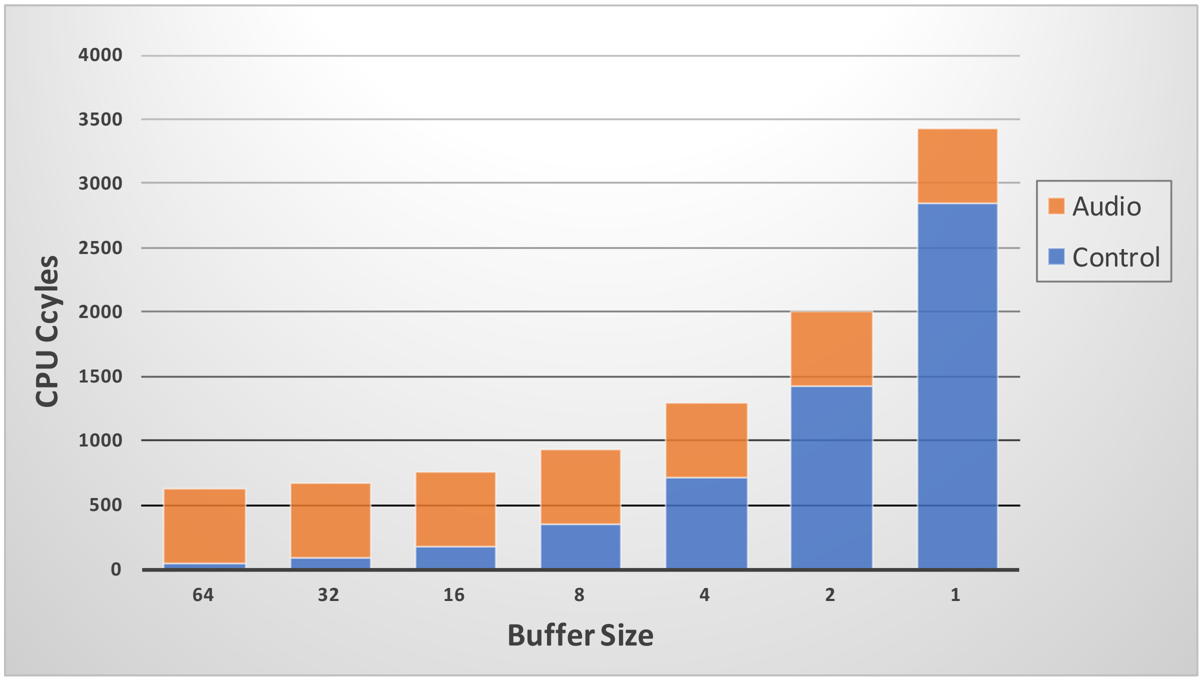


Figure 3.2: CPU cycles required to process 64 audio samples per audio rendering callback for various buffer sizes.

dering callback for the case of the resonant low pass filter with variable arguments. The first would be by adding a simple comparison to check if any control variable changed from the previous callback.

Figure 3.3 shows the CPU cycles required for the resonant low pass filter with variable arguments when the audio buffer size is set to 16 samples and Figure 3.4 shows the impact of adding a comparison check on the control variables in the audio rendering callback to the CPU cycles.

A further improvement could be made by computing the control variables on a thread different from the one where the audio rendering callback executes. This would result in a very efficient audio rendering callback where only audio variables are computed. The thread responsible for computing the control variables would have a lower priority and could be set to run at a lower rate than the thread responsible for the audio rendering callback.

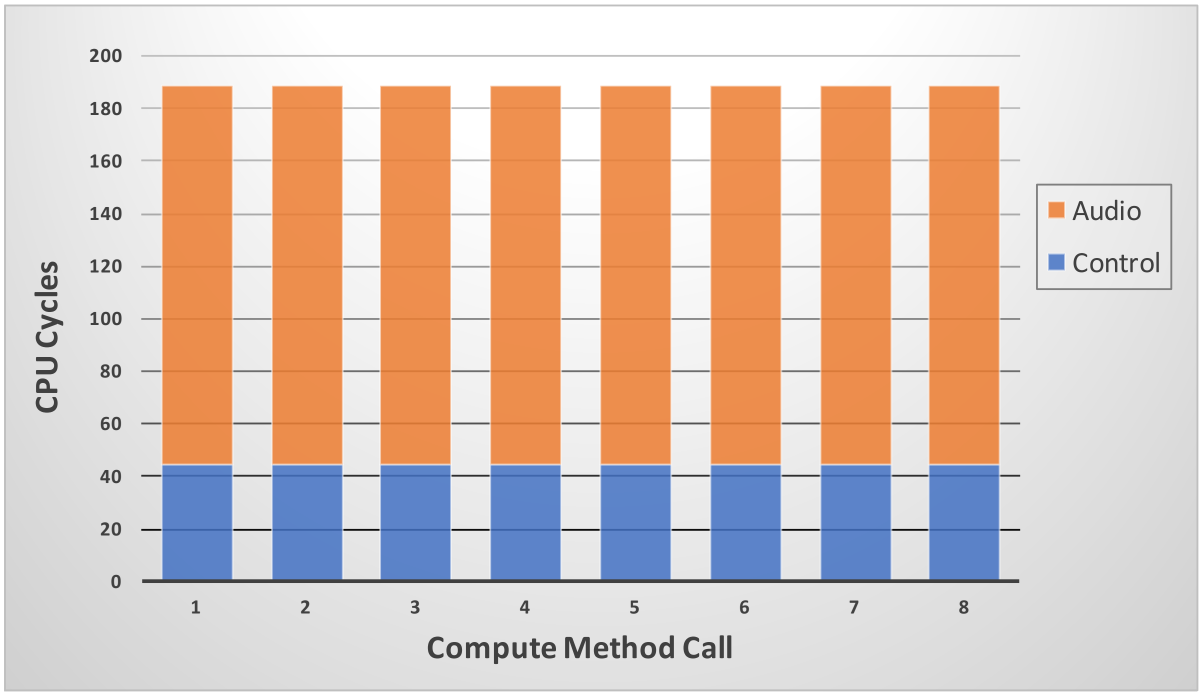


Figure 3.3: CPU cycles required per audio rendering callback for a 16-sample buffer size.

A potential thread profile is shown in Figure 3.5 with 175 CPU cycles available to the processor relative to the rate of the audio rendering callback[4]. The control thread rate is set to half of the audio thread. The chart shows how a control variable change would affect the system and when it would affect the audio samples.

The two-thread approach will not only affect the CPU cycles and performance but also the relative update rate between control and audio variables. The relative rate between processing control variables and audio variables in Faust is fixed and dependent on the audio buffer size. Moreover, both types of variables in Faust are computed within the same method making them synchronous. The relative rate and

[4]CPU cycles required for context switching are ignored.

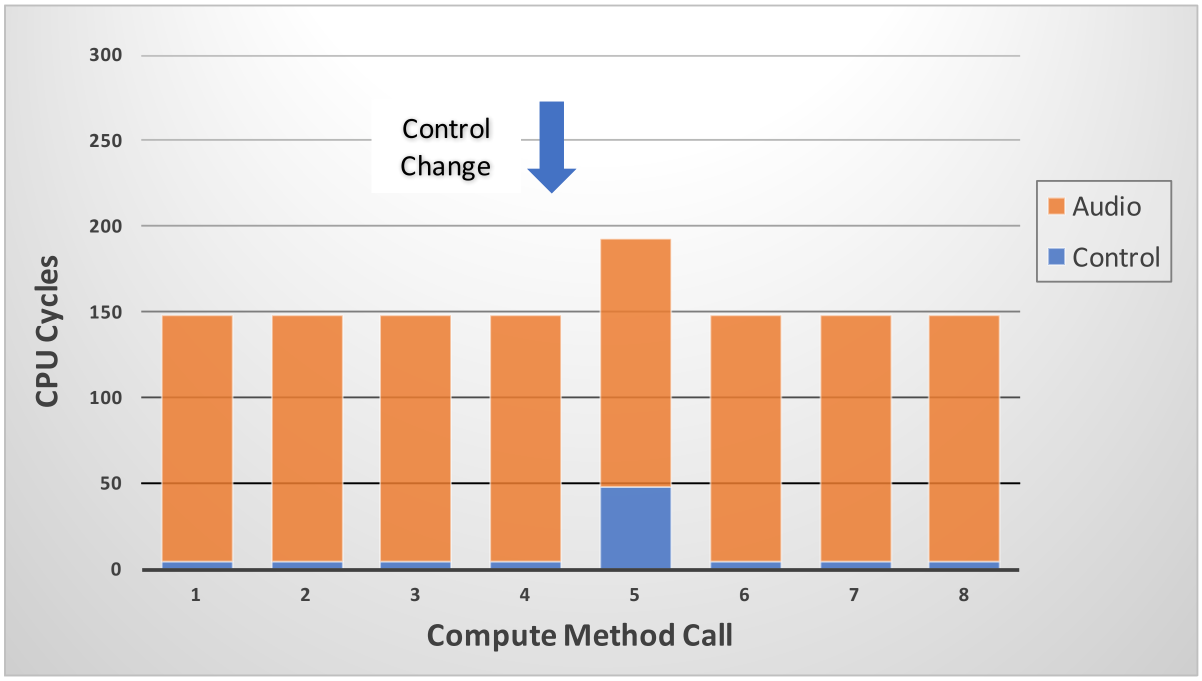


Figure 3.4: CPU cycles required per audio rendering callback for a 16 sample buffer size with a control variable change check.

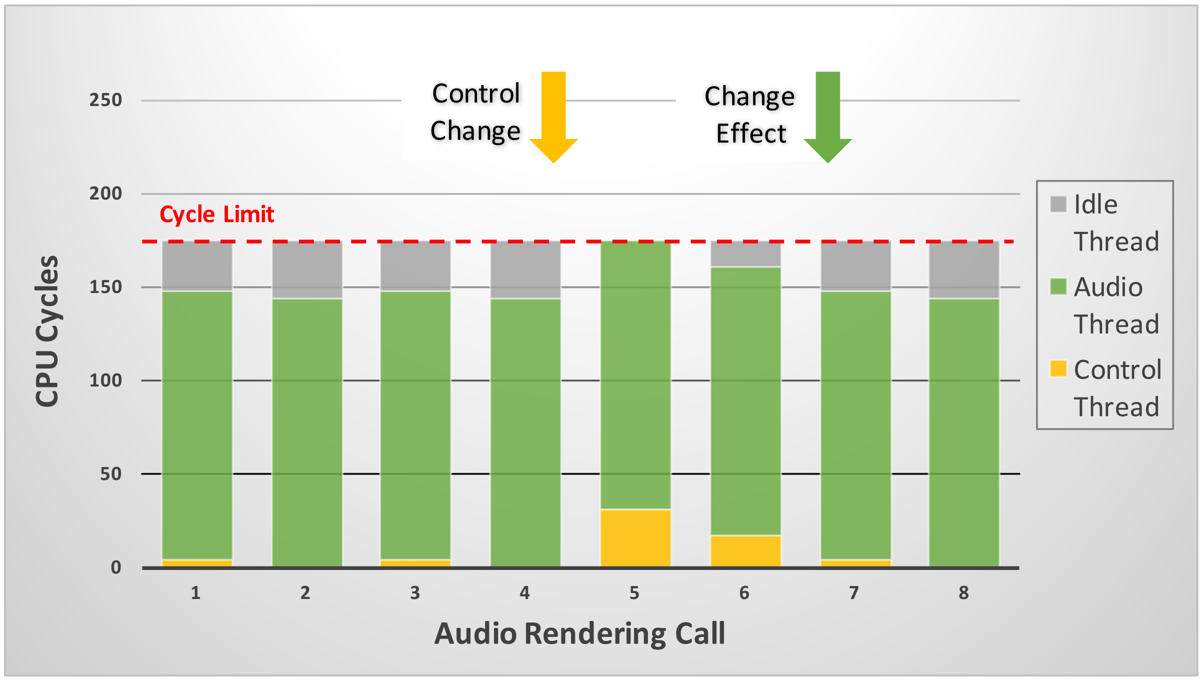


Figure 3.5: CPU thread profile showing the impact of a control change and its effect.

synchronicity affect the output generated by Faust and tie it directly to the size of the audio buffer. With a two-thread implementation, the control and audio variables become asynchronous and the relative rate between them becomes independent of buffer size.

To realize this two-thread approach, significant changes need to be made to the Faust code generation engine. Even if the code generation engine is modified to accommodate this approach, the relationship between various signals will still be target specific and dictated by the specificities of the implementation of the target. Hence, the user will be constrained by the implementation. Giving the user the ability to decide how often and in which thread variables are computed enables them to optimize and tune a system to their need.

3.4 Concurrency

Modern microcontrollers can run complex tasks simultaneously in real-time. Managing concurrency plays an important part in achieving the required real-time performance. Thus, giving the user control over the concurrency scheme is crucial.

The Faust framework does not have a concurrency model built into it. Updating control values and computing audio samples based on control value changes happen sequentially, thus eliminating the need for a concurrency model in the Faust framework itself.

All controls (GUI, MIDI, hardware, etc.) of a target platform are mapped to Faust widgets that are updated on every audio callback. This update happens prior to computing the audio samples and involves the sampling and updating of every control value. This approach moves the necessity of having a concurrency model in the Faust framework to having one in the target platform's software. Instead of defining a concurrency model, target platforms often rely on the atomic data types supported by their processor. When sampled, all control values are stored as `FAUSTFLOAT`, a type definition (typedef) for a floating-point data type in the Faust framework.

To demonstrate this interaction between the Faust framework and a target platform we will consider the Bela platform[5]. Excerpts of the `bela.cpp` target platform definition file[6] are shown in Code 3.6.

The Faust framework expects a target platform definition file to include and implement two methods: `setup()` and `render()`, among other classes and methods.

In the `setup()` method (lines 31 to 48), after allocating memory for the audio buffers, an instance of the DSP object is created. The DSP object is then linked to the user interface of the target platform, where Faust widgets are mapped to controls. By establishing this link, the DSP instance gains access to the sampled control values through widgets.

In the `render()` method (lines 50 to 58), prior to calling the `compute()` method where the audio samples are calculated (as discussed in section 3.1), the `update()` method is called on the user interface instance to read and/or write all the controls and synchronize them. The update occurs through Faust widgets.

[5]`https://bela.io/` [accessed November 7, 2018]
[6]The platform definition file in its entirety is available in Appendix A

The `update()` method of the user interface in turn calls the `BelaWidget.update()` method defined as part of a widget class of the target platform (lines 2 to 17) for every single control that is being utilized. For example, if the 8th analog input on the Bela board is used as a control input, the `kANALOG_7` (line 10) case is invoked where the `analogReadNI()` method (line 11) is called to fetch the value from the corresponding Analog to Digital Converter (ADC) input. The `analogReadNI()` method is implemented as part of the Bela platform and not Faust. If the Bela platform implements a concurrency model it would appear in the `analogReadNI()` method.

```
 1 // The widget class where the update method is impelmeneted
 2 class BelaWidget
 3 {
 4   ...
 5   public:
 6     ...
 7     void update(BelaContext *context) {
 8       switch (fBelaPin) {
 9         ...
10         case kANALOG_7:
11           *fZone = fMin + fRange * analogReadNI(context, 0, (int)
           fBelaPin);
12         break;
13         ...
14       }
15     }
16   ...
17 };
18
19 ...
20 // Array of pointers to context->audioIn data
21 FAUSTFLOAT **gFaustIns;
22 // Array of pointers to context->audioOut data
23 FAUSTFLOAT **gFaustOuts;
24 ...
25 // Bela User Interface (Hardware)
26 BelaUI gControlUI;
27 // Pointer to a Faust DSP instance
28 dsp *gDSP = NULL;
29 ...
30
31 bool setup(BelaContext *context, void *userData) {
32   ...
33   // Allocate deinterleaved inputs
```

```
  gFaustIns = new FAUSTFLOAT *[context->audioInChannels];
  ...
  // Allocate deinterleaded output
  gFaustOuts = new FAUSTFLOAT *[context->audioOutChannels];
  ...
  // Faust DSP instance declaration
  gDSP = new  mydsp ();
  ...
  // Initializing the DSP instance
  gDSP->init(context->audioSampleRate);
  // Mapping Bela Analog/Digital IO and Faust widgets
  gDSP->buildUserInterface(&gControlUI);
  ...
  return true;
}

void render(BelaContext *context, void *userData) {
  ...
  // reads Bela pins and updates corresponding Faust Widgets zones
  gControlUI.update(context);
  // synchronize all GUI controllers
  GUI::updateAllGuis();
  // process Faust DSP
  gDSP->compute(context->audioFrames, gFaustIns, gFaustOuts);
}
```

Code 3.6: Excerpts from the platform definition file for the Bela platform.

3.5 Vector Processing

Some microcontrollers support advanced instruction sets that are capable of operating on multiple data with a single instruction. They are known as Single Instruction Multiple Data (SIMD) instructions. Most compilers are capable of translating C++ code into machine code by utilizing these advanced instructions. However, special data types and code organization are required to trigger the compiler to use these instructions.

The original version of Faust (currently known as Fausto) has a compiler option that directs the code generator to generate C++ code suitable for vector operations. When the option is enabled, the code generator restructures the C++ code in a way to direct the C++ compiler to use SIMD instructions. An example of using this option is shown in [25], where the generated code performs better when compiled with an Intel ICC 11.0 compiler. However, the same code might not trigger the use of SIMD instructions when compiled with a different compiler.

A better approach is needed for the users to express their intent for vector processing and the use of SIMD instructions. The approach should not be tied to a particular compiler or compilers.

3.6 Libraries and APIs

Many microcontrollers come with dedicated libraries optimized to perform specific tasks. These libraries are optimized for performance and size. An example of such a library is the CMSIS DSP Software Library developed by ARM for the Cortex-M series[26]. The library contains a list of optimized signal processing functions. Having the ability to access these libraries is an advantage when it comes to generating efficient code. One way to access them would be through a Foreign Function Interface (FFI) designed into a language.

Faust does not have a FFI mechanism to access such libraries and does not offer a way to utilize these libraries during code generation.

Microcontroller manufacturers also provide APIs to access resources on a device and configure them. If a high-level code generation language like Faust lacks a FFI, modifying configurations or changing the state of resources would not be possible through Faust user code. Configurations and resource allocations will have to be hard coded in external files specific to each platform.

3.7 Summary

In this chapter we identified a few optimization schemes that could result in the generation of efficient code for resource-constrained microcontroller-based embedded systems. We demonstrated how CPU processing loads and relative rates between variables have a big impact on efficiency. Allowing users to control rates and the distribution of computations across multiple threads could result in drastic improvements on the real-time performance of a system by making it more efficient. Allowing users to control the concurrency model could yield similar improvements. Further optimizations could be achieved through building a FFI into the language to access optimized libraries as well as configure and manage device resources.

Chapter 4

Improvements with Stride

In the previous chapter we proposed some code optimization strategies to target resource-constrained microcontroller-based embedded system. One of the strategies was to control the relative rates at which computations are performed and to distribute computations across multiple threads. In this chapter we present a new language and its syntax. This new language enables the user to control the code generation process to realize this strategy. We will demonstrate this with a simple example.

4.1 An Oscillator with Frequency Control

A sine oscillator with frequency control is a basic unit generator. In this section we will examine a simple implementation of this unit generator.

A simple sine oscillator with frequency control can be implemented by tracking its phase over time. The output of the oscillator is the trigonometric sine of the phase. The phase is incremented by a phase increment after computing each output. The phase increment is calculated based on the desired frequency and sampling rate. The phase is wrapped when its values is equal to or greater than two pi. This simple sine oscillator implementation is shown in Code 4.1 as a function in the C language.

Four expressions (lines 7, 10, 13, and 16) are evaluated every time the `SinOsc` function is called. If the frequency of the oscillator does not change from one function call to the next, calculating the `PhaseIncrement` is not necessary. That is, out of the four expressions only the ones directly related to the `Phase` need to be evaluated to compute the next oscillator output.

```
 1 #define M_PI  3.14159265359
 2
 3 void SinOsc(float &output, float frequency) {
 4     static float Phase, PhaseIncrement = 0.;
 5
 6     // Compute the phase increment relative to the frequency and
       sampling rate
 7     PhaseIncrement = 2 * M_PI * frequency / SamplingRate;
 8
 9     // Compute the sin of the phase as the output
10     output = sin(Phase);
11
12     // Increment the phase
13     Phase += PhaseIncrement;
14
15     // Wrap the phase if it is greater than two Pi
16     if (Phase >= 2 * M_PI) Phase -= 2 * M_PI;
17 }
```

Code 4.1: A simple sine oscillator with frequency control in C.

4.2 A New Language

Before attempting to generate code, we will introduce a new language and its syntax. This language is called Stride. Stride will enable the user to declare signals, invoke modules, and connect them to create a dataflow graph. Stride will also enable the user to control its code generator to optimize the generated code.

The new language has two constructs: Block Declarations and Stream Expressions. We will introduce the syntax of these constructs in the following subsections.

4.2.1 Block Declarations

The first construct of the new language is the block declaration. The syntax diagrams to construct declarations are shown in Figure 4.1 and Figure 4.2.

Every block declaration starts with a type and a name. A declaration encloses a set of assignable properties. Block declarations with different types have different properties. The syntax diagram in Figure 4.1 is for a block declaration. Its corresponding grammar is:

```
Block ::= type Name ‘{’(property ‘:’ Expression ‘;’?)*‘}’
```

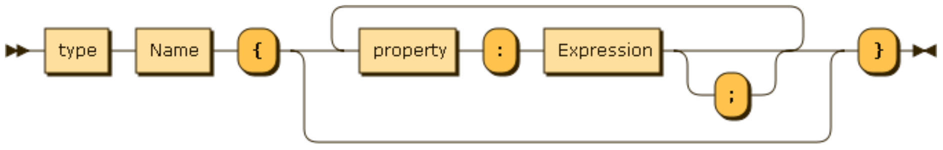


Figure 4.1: Block declaration syntax diagram.

Code 4.2 is an example block declaration. The block is of type `signal` and is called `Block`. The `signal` block has four properties called `default`, `rate`, `domain`, and `meta`. They are assigned the values `0.0`, `AudioRate`, `AudioDomain`, and `"A signal block"` respectively.

```
signal Block {
    default:    0.0                 # Default value
    rate:       AudioRate           # The signal's rate
    domain:     AudioDomain         # The signal's domain
    meta:       "A signal block"    # Meta information
}
```

Code 4.2: A block declaration of type `signal` called `Block`.

The syntax in Figure 4.2 is a block bundle declaration. Blocks in a bundle share the same type and property assignments. The grammar for a block bundle declaration is:

```
Bundle ::= type Name '['Size']' '{'(property ':' Expression ';'?)*'}'
```

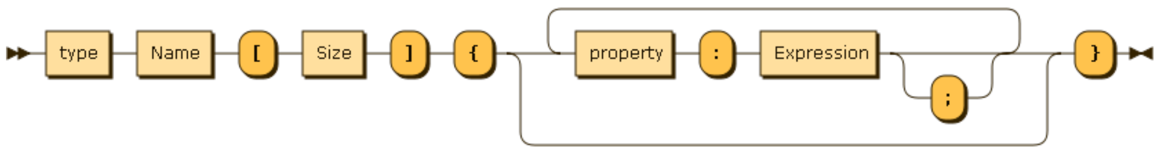


Figure 4.2: Bundle declaration syntax diagram.

Code 4.3 is an example block bundle declaration. The bundle is of type `signal` and it is called `Bundle`. The bundle is composed of two signal blocks.

A block in a bundle can be accessed through indexing. The first block in the bundle is accessed by `Bundle[1]` and the second by `Bundle[2]`.

```
signal Bundle [2] {
    default:    1.0                 # Default value
    rate:       AudioRate           # The signal's rate
    domain:     AudioDomain         # The signal's domain
    meta:       "A signal bundle"   # Meta information
}
```

Code 4.3: A bundle declaration of type `signal` and size 2 called `Bundle`.

4.2.2 Stream Expressions

The second construct of the new language is the stream expression. The syntax diagram to construct a stream expression is shown in Figure 4.3.

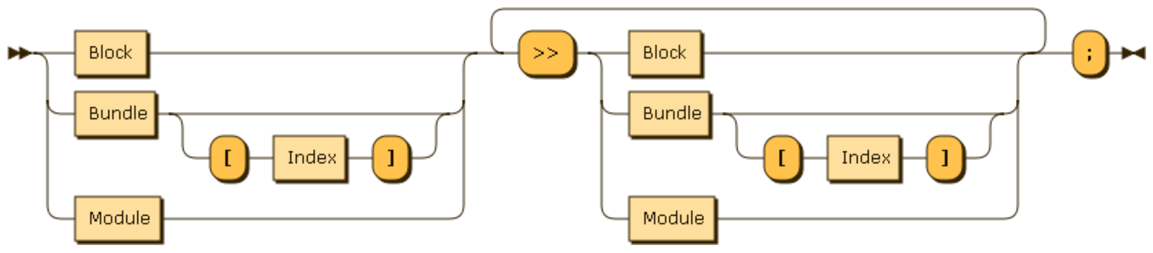


Figure 4.3: Stream expression syntax diagram.

A stream expression is constructed by connecting blocks, bundles and/or modules[1] using the stream operator >>. The grammar for a stream expression is:

```
StreamExpression ::= ( Block | Bundle | Bundle '['Index']' | Module )
        ( '>>' ( Block | Bundle | Bundle '['Index']' | Module ) )+ ';'
```

A module encapsulates blocks, bundles, and stream expressions to perform specific operations. The syntax diagram to invoke a module in a stream expression is shown in Figure 4.4 and its corresponding grammar is:

[1]Modules will be covered in detail in the following chapter.

```
Module ::= Name '(' ( port ':' Expression ';'? )* ')'
```

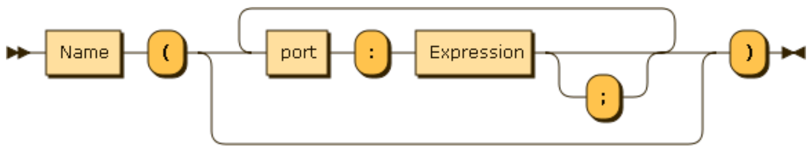


Figure 4.4: Module invocation syntax diagram.

Code 4.4 is an example stream expression. The `Input` signal is connected to the main input port[2] of the `Process` module. The main output port of the `Process` module is connected to the `Output` signal. The `Process` module has a single property port called `property`. The property port is connected to a signal called `Control`.

```
signal Input {}
signal Output {}
signal Control {}

Input >> Process ( property: Control ) >> Output;
```

Code 4.4: A stream expression.

4.3 Code Generation for an Embedded Platform

Now that we have defined a new language and its syntax, we will use it to target a microcontroller-based embedded system. We will deploy a sine oscillator with frequency control on the target platform and generate efficient and optimized code for it.

[2]Ports will be covered in detail in the following chapter.

Let us imagine an audio development board with a microcontroller as its main processor. The microcontroller is coupled with an audio codec[3] with a mono output. A rotary potentiometer is connected to one of the microcontroller's ADC pins. We will refer to this audio development board as the platform.

The goal is to generate code for the platform to play a sine wave though the mono audio output while controlling the wave's frequency through the potentiometer.

Let us assume we have a code generator that could add code to a pre-existing template. The template contains configuration code for the platform and presents the code generator with two functions assigned to hardware triggered callbacks. The code generator can insert code into these two functions. The first function is called `audioTick`. The audio output will be computed in the `audioTick` function. The second function is called `controlCallback`. In this function the potentiometer's value will be captured. To simplify the analysis, let us assume `audioTick` and `controlCallback` will be called at the same rate and `audioTick` in running on a thread that has a higher preemption priority over the thread where `controlCallback` is running.

4.3.1 Oscillator with Frequency Control

Using the new language and its syntax, we declare and define the setup we are trying to realize on the platform. The code is shown in Code 4.5.

The code consists of a block declaration (lines 1-5) and two stream expressions (lines 7-12 and 14-18).

[3]A device or computer program for encoding or decoding a digital data stream or signal.

The declaration is of type `signal` and is called `Frequency`. Three of the `signal`'s properties are shown in the code. The first property, `default`, sets the initial value of the `signal`. The second property, `rate`, sets the rate of the `signal` and is set to `AudioRate`. The third property, `domain`, sets the domain of the `signal` and is set to `AudioDomain`.

A domain abstracts a function. On this platform, `AudioDomain` abstracts the `audioTick` function. The `audioTick` function has a fixed callback rate, equal to the audio sampling rate. `AudioRate` abstracts this rate.

```
signal Frequency {
    default:    440.0
    rate:       AudioRate
    domain:     AudioDomain
}

ControlIn[1]
>> Map (
    minimum:    55.0
    maximum:    880.0
)
>> Frequency;

Oscillator (
    type:       "Sine"
    frequency:  Frequency
)
>> AudioOut;
```

Code 4.5: Stride code to control the frequency of a sine oscillator.

In the first stream expression, `ControlIn[1]` is connected to the main input port of a mapping module called `Map`. The main output port of the module is connected to the `Frequency` signal.

On this platform, `ControlIn[1]` is a signal block. It abstracts the first ADC channel of the microcontroller. The rate and the domain of `ControlIn[1]` are `ControlRate` and `ControlDomain` respectively. `ControlDomain` abstracts the `controlCallback` function.

The rate of the callback is abstracted by `ControlRate`. In this case, `ControlIn[1]` represents the value of the potentiometer normalized to $[0.0,\ 1.0]$. The `Map` module maps `ControlIn[1]` to values between the minimum and maximum values assigned to the module's properties.

In the second stream expression, the main output of a module called `Oscillator` is connected to `AudioOut`. The `type` property of the `Oscillator` is set to ``Sine'' and the `frequency` property is connected to the `Frequency` signal. `AudioOut` is a `signal` which abstracts the mono audio output on the platform. The rate and the domain of `AudioOut` are `AudioRate` and `AudioDomain` respectively.

The `ControlIn` signal block bundle and the `AudioOut` signal block are defined by the platform. Their corresponding rates (`ControlRate` and `AudioRate`) and domains (`ControlDomain` and `AudioDomain`) are also declared by the platform.

Based on the declaration and the two stream expressions in Code 4.5 we expect the code generator to generate code like the one shown in Code 4.6.

When we run the code on the platform, we expect the CPU cycles required by the `audioTick` and `controlCallback` function calls to look like Figure 4.5. Over four function calls the `audioTick` function requires 106 CPU cycles and the `controlCallback` requires 4 CPU cycles.[4]

These results will serve as a baseline. Improvements in efficiency will be measured and compared to this baseline as different optimization schemes are presented and evaluated.

[4]CPU cycles are computed based on the analysis presented in Appendix B.

```
AtomicFloat ControlValue = 0.;

void controlCallback (float *input, int size){
    ControlValue = input[0];
}

void audioTick (float &output){
    static float Phase, Frequency, PhaseIncrement = 0.;

    Frequency = map(ControlValue, 55., 880.);
    PhaseIncrement = 2 * M_PI * Frequency / AudioRate;

    output = sin(Phase);
    Phase += PhaseIncrement;

    if (Phase >= 2 * M_PI) Phase -= 2 * M_PI;
}
```

Code 4.6: Generated code for controlling the frequency of an oscillator.

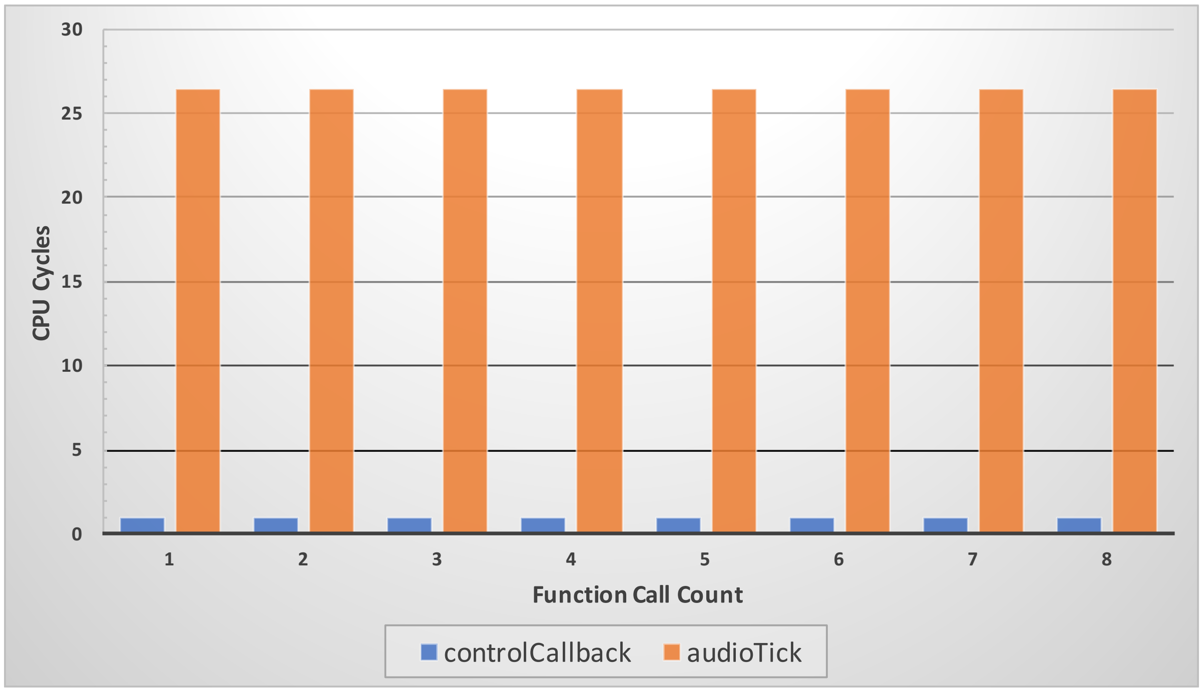


Figure 4.5: CPU cycles required per function call. (Baseline)

4.3.2 Oscillator's Frequency Control at Reduced Rate

The first attempt to reduce the CPU cycles required to compute the `audioTick` and `controlCallback` functions over multiple cycles would be to reduce the update rate of the `Frequency` signal. We can realize this by changing the `rate` property of `Frequency` from `AudioRate` to `AudioRate / 4.0`. This change is shown on line 3 of Code 4.7. This change will reduce the update rate of the `Frequency` signal by 4 times.

With the reduced update rate of the `Frequency` signal, we would expect the code generator to produce code that looks like Code 4.8. Due to the change in rate, an accumulator is added to the generated code. The accumulator increments on every `audioTick` function call. The `PhaseIncrement` is calculated only when the accumulator saturates.

```
 1 signal Frequency {
 2     default:    440.0
 3     rate:       AudioRate / 4.0           # Rate Change
 4     domain:     AudioDomain
 5 }
 6
 7 ControlIn[1]
 8 >> Map (
 9     minimum:    55.0
10     maximum:    880.0
11 )
12 >> Frequency;
13
14 Oscillator (
15     type:       "Sine"
16     frequency:  Frequency
17 )
18 >> AudioOut;
```

Code 4.7: Controlling the frequency of an oscillator at reduced rate.

When we run the code on the platform, we expect the CPU cycles required by the audioTick and controlCallback function calls to look like Figure 4.6. Over four function calls the audioTick function requires 90 CPU cycles and the controlCallback requires 4 CPU cycles. With this change in rate, we have achieved a 15% reduction in CPU cycles.

```
AtomicFloat ControlValue = 0.;
Accumulator compute(4./AudioRate);

void controlCallback (float *input, int size){
    ControlValue = input[0];
}

void audioTick (float &output){
    static float Phase, Frequency, PhaseIncrement = 0.;

    if (compute()){
        Frequency = map(ControlValue, 55., 880.);
        PhaseIncrement = 2 * M_PI * Frequency / AudioRate;
    }

    output = sin(Phase);
    Phase += PhaseIncrement;

    if (Phase >= 2 * M_PI) Phase -= 2 * M_PI;
}
```

Code 4.8: Generated code for controlling the frequency of an oscillator at reduced rate.

In spite of the reduction in rate, the phase increment of the oscillator is still being computed synchronously with the audio samples. Even when the frequency of the oscillator remains constant (that is, the potentiometer has not been rotated), we are still computing a phase increment but only at a reduced rate. An asynchronous and reactive computation of the phase increment can further improve the performance.

Although the rate reduction improved the performance of the process, we note the increase in the size of the memory footprint need by the program because of the additional accumulator.

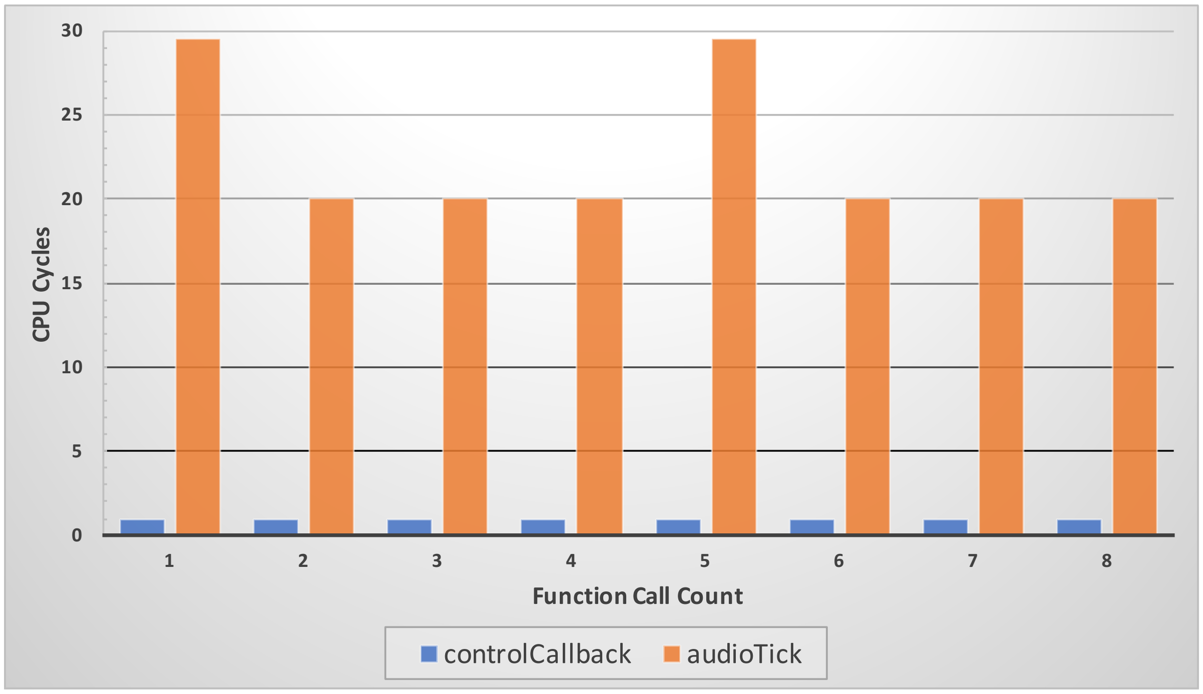


Figure 4.6: CPU cycles required per function call at reduced update rate.

4.3.3 Reactive Control of the Oscillator's Frequency

To achieve reactive control, we need to introduce the following changes to the original code (Code 4.5). First, the `rate` property of `Frequency` should be set to 0 to make the `signal` operate in reactive mode. Second, an `OnChange` module should be introduced in the first stream expression to force the data to flow asynchronously. The changes are shown on lines 3 and 8 in Code 4.9.

With these changes, we expect the code generator to introduce a comparison check between the pervious and current values read from the potentiometer. Only when the values are different the `PhaseIncrement` gets evaluated. The generated code would to look like Code 4.10 based on the changes.

```
signal Frequency {
    default:     440.0
    rate:        0            # Sets Frequency to Asynchronous mode
    domain:      AudioDomain
}

ControlIn[1]
>> OnChange()                 # Updates Frequency when input changes
>> Map (
    minimum:     55.0
    maximum:     880.0
)
>> Frequency;

Oscillator (
    type:        "Sine"
    frequency:   Frequency
)
>> AudioOut;
```

Code 4.9: Controlling the frequency of an oscillator reactively.

```
AtomicFloat ControlValue = 0.;

void controlCallback (float *input, int size){
    ControlValue = input[0];
}

void audioTick (float &output){
    static float Phase, Frequency, PhaseIncrement = 0.;
    static float PreviousValue = 0.0;

    if (ControlValue != PreviousValue){
        Frequency = map(ControlValue, 55., 880.);
        PhaseIncrement = 2 * M_PI * Frequency / AudioRate;
        PreviousValue = ControlValue;
    }

    output = sin(Phase);
    Phase += PhaseIncrement;

    if (Phase >= 2 * M_PI) Phase -= 2 * M_PI;
}
```

Code 4.10: Code generated for controlling the frequency of an oscillator reactively.

When we run the code on the platform, we expect the CPU cycles required by the `audioTick` and `controlCallback` function calls to look like Figure 4.7, where the potentiometer was rotated during the first, fifth, and seventh function calls.

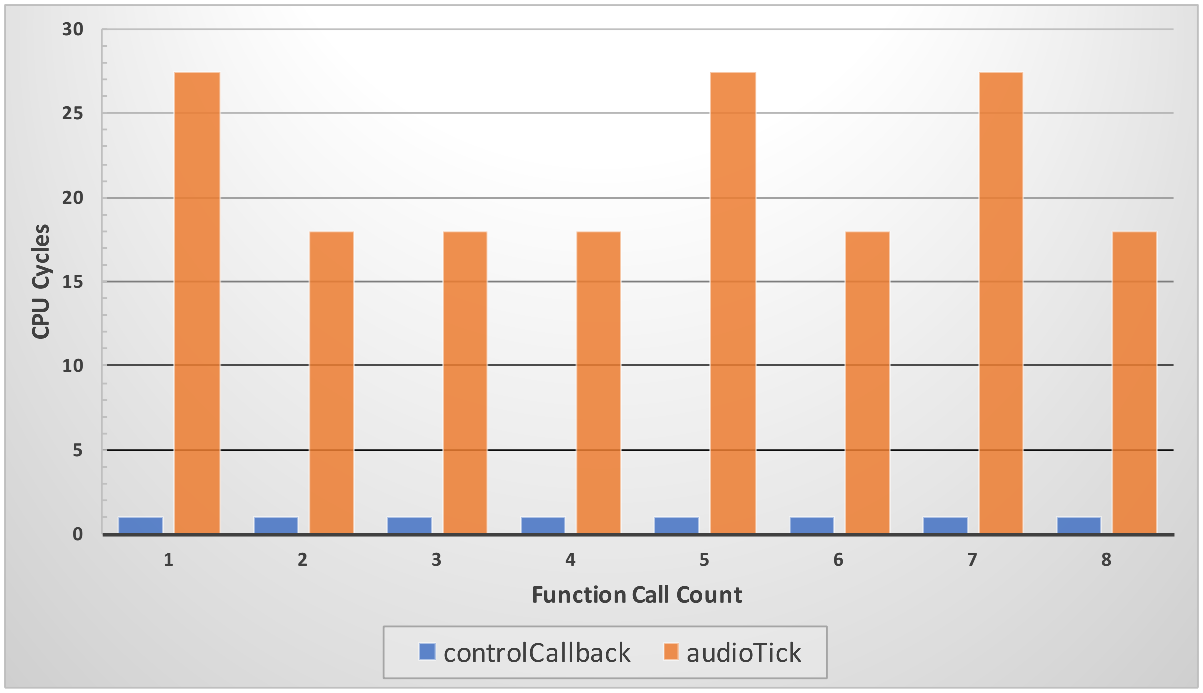


Figure 4.7: CPU cycles required per function call in asynchronous and reactive mode.

If the potentiometer is rotated once over four function calls, the `audioTick` function would require 82 CPU cycles and the `controlCallback` 4 CPU cycles. That is equivalent to a 22% reduction in CPU cycles from the baseline count.

If the potentiometer is not rotated over four function calls, the `audioTick` function would require 72 CPU cycles and the `controlCallback` 4 CPU cycles. That is equivalent to a 31% reduction in CPU cycles from the baseline count.

When it comes to the memory footprint of the program, this approach only adds a single variable to the original code. Unlike the previous case, the gain in performance outweighs the increase in the memory footprint.

4.3.4 Audio Callback Optimization

So far, all computations have happened in the `audioTick` function. This function can be further optimized by moving computations directly related to the frequency to the `controlCallback`. Computing the phase increment due to a change in frequency is one such computation.

To move the computation of the phase increment to the `controlCallback`, we change the `domain` of `Frequency` from `AudioDomain` to `ControlDomain`. This is shown on line 4 of Code 4.11.

```
 1 signal Frequency {
 2     default:     440.0
 3     rate:        0
 4     domain:      ControlDomain    # Domain change
 5 }
 6
 7 ControlIn[1]
 8 >> OnChange()
 9 >> Map (
10     minimum:     55.0
11     maximum:     880.0
12 )
13 >> Frequency;
14
15 Oscillator (
16     type:        "Sine"
17     frequency:   Frequency
18 )
19 >> AudioOut;
```

Code 4.11: Controlling the frequency of an oscillator with optimized audio callback.

The code generator will produce Code 4.12. The only expressions left in the `audioTick` function are ones responsible for computing the next audio sample.

When we run the code on the platform, we expect the CPU cycles required by the `audioTick` and `controlCallback` function calls to look like Figure 4.8, where the potentiometer was rotated during the first, fifth, and seventh function call counts.

```
AtomicFloat PhaseIncrement = 0.;

void controlCallback (float *input, int size){
    static float Frequency, PreviousValue = 0.;

    if (input[0] != PreviousValue){
        Frequency = map(input[0], 55., 880.);
        PhaseIncrement = 2 * M_PI * Frequency / AudioRate;
        PreviousValue = input[0];
    }
}

void audioTick (float &output){
    static float Phase = 0.;

    output = sin(Phase);
    Phase += PhaseIncrement;

    if (Phase >= 2 * M_PI) Phase -= 2 * M_PI;
}
```

Code 4.12: Generated code for controlling the frequency of an oscillator with optimized audio callback.

If the potentiometer is rotated once over four function calls, the `audioTick` function would require 68 CPU cycles and the `controlCallback` 15 CPU cycles. That is equivalent to a 25% reduction in CPU cycles from the baseline count.

If the potentiometer is not rotated over four function calls, the `audioTick` function would require 68 CPU cycles and the `controlCallback` 4 CPU cycles. That is equivalent to a 35% reduction in CPU cycles from the baseline count.

Although this change did not result in tangible performance improvement over the reactive case (subsection 4.3.1), this approach offers other benefits that are discussed in the following section.

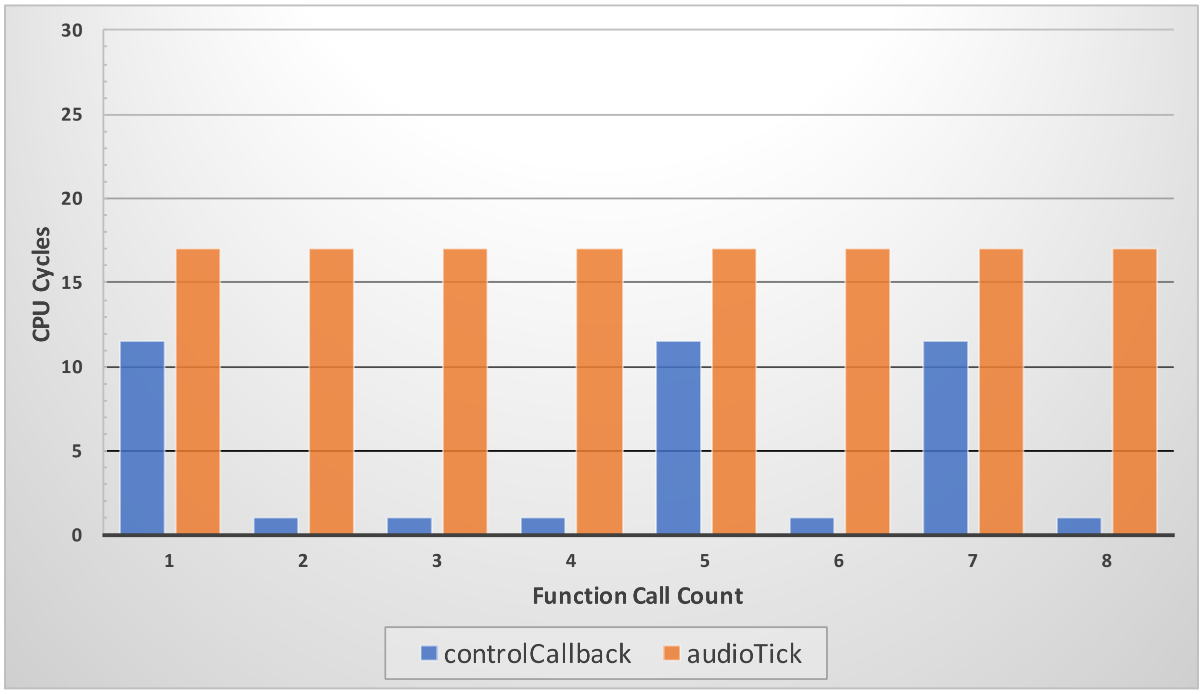


Figure 4.8: CPU cycles required per function call with an optimized `audioTick` function.

4.4 Discussion

In the previous section we presented a few user controlled optimization schemes to generate efficient code. With each scheme we improved the CPU cycles required to compute two callback functions. The improvements are summarized in Table 4.1.

Scheme	**Subsection**	**Potentiometer**	
		Change	**No Change**
Original	4.3.1	0%	0%
Rate Change	4.3.2	15%	15%
Reactive	4.3.3	22%	31%
Optimized	4.3.4	25%	35%

Table 4.1: Improvement in performance with code change.

In the last scheme, the audio callback was fully optimized. This was achieved by moving all expressions that are not directly associated with computing an audio sample out of the function. The excluded expressions were moved to another function where expressions directly related to external controls are captured and evaluated.

In the optimized audio callback scheme, the audio callback function requires the least CPU cycles. The cycle count remains relatively constant from one call to the next (except when the phase is to be wrapped). On a platform where the audio callback thread is assigned the highest preemption priority (ability to interrupt other threads), quick execution of the audio callback is extremely important in order to allow the interrupted threads to resume execution as soon as possible. This becomes even more critical when the buffer size of the audio callback is reduced down to a few samples or even to a single sample, where interruptions become more frequent.

Although we have demonstrated this optimization on the audio callback function, it could be applied to every other synchronous or asynchronous callback function running on the platform. Distributing computations to various threads executing at different rates based on the user's code is the primary objective of the language.

In these examples we worked with a multi-threaded system. However, we did not discuss a concurrency model between the threads. We assumed the variable shared between the control and audio threads was assigned an atomic type supported by the processor of the platform. The concurrency model built into Stride will be presented and discussed in detail in chapter 6.

4.5 Summary

In this chapter we introduced a new high-level language and its syntax. We used this language to realize a simple sine oscillator with frequency control on an audio development platform. Using some of the features of the language we controlled its code generator. We presented various schemes to generate efficient code and tracked the resulting improvements in efficiency.

Chapter 5

Signals, Rates, Domains, and Modules

In the previous chapter, we controlled a code generator through the rate and domain properties of a signal block to generate efficient code.

In this chapter, we will present how rate and domain information propagate in Stride code. We will demonstrate this by creating an oscillator module with frequency control in Stride.

However, before we can define and declare a module block, we first need to fully define the behavior of a signal block based on its rate and domain assignments.

5.1 Behavior of a Signal

The signal block (Code 4.2 is a core building block of Stride and is characterized by its versatile behavior. Simply put, a signal block represents an allocated memory address on a target platform. The allocated memory is initialized with the value assigned to the `default` property of the signal block.

In the following two subsections we will cover the `rate` and `domain` properties of the signal block and how they affect the allocated memory.

5.1.1 Rates

The behavior of a signal block changes depending on the value assigned to its `rate` property.

When the rate of a signal block is assigned a positive integer or real value, the signal block operates in sample-and-hold mode. That is, the signal block samples any block connected to its input at the specified rate, holds the sampled value in the allocated memory it represents, and issues a token with the sampled value to any block connected to its output. In this mode, the input of the signal block can accept a single connection. That is, a signal can sample-and-hold a single source.

When the rate of a signal block is set to zero, the signal block operates in reactive mode. That is, when a token arrives at its input port, the signal block updates the allocated memory it represents and forwards the token to any block connected to its output port. In this mode, the input of the signal block can accept multiple connections and will hold the value carried by the most recent token to arrive at its input port.

The two modes of operation of a signal block allow the user to either push data (reactive mode) or pull data (sample-and-hold mode).

In Code 5.1, three signal blocks with various rates are connected in a stream expression. Signal `A` is connected to `Random`, a random number generator module. Signal `A` samples the generator module at 2Hz. Signal `B` samples signal `A` at 1Hz and signal `C` samples signal `B` at 3Hz. A snapshot of possible values of signals `A`, `B`, and `C` are plotted in Figure 5.1. Since signals `A`, `B`, and `C` are assigned to the same domain (`ClockedDomain`), they are synchronous signals and are synchronized to the domain's clock. Domains and clocks are covered in the following subsection.

```
signal A { rate: 2 domain: ClockedDomain }
signal B { rate: 1 domain: ClockedDomain }
signal C { rate: 3 domain: ClockedDomain }

Random() >> A >> B >> C;
```

Code 5.1: Three signal blocks with various rates operating in sample-and-hold mode.

By changing the rate of `C` to `0`, as shown in Code 5.2, its mode of operation changes from sample-and-hold to reactive. Since `C` is operating in reactive mode, its value will be updated when the value of `B` changes, as shown in Figure 5.2. That is, `C` will be updated at the rate of `B`.

```
signal A { rate: 2 domain: ClockedDomain }
signal B { rate: 1 domain: ClockedDomain }
signal C { rate: 0 domain: ClockedDomain }

Random() >> A >> B >> C;
```

Code 5.2: Signal block `C` operating in reactive mode.

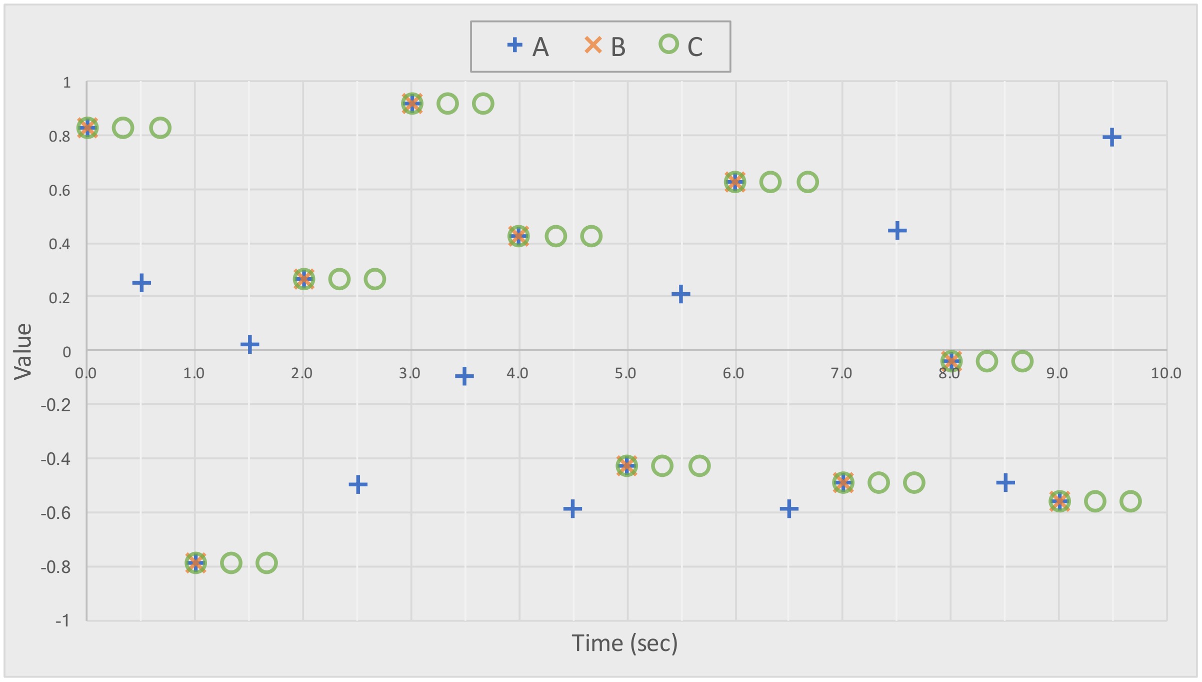


Figure 5.1: The values of three signal blocks with various rates operating in sample-and-hold mode.

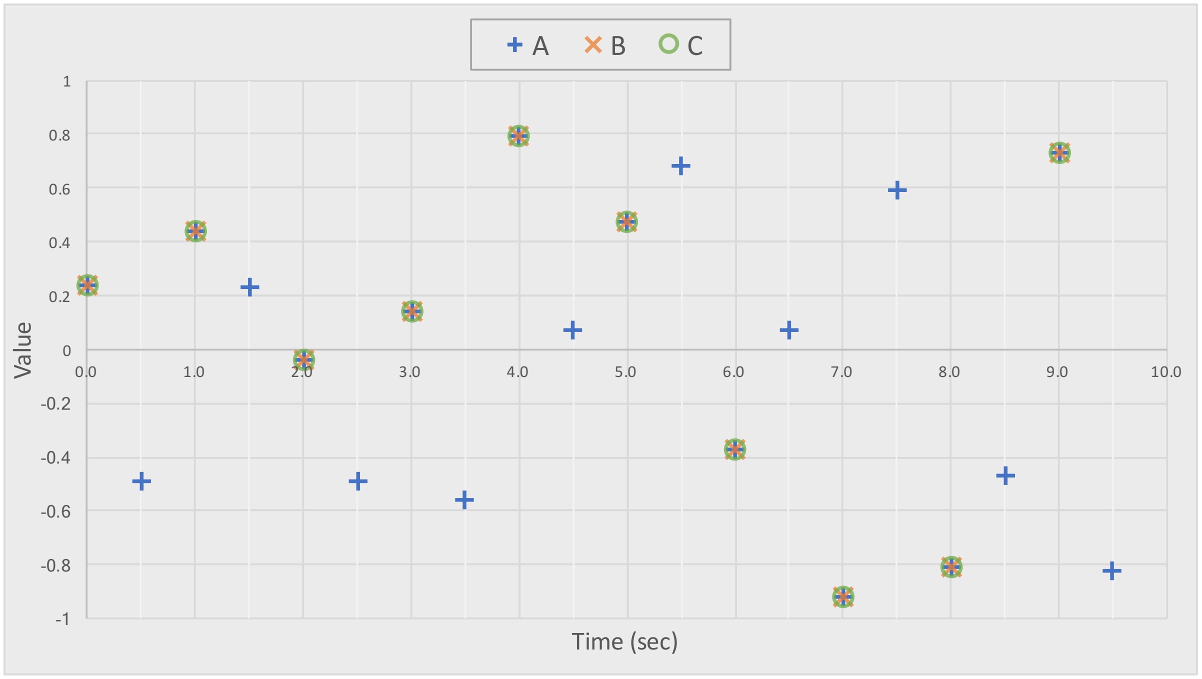


Figure 5.2: The values of three signal blocks, where signal block `C` is operating in reactive mode.

5.1.2 Domains

Domains abstract functions, methods, sub-routines, procedures, etc. on a target platform.

There are no restrictions imposed on the types of functions a domain can abstract. Domains can abstract functions that are called only once during execution, ones assigned to a thread, or ones attached to a system callback or an interrupt routine.

The domain assigned to the `domain` property of a signal block specifies the function where the signal gets evaluated.

Just like signal blocks, domains have a rate assigned to them. A domain whose rate is set to a positive integer[1] is called a clocked domain. A clocked domain derives its clock from a process clock or a hardware clock available on a platform. An example of a clocked domain is a domain abstracting an audio callback function, where the domain's clock is the audio sampling rate. A domain whose rate is set to zero is called an immediate domain. An immediate domain is not tied to any clock. An example of an immediate domain is a domain abstracting an initialization function. An initialization function usually executes once at the beginning of a program to reserve and configure system resources.

A signal block assigned to a clocked domain synchronizes itself to the domain's clock. The rate of a signal block assigned to an immediate domain serves only as a reference. An example would be a signal block assigned to a domain abstracting an initialization function, where the signal block is sampling a sine oscillator module to fill a lookup

[1] Unlike signal block, the rate of a domain cannot be set to a positive real number.

table.

In Code 5.3, signal `A` is assigned to a domain called `SetupDomain`. The rate of signal `A` is set to `SetupRate`. `SetupRate` is the rate of `SetupDomain`. Since signal `A` has the same rate as the domain it is assigned to, it gets evaluated once every time the function the domain abstracts is called.

```
signal A {
    default:    0
    rate:       SetupRate
    domain:     SetupDomain
}

3 + 4 >> A;
```

Code 5.3: Signal block `A` assigned to `SetupDomain`.

If we were to deploy the Stride code on a target platform by first translating it into C code, Code 5.3 would translate to Code 5.4.

If a target platform ran a python interpreter, Code 5.3 would translate to Code 5.5.

The mapping of domains to functions is relatively straight forward. However, complexities arise when domains are mapped to functions assigned to threads running concurrently and signals assigned to these domains are connected in stream expressions. The need for synchronization between these domains becomes a necessity and has to be accounted for when generating code to preserve the integrity of the signals. Stride has a flexible concurrency model built into it. The concurrency model is covered in detail in chapter 6.

For the remainder of this chapter, for the sake of simplicity, we will assume the processor of the target platform supports atomic operations on certain data types and all variables shared between threads have these data types.

```c
// Initialization
int A = 0;

// Definitions
void setup (void) {
    A = 3 + 4;
}

// Execution
int main (void) {
    setup();
    return 0;
}
```

Code 5.4: Generated C code from Code 5.3.

```python
# Initialization
A = 0

# Definitions
def setup ():
    global A
    A = 3 + 4

# Execution
setup()
```

Code 5.5: Generated Python code from Code 5.3.

5.2 A Sine Oscillator Module in Stride

Now that we have defined the behavior of a signal block based on its rate and domain assignments, we will demonstrate its behavior by designing a sine oscillator module with frequency control in Stride.

5.2.1 Sine Oscillator Stream Expressions

In section 4.1 we presented a basic sine oscillator in the C programming language. To compute an output sample of the oscillator four expressions required evaluation. Code 5.6 shows these four expressions in Stride.

```
# Compute the phase increment relative to the frequency and sampling rate
Frequency * 6.28318530718 / SamplingRate >> PhaseInc;

# Compute the sin of the phase as the output
Phase >> Sin () >> Output;

# Increment the phase
Phase + PhaseInc >> Phase;

# Wrap the  phase if it is greater than two Pi
[ Phase, 6.28318530718 ] >> GreaterOrEqual () >> WrapPhase ();
```

Code 5.6: Sine oscillator stream expressions in Stride.

If the frequency of the oscillator remains constant after the first expression has been evaluated, only the last three expressions need to be repeatedly evaluated to compute output samples. While the last three expressions need to be evaluated synchronously at the rate of the output, the first expression can be evaluated either synchronously or asynchronously and at any rate with respect to the last three expressions. The choice depends on the requirements set by the user. If the requirement is to change the frequency synchronously with the output, then all four expressions need to be synchronously evaluated regardless of the relative rate between the frequency change and the output. If the requirement is to change the frequency asynchronously with the output, then the first expression can be evaluated asynchronously at any desired rate.

If code were to be generated for these four expressions to meet either of these two requirements, a single callback function or to two callback functions are required on a target platform. A single function is required for the synchronous case, while two functions are required for the asynchronous case.

Through the domain assignments of the signal blocks in these four expressions, the user could control the mapping of these expressions to one or two callback functions. Through the rate assignments of the signal blocks, the user could control the relative rate of evaluation of these expressions.

The four expressions in Code 5.6 are composed of the following block types:

Constant blocks:	`SamplingRate`
Signal blocks:	`Frequency`, `PhaseInc`, `Phase`, `Output`
Module blocks:	`Sin`, `GreaterOrEqual`
Reaction blocks[2]:	`WrapPhase`

The domain assignments of the four signal blocks `Frequency`, `PhaseInc`, `Phase`, and `Output` control the mapping of these four expressions to callback functions on a target system.

[2]Reaction blocks will be covered in chapter 7.

Synchronous Frequency Update

Let us consider the case where the frequency of the oscillator is synchronously modulated at the same rate as its output. Code 5.7 is the Stride code to realize this case.

All signal blocks in this example are assigned to the same domain (`OscillatorOutput`). All signal blocks operating in sample-and-hold mode (non-zero rates) are assigned the same rate value (`OscillatorOutputRate`). Based on these assignments, the `Output` and `Frequency` signals are synchronized and evaluate at the same rate.

The `Frequency` signal represents a sine wave centered at 220Hz and spans +/- 40Hz at a rate of 1Hz (lines 32-24). The `Frequency` signal is driving the modulation of the oscillator expressed by the four stream expressions (lines 36-39).

Code 5.8 is sample code generated for Code 5.7 in the C language to run on some target platform[3]. On this target platform, the domain `OscillatorOutput` is declared and mapped to the `OscillatorOutput()` function. This function is treated as a callback and is called at $48{,}000$Hz. This rate is represented by `OscillatorOutputRate`.

Based on these domain and rate assignments, the code generator placed the C expressions corresponding to the five stream expressions (lines 32-39 of Code 5.7) in the `OscillatorOutput()` function (lines 12-16 of Code 5.8).

[3]The C code is for demonstration only and is not generated by the Stride code generator.

```
# The sampling rate of the oscillator output
constant SamplingRate {
    value:      48000
}

# All signals are set to OscillatorOutput
signal Frequency {
    default:    440.0
    rate:       OscillatorOutputRate
    domain:     OscillatorOutput
}
signal PhaseInc {
    default:    0.0
    rate:       OscillatorOutputRate
    domain:     OscillatorOutput
}
signal Phase {
    default:    0.0
    rate:       0
    domain:     OscillatorOutput
}
signal Output {
    default:    0.0
    rate:       OscillatorOutputRate
    domain:     OscillatorOutput
}
reaction WrapPhase {
    streams:     Phase - 6.28318530718 >> Phase;
}

# All expressions are evaluated in the OscillatorOutput domain
SineOsc ( frequency: 1.0 )
>> Level ( gain: 40.0 offset: 220.0 )
>> Frequency;

Frequency * 6.28318530718 / SamplingRate >> PhaseInc;
Phase >> Sin () >> Output;
Phase + PhaseInc >> Phase;
[ Phase , 6.28318530718 ] >> GreaterOrEqual () >> WrapPhase ();
```

Code 5.7: The oscillator output and its frequency update synchronously. (Stride)

```
INTEGER_TYPE SamplingRate = 48000;
REAL_TYPE Frequency = 440.0;
ATOMIC_REAL_TYPE PhaseInc = 0.0;
REAL_TYPE Phase = 0.0;
REAL_TYPE Output = 0.0;

```

```
void WrapPhase ( void ) {
    Phase = Phase - 6.28318530718;
}

void OscillatorOutputDomain ( REAL_TYPE &Output ) {
    Frequency = Level( SineOsc( 1.0, SamplingRate ), 40.0, 220.0 );
    PhaseInc = Frequency * 6.28318530718 / SamplingRate;
    Output = Sin( Phase );
    Phase = Phase + PhaseInc;
    if ( Phase >= 6.28318530718 ) WrapPhase();
}
```

Code 5.8: The oscillator output and its frequency update synchronously. (C)

Asynchronous Frequency Update

Let us consider the case where the frequency of the oscillator is asynchronously modulated at a rate different than its output. Code 5.9 is the Stride code to realize this case.

The `Frequency` and `PhaseInc` signal blocks are assigned to the `FrequencyUpdate` domain and their rate is set to `FrequencyUpdateRate`. The `Phase` and `Output` signal blocks are assigned to the `OscillatorOutput` domain. `Phase` is set to run in reactive mode (zero rate) while `Output`'s rate is set to `OscillatorOutputRate`.

Code 5.10 is sample code generated for Code 5.9 in the C language to run on some target platform[4]. On this target platform, the domain `FrequencyUpdate` is declared and mapped to the `FrequencyUpdate()` function. This function is treated as a callback and is called at $1{,}000$Hz. This rate is represented by `FrequencyUpdateRate`. On this target platform, the domain `OscillatorOutput` is declared and mapped to the `OscillatorOutput()` function. This function is treated as a callback and is called at

[4]The C code is for demonstration only and is not generated by the Stride code generator.

$48{,}000$Hz. This rate is represented by `OscillatorOutputRate`.

```
# The sampling rate of the oscillator output
constant SamplingRate {
    value:      48000
}

# Frequency and PhaseInc are set to FrequencyUpdate
signal Frequency {
    default:    440.0
    rate:       FrequencyUpdateRate
    domain:     FrequencyUpdate
}
signal PhaseInc {
    default:    0.0
    rate:       FrequencyUpdateRate
    domain:     FrequencyUpdate
}

# Phase and Output are set to OscillatorOutput
signal Phase {
    default:    0.0
    rate:       0
    domain:     OscillatorOutput
}
signal Output {
    default:    0.0
    rate:       OscillatorOutputRate
    domain:     OscillatorOutput
}
reaction WrapPhase {
    streams:     Phase - 6.28318530718 >> Phase;
}

# The following expressions are evaluated in the FrequencyUpdate
domain
SineOsc ( frequency: 1.0 )
>> Level ( gain: 40.0 offset: 220.0 )
>> Frequency;
Frequency * 6.28318530718 / SamplingRate >> PhaseInc;

# The following expressions are evaluated in the OscillatorOutput
domain
Phase >> Sin () >> Output;
Phase + PhaseInc >> Phase;
[ Phase , 6.28318530718 ] >> GreaterOrEqual () >> WrapPhase ();
```

Code 5.9: The oscillator output and fits requency update asynchronously. (Stride)

Based on the domain and rate assignments, the code generator placed the C expressions corresponding to the first two stream expressions (lines 34-37 of Code 5.9) in the `FrequencyUpdate()` function (lines 12-13 of Code 5.10) and placed the C expressions corresponding to the last three stream expressions (lines 40-42 of Code 5.9) in the `OscillatorOutput()` function (lines 17-19 of Code 5.10).

```
 1 INTEGER_TYPE SamplingRate = 48000;
 2 REAL_TYPE Frequency = 440.0;
 3 ATOMIC_REAL_TYPE PhaseInc = 0.0;
 4 REAL_TYPE Phase = 0.0;
 5 REAL_TYPE Output = 0.0;
 6
 7 void WrapPhase ( void ) {
 8     Phase = Phase - 6.28318530718;
 9 }
10
11 void FrequencyUpdate ( void ) {
12     Frequency = Level( SineOsc( 1.0, SamplingRate ), 40.0, 220.0 );
13     PhaseInc = Frequency * 6.28318530718 / SamplingRate;
14 }
15
16 void OscillatorOutput ( REAL_TYPE &Output ) {
17     Output = Sin( Phase );
18     Phase = Phase + PhaseInc;
19     if ( Phase >= 6.28318530718 ) WrapPhase();
20 }
```

Code 5.10: The oscillator output and fits requency update asynchronously. (C)

5.2.2 Sine Oscillator Module

In Stride, a module block encapsulates block declarations and stream expressions to perform a particular function. The internal blocks of a module connect with external blocks through ports. A module can have one or many ports.

Stride defines two port types: **main port** and **property port**. Both types have a direction. They can either be an **input port** or an **output port**. A module must have at least one main port. A module can only have a single main input port and a single main output port. A module can have a single property port, multiple property ports, or none. Connections with the main ports are established using the stream operator (>>) in stream expressions. Connections with property ports are established by assignment when a module is added to a stream expression.

Ports in Stride provide an interface for blocks declared inside a module to access property assignment information of blocks connected to the module's ports. This interface enables the configuration of the properties of internal blocks with respect to external ones. This interface also enables querying the size of block bundles connected to the module's ports.

To create a sine oscillator module with frequency control, the stream expressions and corresponding block declarations in Code 5.8 will have to be encapsulated inside a module block. The properties of the encapsulated signal blocks will have to be configured based on the properties of the blocks that get connected to the module's ports when the module is added to a stream expression.

Code 5.11 is the Stride module block declaration for the sine oscillator with frequency control. The module block has five properties: `ports`, `blocks`, `constraints`, `streams`, and `meta`.

The `ports` property of a module block lists the ports of a module. Four port types are defined in Stride that can be added to the list: `mainInputPort`, `mainOutputPort`, `propertyInputPort`, and `propertyOutputPort`. The names of these ports represent their type and direction. Each port type has a set of assignable properties.

```
module SineOsc {
    ports:  [
        mainOutputPort OutputPort {
            block:      Output
        }
        propertyInputPort FrequencyPort {
            name:       "frequency"
            block:      Frequency
            default:    440.0
            meta:       "The frequency of the oscillator in Hz."
        }
        propertyInputPort ResetPort {
            name:       "reset"
            block:      Reset
            default:    none
            meta:       "Resets the Phase of the oscillator. Accepts
            a switch or a trigger."
        }
    ]
    blocks:  [
        signal Output {
            default:    0.0
            type:       OutputPort.type
            rate:       OutputPort.rate
            domain:     OutputPort.domain
        }
        signal Frequency {
            default:    FrequencyPort.default
            type:       FrequencyPort.type
            rate:       FrequencyPort.rate
            domain:     FrequencyPort.domain
        }
        trigger Reset {
            mode:       "Rising"
            domain:     ResetPort.domain
        }
        signal Phase {
            default:    0.0
            type:       OutputPort.type
            rate:       0
            domain:     OutputPort.domain
            reset:      Reset
        }
        signal PhaseInc {
            default:    FrequencyPort.default * 6.28318530718 /
            OutputPort.rate
            type:       OutputPort.type
            rate:       FrequencyPort.rate
            domain:     FrequencyPort.domain
        }
        reaction WrapPhase {
            streams:    Phase - 6.28318530718 >> Phase;
        }
    ]
    constraints:    [
        [ OutputPort.rate, 0 ] >> LessOrEqual () >> Error ( message:
        "The rate of the signal block connected to the main output
        port of the SineOsc module cannot be less than or equal to
        zero.");
    ]
    streams: [
        Frequency * 6.28318530718 / OutputPort.rate >> PhaseInc;
        Phase >> Sin () >> Output;
        Phase + PhaseInc >> Phase;
        [ Phase, 6.28318530718 ] >> GreaterOrEqual () >> WrapPhase ()
        ;
    ]
    meta:   "Sine oscillator with frequency control. Bipolar output
    with range [ -1. , 1. ]."
}
```

Code 5.11: Sine oscillator module with frequency control in Stride. (SineOsc)

For the sine oscillator with frequency control, we need at least two ports[5]. We need one to access the output of the oscillator and another to control its frequency. Functionally, the port type of the port to access the output of the oscillator should be `mainOutputPort`, in order to connect the module to other blocks using the stream operator in a stream expression. Setting the frequency of the oscillator through a property port rather than a main port is an appropriate choice, since the frequency is a property of the oscillator. So, the port type of the frequency port should be `propertyInputPort`.

During the module's declaration, the two port type blocks are each assigned a unique name. The `mainOutputPort` port is called `OutputPort` and the `propertyInputPort` port is called `FrequencyPort`. The name assigned to a port is used to access the prop-

[5]To simplify the presentation, the `ResetPort` port and the `Reset` trigger in Code 5.11 will not be covered in this section but in a later chapter.

erties of a block that gets connected to the port of the module in a stream expression. A property is accessed by using the name of the port with a "dot" operator followed by the name of the property. The syntax is `PortName.propertyName`.

At declaration, each port is also assigned an internal block. The internal blocks of a module are declared under the `blocks` property. The main port `OutputPort`, is assigned the `Output` signal block and the property port `FrequencyPort`, is assigned the `Frequency` signal block. Two external blocks connected to the ports of the module in a stream expression will be directly connected to these two signal blocks inside the module. The property port `FrequencyPort` is also assigned a default value. If a block is not connected to the property port of the module when the module is added in a stream expression, this constant default value is connected to the internal signal block assigned to the port. The property port `FrequencyPort` also has a property called `name`. The constant string assigned to this property is the name of the property port as it appears when the module is added in a stream expression and an assignment is made to the port. In this case the property port is named `frequency`.

Along with the `Output` and `Frequency` signal blocks, three other blocks are declared in the `blocks` property of the module. `Phase` and `PhaseInc` are declared as signal blocks, while `WrapPhase` is declared as a reaction block. The scope of all the block declarations is local to the module.

So far, in previous code examples, signal blocks were assigned domains and rates that were pre-defined on a target platform. For a module to be reusable and compatible with any target platform, the domains and rates of blocks declared inside a module need to be abstracted and derived from its ports.

Based on the two examples of synchronous and asynchronous evaluation of signal blocks in the previous subsections, the domain and rate assignments of the internal signal blocks `Output` and `Phase` of the module have to be derived from its main output port (`OutputPort`), and assigned the values `OutputPort.domain` and `OutputPort.rate` respectively. These assignments place the evaluation of the stream expressions related to the `Output` and `Phase` signal blocks in the same domain as the signal block the module's output port gets connected to.

Based on the same examples, the domain and rate assignments of the signal blocks `Frequency` and `PhaseInc` are derived from the property input port (`FrequencyPort`). The domains are assigned the value `FrequencyPort.domain` and the rates are set to `FrequencyPort.rate`. If the signal block connected to the `frequency` property port of the module happens to be in the same domain as the signal block connected to its output port, all four expressions are evaluated synchronously in the domain of the signal block connected to the output. If the signal block connected to the `frequency` property port of the module happens to be in a domain different from the domain of the signal block connected to its output port, then the stream expression related to the signal blocks `Frequency` and `PhaseInc` is evaluated in this other domain.

At declaration, constraints can be added to a module block through its `constraints` property. The constraints are a set of conditions imposed on blocks and their property assignments. When the conditions of a constraint are not satisfied, a compile-time error is generated. For the sine oscillator module, the external block connected to the output of the module cannot have a rate equal to zero, since `OutputPort.rate` is used as a divisor in the stream expression evaluating `PhaseInc`.

The `streams` property of the module accepts a list of stream expressions. This is where stream expressions get encapsulated in a module.

The `meta` property accepts a sting constant. The string should describe the specific function a module performs. The description is incorporated into the auto-generated documentation of a module.

5.2.3 Code Generation for the Sine Oscillator Module

Code 5.12 is a C++ template class generated based on the sine oscillator module with frequency control (Code 5.11).

The domains defined by the main and property ports in the module are translated into methods of the class. The domains `OutputPort.domain` and `FrequencyPort.domain` are mapped to the `process_OutputDomain` and `process_FrequencyPortDomain` methods respectively. The default values of signal blocks are computed in initialization functions designated with the `init_` prefix, while `OutputPort_rate` constant is set through a class constructor.

Information between domains is exchanged over bridge signals that are instantiated outside the class definition. The concurrency requirement between these two domains will dictate how these bridge signals are instantiated and managed. Bridge signals and the concurrency are discussed in detail in chapter 6.

```
template<class OutputDataType, class FrequencyDataType>
class SineOsc {
public:
    SineOsc(float outputRate) : OutputPort_Rate(outputRate){
    }

    void process_OutputDomain(OutputDataType *Output, OutputDataType *Phase, OutputDataType PhaseInc) {
        Sin_00.process_OutputDomain(*Phase, &Sin_00_Output);
        *Output = Sin_00_Output;
        *Phase = *Phase + PhaseInc;
        OutputDataType BundleConnector_00[2];
        BundleConnector_00[0] = *Phase;
        BundleConnector_00[1] = 6.28318530718;
        GreaterOrEqual_00.process_OutputDomain(BundleConnector_00, &GreaterOrEqual_00_Output);
        if (GreaterOrEqual_00_Output){
            reaction_WrapPhase(Phase);
        }
    }

    void process_FrequencyPortDomain(FrequencyDataType Frequency, OutputDataType *PhaseInc) {
        *PhaseInc = Frequency * 6.28318530718 / OutputPort_Rate;
    }

    void init_Frequency(FrequencyDataType *Frequency) {
        *Frequency = FrequencyDataType(440.0);
    }

    void init_Phase(OutputDataType *Phase) {
        *Phase = OutputDataType(0.0);
    }

    void init_PhaseInc(OutputDataType *PhaseInc) {
        FrequencyDataType Frequency;
        init_Frequency(&Frequency);
        *PhaseInc = OutputDataType(Frequency) * 6.28318530718 / OutputPort_Rate;
    }

    void reaction_WrapPhase (OutputDataType *Phase) {
         *Phase = *Phase - 6.28318530718;
    }

private:
    using GreaterOrEqual_00_Type = GreaterOrEqual<OutputDataType,bool>;
    GreaterOrEqual_00_Type GreaterOrEqual_00;
    bool GreaterOrEqual_00_Output;
    using Sin_00_Type = Sin<OutputDataType>;
    Sin_00_Type Sin_00;
    OutputDataType Sin_00_Output;

    float OutputPort_Rate;
};
```

Code 5.12: C++ class generated for the SineOsc module in Code 5.11.

5.3 Using Modules in Stride

In the following subsections we will use the sine oscillator module (`SineOsc`) declared in the previous section to perform frequency modulation. We will consider two cases, where we will update the frequency of the sine oscillator synchronously and asynchronously with its output.

We will use a second module called `Level` along with the `SineOsc` module. `Level` is designed to apply a gain followed by an offset to a signal connected to its input. The Stride code for `Level` and the C++ template generated for it and shown in the following subsection.

5.3.1 Level Module

Code 5.13 is the code for the `Level` module in Stride. The module samples its main input port at the rate of the output port. It applies a gain to the incoming signal followed by an offset. The processed signal is presented at the output port. The processing happens in the domain of the output port.

Code 5.14 is the C++ template class generated for the `Level` module.

```
module Level {
    ports:      [
        mainInputPort InputPort {
            block:      Input
        }
        mainOutputPort OutputPort {
            block:      Output
        }
        propertyInputPort GainProperty {
            name:       "gain"
            block:      Gain
            default:    1.0
            meta:       "Amplifies or attenuates the signal."
        }
        propertyInputPort OffsetProperty {
            name:       "offset"
            block:      Offset
            default:    0.0
            meta:       "Offsets the signal after applying the gain."
        }
    ]
    blocks:     [
        signal Input {
            default:    0.0
            type:       OutputPort.type
            rate:       OutputPort.rate
            domain:     OutputPort.domain
        }
        signal Output {
            default:    0.0
            type:       OutputPort.type
            rate:       OutputPort.rate
            domain:     OutputPort.domain
        }
        signal Gain {
            default:    GainPort.default
            type:       GainPort.type
            rate:       GainPort.rate
            domain:     GainPort.domain
        }
        signal Offset {
            default:    OffsetPort.default
            type:       OffsetPort.type
            rate:       OffsetPort.rate
            domain:     OffsetPort.domain
        }
    ]
    streams:     Input * Gain + Offset >> Output;
    meta:        "Scales the input signal and applies an offset.
                  Formula:     output = input * gain + offset"
}
```

Code 5.13: Level module in Stride.

```
template<class OutputDataType, class GainDataType, class OffsetDataType>
class Level {
public:
    Level() {
    }

    void process_OutputDomain(OutputDataType Input, OutputDataType *Output, GainDataType Gain, OffsetDataType Offset) {
        *Output = ((Input * Gain) + Offset);
    }

    void process_GainPropertyDomain(GainDataType Gain, GainDataType *Gain_) {
        *Gain_ = Gain;
    }

    void process_OffsetPropertyDomain(OffsetDataType Offset, OffsetDataType *Offset_) {
        *Offset_ = Offset;
    }

    void init_Gain(GainDataType *Gain) {
        *Gain = OutputDataType(1.0);
    }

    void init_Offset(OffsetDataType *Offset) {
        *Offset = OutputDataType(0.0);
    }

private:
};
```

Code 5.14: C++ class generated for the Level module in Code 5.13.

5.3.2 Synchronous Frequency Modulation

Frequency modulation is achieved with two `SineOsc` modules and a `Level` module connected in two stream expressions. The Stride code for frequency modulation is shown in Code 5.15.

In the first stream expression, the frequency of the first `SineOsc` module instance is set to 1.0Hz. The module generates a bipolar signal in the range [-1.0, 1.0]. The output of the `SineOsc` module instance is connected to the input of the `Level` module. The `gain` and `offset` properties of the module are set to 40.0 and 220.0 respectively. The output of the `Level` module is connected to a signal called `Modulation`. The values of `Modulation` represent a sine wave oscillating at 1Hz, centered around 220.0Hz with a span of 80.0Hz.

```
signal Modulation {
    default:    0.0
    rate:       AudioRate
    domain:     AudioDomain
}

signal Output {
    default:    0.0
    rate:       AudioRate
    domain:     AudioDomain
}

SineOsc ( frequency: 1.0 )
>> Level ( gain: 40.0 offset: 220.0 )
>> Modulation;

SineOsc ( frequency: Modulation )
>> Output;
```

Code 5.15: Synchronous frequency modulation using SineOsc and Level modules.

The `Modulation` signal's rate and domain will first propagate into the `Level` module and consequently into the `SineOsc` module instance. The main processes inside these modules will be evaluated in the same domain as the `Modulation` signal's domain. Since the property ports of both modules are set to constant values, computations related to these ports will happen in a domain designated for evaluating constant expressions on the target platform.

In the second expression, the frequency of the second `SineOsc` module instance is connected to the `Modulation` signal. The output of the module is connected to a signal called `Output`. Based on these connections, the `SineOsc` module instance will be evaluated in the domains the `Modulation` and the `Output` signals are assigned to.

In Code 5.15, both the `Modulation` and `Output` signals are assigned to `AudioDomain` and run at the domain's rate, `AudioRate`. This makes the two signals synchronous to each other. Thus, the result is synchronous frequency modulation.

Based on the two signal declarations and constant value assignments, all expressions in the `SineOsc` and `Level` modules will be evaluated either in the `AudioDomain` or in the `ConstantDomain`.

The domain `AudioDomain` maps to a function called `AudioTick()`. This function represents the audio callback function on the target platform. The domain `ConstantDomain` is mapped to a function called `Constants()`. This function is called once at the beginning of the `main()` function of the target platform at the start of the program.

During code generation multiple bridge signals are created to connect the input(s) and output(s) of the methods related to the instantiated `SineOsc` and `Level` classes.

The mapping of domains, the instantiation of modules, and the connections established through bridge signals are shown in Code 5.16.

The `SineOsc` modules are instantiated with a sampling rate of `48,000`Hz, since this is the value of `AudioRate` on the target platform.

The generated code in its entirety can be found in Appendix C.

```
float   Modulation_AudioTick = 0.0;
float   Output_AudioTick = 0.0;

using  SineOsc_00_Type = SineOsc<float,float>;
SineOsc_00_Type SineOsc_00{48000};
float   SineOsc_00_Output_AudioTick;
float   SineOsc_00_Phase_AudioTick;
float   SineOsc_00_PhaseInc_Constant;

using  Level_00_Type = Level<float>;
Level_00_Type Level_00;
float   Level_00_Gain_Constant;
float   Level_00_Offset_Constant;

using  SineOsc_01_Type = SineOsc<float,float>;
SineOsc_01_Type SineOsc_01{48000};
float   SineOsc_01_Phase_AudioTick;
float   SineOsc_01_PhaseInc_AudioTick;

void AudioTick (float &ProcessOutput) {
    SineOsc_00.process_OutputDomain(&SineOsc_00_Output_AudioTick, &
    SineOsc_00_Phase_AudioTick, SineOsc_00_PhaseInc_Constant);
    Level_00.process_OutputDomain(SineOsc_00_Output_AudioTick, &
    Modulation_AudioTick, Level_00_Gain_Constant,
    Level_00_Offset_Constant);
    SineOsc_01.process_FrequencyPortDomain(Modulation_AudioTick, &
    SineOsc_01_PhaseInc_AudioTick);
    SineOsc_01.process_OutputDomain(&Output_AudioTick, &
    SineOsc_01_Phase_AudioTick, SineOsc_01_PhaseInc_AudioTick);
    ProcessOutput = Output_AudioTick;
}

void Constants () {
    SineOsc_00.process_FrequencyPortDomain(1.0, &
    SineOsc_00_PhaseInc_Constant);
    Level_00.process_GainPropertyDomain(40.0, &Level_00_Gain_Constant
    );
    Level_00.process_OffsetPropertyDomain(220.0, &
    Level_00_Offset_Constant);
}

void Initialize () {
    SineOsc_00.init_Phase(&SineOsc_00_Phase_AudioTick);
    SineOsc_01.init_Phase(&SineOsc_01_Phase_AudioTick);
    SineOsc_01.init_PhaseInc(&SineOsc_01_PhaseInc_AudioTick);
}
```

Code 5.16: C++ code generated for synchronous frequency modulation.

5.3.3 Asynchronous Frequency Modulation

By changing the domain assignment of the `Modulation` signal and setting it to a domain different than `AudioDomain`, `Modulation` can be evaluated asynchronously to the `Output` signal.

In Code 5.17, the domain of `Modulation` is assigned to `ControlDomain` and its rate is set to `ControlRate`. The `ControlDomain` domain is mapped to a function called `ControlTick()` on the target platform. `ControlTick()` is periodically called at $1{,}000$Hz (This value is represented by `ControlRate`).

The mapping of domains, the instantiation of modules, and the connections established through bridge signals are shown in Code 5.18.

With this domain assignment and rate change, the first `SineOsc` module instance is now instantiated with a sampling rate of `1,000`Hz, the rate of `ControlDomain`.

The generated code in its entirety can be found in Appendix C.

```
signal Modulation {
    default:    0.0
    rate:       ControlRate
    domain:     ControlDomain
}

signal Output {
    default:    0.0
    rate:       AudioRate
    domain:     AudioDomain
}

SineOsc ( frequency: 1.0 )
>> Level ( gain: 40.0 offset: 220.0 )
>> Modulation;

SineOsc ( frequency: Modulation )
>> Output;
```

Code 5.17: Asynchronous frequency modulation using SineOsc and Level modules.

```
float   Modulation_AudioTick = 0.0;
float   Output_AudioTick = 0.0;

using  SineOsc_00_Type = SineOsc<float, float>;
SineOsc_00_Type SineOsc_00{1000};
float   SineOsc_00_Output_ControlTick;
float   SineOsc_00_Phase_ControlTick;
float   SineOsc_00_PhaseInc_Constant;

using  Level_00_Type = Level<float, float, float>;
Level_00_Type Level_00;
float   Level_00_Gain_Constant;
float   Level_00_Offset_Constant;

using  SineOsc_01_Type = SineOsc<float, float>;
SineOsc_01_Type SineOsc_01{48000};
float   SineOsc_01_Phase_AudioTick;
float   SineOsc_01_PhaseInc_AudioTick_ControlTick;

void AudioTick (float &ProcessOutput) {
    SineOsc_01.process_OutputDomain(&Output_AudioTick, &
    SineOsc_01_Phase_AudioTick,
    SineOsc_01_PhaseInc_AudioTick_ControlTick);
    ProcessOutput = Output_AudioTick;
}

```

```
void ControlTick () {
    SineOsc_00.process_OutputDomain(&SineOsc_00_Output_ControlTick, &
    SineOsc_00_Phase_ControlTick, SineOsc_00_PhaseInc_Constant);
    Level_00.process_OutputDomain(SineOsc_00_Output_ControlTick, &
    Modulation_AudioTick, Level_00_Gain_Constant,
    Level_00_Offset_Constant);
    SineOsc_01.process_FrequencyPortDomain(Modulation_AudioTick, &
    SineOsc_01_PhaseInc_AudioTick_ControlTick);
}

void Constants () {
    SineOsc_00.process_FrequencyPortDomain(1.0, &
    SineOsc_00_PhaseInc_Constant);
    Level_00.process_GainPropertyDomain(40.0, &Level_00_Gain_Constant
    );
    Level_00.process_OffsetPropertyDomain(220.0, &
    Level_00_Offset_Constant);
}

void Initialize () {
    SineOsc_00.init_Phase(&SineOsc_00_Phase_ControlTick);
    SineOsc_01.init_Phase(&SineOsc_01_Phase_AudioTick);
    SineOsc_01.init_PhaseInc(&
    SineOsc_01_PhaseInc_AudioTick_ControlTick);
}
```

Code 5.18: C++ code generated for asynchronous frequency modulation.

The only bridge signal shared between the `AudioDomain` and `ControlDomain` domains is `SineOsc_01_PhaseInc_AudioTick_ControlTick`, the phase increment of the second `SineOsc` module instance. These two domains are running concurrently. The bridge signal is declared with a `float` data type. If an atomic operation on this data type is supported on the target platform, a mutual exclusion on this bridge signal is not required. However, if an atomic operation is not supported, there is a need for a synchronization model between the two concurrent domains to avoid memory corruption. Mutual exclusion schemes and synchronization policies are discussed in detain in chapter 6.

5.4 Summary

In this chapter, we presented and discussed the behavior of signal blocks in Stride based on their domain and rate assignments. Through the design of a sine oscillator with frequency control in Stride, we demonstrated how domain and rate property assignments of signals can be used to control the code generation process. We also introduced module blocks and presented how information propagates from the outside to the inside of these modules through ports. Next, we demonstrated how synchronous and asynchronous frequency modulation can be performed in Stride by using signals and modules.

Chapter 6

Domains and Concurrency

In the previous chapter we presented module blocks in Stride. We also showed the C++ code generated by the Stride code generator for a sine oscillator module. The generated C++ template class was characterized by its lack of internal state. The class encapsulated initialization and processing methods only. All state carrying variables were declared alongside the instantiation of their corresponding C++ template class. This approach simplified the distribution of code to different functions based on the domain assignments by the user.

In this chapter, we will discuss how generating stateless C++ template classes from modules simplifies code generation. It also accounts for the concurrency requirements set forth by the user to go beyond relying on atomic types as was the case in the previous chapter.

Next, we will present how the user defines and controls concurrency in Stride.

6.0.1 Domain Execution Order

As we mentioned in previous chapters, domains in Stride abstract functions. These functions form the main building blocks of programs generated, compiled, and executed by Stride. Some of these functions execute once, while others are passed as callback function executing on concurrent threads, either periodically or intermittently. An example of a function that executes once is a setup function where resources are allocated and configured at the beginning of a program. An example of a function that is passed as a callback function to a process (thread) is the audio callback function where audio samples get calculated periodically.

Functions in any program execute either sequentially or concurrently and so do domains in Stride. The user defines and declares the order of domain execution in Stride. A stream expression shown in Code 6.1 demonstrates the execution order of seven domains as declared and defined by the user. The domains are `InitializationDomain`, `ConstantsDomain`, `AudioDomain`, `ControlDomain`, `GuiDomain`, `TerminationDomain`, and `CleanupDomain`. Some of these domains are set to execute sequentially while others execute in parallel (concurrently). Domains are designed such that upon completing execution they trigger other domains to which they are connected.

```
InitializationDomain
>> ConstantsDomain
>> [AudioDomain, ControlDomain, GuiDomain, TerminationDomain]
>> And ()
>> CleanupDomain;
```

Code 6.1: Domain triggering for sequential and parallel execution.

The first domain to execute is `InitializationDomain`, where resources are initialized. Upon completing execution, `InitializationDomain` trigger `ConstantsDomain`, where expressions that result in constant values are computed. This is an example of sequential execution of domains.

When `ConstantsDomain` completes execution, it triggers four domains: `AudioDomain`, `ControlDomain`, `GuiDomain`, and `TerminationDomain`. These four domains run concurrently. The order in which they start executing is dictated by the order in which they appear in the bundle. When `TerminationDomain` completes execution, all its concurrent domains stop executing and `CleanupDomain` starts executing. The program terminates when `CleanupDomain` completes execution.

The four domains running concurrently might have to share memory to exchange information between them. In this example, `ControlDomain` might share control variables with `AudioDomain` and `AudioDomain` might share variables with the `GuiDomain`. If the variables being shared between these domains represent data types that are not atomic on the target platform, a synchronization policy and a mutual exclusion scheme are required to protect the integrity of these shared variables.

6.0.2 Concurrency Declaration

To handle shared memory between domains running concurrently in Stride, mutual exclusion rules can be created by the user to dictate how domains access shared memory. In Stride, these rules are known as policies. Code 6.2, is an example of a synchronization policy declared by the user.

The `mutualExclusion` declaration block called `TryLockOnReadLockOnWrite` defines a mutual exclusion scheme. In this scheme, the domain reading from a shared memory is directed to try to lock the mutual exclusion flag if the flag is available, or else continue with execution if the flag is not immediately available. The domain writing to the shared memory is directed to lock the mutual exclusion flag if it is available or wait until it becomes available and lock it.

```
mutualExclusion TryLockOnReadLockOnWrite {
    read:    TryLock
    write:   Lock
}

synchronization AudioReadControlWrite {
    readDomain:      AudioDomain
    writeDomain:     ControlDomain
    scheme:          TryLockOnReadLockOnWrite
}
```

Code 6.2: Mutual exclusion scheme and synchronization policy.

`AudioReadControlWrite` is a policy declared and defined between `AudioDomain` and `ControlDomain`. The policy calls for a mutual exclusion scheme to be used between `AudioDomain` and `ControlDomain` when they share a variable. The assigned scheme is `TryLockOnReadLockOnWrite`. The policy applies when `AudioDomain` is reading from the shared variable and `ControlDomain` is writing to the share variable.

6.1 Concurrency and Stateless C++ Template Classes

Giving the user the ability to control the concurrency model is important, especially when the user is trying to achieve real-time performance on a microcontroller through optimization. However, having a concurrency model becomes crucial when the user is targeting a platform designed around a 8-bit or 16-bit architecture and there is a need for single precision (32-bit) or double precision (64-bit) floating-point data types and computations. Defining and handling mutual exclusion in these cases becomes a necessity and is no longer considered an additional feature.

Because of the various data types and synchronization policies that can occur within Stride, a new approach is required when it comes to generating code to account for all possibilities. The conventional method of generating a class that holds internal state will no longer work because it will require generating a new class for each data types and concurrency policy.Let's consider a sine oscillator to demonstrate the problem and present a new approach to generating code that solves it.

A sine oscillator needs to track the state of two variables. The variables are its phase and phase increment. Variables representing state are referred to as bridge signals in Stride. If we inspect the C++ code generated for the sine oscillator module in Code 5.12, we notice the phase and phase increment variables are not part of the generated C++ template class. They appear as arguments to the methods of the generated class.

The methods of the generated class can be divided into two sets. The first set are initialization methods and start with the `init_` prefix. The second set are processing methods and start with the `process_` prefix. The initialization methods, as the prefix indicates, initialize and reset variables. The processing methods perform computations on these variables to update them. All these methods can be called from any domain in Stride as long as the variables passed to them adhere to the concurrency policies defined between the domains.

Given the generated class is a stateless template class, it is the only implementation needed, since it accepts any data type and satisfies any concurrency scheme applied to its variables.

6.1.1 Asynchronous Frequency Modulation with Concurrency

In Code 5.17 we presented asynchronous frequency modulation in Stride. In the generated code, the bridge signal `SineOsc_01_PhaseInc_AudioTick_ControlTick` (the phase increment of the second oscillator), is a variable shared between two domains running concurrently. While the bridge signal is being read from in `AudioDomain`, it is being written to in `ControlDomain`. During code generation, the assumption was that the data type assigned to this bridge signal is an atomic type on the target platform. If that was not the case, a mutual exclusion would have been required to guarantee the data integrity of the bridge signal. To accommodate this requirement a mutual exclusion scheme and a concurrency policy could be specified to instruct the Stride code generator to generate the necessary mutual exclusion code.

The generated code for the asynchronous frequency modulation with the concurrency policy specified in Code 6.2 is shown in Code 6.3. The Stride code and the generated code in their entirety can be found in Appendix C.

```
std::mutex R_AudioTick_W_ControlTick_Mutex;

float   Modulation_AudioTick = 0.0;
float   Output_AudioTick = 0.0;

using  SineOsc_00_Type = SineOsc<float, float>;
SineOsc_00_Type SineOsc_00{1000};
float   SineOsc_00_Output_ControlTick;
float   SineOsc_00_Phase_ControlTick;
float   SineOsc_00_PhaseInc_Constant;

using  Level_00_Type = Level<float, float, float>;
Level_00_Type Level_00;
```

```cpp
float   Level_00_Gain_Constant;
float   Level_00_Offset_Constant;

using  SineOsc_01_Type = SineOsc<float, float>;
SineOsc_01_Type SineOsc_01{48000};
float   SineOsc_01_Phase_AudioTick;
float   SineOsc_01_PhaseInc_AudioTick;
float   SineOsc_01_PhaseInc_AudioTick_ControlTick;

void AudioTick (float &ProcessOutput) {
    if (R_AudioTick_W_ControlTick_Mutex.try_lock()) {
        SineOsc_01_PhaseInc_AudioTick =
        SineOsc_01_PhaseInc_AudioTick_ControlTick;
        R_AudioTick_W_ControlTick_Mutex.unlock();
    }
    SineOsc_01.process_OutputDomain(&Output_AudioTick, &
    SineOsc_01_Phase_AudioTick, SineOsc_01_PhaseInc_AudioTick);
    ProcessOutput = Output_AudioTick;
}

void ControlTick () {
    SineOsc_00.process_OutputDomain(&SineOsc_00_Output_ControlTick, &
    SineOsc_00_Phase_ControlTick, SineOsc_00_PhaseInc_Constant);
    Level_00.process_OutputDomain(SineOsc_00_Output_ControlTick, &
    Modulation_AudioTick, Level_00_Gain_Constant,
    Level_00_Offset_Constant);
    R_AudioTick_W_ControlTick_Mutex.lock();
    SineOsc_01.process_FrequencyPortDomain(Modulation_AudioTick, &
    SineOsc_01_PhaseInc_AudioTick_ControlTick);
    R_AudioTick_W_ControlTick_Mutex.unlock();
}

void Constants () {
    SineOsc_00.process_FrequencyPortDomain(1.0, &
    SineOsc_00_PhaseInc_Constant);
    Level_00.process_GainPropertyDomain(40.0, &Level_00_Gain_Constant
    );
    Level_00.process_OffsetPropertyDomain(220.0, &
    Level_00_Offset_Constant);
}

void Initialize () {
    SineOsc_00.init_Phase(&SineOsc_00_Phase_ControlTick);
    SineOsc_01.init_Phase(&SineOsc_01_Phase_AudioTick);
    SineOsc_01.init_PhaseInc(&SineOsc_01_PhaseInc_AudioTick);
    SineOsc_01.init_PhaseInc(&
    SineOsc_01_PhaseInc_AudioTick_ControlTick);
}
```

Code 6.3: C++ code generated for asynchronous frequency modulation with concurrency.

If we compare the two versions of the generated code for the asynchronous frequency control (Code 5.18 and Code 6.3), we notice the addition of a `mutex` (line 1 of Code 6.3) and the locking and unlocking sequences inserted into the `AudioTick()` and `ControlTick()` functions (lines 24, 35, and 37 of Code 6.3) where the bridge signal is being read from or written to. The rest of the code is identical in the two versions, including the implantation of the `SineOsc` and `Level` C++ template classes.

6.2 Discussion

Had the generated C++ template class for the `SineOsc` held the states of its variables internally, we would have had to generate two different versions of the class in order to satisfy the requirements of the two versions of asynchronous frequency control we presented (atomic type vs mutual exclusion). The number of classes to be generated would have proportionally increased with the increase in requirements and would have resulted in the need for a far more complex code generator design.

This approach to generating stateless C++ template classes allows for the design of helper classes that could be utilized to further simplify the task of the code generator, which in turn simplifies its design.

This approach to creating stateless C++ template classes for unit generators, like the `SineOsc`, is a departure from the conventional way most unit generators are implemented in music programming languages. These languages were not designed to handle concurrency the way Stride does. Stride gives the user full control over where (domain) and how often (rate) signals are evaluated, without corrupting their integrity, while achieving high levels of performance optimization and efficiency.

6.3 Summary

In this chapter we presented how Stride handles concurrency. We presented how the user defines and declares mutual exclusion schemes and concurrency policies to generate and distribute code on multiple concurrently running domains while maintaining the integrity of bridge signals. We presented an approach to handle the requirements posed by complex concurrency models through the generation of stateless C++ template classes. This flexible approach accommodates the synchronization requirements set forth by the user without having to generate custom C++ classes for different scenarios.

Chapter 7

Interaction Design with Triggers and Reactions

In Stride, interaction design is abstracted through `trigger` and `reaction` blocks.

Triggers allow synchronous or asynchronous events to propagate within a domain or across multiple domains. Reaction blocks, like module blocks, enclose stream expressions. Expressions enclosed in a reaction block are evaluated when the reaction block is activated. Reactions are activated using trigger blocks or switch blocks.

In this chapter, we will first cover the behavior of the switch block followed by the trigger block. Next, we will design interaction using reaction blocks and activating them with switch and trigger blocks.

7.1 The Switch Block

The behavior of a switch block is identical to that of a signal block. The only difference between the two is that switch blocks have Boolean states. A switch block is either in a true state or a false state. The keywords `on` and `off` in Stride represent the true and false states of a switch respectively. Code 7.1 shows the default declaration of a switch block.

```
switch BlockName {
    default:     on                   # Default value
    rate:        PlatformRate         # The switch's rate
    domain:      PlatformDomain       # The switch's domain
    reset:       none                 # Resets switch to default value
    meta:        "A switch block"     # Meta information
}
```

Code 7.1: Switch block declaration.

Code 7.2 shows the declaration of a switch block called `BypassSwitch`. `BypassSwitch` samples the `Greater` module at `ControlRate`. `BypassSwitch` is `on` when the `SineOsc` module's output is positive. Since `SineOsc`'s frequency is set to 0.5Hz, its output will alternate between positive and negative values every 1 second. Thus, `BypassSwitch` represents a unipolar square signal at `0.5`Hz with a 50% duty cycle.

```
switch BypassSwitch {
    default:     off
    rate:        ControlRate
    domain:      ControlDomain
    meta:        "A unipolar square signal"
}

[ SineOsc ( frequency: 0.5 ) , 0.0 ] >> Greater () >> BypassSwitch;

Input >> Process ( property: Value bypass: BypassSwitch ) >> Output;
```

Code 7.2: An example of a switch controlling the state of a module.

The `BypassSwitch` switch block is connected to the `bypass` port of the `Process` module. When `BypassSwitch` is `off`, the `Output` signal represents the processed values of the `Input` signal through the `Process` module. When `BypassSwitch` is `on`, the `Output` signal has the same value as the `Input` signal because `Process` is in pass-through mode.

Signal blocks and switch blocks are interchangeable in Stride. A signal block holding a non-zero value is equivalent to a switch block with an `on` state. A signal block holding a zero value is equivalent to a switch block with an `off` state. A switch block with an `on` state is equivalent to a signal block with value `1` if the `type` port of the signal block is set to "Integer" or `1.0` if the `type` port of the signal block is set to "Real". A switch block with an `off` state is equivalent to a signal block with value `0` if the `type` port of the signal block is set to "Integer" or `0.0` if the `type` port of the signal block is set to "Real".

7.2 The Trigger Block

In Stride, triggers communicate synchronous and asynchronous events and are designed to automatically re-arm after they have been triggered. Code 7.3 shows the default declaration of a trigger block.

```
trigger TriggerName {
    edge:    "Rising"              # The edge that triggers the trigger
    domain:  PlatformDomain        # The trigger's domain
    meta:    "A trigger block."    # Meta information
}
```

Code 7.3: Trigger block declaration.

Assigning a domain to the `domain` port of a trigger block is required at declaration. A trigger's state is evaluated and re-armed (if triggered) in the domain it is assigned to. Since a trigger is not assigned a rate, it is evaluated and re-armed at the rate of its assigned domain.

Like the switch block, the trigger block has Boolean states. A trigger is in the `on` state when it is triggered and `off` state when it is armed. When triggered, a trigger transitions from the `off` state to the `on` state. A trigger stays in the `on` state for a single clock cycle of the domain it is assigned to, until it is re-armed. The domain a trigger is assigned to is responsible for re-arming the trigger by switching its state from `on` to `off`. The user cannot re-arm a trigger.

The `edge` property of a trigger block can be assigned to one of the following edge transition types: "Rising", "Falling", or "Both". This property is relevant only if a switch block or a signal block is connected to its main input port. Otherwise, this property is ignored by the Stride interpreter. The edge transition type indicates the edge transition(s) of a signal block or a switch block that would result in triggering a trigger.

7.2.1 Single Domain Trigger Example

Code 7.4 is an example of a trigger synchronously resetting a signal every second. The example starts with two block declarations. The first is a signal block called `Count` and the second is a trigger block called `ResetCount`. `ResetCount` is connected to the `reset` port of `Count`. Every time `ResetCount` is triggered, `Count` is reset to its default value `0`. Both blocks are assigned to a domain called `ControlDomain`. The signal and the trigger are synchronous, since they are assigned to the same domain. `ControlDomain` runs at `ControlRate`. Let us assume `ControlRate` is 100Hz. `Count` is assigned a rate of 10Hz. That is, the value of `Count` is updated every 0.1 seconds. `ResetCount` is evaluated at the rate of `ControlDomain`. Thus, `ResetCount` is evaluated every 0.01 seconds.

In the first stream expression, the `SineOsc` module and the `Greater` module are also evaluated at `ControlRate`. They both derive the rate of their internal blocks from the rate of `ResetCount`.

The `frequency` property port of `SineOsc` is set to 1.0Hz and its output is compared to `0.0` by the `Greater` module. The output of `Greater` will transition from `off` to `on` at 1.0Hz (once every second) when `SineOsc` transitions from its negative to its positive swing. `Greater`'s output's transition represents a rising edge. `ResetCount` is set to be triggered on a rising edge, since its `edge` port is set to `"Rising"`. Therefore, `ResetCount` is triggered and automatically re-armed once every second.

```
signal Count {
    default:    0
    rate:       10.0
    domain:     ControlDomain
    reset:      ResetCount
}

trigger ResetCount {
    edge:       "Rising"
    domain:     ControlDomain
    meta:       "A trigger to reset the Count signal."
}

SineOsc ( frequency: 1.0 ) , 0.0 ] >> Greater () >> ResetCount;

Count + 1 >> Count;
```

Code 7.4: An example of a trigger resetting a signal.

In the second stream expression, `Count` is incremented by 1 every 0.1 seconds. Since

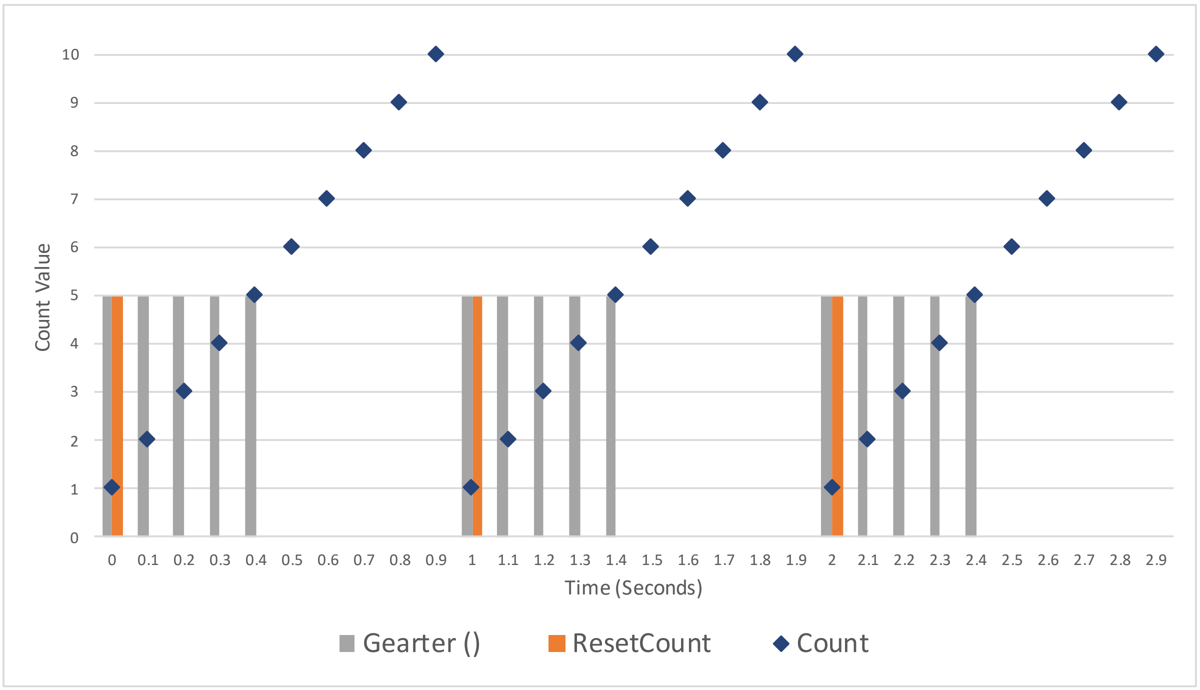


Figure 7.1: A trigger resetting a signal.

`ResetCount` and `Count` are synchronized and `ResetCount` is triggered every second, `Count` gets reset every second.

The states of `Greater`'s output and `ResetCount` and the value of `Count` are shown in Figure 7.1 over a 3 seconds interval.

7.2.2 Multiple Domain Trigger Example

Triggers in Stride can be utilized to communicate asynchronous events across domains. Triggers shared across domains adhere to concurrency policies declared by the user. In Stride, signal and switch blocks can be directly connected to the input of trigger blocks, while triggers can be directly connected to the `reset` property of signal and switch blocks. To cover all connection possibilities between different block types across multiple domains, triggers in Stride are implemented using the observer pattern[27]. Triggers capture and pass events to their observers when they are triggered. That is, any port or block connected to a trigger registers with it as an observer. When the trigger is triggered, ports and blocks registered with it get notified.

The example shown in Code 7.5 demonstrates how asynchronous event communication is achieved between two domains while adhering to a mutual exclusion scheme set by the user. All connections between the domains and their related modules, signals, switches, and triggers are shown in Figure 7.2. Blocks and modules highlighted in blue belong to the `AudioDomain` domain, while ones highlighted in orange belong to the `ControlDomain` domain. Excerpts from the generated C++ code are shown in Code 7.8.

```
use RtAudioWithBoost on Current

mutualExclusion LockOnReadLockOnWrite {
    read:   Lock
    write:  Lock
}

synchronization AudioReadControlWrite {
    readDomain:     AudioDomain
    writeDomain:    ControlDomain
    mode:           LockOnReadLockOnWrite
}

switch Positive {
    default:    off
    rate:       ControlRate
    domain:     ControlDomain
}

signal Ramp {
    default:    0
    rate:       ControlRate
    domain:     ControlDomain
}
```

```

trigger RampRolled {
    edge:         "Rising"
    domain:       AudioDomain
}

signal SawTooth {
    default:     0.0
    rate:         AudioRate
    domain:       AudioDomain
}

signal Output {
    default:     0.0
    rate:         AudioRate
    domain:       AudioDomain
}

[ SineOsc ( frequency: 1.0 ), 0.0 ] >> Greater () >> Positive;

Counter ( start: 0 increment: 1 roll: 4 reset: Positive rolled:
RampRolled ) >> Ramp;

Counter ( start: 0.0 increment: 0.025 roll: 10000.0 reset: RampRolled
 ) >> SawTooth;

SineOsc ( frequency: 220.0 reset: Positive )
>> ResonantLowPass ( frequency: SawTooth + 100.0 qFactor: 4.0 reset:
Positive )
>> Level ( gain: 0.2 )
>> Output;

Output >> AudioOut[1:2];
```

Code 7.5: An example with triggers in two domains.

The example starts with a mutual exclusion scheme declaration, followed by a synchronization policy declaration between two domains running concurrently. The two concurrently running domains are `AudioDomain` and `ControlDomain`. `AudioDomain` runs at `AudioRate` (48KHz) and `ControlDomain` runs at `ControlRate` (10Hz).

The `Positive` switch block and the `Ramp` signal block are assigned to `ControlDomain`. The `rampRolled` trigger block and the `SawTooth` and `Output` signal blocks are assigned

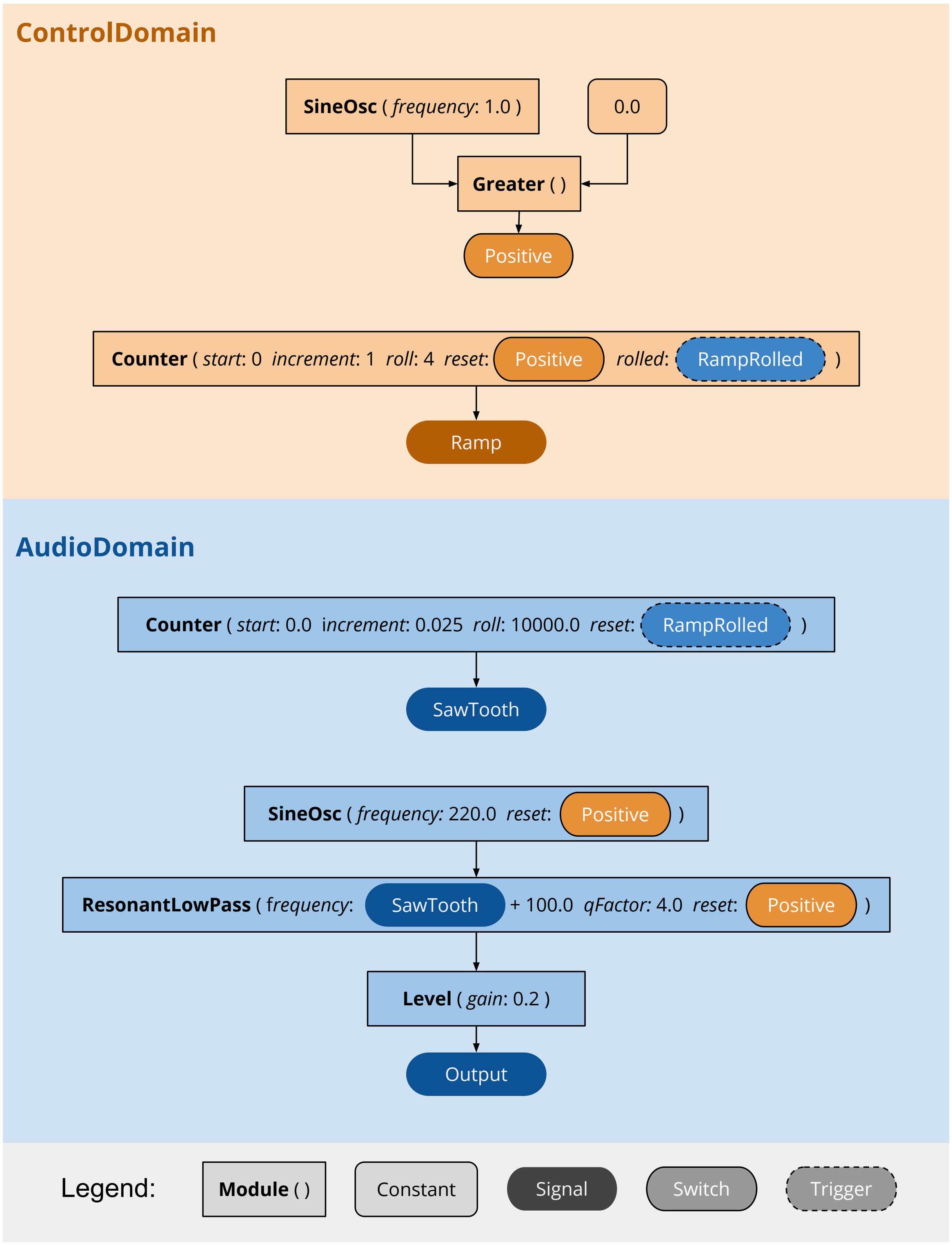

ControlDomain
SineOsc (frequency: 1.0)
0.0
Greater ( )
Positive
Counter (start: 0 increment: 1 roll: 4 reset: Positive rolled: RampRolled)
Ramp
AudioDomain
Counter (start: 0.0 increment: 0.025 roll: 10000.0 reset: RampRolled)
SawTooth
SineOsc (frequency: 220.0 reset: Positive)
ResonantLowPass (frequency: SawTooth + 100.0 qFactor: 4.0 reset: Positive)
Level (gain: 0.2)
Output
Legend:
Module ( )
Constant
Signal
Switch
Trigger

Figure 7.2: The domain assignments of blocks and their relationships in Code 7.5.

to AudioDomain.

The first stream expression produces a 1Hz unipolar square wave available through the Positive switch block. Every module in this stream expression is evaluated in ControlDomain. Based on the connections established in the following stream expressions, Positive triggers other triggers in AudioDomain and ControlDomain.

In the second stream expression, a five-step ramp signal is generated by the Counter module at ControlRate. The Counter derives its rate from the Ramp signal[1].

The Stride code and the generated C++ class for the Counter module are shown in Code 7.6 and Code 7.7 respectively.

```
module Counter {
    ports:      [
        mainOutputPort OutputPort {
            block:      Output
        }
        propertyInputPort StartValuePort {}
            name:       "start"
            block:      StartValue
            default:    0.0
        }
        propertyInputPort IncrementValuePort {
            name:       "increment"
            block:      IncrementValue
            default:    0.001
        }
        propertyInputPort RollValuePort {
            name:       "roll"
            block:      RollValue
            default:    1.0
        }
        propertyInputPort ResetPort {
            name:       "reset"
            block:      ResetCounter
            default:    none
        }
        propertyOutputPort RolledPort {
```

[1] Ramp takes the following values every 0.1 seconds: 0, 1, 2, 3, 4, 5, 0, 1, 2, ...

```
            name:        "rolled"
            block:       CounterRolled
        }
    ]
    blocks:     [
        signal Output {
            type:        OutputPort.type
            rate:        OutputPort.rate
            domain:      Output.domain
        }
        constant StartValue {
            type:        OutputPort.type
            value:       none
            domain:      auto
        }
        constant IncrementValue {
            type:        OutputPort.type
            value:       none
            domain:      auto
        }
        constant RollValue {
            type:        OutputPort.type
            value:       none
            domain:      auto
        }
        trigger ResetCounter {
            edge:        "Rising"
            domain:      ResetPort.domain
        }
        trigger CounterRolled {
            edge:        "Rising"
            domain:      OutputPort.domain
        }
        signal Accumulator {
            default:     StartValue
            type:        OutputPort.type
            rate:        OutputPort.rate
            domain:      OutputPort.domain
            reset:       [ CounterRolled , ResetCounter ]
        }
    ]
    streams:    [
        Accumulator >> Output;
        Accumulator + IncrementValue >> Accumulator;
        [ Accumulator, RollValue ] >> Greater () >> CounterRolled;
    ]
}
```

Code 7.6: Counter module in Stride.

```cpp
template <class OutputDataType>
class Counter {
public:
    Counter(float outputRate, OutputDataType startValue,
    OutputDataType incrementValue, OutputDataType rollValue) :
    OutputPort_Rate(outputRate), StartValue(startValue),
    IncrementValue(incrementValue), RollValue(rollValue) {
    }

    void process_OutputDomain(OutputDataType *Output, stride::Trigger
     *CounterRolled, OutputDataType *Accumulator) {
        *Output = *Accumulator;
        *Accumulator = *Accumulator + IncrementValue;
        OutputDataType BundleConnector_00[2];
        BundleConnector_00[0] = *Accumulator;
        BundleConnector_00[1] = RollValue;
        Greater_00.process_OutputDomain(BundleConnector_00, &
        Greater_00_Output);
        CounterRolled->Update(Greater_00_Output);
    }

    void process_ResetPortDomain() {
    }

    void init_Accumulator(OutputDataType *Accumulator) {
        *Accumulator = StartValue;
    }

private:
    float OutputPort_Rate;
    const OutputDataType StartValue;
    const OutputDataType IncrementValue;
    const OutputDataType RollValue;

    using Greater_00_Type = Greater<OutputDataType,bool>;
    Greater_00_Type Greater_00;
    bool Greater_00_Output;
};
```

Code 7.7: C++ class generated from the Counter module.

Just like the `SineOsc` module, covered in the previous chapters, the generated C++ template class does not have any variables as members of the class. This exclusion also applies to triggers and in this case to the `ResetCounter` and `CounterRolled` triggers. Just like signals, triggers can be shared between domains. Concurrency policies applied to signals are also applied to triggers.

The `Counter` module has two trigger ports. The ports are called `reset` and `rolled`. `reset` is an input port while `rolled` is an output port. When a trigger connected to the `reset` port is triggered the counter's accumulator is reset to its default value. The counter's accumulator is also reset if a switch connected to the `reset` port transitions from `off` to `on`. This computation (resetting the accumulator) is performed in the domain of the trigger block or the switch block connected to the `reset` port. In this stream expression the `Positive` switch block is connected the `reset` port. When the `Positive` switch block transitions from `off` to `on`, the accumulator is reset to `0`. This computation happens every second in `ControlDomain`.

A trigger connected to the `rolled` port will be triggered when the accumulator is greater than the constant value assigned to the `roll` port. Any module, signal, or switch block connected to this trigger will be notified and evaluated in the domain the trigger is assigned at declaration. In this stream expression, the trigger `RampRolled` is connected to the `rolled` port. This makes the `RampRolled` trigger an observer of the `CounterRolled`. `CounterRolled` is declared inside the `Counter` module. These two triggers are in different domains. `RampRolled` is declared in the `AudioDomain` while `CounterRolled` is assigned to `ControlDomain` by since it derives its domain from the signal connected to the output port of the `Counter` module. Through the connection established between these two triggers, an event in one domain is propagated to another domain.

In the third stream expression, a second ramp is generated. The value of the ramp is represented by the `SawTooth` signal. Unlike the `Counter` module in the second stream expression, this one is evaluated in `AudioDomain`. The `reset` port of this `Counter` module is connected the `RampRolled` trigger. `RampRolled` is triggered every time this `Counter` module rolls. However, this `Counter` module never rolls since its accumulator is reset every second. The accumulator reaches a maximum value of $1{,}200$[2] before it is reset to its default value 0.

In the fourth stream expression, a `SineOsc` module is connected to a `ResonantLowPass` module, a resonant low pass filter module. The `ResonantLowPass` module is connected to a `Level` module. `Level`'s main output is connected to the `Output` signal. `SineOsc`'s and `ResonantLowPass`'s `reset` property ports are connected to the `Positive` switch. The cutoff frequency of the resonant low pass filter is controlled by `SawTooth`. Most computations in this stream expression happen in `AudioDomain`, since `Output` and `Sawtooth` are assigned to `AudioDomain`. Only computations related to resetting the state of signals happen in `ControlDomain`. One such signal is the `Phase` signal of the `SineOsc` module. The trigger responsible for resetting the `Phase` signal derives its rate from the `Positive` signal connected to `SineOsc`'s main output and `Positive` is in the `ControlDomain`.

The C++ code generated from Code 7.5 is shown in Code 7.8. The generated code relies on the Stride helper classes[3]. These classes have been designed to account for the different conditions and requirements signals, switches, and triggers have to meet in generated code. The helper classes simplify the code generation process.

```
std::mutex RW_AudioTick_Rst_ControlTick_Mutex;
std::mutex R_AudioTick_W_ControlTick_Mutex;

bool     Positive_ControlTick = false;
```

[2]0.025 x 48,000 samples/second x 1 second = 1,200
[3]The full code of the Stride helper classes used in this example are shown in Appendix D

```
int      Ramp_ControlTick = 0;
float    Output_AudioTick = 0.0;
float    SawTooth_AudioTick = 0.0;

using RampRolled_Type = stride::Trigger_MD_TriggerControlled<stride::
sync::lock, stride::sync::lock>;
RampRolled_Type RampRolled_AudioTick(&R_AudioTick_W_ControlTick_Mutex
);
stride::TriggerObserverBlock<RampRolled_Type>
RampRolled_AudioTick_Observer(&RampRolled_Type::Fire, &
RampRolled_AudioTick);

using SineOsc_00_Type = SineOsc<float, float>;
SineOsc_00_Type SineOsc_00{1.0/(CONTROL_TIME_MS/1000.0)};
float    SineOsc_00_Output_ControlTick;
float    SineOsc_00_Phase_ControlTick;
float    SineOsc_00_PhaseInc_Constant;

using Greater_00_Type = Greater<float, bool>;
Greater_00_Type Greater_00;

using Counter_00_Type = Counter<int>;
Counter_00_Type Counter_00{10};
stride::Trigger_SwitchControlled Counter_00_ResetCounter_ControlTick(
stride::Trigger_SwitchControlled::TriggerMode::RISING);
stride::Trigger_SwitchControlled Counter_00_CounterRolled_ControlTick
(stride::Trigger_SwitchControlled::TriggerMode::RISING);
using Counter_00_Accumulator_Type = stride::Signal_SDRWRst<
Counter_00_Type, int>;
Counter_00_Accumulator_Type Counter_00_Accumulator_ControlTick(&
Counter_00_Type::init_Accumulator, &Counter_00);
stride::TriggerObserverBlock<Counter_00_Accumulator_Type>
Counter_00_Accumulator_ControlTick_Observer(&
Counter_00_Accumulator_Type::Reset, &
Counter_00_Accumulator_ControlTick);

using Counter_01_Type = Counter<float>;
Counter_01_Type Counter_01{48000, 0.0, 0.025, 10000.0};
stride::Trigger_SD_TriggerControlled
Counter_01_ResetCounter_AudioTick;
stride::TriggerObserverBlock<stride::Trigger_SD_TriggerControlled>
Counter_01_ResetCounter_AudioTick_Observer(&stride::
Trigger_SD_TriggerControlled::Fire, &
Counter_01_ResetCounter_AudioTick);
stride::Trigger_SwitchControlled Counter_01_CounterRolled_AudioTick(
stride::Trigger_SwitchControlled::TriggerMode::RISING);
using Counter_01_Accumulator_Type = stride::Signal_SDRWRst<
Counter_01_Type, float>;
Counter_01_Accumulator_Type Counter_01_Accumulator_AudioTick(&
Counter_01_Type::init_Accumulator, &Counter_01);
```

```
stride::TriggerObserverBlock<Counter_01_Accumulator_Type> Counter_01_Accumulator_AudioTick_Observer(&Counter_01_Accumulator_Type::Reset, &Counter_01_Accumulator_AudioTick);

using SineOsc_01_Type = SineOsc<float, float>;
SineOsc_01_Type SineOsc_01{48000};
float SineOsc_01_Output_AudioTick;
stride::Trigger_SwitchControlled SineOsc_01_Reset_ControlTick(stride::Trigger_SwitchControlled::TriggerMode::RISING);
using SineOsc_01_Phase_Type = stride::Signal_SDRW_MDRst<SineOsc_01_Type, float, stride::sync::lock, stride::sync::lock>;
SineOsc_01_Phase_Type SineOsc_01_Phase_AudioTick(&SineOsc_01_Type::init_Phase, &SineOsc_01, &RW_AudioTick_Rst_ControlTick_Mutex);
stride::TriggerObserverBlock<SineOsc_01_Phase_Type> SineOsc_01_Phase_AudioTick_Observer(&SineOsc_01_Phase_Type::Reset, &SineOsc_01_Phase_AudioTick);
float SineOsc_01_PhaseInc_Constant;

using ResonantLowPass_00_Type = ResonantLowPass<float, float, float>;
ResonantLowPass_00_Type ResonantLowPass_00{48000};

// Resonant Low Pass declaration code has been removed

using Level_00_Type = Level<float, float, float>;
Level_00_Type Level_00;
float   Level_00_Gain_Constant;
float   Level_00_Offset_Constant;

void AudioTick (float &ProcessOutput) {
  RampRolled_AudioTick.Update();

  Counter_01_ResetCounter_AudioTick.Update();
  Counter_01_Accumulator_AudioTick.Swap();
  Counter_01.process_OutputDomain(&SawTooth_AudioTick, &Counter_01_CounterRolled_AudioTick, Counter_01_Accumulator_AudioTick.Write());

  SineOsc_01_Phase_AudioTick.Swap();
  SineOsc_01.process_OutputDomain(&SineOsc_01_Output_AudioTick, SineOsc_01_Phase_AudioTick.Write(), SineOsc_01_PhaseInc_Constant);

// Resonant Low Pass processing method calls have been removed

  Level_00.process_OutputDomain(ResonantLowPass_00_Output_AudioTick, &Output_AudioTick, Level_00_Gain_Constant, Level_00_Offset_Constant);

  ProcessOutput = Output_AudioTick;
}
```

```

void ControlTick () {
  SineOsc_00.process_OutputDomain(&SineOsc_00_Output_ControlTick, &
  SineOsc_00_Phase_ControlTick, SineOsc_00_PhaseInc_Constant);

  float BundleConnector_00[2];
  BundleConnector_00[0] = SineOsc_00_Output_ControlTick;
  BundleConnector_00[1] = 0.0;
  Greater_00.process_OutputDomain(BundleConnector_00, &
  Positive_ControlTick);

  Counter_00_ResetCounter_ControlTick.Update(Positive_ControlTick);
  Counter_00_Accumulator_ControlTick.Swap();
  Counter_00.process_OutputDomain(&Ramp_ControlTick, &
  Counter_00_CounterRolled_ControlTick,
  Counter_00_Accumulator_ControlTick.Write());

  SineOsc_01_Reset_ControlTick.Update(Positive_ControlTick);

// Resonant Low Pass reset call has been removed
}

void Constants () {
  SineOsc_00.process_FrequencyPortDomain(1.0, &
  SineOsc_00_PhaseInc_Constant);
  SineOsc_01.process_FrequencyPortDomain(220.0, &
  SineOsc_01_PhaseInc_Constant);

// Resonant Low Pass constant computations have been removed

  Level_00.process_GainPropertyDomain(0.2, &Level_00_Gain_Constant);
}

void Initialize () {
  SineOsc_00.init_Phase(&SineOsc_00_Phase_ControlTick);

  Counter_00_CounterRolled_ControlTick.Register(&
  RampRolled_AudioTick_Observer);
  Counter_00_CounterRolled_ControlTick.Register(&
  Counter_00_Accumulator_ControlTick_Observer);
  Counter_00_ResetCounter_ControlTick.Register(&
  Counter_00_Accumulator_ControlTick_Observer);

  RampRolled_AudioTick.Register(&
  Counter_01_ResetCounter_AudioTick_Observer);

  Counter_01_CounterRolled_AudioTick.Register(&
  Counter_01_Accumulator_AudioTick_Observer);
  Counter_01_ResetCounter_AudioTick.Register(&
  Counter_01_Accumulator_AudioTick_Observer);
112
113   SineOsc_01_Reset_ControlTick.Register(&
      SineOsc_01_Phase_AudioTick_Observer);
114
115 // Resonant Low Pass trigger registrations have been removed
116
117   Level_00.init_Offset(&Level_00_Offset_Constant);
118 }
```

Code 7.8: Excerpts of the C++ code generated for triggers across two domains example.

On line 9 of Code 7.8, the type of the `RampRolled` trigger block is declared. The type is a `Trigger_MD_TriggerControlled`, where "MD" stands for multi-domain and "TriggerControlled" means a trigger is connected to `RampRolled`'s input rather than a signal or a switch. During the type declaration, the mutual exclusion conditions are passed to the constructor. The conditions are based on the synchronization policy between `ControlDomain` and `AudioDomain`, because `RampRolled` is triggered in `ControlDomain` and gets evaluate in `AudioDomain`.

On line 10, the trigger object is instantiated based on the type declared on line 9. The mutex associated with `ControlDomain` and `AudioDomain` is passed to the constructor. Next, is the declaration of a trigger observer for this variable (line 11). The observer object is instantiated with a callback method passed to its constructor. This method is called when the trigger gets triggered. This observer object is registered with any signal, switch, or trigger class this trigger observes. In this case, on line 104, the observer object of `RampRolled` is registered with the first counter's `CounterRolled` trigger. On line 108, the observer of the second counter's `ResetCounter` is registered with the `RampRolled` trigger.

On line 59, the state of the `RampRolled` trigger is evaluated in the `AudioTick()` function, since `RampRolled` is assigned to `AudioDomain`. If `CounterRolled` was triggered at some point prior to the evaluation of `RampRolled`'s state, the method `RampRolled` registered with `CounterRolled` would have been called and `RampRolled`'s state would have changed to a triggered state. If `RampRolled`'s state is evaluated as a triggered state, the method or methods registered with it are called. In this case, the method `ResetCounter` registered with `RampRolled` is called. `RampRolled` is re-armed once the method or methods registered with it complete executing. While executing these calls, all synchronization policies are respected, and the mutual exclusion schemes are executed by locking and unlocking the associated mutex.

All remaining triggers in the code follow the same code generation scheme. Each trigger gets instantiated using the appropriate helper class type and a corresponding observer is instantiated for it with a callback method. Next, the observer object is registered with associated triggers. These triggers call this callback method when their state is checked, and they happen to be in a triggered state.

7.3 Reactions

Just like module blocks, reaction blocks enclose stream expressions. However, there are fundamental differences in the way these two block types behave in Stride.

The main input port of a reaction block is always a switch block. This switch block has to be in an `on` state for the stream expressions enclosed in the reaction block to get evaluated. A reaction block can also access signals and switches declared directly outside of its scope without the need to connect these signals to dedicated property ports. A reaction block can be declared inside a module block, while a module block cannot be declared inside a reaction block. Code 7.9 shows the declaration structure of a reaction block in Stride.

```
reaction BlockName {
    ports:       [                       # Reaction's ports
        mainInputPort InputPort {        # Default input port
            block:  Switch
            meta:   "Built-in main input port."
        }
    ]
    blocks:      [                       # Reaction's internal blocks
            switch Switch {              # Default switch block
                default:    off
                rate:       InputPort.rate
                domain:     InputPort.domain
                meta:       "Built-in default switch."
            }
        ]
    streams:    [
                                         # Stream expressions
        ]
    meta:       "A reaction block."
}
```

Code 7.9: Default reaction block declaration.

The reaction block has a built-in switch block called `Switch`. By default, `Switch` gets its rate and domain from the block connected to the reaction block's main input port. The user can override the default behavior of `Switch` by replacing it with their own switch block declaration.

A reaction called `WrapPhase` was previously declared and used in the sine oscillator module, `SineOsc`, in Code 5.11. The stream expression enclosed in `WrapPhase` was evaluated only when the phase of the oscillator became greater than or equal to two pi.

Code 7.10 is an example where a reaction is used to double the frequency of a sine oscillator on every impulse generated by an impulse train generator.

The second instance of the impulse train generator (`ImpulseTrain`) generates impulses a 1Hz (line 26). This translates to an impulse every second. These impulses activate the `DoubleFrequency` reaction, causing the the frequency of the sine oscillator to double every second through the `Frequency` signal. The frequency of the sine oscillator is reset to its default value every 10 seconds, since the first instance of the impulse train generator is triggering the `Reset` trigger at 0.1Hz (line 24) and `Reset` is connected to the `reset` port of the `Frequency` signal (line 5).

```
 1 signal Frequency {
 2     default:    55.0
 3     rate:       AudioRate
 4     domain:     AudioDomain
 5     reset:      Reset
 6 }
 7
 8 trigger Reset {
 9     edge:       "Rising"
10     domain:     AudioDomain
11 }
12
13 switch Impulse {
14     default:    off
15     rate:       AudioRate
16     domain:     AudioDomain
17 }
18
19 reaction DoubleFrequency {
20     streams:    Frequency * 2.0 >> Frequency;
21     meta:       "Doubles the frequency."
22 }
23
24 ImpulseTrain ( frequency: 0.1 ) >> Reset;
25
26 ImpulseTrain ( frequency: 1 ) >> Impulse >> DoubleFrequency();
27
28 SineOsc ( frequency: Frequency ) >> Output;
```

Code 7.10: A reaction to double the frequency of an oscillator every second.

7.4 Attack/Decay Envelope in Stride

An attack/decay (AD) envelope is a great example to demonstrate how reactions, switches, and triggers are used in Stride to design interaction. This example also demonstrates how a state machine can be created in Stride using these three blocks.

The AD envelope module has an attack phase and a decay phase. It is triggered through its `trigger` port. The envelope module first goes through an attack phase. The attack phase lasts for the duration of the attack time. The attack phase is followed by a decay phase. The decay phase lasts for the duration of the decay time. The duration of each phase is controlled through signals connected to the `attackTime` and `decayTime` property ports of the envelope module. When the envelope completes its decay phase, it issues a trigger on its `completed` property port. The envelope is designed to switch to the attack phase when re-triggered while in the decay phase.

The Stride code for the AD envelope is shown in Code 7.11. The C++ template class generated from the Stride code is shown in Code 7.12.

```
module AD {
    ports:      [
        mainInputPort InputPort {
            block:      Input
        }
        mainOutputPort OutputPort {
            block:      Output
        }
        propertyInputPort AttackPort {
            name:       "attackTime"
            block:      AttackTime
            default:    0.125
            meta:       "Attack time in seconds."
        }
        propertyInputPort DecayPort {
            name:       "decayTime"
            block:      DecayTime
            default:    0.125
            meta:       "Decay time in seconds."
        }
        propertyInputPort TriggerPort {
            name:       "trigger"
            block:      Trigger
            default:    none
            meta:       "Triggers the AD envelope. Accepts a trigger or a switch."
        }
        propertyOutputPort CompletedPort {
            name:       "completed"
            block:      Completed
            meta:       "A trigger is generated on this port when the envelope has completed its decay phase."
        }
    ]
    blocks:     [
        signal Input {
            default:    0.0
            type:       OutputPort.type
            rate:       OutputPort.rate
            domain:     OutputPort.domain
        }
        signal Output {
            default:    0.0
            type:       OutputPort.type
            rate:       OutputPort.rate
            domain:     OutputPort.domain
        }
        signal AttackTime {
            default:    AttackPort.default
            rate:       AttackPort.rate
            domain:     AttackPort.domain
        }
        signal DecayTime {
            default:    DecayPort.default
            rate:       DecayPort.rate
            domain:     DecayPort.domain
        }
        trigger Trigger {
            edge:   "Rising"
            domain: OutputPort.domain
        }
        trigger Completed {
            edge:   "Rising"
            domain: OutputPort.domain
        }
        signal AttackSlope {
            default:    1.0 / ( AttackTime * OutputPort.rate )
            type:        OutputPort.type
            rate:        AttackPort.rate
            domain:      AttackPort.domain
        }
        signal DecaySlope {
            default:     - 1.0 / ( DecayTime * OutputPort.rate )
            type:        OutputPort.type
            rate:        DecayPort.rate
            domain:      DecayPort.domain
        }
        switch AttackPhase {
            default:     off
            rate:        0
            domain:      OutputPort.domain
        }
        switch DecayPhase {
            default:     off
            rate:        0
            domain:      OutputPort.domain
        }
        signal EnvelopeValue {
            default:     0.0
            type:        OutputPort.type
            rate:        0
            domain:      OutputPort.domain
        }
        reaction StartAttackPhase {
            streams:     on >> AttackPhase;
        }
        reaction SwitchToAttackPhase {
            streams:     [
                on >> AttackPhase;
                off >> DecayPhase;
            ]
        }
        reaction EnvelopeValueUpperLimit {
            stream:      [
                1.0 >> EnvelopeValue;
                off >> AttackPhase;
                on >> DecayPhase;
            ]
        }
        reaction EnvelopeValueLowerLimit {
            stream:      [
                0.0 >> EnvelopeValue;
                off >> DecayPhase;
            ]
        }
        reaction NextAttackValue {
            streams:     [
                EnvelopeValue + AttackSlope >> EnvelopeValue;
                [ EnvelopeValue , 1.0 ] >> GreaterOrEqual () >> EnvelopeValueUpperLimit();
        }
        reaction NextDecayValue {
            streams:    [
                EnvelopeValue + DecaySlope >> EnvelopeValue;
                [ EnvelopeValue , 0.0 ] >> LessOrEqual () >> [ EnvelopeValueLowerLimit(), Completed ] ;
        }
    ]
    streams:    [
        1.0 / ( AttackTime * OutputPort.rate ) >> AttackSlope;

        - 1.0 / ( DecayTime * OutputPort.rate ) >> DecaySlope;

        Trigger and not ( AttackPhase or DecayPhase ) >> StartAttackPhase();
        Trigger and DecayPhase >> SwitchToAttackPhase();
        AttackPhase >> NextAttackValue();
        DecayPhase >> NextDecayValue();
        Input * EnvelopeValue >> Output;
    ]
    meta:       "Attack/Decay envelope, triggered through the trigger port. If triggered while in the decay phase, the envelope will switch back to the attack phase."
}
```

Code 7.11: Attack/Decay envelope module in Stride.

```
template <class OutputDataType, class AttackTimeDataType, class DecayTimeDataType>
class AD {
public:
    AD(float outputRate) : OutputPort_Rate(outputRate) {
    }

    void process_OutputDomain(OutputDataType Input, OutputDataType *Output, stride::Trigger_State *Trigger, stride::Trigger *Completed, OutputDataType AttackSlope, OutputDataType DecaySlope, bool *AttackPhase, bool *DecayPhase, OutputDataType *EnvelopeValue) {
        if(Trigger->State() and not (*AttackPhase or *DecayPhase)) {
            reaction_StartAttackPhase(AttackPhase);
        }
        if(Trigger->State() and *DecayPhase) {
            reaction_SwitchToAttackPhase(AttackPhase, DecayPhase);
        }
        if (*AttackPhase) {
            reaction_NextAttackValue(AttackSlope, AttackPhase, DecayPhase, EnvelopeValue);
        }
        if (*DecayPhase) {
            reaction_NextDecayValue(Completed, DecaySlope, DecayPhase, EnvelopeValue);
        }
        *Output = Input * *EnvelopeValue;
    }

    void process_AttackPortDomain(AttackTimeDataType AttackTime, OutputDataType *AttackSlope){
        *AttackSlope = OutputDataType(1.0) / (OutputDataType(AttackTime) * OutputPort_Rate);
    }

    void process_DecayPortDomain(DecayTimeDataType DecayTime, OutputDataType *DecaySlope){
        *DecaySlope = OutputDataType(-1.0) / (OutputDataType(DecayTime) * OutputPort_Rate);
    }

    void init_AttackTime(AttackTimeDataType *AttackTime) {
        *AttackTime = AttackTimeDataType(0.125);
    }

    void init_DecayTime(DecayTimeDataType *DecayTime) {
        *DecayTime = DecayTimeDataType(0.125);
    }

    void init_AttackSlope(OutputDataType *AttackSlope) {
        AttackTimeDataType AttackTime;
        init_AttackTime(&AttackTime);
        *AttackSlope = OutputDataType(1.0) / (OutputDataType(AttackTime) * OutputPort_Rate);
    }

    void init_DecaySlope(OutputDataType *DecaySlope) {
        DecayTimeDataType DecayTime;
        init_DecayTime(&DecayTime);
        *DecaySlope = OutputDataType(-1.0) / (OutputDataType(DecayTime) * OutputPort_Rate);
    }

    void init_AttackPhase(bool *AttackPhase) {
        *AttackPhase = false;
    }

    void init_DecayPhase(bool *DecayPhase) {
        *DecayPhase = false;
```

```
    }

    void reaction_StartAttackPhase(bool *AttackPhase) {
        *AttackPhase = true;
    }

    void reaction_SwitchToAttackPhase(bool *AttackPhase, bool *
    DecayPhase) {
        *AttackPhase = true;
        *DecayPhase = false;
    }

    void reaction_EnvelopeValueUpperLimit(OutputDataType *
    EnvelopeValue, bool *AttackPhase,bool *DecayPhase) {
        *EnvelopeValue = OutputDataType(1.0);
        *DecayPhase = true;
        *AttackPhase = false;
    }

    void reaction_EnvelopeValueLowerLimit(OutputDataType *
    EnvelopeValue, bool *DecayPhase) {
        *EnvelopeValue = OutputDataType(0.0);
        *DecayPhase = false;
    }

    void reaction_NextAttackValue (OutputDataType AttackSlope, bool *
    AttackPhase, bool *DecayPhase, OutputDataType *EnvelopeValue) {
        *EnvelopeValue = *EnvelopeValue + AttackSlope;
        OutputDataType BundleConnector_00[2];
        BundleConnector_00[0] = *EnvelopeValue;
        BundleConnector_00[1] = 1.0;
        GreaterOrEqual_00.process_OutputDomain(BundleConnector_00, &
        GreaterOrEqual_00_Output);
        if (GreaterOrEqual_00_Output) {
            reaction_EnvelopeValueUpperLimit(EnvelopeValue,
            AttackPhase, DecayPhase);
        }
    }

    void reaction_NextDecayValue (stride::Trigger *Completed,
    OutputDataType DecaySlope, bool *DecayPhase, OutputDataType *
    EnvelopeValue) {
        *EnvelopeValue = *EnvelopeValue + DecaySlope;
        OutputDataType BundleConnector_00[2];
        BundleConnector_00[0] = *EnvelopeValue;
        BundleConnector_00[1] = 0.0;
        LessOrEqual_00.process_OutputDomain(BundleConnector_00, &
        LessOrEqual_00_Output);
        if (LessOrEqual_00_Output) {
            reaction_EnvelopeValueLowerLimit(EnvelopeValue,
                DecayPhase);
                Completed->Update(LessOrEqual_00_Output);
            }
        }

private:
    float OutputPort_Rate;

    using GreaterOrEqual_00_Type = GreaterOrEqual<OutputDataType,
    bool>;
    GreaterOrEqual_00_Type GreaterOrEqual_00;
    bool GreaterOrEqual_00_Output;
    using LessOrEqual_00_Type = LessOrEqual<OutputDataType, bool>;
    LessOrEqual_00_Type LessOrEqual_00;
    bool LessOrEqual_00_Output;
};
```

Code 7.12: C++ class generated from the Attack/Decay envelope module.

Multiple reactions, switches, and triggered are declared inside the AD envelope module. They are all interconnected in the stream expressions of the module.

The two triggers, `Trigger` and `Completed` are connected to the `trigger` and `completed` ports of the module respectively. The `Trigger` trigger is connected to two reactions, `StartAttackPhase` and `SwitchToAttackPhase`. The stream expressions enclosed in these two reactions are evaluated when `Trigger` is active and the `AttackPhase` switch and the `DecayPhase` switch have the correct state. These two switches represent the phase the AD envelope is in. The two phases are mutually exclusive. When the AD envelope is in one of these phases the stream expressions in the corresponding `NextAttackValue` or `NextDecayValue` reactions are evaluated. In these two reactions the `EnvelopeValue` signal is calculated. This signal is the multiplier by which the `Input` signal is multiplied to produce the `Output` signal. Two other reactions, `EnvelopeValueUpperLimit` and `EnvelopeValueLowerLimit`, which are also evaluated in the `NextAttackValue` or `NextDecayValue` reactions, check whether `EnvelopeValue` has reached its upper or lower limits. The latter reactions are also responsible for

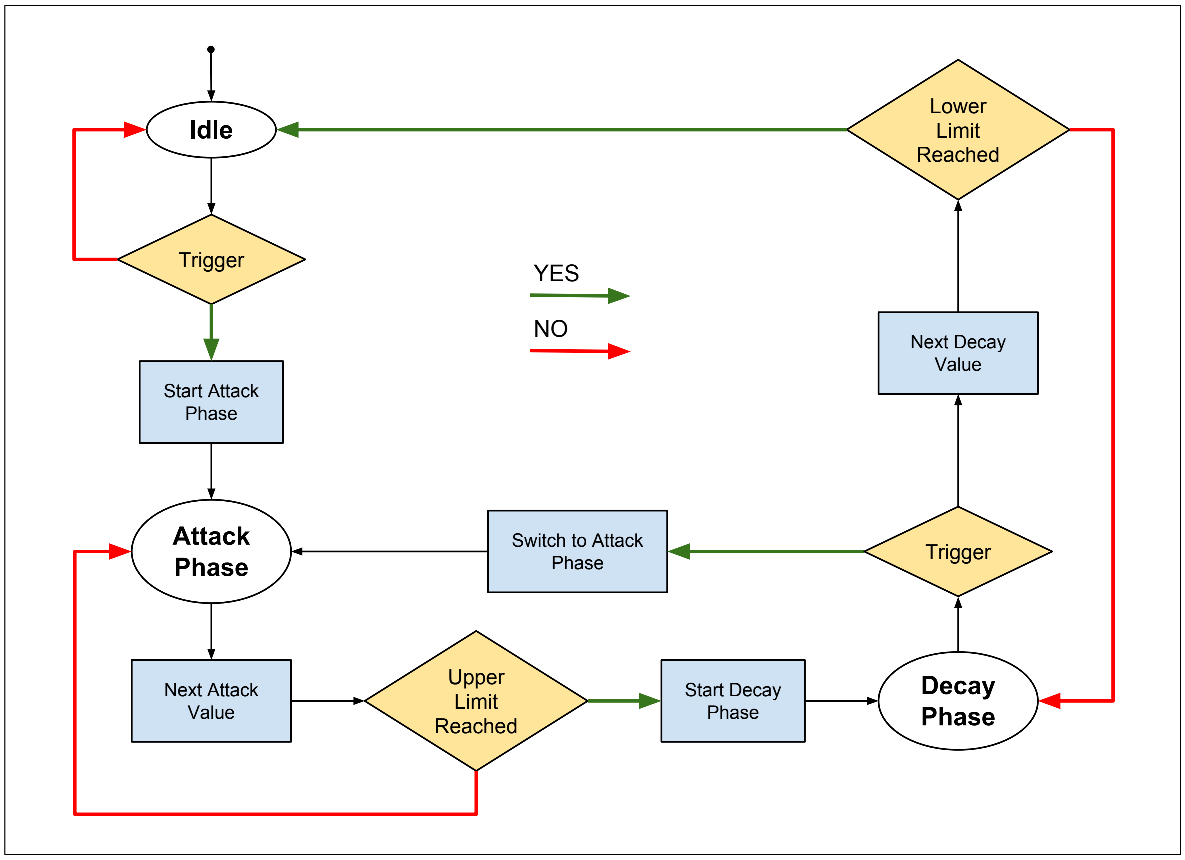


Figure 7.3: Attack/Decay envelope state machine.

updating the state of the envelope.

The connections between these reactions, switches, and triggers create a state machine. The envelope transitions from an "Idle" mode, where the AD envelope is neither in the attack phase nor in the decay phase, to an "Attack Phase" mode. From there it transitions to a "Decay Phase" mode and finally returning to the "Idle" mode awaiting a trigger. This state machine is shown in Figure 7.3.

The reaction blocks in the AD envelope module were translated to methods of the C++ template class shown in Code 7.12. These methods where then placed in the statements section of an "if" statement with the expression constructed using the triggers and switched connected to the input port of the reaction.

7.5 Summary

In this chapter we presented the switch, trigger, and reaction blocks in Stride. Through multiple examples we demonstrated how these blocks behave and how they could be used together to design interactions in Stride. We also demonstrated how these blocks could be used to create a state machine.

Chapter 8

Advanced Blocks in Stride

In this chapter we will present the advanced blocks in Stride and demonstrate their use.

8.1 The Buffer Block

A buffer block in Stride represent a First In First Out (FIFO) data buffer. A buffer block samples its input port at the rate assigned to it. It performs the sampling in the domain it is assigned to. The size of a buffer block is fixed and assigned at declaration. Signal and switch blocks can be connected to the input and output ports of a buffer block. The buffer block can be used to create delay lines, perform vector operations on data, and serve as an abstraction for data structures exchanged been hardware and software.

A buffer's data can be accessed using the indexing operator in Stride. The syntax is identical to accessing a block in a block bundle (subsection 4.2.1).

Code 7.1 shows the default declaration of a buffer block.

```
buffer BlockName {
    default:     0.0                 # Default value
    type:        auto                # Buffer's data type
    size:        1                   # Buffer's size
    rate:        PlatformRate        # Buffer's rate
    domain:      PlatformDomain      # Buffer's domain
    reset:       none                # Resets buffer to default value
    meta:        "A buffer block"    # Meta information
}
```

Code 8.1: Buffer block declaration.

8.1.1 Buffer Block as Delay Line

A buffer holds the previous values of a signal connected to its input, given the rates of the signal and the buffer match. The size of the buffer determines the length of the data retained by the buffer. In this scenario the buffer block represents a delay line whose memory can be tapped into by indexing the buffer block.

Code 8.2 is an example of buffer block used as a 3-sample delay line. The `Counter` module cyclically generates an integer valued ramp from `1` to `5`. The `Count` signal holds the most recent value generated by the `Counter` module. `Count` is connected to the input of the `Buffer` buffer block. `Buffer`'s size is set to `3` at declaration. Since `Count` and `Buffer` have the same rate and belong to the same domain, the values stored in `Buffer` are the past values of `Count`. The signal `DelayedCount` is connected to the output of the `Buffer` buffer block. Since `DelayedCount` has the same rate and is in the same domain as `Buffer`, the value of `DelayedCount` holds the previous values of `Count` delayed by three clock ticks of the domain they both belong to. The values of `Count` and `DelayedCount` are shown in Figure 8.1.

```
signal Count {
    rate:        ComputationRate
    domain:      ComputationDomain
}

buffer Buffer {
    default:     0
    size:        3
    rate:        ComputationRate
    domain:      ComputationDomain
}

signal DelayedCount {
    rate:        ComputationRate
    domain:      ComputationDomain
}

Counter (
    start:       1
    increment:   1
    roll:        5
)
>> Count
>> Buffer
>> DelayedCount;
```

Code 8.2: A buffer block used as a delay line.

The internal values of the `Buffer` buffer block and their relation to the signal blocks `Count` and `DelayedCount` are shown in Table 8.1. The index of the buffer block has the same value as the amount by which the input signal is delayed. While `Buffer[3]` represents the output of the buffer block and three samples delay, `Buffer[2]` is a tap into the buffer whose value represents the value of `Count` delayed by two samples.

On a given clock tick, the buffer block samples the signal connected to its input port

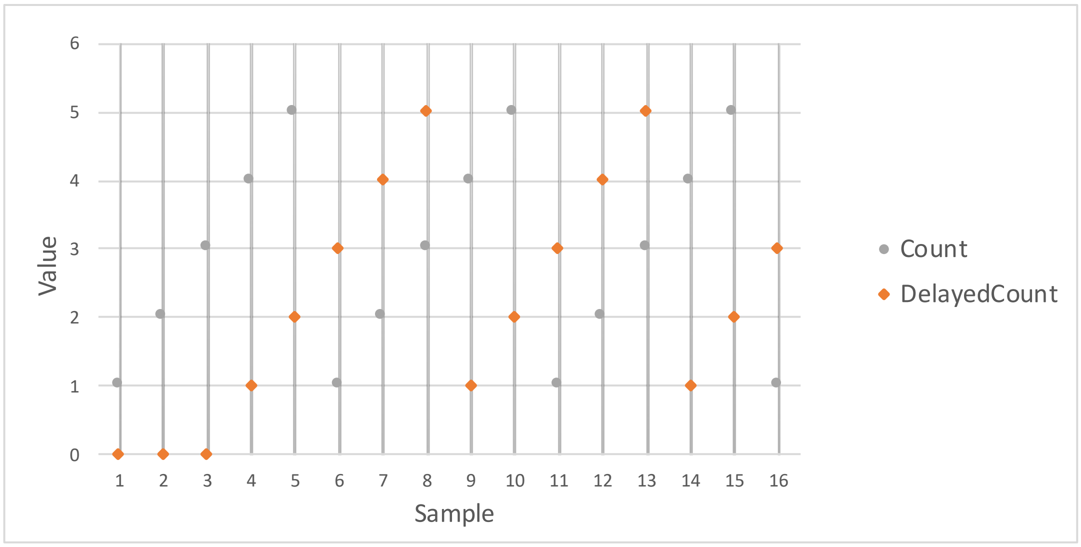


Figure 8.1: Values of `Count` and `DelayedCount`.

Clock Tick	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Count	1	2	3	4	5	1	2	3	4
_BufferInput	1	2	3	4	5	1	2	3	4
Buffer[1]	0	1	2	3	4	5	1	2	3
Buffer[2]	0	0	1	2	3	4	5	1	2
Buffer[3]	0	0	0	1	2	3	4	5	1
DelayedCount	0	0	0	1	2	3	4	5	1

Table 8.1: Values held by the buffer on every clock tick.

and stores its value in its internal memory. This memory is not accessible by the user and is represented by `_BufferInput` in Table 8.1. On the following clock tick, before the buffer block samples its input to replace the value of `_BufferInput` with the new sampled value, it pushes the value held in `_BufferInput` into `Buffer[1]` and the value of `Buffer[1]` into `Buffer[2]` and so forth.

8.1.2 Buffers and Hardware IO Abstraction

Data is most often exchanged between hardware and software through data structures known as buffers. The buffer block in Stride can be used to represent such data structures.

Operations on the data contained in a buffer can be performed either on the entire buffer or on each data point individually. To operate on the entire buffer, the name of the buffer is used when performing operations. Using the buffer name directs the code generator to use vector operations. To operate on each data point individually, a signal block is connected to the output of the buffer block and operations are performed on the signal. The signal and the buffer must have the same rate. The latter case is equivalent to performing per sample processing.

By working with buffers, the user gains the ability to direct the Stride code generator to use vector operations rather than operate on individual samples.

Sample Processing on Hardware IO

`AudioInBuffer` and `AudioOutBuffer` in Code 8.3 are two buffer block bundles. They abstract the left and right audio input and output data buffers. Software shares these buffers with hardware through a hardware abstraction software interface generally known as a driver.

```
buffer AudioInBuffer [2] {
    default:    0.0
    type:       "Real"
    size:       256
    rate:       AudioRate
    domain:     AudioDomain
    meta:       "Left and right audio input buffers from audio
    hardware."
}

buffer AudioOutBuffer [2] {
    default:    0.0
    type:       "Real"
    size:       256
    rate:       AudioRate
    domain:     AudioDomain
    meta:       "Left and right audio output buffers to audio
    hardware."
}

signal AudioIn [2] {
    default:    0.0
    type:       "Real"
    rate:       AudioRate
    domain:     AudioDomain
    meta:       "Left and right audio signals."
}

signal Mono {
    default:    0.0
    type:       "Real"
    rate:       AudioRate
    domain:     AudioDomain
    meta:       "Mono signal created from left and right audio
    signals."
}

AudioInBuffer >> AudioIn;

( AudioIn[1] + AudioIn[2] ) / 2.0 >> Mono;

Mono >> AudioOutBuffer;
```

Code 8.3: Per sample operation performed on data contained in buffer blocks.

In this example, the `AudioInBuffer` buffer bundle is connected to the `AudioIn` signal bundle. Arithmetic operations are performed on the two `AudioIn` signals to produce a signal represented by `Mono`. Since the arithmetic operations are performed directly on the signals and not the buffers, Stride performs the operations on a per sample basis and does not apply any vector operations. In this case the Stride code generator generates a for-loop and iterates over every data point of the two buffers to compute the `Mono` signal. Next, the resulting signal is sampled by the two buffers of the `AudioOutBuffer` buffer block bundle.

Vector Processing on Hardware IO

The same operation performed in Code 8.3 could be performed using vector processing as shown in Code 8.4. Although the outputs will be identical, the difference is in the generated code, where in the latter case the code is optimized for performance.

In Code 8.4, `Mono` is declared as a buffer block rather than a signal block. `Mono` has the same size and rate as the buffers in the `AudioInBuffer` buffer bundle. The arithmetic operations are performed on the buffers of the `AudioInBuffer` buffer bundle and stored into the `Mono` buffer. The Stride code generator would then attempt to generate code using methods capable of operating on vectors rather than samples to perform the arithmetic operations. This is only possible if such methods are available on the platform Stride is targeting.

The use of vector operations in code generation to generate performance optimized code is further discussed in the following subsection.

```
buffer AudioInBuffer [2] {
    default:     0.0
    type:        "Real"
    size:        256
    rate:        AudioRate
    domain:      AudioDomain
    meta:        "Left and right audio input buffers from audio
    hardware."
}

buffer AudioOutBuffer [2] {
    default:     0.0
    type:        "Real"
    size:        256
    rate:        AudioRate
    domain:      AudioDomain
    meta:        "Left and right audio output buffers to audio
    hardware."
}

buffer Mono {
    default:     0.0
    type:        "Real"
    size:        256
    rate:        AudioRate
    domain:      AudioDomain
    meta:        "Mono buffer calculated from the input buffers."
}

AudioInBuffer[1] + AudioInBuffer[2] / 2.0 >> Mono >> AudioOutBuffer;
```

Code 8.4: Vector operations on audio input and output buffers.

8.1.3 Buffers and Vector Operations

Buffers in Stride are yet another means available to the user to direct the Stride code generator to optimize for performance. Buffers can be used to perform vector operations on data. If the processor of the targeted platform can operate on multiple data simultaneously with a single instruction, buffers can be used to invoke this type of operations.

Multiple variants of the same module can exist in Stride. Each variant can be designed to operate on different block types connected to its ports or different data types being passed on those ports. This is equivalent to compile time polymorphism in object oriented languages through function overloading. To illustrate this, we will consider a module called `Offset`. The module is designed to offset a signal by some value. Two variants of the `Offset` module are shown in Code 8.5 and Code 8.6.

Both `Offset` modules have identical `ports`. The first difference appears in the type of the blocks connected to these ports. The first variant accepts connections from signal blocks at its main input and main output ports, while the second variant accepts connections from buffer blocks. The Stride code generator considers the type of the block connected to a port of a module and accordingly decides on the implementation it generates.

```
module Offset {
    ports:       [
        mainInputPort InputPort {
            block:       Input
        }
        mainOutputPort OutputPort {
            block:       Output
        }
        propertyInputPort OffsetPort {
            name:        "value"
            block:       OffsetValue
            default:     1.0
            meta:        "Offset value."
        }
    ]
    blocks:      [
        signal Input {
            default:     0.0
            type:        OutputPort.type
            rate:        OutputPort.rate
            domain:      OutputPort.domain
        }
        signal Output {
            default:    0.0
            type:       OutputPort.type
            rate:       OutputPort.rate
            domain:     OutputPort.domain
        }
        signal OffsetValue {
            default:    OffsetPort.default
            type:       OutputPort.type
            rate:       OffsetPort.rate
            domain:     OffsetPort.domain
        }
    ]
    streams:    [
        Input + OffsetValue >> Output;
    ]
    meta:       "Add an offset to a signal."
}
```

Code 8.5: A signal offsetting module.

```
module Offset {
    ports:      [
        mainInputPort InputPort {
            block:      Input
        }
        mainOutputPort OutputPort {
            block:      Output
        }
        propertyInputPort OffsetPort {
            name:       "value"
            block:      OffsetValue
            default:    1.0
            meta:       "Offset value."
        }
    ]
    blocks:     [
        buffer Input {
            default:    0.0
            type:       OutputPort.type
            size:       OutputPort.size
            rate:       OutputPort.rate
            domain:     OutputPort.domain
        }
        buffer Output {
            default:    0.0
            type:       OutputPort.type
            size:       OutputPort.size
            rate:       OutputPort.rate
            domain:     OutputPort.domain
        }
        signal OffsetValue {
            default:    OffsetPort.default
            type:       OutputPort.type
            rate:       OutputPort.rate / OutputPort.size
            domain:     OutputPort.domain
        }
    ]
    streams:    [
        Input >> _VectorOffset ( value: OffsetValue ) >> Output;
    ]
    meta:       "Add an offset to a buffer."
}
```

Code 8.6: A buffer offsetting module.

The second difference is in the rate and domain of the `OffsetValue` signal. In the first variant, the rate and the domain of `OffsetValue` are assigned the same values as the signal connected to the `value` port. However, in the second variant, the rate and the domain are derived from the signal connected to the main output port. If the Stride code generator is to perform vector processing, the offset value has to remain constant for that vector. The only way to guarantee this is to either have a constant block connected to the `value` port or have a signal whose rate is equal to the rate of the buffer divided by the buffer size connected to the main output port. In the latter case the `value` signal and the buffer have to be in the same domain.

The third difference is in the stream expressions in the `streams` property of the modules. In the first variant, the `Input` and `OffsetValue` are simply added together to calculate `Output`. In the second variant, a module called `_VecotrOffset` is used to perform this computation. `_VecotrOffset` is a Foreign Function Interface (FFI) module block. This FFI block wraps a function or multiple functions (to handle different data types) capable of performing optimized vector processing. These functions are available on the target platform for adding an offset to a vector.

To illustrate which of these two modules is used based on the Stride code written by the user, let us consider Code 8.7 and Code 8.8. Let us assume we are targeting a platform designed with a 32-bit ARM microcontroller with SIMD instruction support and a library to perform optimized vector operations on various data types.

Let us assume we have a stream of 16-bit signed integer data coming in from an ADC and the data needs to be biased (offset) in a computation domain available on the target platform. The data arrives to the domain in sets of eight values at a time.

```
constant BufferSize {
    value:  8
}

signal Input {
    type:   INT16
    rate:   ComputationRate
    domain: ComputationDomain
}

signal Output {
    type:   INT16
    rate:   ComputationRate
    domain: ComputationDomain
}

signal OffsetValue {
    type:   INT16
    rate:   ComputationRate
    domain: ComputationDomain
}

# Update OffsetValue

Input >> Offset ( value: OffsetValue ) >> Output;
```

Code 8.7: Adding an offset to a signal in Stride.

```
constant BufferSize {
    value:  8
}

buffer Input {
    size:   BufferSize
    type:   INT16
    rate:   ComputationRate
    domain: ComputationDomain
}

buffer Output {
    size:   BufferSize
    type:   INT16
    rate:   ComputationRate
    domain: ComputationDomain
}

signal OffsetValue {
    type:   INT16
    rate:   ComputationRate / BufferSize
    domain: ComputationDomain
}

# Update OffsetValue

Input >> Offset ( value: OffsetValue ) >> Output;
```

Code 8.8: Adding an offset to a buffer in Stride.

If the user chooses to perform this operation using signal blocks as shown in Code 8.7, the Stride code generator generates code similar to the one shown in Code 8.9, where a for-loop is generated to perform per sample processing.

If the user chooses to perform this operation using buffer blocks instead, as shown in Code 8.8, the Stride code generator generates code similar to the one shown in Code 8.10. A function called `arm_offset_q15` is used to perform a vector offset operation. This function utilized a SIMD addition instruction, that is able to add the offset to four 16-bit integer values simultaneously. By performing this operation using buffers, performance would improve by four times.

```cpp
#define BUFFER_SIZE             8

static q15_t input  = 0;
static q15_t output = 0;
static q15_t offsetValue;

void computationCallback ( ... ) {

    for (unsigned int i = 0; i < BUFFER_SIZE; i++) {
        // Get input
        // Update offsetValue
        output = input + offsetValue;
        // Use output
    }

}
```

Code 8.9: C++ code generated for offsetting a signal.

```cpp
#define BUFFER_SIZE             8

static q15_t input [BUFFER_SIZE];
static q15_t output [BUFFER_SIZE];
static q15_t offsetValue;

void computationCallback ( ... ) {

    // Get input array
    // Update offsetValue
    arm_offset_q15 (input, offsetValue, output, BUFFER_SIZE);
    // Use output array

}
```

Code 8.10: C++ code generated for offsetting a buffer.

8.2 The Loop Block

Loop blocks in Stride enable iterating over signals in a bundle block or data in a buffer block. Iterating over bundle and buffer blocks is possible without using the loop block. However, by using a loop block the process can be significantly simplified.

The default declaration of a `loop` block is shown in Code 8.11.

The block looks similar to a module block but has two additional properties called `onExecution` and `terminateWhen`. To make a valid loop declaration, all blocks declared within a loop have to belong to the same domain. The loop executes at the rate assigned to the `Input` signal bundle connected to the main input port of the block. By default, the rate of the `Input` signal bundle is assigned the rate of the signal connected to the main output port. When the loop executes, a trigger is generated on the `onExecution` property. Any trigger connected to this property port is triggered prior to the evaluation of the stream expressions in the `streams` property. A trigger called `Reset` is connected to this port by default and can be used to reset any internal block in the loop. When the loop starts executing, it suspends the rates of all blocks declared within it. That is, the loop treats all blocks as if their rates were set to 0 and run in reactive mode. The stream expressions keep executing until a trigger connected to the `terminateWhen` property port is trigger. By default, a trigger called `Done` is connected to the `terminateWhen` port. The loop can be terminated by attaching a logical expression to the input port of the `Done` trigger. The loop block has an additional internal signal block called `Index`. This signal can be used to iterate over the `Input` signal bundle. The `Index` signal's `reset` port is connected to the `Reset` trigger and gets reset when the loop starts executing.

```
loop BlockName {
    ports:           [          # Default ports of the loop
        mainInputPort InputPort {
            block:       Input
            meta:        ""
        }
        mainOutputPort OutputPort {
            block:       Output
            meta:        ""
        }
    ]
    blocks:          [          # Buffer's data type
        signal Input [InputPort.size] {
            rate:       OutputPort.rate
            domain:     OutputPort.domain
        }
        signal Output {
            default:    0.0
            rate:       0
            domain:     OutputPort.domain
            reset:      Reset
        }
        signal Index {
            default:    1
            rate:       0
            domain:     Output.domain
            reset:      Reset
        }
        trigger Done {
            edge:       "Rising"
            domain:     Output.domain
        }
        trigger Reset {
            domain:     Output.domain
        }
    ]
    onExecution:    Reset       # Trigger Output
                                # Triggered when the loop executes
    terminateWhen:  Done        # Trigger Input
                                # Stops the loop when triggered
    streams:        [           # Streams executed by the loop

        # ###
        # Stream expressions added here by the user
        # ###

        Index + 1 >> Index;
        [ Index , Input.size ] >>  Greater () >> Done;
    meta:           "Default loop block"        # Meta information
}
```

Code 8.11: Loop block declaration.

Code 8.12 is a loop block called `Sum` designed to calculate the sum of: multiple signals bundled together, signals in a signal bundle, or the data contained in a buffer.

```
loop Sum {
    ports:          [
        mainInputPort InputPort {
            block:      Input
            meta:       ""
        }
        mainOutputPort OutputPort {
            block:      Output
            meta:       ""
        }
    ]
    blocks:         [
        signal Input [InputPort.size] {
            rate:       OutputPort.rate
            domain:     OutputPort.domain
        }
        signal Output {
            default:    0.0
            rate:       0
            domain:     OutputPort.domain
            reset:      Reset
        }
        signal Index {
            default:    1
            rate:       0
            domain:     Output.domain
            reset:      Reset
        }
        trigger Done {
            edge:       "Rising"
            domain:     Output.domain
        }
        trigger Reset {
            domain:     Output.domain
        }
    ]
    onExecution:    Reset
    terminateWhen:  Done
    streams:        [
        Input[Index] + Output >>  Output;
        Index + 1 >> Index;
        [ Index , Input.size ] >>  Greater () >> Done;
    ]
}
```

Code 8.12: Sum loop in Stride.

In Code 8.13, the `Sum` loop is used to calculate the sum of all the signals in the `Inputs` signal bundle.

```
signal Inputs [4] {
    rate:   AudioRate
    domain: AudioDomain
}

signal InputsTotal {
    rate:   AudioRate
    domain: AudioDomain
}

Inputs >> Sum () >> InputsTotal;
```

Code 8.13: Summing signals in a bundle.

In Code 8.14, the `Sum` loop is used to calculate the sum of the data in the `Buffer` buffer block. The `Sum` loop is executed once every four clock ticks since the rate of the `Total` signal is assigned a clock rate 4 times slower than the default clock rate of the domain it is assigned.

By varying the rate of the signal connected to the main output port of a loop, it is possible to perform operations on the buffer with overlapping data from the `Count` signal. This can be used to perform overlap-add operations. This type of operation is common in digital signal processing[28].

The values held by the signals and buffer in Code 8.14 are shown in Table 8.2. The values of `Total` in boldface indicate when the `Sum` loop was executed, and its value was updated.

```
signal Count {
    default:    0
    rate:       ComputationRate
    domain:     ComputationDomain
}

```

```
buffer Buffer {
    default:     0
    size:        4
    rate:        ComputationRate
    domain:      ComputationDomain
}

signal Total {
    default:     0
    rate:        ComputationRate / 4
    domain:      ComputationDomain
}

Counter (
    start:       1
    increment:   1
    roll:        4
)
>> Count
>> Buffer
>> Sum ()
>> Total;
```

Code 8.14: Summing data in a buffer.

Clock Tick	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Count	1	2	3	4	1	2	3	4	1	2
Buffer[1]	0	1	2	3	4	1	2	3	4	1
Buffer[2]	0	0	1	2	3	4	1	2	3	4
Buffer[3]	0	0	0	1	2	3	4	1	2	3
Buffer[4]	0	0	0	0	1	2	3	4	1	2
Total	**0**	0	0	0	**10**	10	10	10	**10**	10

Table 8.2: Values of signals and the buffer at every clock tick.

Code 8.15 is a declaration of a module block called `Average`. The module can be used to calculate the average value of the signals in a signal bundle or the data contained in a buffer. The module uses the `Sum` loop internally.

```
module Average {
    ports:       [
        mainInputPort InputPort {
            block:       Input
            meta:        ""
        }
        mainOutputPort OutputPort {
            block:      Output
            meta:       ""
        }
    ]
    blocks:     [
        signal Input [InputPort.size] {
            rate:       OutputPort.rate
            domain:     OutputPort.domain
        }
        signal Output {
            rate:       Output.rate
            domain:     OutputPort.domain
        }
        signal Total {
            rate:       Output.rate
            domain:     OutputPort.domain
        }
    ]
    streams:    [
        Input >> Sum() >> Total;
        Total / InputPort.size >> Output;
    ]
}
```

Code 8.15: Average module block in Stride.

8.3 The Group Block

Signals and signal bundles can be grouped together. Grouping of signals and signal bundles is possible if they belong to the same domain, run at the same rate, and share the same reset trigger.

By grouping signals and signal bundles, the user directs the Stride code generator to treat signals in a group as a single entity in order to protect the integrity of the data carried by each signal in the group across domains. A group block declaration is shown in Code 8.16.

```
group BlockName {
    signals:   [ ]                         # Signals and signal bundles
    meta:      "Default group block"       # Meta information
}
```

Code 8.16: Group block declaration.

Code 8.17 is a resonant low pass filter module. The filter is implemented as a digital biquad[1] filter direct form I[29, chapter 6]. The coefficients of the filter are represented by the signal bundles `A` and `B` in the `blocks` property of the module. Signal bundles `A` and `B` are then grouped together in a group block called `FilterCoefficients`. Although the group block is not used in the stream expressions of the module, it plays a significant role in the generated code shown in Code 8.18.

The reason behind grouping the coefficient signal bundles is to make sure the two coefficient sets of the filter are updated together and not independently, especially if the coefficient update occurs in a domain different from the one where they are read and used by the filter. If one of the sets is updated and the update process of the second set is interrupted by another thread prior to completion, and the filter coefficients are used in this state to filter data, the output of the filter will be corrupted because the sets used by the filter do not correspond to the same cutoff frequency or quality factor.

In the generated code shown in Code 8.18, there is no reference made to the signal bundles `A` and `B`. The only reference to them is through the `FilterCoefficients` array. Since the array is not declared as a member of the class, reading and writing to it can be controlled using a synchronization policy to guarantee the integrity of the coefficients.

[1]Second order infinite impulse response

```
module ResonantLowPass {
    ports:          [
        mainInputPort InputPort {
            block:      Input
        }
        mainOutputPort OutputPort {
            block:      Output
        }
        propertyInputPort FrequencyPort {
            name:       "frequency"
            block:      Fc
            default:    1000.0
            meta:       "The frequency of the ResonantLowPass in Hz."
        }
        propertyInputPort QFactorPort {
            name:       "qFactor"
            block:      Q
            default:    0.7071
            meta:       "The quality factor of the ResonantLowPass."
        }
        propertyInputPort ResetPort {
            name:       "reset"
            block:      Reset
            default:    none
            meta:       "Resets the state of the resonant low pass
            filter. Accepts a switch or a trigger."
        }
    ]
    blocks:         [
        signal Input {
            default:    0.0
            type:       OutputPort.type
            rate:       OutputPort.rate
            domain:     OutputPort.domain
        }
        signal Output {
            default:    0.0
            type:       OutputPort.type
            rate:       OutputPort.rate
            domain:     OutputPort.domain
        }
        signal Fc {
            default:    FrequencyPort.default
            type:       FrequencyPort.type
            rate:       FrequencyPort.rate
            domain:     FrequencyPort.domain
        }
        signal Q {
            default:    QFactorPort.default
            type:       QFactorPort.type
            rate:       QFactorPort.rate
            domain:     QFactorPort.domain
        }
        trigger Reset {
            mode:       "Rising"
            domain:     ResetPort.domain
        }
        signal InputBuffer {
            default:    0.0
            size:       2
            type:       OutputPort.type
            rate:       OutputPort.rate
            domain:     OutputPort.domain
            reset:      Reset
        }
        signal OutputBuffer {
            default:    0.0
            size:       2
            type:       OutputPort.type
            rate:       OutputPort.rate
            domain:     OutputPort.domain
            reset:      Reset
        }
        signal Xn [2] {
            default:    0.0
            type:       OutputPort.type
            rate:       OutputPort.rate
            domain:     OutputPort.domain
            reset:      Reset
        }
        signal Yn [2] {
            default:    0.0
            type:       OutputPort.type
            rate:       OutputPort.rate
            domain:     OutputPort.domain
            reset:      Reset
        }
        signal K {
            default:    0.0
            type:       FrequencyPort.type
            rate:       FrequencyPort.rate
            domain:     FrequencyPort.domain
        }
        signal Norm {
            default:    0.0
            type:       OutputPort.type
            rate:       0
            domain:     [ FrequencyPort.domain, QFactorPort.domain ]
        }
        signal A[2] {
            default:     0.0
            type:        OutputPort.type
            rate:        0
            domain:      [ FrequencyPort.domain, QFactorPort.domain ]
        }
        signal B[3] {
            default:     0.0
            type:        OutputPort.type
            rate:        0
            domain:      [ FrequencyPort.domain, QFactorPort.domain ]
        }
        group FilterCoefficients {
            signals:     [ A , B ]
        }
    ]
    streams:         [
        3.14159265359 * Fc / OutputPort.rate >> Tan () >> K;

        1.0 / (1.0 + K / Q + K * K) >> Norm;
        K * K * Norm >> B[1];
        2.0 * B[1] >> B[2];
        B[1] >> B[3];
        2.0 * (K * K - 1) * Norm >> A[1];
        (1.0 - K / Q + K * K) * Norm >> A[2];

        InputBuffer >> Xn;
        OutputBuffer >> Yn;

        Input * B[1] + Xn[1] * B[2] + Xn[2] * B[3] - Yn[1] * A[1] -
        Yn[2] * A[2]  >> Output;

        Input >> InputBuffer;
        Output >> OutputBuffer;
    ]
    meta:            "Resonant low pass filter"
}
```

Code 8.17: Resonant low pass module in Stride.

```
template<class OutputDataType, class FrequencyDataType, class
QFactorDataType >
class ResonantLowPass {
public:
  ResonantLowPass(float outputRate) : OutputPort_Rate(outputRate) {
    Norm = OutputDataType(0.0);
  }

  void process_OutputDomain(OutputDataType Input, OutputDataType *
  Output, stride::Buffer<OutputDataType> *InputBuffer, stride::Buffer
  <OutputDataType> *OutputBuffer, OutputDataType Xn[], OutputDataType
```

```
   Yn[], OutputDataType FilterCoefficients[]) {
    InputBuffer->Read(Xn);
    OutputBuffer->Read(Yn);

    *Output = Input * FilterCoefficients[2] + Xn[0] *
    FilterCoefficients[3] + Xn[1] * FilterCoefficients[4] - Yn[0] *
    FilterCoefficients[0] - Yn[1] * FilterCoefficients[1];

    InputBuffer->Write(Input);
    OutputBuffer->Write(*Output);
  }

  void process_FrequencyPortDomain(FrequencyDataType Fc,
  FrequencyDataType *K ) {
    *K = std::tan(3.14159265359 * Fc / OutputPort_Rate);
  }

  void process_QFactorDomain(QFactorDataType Q, QFactorDataType *Q_)
  {
    *Q_ = Q;
  }

  void process_FrequencyPortDomain_QFactorPortDomain(QFactorDataType
  Q, FrequencyDataType K, OutputDataType FilterCoefficients[]) {
    Norm = 1.0 / (1.0 + K / Q + K * K);
    FilterCoefficients[2] = K * K * Norm;
    FilterCoefficients[3] = 2.0 *  FilterCoefficients[2];
    FilterCoefficients[4] =  FilterCoefficients[2];
    FilterCoefficients[0] = 2.0 * (K * K - 1) * Norm ;
    FilterCoefficients[1] = (1.0 - K / Q + K * K) * Norm;
  }

  void init_Fc(FrequencyDataType *Fc) {
    *Fc = FrequencyDataType(1000.0);
  }

  void init_Q(QFactorDataType *Q) {
    *Q = QFactorDataType(0.7071);
  }

  void init_Xn(OutputDataType Xn[]) {
    for (int i = 0; i < 2; i++) {
      Xn[i] = 0.0;
    }
}

  void init_Yn(OutputDataType Yn[]) {
    for (int i = 0; i < 2; i++) {
      Yn[i] = 0.0;
    }
  }

  void init_K(FrequencyDataType *K) {
    FrequencyDataType Fc;
    init_Fc(&Fc);
    *K = std::tan(3.14159265359 * Fc / OutputPort_Rate);
  }

  void init_FilterCoefficients (OutputDataType FilterCoefficients[])
  {
    QFactorDataType Q;
    init_Q(&Q);
    FrequencyDataType K;
    init_K(&K);
    Norm = 1.0 / (1.0 + K / Q + K * K);
    FilterCoefficients[2] = K * K * Norm;
    FilterCoefficients[3] = 2.0 *  FilterCoefficients[2];
    FilterCoefficients[4] =  FilterCoefficients[2];
    FilterCoefficients[0] = 2.0 * (K * K - 1) * Norm ;
    FilterCoefficients[1] = (1.0 - K / Q + K * K) * Norm;
  }

private:
  float OutputPort_Rate;

  OutputDataType Norm;
};
```

Code 8.18: C++ class generated from the resonant low pass module.

8.4 Summary

In this chapter we have presented a few advanced blocks in Stride that make it easier and more efficient to write code. These blocks also give the user more control over the generated code. `buffer` blocks can be used: as delay lines, to perform vector operations, or to abstract hardware buffers. `loop` blocks can be used to iterate over signals in signal bundles or data contained in buffers. Finally, `group` blocks allow for grouping signals together to preserve their integrity across domains.

Chapter 9

Stride

Stride is a programming language for real-time sound synthesis, processing, and interaction design. Stride is designed to abstract hardware and software architectures, thus simplifying the process of software and hardware integration, while giving the user control over the code generation process. These abstractions are defined in Stride systems which represent the inner workings of the target hardware and software, while exposing them in a simple and consistent manner across platforms.

The Stride language is part of the Stride environment which also encompasses the Stride integrated development environment (Stride IDE), the Stride interpreter, a target code generator, along with a set of Stride systems.

The Stride language presented in this dissertation is Stride version 1.0 and is licensed under the terms of the 3-clause BSD license. Copyright ©2017. The Regents of the University of California. All rights reserved.

Stride is available online at `http://StrideLang.org`.

9.1 Language Features

Stride is designed around the declarative and dataflow paradigms. The language has two constructs: block declarations and stream expressions. Stride allows both push (reactive) and pull programming, achieved by controlling the rate of signals. Signals are the fundamental building block of the language. The choice of making Stride declarative was to separate semantics from any particular implementation.

Stride borrows some of the best features of other programming languages like multi-channel expansion, single operator interfacing, multiple control rates, and per sample processing. Stride is also a self-documenting language.

The novel and unique aspect of Stride is making rates and hardware computation cores an intrinsic part of the language by introducing computation domains and synchronizing rates to them. This concept enables the distribution of synchronous and asynchronous computations, encapsulated within a single code block, to execute in different interrupt routines or threads on the hardware. The domains can potentially be part of a heterogeneous architecture. Rather than just being a unit generator and audio graph management tool, Stride enables the user to segment computations encapsulated in a unit generator during target code generation while handling it as a single unit in their code.

Stride enables its user to declare the frequency at which Stride expressions are evaluated and provides the user with the ability to control and fine tune the quality of the sounds they seek to generate or process. Stride also enables its user to control where expressions get evaluated and computed. This type of control is essential to optimizing code running on a resource-constrained device such as a microcontroller.

A user of Stride can also design interaction using `reactions`, an abstraction to handle asynchronous events. A reaction in Stride is similar to an "if" statements in procedural languages. However, in Stride, a reaction can enclose expressions executing in different domains, a feature that is not achievable by an "if" statement in a procedural language.

Stride is designed with embedded hardware in mind. Stride is platform agnostic and can target platforms like Bela[30], Axoloti[31], and OWL[32]. Stride is not restricted to a fixed number of building blocks or objects compared to the languages and tools used to target these platforms. Stride is designed to perform low-level signal processing functions and generate code that can run at native speed.

Although Stride is a textual language inheriting concepts from unit generator languages like Csound[33], SuperCollider[16][17] and ChucK[34], its basic construct is the streaming operator $\gg$ which makes it conceptually similar to dataflow languages like Pure Data[13] and Max[35]. Stride is not a dynamic unit generator graph manager, but rather a code generator like Faust[36]. Additionally, Stride is designed to facilitate both low-level signal processing algorithms and high-level constructs, like granular synthesis and frequency domain processing, using the same syntax.

A central consideration during the design of Stride was to treat the language as an interface and try to make it as "ergonomic" as possible. Two other criteria were read-ability and flow. That is, users should not need to read documentation to understand code and should be able to write code with as little friction as possible as the language works in a "physically intuitive" way similar to interfacing instruments, effects processors, amplifiers, and speakers in the physical world. To achieve this, features from popular and widely used general-purpose and domain specific languages were incorporated into Stride, like:

- Multichannel expansion from Nyquist[37]
- Single operator interface and multiple control rates from ChucK
- Per sample processing and discarding control flow statements from Faust
- Polychronous data-flow from synchronous and reactive programming languages like SIGNAL[38]
- Declarations and properties from Qt Meta Language[39]
- Slicing notation for indexing from Python
- Stream operator from C++

9.2 Stride Environment

The Stride environment comprises the Stride language, Stride systems, the Stride compiler, and the Stride IDE.

9.2.1 Stride Systems

A `system` in Stride is an abstraction of software and hardware target platforms. Stride system exposes the inner workings of a target computer and its peripherals to the user in an abstracted form. Stride does not only abstract the hardware but also the software architecture used to organize various processes. These abstractions grant the user full control of the underlying system without them having to know the implementation details.

Because of Stride's ability to abstract hardware, heterogeneous systems can be defined and consolidated under a single Stride system. This is achieved by abstracting the communication between the hardware and software platforms encompassing the heterogeneous system. In other words, different pieces of hardware (e.g. Arduino[1], Raspberry Pi[2], Desktop, etc.) can be grouped together to appear within Stride as a single system, as the communication between the devices is handled internally by Stride according to the system definition.

[1]`https://www.arduino.cc/` [accessed November 7, 2018]
[2]`https://www.raspberrypi.org/` [accessed November 7, 2018]

9.2.2 Stride Compiler

The Stride compiler is built out of a few independent modules. Any of these modules can be replaced in future versions of Stride. The compiler modules are the interpreter (lexical analyzer, parser, and intermediate code generator) and the target code generator. The compiler modules of Stride version 1.0 are shown in Figure 9.1.

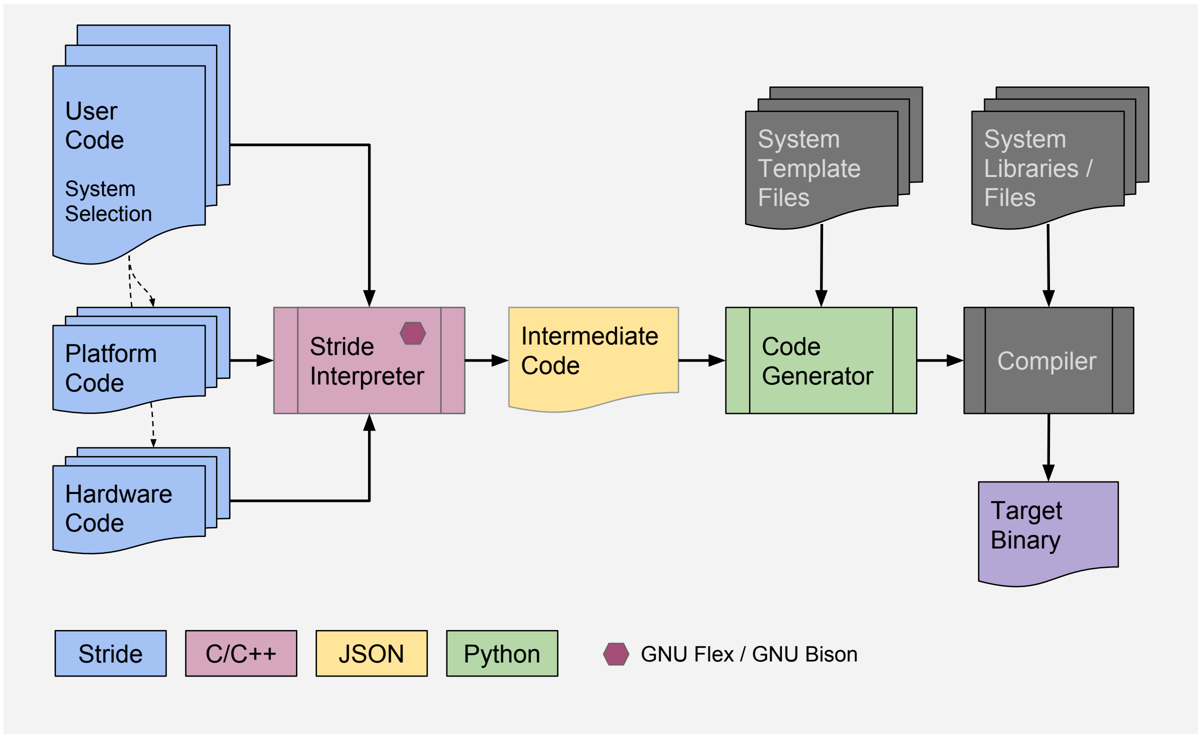


Figure 9.1: The Stride compiler.

The interpreter in written in C/C++ and outputs data in the JSON file format. The JSON file serves as an input to the target code generator which is responsible for generating and compiling code for target systems. This approach decouples the interpreter from the code generator.

Interpreter

Lexical Analysis

The first stage of the interpreter is the lexical analyzer. The lexical analyzer breaks down Stride code into tokens and passes them to the parser. The lexical analyzer is created and generated using GNU Flex[40], a fast lexical analyzer generator. The C files generated by Flex can be integrated into a parser.

The lexeme of Stride is shown in full in Appendix E.

Parsing

The second stage of the interpreter is the parser. The parser is generated using GNU Bison[41], a general-purpose parser generator. Bison interfaces well with Flex[42]. Using the Stride grammar, the parser generates an Abstract Syntax Tree (AST) based on the tokens passed to it by the lexical analyzer. When ready, the AST is passed to the intermediate code generator.

The grammar of Stride is shown in full in Appendix E

Intermediate Code Generation

The intermediate code generator takes in the AST generated by the parser and analyzes it by preforming multiple passes on the AST. The generator attempts to complete all the missing information in the user code (such as unassigned block properties during declaration) by following the rules of the language. Next, the generator expands all the stream expressions that need to undergo parallel expansion. The generator also replaces all expressions that evaluate to constant values with the evaluated constant values.

The output of the intermediate code generator is a JSON file. The JSON output file can then be used by any code generator to generate target code for any platform. The intermediate code generator of Stride version 1.0 is written in C++.

Code Generator

Target Code Generator

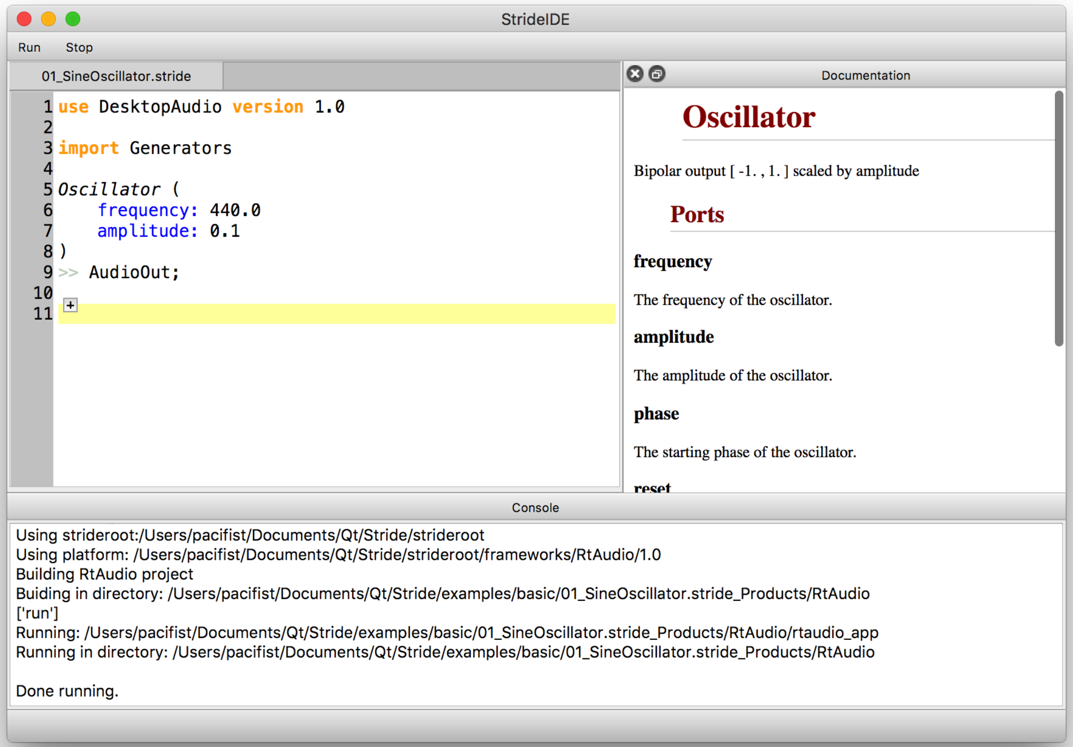
The Stride code generator takes in the JSON file generated by the interpreter and generates target code based on the Stride system specified by the user. The generator uses template files, libraries, and helper classes to generate the final source code. The code generator in Stride V1.0 is written in Python.

Deployment

Once the generated source code has been successfully compiled (or cross-compiled), Stride deploys the generated binary file on the target system.

9.3 Stride IDE

The Stride IDE is designed using the Qt framework[43] to support all three major operating systems including Windows, macOS, and Linux. A snapshot of the IDE is shown in Figure 9.2



```
use DesktopAudio version 1.0

import Generators

Oscillator (
    frequency: 440.0
    amplitude: 0.1
)
>> AudioOut;
```

Oscillator

Bipolar output [-1. , 1.] scaled by amplitude

Ports

frequency

The frequency of the oscillator.

amplitude

The amplitude of the oscillator.

phase

The starting phase of the oscillator.

reset

```
Using strideroot:/Users/pacifist/Documents/Qt/Stride/strideroot
Using platform: /Users/pacifist/Documents/Qt/Stride/strideroot/frameworks/RtAudio/1.0
Building RtAudio project
Buiding in directory: /Users/pacifist/Documents/Qt/Stride/examples/basic/01_SineOscillator.stride_Products/RtAudio
['run']
Running: /Users/pacifist/Documents/Qt/Stride/examples/basic/01_SineOscillator.stride_Products/RtAudio/rtaudio_app
Running in directory: /Users/pacifist/Documents/Qt/Stride/examples/basic/01_SineOscillator.stride_Products/RtAudio

Done running.
```

Figure 9.2: The Stride integrated development environment.

The IDE has a multi-tab code editor with a built-in autocomplete feature and syntax highlighter. The editor also marks and highlights errors related to syntax and grammar.

The IDE has a console window where build information, errors, and warnings generated during code generation are displayed.

The IDE also has a built-in web engine to display HTML[3] documentation pertaining to some advanced blocks in Stride. The HTML documentation is directly rendered from the Stride code and includes the information provided in the `meta` property of blocks.

9.4 Stride Syntax

Stride has two syntactic constructs: Block Declarations and Stream Expressions.

A block is declared through a block declaration statement. A block is assigned a `type` and a unique `name`. Block names must start with a capital letter and can include digits and the underscore character. A block's properties are part of the declaration and define its behavior. Properties of a block can only be assigned at declaration. Some properties are required, some are optional, while others are assumed if they are not explicitly assigned. In the latter case, the assumptions are made based on the rules of the language and the assignments to the other properties of the block.

Blocks in Stride are divided into two groups: Basic and Advanced. Basic blocks constitute the core types of the language. Advanced blocks encapsulate basic blocks and stream expressions to perform specific functions. Basic blocks can be declared as a bundle while advanced ones cannot.

Blocks in Stride are connected in stream expressions with the stream operator $>>$. All stream expressions are evaluated at least once from left to right and in the top-down

[3]Hypertext Markup Language

order in which they appear in the user code.

Stream expressions undergo parallel expansion. The expansion depends on the constituent blocks of the stream expression and the values assigned to the properties of the blocks. The expansion is resolved from left to right starting with leftmost element in a stream expression.

The syntax to declare blocks, bundles, and stream expressions is shown in section 4.2 and Appendix E.

The following subsection will cover all block declarations in Stride and the definitions of their properties. The subsequent subsection will provide examples of stream expressions constructed using blocks and block bundles to demonstrate parallel expansion.

9.4.1 Basic Blocks

The following block types make up the core building blocks of the language.

Constant

Declaration

```
constant BlockName {
    value:  none
    type:   auto
    domain: ConstantDomain
    meta:   ""
}
```

Code 9.1: Constant block declaration.

Definitions

`value`	*The value of the constant block.* Port accepts a value of the `AlphaNumericTypeClass` class. Default value is `none` but an assignment is required.
`type`	*The type of the constant block.* Port accepts an item of the `DataTypeList` list. Default value is `auto`. If not set, the value is derived from the type of the `default` port value.
`domain`	*The domain of the constant block.* Port accepts an item of the `DomainTypeList` list. Default value is `ConstantDomain`.
`meta`	*A description tag.* Port is `StringType`. Default value is an empty string.

Shorthand Declaration

```
# Declaration of an integer constant called IntegerConstant
1 >> IntegerConstant

# Declaration of a real constant called RealConstant
1.0 >> RealConstant;

# Declaration of a string constant called StringConstant
"This is a constant String." >> StringConstant;
```

Code 9.2: Shorthand constant block declarations.

Signal

Declaration

```
signal BlockName {
    default:     0.0
    type:        auto
    rate:        auto
    domain:      PlatformDomain
    reset:       none
    meta:        ""
}
```

Code 9.3: Signal block declaration.

Definitions

`default`	*The default value of the signal block.* Port accepts a value of the `AlphaNumericTypeClass` class. Default value is 0.0.
`type`	*The type of the signal block.* Port accepts an item of `DataTypeList` list. Default value is `auto`. If not set, the value is derived from the type of the `default` port value.
`rate`	*The rate of the signal block.* Port accepts a value of the `NumericTypeClass` class. Default value is `auto`. If a rate is not specified, the rate is set to the rate of the domain the signal is assigned to. If the rate is set to a non-zero real or integer value, the signal operates in sample-and-hold mode. If the rate is set to zero, the signal operates in reactive mode.

`domain`	*The domain of the signal block.*
	Port accepts an item of the `DomainTypeList` list.
	Default value is `PlatformDomain`.
`reset`	*Resets the signal block to its default value.*
	Port is an input `TriggerBlockType` type.
	Default value is `none`.
`meta`	*A description tag.*
	Port is `StringType`.
	Default value is an empty string.

Switch

Declaration

```
switch BlockName {
    default:     off
    rate:        auto
    domain:      PlatformDomain
    reset:       none
    meta:        ""
}
```

Code 9.4: Switch block declaration.

Definitions

`default`	*The default value of the switch block.*
	Port is `BooleanType`.

`rate`	*The rate of the switch block.* Port accepts a value of the `NumericTypeClass` class. Default value is `auto`. If a rate is not specified, the rate is set to the rate of the domain the signal is assigned to. If the rate is set to a non-zero real or integer value, the signal operates in sample-and-hold mode. If the rate is set to zero, the signal operates in reactive mode.
`domain`	*The domain of the switch block.* Port accepts an item of the `DomainTypeList` list. Default value is `PlatformDomain`.
`reset`	*Resets the switch block to its default value.* Port is an input `TriggerBlockType` type. Default value is `none`.
`meta`	*A description tag.* Port is `StringType`. Default value is an empty string.

Buffer

Declaration

```
buffer BlockName {
    default:     0.0
    type:        auto
    size:        none
    rate:        auto
    domain:      PlatformDomain
    reset:       none
    meta:        ""
}
```

Code 9.5: Buffer block declaration.

Definitions

`default` *The default values of the buffer block.*

Port accepts a value of the `AlphaNumericTypeClass` class.

Default value is 0.0.

`type` *The type of the buffer block.*

Port accepts an item of the `DataTypeList` list.

Default value is `auto`. If not set, the value is derived from the type of the `default` port value.

`size` *The size of the buffer block.*

Port is `UnsignedIntegerType`.

Default value is `none` but an assignment is required.

`rate` *The rate of the buffer block.*

Port is a value of `NumericTypeClass` class.

Default value is `auto`. If a rate is not specified, the rate is set to the rate of the domain the buffer is assigned to.

If the rate is set to a non-zero real or integer value, the buffer operates in sample-and-hold mode. If the rate is set to zero, the buffer operates in reactive mode.

`domain` *The domain of the buffer block.*

Port accepts an item of the `DomainTypeList` list.

Default value is `PlatformDomain`.

`reset` *Resets the buffer block to its default value.*

Port is an input `TriggerBlockType` type.

Default value is `none`.

`meta` *A description tag.*

Port is `StringType`.

Default value is an empty string.

Trigger

Declaration

```
trigger BlockName {
    edge:         "Rising"
    domain:       PlatformDomain
    meta:         ""
}
```

Code 9.6: Trigger block declaration.

Definitions

`edge` *The edge type that triggers the trigger when controlled by a switch block.*

Port is an item of `EdgeTypeList` list.

Default value is `"Rising"`.

Default items of `EdgeTypeList` are `"Rising"`, `Falling`, or `"Both"`.

`domain` *The domain of the trigger block.*

Port accepts an item of the `DomainTypeList` list.

Default value is `PlatformDomain`.

When triggered, the trigger is `on` for one clock cycle of this domain before it is rearmed.

`meta` *A description tag.*

Port is `StringType`.

Default value is an empty string.

9.4.2 Block Bundles

All basic blocks in Stride can be declared as bundles. Blocks of a bundle share the same property assignments.

Declaration

```
blockType BundleName [SIZE] {
    ...
}
```

Code 9.7: Bundle declaration.

Definitions

`SIZE` *The size of the bundle.*

port is `UnsignedIntegerType`.

9.4.3 Advanced Blocks

The following block types make up the advanced blocks of the language.

Module

Declaration

```
module BlockName {
    ports:          [ ]
    blocks:         [ ]
    constraints:    [ ]
    streams:        [ ]
    meta:           ""
}
```

Code 9.8: Module block declaration.

Definitions

`ports` *List of port declarations.*

Port accepts an item of the `ModulePortsList` list.

Items of `ModulePortsList` are `mainInputPort`, `mainOutputPort`, `propertyInputPort`, and `propertyOutputPort`.

`blocks`	*List of internal block declarations.* Port accepts an item of the `ModuleBlocksList` list. Items of `ModuleBlocksList` are `signal`, `switch`, `constant`, `trigger`, and `reaction`.
`constraints`	*The constraints of the module block.* Port is `StreamListType` type. Default value is `[ ]` (an empty stream list).
`streams`	*The streams of the module block.* Port is `StreamListType` type. Default value is `[ ]` (an empty stream list).
`meta`	*A description tag.* Port is `StringType`. Default value is an empty string.

Reaction

Declaration

```
reaction BlockName {
    ports:          [ ]
    blocks:         [ ]
    streams:        [ ]
    meta:           ""
}
```

Code 9.9: Reaction block declaration.

Definitions

`ports`	*List of port declarations.* Port accepts an item of the `ReactionPortsList` list. Items of `ReactionPortsList` are `mainInputPort`, `mainOutputPort`, `propertyInputPort`, and `propertyOutputPort`.
`blocks`	*List of internal block declarations.* Port accepts an item of the `ReactionBlocksList` list. Items of `ReactionBlocksList` are `signal`, `switch`, `constant`, `trigger`, `module`, `loop`, and `reaction`.
`streams`	*The streams of the reaction block.* Port is `StreamListType` type. Default value is `[ ]` (an empty stream list).
`meta`	*A description tag.* Port is `StringType`. Default value is an empty string.

Loop

Declaration

```
loop BlockName {
    ports:          [ ]
    blocks:         [ ]
    constraints:    [ ]
    onExecution:    none
    terminateWhen:  none
    streams:        [ ]
    meta:           ""
}
```

Code 9.10: Loop block declaration.

Definitions

`ports`	*List of port declarations.* Port accepts an item of the `LoopPortsList` list. Items of `LoopPortsList` are `mainInputPort`, `mainOutputPort`, `propertyInputPort`, and `propertyOutputPort`.
`blocks`	*List of internal block declarations.* Port accepts an item of the `LoopBlocksList` list. Items of `LoopBlocksList` are `signal`, `switch`, `constant`, `trigger`, and `module`.
`constraints`	*The constraints of the loop block.* Port is `StreamListType` type. Default value is `[ ]` (an empty stream list).
`onExecution`	*Trigger output. Triggers when the loop executes.*

Port is an output `TriggerBlockType` type.

Default value is `none`.

`terminateWhen` *Trigger input. Terminates the loop when triggered.*

Port is an input `TriggerBlockType` type.

Default value is `none`.

`streams` *The streams of the loop block.*

Port is `StreamListType` type.

Default value is `[ ]` (an empty stream list).

`meta` *A description tag.*

Port is `StringType`.

Default value is an empty string.

Group

Declaration

```
group BlockName {
    signals:    [ ]
    meta:       ""
}
```

Code 9.11: Group block declaration.

Definitions

`signals` *List of block declarations.*

Port accepts an item of `GroupBlocksList` list.

Items of `GroupBlocksList` are `signal` and `switch`.

`meta`	*A description tag.*
	Port is `StringType`.
	Default value is an empty string.

9.4.4 Stream Expressions

The following examples demonstrate how stream expressions are resolved and undergo parallel expansion.

Signals and Bundles

The following stream expression examples cover connections between signals and bundles.

Signal to Signal

Code 9.12 is an example of a direct signal to signal connection. The main output port of the `Input` signal is connected to the main input port of the `Output` signal. The resulting graph is shown in Figure 9.3.

```
signal Input {}
signal Output {}

Input >> Output;
```

Code 9.12: Signal to signal connection.

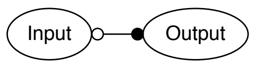


Figure 9.3: Signal to signal connection.

Signal to Bundle

Code 9.13 is an example of a direct signal to bundle connection. The main output port of the `Input` signal is connected the main input ports of the two signals that make up the `Output` bundle. The resulting graph is shown in Figure 9.4.

```
signal Input {}
signal Output [2] {}

Input >> Output;
```

Code 9.13: Signal to bundle connection.

The long form of the same code is shown in Code 9.14 where the `Input` signal is individually connected to the signals of the `Output` bundle, `Output[1]` and `Output[2]` respectively.

```
signal Input {}
signal Output [2] {}

Input >> Output[1];
Input >> Output[2];
```

Code 9.14: Expanded signal to bundle connection.

Bundle to Bundle

Code 9.15 is an example of a direct bundle to bundle connection. Both bundles have the same size. The main output ports of the two signals that make up the `Input`

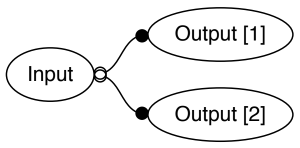


Figure 9.4: Signal to bundle connection.

bundle are connected to the main input ports of the two signals that make up the `Output` bundle respectively. The resulting graph is shown in Figure 9.5.

```
signal Input [2] {}
signal Output [2] {}

Input >> Output;
```

Code 9.15: Bundle to bundle connection of same size.

The long form of the same code is shown in Code 9.16, where the signals of the `Input` bundle, `Input[1]` and `Input[2]`, are individually connected to the signals of the `Output` bundle, `Output[1]` and `Output[2]` respectively.

```
signal Input [2] {}
signal Output [2] {}

Input[1] >> Output[1];
Input[2] >> Output[2];
```

Code 9.16: Expanded bundle to bundle connection of same size.

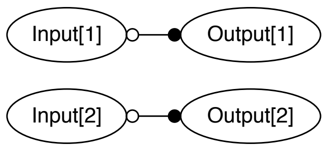


Figure 9.5: Bundle to bundle connection of same size.

Code 9.17 is an example of a direct bundle to bundle connection. The bundles have different sizes. The size of the bundle to the right of the stream operator is a multiple of the size of the one to the left. In this case, the connection between the signals of the `Input` bundle alternate with the signals of the `Output` bundle. The resulting graph is shown in Figure 9.6.

```
signal Input [2] {}
signal Output [4] {}

Input >> Output;
```

Code 9.17: Bundle to bundle connection where the size of one is a multiple of the other.

The expanded form of the same code is shown in Code 9.18.

```
signal Input [2] {}
signal Output [4] {}

Input[1] >> Output[1];
Input[2] >> Output[2];
Input[1] >> Output[3];
Input[2] >> Output[4];
```

Code 9.18: Expanded bundle to bundle connection where the size of one is a multiple of the other.

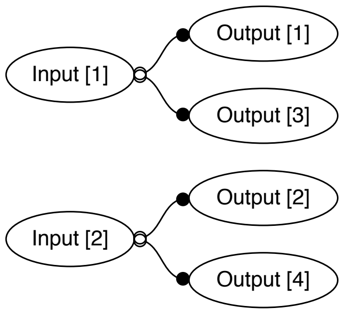


Figure 9.6: Bundle to bundle connection where the size of one is a multiple of the other.

Signals, Bundles, and Modules

The following stream expressions cover connections between signals and bundles with modules placed between them.

Signal to Module to Signal

Code 9.19 is an example of a signal connected to a module that is in turn connected to another signal. The main output port of the `Input` signal is connected to the main input port of the `Level` module and the main output port of the `Level` module is connected to the main input port of the `Output` signal. The resulting graph is shown in Figure 9.7.

```
signal Input {}
signal Output {}

Input >> Level (gain: 0.1) >> Output;
```

Code 9.19: Signal to module to signal connection.

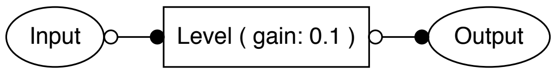


Figure 9.7: Signal to module to signal connection.

Bundle to Modules to Bundle

Code 9.20 is an example of a bundle connected to a module that is in turn connected to another bundle. The main output ports of the signals in the `Input` bundle are connected to the main input ports of two `Level` modules.

```
signal Input [2] {}
signal Output [2] {}

Input >> Level (gain: 0.1) >> Output;
```

Code 9.20: Implicit expansion of a second module driven by the size of the `Input` bundle.

Although a single `Level` module appears in the stream expression, two instance of `Level` are generated by the Stride code generator, since there are two signals on the left side of the stream operator at the input of the module and `Level` accepts a single signal at its main input port.

The main output ports of the two `Level` modules instances are connected to the main input ports of the two signals of the `Output` bundle. The resulting graph is shown in Figure 9.8.

The expanded version of Code 9.20 is shown in Code 9.21.

```
signal Input [2] {}
signal Output [2] {}

Input[1] >> Level (gain: 0.1) >> Output[1];
Input[2] >> Level (gain: 0.1) >> Output[2];
```

Code 9.21: Expanded bundle to module to bundle connection.

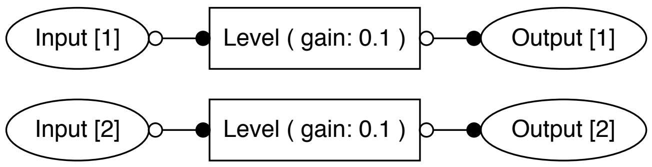


Figure 9.8: Bundle to modules to bundle connection with implicit expansion of a second module driven by the size of the `Input` bundle.

In Code 9.22 two `Level` modules are explicitly declared, through port expansion, by connecting a bundle of constants to the `gain` property port of the module. The resulting graph is shown in Figure 9.9.

```
signal Input [2] {}
signal Output [2] {}

Input >> Level (gain: [0.1, 0.3]) >> Output;
```

Code 9.22: Explicit declaration of two modules.

The expanded version of Code 9.22 is shown in Code 9.23.

```
signal Input [2] {}
signal Output [2] {}

Input[1] >> Level (gain: 0.1) >> Output[1];
Input[2] >> Level (gain: 0.3) >> Output[2];
```

Code 9.23: Expansion of bundle to modules to bundle connection.

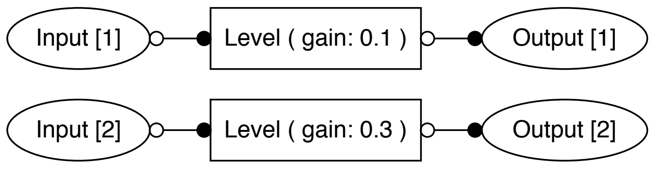


Figure 9.9: Bundle to modules to bundle connection with explicit declaration of two modules.

In Code 9.24 the size of the `Output` bundle is a multiple of both the size of the `Input` bundle and the number of `Level` module instances.

Since the expansion of the stream expressions is driven from the left side, the main output of the first `Level` module is connected to the inputs of the first and third signals in the `Output` bundle and the main output of the second `Level` module is connected to the second and fourth signals of the `Output` bundle. The resulting graph is shown in Figure 9.10.

```
signal Input [2] {}
signal Output [4] {}

Input >> Level (gain: [0.1, 0.3]) >> Output;
```

Code 9.24: Bundle to modules to bundle connection with different sizes.

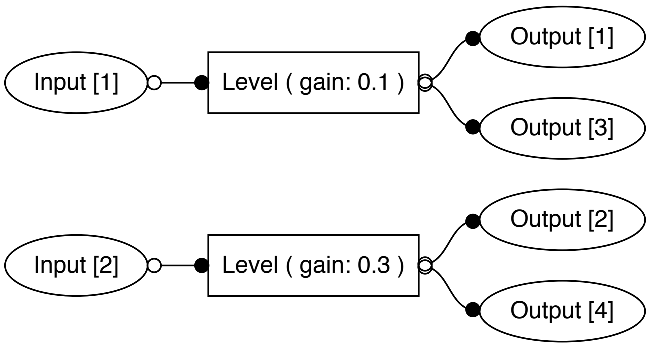


Figure 9.10: Bundle to modules to bundle connection with different sizes.

If the size of the `Input` bundle is doubled in Code 9.24, as shown in Code 9.25, the result would be the generation of four `Level` module instances. The resulting graph is shown in Figure 9.11.

```
signal Input [4] {}
signal Output [4] {}

Input >> Level (gain: [0.1, 0.3]) >> Output;
```

Code 9.25: Implicit and explicit expansion of modules.

Code 9.26 is an example of a bundle connected to an `Add` module. `Add` accepts a signal bundle of size two at its main input port. The module's main output is a signal and is connected to the input of the `Output` signal. The resulting graph is shown in Figure 9.12.

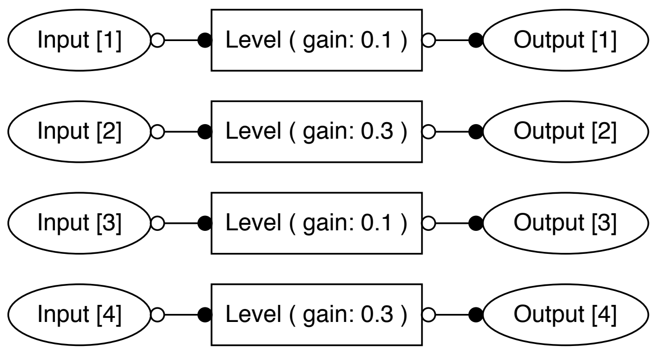


Figure 9.11: Implicit and explicit expansion of modules.

```
signal Input [2] {}
signal Output {}

Input >> Add () >> Output;
```

Code 9.26: Bundle to multi-input module to signal connection.

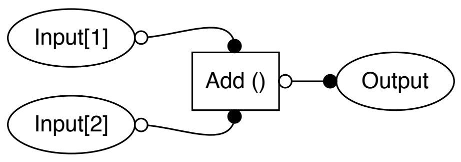


Figure 9.12: Bundle to multi-input module to signal connection.

If the `Output` signal in Code 9.26 is replaced by a bundle of size two, the main output of the `Add` module gets connected to the main input of the two signals of the `Output` bundle as shown in Code 9.27. The resulting graph is shown in Figure 9.13.

```
signal Input [2] {}
signal Output [2] {}

Input >> Add () >> Output;
```

Code 9.27: Bundle to multi-input module to bundle connection.

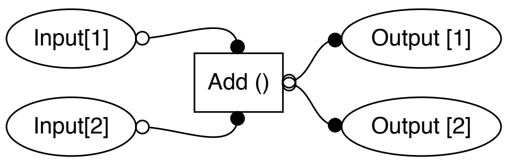


Figure 9.13: Bundle to multi-input module to bundle connection.

Code 9.28 and Code 9.29 are examples of two different modules, `Level` and `Add`, appearing between two bundles with different and similar sizes respectively. The modules are implicitly and explicitly expanded driven by the size of the `Input` bundle. The expression results in four `Level` modules and two `Add` modules. The resulting graphs are shown in Figure 9.14 and Figure 9.15 respectively.

```
signal Input [4] {}
signal Output [2] {}

Input >> Level (gain: [0.1, 0.3]) >> Add () >> Output;
```

Code 9.28: Implicit and explicit expansion of multiple modules between bundles of different sizes.

```
signal Input [4] {}
signal Output [4] {}

Input >> Level (gain: [0.1, 0.3]) >> Add () >> Output;
```

Code 9.29: Implicit and explicit expansion of multiple modules between bundles of the same size.

Module to Bundle

Code 9.30 is an example of a module connected to a bundle of size two. The main output of the `Oscillator` module is connected to the main input ports of the two signals in the `Output` bundle. The resulting graph is shown in Figure 9.16.

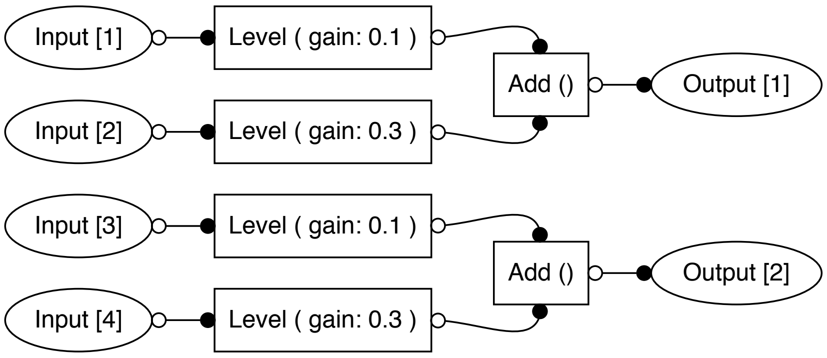


Figure 9.14: Implicit and explicit expansion of multiple modules between bundles of different sizes.

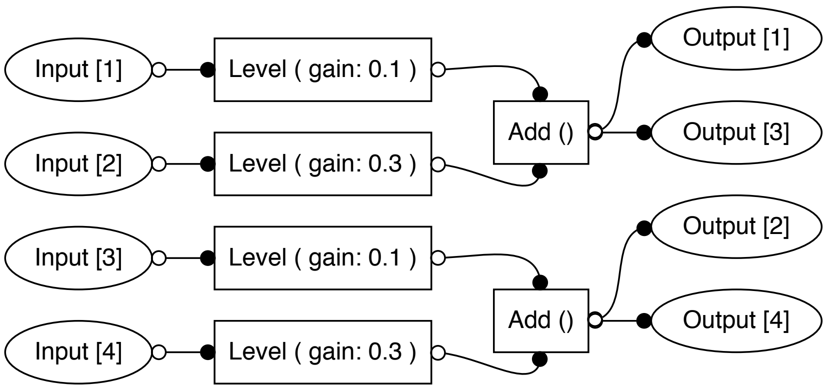


Figure 9.15: Implicit and explicit expansion of multiple modules between bundles of the same size.

```
signal Output [2] {}

Oscillator (frequency: 220.0) >> Output;
```
Code 9.30: Single module connected to a bundle.

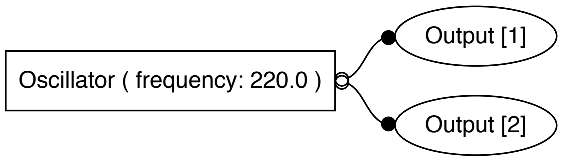


Figure 9.16: Single module connected to a bundle.

Modules to Bundle

Code 9.31 is an example of two module that are explicitly declared through port expansion and connected to a bundle of size two. The main output of each `Oscillator` module is connected to the main input of each signal of the `Output` bundle respectively. The resulting graph is shown in Figure 9.17.

```
signal Output [2] {}

Oscillator (frequency: [220.0, 440.0]) >> Output;
```
Code 9.31: Two modules connected to a bundle of size two.

The expanded version of Code 9.31 is shown in Code 9.32.

```
signal Output [2] {}

Oscillator (frequency: 220.0) >> Output[1];
Oscillator (frequency: 440.0) >> Output[2];
```
Code 9.32: Expanded form of two modules connected to a bundle of size two.

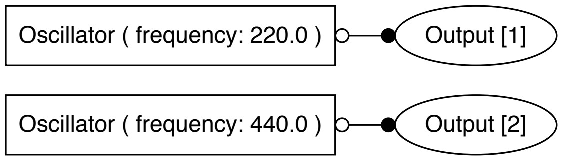

Oscillator (frequency: 220.0)
Output [1]
Oscillator (frequency: 440.0)
Output [2]

Figure 9.17: Two generators connected to two outputs.

Modules to Module to Bundle

Code 9.33 is an example where the main outputs of two `Oscillator` modules are connected to the main input ports of an `Add` module, whose main output is connected to the main input of the two signals of the `Output` bundle. The resulting graph is shown in Figure 9.18.

```
signal Output [2] {}

Oscillator (frequency: [220.0, 440.0]) >> Add () >> Output;
```

Code 9.33: Two modules connected to another module and then to a bundle of size two.

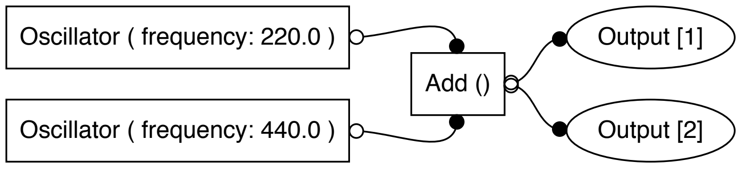

Oscillator (frequency: 220.0)
Oscillator (frequency: 440.0)
Add ()
Output [1]
Output [2]

Figure 9.18: Two modules connected to another module and then to a bundle of size two.

9.5 Summary

In this chapter, we presented the Stride environment comprising the Stride language, code generator, and the IDE. We also presented the formal declaration of blocks and defined their properties. We also covered the parallel expansion of stream expressions through multiple examples.

Chapter 10

Conclusion

This dissertation presented Stride, a language for sound synthesis, processing, and interaction design. The language is part of the Stride environment which also comprises a compiler and an integrated development environment.

This dissertation makes multiple contributions to the field of computer music especially when it comes to targeting resource-constrained microcontroller-based embedded systems for real-time sound synthesis and processing applications.

10.1 Summary

Prior to designing a new language for sound synthesis, processing, and interaction design to target resource-constrained microcontroller-based embedded system, we considered some of the most popular music programming languages as potential candidates for the task. We evaluated them against the specifications set forth by the research questions posed by this dissertation. Faust emerged as a potential candidate because of its capability to generate efficient C++ code that could be used to target a microcontroller. Although many of the programming languages did not meet the specifications, we noted some of their features, in order to adopt them into a new language, if we were to design one.

We took a closer look at Faust and identified some of its limitations and shortcomings when it came to its fixed approach to computation rates and its ability to distribute computations across various processes. We observed Faust followed a fixed concurrency model that is not ideal for microcontrollers, especially when running bare metal. Although Faust is able to generate code optimized for vector processing, it cannot use optimized libraries designed for specific target devices. This is due to the lack of facilities to add foreign functions to access external libraries through an API.

While researching Faust, we established that giving the user more control over the code generator would result in extremely efficient and optimized target code. By allowing the user to specify the rate at which computations occur and specify the thread where computations are made would result in significant performance improvements. We also established that having a flexible concurrency model built into the language would allow the user to achieve real-time performance on resource-constrained systems running concurrent threads, while preserving data integrity.

With these observations in mind, we designed a new language (Stride) with a declarative syntax to allow the user to control its code generator. The language was designed with only two constructs: block declarations and stream expressions. Through a set of basic examples, we demonstrated the improvements in efficiency that could be achieved on a microcontroller-based audio development platform. We presented various schemes to produce efficient code and we measured and compared the improvements to a baseline.

Next, we introduced the core building block of Stride: The `signal` block. We discussed its behavior based on its rate and domain assignments by the user. Through the design of a sine oscillator with frequency control, we demonstrated how the rate and domain assignments can influence the code generator. We also introduced the `module` block in Stride, which encapsulates block declarations and stream expressions to perform a specific function. We discussed how rate and domain assignments propagate from blocks declared outside the module to blocks declared inside of it. By using signal and module blocks, we demonstrated the information propagation mechanism by performing synchronous and asynchronous frequency modulation in Stride.

Next, we shifted our focus to present the user-controlled concurrency model built into Stride. We demonstrated how the user could define mutual exclusion schemes and concurrency policies to achieve the performance and optimization they desire. We then focused on presenting how this flexible user-controlled concurrency model was made possible through the generation of stateless C++ template classes that would accommodate any requirement set forth by the user without having to generate custom C++ classes for each concurrency scenario.

Next, we presented the `switch` block, the `trigger` block, and the `reaction` block in Stride. Through multiple examples we demonstrated the behavior of these blocks and how they could be used to design interaction in Stride. We also demonstrated how these blocks could be used to design a state machine.

Next, we presented some of the advanced blocks in Stride that make writing code easier. The advanced blocks are the `buffer` block, the `loop` block, and the `group` block. We also presented how these blocks give the user more control over the code generator.

Finally, we presented the Stride environment, which comprises the Stride language, the compiler, and the integrated development environment. We discussed the architecture of Stride and presented some of the tools used to design it. Next, we presented the formal declarations of blocks and defined their properties. We also covered the parallel expansion of stream expressions through multiple example.

10.2 Discussion

Although Stride is designed with resource-constrained devices in mind, the language can target general-purpose computers and heterogenous systems alike. This is possible due to multiple novel approaches Stride takes which were covered in detail in this dissertation.

With only two syntactic constructs, Stride meets all of the specifications that were set forth prior to its creation. Making the language declarative facilitated many of its goals. Declarative entities in the language can be connected using a single operator, thus simplifying the interface. Parallel expansion of entities and interfaces is achieved through bundles and is handled automatically by the code interpreter. Static allocation of entities is the default allocation method carried out by the interpreter. Dynamic allocation happens through block types that allow the construction and destruction of entities. By default, Stride performs computation on a per sample basis unless otherwise stated by the user. Through rates and the use of buffers the user controls the code generator, allowing the use of vector operations rather than operating on individual samples, thus making computations more efficient. By assigning rates and domains to signals, the user can control the synchronization of data and computations. Asynchronous events are handled through a special entity in the language, known as a reaction, capable of triggering the computation of expressions distributed in various threads. Signals at different rates can be seamlessly connected to each other in Stride, making Stride a multi-rate signal processing language. Since Stride is declarative, hardware drivers, software libraries, and real-time operating systems can be abstracted and presented to the user through a common interface.

Users of Stride can design unit generators and processors. The computations enclosed in such units can be designed to be evaluated at various rates and distributed across multiple threads which might be running on different devices. This is possible thanks to a robust concurrency model designed into the language, which the user controls by declaring and defining policies between threads sharing memory. The implemented concurrency model works because of a novel approach of generating stateless C++ template classes that represent the unit generators and processors. Variables which hold state are declared when instances of the stateless templates are instantiated. The methods of these classes are then invoked to operate on these variables in the threads specified by the user once the proper concurrency directives are met in order to protect the integrity of the data carried by these variables.

Stride separates semantics from implementation. Stride code is simply a collection of declarations coupled together using a single operator. Users' code simply represents their intent rather than a specific implementation. This makes it possible to use the same Stride code to target any device, as long as the device hardware is abstracted in Stride. Individual device abstractions can be combined to create heterogenous systems by declaring and defining Stride systems. These systems can also abstract the communication between devices and allow for seamless connection between signals declared on device specific domains.

Interaction design in Stride is abstracted through triggers and reactions. This abstraction allows for swapping any interaction with the target to trigger any event declared in Stride. Events are contained within a reaction. A single reaction may result in the evaluation of expressions distributed across multiple domains. This is possible because the Stride interpreter generates the necessary triggers and notifications necessary to propagate information between the domains.

10.3 Future Work

Currently, Stride is at a proof-of-concept development stage where many of its concepts can be successfully demonstrated. Considerable effort is required to fully implement all the concepts presented in this dissertation and to make the Stride compiler stable and "production ready". A library of modules must also be written to support basic synthesis and signal processing tasks.

One of the abstractions that should be fully defined and built into Stride is a type class system that is common in purely functional programming languages like Haskell. Although Stride is a strongly typed languages and types are strictly checked when connections between entities are made, formally adding type classes can simplify the declaration of modules and enhance the polymorphic capabilities of Stride.

Dynamic allocation of entities has been thoroughly examined but not yet fully implemented. Dynamically creating and destroying entities in Stride could happen by defining new block types that are capable of constructing and destructing other blocks.

Further abstractions can be added to Stride and its interpreter, which could be further improved by building code analysis functionality into it that could assist the user with optimizations.

A graph analysis tool that could analyze all interconnected signals and clusters them into groups that could be distributed across various domains available on a system would be of great value.

A graph visualization tool would also be beneficial to users. We envision the tool for graphically rendering related stream expressions to visually display the data flow as well as assign colors to signals based on their domain assignment to indicate where they are evaluated.

Finally, adding debugging and data monitoring blocks into Stride could be extremely useful.

Appendix A

Faust DSP and Generated Code

This appendix contains a set of Faust DSP code and the generated C++ code using the Faust online compiler. The compiler was at version 2.3.4 at the time compilation. The code has been compiled with the language set to C++ and the architecture set to Linux. The compiler options were *"-scal -ftz 0"*. The compiler is available online at `http://faust.grame.fr/onlinecompiler/`.

This appendix also contains a Faust template file for the Bela platform.

A.1 Resonant Low Pass with Constant Arguments

The following code represents a resonant low pass filter with constant arguments. The code

A.1.1 Faust DSP Code

```
import("stdfaust.lib");

// Cutoff Frequency
ctFreq = 500;
// Q Factor
q = 5;
// Gain
gain = 1;

// Resonant Low Pass
process = fi.resonlp(ctFreq,q,gain);
```

Code A.1: Faust resonant low pass filter with constant arguments.

A.1.2 C++ Generated Code

```
/* ------------------------------------------------------------
name: "RLP_Const"
Code generated with Faust 2.3.4 (http://faust.grame.fr)
Compilation options: -scal -ftz 0
------------------------------------------------------------ */

#ifndef  __mydsp_H__
#define  __mydsp_H__

#ifndef FAUSTFLOAT
#define FAUSTFLOAT float
#endif

#include <math.h>

float mydsp_faustpower2_f(float value) {
return (value * value);
}

#ifndef FAUSTCLASS
#define FAUSTCLASS mydsp
#endif

class mydsp : public dsp {

private:

int fSamplingFreq;
```

```
float fConst0;
float fConst1;
float fConst2;
float fConst3;
float fConst4;
float fRec0[3];

public:

void metadata(Meta* m) {
    m->declare("filters.lib/name", "Faust Filters Library");
    m->declare("filters.lib/version", "0.0");
    m->declare("maths.lib/author", "GRAME");
    m->declare("maths.lib/copyright", "GRAME");
    m->declare("maths.lib/license", "LGPL with exception");
    m->declare("maths.lib/name", "Faust Math Library");
    m->declare("maths.lib/version", "2.0");
    m->declare("name", "myFaustProgram");
}

virtual int getNumInputs() {
    return 1;
}
virtual int getNumOutputs() {
    return 1;
}
virtual int getInputRate(int channel) {
    int rate;
    switch (channel) {
        case 0: {
            rate = 1;
            break;
        }
        default: {
            rate = -1;
            break;
        }
    }
    return rate;
}
virtual int getOutputRate(int channel) {
    int rate;
    switch (channel) {
        case 0: {
            rate = 1;
            break;
        }
        default: {
            rate = -1;
            break;
        }
    }
    return rate;
}

static void classInit(int samplingFreq) {
}

virtual void instanceConstants(int samplingFreq) {
    fSamplingFreq = samplingFreq;
    fConst0 = tanf((1570.79639f / min(192000.0f, max(1000.0f, float(
    fSamplingFreq)))));
    fConst1 = (1.0f / fConst0);
    fConst2 = (1.0f / (((fConst1 + 0.200000003f) / fConst0) + 1.0f));
    fConst3 = (((fConst1 + -0.200000003f) / fConst0) + 1.0f);
    fConst4 = (2.0f * (1.0f - (1.0f / mydsp_faustpower2_f(fConst0))))
    ;
}

virtual void instanceResetUserInterface() {
}

virtual void instanceClear() {
    for (int l0 = 0; (l0 < 3); l0 = (l0 + 1)) {
        fRec0[l0] = 0.0f;
    }
}

virtual void init(int samplingFreq) {
    classInit(samplingFreq);
    instanceInit(samplingFreq);
}
virtual void instanceInit(int samplingFreq) {
    instanceConstants(samplingFreq);
    instanceResetUserInterface();
    instanceClear();
}

virtual mydsp* clone() {
    return new mydsp();
}
virtual int getSampleRate() {
    return fSamplingFreq;
}

virtual void buildUserInterface(UI* ui_interface) {
    ui_interface->openVerticalBox("myFaustProgram");
    ui_interface->closeBox();
}

```

```
virtual void compute(int count, FAUSTFLOAT** inputs, FAUSTFLOAT** outputs) {
    FAUSTFLOAT* input0 = inputs[0];
    FAUSTFLOAT* output0 = outputs[0];
    for (int i = 0; (i < count); i = (i + 1)) {
        fRec0[0] = (float(input0[i]) - (fConst2 * ((fConst3 * fRec0[2]) + (fConst4 * fRec0[1]))));
        output0[i] = FAUSTFLOAT((fConst2 * (fRec0[2] + (fRec0[0] + (2.0f * fRec0[1])))));
        fRec0[2] = fRec0[1];
        fRec0[1] = fRec0[0];
    }
}

};

#endif
```

Code A.2: Generated C++ code for resonant low pass filter with constant arguments.

A.2 Resonant Low Pass with Variable Arguments

The following code represents a resonant low pass filter with variable arguments. The arguments are controlled with horizontal sliders.

A.2.1 Faust DSP Code

```
import("stdfaust.lib");

// Cutoff Frequency Horizontal Slider
ctfreq = hslider("cutoffFrequency",500,50,10000,0.01);
// Q Factor Horizontal Slider
q = hslider("q",5,1,30,0.1);
// Gain Horizontal Slider
gain = hslider("gain",1,0,1,0.01);

// Resonant Low Pass
process = fi.resonlp(ctFreq,q,gain);
```

Code A.3: Faust resonant low pass filter with variable arguments.

A.2.2 C++ Generated Code

```
/* ------------------------------------------------------------
name: "RLP_Var"
Code generated with Faust 2.3.4 (http://faust.grame.fr)
Compilation options: -scal -ftz 0
------------------------------------------------------------ */

#ifndef  __mydsp_H__
#define  __mydsp_H__

#ifndef FAUSTFLOAT
#define FAUSTFLOAT float
#endif

#include <math.h>

float mydsp_faustpower2_f(float value) {
    return (value * value);
}

#ifndef FAUSTCLASS
#define FAUSTCLASS mydsp
#endif

class mydsp : public dsp {

private:

FAUSTFLOAT fHslider0;
FAUSTFLOAT fHslider1;
int fSamplingFreq;
float fConst0;
FAUSTFLOAT fHslider2;
float fRec0[3];

public:

void metadata(Meta* m) {
    m->declare("filters.lib/name", "Faust Filters Library");
    m->declare("filters.lib/version", "0.0");
    m->declare("maths.lib/author", "GRAME");
    m->declare("maths.lib/copyright", "GRAME");
    m->declare("maths.lib/license", "LGPL with exception");
    m->declare("maths.lib/name", "Faust Math Library");
    m->declare("maths.lib/version", "2.0");
    m->declare("name", "myFaustProgram");
}

```

```
virtual int getNumInputs() {
    return 1;
}
virtual int getNumOutputs() {
    return 1;

}
virtual int getInputRate(int channel) {
    int rate;
    switch (channel) {
        case 0: {
            rate = 1;
            break;
        }
        default: {
            rate = -1;
            break;
        }
    }
    return rate;
}
virtual int getOutputRate(int channel) {
    int rate;
    switch (channel) {
        case 0: {
            rate = 1;
            break;
        }
        default: {
            rate = -1;
            break;
        }
    }
    return rate;
}

static void classInit(int samplingFreq) {
}

virtual void instanceConstants(int samplingFreq) {
    fSamplingFreq = samplingFreq;
    fConst0 = (3.14159274f / min(192000.0f, max(1000.0f, float(
    fSamplingFreq))));
}

virtual void instanceResetUserInterface() {
    fHslider0 = FAUSTFLOAT(1.0f);
    fHslider1 = FAUSTFLOAT(5.0f);
    fHslider2 = FAUSTFLOAT(500.0f);
}
```

```

virtual void instanceClear() {
    for (int l0 = 0; (l0 < 3); l0 = (l0 + 1)) {
        fRec0[l0] = 0.0f;

    }
}

virtual void init(int samplingFreq) {
    classInit(samplingFreq);
    instanceInit(samplingFreq);
}
virtual void instanceInit(int samplingFreq) {
    instanceConstants(samplingFreq);
    instanceResetUserInterface();
    instanceClear();
}

virtual mydsp* clone() {
    return new mydsp();
}
virtual int getSampleRate() {
    return fSamplingFreq;
}

virtual void buildUserInterface(UI* ui_interface) {
    ui_interface->openVerticalBox("myFaustProgram");
    ui_interface->addHorizontalSlider("cutoffFrequency", &fHslider2,
    500.0f, 50.0f, 10000.0f, 0.00999999978f);
    ui_interface->addHorizontalSlider("gain", &fHslider0, 1.0f, 0.0f,
     1.0f, 0.00999999978f);
    ui_interface->addHorizontalSlider("q", &fHslider1, 5.0f, 1.0f,
    30.0f, 0.100000001f);
    ui_interface->closeBox();
}

virtual void compute(int count, FAUSTFLOAT** inputs, FAUSTFLOAT**
outputs) {
    FAUSTFLOAT* input0 = inputs[0];
    FAUSTFLOAT* output0 = outputs[0];
    float fSlow0 = (1.0f / float(fHslider1));
    float fSlow1 = tanf((fConst0 * float(fHslider2)));
    float fSlow2 = (1.0f / fSlow1);
    float fSlow3 = (((fSlow0 + fSlow2) / fSlow1) + 1.0f);
    float fSlow4 = (float(fHslider0) / fSlow3);
    float fSlow5 = (1.0f / fSlow3);
    float fSlow6 = (((fSlow2 - fSlow0) / fSlow1) + 1.0f);
    float fSlow7 = (2.0f * (1.0f - (1.0f / mydsp_faustpower2_f(fSlow1
    ))));
    for (int i = 0; (i < count); i = (i + 1)) {
        fRec0[0] = (float(input0[i]) - (fSlow5 * ((fSlow6 * fRec0[2])
         + (fSlow7 * fRec0[1]))));
        output0[i] = FAUSTFLOAT((fSlow4 * (fRec0[2] + (fRec0[0] +
        (2.0f * fRec0[1])))));
        fRec0[2] = fRec0[1];
        fRec0[1] = fRec0[0];
    }
}

};

#endif
```

Code A.4: Generated C++ code for resonant low pass filter with variable arguments.

A.3 Bela Template Code for Faust

The following code is the platform definition file for the Bela platform. The source file could be found on Faust's GitHub page at `https://github.com/grame-cncm/faust` [accessed November 7, 2018].

The snapshot shown here is commit 386ec90c5776c8324239bcdeadc95c5eabbd7fdc of `bela.cpp` file.

The comments in the file, including the copyright information, have been removed or modified to fit the page margins. The code is copyright of Centre National de Creation Musicale and Augmented Instruments Laboratory.

```
#ifndef __FaustBela_H__
#define __FaustBela_H__

#include <cstddef>
#include <string>
#include <math.h>
#include <strings.h>
#include <cstdlib>
#include <Bela.h>
#include <Utilities.h>
```

```
#include "faust/gui/JSONUIDecoder.h"

using namespace std;

#include "faust/dsp/dsp.h"
#include "faust/gui/UI.h"

// For MIDI
#ifdef MIDICTRL
#include "faust/gui/MidiUI.h"
#include "faust/midi/bela-midi.h"
#endif

// For OSC
#ifdef OSCCTRL
#include "faust/gui/OSCUI.h"
#include "faust/gui/BelaOSCUI.h"
#endif

// For POLY
#include "faust/dsp/poly-dsp.h"

// POLY2 = POLY with effect
#ifdef POLY2
#include "faust/dsp/dsp-combiner.h"
#include "effect.cpp"
#endif

const char *const pinNamesStrings[] =
{
  "ANALOG_0",
  "ANALOG_1",
  "ANALOG_2",
  "ANALOG_3",
  "ANALOG_4",
  "ANALOG_5",
  "ANALOG_6",
  "ANALOG_7",
  "ANALOG_8",
  "DIGITAL_0",
  "DIGITAL_1",
  "DIGITAL_2",
  "DIGITAL_3",
  "DIGITAL_4",
  "DIGITAL_5",
  "DIGITAL_6",
  "DIGITAL_7",
  "DIGITAL_8",
  "DIGITAL_9",
  "DIGITAL_10",
  "DIGITAL_11",
  "DIGITAL_12",
  "DIGITAL_13",
  "DIGITAL_14",
  "DIGITAL_15",
  "ANALOG_OUT_0", // outputs
  "ANALOG_OUT_1",
  "ANALOG_OUT_2",
  "ANALOG_OUT_3",
  "ANALOG_OUT_4",
  "ANALOG_OUT_5",
  "ANALOG_OUT_6",
  "ANALOG_OUT_7",
  "ANALOG_OUT_8"};

enum EInOutPin
{
  kNoPin = -1,
  kANALOG_0 = 0,
  kANALOG_1,
  kANALOG_2,
  kANALOG_3,
  kANALOG_4,
  kANALOG_5,
  kANALOG_6,
  kANALOG_7,
  kANALOG_8,
  kDIGITAL_0,
  kDIGITAL_1,
  kDIGITAL_2,
  kDIGITAL_3,
  kDIGITAL_4,
  kDIGITAL_5,
  kDIGITAL_6,
  kDIGITAL_7,
  kDIGITAL_8,
  kDIGITAL_9,
  kDIGITAL_10,
  kDIGITAL_11,
  kDIGITAL_12,
  kDIGITAL_13,
  kDIGITAL_14,
  kDIGITAL_15,
  kANALOG_OUT_0,
  kANALOG_OUT_1,
  kANALOG_OUT_2,
  kANALOG_OUT_3,
  kANALOG_OUT_4,
  kANALOG_OUT_5,
  kANALOG_OUT_6,
  kANALOG_OUT_7,
  kANALOG_OUT_8,
  kNumInputPins
};

class BelaWidget
{
protected:
  EInOutPin fBelaPin;
  FAUSTFLOAT *fZone;  // Faust widget zone
  const char *fLabel; // Faust widget label
  FAUSTFLOAT fMin;    // Faust widget minimal value
  FAUSTFLOAT fRange;  // Faust widget value range (max-min)

public:
  BelaWidget()
      : fBelaPin(kNoPin), fZone(0), fLabel(""), fMin(0), fRange(1)
  {
  }

  BelaWidget(const BelaWidget &w)
    : fBelaPin(w.fBelaPin), fZone(w.fZone), fLabel(w.fLabel), fMin(w.
    fMin), fRange(w.fRange)
  {
  }

  BelaWidget(EInOutPin pin, FAUSTFLOAT *z, const char *l, FAUSTFLOAT
  lo, FAUSTFLOAT hi)
    : fBelaPin(pin), fZone(z), fLabel(l), fMin(lo), fRange(hi - lo)
  {
  }

  void update(BelaContext *context)
  {
    switch (fBelaPin)
    {
    case kANALOG_0:
    case kANALOG_1:
    case kANALOG_2:
    case kANALOG_3:
    case kANALOG_4:
    case kANALOG_5:
    case kANALOG_6:
    case kANALOG_7:
      *fZone = fMin + fRange * analogReadNI(context, 0, (int)fBelaPin
      );
      break;
    case kDIGITAL_0:
    case kDIGITAL_1:
    case kDIGITAL_2:
```

```
    case kDIGITAL_3:
    case kDIGITAL_4:
    case kDIGITAL_5:
    case kDIGITAL_6:
    case kDIGITAL_7:
    case kDIGITAL_8:
    case kDIGITAL_9:
    case kDIGITAL_10:
    case kDIGITAL_11:
    case kDIGITAL_12:
    case kDIGITAL_13:
    case kDIGITAL_14:
    case kDIGITAL_15:
      *fZone = digitalRead(context, 0, ((int)fBelaPin - kDIGITAL_0))
      > 0 ? fMin : fMin + fRange;
      break;
    case kANALOG_OUT_0:
    case kANALOG_OUT_1:
    case kANALOG_OUT_2:
    case kANALOG_OUT_3:
    case kANALOG_OUT_4:
    case kANALOG_OUT_5:
    case kANALOG_OUT_6:
    case kANALOG_OUT_7:
      analogWriteNI(context, 0, ((int)fBelaPin) - kANALOG_OUT_0, (*
      fZone - fMin) / (fRange + fMin));
      break;

    default:
      break;
    };
  }
};

#define MAXBELAWIDGETS 16

class BelaUI : public UI
{
private:
  // number of BelaWidgets collected so far
  int fIndex;
  // current pin id
  EInOutPin fBelaPin;
  // kind of static list of BelaWidgets
  BelaWidget fTable[MAXBELAWIDGETS];

  // check if the widget is linked to a Bela parameter and, if so,
  // add the corresponding BelaWidget
  void addBelaWidget(const char *label, FAUSTFLOAT *zone, FAUSTFLOAT
  lo, FAUSTFLOAT hi)
  {
    if (fBelaPin != kNoPin && (fIndex < MAXBELAWIDGETS))
    {
      fTable[fIndex] = BelaWidget(fBelaPin, zone, label, lo, hi);
      fIndex++;
    }
    fBelaPin = kNoPin;
  }

  // we dont want to create a widget but we clear fBelaPin just in
  case
  void skip()
  {
    fBelaPin = kNoPin;
  }

public:
  BelaUI()
      : fIndex(0), fBelaPin(kNoPin)
  {
  }

  virtual ~BelaUI() {}

  // should be called before compute() to update widget's zones
  // registered as Bela parameters
  void update(BelaContext *context)
  {
    for (int i = 0; i < fIndex; i++)
    {
      fTable[i].update(context);
    }
  }

  // -- widget's layouts
  virtual void openTabBox(const char *label) {}
  virtual void openHorizontalBox(const char *label) {}
  virtual void openVerticalBox(const char *label) {}
  virtual void closeBox() {}

  // -- active widgets
  virtual void addButton(const char *label, FAUSTFLOAT *zone) { skip
  (); }
  virtual void addCheckButton(const char *label, FAUSTFLOAT *zone) {
  skip(); }
  virtual void addVerticalSlider(const char *label, FAUSTFLOAT *zone,
   FAUSTFLOAT init, FAUSTFLOAT lo, FAUSTFLOAT hi, FAUSTFLOAT step) {
  addBelaWidget(label, zone, lo, hi); }
  virtual void addHorizontalSlider(const char *label, FAUSTFLOAT *
  zone, FAUSTFLOAT init, FAUSTFLOAT lo, FAUSTFLOAT hi, FAUSTFLOAT
  step) { addBelaWidget(label, zone, lo, hi); }
  virtual void addNumEntry(const char *label, FAUSTFLOAT *zone,
  FAUSTFLOAT init, FAUSTFLOAT lo, FAUSTFLOAT hi, FAUSTFLOAT step) {
  addBelaWidget(label, zone, lo, hi); }

  // -- passive widgets
  virtual void addHorizontalBargraph(const char *label, FAUSTFLOAT *
  zone, FAUSTFLOAT lo, FAUSTFLOAT hi) { addBelaWidget(label, zone, lo
  , hi); }
  virtual void addVerticalBargraph(const char *label, FAUSTFLOAT *
  zone, FAUSTFLOAT lo, FAUSTFLOAT hi) { addBelaWidget(label, zone, lo
  , hi); }

  // -- soundfiles
  virtual void addSoundfile(const char *label, const char *filename,
  Soundfile **sf_zone) {}

  // -- metadata declarations
  virtual void declare(FAUSTFLOAT *z, const char *k, const char *id)
  {
    if (strcasecmp(k, "BELA") == 0)
    {
      for (int i = 0; i < kNumInputPins; i++)
      {
        if (strcasecmp(id, pinNamesStrings[i]) == 0)
        {
          fBelaPin = (EInOutPin)i;
        }
      }
    }
  }
};

#endif // __FaustCommonInfrastructure__

<< includeIntrinsic >>
<< includeclass >>

std::list<GUI *> GUI::fGuiList;
ztimedmap GUI::gTimedZoneMap;

#ifdef MIDICTRL
bela_midi gMIDI;
MidiUI *gMidiInterface = NULL;
#endif

#ifdef OSCCTRL
#define OSC_IP_ADDRESS "192.168.7.1"
#define OSC_IN_PORT 5510
#define OSC_OUT_PORT 5511
```

```
BelaOSCUI gOSCUI(OSC_IP_ADDRESS, OSC_IN_PORT, OSC_OUT_PORT);
#endif

// array of pointers to context->audioIn data
FAUSTFLOAT **gFaustIns;
// array of pointers to context->audioOut data
FAUSTFLOAT **gFaustOuts;

int nvoices = 0;
BelaUI gControlUI;
dsp *gDSP = NULL;

void Bela_userSettings(BelaInitSettings *settings)
{
  // Faust code uses non-interleaved buffers
  settings->uniformSampleRate = 1;
  settings->interleave = 0;
  settings->analogOutputsPersist = 0;
}

bool setup(BelaContext *context, void *userData)
{

#ifdef NVOICES
  nvoices = NVOICES;
#endif

  // Allocate deinterleaded inputs
  gFaustIns = new FAUSTFLOAT *[context->audioInChannels];
  for (unsigned int ch = 0; ch < context->audioInChannels; ch++)
  {
    gFaustIns[ch] = (float *)&context->audioIn[ch * context->
    audioFrames];
  }

  // Allocate deinterleaded output
  gFaustOuts = new FAUSTFLOAT *[context->audioOutChannels];
  for (unsigned int ch = 0; ch < context->audioOutChannels; ch++)
  {
    gFaustOuts[ch] = (float *)&context->audioOut[ch * context->
    audioFrames];
  }

// Polyphonique with effect
#ifdef POLY2
  mydsp_poly *dsp_poly = new mydsp_poly(new mydsp(), nvoices, true,
  true);
  gDSP = new dsp_sequencer(dsp_poly, new effect());
// Polyphonique without effect
#elif NVOICES
```

```
  // is several voices, then its a simple Poly
  if (nvoices > 0)
  {
    mydsp_poly *dsp_poly = new mydsp_poly(new mydsp(), nvoices, true,
     true);
    gDSP = dsp_poly;
    // If no voices, this is not an instrument (like an FX for
    example)
  }
  else
  {
    gDSP = new mydsp();
  }
#else
  gDSP = new mydsp();
#endif

  gDSP->init(context->audioSampleRate);
  // Maps Bela Analog/Digital IO and Faust widgets
  gDSP->buildUserInterface(&gControlUI);

// If MIDI, different behaviour in Poly and non Poly
#ifdef MIDICTRL
#ifdef NVOICES
  gMIDI.addMidiIn(gDSPPoly);
#endif
  gMidiInterface = new MidiUI(&gMIDI);
  gDSP->buildUserInterface(gMidiInterface);
  gMidiInterface->run();
#endif

// OSC
#ifdef OSCCTRL
  DSP->buildUserInterface(&gOSCUI);
  gOSCUI.run();
#endif

  return true;
}

void render(BelaContext *context, void *userData)
{
  // OSC
#ifdef OSCCTRL
  gOSCUI.scheduleOSC();
#endif
  // reads Bela pins and updates corresponding Faust Widgets zones
  gControlUI.update(context);
  // synchronize all GUI controllers
  GUI::updateAllGuis();
  // process Faust DSP
  gDSP->compute(context->audioFrames, gFaustIns, gFaustOuts);
}

void cleanup(BelaContext *context, void *userData)
{
  delete[] gFaustIns;
  delete[] gFaustOuts;
  delete gDSP;

#ifdef MIDICTRL
  delete gMidiInterface;
#endif
}
```

Code A.5: Faust platform definition file for the Bela platform. Source file: (`bela.cpp`).

Appendix B

Relative Computation Cost of Floating-Point Operations

This appendix contains the code and the results used to measure the relative computation cost of floating-point arithmetic and trigonometric operations, to evaluate the cost of computing control signals relative to audio signals in an audio rendering method generated by Faust.

B.1 Relative Computation Cost Measurement

The results presented in the following subsection were calculated on PC running Windows 7 Professional with an Intel Core i3-2120 CPU @ 3.30GHz with 8 GB of RAM.

The compiler used was MinGW version 5.3.2 by running the following command:

```
$ g++ -std=c++11 source.cpp -o results
```

B.1.1 Results

The relative cycles presented here are calculated relative to the addition operation. The results are rounded to the nearest half.

Operation	Relative Cycles
+	1
-	1
*	1
/	2.5
sqrt	2.5
sin	13
cos	14
tan	24
atan	26
exp	19.5

Table B.1: Relative computation cost by floating-point operations. (Normalized to addition)

B.1.2 Source Code

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <chrono>

using namespace std;
using namespace std::chrono;

#define SIZE 20000000

double base = 0.0;
double current = 0.0;

// https://gist.github.com/gongzhitaao/7062087
// accessed November 7, 2018
class Timer
```

```
{
public:
    Timer() : beg_(clock_::now()) {}
    void reset() { beg_ = clock_::now(); }
    double elapsed() const {
            return duration_cast<second_>
                    (clock_::now() - beg_).count();
    }

private:
    typedef high_resolution_clock clock_;
    typedef duration<double, ratio<1>> second_;
    time_point<clock_> beg_;
};

int main() {

    Timer tmr;
    srand(time(NULL));

    double * S1 = new double [SIZE];
    double * S2 = new double[SIZE];
    double * D = new double[SIZE];
    double * pS1;
    double * pS2;
    double * pD;

    pS1 = S1;
    pS2 = S2;

    for (int i = 0; i < SIZE; i++) {
            *pS1++ = ((double)rand()) / ((double)(RAND_MAX));
            *pS2++ = ((double)rand()) / ((double)(RAND_MAX));
    }

// First Run
    pS1 = S1;
    pS2 = S2;
    pD = D;

    for (int i = 0; i < SIZE; i++) {
            *pD++ = (*pS1++) + (*pS2++);
    }

// ADD
    pS1 = S1;
    pS2 = S2;
    pD = D;

    tmr.reset();

    for (int i = 0; i < SIZE; i++) {
            *pD++ = (*pS1++) + (*pS2++);
    }

    base = tmr.elapsed();
    printf("ADD %.7f\n", base);

// SUB
    pS1 = S1;
    pS2 = S2;
    pD = D;

    tmr.reset();

    for (int i = 0; i < SIZE; i++) {
            *pD++ = (*pS1++) - (*pS2++);
    }

    current = tmr.elapsed();
    printf("SUB %.7f\n", current/base);

// MUL
    pS1 = S1;
    pS2 = S2;
    pD = D;

    tmr.reset();

    for (int i = 0; i < SIZE; i++) {
            *pD++ = (*pS1++) * (*pS2++);
    }

    current = tmr.elapsed();
    printf("MUL %.7f\n", current/base);

// DIV
    pS1 = S1;
    pS2 = S2;
    pD = D;

    tmr.reset();

    for (int i = 0; i < SIZE; i++) {
            *pD++ = (*pS1++) / (*pS2++);
    }

    current = tmr.elapsed();
    printf("DIV %.7f\n", current/base);

```

```
// SQRT
    pS1 = S1;
    pS2 = S2;
    pD = D;

    tmr.reset();

    for (int i = 0; i < SIZE; i++) {
            *pD++ = sqrt(*pS1++);
    }

    current = tmr.elapsed();
    printf("SQRT %.7f\n", current/base);

// SIN
    pS1 = S1;
    pS2 = S2;
    pD = D;

    tmr.reset();

    for (int i = 0; i < SIZE; i++) {
            *pD++ = sin(*pS1++);
    }

    current = tmr.elapsed();
    printf("SIN %.7f\n", current/base);

// COS
    pS1 = S1;
    pS2 = S2;
    pD = D;

    tmr.reset();

    for (int i = 0; i < SIZE; i++) {
            *pD++ = cos(*pS1++);
    }

    current = tmr.elapsed();
    printf("COS %.7f\n", current/base);


// TAN
    pS1 = S1;
    pS2 = S2;
    pD = D;

    tmr.reset();

    for (int i = 0; i < SIZE; i++) {
            *pD++ = tan(*pS1++);
    }

    current = tmr.elapsed();
    printf("TAN %.7f\n", current/base);

//  ATAN
    pS1 = S1;
    pS2 = S2;
    pD = D;

    tmr.reset();

    for (int i = 0; i < SIZE; i++) {
            *pD++ = atan(*pS1++);
    }

    current = tmr.elapsed();
    printf("ATAN %.7f\n", current/base);

//  EXP
    pS1 = S1;
    pS2 = S2;
    pD = D;

    tmr.reset();

    for (int i = 0; i < SIZE; i++) {
            *pD++ = exp(*pS1++);
    }

    current = tmr.elapsed();
    printf("EXP %.7f\n", current/base);


    delete [] S1;
    delete [] S2;
    delete [] D;

    printf("Press Enter to Quit... ");
    char input = getchar();
    getchar();

}
```

Code B.1: C++ code to measure relative computation cost.

Appendix C

Frequency Modulation in Stride

This appendix contains Stride code to perform synchronous and asynchronous frequency modulation in Stride. The generated C++ code relies on RtAudio and Boost libraries.

The code was compiled and tested using RtAudio version 5.0.0 and Boost version 1.66.0 with C++11 support. The following linker flag are required to successfully compile the code on Boost compatible operating systems: *"rtaudio boost_system boost_chrono boost_thread-mt"*

RtAudio is available online at `https://github.com/thestk/rtaudio`. Boost is available online at `http://www.boost.org/`. [accessed November 7, 2018]

C.1 Synchronous and Asynchronous Modulation

The following sections present synchronous frequency modulation in Stride on system called `RtAudioWithBoost` targeting the `Current` device.

The system `RtAudioWithBoost` defines three domains and their corresponding rates.

The first domain is called `AudioDomain` with rate called `AudioRate` set to $48{,}000$Hz. The domain `AudioDomain` abstracts the callback function assigned to the RtAudio IO stream. The system also defines the signal bundle `AudioOut` which abstracts the hardware audio output channels access by RtAudio.

The second domain is called `ControlDomain` with a rate called `ControlRate` set to $1{,}000$Hz. The domain `ControlDomain` abstracts a callback function assigned to an asynchronous Boost timer.

The third domain is called `ConstantDomain`. The rate of the domain `ConstantDomain` is set to zero. the domain abstracts a function called `Constants` that is called at the start of program execution to evaluate all expressions assigned constant values or set the values of all constant blocks in the code.

C.1.1 Synchronous Frequency Modulation

Stride Code

```
use RtAudioWithBoost on Current

```

```
signal Modulation {
    default:    0.0
    rate:       AudioRate
    domain:     AudioDomain
}

signal Output {
    default:    0.0
    rate:       AudioRate
    domain:     AudioDomain
}

SineOsc ( frequency: 1.0 )
>> Level ( gain: 40.0 offset: 220.0 )
>> Modulation;

SineOsc ( frequency: Modulation )
>> Output;

Output >> AudioOut[1:2];
```

Code C.1: Synchronous frequency modulation in Stride using RtAudio and Boost libraries.

C++ Generated Code

```
//[[Includes]]

#include <iostream>
#include <cmath>

#include <RtAudio.h>

#include <boost/asio.hpp>
#include <boost/bind.hpp>
#include <boost/thread.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>

#define NUM_IN_CHANNELS     2
#define NUM_OUT_CHANNELS    2

typedef float MY_TYPE;
#define FORMAT RTAUDIO_FLOAT32

//[[/Includes]]

//[[Declarations]]

template<class InputDataType, class OutputDataType>
```

```
class GreaterOrEqual {
public:
    GreaterOrEqual() {
    }

    void process_OutputDomain(InputDataType Input[], OutputDataType *
    Output) {
        *Output = Input[0] >= Input[1];
    }

private:
};

template <class OutputDataType>
class Sin {
public:
    Sin(){
    }

    void process_OutputDomain(OutputDataType Input, OutputDataType *
    Output) {
        *Output = std::sin(Input);
    }
};

template<class OutputDataType, class FrequencyDataType>
class SineOsc {
public:
    SineOsc(float outputRate) : OutputPort_Rate(outputRate){
    }

    void process_OutputDomain(OutputDataType *Output, OutputDataType
    *Phase, OutputDataType PhaseInc) {
        Sin_00.process_OutputDomain(*Phase, &Sin_00_Output);
        *Output = Sin_00_Output;
        *Phase = *Phase + PhaseInc;
        OutputDataType BundleConnector_00[2];
        BundleConnector_00[0] = *Phase;
        BundleConnector_00[1] = 6.28318530718;
        GreaterOrEqual_00.process_OutputDomain(BundleConnector_00, &
        GreaterOrEqual_00_Output);
        if (GreaterOrEqual_00_Output){
            reaction_WrapPhase(Phase);
        }
    }

    void process_FrequencyPortDomain(FrequencyDataType Frequency,
    OutputDataType *PhaseInc) {
        *PhaseInc = Frequency * 6.28318530718 / OutputPort_Rate;
    }

    void init_Frequency(FrequencyDataType *Frequency) {
        *Frequency = FrequencyDataType(440.0);
    }

    void init_Phase(OutputDataType *Phase) {
        *Phase = OutputDataType(0.0);
    }

    void init_PhaseInc(OutputDataType *PhaseInc) {
        FrequencyDataType Frequency;
        init_Frequency(&Frequency);
        *PhaseInc = OutputDataType(Frequency) * 6.28318530718 / OutputPort_Rate;
    }

    void reaction_WrapPhase (OutputDataType *Phase) {
         *Phase = *Phase - 6.28318530718;
    }

private:
    using GreaterOrEqual_00_Type = GreaterOrEqual<OutputDataType,bool>;
    GreaterOrEqual_00_Type GreaterOrEqual_00;
    bool GreaterOrEqual_00_Output;
    using Sin_00_Type = Sin<OutputDataType>;
    Sin_00_Type Sin_00;
    OutputDataType Sin_00_Output;

    float OutputPort_Rate;
};

template<class OutputDataType, class GainDataType, class OffsetDataType>
class Level {
public:
    Level() {
    }

    void process_OutputDomain(OutputDataType Input, OutputDataType *Output, GainDataType Gain, OffsetDataType Offset) {
        *Output = ((Input * Gain) + Offset);
    }

    void process_GainPropertyDomain(GainDataType Gain, GainDataType *Gain_) {
        *Gain_ = Gain;
    }

    void process_OffsetPropertyDomain(OffsetDataType Offset,
    OffsetDataType *Offset_) {
        *Offset_ = Offset;
    }

    void init_Gain(GainDataType *Gain) {
        *Gain = OutputDataType(1.0);
    }

    void init_Offset(OffsetDataType *Offset) {
        *Offset = OutputDataType(0.0);
    }

private:
};

float   Modulation_AudioTick = 0.0;
float   Output_AudioTick = 0.0;

using  SineOsc_00_Type = SineOsc<float, float>;
SineOsc_00_Type SineOsc_00{48000};
float   SineOsc_00_Output_AudioTick;
float   SineOsc_00_Phase_AudioTick;
float   SineOsc_00_PhaseInc_Constant;

using  Level_00_Type = Level<float, float, float>;
Level_00_Type Level_00;
float   Level_00_Gain_Constant;
float   Level_00_Offset_Constant;

using  SineOsc_01_Type = SineOsc<float, float>;
SineOsc_01_Type SineOsc_01{48000};
float   SineOsc_01_Phase_AudioTick;
float   SineOsc_01_PhaseInc_AudioTick;

void AudioTick (float &ProcessOutput) {
    SineOsc_00.process_OutputDomain(&SineOsc_00_Output_AudioTick, &
    SineOsc_00_Phase_AudioTick, SineOsc_00_PhaseInc_Constant);
    Level_00.process_OutputDomain(SineOsc_00_Output_AudioTick, &
    Modulation_AudioTick, Level_00_Gain_Constant,
    Level_00_Offset_Constant);
    SineOsc_01.process_FrequencyPortDomain(Modulation_AudioTick, &
    SineOsc_01_PhaseInc_AudioTick);
    SineOsc_01.process_OutputDomain(&Output_AudioTick, &
    SineOsc_01_Phase_AudioTick, SineOsc_01_PhaseInc_AudioTick);
    ProcessOutput = Output_AudioTick;
}

void Constants () {
    SineOsc_00.process_FrequencyPortDomain(1.0, &
    SineOsc_00_PhaseInc_Constant);
    Level_00.process_GainPropertyDomain(40.0, &Level_00_Gain_Constant);
    Level_00.process_OffsetPropertyDomain(220.0, &Level_00_Offset_Constant);
}

void Initialize () {
    SineOsc_00.init_Phase(&SineOsc_00_Phase_AudioTick);
    SineOsc_01.init_Phase(&SineOsc_01_Phase_AudioTick);
    SineOsc_01.init_PhaseInc(&SineOsc_01_PhaseInc_AudioTick);
}

//[[/Declarations]]

//[[Processing]]

int audio_buffer_process( void *outputBuffer, void *inputBuffer, unsigned int nBufferFrames, double streamTime, RtAudioStreamStatus status, void *data )
{
    if (status) std::cout << "Stream over/underflow detected." << std::endl;

    MY_TYPE *in = (MY_TYPE *)inputBuffer;
    MY_TYPE *out = (MY_TYPE *)outputBuffer;
    MY_TYPE output = 0.0;
    while(nBufferFrames-- > 0) {
        AudioTick (output);
        out[0] = output;
        out[1] = output;
        in += NUM_IN_CHANNELS;
        out += NUM_OUT_CHANNELS;
    }

    return 0;
}

class EndOnInput {
public:
    EndOnInput(RtAudio &rtAudio) : p_rtAudio(rtAudio) { }

    void operator()() {
        char enter;
        std::cout << std::endl << "Press <enter> to quit!" << std::endl;
        std::cin.get(enter);

        try {
            if (p_rtAudio.isStreamRunning()) p_rtAudio.stopStream();
            if (p_rtAudio.isStreamOpen()) p_rtAudio.closeStream();
        }
        catch (RtAudioError& e) {
            e.printMessage();
        }

        return;
    }

private:
    RtAudio &p_rtAudio;
};

//[[/Processing]]

int main() {

    // Initialize
    Initialize();

    // Process Constants
    Constants();

    // Check for audio devices
    RtAudio rtAudio;
    if (rtAudio.getDeviceCount() < 1) {
        std::cout << std::endl << "No audio devices found!" << std::
        endl;
        exit(-1);
    }

    // Setup up termination on user input
    EndOnInput endOnInput(rtAudio);

    // Run user termination on a separate thread
    boost::thread endOnInputThread(endOnInput);

    // Set the same number of channels for both input and output.
    unsigned int bufferBytes;
    unsigned int bufferFrames = 512;
    unsigned int fs = 48000;

    bufferBytes = bufferFrames * NUM_OUT_CHANNELS * sizeof( MY_TYPE )
    ;

    RtAudio::StreamParameters iParams;
    iParams.deviceId = rtAudio.getDefaultInputDevice();
    iParams.nChannels = NUM_IN_CHANNELS;

    RtAudio::StreamParameters oParams;
    oParams.deviceId = rtAudio.getDefaultOutputDevice();
    oParams.nChannels = NUM_OUT_CHANNELS;

    RtAudio::StreamOptions options;

    try {
        rtAudio.openStream( &oParams, &iParams, FORMAT, fs, &
        bufferFrames, &audio_buffer_process, (void *)&bufferBytes, &
        options);
    }
    catch (RtAudioError& e) {
        e.printMessage();
        exit(-1);
    }

    // Start Audio Streams
    try {
        rtAudio.startStream();
    }
    catch (RtAudioError& e) {
        e.printMessage();
        if (rtAudio.isStreamOpen()) rtAudio.closeStream();
        exit (-1);
    }

    // Join user termination
    endOnInputThread.join();

    return 0;
}
```

Code C.2: Generated C++ code for synchronous frequency modulation.

C.1.2 Asynchronous Frequency Modulation

Stride Code

```
use RtAudioWithBoost on Current

signal Modulation {
    default:    0.0
    rate:       ControlRate
    domain:     ControlDomain
}

signal Output {
    default:    0.0
    rate:       AudioRate
    domain:     AudioDomain
}

SineOsc ( frequency: 1.0 )
>> Level ( gain: 40.0 offset: 220.0 )
>> Modulation;

SineOsc ( frequency: Modulation )
>> Output;

Output >> AudioOut[1:2];
```

Code C.3: Asynchronous frequency modulation in Stride using RtAudio and Boost libraries.

C++ Generated Code

```
//[[Includes]]

#include <iostream>
#include <cmath>

#include <RtAudio.h>

#include <boost/asio.hpp>
#include <boost/bind.hpp>
#include <boost/thread.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>

#define NUM_IN_CHANNELS     2
#define NUM_OUT_CHANNELS    2
#define  CONTROL_TIME_MS    1

typedef float MY_TYPE;
#define FORMAT RTAUDIO_FLOAT32

//[[/Includes]]

//[[Declarations]]

template<class InputDataType, class OutputDataType>
class GreaterOrEqual {
public:
    GreaterOrEqual() {
    }

    void process_OutputDomain(InputDataType Input[], OutputDataType *
    Output) {
        *Output = Input[0] >= Input[1];
    }

private:
};

template <class OutputDataType>
class Sin {
public:
    Sin(){
    }

    void process_OutputDomain(OutputDataType Input, OutputDataType *
    Output) {
        *Output = std::sin(Input);
    }
};

template<class OutputDataType, class FrequencyDataType>
class SineOsc {
public:
    SineOsc(float outputRate) : OutputPort_Rate(outputRate){
    }

    void process_OutputDomain(OutputDataType *Output, OutputDataType
    *Phase, OutputDataType PhaseInc) {
        Sin_00.process_OutputDomain(*Phase, &Sin_00_Output);
        *Output = Sin_00_Output;
        *Phase = *Phase + PhaseInc;
        OutputDataType BundleConnector_00[2];
        BundleConnector_00[0] = *Phase;
        BundleConnector_00[1] = 6.28318530718;
        GreaterOrEqual_00.process_OutputDomain(BundleConnector_00, &
        GreaterOrEqual_00_Output);
        if (GreaterOrEqual_00_Output){
            reaction_WrapPhase(Phase);
        }
    }

    void process_FrequencyPortDomain(FrequencyDataType Frequency,
    OutputDataType *PhaseInc) {
        *PhaseInc = Frequency * 6.28318530718 / OutputPort_Rate;
    }

    void init_Frequency(FrequencyDataType *Frequency) {
        *Frequency = FrequencyDataType(440.0);
    }

    void init_Phase(OutputDataType *Phase) {
        *Phase = OutputDataType(0.0);
    }

    void init_PhaseInc(OutputDataType *PhaseInc) {
        FrequencyDataType Frequency;
        init_Frequency(&Frequency);
        *PhaseInc = OutputDataType(Frequency) * 6.28318530718 /
        OutputPort_Rate;
    }

    void reaction_WrapPhase (OutputDataType *Phase) {
         *Phase = *Phase - 6.28318530718;
    }

private:
    using GreaterOrEqual_00_Type = GreaterOrEqual<OutputDataType,bool
    >;
    GreaterOrEqual_00_Type GreaterOrEqual_00;
    bool GreaterOrEqual_00_Output;
    using Sin_00_Type = Sin<OutputDataType>;
    Sin_00_Type Sin_00;
    OutputDataType Sin_00_Output;

    float OutputPort_Rate;
};

template<class OutputDataType, class GainDataType, class
OffsetDataType>
class Level {
public:
    Level() {
    }

    void process_OutputDomain(OutputDataType Input, OutputDataType *
    Output, GainDataType Gain, OffsetDataType Offset) {
        *Output = ((Input * Gain) + Offset);
    }

    void process_GainPropertyDomain(GainDataType Gain, GainDataType *
    Gain_) {
        *Gain_ = Gain;
    }

    void process_OffsetPropertyDomain(OffsetDataType Offset,
    OffsetDataType *Offset_) {
        *Offset_ = Offset;
    }

    void init_Gain(GainDataType *Gain) {
        *Gain = OutputDataType(1.0);
    }

    void init_Offset(OffsetDataType *Offset) {
        *Offset = OutputDataType(0.0);
    }

private:
};

float   Modulation_AudioTick = 0.0;
float   Output_AudioTick = 0.0;

using  SineOsc_00_Type = SineOsc<float, float>;
SineOsc_00_Type SineOsc_00{1.0/(CONTROL_TIME_MS/1000.0)};
float   SineOsc_00_Output_ControlTick;
float   SineOsc_00_Phase_ControlTick;
float   SineOsc_00_PhaseInc_Constant;

using  Level_00_Type = Level<float, float, float>;
Level_00_Type Level_00;
float   Level_00_Gain_Constant;
float   Level_00_Offset_Constant;

using  SineOsc_01_Type = SineOsc<float, float>;
SineOsc_01_Type SineOsc_01{48000};
float   SineOsc_01_Phase_AudioTick;
float   SineOsc_01_PhaseInc_AudioTick_ControlTick;

void AudioTick (float &ProcessOutput) {
    SineOsc_01.process_OutputDomain(&Output_AudioTick, &
    SineOsc_01_Phase_AudioTick,
    SineOsc_01_PhaseInc_AudioTick_ControlTick);
    ProcessOutput = Output_AudioTick;
}

void ControlTick () {
    SineOsc_00.process_OutputDomain(&SineOsc_00_Output_ControlTick, &
    SineOsc_00_Phase_ControlTick, SineOsc_00_PhaseInc_Constant);
    Level_00.process_OutputDomain(SineOsc_00_Output_ControlTick, &
    Modulation_AudioTick, Level_00_Gain_Constant,
    Level_00_Offset_Constant);
    SineOsc_01.process_FrequencyPortDomain(Modulation_AudioTick, &
    SineOsc_01_PhaseInc_AudioTick_ControlTick);
}

void Constants () {
    SineOsc_00.process_FrequencyPortDomain(1.0, &
    SineOsc_00_PhaseInc_Constant);
    Level_00.process_GainPropertyDomain(40.0, &Level_00_Gain_Constant
    );
    Level_00.process_OffsetPropertyDomain(220.0, &
    Level_00_Offset_Constant);
}

void Initialize () {
    SineOsc_00.init_Phase(&SineOsc_00_Phase_ControlTick);
    SineOsc_01.init_Phase(&SineOsc_01_Phase_AudioTick);
    SineOsc_01.init_PhaseInc(&
    SineOsc_01_PhaseInc_AudioTick_ControlTick);
}

//[[/Declarations]]

//[[Processing]]

int audio_buffer_process( void *outputBuffer, void *inputBuffer,
unsigned int nBufferFrames, double streamTime, RtAudioStreamStatus
status, void *data )
{
    if (status) std::cout << "Stream over/underflow detected." << std
    ::endl;

    MY_TYPE *in = (MY_TYPE *)inputBuffer;
    MY_TYPE *out = (MY_TYPE *)outputBuffer;
    MY_TYPE output = 0.0;
    while(nBufferFrames-- > 0) {
        AudioTick (output);
        out[0] = output;
        out[1] = output;
        in += NUM_IN_CHANNELS;
        out += NUM_OUT_CHANNELS;
    }

    return 0;
}

class Control {
public:
    Control(boost::asio::deadline_timer &timer, long time, void (*
    callBack) ()) : p_timer(timer), p_time(time), p_callBack(callBack
    ) {
        p_setupWait();
    }

    void tick(const boost::system::error_code &e) {
        if (e) return;
        p_callBack ();
        //std::cout << p_time << " : " << p_timer.expires_at() << std
        ::endl;
        p_timer.expires_at(p_timer.expires_at() + boost::posix_time::
        millisec(p_time));
        p_setupWait();
    }

    void cancel() {
        p_timer.cancel();
    }

private:
    boost::asio::deadline_timer &p_timer;
    long p_time;
    void (*p_callBack) ();
    void p_setupWait() {
        p_timer.async_wait(boost::bind(&Control::tick, this, boost::
        asio::placeholders::error));
    }
};

class EndOnInput {
public:
    EndOnInput(Control &control, RtAudio &rtAudio) : p_control(
    control), p_rtAudio(rtAudio) { }

    void operator()() {
        char enter;
        std::cout << std::endl << "Press <enter> to quit!" << std::
        endl;
        std::cin.get(enter);

        p_control.cancel();

        try {
            if (p_rtAudio.isStreamRunning()) p_rtAudio.stopStream();
            if (p_rtAudio.isStreamOpen()) p_rtAudio.closeStream();
        }
        catch (RtAudioError& e) {
            e.printMessage();
        }

        return;
    }

private:
    Control &p_control;
    RtAudio &p_rtAudio;
};

//[[/Processing]]

```

```
int main() {

    // Initialize
    Initialize();

    // Process Constants
    Constants();

    // Setup IO service
    boost::asio::io_service io;
    // Setup Control Timer
    boost::asio::deadline_timer controlTimer(io, boost::posix_time::
    millisec(CONTROL_TIME_MS));
    // Start Control Timer Callback
    Control control(controlTimer, CONTROL_TIME_MS, &ControlTick);

    // Check for audio devices
    RtAudio rtAudio;
    if (rtAudio.getDeviceCount() < 1) {
        std::cout << std::endl << "No audio devices found!" << std::
        endl;
        exit(-1);
    }

    // Setup up termination on user input
    EndOnInput endOnInput(control, rtAudio);

    // Run user termination on a separate thread
    boost::thread endOnInputThread(endOnInput);

    // Set the same number of channels for both input and output.
    unsigned int bufferBytes;
    unsigned int bufferFrames = 512;
    unsigned int fs = 48000;

    bufferBytes = bufferFrames * NUM_OUT_CHANNELS * sizeof( MY_TYPE )
    ;

    RtAudio::StreamParameters iParams;
    iParams.deviceId = rtAudio.getDefaultInputDevice();
    iParams.nChannels = NUM_IN_CHANNELS;

    RtAudio::StreamParameters oParams;
    oParams.deviceId = rtAudio.getDefaultOutputDevice();
    oParams.nChannels = NUM_OUT_CHANNELS;

    RtAudio::StreamOptions options;

    try {
        rtAudio.openStream( &oParams, &iParams, FORMAT, fs, &
        bufferFrames, &audio_buffer_process, (void *)&bufferBytes, &
        options);
    }
    catch (RtAudioError& e) {
        e.printMessage();
        exit(-1);
    }

    // Start Audio Streams
    try {
        rtAudio.startStream();
    }
    catch (RtAudioError& e) {
        e.printMessage();
        if (rtAudio.isStreamOpen()) rtAudio.closeStream();
        exit (-1);
    }

    // Start IO service
    io.run();

    // Join user termination
    endOnInputThread.join();

    return 0;
}
```

Code C.4: Generated C++ code for asynchronous frequency modulation.

C.1.3 Asynchronous Frequency Modulation with Concurrency

Stride Code

```
use RtAudioWithBoost on Current

mutualExclusion TryLockOnReadLockOnWrite {
    read:   TryLock
    write:  Lock
}

synchronization AudioReadControlWrite {
    readDomain:     AudioDomain
    writeDomain:    ControlDomain
    mode:           TryLockOnReadLockOnWrite
}
```

```

signal Modulation {
    default:    0.0
    rate:       ControlRate
    domain:     ControlDomain
}

signal Output {
    default:    0.0
    rate:       AudioRate
    domain:     AudioDomain
}

SineOsc ( frequency: 1.0 )
>> Level ( gain: 40.0 offset: 220.0 )
>> Modulation;

SineOsc ( frequency: Modulation )
>> Output;

Output >> AudioOut[1:2];
```

Code C.5: Asynchronous frequency modulation in Stride using RtAudio and Boost libraries with concurrency control.

C++ Generated Code

```
//[[Includes]]

#include <iostream>
#include <cmath>
#include <mutex>

#include <RtAudio.h>

#include <boost/asio.hpp>
#include <boost/bind.hpp>
#include <boost/thread.hpp>
#include <boost/date_time/posix_time/posix_time.hpp>

#define NUM_IN_CHANNELS     2
#define NUM_OUT_CHANNELS    2
#define  CONTROL_TIME_MS    1

typedef float MY_TYPE;
#define FORMAT RTAUDIO_FLOAT32

//[[/Includes]]
```

```

//[[Declarations]]

template<class InputDataType, class OutputDataType>
class GreaterOrEqual {
public:
    GreaterOrEqual() {
    }

    void process_OutputDomain(InputDataType Input[], OutputDataType *Output) {
        *Output = Input[0] >= Input[1];
    }

private:
};

template <class OutputDataType>
class Sin {
public:
    Sin(){
    }

    void process_OutputDomain(OutputDataType Input, OutputDataType *Output) {
        *Output = std::sin(Input);
    }
};

template<class OutputDataType, class FrequencyDataType>
class SineOsc {
public:
    SineOsc(float outputRate) : OutputPort_Rate(outputRate){
    }

    void process_OutputDomain(OutputDataType *Output, OutputDataType *Phase, OutputDataType PhaseInc) {
        Sin_00.process_OutputDomain(*Phase, &Sin_00_Output);
        *Output = Sin_00_Output;
        *Phase = *Phase + PhaseInc;
        OutputDataType BundleConnector_00[2];
        BundleConnector_00[0] = *Phase;
        BundleConnector_00[1] = 6.28318530718;
        GreaterOrEqual_00.process_OutputDomain(BundleConnector_00, &GreaterOrEqual_00_Output);
        if (GreaterOrEqual_00_Output){
            reaction_WrapPhase(Phase);
        }
    }

    void process_FrequencyPortDomain(FrequencyDataType Frequency,
    OutputDataType *PhaseInc) {
        *PhaseInc = Frequency * 6.28318530718 / OutputPort_Rate;
    }

    void init_Frequency(FrequencyDataType *Frequency) {
        *Frequency = FrequencyDataType(440.0);
    }

    void init_Phase(OutputDataType *Phase) {
        *Phase = OutputDataType(0.0);
    }

    void init_PhaseInc(OutputDataType *PhaseInc) {
        FrequencyDataType Frequency;
        init_Frequency(&Frequency);
        *PhaseInc = OutputDataType(Frequency) * 6.28318530718 /
        OutputPort_Rate;
    }

    void reaction_WrapPhase (OutputDataType *Phase) {
         *Phase = *Phase - 6.28318530718;
    }

private:
    using GreaterOrEqual_00_Type = GreaterOrEqual<OutputDataType,bool
    >;
    GreaterOrEqual_00_Type GreaterOrEqual_00;
    bool GreaterOrEqual_00_Output;
    using Sin_00_Type = Sin<OutputDataType>;
    Sin_00_Type Sin_00;
    OutputDataType Sin_00_Output;

    float OutputPort_Rate;
};

template<class OutputDataType, class GainDataType, class
OffsetDataType>
class Level {
public:
    Level() {
    }

    void process_OutputDomain(OutputDataType Input, OutputDataType *
    Output, GainDataType Gain, OffsetDataType Offset) {
        *Output = ((Input * Gain) + Offset);
    }

    void process_GainPropertyDomain(GainDataType Gain, GainDataType *
    Gain_) {
        *Gain_ = Gain;
    }

    void process_OffsetPropertyDomain(OffsetDataType Offset,
    OffsetDataType *Offset_) {
        *Offset_ = Offset;
    }

    void init_Gain(GainDataType *Gain) {
        *Gain = OutputDataType(1.0);
    }

    void init_Offset(OffsetDataType *Offset) {
        *Offset = OutputDataType(0.0);
    }

private:
};

std::mutex R_AudioTick_W_ControlTick_Mutex;

float   Modulation_AudioTick = 0.0;
float   Output_AudioTick = 0.0;

using  SineOsc_00_Type = SineOsc<float, float>;
SineOsc_00_Type SineOsc_00{1.0/(CONTROL_TIME_MS/1000.0)};
float   SineOsc_00_Output_ControlTick;
float   SineOsc_00_Phase_ControlTick;
float   SineOsc_00_PhaseInc_Constant;

using  Level_00_Type = Level<float, float, float>;
Level_00_Type Level_00;
float   Level_00_Gain_Constant;
float   Level_00_Offset_Constant;

using  SineOsc_01_Type = SineOsc<float, float>;
SineOsc_01_Type SineOsc_01{48000};
float   SineOsc_01_Phase_AudioTick;
float   SineOsc_01_PhaseInc_AudioTick;
float   SineOsc_01_PhaseInc_AudioTick_ControlTick;

void AudioTick (float &ProcessOutput) {
    if (R_AudioTick_W_ControlTick_Mutex.try_lock()) {
        SineOsc_01_PhaseInc_AudioTick =
        SineOsc_01_PhaseInc_AudioTick_ControlTick;
        R_AudioTick_W_ControlTick_Mutex.unlock();
    }
    SineOsc_01.process_OutputDomain(&Output_AudioTick, &
    SineOsc_01_Phase_AudioTick, SineOsc_01_PhaseInc_AudioTick);
    ProcessOutput = Output_AudioTick;
```

```
}

void ControlTick () {
    SineOsc_00.process_OutputDomain(&SineOsc_00_Output_ControlTick, &
    SineOsc_00_Phase_ControlTick, SineOsc_00_PhaseInc_Constant);
    Level_00.process_OutputDomain(SineOsc_00_Output_ControlTick, &
    Modulation_AudioTick, Level_00_Gain_Constant,
    Level_00_Offset_Constant);
    R_AudioTick_W_ControlTick_Mutex.lock();
    SineOsc_01.process_FrequencyPortDomain(Modulation_AudioTick, &
    SineOsc_01_PhaseInc_AudioTick_ControlTick);
    R_AudioTick_W_ControlTick_Mutex.unlock();
}

void Constants () {
    SineOsc_00.process_FrequencyPortDomain(1.0, &
    SineOsc_00_PhaseInc_Constant);
    Level_00.process_GainPropertyDomain(40.0, &Level_00_Gain_Constant
    );
    Level_00.process_OffsetPropertyDomain(220.0, &
    Level_00_Offset_Constant);
}

void Initialize () {
    SineOsc_00.init_Phase(&SineOsc_00_Phase_ControlTick);
    SineOsc_01.init_Phase(&SineOsc_01_Phase_AudioTick);
    SineOsc_01.init_PhaseInc(&SineOsc_01_PhaseInc_AudioTick);
    SineOsc_01.init_PhaseInc(&
    SineOsc_01_PhaseInc_AudioTick_ControlTick);
}

//[[/Declarations]]

//[[Processing]]

int audio_buffer_process( void *outputBuffer, void *inputBuffer,
unsigned int nBufferFrames, double streamTime, RtAudioStreamStatus
status, void *data )
{
    if (status) std::cout << "Stream over/underflow detected." << std
    ::endl;

    MY_TYPE *in = (MY_TYPE *)inputBuffer;
    MY_TYPE *out = (MY_TYPE *)outputBuffer;
    MY_TYPE output = 0.0;
    while(nBufferFrames-- > 0) {
        AudioTick (output);
        out[0] = output;
        out[1] = output;
        in += NUM_IN_CHANNELS;
        out += NUM_OUT_CHANNELS;
    }

    return 0;
}

class Control {
public:
    Control(boost::asio::deadline_timer &timer, long time, void (*
    callBack) ()) : p_timer(timer), p_time(time), p_callBack(callBack
    ) {
        p_setupWait();
    }

    void tick(const boost::system::error_code &e) {
        if (e) return;
        p_callBack ();
        //std::cout << p_time << " : " << p_timer.expires_at() << std
        ::endl;
        p_timer.expires_at(p_timer.expires_at() + boost::posix_time::
        millisec(p_time));
        p_setupWait();
    }

    void cancel() {
        p_timer.cancel();
    }

private:
    boost::asio::deadline_timer &p_timer;
    long p_time;
    void (*p_callBack) ();
    void p_setupWait() {
        p_timer.async_wait(boost::bind(&Control::tick, this, boost::
        asio::placeholders::error));
    }
};

class EndOnInput {
public:
    EndOnInput(Control &control, RtAudio &rtAudio) : p_control(
    control), p_rtAudio(rtAudio) { }

    void operator()() {
        char enter;
        std::cout << std::endl << "Press <enter> to quit!" << std::
        endl;
        std::cin.get(enter);

        p_control.cancel();

        try {
            if (p_rtAudio.isStreamRunning()) p_rtAudio.stopStream();
            if (p_rtAudio.isStreamOpen()) p_rtAudio.closeStream();
        }
        catch (RtAudioError& e) {
            e.printMessage();
        }

        return;
    }

private:
    Control &p_control;
    RtAudio &p_rtAudio;
};

//[[/Processing]]

int main() {

    // Initialize
    Initialize();

    // Process Constants
    Constants();

    // Setup IO service
    boost::asio::io_service io;
    // Setup Control Timer
    boost::asio::deadline_timer controlTimer(io, boost::posix_time::
    millisec(CONTROL_TIME_MS));
    // Start Control Timer Callback
    Control control(controlTimer, CONTROL_TIME_MS, &ControlTick);

    // Check for audio devices
    RtAudio rtAudio;
    if (rtAudio.getDeviceCount() < 1) {
        std::cout << std::endl << "No audio devices found!" << std::
        endl;
        exit(-1);
    }

    // Setup up termination on user input
    EndOnInput endOnInput(control, rtAudio);

    // Run user termination on a separate thread
    boost::thread endOnInputThread(endOnInput);

    // Set the same number of channels for both input and output.
    unsigned int bufferBytes;
    unsigned int bufferFrames = 512;
    unsigned int fs = 48000;

    bufferBytes = bufferFrames * NUM_OUT_CHANNELS * sizeof( MY_TYPE )
    ;

    RtAudio::StreamParameters iParams;
    iParams.deviceId = rtAudio.getDefaultInputDevice();
    iParams.nChannels = NUM_IN_CHANNELS;

    RtAudio::StreamParameters oParams;
    oParams.deviceId = rtAudio.getDefaultOutputDevice();
    oParams.nChannels = NUM_OUT_CHANNELS;

    RtAudio::StreamOptions options;

    try {
        rtAudio.openStream( &oParams, &iParams, FORMAT, fs, &
        bufferFrames, &audio_buffer_process, (void *)&bufferBytes, &
        options);
    }
    catch (RtAudioError& e) {
        e.printMessage();
        exit(-1);
    }

    // Start Audio Streams
    try {
        rtAudio.startStream();
    }
    catch (RtAudioError& e) {
        e.printMessage();
        if (rtAudio.isStreamOpen()) rtAudio.closeStream();
        exit (-1);
    }

    // Start IO service
    io.run();

    // Join user termination
    endOnInputThread.join();

    return 0;
}
```

Code C.6: Generated C++ code for asynchronous frequency modulation with concurrency.

Appendix D

Stride Helper Classes

The code in the following sections are a selection of helper classes used by the Stride code generator that appeared in examples used in preceding chapters.

D.1 Synchronization

```
namespace sync {

class scoped {};
class unscoped {};

class lock {};
class try_lock {};

template <class LockType>
class Synchronization {
public:
    Synchronization(std::mutex *m, sync::lock, sync::unscoped) {
        m->lock();
    }

    Synchronization(std::mutex *m, sync::try_lock, sync::unscoped) {
        m_LockOwned = m->try_lock();
    }

    Synchronization(std::mutex *m, sync::lock, sync::scoped) :
    m_ScopedResetLock(*m) {
    }

    Synchronization(std::mutex *m, sync::try_lock, sync::scoped) :
    m_ScopedResetLock(*m, std::try_to_lock) {
        m_LockOwned = m_ScopedResetLock.owns_lock();
    }

    bool operator()(LockType) {
        return true;
    }

private:
    std::unique_lock<std::mutex> m_ScopedResetLock;
    bool m_LockOwned;
};

template<> bool Synchronization<sync::try_lock>::operator()(sync::
try_lock) {
    return m_LockOwned;
}

}
```

Code D.1: Synchronization class.

D.2 Signals

```
template<class DataType>
class SignalReadWriteResetInterface {
public:
    virtual void Swap(void) = 0;
    virtual DataType Read(void) = 0;
    virtual bool Lock(void) = 0;
    virtual DataType * Write(void) = 0;
    virtual void Unlock(void) = 0;
};
```

Code D.2: Signal interface class.

```
template<class ClassType, class DataType>
class Signal_SDRWRst : public SignalReadWriteResetInterface<DataType>
 {
public:
    Signal_SDRWRst(void (ClassType::*Init)(DataType *), ClassType *
    object) {
        (object->*Init)(&m_Signal_Default);
        m_Signal = m_Signal_Default;
    }

    void Swap(void) {
    }

    DataType Read(void) {
        return m_Signal;
    }

    bool Lock(void) {
        return true;
    }

    DataType* Write(void) {
        return &m_Signal;
    }

    void Unlock(void) {
    }

    void Reset(void) {
        m_Signal = m_Signal_Default;
    }

private:
    DataType m_Signal_Default;
    DataType m_Signal;
};
```

Code D.3: Single domain read, write, and reset signal class.

```
template<class ClassType, class DataType, class ResetReadLockType,
class ResetWriteLockType>
class Signal_SDRW_MDRst : public SignalReadWriteResetInterface<
DataType> {
public:
    Signal_SDRW_MDRst(void (ClassType::*Init)(DataType *), ClassType
    *object, std::mutex *resetMutex) : m_ResetMutex(resetMutex) {
        (object->*Init)(&m_Signal_Default);
        m_Signal = m_Signal_Default;
    }

    void Swap(void) {
        sync::Synchronization<ResetReadLockType> ResetSync(
        m_ResetMutex, ResetReadLockType(), sync::scoped());
        if (ResetSync(ResetReadLockType())) {
            if (m_Reset_Invoked) {
                m_Reset_Invoked = false;
                m_Signal = m_Signal_Default;
            }
        }
    }

    DataType Read(void) {
        return m_Signal;
    }

    bool Lock(void) {
        return true;
    }

    DataType* Write(void) {
        sync::Synchronization<ResetReadLockType> ResetSync(
        m_ResetMutex, ResetReadLockType(), sync::scoped());
        if (ResetSync(ResetReadLockType())) {
            m_Reset_Invoked = false;
        }
        return &m_Signal;
    }

    void Unlock(void) {
    }

    void Reset(void) {
        sync::Synchronization<ResetWriteLockType> ResetSync(
        m_ResetMutex, ResetWriteLockType(), sync::scoped());
        if (ResetSync(ResetWriteLockType())) {
            m_Reset_Invoked = true;
        }
    }

private:
    std::mutex *m_ResetMutex;

    DataType m_Signal;
    DataType m_Signal_Default;
    bool m_Reset_Invoked = false;
};
```

Code D.4: Single domain read, write, and multi domain reset signal class.

D.3 Trigger Observers

```
class TriggerObserver {
public:
    virtual void Update(void) = 0;
};
```

Code D.5: Trigger observer interface class.

```
template<class ClassType>
class TriggerObserverBlock : public TriggerObserver {
public:
    TriggerObserverBlock(void (ClassType::*method)(), ClassType *
    object) : m_Object(object), m_Method(method) {
    }

    void Update(void) {
        (m_Object->*m_Method)();
    }

private:
    ClassType *m_Object;
    void (ClassType::*m_Method)();
};
```

Code D.6: Trigger observer block class.

D.4 Triggers

```
class Trigger {
public:
    virtual void Register(TriggerObserver *Observer) = 0;
    virtual void Update(void) { assert(false); }
    virtual void Update(bool) { assert(false); }
};
```

Code D.7: Trigger interface class.

```
class Trigger_SD_TriggerControlled : public Trigger {
public:
    Trigger_SD_TriggerControlled(void) {
    }

    void Register(TriggerObserver *Observer) {
        ObserverList.push_front(Observer);
    }

    void Update(void) {
        if (m_Triggered) {
            m_Triggered = false;
            m_Process = true;
        }
        if (m_Process) {
            Execute();
            m_Process = false;
        }
    }

    void Fire(void) {
        m_Triggered = true;
    }

private:
    bool m_Triggered = false;
    bool m_Process = false;
    std::forward_list<TriggerObserver *> ObserverList;

    void Execute (void) {
        for (auto Observer : ObserverList) Observer->Update();
    }
};
```

Code D.8: Single domain trigger-controlled trigger class.

```
template<class TriggerReadLockType, class TriggerWriteLockType>
class Trigger_MD_TriggerControlled : public Trigger {
public:
    Trigger_MD_TriggerControlled(std::mutex *triggerMutex) :
    m_TriggerMutex(triggerMutex) {
    }

    void Register(TriggerObserver *Observer) {
        ObserverList.push_front(Observer);
    }

    void Update(void) {
        sync::Synchronization<TriggerReadLockType> TriggerSync(
        m_TriggerMutex, TriggerReadLockType(), sync::unscoped());
        if (TriggerSync(TriggerReadLockType())) {
            if (m_Triggered) {
                m_Triggered = false;
                m_Process = true;
            }
            m_TriggerMutex->unlock();
        }
        if (m_Process) {
            Execute();
            m_Process = false;
        }
    }

    void Fire(void) {
        sync::Synchronization<TriggerWriteLockType> TriggerSync(
        m_TriggerMutex, TriggerWriteLockType(), sync::scoped());
        if (TriggerSync(TriggerWriteLockType())) {
            m_Triggered = true;
        }
    }

private:
    std::mutex *m_TriggerMutex;

    bool m_Triggered = false;
    bool m_Process = false;
    std::forward_list<TriggerObserver *> ObserverList;

    void Execute (void) {
        for (auto Observer : ObserverList) Observer->Update();
    }
};
```

Code D.9: Multi domain trigger-controlled trigger class.

```
class Trigger_SwitchControlled : public Trigger {
public:
    enum TriggerMode {
        RISING = 1,
        FALLING = 2,
        BOTH = 3
    };

    Trigger_SwitchControlled(TriggerMode mode) : m_Trigger_Mode(mode)
     {
    }

    void Register(TriggerObserver *Observer) {
        ObserverList.push_front(Observer);
    }

    void Update(bool switchState) {
        switch(m_Trigger_Mode) {
            case TriggerMode::RISING:
                if (switchState - m_Previous_Value > 0)
```

```
                    Execute();
                    m_Previous_Value = switchState;
                    break;
                case TriggerMode::FALLING:
                    if (switchState  - m_Previous_Value < 0)
                    Execute();
                    m_Previous_Value = switchState;
                    break;
                case TriggerMode::BOTH:
                    if (switchState != m_Previous_Value)
                    Execute();
                    m_Previous_Value = switchState;
                    break;
                default:
                    break;
            }
        }

private:
    int m_Previous_Value = 0;
    TriggerMode m_Trigger_Mode;
    std::forward_list<TriggerObserver *> ObserverList;

    void Execute (void) {
        for (auto Observer : ObserverList) Observer->Update();
    }
};
```

Code D.10: Switch controlled trigger class.

Appendix E

Stride Lexeme and Grammar

This appendix contains the lexeme and grammar used by Stride's lexical analyzer and parser respectively. The lexeme and grammar correspond to Stride V1.0.

E.1 Stride Lexeme

The lexeme of Stride is presented here as regular expressions. They are compliant with the flex[40] lexical analyzer.

The token names (in boldface) are used in the grammar presented in the next section.

Definitions:

```
    DIGIT        [0-9]
    LETTER       [a-z]
    CLETTER      [A-Z]
```

USE:

```
"use"
```

VERSION:

```
"version"
```

IMPORT:

```
"import"
```

AS:

```
"as"
```

NONE:

```
"none"
```

AND:

```
"and"
"&&"
```

OR:

```
"or"
"||"
```

NOT:

```
"not"
```

ON:

```
"on"
```

OFF:

```
"off"
```

UVAR:

```
(_)*{CLETTER}({LETTER}|{CLETTER}|{DIGIT}|_)*
```

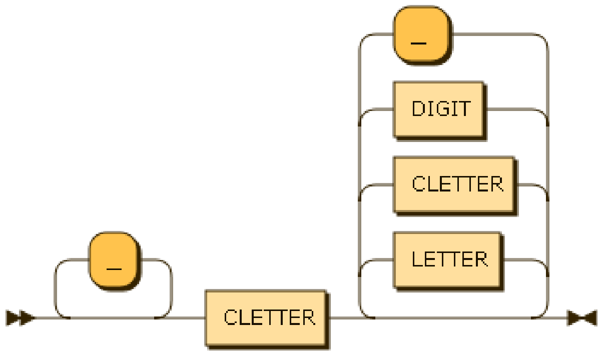


Figure E.1: UVAR

WORD:

```
(_)*{LETTER}({LETTER}|{CLETTER}|{DIGIT})*
```

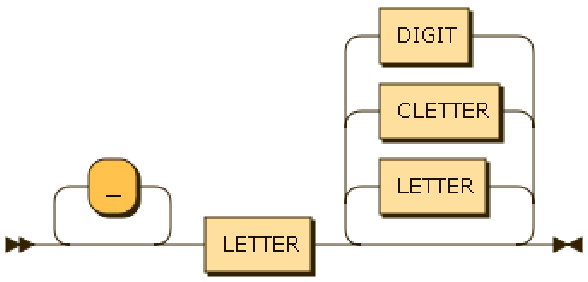


Figure E.2: WORD

INT:

```
{DIGIT}+
```

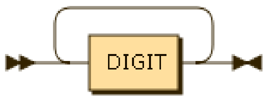


Figure E.3: INT

REAL:

```
{DIGIT}+\.{DIGIT}*
{DIGIT}*\.{DIGIT}+
```

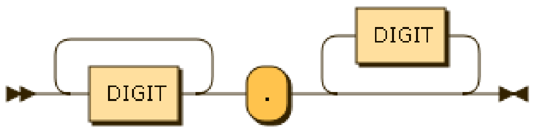


Figure E.4: REAL

STRING:

```
'[^']*'
\"[^\"]*\"
```

White Space:

```
[ \t\n]
```

Comments:

```
"#".*
```

ERROR:

```
.
```

E.2 Stride Grammar

The Stride grammar is presented here.

entry:

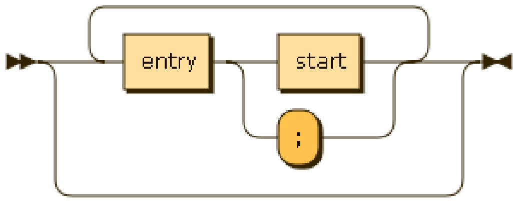


Figure E.5: entry

entry ::= (entry (start | ‘;’))*

referenced by:

- entry

start:

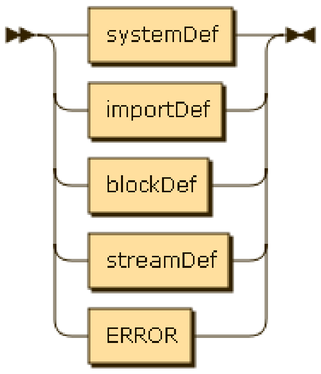


Figure E.6: start

start ::= systemDef | importDef | blockDef | streamDef | ERROR

referenced by:

- entry

systemDef:

languagePlatform

Figure E.7: systemDef

systemDef ::= languagePlatform

referenced by:

- start

languagePlatform:

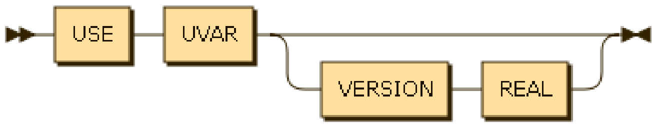


Figure E.8: languagePlatform

languagePlatform ::= USEUVAR (VERSIONREAL)?

referenced by:

- systemDef

importDef:

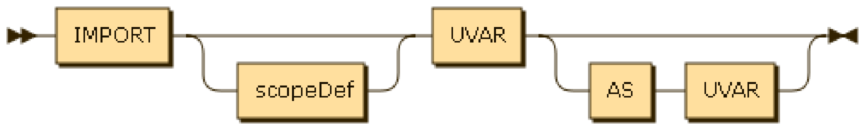


Figure E.9: importDef

importDef ::= IMPORTscopeDef? UVAR (ASUVAR)?

referenced by:

- start

blockDef:

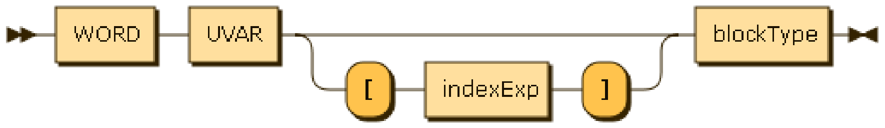


Figure E.10: blockDef

blockDef ::= WORDUVAR ('[' indexExp ']')? blockType

referenced by:

- listDef
- start

blockType:

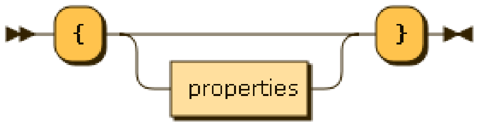


Figure E.11: blockType

blockType ::= ” properties? “

referenced by:

- blockDef
- propertyType

streamDef:

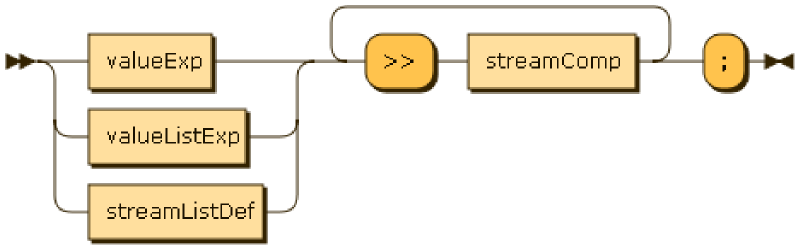


Figure E.12: streamDef

streamDef ::= (valueExp | valueListExp | streamListDef) ('>>' streamComp)+ ';'

referenced by:

- start
- streamListDef

scopeDef:

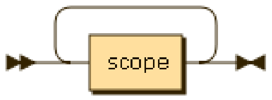


Figure E.13: scopeDef

scopeDef ::= scope+

referenced by:

- bundleDef

- functionDef
- importDef
- indexComp
- streamComp
- valueComp

scope:

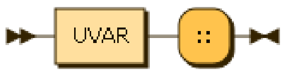


Figure E.14: scope

scope ::= UVAR '::'

referenced by:

- scopeDef

bundleDef:

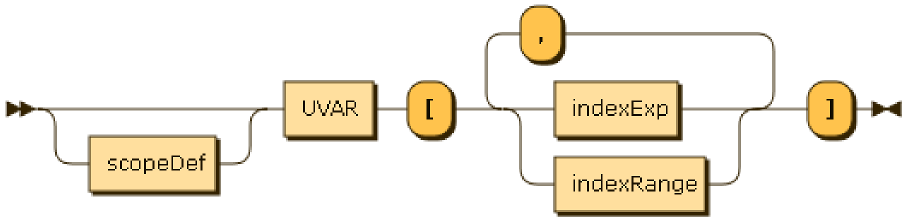


Figure E.15: bundleDef

bundleDef ::= scopeDef? UVAR '[' (indexExp | indexRange) (',' (indexExp | indexRange))* ']'

referenced by:

- indexComp
- streamComp

- valueComp

functionDef:

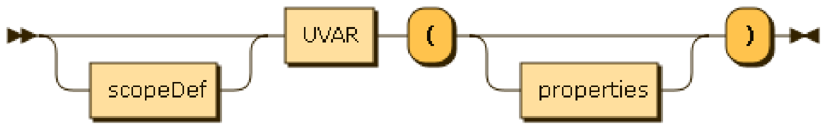


Figure E.16: functionDef

functionDef ::= scopeDef? UVAR '(' properties? '')'

referenced by:

- streamComp
- valueComp

properties:

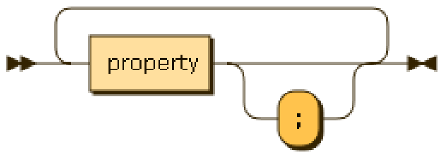


Figure E.17: properties

properties ::= (property ';'?)+

referenced by:

- blockType
- functionDef

property:

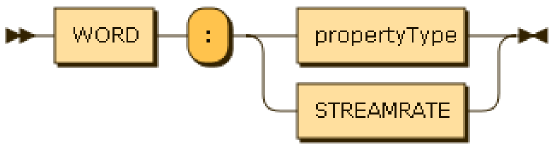


Figure E.18: property

property ::= WORD ':' (propertyType | STREAMRATE)

referenced by:

- properties

propertyType:

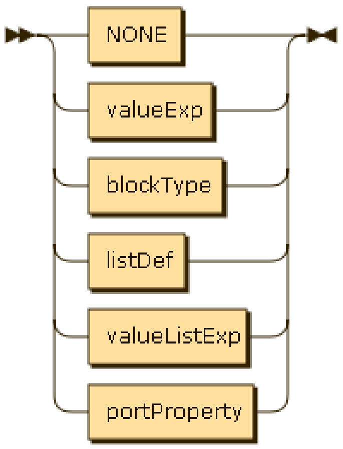


Figure E.19: propertyType

propertyType ::= NONE | valueExp | blockType | listDef | valueListExp | portProperty

referenced by:

- property

portPropertyDef:

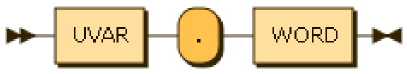


Figure E.20: portPropertyDef

portPropertyDef ::= UVAR '.' WORD

referenced by:

- valueComp

valueListDef:

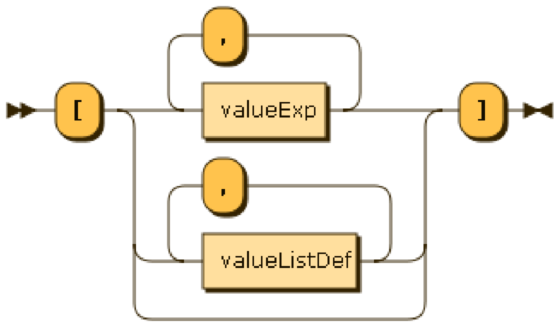


Figure E.21: valueListDef

valueListDef ::= '[' (valueExp (',' valueExp)* | valueListDef (',' valueListDef)*)? "]'

referenced by:

- streamComp
- valueListDef
- valueListExp

listDef:

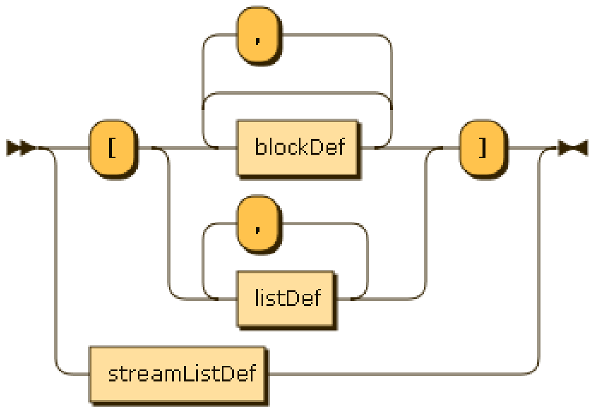


Figure E.22: listDef

listDef ::= '[' (blockDef (','? blockDef)* | listDef (',' listDef)*) "]' | streamListDef

referenced by:

- listDef
- propertyType

streamListDef:

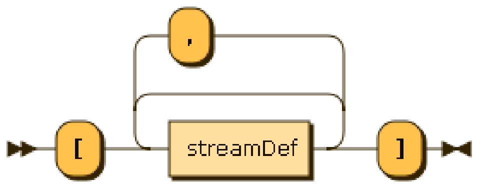


Figure E.23: streamListDef

streamListDef ::= '[' streamDef (','? streamDef)* "]'

referenced by:

- listDef
- streamComp

- streamDef

indexRange:

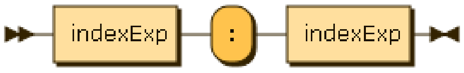


Figure E.24: indexRange

indexRange ::= indexExp ‘:’ indexExp

referenced by:

- bundleDef

indexExp:

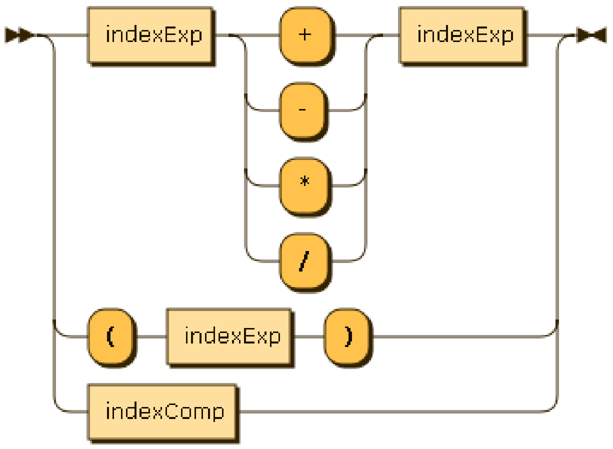


Figure E.25: indexExp

indexExp ::= indexExp (‘+’ | ‘-’ | ‘*’ | ‘/’) indexExp | ‘(’ indexExp ‘)’ | indexComp

referenced by:

- blockDef
- bundleDef
- indexExp

- indexRange

valueListExp:

valueListExp ::= valueListDef (('+' | '-' | '*' | '/' | AND | OR | '&' | '|' | ' ') (valueExp | valueListDef))? | valueExp ('+' | '-' | '*' | '/' | AND | OR | '&' | '|' | ' ') valueListDef

referenced by:

- propertyType
- streamDef

valueExp:

valueExp ::= (valueExp ('+' | '-' | '*' | '/' | AND | OR | '&' | '|' | ' ' |) | '-' | NOT) valueExp | '(' valueExp ")' | valueComp

referenced by:

- propertyType
- streamDef
- valueExp
- valueListDef
- valueListExp

indexComp:

indexComp ::= INT | scopeDef? UVAR | bundleDef

referenced by:

- indexExp

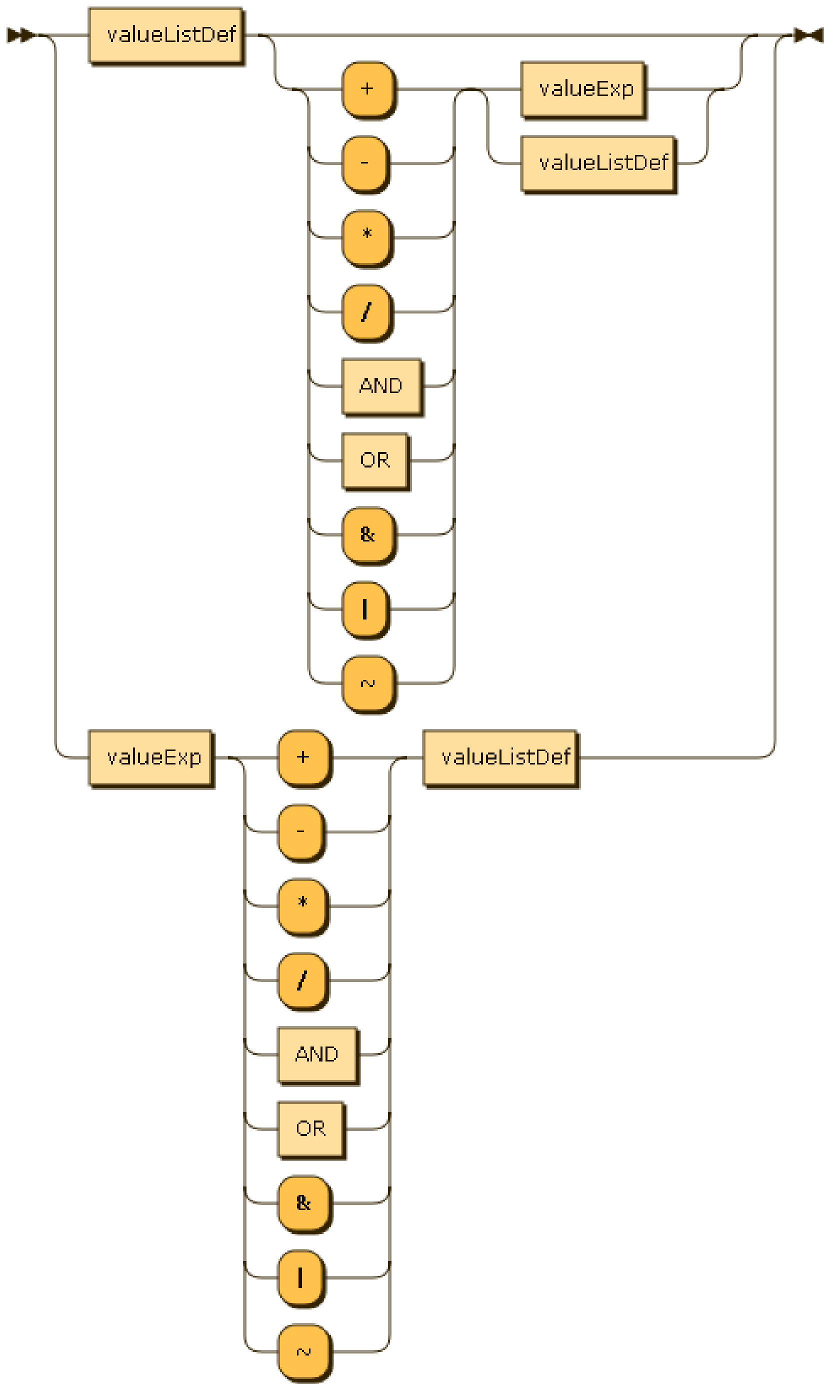


Figure E.26: valueListExp

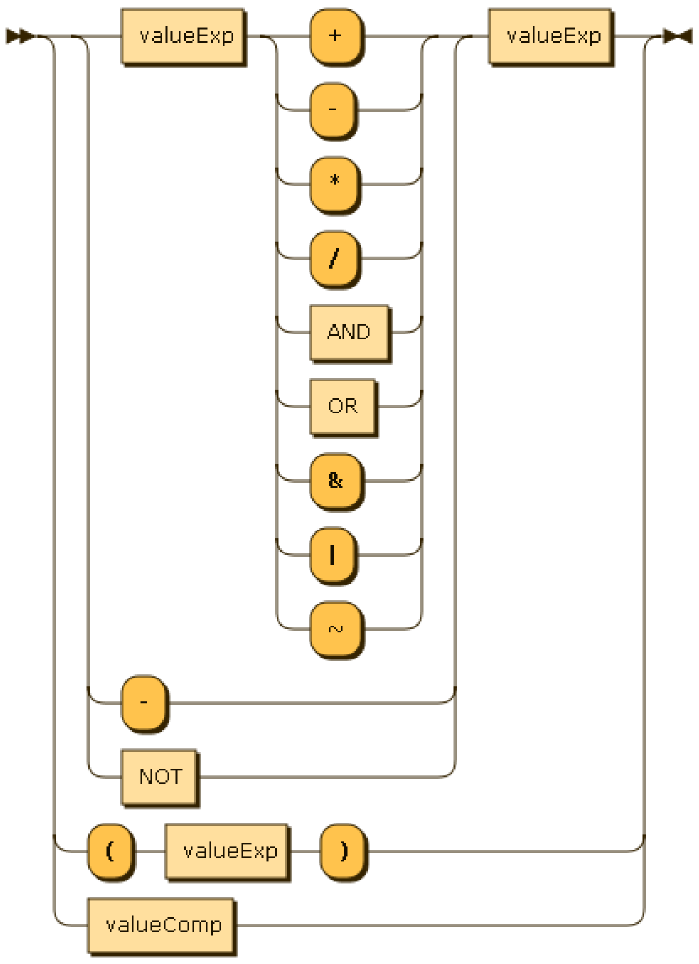


Figure E.27: valueExp

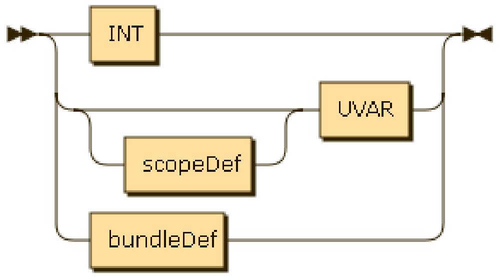


Figure E.28: indexComp

streamComp:

streamComp ::= scopeDef? UVAR | bundleDef | functionDef | valueListDef | stream-ListDef

referenced by:

- streamDef

valueComp:

valueComp ::= INT | REAL | STRING | ON | OFF | WORD | scopeDef? UVAR | bun-dleDef | functionDef | portPropertyDef

referenced by:

- valueExp

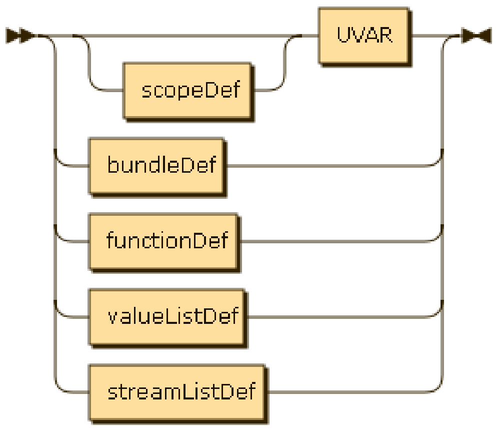


Figure E.29: streamComp

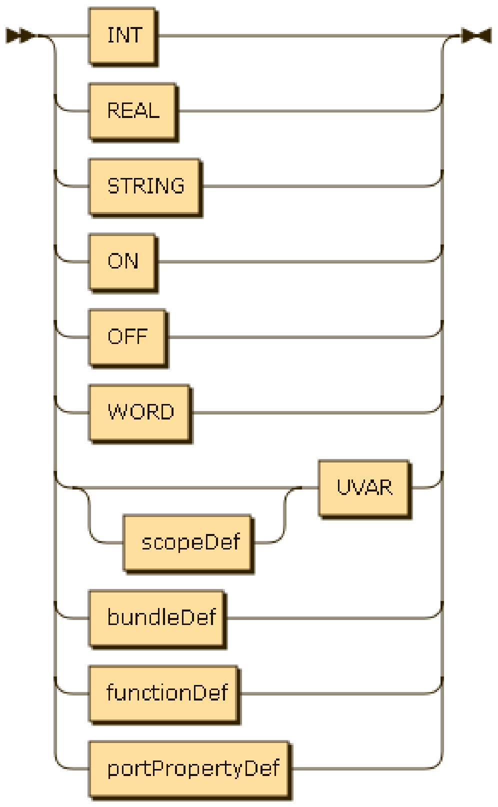


Figure E.30: valueComp

Bibliography

[1] MIDI Manufacturers Association, "The Official MIDI Specifications." `https://www.midi.org/specifications`. [Online; accessed November 7, 2018].

[2] M. Wright and A. Freed, *Open sound control: A new protocol for communicating with sound synthesizers*, in *Proceedings of the 1997 International Computer Music Conference*, (Thessaloniki), 1997.

[3] C. Abbott, *The 4ced program*, *Computer Music Journal* **5** (1981), no. 1 13–33.

[4] J. Moorer, A. Chauveau, C. Abbott, P. Eastty, and J. Lawson, *The 4c machine*, *Computer Music Journal* **3** (1979), no. 3 16–24.

[5] M. Puckette, *The patcher*, in *Proceedings of the 1988 International Computer Music Conference*, (Cologne), 1988.

[6] M. Puckette, *Combining event and signal processing in the max graphical programming environment*, *Computer Music Journal* **15** (1991), no. 3 68–77.

[7] E. Lindemann, M. Puckette, E. Viara, and M. Starkier, *The IRCAM signal processing workstation – An environment for research in real-time musical signal processing and performance*, *Microprocessing and Microprogramming* **30** (1990) 167–174.

[8] C. Roads, *The Computer Music Tutorial*. The MIT Press, Cambridge, Massachusetts, 1996.

[9] B. Vercoe, *A manual for the audio processing system and supporting programs with tutorials*, *Media Lab, MIT* (1986).

[10] N. Bailey, A. Purvis, I. Bowler, and P. Manning, *An implementation of csound on a transputer*, in *Proceedings of the First International Conference on Applications of Transputers*, (Liverpool), 1989.

[11] B. Vercoe and D. Ellis, *Real-time Csound: Software Synthesis with Sensing and Control*, in *Proceedings of the 1990 International Computer Music Conference*, (Glasgow), pp. 209–2011, 1990.

[12] B. Vercoe, *Extended Csound*, in *Proceedings of the 1996 International Computer Music Conference*, (Hong Kong), pp. 141–142, 1996.

[13] M. Puckette, *Pure data*, in *Proceedings of the 1997 International Computer Music Conference*, (Thessaloniki), 1997.

[14] M. Puckette, *Fts: A real-time monitor for multiprocessor music synthesis*, *Computer Music Journal* **15** (1991), no. 3 58–67.

[15] Enzine Audio, "The Heavy hvcc Compiler for Pure Data Patches." `https://github.com/enzienaudio/hvcc`. [Online; accessed November 7, 2018].

[16] J. McCartney, *Supercollider: a new real time synthesis language*, in *Proceedings of the 1996 International Computer Music Conference*, (Hong Kong), pp. 257–258, 1996.

[17] J. McCartney, *Rethinking the computer music language: Supercollider*, *Computer Music Journal* **26** (2002), no. 4 61–68.

[18] Y. Orlarey, D. Fober, and S. Letz, *An algebra for block diagram languages*, in *Proceedings of International Computer Music Conference* (ICMA, ed.), pp. 542–547, 2002.

[19] V. Norilo, *Introducing kronos – a novel approach to signal processing languages*, in *Proceedings of the Linux Audio Conference*, (Maynooth), pp. 9–16, 2011.

[20] V. Norilo and M. Laurson, *Kronos - a vectorizing compiler for music dsp*, in *Proceedings of the 12th International Conference on Digital Audio Effects (DAFx-09)*, (Como), pp. 180–183, 2009.

[21] M. Verstraelena, J. Kuper, and G. Smit, *Declaratively programmable ultra low-latency audio effects processing on fpga*, in *Proceedings of the 17th International Conference on Digital Audio Effects (DAFx-14)*, (Erlangen), 2014.

[22] F. Pfeifle and R. Bader, *Real-time finite difference physical models of musical instruments on a field programmable gate array (fpga)*, in *Proceedings of the 15th International Conference on Digital Audio Effects (DAFx-12)*, (York), 2012.

[23] V. Norilo, M. Verstraelen, and V. Välimäki, *Implementing a low-latency parallel graphic equalizer with heterogeneous computing*, in *Proceedings of the 18th International Conference on Digital Audio Effects (DAFx-15)*, (Trondheim), 2015.

[24] X. Fan, *Real-Time Embedded Systems: Design Principles and Engineering Practices*. Newnes, first ed., 2015.

[25] GRAME, *FAUST Quick Reference*. Centre National de Création Musicale, 0.9.65 ed., January, 2014.

[26] ARM Ltd., “Cortex Microcontroller Software Interface Standard.” `https://arm-software.github.io/CMSIS_5/General/html/index.html`. [Online; accessed November 7, 2018].

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[28] J. O. Smith III, *Spectral Audio Signal Processing*. W3K, 2011.

[29] R. G. Lyons, *Understanding Digital Signal Processing*. Pearson, third ed., 2011.

[30] G. Moro, A. Bin, R. H. Jack, C. Heinrichs, and A. P. McPherson, *Making high-performance embedded instruments with bela and pure data*, in *Proceedings of the 2016 International Conference on Live Interfaces*, (Brighton), 2016.

[31] J. Taelman, “Axoloti.” `http://www.axoloti.com/`. [Online; accessed November 7, 2018].

[32] T. Webster, G. LeNost, and M. Klang, *The owl programmable stage effects pedal: Revising the concept of the on-stage computer for live music performance*, in *Proceedings of the 2014 International Conference on New Interfaces for Musical Expression*, (London), 2014.

[33] R. Boulanger, ed., *The Csound Book: Tutorials in Software Synthesis and Sound Design*. MIT Press, 2000.

[34] G. Wang and P. Cook, *Chuck: A concurrent, on-the-fly, audio programming language*, in *Proceedings of the 2003 International Computer Music Conference*, (Singapore), 2003.

[35] Cycling ’74, “Max visual programming language.” `https://cycling74.com/products/max`. [Online; accessed November 7, 2018].

[36] Y. Orlarey, D. Fober, and S. Letz, *Syntactical and semantical aspects of faust*, *Soft Computing* **8** (2004), no. 9 623–632.

[37] R. Dannenberg, *Machine tongues xix: Nyquist, a language for composition and sound synthesis*, *Computer Music Journal* **21** (1997), no. 3 50.

[38] A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language*. Springer, 2010.

[39] The Qt Company, “Qml applications | qt 5.5.” `http://doc.qt.io/qt-5/qmlapplications.html`. [Online; accessed November 7, 2018].

[40] Multiple Authors, "Flex the fast lexical analyzer." `https://github.com/westes/flex`. [Online; accessed November 7, 2018].

[41] Multiple Authors, "Bison general-purpose parser generator." `https://savannah.gnu.org/projects/bison/`. [Online; accessed November 7, 2018].

[42] J. Levine, *flex & bison*. O'Reilly Media, first ed., 2009.

[43] The Qt Company, "Qt software development framework." `http://doc.qt.io/`. [Online; accessed November 7, 2018].

Terms and Abbreviations

ADC	Analog to Digital Converter
API	Application Programming Interface
AST	Abstract Syntax Tree
CPU	Central Processing Unit
DAC	Digital to Analog Converter
DAW	Digital Audio Workstation
DSP	Digital Signal Processor
FFI	Foreign Function Interface
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
GUI	Graphical User Interface
IC	Integrated Circuit
MIDI	Musical Instrument Digital Interface
OSC	Open Sound Control
SIMD	Single Instruction Multiple Data

Glossary

bare metal	A computer system that does not contain an operating system.
codec	Is a device or computer program for encoding or decoding a digital data stream or signal.
driver	A piece of software that abstracts hardware and enables an operating system or other software to communicate with the hardware.
flash memory	A solid-state non-volatile computer storage medium that can be electrically erased and reprogrammed.
state machine	A model for describing computation, consisting of a set of states and a transition function describing when to move from one state to another.

List of Figures

3.1 CPU cycles required per audio rendering callback for various buffer sizes. 29
3.2 CPU cycles required to process 64 audio samples per audio rendering callback for various buffer sizes. 30
3.3 CPU cycles required per audio rendering callback for a 16-sample buffer size. 31
3.4 CPU cycles required per audio rendering callback for a 16 sample buffer size with a control variable change check. 32
3.5 CPU thread profile showing the impact of a control change and its effect. 32

4.1 Block declaration syntax diagram. 41
4.2 Bundle declaration syntax diagram. 42
4.3 Stream expression syntax diagram. 43
4.4 Module invocation syntax diagram. 44
4.5 CPU cycles required per function call. (Baseline) 48
4.6 CPU cycles required per function call at reduced update rate. 51
4.7 CPU cycles required per function call in asynchronous and reactive mode. 53
4.8 CPU cycles required per function call with an optimized `audioTick` function. 56

5.1 The values of three signal blocks with various rates operating in sample-and-hold mode. 62
5.2 The values of three signal blocks, where signal block `C` is operating in reactive mode. 62

7.1 A trigger resetting a signal. 104
7.2 The domain assignments of blocks and their relationships in Code 7.5. . 107
7.3 Attack/Decay envelope state machine. 127

8.1 Values of `Count` and `DelayedCount`. 132

9.1 The Stride compiler. 160
9.2 The Stride integrated development environment. 163
9.3 Signal to signal connection. 179

9.4 Signal to bundle connection. 180
9.5 Bundle to bundle connection of same size. 180
9.6 Bundle to bundle connection where the size of one is a multiple of the other. 181
9.7 Signal to module to signal connection. 182
9.8 Bundle to modules to bundle connection with implicit expansion of a second module driven by the size of the `Input` bundle. 183
9.9 Bundle to modules to bundle connection with explicit declaration of two modules. 184
9.10 Bundle to modules to bundle connection with different sizes. 185
9.11 Implicit and explicit expansion of modules. 186
9.12 Bundle to multi-input module to signal connection. 186
9.13 Bundle to multi-input module to bundle connection. 187
9.14 Implicit and explicit expansion of multiple modules between bundles of different sizes. 188
9.15 Implicit and explicit expansion of multiple modules between bundles of the same size. 188
9.16 Single module connected to a bundle. 189
9.17 Two generators connected to two outputs. 190
9.18 Two modules connected to another module and then to a bundle of size two. 190

E.1 UVAR 258
E.2 WORD 258
E.3 INT 258
E.4 REAL 259
E.5 entry 260
E.6 start 260
E.7 systemDef 261
E.8 languagePlatform 261
E.9 importDef 262
E.10 blockDef 262
E.11 blockType 262
E.12 streamDef 263
E.13 scopeDef 263
E.14 scope 264
E.15 bundleDef 264
E.16 functionDef 265
E.17 properties 265
E.18 property 266
E.19 propertyType 266
E.20 portPropertyDef 267

E.21 valueListDef 267
E.22 listDef 268
E.23 streamListDef 268
E.24 indexRange 269
E.25 indexExp 269
E.26 valueListExp 271
E.27 valueExp 272
E.28 indexComp 272
E.29 streamComp 274
E.30 valueComp 274

List of Tables

4.1 Improvement in performance with code change. 56

8.1 Values held by the buffer on every clock tick. 132
8.2 Values of signals and the buffer at every clock tick. 147

B.1 Relative computation cost by floating-point operations. (Normalized to addition) 218

List of Codes

3.1 Faust resonant low pass filter with constant arguments. 23
3.2 Faust generated `instanceConstant` method for a resonant low pass filter with constant arguments. 24
3.3 Faust generated `compute` method for a resonant low pass filter with constant arguments. 25
3.4 Faust resonant low pass filter with variable arguments. 25
3.5 Faust generated `instanceConstant` and `compute` methods for a resonant low pass filter with variable arguments. 26
3.6 Excerpts from the platform definition file for the Bela platform. 35

4.1 A simple sine oscillator with frequency control in C. 40
4.2 A block declaration of type `signal` called `Block`. 42
4.3 A bundle declaration of type `signal` and size 2 called `Bundle`. 43
4.4 A stream expression. 44
4.5 Stride code to control the frequency of a sine oscillator. 46
4.6 Generated code for controlling the frequency of an oscillator. 47
4.7 Controlling the frequency of an oscillator at reduced rate. 49
4.8 Generated code for controlling the frequency of an oscillator at reduced rate. 50
4.9 Controlling the frequency of an oscillator reactively. 51
4.10 Code generated for controlling the frequency of an oscillator reactively. 52
4.11 Controlling the frequency of an oscillator with optimized audio callback. 54
4.12 Generated code for controlling the frequency of an oscillator with optimized audio callback. 55

5.1 Three signal blocks with various rates operating in sample-and-hold mode. 61
5.2 Signal block `C` operating in reactive mode. 61
5.3 Signal block `A` assigned to `SetupDomain`. 64
5.4 Generated C code from Code 5.3. 65
5.5 Generated Python code from Code 5.3. 65
5.6 Sine oscillator stream expressions in Stride. 66
5.7 The oscillator output and its frequency update synchronously. (Stride) . 68
5.8 The oscillator output and its frequency update synchronously. (C) 69

5.9 The oscillator output and fits requency update asynchronously. (Stride) 71
5.10 The oscillator output and fits requency update asynchronously. (C) . . . 72
5.11 Sine oscillator module with frequency control in Stride. (SineOsc) 74
5.12 C++ class generated for the SineOsc module in Code 5.11. 79
5.13 Level module in Stride. 81
5.14 C++ class generated for the Level module in Code 5.13. 82
5.15 Synchronous frequency modulation using SineOsc and Level modules. . 83
5.16 C++ code generated for synchronous frequency modulation. 85
5.17 Asynchronous frequency modulation using SineOsc and Level modules. 86
5.18 C++ code generated for asynchronous frequency modulation. 87

6.1 Domain triggering for sequential and parallel execution. 91
6.2 Mutual exclusion scheme and synchronization policy. 93
6.3 C++ code generated for asynchronous frequency modulation with concurrency. 95

7.1 Switch block declaration. 100
7.2 An example of a switch controlling the state of a module. 100
7.3 Trigger block declaration. 101
7.4 An example of a trigger resetting a signal. 103
7.5 An example with triggers in two domains. 105
7.6 Counter module in Stride. 108
7.7 C++ class generated from the Counter module. 110
7.8 Excerpts of the C++ code generated for triggers across two domains example. 112
7.9 Default reaction block declaration. 118
7.10 A reaction to double the frequency of an oscillator every second. 119
7.11 Attack/Decay envelope module in Stride. 120
7.12 C++ class generated from the Attack/Decay envelope module. 123

8.1 Buffer block declaration. 130
8.2 A buffer block used as a delay line. 131
8.3 Per sample operation performed on data contained in buffer blocks. . . 134
8.4 Vector operations on audio input and output buffers. 136
8.5 A signal offsetting module. 137
8.6 A buffer offsetting module. 138
8.7 Adding an offset to a signal in Stride. 140
8.8 Adding an offset to a buffer in Stride. 141
8.9 C++ code generated for offsetting a signal. 142
8.10 C++ code generated for offsetting a buffer. 142
8.11 Loop block declaration. 143
8.12 Sum loop in Stride. 145

8.13 Summing signals in a bundle. 146
8.14 Summing data in a buffer. 146
8.15 Average module block in Stride. 147
8.16 Group block declaration. 149
8.17 Resonant low pass module in Stride. 150
8.18 C++ class generated from the resonant low pass module. 152

9.1 Constant block declaration. 165
9.2 Shorthand constant block declarations. 166
9.3 Signal block declaration. 167
9.4 Switch block declaration. 168
9.5 Buffer block declaration. 170
9.6 Trigger block declaration. 171
9.7 Bundle declaration. 172
9.8 Module block declaration. 173
9.9 Reaction block declaration. 174
9.10 Loop block declaration. 176
9.11 Group block declaration. 177
9.12 Signal to signal connection. 178
9.13 Signal to bundle connection. 179
9.14 Expanded signal to bundle connection. 179
9.15 Bundle to bundle connection of same size. 180
9.16 Expanded bundle to bundle connection of same size. 180
9.17 Bundle to bundle connection where the size of one is a multiple of the other. 181
9.18 Expanded bundle to bundle connection where the size of one is a multiple of the other. 181
9.19 Signal to module to signal connection. 182
9.20 Implicit expansion of a second module driven by the size of the `Input` bundle. 183
9.21 Expanded bundle to module to bundle connection. 183
9.22 Explicit declaration of two modules. 184
9.23 Expansion of bundle to modules to bundle connection. 184
9.24 Bundle to modules to bundle connection with different sizes. 185
9.25 Implicit and explicit expansion of modules. 185
9.26 Bundle to multi-input module to signal connection. 186
9.27 Bundle to multi-input module to bundle connection. 186
9.28 Implicit and explicit expansion of multiple modules between bundles of different sizes. 187
9.29 Implicit and explicit expansion of multiple modules between bundles of the same size. 187
9.30 Single module connected to a bundle. 189

9.31 Two modules connected to a bundle of size two. 189
9.32 Expanded form of two modules connected to a bundle of size two. . . . 189
9.33 Two modules connected to another module and then to a bundle of size two. 190

A.1 Faust resonant low pass filter with constant arguments. 200
A.2 Generated C++ code for resonant low pass filter with constant arguments.200
A.3 Faust resonant low pass filter with variable arguments. 203
A.4 Generated C++ code for resonant low pass filter with variable arguments.204
A.5 Faust platform definition file for the Bela platform. Source file: (`bela.cpp`).207

B.1 C++ code to measure relative computation cost. 218

C.1 Synchronous frequency modulation in Stride using RtAudio and Boost libraries. 224
C.2 Generated C++ code for synchronous frequency modulation. 225
C.3 Asynchronous frequency modulation in Stride using RtAudio and Boost libraries. 231
C.4 Generated C++ code for asynchronous frequency modulation. 232
C.5 Asynchronous frequency modulation in Stride using RtAudio and Boost libraries with concurrency control. 239
C.6 Generated C++ code for asynchronous frequency modulation with concurrency. 240

D.1 Synchronization class. 248
D.2 Signal interface class. 249
D.3 Single domain read, write, and reset signal class. 250
D.4 Single domain read, write, and multi domain reset signal class. 250
D.5 Trigger observer interface class. 252
D.6 Trigger observer block class. 252
D.7 Trigger interface class. 252
D.8 Single domain trigger-controlled trigger class. 252
D.9 Multi domain trigger-controlled trigger class. 253
D.10 Switch controlled trigger class. 254