April 2011

The Shogun MIDI Control System

Benjamin Michael LaVerriere
Worcester Polytechnic Institute

Project Number: MXC–0310

The Shogun MIDI Control System

A Major Qualifying Project Report
submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
by
Ben LaVerriere

28 April 2011

Approved:

Professor Michael J. Ciaraldi, Advisor

Abstract

This report details the design and development of the Shogun MIDI control system, which helps keyboardists send MIDI commands quickly and with little possibility of error. Emphasis is given to the design choices made in the course of this project, including the implementation of a custom configuration-file syntax and parser. Also included in this report are an overview of relevant MIDI concepts, suggestions for future work, and documentation for the entire Shogun codebase.

Keywords: MIDI, Software, Design

Contents

List of Figures

Source Code Listings

Chapter 1

Project Overview

When Claude-Michel Schönberg composed *Les Miserables*, he created what would become the longest-running musical of all time. When John Cameron orchestrated *Les Miserables*, he created both a score for the ages and a logistical challenge for keyboardists. The score calls for two keyboardists — nothing unusual in the modern age of musical theatre — who originally would play the show on the Yamaha DX7 synthesizer. However, the state of the art in 1983 meant that each keyboard only had enough memory to hold 32 distinct voices, or patches, each corresponding to a particular type of sound. As a result, each keyboard player would use two DX7s, stacked on top of each other, to accommodate a full 64 patches each.

In modern productions, the problem of memory limitations has all but evaporated, and keyboardists have little trouble programming or finding a diverse collection of voices to use in *Les Miserables*. Newer keyboards present a different complication, however. The DX7 could easily provide a set of 32 buttons on its front panel for quick patch-changing. A modern synthesizer, on the other hand, could never provide a front-panel button for each of its voices, as these number in the hundreds, if not thousands. For the modern theatrical musician, the question becomes one of accessibility: how can a keyboardist switch between a large number of presets with speed and accuracy?

Since a keyboard must provide some way of accessing its patches — usually a number pad or some similar device on the front panel — a daring keyboardist could use the "native" patch-selection method during a live show. For a show with a small number of keyboard sounds, this may work very well. On the other hand, even if the show only uses a few patches, the keyboardist may still make a mistake and end up playing an accordion part with

a gritty, loud synth patch.[1]

The possibility of making a mistake while changing patches is a very real one, even for skilled musicians. Musical theatre orchestras often play in near-darkness, and must adapt to fluctuating tempos, inaudible singers, and the occasional skipped verse. Consequently, an important metric for any patch-change solution is its ability to prevent accidental or incorrect patch changes. Keyboardists, and to some degree keyboard manufacturers, have developed a number of solutions to this problem over the years. Some of the more promising options are discussed in Chapter 3.

The existing solutions to live patch-changes are not ideal for the musical-theatre keyboardist. While the possibility of stepping through a sequence of voices exists for some of these devices, many appear to have been designed to provide quick *random* access to a set of sounds rather than *sequential* access. A device that advertises "up to 100 voices" may be great for a keyboardist in a Beatles tribute band, but a series of a hundred patch changes would only cover the first act of *Les Miserables* at best.

The fact that many existing solutions require the performer to purchase specialized hardware — particularly when that hardware is expensive and has more features than simply changing patches — provided additional motivation for the Shogun project. As discussed in Chapter 3, a solution that could run on a wide variety of computers, operating systems, and so on would be less expensive, more widely usable, and could potentially make it easier for users to program their own lists of patch changes.

It was from these limitations that Shogun was born. At its core, Shogun is a piece of software that sends patch-change messages, as defined previously by the user, to one or more devices in real time. Specifically, Shogun reads a list of patch changes from a file, which specifies an order for these data as well as (optionally) some metadata about the file and its purpose. The user then uses the Shogun interface (which includes keyboard controls as well as a graphical interface) to navigate through the patch list, with each patch-change selection being sent to the appropriate connected devices. The interface also provides mechanisms for handling various exceptional situations: if the user needs to jump to a particular voice out-of-sequence, for example.

Shogun was originally written as a hacked-together Python script that was very closely tailored to the author's particular equipment. While this system provided a great increase in convenience, it was far from usable on a wide range of synthesizers (or even generic MIDI devices) and there existed a great

[1] The author can neither confirm nor deny whether this example comes from personal experience.

deal of additional functionality that could be added to make the system more intuitive and failure-resistant. Chapter 4 describes the current feature set of the SHOGUN system, and explains the ways in which the current system is more widely-usable than its shoddy predecessor.

A number of options for future development of the SHOGUN system also exist, as discussed in Chapter 7. For example, SHOGUN will may eventually provide some setup-testing functionality, allowing the user to verify that the proper data are being sent to the appropriate device(s) in the system.

This paper will provide a brief overview of the Musical Instrument Digital Interface (MIDI), the protocol used to communicate between the system running SHOGUN and the associated devices. SHOGUN only uses a limited subset of MIDI, so at worst its users would only need a rudimentary knowledge of the protocol. However, even this minimal amount of technical 'wizardry' may be discouraging to some users; the SHOGUN system will include a method of composing patch-change files that requires no MIDI knowledge beyond the concept of assigning numbers to distinct patches.[2]

After discussing the basics of MIDI, this paper will review some of the existing live-patch-change solutions in use today. These include commercial products as well as custom-built systems; in each case, we will note both the useful features of the solution as well as its shortcomings with regard to SHOGUN's intended purpose. Similarly, this paper will provide a survey of the available programming toolkits that deal with MIDI data. As before, this evaluation will consider both the advantages of each toolkit as well as its appropriateness for SHOGUN in particular.[3]

Next, we will discuss some of the design challenges particular to the SHOGUN project. These challenges include:

- the design of a user interface for time-critical use that prevents accidental or erroneous input
- the design of a method of patch-file construction that both affords technically-skilled users a high degree of flexibility and allows novice users to use the system quickly and easily
- the balance between providing a flexible MIDI-enabled tool and creating a bloated, frustrating piece of software.

[2] That is, patch `001` may correspond to "Acoustic Guitar," patch `002` to "Flute," and so on. These values are defined for each synthesizer, either by the manufacturer or, in the case of custom-programmed patches, by the user.

[3] 'Appropriateness' in this case includes the author's familiarity with the language in which a toolkit is written, a factor which significantly affects the development process.

This material is followed by a discussion of SHOGUN's internal data structure and functionality, and a review of the more salient features of the development process. Finally, the various possibilities for future development are described.

Chapter 2

Introduction to MIDI

While the MIDI protocol has been relatively successful, as protocols go, MIDI software remains somewhat of a niche market. Because SHOGUN is, even in this narrow context, a tool with a narrow focus, only a brief overview of the MIDI protocol is required to understand SHOGUN's functionality. In this chapter, we discuss the fundamentals of MIDI communication, the specifics of SHOGUN's MIDI needs, and the particular MIDI messages used by SHOGUN. The final portion of this chapter contains a survey of many MIDI programming toolkits and libraries, along with a discussion of their applicability to the SHOGUN project.

2.1 MIDI Overview[1]

The MIDI standard defines a protocol (and corresponding physical interface) to enable communication between devices using a musical vocabulary. MIDI was originally used to link synthesizers, sound generators, and similar devices, and so the protocol can express many facets of a musical performance: the pitch, volume, and duration of a note, for example, or more nuanced attributes like vibrato.

However, because MIDI was developed in the context of electronic music, rather than simply music in general, the protocol includes features that relate to the low-level mechanisms involved in configuring electronic musical instruments as well as producing sound with them. For example, the desire to produce a wide variety of sounds using a single device corresponds to the part of the MIDI standard with which SHOGUN is primarily concerned: pro-

[1] The description of MIDI in this section is technically based on Rumsey (1990), although nearly every book written about MIDI includes some variant of this material.

gram changes.[2] According to the MIDI standard, a device should respond to a 'program change' message by loading a different voice or waveform — in other words, by changing the type of sound it produces. These sounds are often predefined by the manufacturer of the device, but depending on the type of device, may also be programmable by the user. It is important to note that the actual sound data for a patch, be it a waveform with modulation, a set of samples, or some other type of sound-storage, is neither generated nor interacted with by MIDI data.[3]

A single MIDI device can send data to one or more of sixteen MIDI *channels*, and devices may in turn "listen" on one or more channel as well. Consider a musician with a MIDI-capable device that can produce many kinds of sounds at once, and is able to respond to data on multiple MIDI channels. That musician could then connect multiple MIDI controllers[4] to the sound-generating device. If each controller is set up to send its data on a different MIDI channel, the device will be able to assign a different sound to the notes played on each controller.

2.2 Shogun-specific MIDI Messages

The particular MIDI messages used by Shogun are shown in Figure 2.1. Each message defined by the MIDI specification consists of one *status byte* followed by one or more *data bytes*. Status bytes always have a most-significant bit of `1`, while data bytes begin with `0`. Bits 2–4 of a status byte indicate which status is being sent, and the remaining four bits indicate which of the sixteen MIDI channels is being addressed.

A program-change message (status `100`) needs only one data byte, which specifies which of the 128 available patches is to be loaded. The bank-change message, in contrast, requires two data bytes. This is not because it has any larger a selection space, but rather because "bank select" is one of the many *controllers* defined in the MIDI standard.[5] A wide variety of these controllers exist, and status bytes beginning with `1011` simply indicate that some sort of

[2] Because of the conflicting use of the word "program" between software design and MIDI communication — that is, whether "program" means a piece of software or a MIDI voice — I shall avoid using "program" at all and favor "patch" or "voice" for the MIDI term and "system" or "application" for the software concept.

[3] There is one exception to this statement: some devices can be triggered by MIDI to "dump" their sound data for one or more patches over their MIDI OUT port. However, such an action is not used frequently, and does not appear in the Shogun system.

[4] A MIDI controller is any device capable of producing and sending MIDI data.

[5] Note that the use of the term "controller" in this context is distinct from the previous use. Here a "controller" is an internal, adjustable parameter in a MIDI device, not a piece of hardware that generates MIDI data.

Figure 2.1: Bit-level diagram of those MIDI messages used by SHOGUN. Bits indicating whether a byte is a status byte or a data byte are shown in gray boxes, bits identifying a particular status byte in pink, and bits specifying a channel number in green (with data nnnn).

control change will follow. The first data byte after a control-change status byte specifies which controller will be modified, and then the next byte(s) specify the particular data for that controller. Thus, to effect a bank change (controller 0), status byte `1011nnnn` is followed by data byte `00000000` and then a byte to specify which of the 128 available banks is to be loaded. (Here `nnnn` refers to the four-bit MIDI channel specification present in each status byte.)

It may be worthwhile to note at this point a subtle detail of the MIDI standard, specifically, that a bank-change message *alone* should not result in any audible change from the target device. Only when a device has received a program-change message (optionally preceded by a bank-change message) should it load the new sound or setting. To prevent users of SHOGUN from either needing to know this subtlety of the MIDI standard or experiencing unexpected results when stepping through a patchfile, SHOGUN's patchfile syntax and MIDI backend make it impossible to send an isolated bank-change message. In other words, for each step, SHOGUN will always send a bank-change message *and* a program-change message.

One final note: the bank-change message described above sends a byte that will be used as the *most-significant byte* by the receiving device. A second controller (#32) exists with which the *least-significant byte* can be specified, but most MIDI devices of recent manufacture use only the MSB command. As

such, SHOGUN currently only supports bank changes with controller #0. The ability to specify one or more custom bank-change commands may exist in a future version of SHOGUN; such developments are discussed in Chapter 7.

2.3 MIDI Programming Toolkits

While MIDI programming remains somewhat of a niche market, the popularity of the MIDI protocol itself has led to the development of a great number of MIDI toolkits for a wide variety of programming languages. In this section, we review some of these toolkits, indicating first those toolkits whose functionality and documentation distinguished them as candidates for use in the SHOGUN development process. Following the discussion of preferred toolkits is a cursory discussion of a number of less-appropriate choices, including the reasons for which they were discarded from consideration.

2.3.1 Preferred Toolkits

pyPortMidi `http://alumni.media.mit.edu/~harrison/pyportmidi.html`
pyPortMidi (Harrison, 2010) is a Python port of the cross-platform PortMidi C/C++ library, which is described below. The original version of SHOGUN (described in Chapter 1) used this library, and experience suggests that it is a reasonably reliable toolkit. In addition, the library can be used on Windows, Linux, and OS X systems, and the distribution package includes compiled versions of the code for each of these platforms. One possible disadvantage of this library is that its documentation (`http://cratel.wichita.edu/cratel/cratel%20pyportmidi`) is minimal, and the project appears to have had no significant development since 2008.

RtMidi `http://www.music.mcgill.ca/~gary/rtmidi/`
According to its documentation, RtMidi is "a set of C++ classes (RtMidiIn and RtMidiOut) that provides a common API (Application Programming Interface) for realtime MIDI input/output across Linux (ALSA), Macintosh OS X, SGI, and Windows (Multimedia Library) operating system" (Scavone, 2011). The library appears to remain under development — or perhaps "maintenance" is more appropriate — with the most recent update being released in January of 2010. The author provides not only class-, file-, and method-level documentation, but also some well-written and comprehensive tutorials showing potential uses of the toolkit. Because of the sufficient (and clear) functionality and the excellent documentation, RtMidi was selected as the MIDI

library used in the SHOGUN project.

pyrtmidi `http://trac2.assembla.com/pkaudio/wiki/pyrtmidi`
pyrtmidi is a Python wrapper for RtMidi whose API is copied "*near* verbatim from the C++ code" (Stinson, 2010). As such, a programmer can take advantage of RtMidi's pleasant documentation and functionality, while continuing to develop in Python. Since SHOGUN was originally written in Python, this approach did have some appeal. That being said, with RtMidi already being such a strong library, I was concerned that adding another layer of abstraction/porting would be more likely to introduce complications than to improve the programming experience.

PortMidi `http://portmedia.svn.sourceforge.net/viewvc/portmedia/portmidi/trunk/pm_common/portmidi.h?view=markup`
PortMidi is one of the more well-known MIDI programming toolkits, and is, with RtMidi, one of the two main choices for C/C++ MIDI programming (PortAudio Developers, 2009). Cross-platform compilation is possible, and PortMidi appears to provide the necessary MIDI features (and more) for the SHOGUN system. However, after examining the documentation for both systems, and after reading a comparison of the two libraries (Capocasa, 2006), it seems clear that RtMidi is not only an excellent library on its own, but it also surpasses PortMidi for ease of use and performance.

jdksmidi `http://github.com/jdkoftinoff/jdksmidi`
Despite the presence of 'JDK' in the name, this library is a general-purpose C++ MIDI toolkit. It appears to provide an ample feature set, placing it in the same category as RtMidi and PortMidi with regard to functionality (Koftinoff, 2011). However, despite the README's request that users "`please see the documentation in the subdirectory: docs`," I have been unable to find any actual documentation. Source files appear to be minimally-commented, so the in-place documentation does not make up for the missing `docs` folder.[6]

2.3.2 Other Toolkits

The following libraries have one or more characteristics or deficiencies that make them less attractive for use in the SHOGUN project. As such, they are

[6] In fact, the developers provide a link that, they claim, contains documentation generated by the Doxygen utility, but this link is currently broken.

listed here with minimal description, with focus being given to the reason for their being discarded.

portmidizero `http://gitorious.org/portmidizero/`
portmidizero is "a simple ctypes wrapper for PortMidi in pure python," but it is also minimally-documented and therefore inferior to pyPortMidi, another Python wrapper for PortMidi (enoki, 2009). That portmidizero requires a platform-specific dynamic library for PortMidi, and that these libraries are not packaged with the toolkit, make it even less attractive for this project.

pygame.midi `http://www.pygame.org/docs/ref/midi.html`
The Pygame project itself is well-established and widely used for Python multimedia/game development (PyGame Developers, 2010). The MIDI module, however, is simply a wrapper for the `pyportmidi` library, itself a Python port of the PortMidi toolkit. If I want to use [py]PortMidi, I see no reason to include another level of abstraction.

PyMidi `http://hyperreal.org/~est/python/MIDI/`
This Python library is designed solely for responding to MIDI input, and as such cannot possibly meet the needs of the Shogun system (Tiedemann, 2000).

Python Midi Package `http://www.mxm.dk/products/public/pythonmidi`
This package is listed as "experimental" and cannot interface with MIDI ports for realtime data sending or receiving (Max M, 2005). As with PyMidi, above, this library cannot possibly satisfy the needs of the Shogun system.

javax.sound.midi `http://download.oracle.com/javase/1.4.2/docs/api/javax/sound/midi/package-summary.html`
This toolkit appears to be the most prominent, if not the only, MIDI toolkit for Java (Oracle, 2010). Possibly because of the "heaviness" of the JVM, Java does not appear to be widely used for MIDI software, and I was unable to find many non-trivial examples of the `javax.sound.midi` library in action. It may not be the most concrete of reasons, but it seems that this library exists so that Java can say it supports MIDI, not because there is great demand for MIDI programming in Java.

C# MIDI Toolkit `http://www.codeproject.com/KB/audio-video/MIDIToolkit.aspx`

A brief review of this and other documentation for the C# MIDI Toolkit suggests that the toolkit itself is reliable and has been used by a number of projects (Sanford, 2007). However, the package appears not to provide any dramatically different or more usable features than other, similar libraries. Because I do not currently have any significant familiarity with C#, I believe it would be counterproductive to use this package when equivalent (or better) options exist for languages with which I have already worked.

midi-dot-net `http://code.google.com/p/midi-dot-net/`
As with the C# MIDI Toolkit, this library seems reasonably useful, but requires the use of a language and framework with which I am not familiar (Lokovic, 2009). The fact that there is no option for cross-platform porting only serves to make it less attractive for use in the SHOGUN project.

CMU MIDI Toolkit `http://www.cs.cmu.edu/~music/cmt/`
This toolkit, while apparently very useful in its day, is self-professedly out-of-date — an easy rejection (Dannenberg, 93).

MusicNoteLib `http://gpalem.web.officelive.com/CFugue.html`
As the name suggests, this C/C++ library deals only in MIDI notes, functioning as a "beautiful abstraction that lets you concentrate on... the *Music* [sic] rather than worry about the MIDI nuances" (Palem, 2010). Because SHOGUN's main functionality involves non-note MIDI data, this library cannot serve the needs of the project.

Arduino MIDI Library `http://www.arduino.cc/playground/Main/MIDILibrary`
As a point of interest, a later version of SHOGUN could possibly be developed on the Arduino microcontroller platform (Best, 2011). Such a design could incorporate a custom physical interface, possibly based on the interface research and design to be included in this project.

Chapter 3

Existing Solutions

Having existed since the early 1980s, the MIDI protocol has spawned all manner of MIDI-enabled instruments, devices, and gadgets. As a result, the modern keyboardist does have a few options to achieve live patch changes with greater convenience than manual entry. A brief survey of some of these systems will serve to demonstrate the need for a tool with Shogun's functionality by illustrating the shortcomings present in the existing solutions.

Probably the most frequently-mentioned live patch-change manager is the Opcode Studio 5, a MIDI interface first manufactured in 1991 for the Macintosh platform (Gallant, 2008; Peck et al., 2007). The Studio 5 has 15 MIDI IN ports and 15 MIDI out ports, allowing for communication with up to 240 MIDI channels, and could be programmed to send a wide variety of MIDI messages according to user input. However, Opcode ceased manufacturing in 1999, and those units that still exist were only designed to work with versions 7, 8, and 9 of the Macintosh OS — not the 10.* series currently in use. The functionality of the Studio 5 remains highly desirable, but the impracticality of obtaining and maintaining the system (including an appropriately old Macintosh) takes it out of consideration for most musicians.

A few devices of more modern manufacture offer some of the same features of the Studio 5. The Roland FC-300 is a hardware device that acts as a multipurpose MIDI controller for keyboardists, and can be used to access a set of predefined patches (Roland Corporation, 2007). However, this list is limited to 100 entries, and the device's interaction style seems more suited to random repeated access than stepping through a long sequence of voices. (The latter style of interaction requires less thought on the part of the user, and is therefore preferable when dealing with a long sequence of patch changes.) In addition, because the FC-300 is designed to provide a much wider range of

functionality than SHOGUN, it sells for around $400, putting it out of consideration for many amateur or semiprofessional musicians.

The Genovation MIDI Patch Changer is probably the best commercially-available solution to the problems SHOGUN seeks to solve. The form factor, as shown in Figure 3.1, certainly reflects a dedication to the single purpose of effecting patch changes. However, as with the FC-300, the number of predefined patches that can be accessed sequentially (or randomly) is limited, here to 100 entries (Genovation, 2010). Additionally, the process of programming the Patch Changer appears quite complex, requiring some detailed knowledge of MIDI and the use of a somewhat unattractive and forbidding interface.

Figure 3.1: The Genovation MIDI Patch Changer

Finally, for the daring "power" user, there exists the possibility of constructing a custom MIDI-enabled device to enact live patch charges. The uCApps/MIDIbox project provides a development platform for MIDI gadgets of all sorts, including patch-changers (Klose, 2011). Similarly, the Arduino MIDI library discussed in Chapter 2 could be used to develop a custom device to accomplish SHOGUN's goals.

If building a custom electronic device seems more promising than any extant device, however, why not simplify matters and create a custom software

solution? Even though the open-source hardware movement is growing and thriving, a piece of software remains easier to distribute and, for most end users, easier to implement than even the clearest and most detailed schematics.

If this software is suitably platform independent, it can avoid the fate of the Studio 5, and if the project is open-source, development could continue to adapt the software to any number of future platforms. A USB-to-MIDI interface can be obtained relatively cheaply, as can a bare-bones laptop, if needed. Such a system is within the reach of many musicians, and will be sufficiently powerful to run SHOGUN.

Chapter 4

Features and Functionality

Because SHOGUN was created to address a particular need, it has always had a relatively small feature set. The idea of *simple* software to serve a *particular purpose* has been, and should continue to be, a key principle guiding SHOGUN's development.

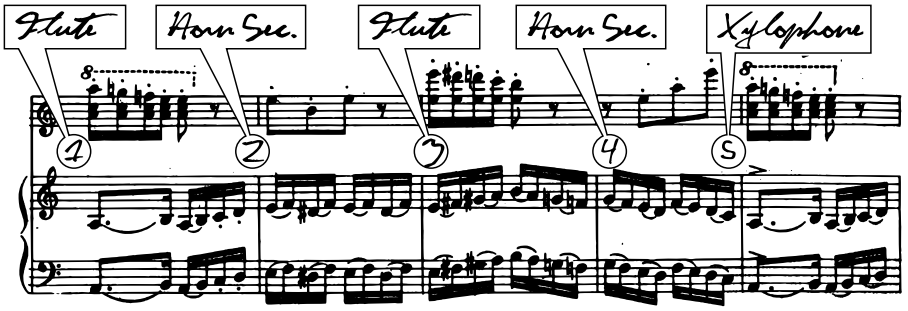
MIDI communication is at the center of SHOGUN's functionality. In particular, SHOGUN must be able to generate MIDI program-change and bank-change messages and transmit these to a MIDI device driver to be output. (A detailed discussion of these messages can be found in Chapter 2.) Additionally, SHOGUN must be able to send these messages to any MIDI channel, and provide a convenient method by which users can specify to which channel each message should be sent.

The operating system-level details of interacting with MIDI device drivers can be handled by one of the many extant MIDI programming libraries. This fact leaves SHOGUN the tasks of constructing the appropriate messages, addressing them to the specified channels, and of course providing syntax allowing users to control these operations.

We call each set of MIDI messages to be sent by SHOGUN a *step*, and a sequence of these steps a *patch list*. Each step may address one or more MIDI channels, and will be sent when indicated by the user. Therefore, SHOGUN must allow users to navigate through a patch list in a few ways. Sequential navigation is the most common case, and SHOGUN should allow users to advance one step at a time using either the graphical interface or the keyboard. (The same is true, naturally, for navigating backwards one step at a time.) Because live theatre is replete with the unexpected, SHOGUN should also allow users to jump to an arbitrary step in the patch list by its numeric position in the sequence. (If it seems unusual to require users to know the ordinal

of each step, recall that most users will have already annotated their musical scores with patch changes, and it is a simple task to number some or all of these steps. An example of how users might annotate a score can be found in Figure 4.1.)

Figure 4.1: A sample passage from a musical score, annotated with numbered patch changes.



In the first version of SHOGUN, the available syntax corresponded directly to the specific labels and properties of the author's own keyboard. For example, banks were addressed by the names provided on the keyboard itself — 'A,' 'B,' 'C,' and so on — with a hard-coded 'translation' to numeric MIDI bank values in the SHOGUN software. An important goal of this project has been that SHOGUN should be usable on as many MIDI devices as possible, in compliance with the MIDI standard. For example, meeting this goal requires a method of specifying steps that is not device-specific but still allows for semantic richness from the user's perspective.

The simplicity of SHOGUN's functionality should be reflected in its user interface as well. Only those interface elements necessary to SHOGUN's essential functionality or for displaying the current MIDI system state should be allotted screen space, and primacy should be given to those elements most useful at a quick glance.

Many MIDI software applications include some variant on the "MIDI analyzer" functionality — the ability to monitor MIDI input, often with filtering capabilities, as a diagnostic or troubleshooting tool. Since all of SHOGUN's primary functions are possible with MIDI output alone, it was decided that a full MIDI analysis function would be outside SHOGUN's purview. However, it would be useful if SHOGUN could assist users in verifying that their MIDI

devices were properly connected for use with SHOGUN, and so a "diagnostic mode" interface is intended for future development of the SHOGUN application (and discussed in Chapter 7).

This concise set of operations should allow users to perform satisfactory MIDI-network troubleshooting without overcomplicating SHOGUN itself.

Chapter 5

System Architecture

5.1 Data Flow

From the user's perspective, the *patch-change step* is the atomic element of a SHOGUN production. Each step represents one or more voices to be sent to one or more MIDI devices. These steps, along with various metadata, are defined in SHOGUN *patchfiles*, which are discussed in greater detail below in Section 5.1.1.

When the user loads a patchfile into SHOGUN, the textual data are parsed into SHOGUN's internal storage format, discussed in Section 5.1.2.

From that point, the user may step forward and backwards through the sequence of steps as desired, or jump to an arbitrary step by number. At each new step, SHOGUN sends the corresponding MIDI data to whichever MIDI output device has been selected by the user. These data may be sent over a single channel or over multiple channels, depending on the particular specification for the step being accessed.

This chapter explores the various elements of SHOGUN's implementation in detail, including the SHOGUN patchfile syntax, internal data representation, MIDI communication, and user interface.

5.1.1 Patchfiles

A SHOGUN patchfile, as far as the software itself is concerned, serves to define a set of MIDI commands. Although these commands have an inherent sequentiality, they may also be accessed in arbitrary order. A full context-free grammar specifying patchfile syntax can be found in Appendix A.

From the user's perspective, the task of defining these MIDI commands

with a patchfile can be divided into three main goals. First, and most importantly, the patchfile contains a series of lines defining all patch changes in the production. These are the *steps* mentioned previously, and consist of one or more MIDI bank- and program-change commands. Second, a patchfile allows the user to assign semantic names to various MIDI banks, patches, and bank-patch pairs. (In this report, and in much of the SHOGUN documentation, a bank-patch pair will be referred to as a *voice*.) Finally, a patchfile allows the user a convenient method of routing commands to particular MIDI channels, either as part of a bank, patch, or voice definition, or in the context of a particular step.

Conceptually, we can differentiate three sections in a patchfile: metadata, MIDI definitions, and steps. It is important to note, however, that in general the SHOGUN parser does not require these sections to be separate. A MIDI definition, for example, could occur between two patch-change steps without necessarily resulting in invalid or indeterminate data. Specifics of SHOGUN's parsing process, including the ways in which it handles unclear or incorrect input, are discussed below in Section 5.1.2.

In general, SHOGUN's patchfile syntax has been designed with the intent of being concise without becoming cryptic. The possibility of composing this syntax in some existing format was considered, with the foremost option being eXtensible Markup Language (XML). However, even with the most compact XML schema that was still easily comprehensible, formatting a patchfile as XML resulted in a much longer patchfile than the minimal syntax now used by SHOGUN. Additionally, the fact that XML has a very strict syntax also means that it may be less than forgiving for users unfamiliar with the syntax. SHOGUN's own syntax can hardly claim the pedigree of XML, but the syntax has been designed to be simple and easy to learn. In addition, the patchfile parsing process is designed to ignore invalid input rather than "complaining" about it.

Metadata are defined with *hashtags* of the form `#tagname` followed by a double-quoted string. (In the current version of patchfile syntax, strings are defined as "any string of alphanumeric characters, optionally including underscores or spaces." This definition may be too general to allow for future internationalization, and would therefore need to be revised in versions greater than 1.0.)

SHOGUN defines four metadata fields: title, author, date, and syntax version. The title, author, and date need not obey any particular format (note in particular that any date format may be used), as these fields are provided for human reference rather than machine parsing. The `#version` tag must be

followed by a version number, which is a positive integer optionally followed by a decimal point and a single digit. The use of these tags is demonstrated in Listing 5.1.

Listing 5.1: Shogun's metadata tags (from Listing B.1)

```
4  #title   "My Big Shogun Musical!  "
5  #author  "Ben LaVerriere"
6  #date    "8 December 2010"
7  #version "1.0"
```

While Shogun's metadata fields require double-quoted strings for values, the MIDI definition tags require non-quoted values containing no white space. This serves not only to distinguish the two sets of syntactic features, but also to signify that data supplied in MIDI definition tags are used literally (as integers, for example) by Shogun, not merely as text. The use of these tags is demonstrated in Listing 5.2.

Listing 5.2: Shogun's MIDI definition tags (from Listing B.1)

```
9  #defaultchannel 0
10
11 #bank orch 0
12 #bank synth 1
13
14 #voice cello 2 24
15 #voice piano 3 15
16 #voice flute 5 120 c2
```

Specifically, Shogun allows three types of MIDI definition: bank declarations, voice declarations, and a default channel declaration. Let us first consider the lengthiest of these definitions, the voice declaration. To assign a semantic name to a MIDI bank-patch pair, the user writes `#voice` ⟨name⟩ ⟨bank⟩ ⟨patch⟩. Here ⟨name⟩ is a string with no white space, ⟨bank⟩ may be a previously-defined bank name or a MIDI-standard bank number, ⟨patch⟩ is a non-negative integers as dictated by the MIDI standard, and ⟨channel⟩ is an optional MIDI channel number prefixed with '`c`'.

All of Shogun's MIDI definitions have syntax based on that of the `voice` tag: the tag name is followed by the semantic name being assigned, which in

turn is followed by one or more data values. In addition, the specific order of these data values (bank, patch, channel) is obeyed whenever these values appear in a patchfile — in steps as well as definitions. As a result, we have the `#bank` tag, for which only a name and a bank number must be supplied. Similarly, the `#defaultchannel` tag, which specifies a MIDI channel to which to send steps without an explicitly-specified channel, takes only a single number as its datum.

Listing 5.3: Various ways of writing Shogun patchfile steps (from Listing B.1)

```
18 // just a voice name
19 cello
20
21 // voice name and internally-specified channel
22 flute
23
24 // voice name and channel number
25 piano c3
26
27 // bank name and patch number
28 orch 15
29
30 // bank name, patch number, and channel
31 orch 16 c2
32
33 // bank number and patch number
34 1 52
35
36 // bank number, patch number, and channel
37 2 49 c2
38
39 // multi-channel step
40 2 15 c1, 3 27 c2
```

Step specification, by dint of its role as the essential element of Shogun's functionality, is the most flexible portion of Shogun's patchfile syntax. We can easily derive all possible ways to specify a step by considering the necessary and optional data for each step: each step must contain (in some form) a bank specification and a patch specification, and each step may optionally

also include a channel specification. The bank specification may be expressed as a number, a name, or as part of a named voice. The various permutations of these options are shown in Listing 5.3, including the possibility of addressing multiple channels in one step. As mentioned above, the values for banks, patches, and channels always appear in the same order (as with definitions) and channel numbers are always prefixed with a `c`.

Note that each comma-separated portion of a multi-channel step can take any of the forms acceptable for a single-channel step. That is, in the same patchfile shown in Listing 5.3, a step written as `cello, flute, piano c3` would be perfectly valid.

5.1.2 Internal Representation

In this section, we discuss the most important aspects of SHOGUN's internal data structure. (For a detailed reference for these or any other aspects of SHOGUN's codebase, please refer to Appendix C.) These include the ShogunReader class, which is used to parse patchfiles, and the ShogunShow class, which represents a parsed patchfile. The ShogunMidiController class, which abstracts a subset of RtMidi's functionality for use within SHOGUN, is discussed in the next section.

ShogunReader

The ShogunReader class reads and parses a SHOGUN patchfile, producing a ShogunShow object representing the patchfile's contents.ShogunReader performs a single pass of parsing on a patchfile.[1] For each line in the patchfile which is not empty, ShogunReader determines whether the line is a definition (beginning with '`#`'), a comment (beginning with '`//`'), or a step. (The possibility that a line does not represent valid input is handled by the definition- and step-parsing portions of the routine.) After parsing is complete, the `populate()` method of ShogunShow is called, which ensures that the ShogunShow is in "performance-ready" state and can be returned to the calling class.

The process of parsing a definition is relatively straightforward: determine which type of definition is represented, parse the data appropriately, and store the results in the ShogunShow. The metadata fields are all parsed in the same manner, because their data are double-quoted strings, while `#voice`, `#bank`, and `#defaultchannel` declarations have slightly different parsing rou-

[1] The choice of making ShogunReader a single-pass parser was not without its disadvantages. Voice and bank names must be defined before they are used, although such definitions can occur anywhere in the patchfile — including the line directly before the name's first use.

tines based on their different data formats. If the definition tag beginning the line does not correspond to any known SHOGUN syntax element, the line is discarded without being processed. In general, the parser attempts to "accept" as much of a patchfile as possible, so that users will experience as few disruptions to their performances as possible. (A possible extension to the parser, adding a more strict parsing mode, is discussed in Chapter 7.)

To store a patch-change step, a slightly more complex process is needed. First, because multiple devices (or rather, MIDI channels) may be addressed on a single line, the line under consideration is split at all occurrences of the comma (`,`) so that each patch change may be parsed individually. In this way, a single invalid patch change in a multi-channel step does not invalidate the entire line. For each of these single-channel patch changes, a new ShogunVoice is created containing the appropriate data. The ShogunVoices created in this manner are added to a ShogunStep object, which not only stores the patch changes to be sent but also tracks which MIDI channels are used by each step. If a ShogunVoice does not specify a channel, the `#defaultchannel` is used; if that value has not been defined by the user, SHOGUN uses its internal default of `0`.

ShogunShow

The ShogunShow class represents a fully-parsed SHOGUN patchfile, including metadata, voice and bank definitions, and patch-change steps. Most of this class's methods are concerned with creating and modifying these data in unremarkable ways — getters and setters, primarily — but of particular interest is the `populate()` method. Since SHOGUN's users may wish (or need) to jump to an arbitrary patch-change step during a performance, the application must ensure that the result of *arriving* at a given step is the same regardless of the step which was last accessed. In other words, if a user jumps from step 32 to step 50, every MIDI device connected to SHOGUN should have the same bank and patch loaded as if the user had stepped sequentially through steps 33–49. It would obviously be inefficient to send all such "interstitial" MIDI data when performing a jump-to-step operation. Instead, the `populate()` method preemptively modifies the ShogunShow after it has been filled with the parsed data, "carrying forward" each channel's most recent value to subsequent steps.

For example, consider the following sequence of patch changes:

Channel 0	Channel 1	Channel 2
1 15		
1 16	1 20	
1 17		1 37
1 18		
1 19		1 38

If the user were to jump from the first step to the third step, the device connected to channel 1 would not be sent any patch-change data, violating the principle of "source-step ignorance" described above. Instead, we call `populate()` on this sequence, and the following sequence results (with new values indicated with plus signs):

Channel 0	Channel 1	Channel 2
1 15		
1 16	1 20	
1 17	+ 1 20	1 37
1 18	+ 1 20	+ 1 37
1 19	+ 1 20	1 38

Note that none of the values supplied by the user have been modified, and that jumping from any step to any other step is equivalent to arriving at the second step by moving one step at a time from the beginning of the sequence. Additionally, each entry generated by `populate()` — that is, each entry not originally supplied by the user — has the boolean member `generated` set to true, so that Shogun can identify or recreate the original patchfile, if needed.[2] For the steps in which a channel has not yet been used, `populate()` makes no modifications for the unused channels, since these entries are effectually undefined. At the beginning of the show, the devices on these channels may take some default value, but neither that value nor any value Shogun could supply would be useful or "intentional" in any way.

5.2 MIDI Communication

As discussed in Chapter 2, the RtMidi library provides a useful set of abstractions for MIDI communication. Among the many abilities of this library, Shogun uses RtMidi to obtain a list of available MIDI output devices, connect to one of those devices, and send multi-byte messages to that

[2] This functionality is not currently used by Shogun, since there is currently no need for Shogun to generate patchfiles, but the minimal infrastructure needed to support it remains.

device. The ShogunMidiController class provides a convenient encapsulation of only those features of RtMidi used by SHOGUN, including the ability to send a ShogunVoice or an entire ShogunStep automatically. None of ShogunMidiController's functions is terribly complex, but the simplification of RtMidi's functionalityit provides makes MIDI programming elsewhere in SHOGUN more compact and functionally self-evident.

There are a few design decisions made with respect to MIDI communication. One of the more significant is that SHOGUN only uses a MIDI output device, with no "talkback" on a corresponding input device. Since all of SHOGUN's primary functionality can be accomplished by *sending* MIDI data, it was decided that including the ability to read *incoming* MIDI data added unnecessarily complexity to the system.

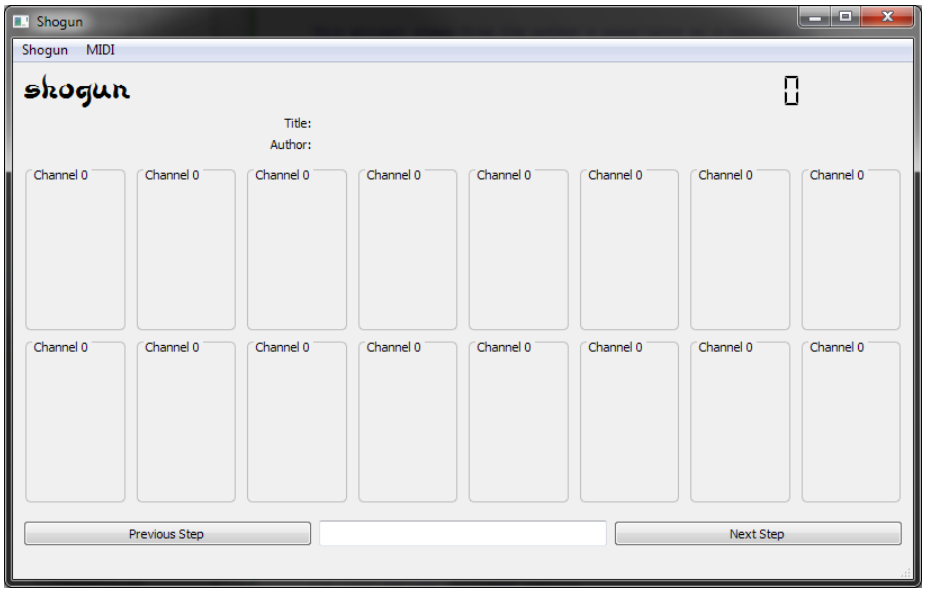
A second MIDI-related design decision has already been discussed: the `#defaultchannel` syntax element, and SHOGUN's corresponding internal default MIDI channel. Though it is far from an official rule, it is common practice that MIDI communication uses channel zero unless otherwise specified, and SHOGUN obeys this convention. The `#defaultchannel` tag is provided so that users need not specify a single, non-zero channel number for each patch-change step, but is optional so that users may also use the default-to-zero convention if they desire.

5.3 GUI

Figure 5.1 shows SHOGUN's main graphical user interface (GUI). In addition to a menu bar, the main SHOGUN window has three primary divisions: the top "information" area, the bottom "navigation" area, and the large "heads-up display" area in the middle. In the information area, the title and author of the currently-loaded patchfile are displayed for reference, along with the current step number. (Technically, this number refers to the last step for which MIDI data was sent.) The navigation area contains the "previous step" and "next step" buttons, which navigate through the patch-change list one step at a time, and the jump-to-step text box, which allows users to access steps in arbitrary order by entering step numbers. Finally, the majority of the screen has been devoted to a display of the current system state, as far as SHOGUN can know it.

SHOGUN's GUI was developed with a number of usability or user-experience goals in mind. In particular, because of the constraints discussed in Chapter 1, SHOGUN's GUI needed to be "minimal" in a variety of ways. Only those elements necessary to SHOGUN's functionality, or to showing the system state,

Figure 5.1: SHOGUN's main interface, before any patchfile has been loaded.

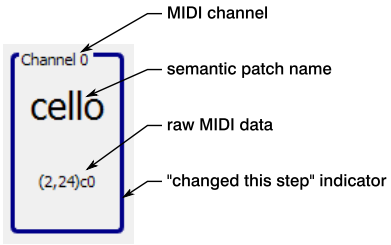


should be granted valuable screen space, and these should be designed to make the most useful or frequently-referenced information the easiest to discern. To that end, I developed a "heads-up display" GUI module, which could be associated with a single MIDI channel and would display the salient elements of that channel's status.[3] Specifically, as shown in Figure 5.2, each heads-up display (HUD) displays the channel with which it is associated, the name of the last voice sent to that channel, and the full bank-patch-channel triad that was sent. (SHOGUN attempts to provide the most semantically-rich name to the user for each HUD, but if no voice or bank name was declared, the GUI will display the last-sent bank and patch numbers.) Each time a new voice is sent to a channel, that channel's HUD will be emphasized with a thick, colored border.

The SHOGUN GUI has also been designed to be "scalable" in a few ways. Every element of the interface has been assigned particular rules for resizing as the overall available space for the GUI changes, with those elements most

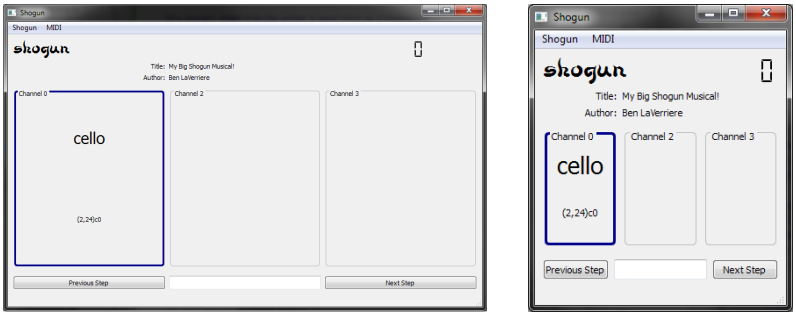
[3]Recall, however, that because MIDI is a unidirectional protocol, and because SHOGUN is not designed to monitor a MIDI input device, any manual changes effected on a device connected to SHOGUN will *not* be reflected in this display.

Figure 5.2: Components of a single MIDI-channel HUD



useful for quick reference taking up the most space. (See Figure 5.3 for a comparison of the interface at various sizes.) In particular, the HUDs are instructed to take up as much space as possible, while the navigation buttons are instructed to expand only in the horizontal direction.[4]

Figure 5.3: Demonstration of the SHOGUN GUI's ability to adapt to a wide range of screen sizes.

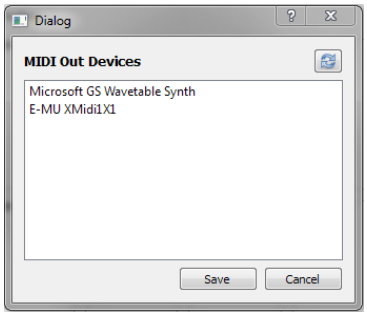


The GUI is also scalable with respect to the layout of the HUDs themselves. Since most patchfiles will not use all sixteen MIDI channels, the process of loading a patchfile triggers a dynamic layout method that adds only as many HUDs as necessary for that particular patchfile. A specific layout for each possible number of HUDs is predefined for SHOGUN, and each HUD is as-

[4] All GUI development for SHOGUN was done with a maximum window size of 800×400 pixels, the smallest commonly-used screen size for "netbook" computers (BariAngel, 2009), and the interface can in fact scale to even smaller screens.

signed one of the channels used by the patchfile and placed onscreen according to the appropriate layout, in ascending numerical order. This behavior is demonstrated in Figure 5.5.

Figure 5.4: SHOGUN's MIDI device selection dialog



A few guidelines for the user's interaction with SHOGUN were also developed. One principle was that SHOGUN should prevent, as much as reasonably practicable, any accidental or erroneous input, and should provide the ability to quickly recover from such input if it occurs. The graphical interface, for example, contains two large buttons, a text field, and a menu bar as its only interactive elements. If the user intends to press the "next step" button, it is difficult to hit the "previous step" button by mistake, because they are on opposite sides of the screen. Similarly, the interface elements likely to be used during a performance — the buttons and text field — are positioned at the bottom of the screen, as far away from the menu bar as could be.

Another principle of SHOGUN's interactivity is that as many actions as possible should be able to be performed with either the mouse or the keyboard, as desired by the user. Thus, to move forward one step, the user can either click the "next step" button, press the right-arrow key, or press the space bar. In fact, with the exception of entering a step number (which can only be executed with some sort of keyboard or number pad), each action provided by SHOGUN's primary GUI can also be executed with the keyboard.

For completeness, we should also briefly mention SHOGUN's MIDI output device selection window, shown in Figure 5.4. This portion of SHOGUN's interface uses the ShogunMidiController to generate a list of available MIDI output devices, which may be refreshed using the button in the upper right of the window. "Business logic" is included that prevents the user from at-

Figure 5.5: Demonstration of the Shogun GUI's dynamic layout system for a variable number of HUDs.

tempting to select more than one device at once, or from selecting no device at all (unless no devices are available). This window appears when SHOGUN is launched, and may be accessed at any time through the MIDI menu item.

Chapter 6

Procedure

In this chapter, we review SHOGUN's development process. Emphasis is given to those parts of the process which will be of most use for future development of the SHOGUN system.

6.1 Technology Choices

As discussed in Chapter 2, the choice of a MIDI programming toolkit was one that required research and careful consideration. The selection of RtMidi as SHOGUN's MIDI toolkit meant that development would proceed in C++, but a number of other technical elements remained to be decided. What tools would be used to construct SHOGUN's GUI, and would any additional libraries be needed? This section addresses these choices, and then Section 6.2 explores how the components fit together in the course of development.

There are three main external programming components that have been part of the SHOGUN development process: the C++ Standard Template Library (STL), selected libraries from the Boost project, and the Qt framework.

The STL is likely the most widely-used C++ library in use today. What began as a library of container types written by Alex Stepanov went on to become "the containers and algorithm framework of the ISO C++ standard library" (Stroustrup, 2005). Because of its prevalence, the STL was a natural choice for SHOGUN's data structure libraries. The *map* and *vector* structures were used particularly frequently.

In addition to classes from the STL, the SHOGUN development process included various libraries from the Boost project. The Boost libraries are designed to provide generic, widely-useful components for C++ programming

that complement both C++ itself and the STL. In particular, a selection of Boost classes related to string manipulation, tokenizing, and so on proved useful when building the ShogunReader class.

The Qt (pronounced "cute") framework became the final third-party component of the SHOGUN development project. Qt is a "cross-platform application and UI framework [including] a cross-platform class library, integrated development tools and a cross-platform IDE" (Nokia Corporation, 2010b). Although Qt was originally selected only as a GUI toolkit, as development progressed, more of the framework was integrated into SHOGUN's codebase.

Another key element of the Qt framework is a tool called `qmake`, which automatically generates Makefiles for Qt projects. This tool is a crucial part of the Qt development process, and its functionality is best described by its own manual:

> `qmake` is a tool that helps simplify the build process for development project across different platforms. `qmake` automates the generation of Makefiles so that only a few lines of information are needed to create each Makefile. `qmake` can be used for any software project, whether it is written in Qt or not.
>
> `qmake` generates a Makefile based on the information in a project file. Project files are created by the developer, and are usually simple, but more sophisticated project files can be created for complex projects. `qmake` contains additional features to support development with Qt, automatically including build rules for moc and uic.[1] `qmake` can also generate projects for Microsoft Visual studio without requiring the developer to change the project file. (Nokia Corporation, 2010a)

6.2 Development Process

In retrospect, we can divide SHOGUN's development into two phases. The first of these was dedicated to the development of the functional, "back-end" components of the system, while the second began with the addition of Qt as GUI toolkit. Unexpectedly, incorporating Qt turned out to be a complex task — more involved than simply referencing a few header files — and resulted in a complete restructuring of the project and its compilation process.

In this section, we review those aspects of the development process that are most relevant to future development of SHOGUN. Since many of the difficulties that arose in this process were related to the compilation of the SHOGUN

[1] `moc` and `uic` are Qt's *Meta-Object Compiler* and *User Interface Compiler*, respectively, which are used to generate pure C++ from files containing various Qt extensions to the language.

application, particular emphasis is given to changes made in the compilation process in the course of development.

6.2.1 Phase 1: Boost and the STL

As mentioned above, development of SHOGUN began with the data structures, back-end algorithms, and other non-visual components. At this point, development was conducted in the Eclipse integrated development environment (IDE), using the C/C++ Development Tooling (CDT). Makefiles were generated automatically by the CDT, and the Minimalist GNU for Windows (MinGW) environment's port of `g++` and related tools were used by Eclipse for automated compilation.

Although the Eclipse CDT makes many aspects of C/C++ development easier, the degree to which certain aspects of project management are abstracted by the IDE occasionally made it difficult to determine the proper way to change the project's configuration. For example, when it first came time to incorporate some of the STL data structures, it was difficult to determine where to place the appropriate header files so that they would be on the build path. Eventually, it was determined that taking every action *through* Eclipse's interface (rather than trying to manually add files to a folder) would be the most efficient method of changing project configuration and would guarantee that compilation would proceed properly.

6.2.2 Phase 2: Qt

When it came time to begin the development of SHOGUN's GUI, it seemed that incorporating Qt would be a relatively painless task. Not only did the Qt project acknowledge that development with Qt was possible with the Eclipse CDT, the project also provided a "Qt Eclipse Integration" plugin that appeared to provide a wide range of features to assist with Qt development in Eclipse. As a result, I assumed I could install the integration plugin, place the Qt libraries in an appropriate location, adjust the configuration settings of the SHOGUN Eclipse project, and continue developing as before.

This was not the case. Qt and the CDT each have their own way of storing project settings, and these turn out to be just about as mutually incompatible as possible. A Qt project is defined by a `*.pro` file, such as the one shown in Listing 6.1. Note that these settings all relate to the source code itself, not the development environment. By contrast, an Eclipse CDT project is defined by a file called `.cproject`, which intermixes compilation unit definitions, Eclipse-specific settings, and a wide range of other data. As a result, not only are

Eclipse CDT projects almost completely inextricable from Eclipse, but it is impossible to "add" Qt to such a project.[2] Instead, one must create a new project, using the Qt integration plugin, that is defined from the start in Qt's style. That project will then have both Qt-style and Eclipse-style configuration files.

This solution appeared to work for a while. The Qt integration plugin kept the `.pro` file up to date, and building the project was successful. Soon, however, it became evident that the Qt integration with Eclipse was less than ideal. The Eclipse CDT seemed somewhat unwilling to "play nicely" with this foreign compilation system, so to speak.

As a result, the development process continued not in Eclipse but in Qt's very own Creator IDE. Qt Creator proved to be a convenient and full-featured development environment that served SHOGUN's development needs very well, not least because of its native support for Qt project files and `qmake`. At the same time, a great deal of the existing SHOGUN code was "ported" to Qt. All instances of `std::strings` were replaced with `QStrings`, which not only simplified interaction with Qt-native methods but also effortlessly made SHOGUN Unicode-compatible. Similarly, all STL and Boost containers were replaced with their Qt counterparts, making it easier for future developers to find documentation for those data structures and taking advantage of Qt's status as a current and rapidly-developed software project.

6.3 Shogun as Cross-Platform Software

Since SHOGUN is intended to be cross-platform software, a few remarks are in order on the topic of cross-platform portability and compilation. Before making those remarks, however, it is important to note that none of these claims have been tested — here we can only say that things *should* work, not that they have been shown to do so.

Some components of the SHOGUN system are inherently cross-platform: C++ is platform-agnostic at its core, and Qt has also been developed to be cross-compilable. RtMidi provides preprocessor definitions (e.g. `__WINDOWS_MM__`, `__LINUX_ALSAEQ__`) that allow it to be compiled for various combinations of operating system (OS) and MIDI application programming interface (API) (Scavone, 2011). (The examples cited here refer to the Windows Multimedia Library and the Linux ALSA Sequencer.)

[2] It is also fairly difficult to find any definitive statement, either in official documentation or elsewhere, that the CDT-to-Qt project conversion is strictly impossible, a fact which led me to spend a great deal of time attempting such an impossible conversion.

Listing 6.1: SHOGUN's Qt project-definition file

```
TEMPLATE = app
TARGET = shogun_qt
QT += core \
    gui
HEADERS += RtError.h \
    RtMidi.h \
    ShogunMidiController.h \
    ShogunBank.h \
    ShogunShow.h \
    ShogunStep.h \
    ShogunVoice.h \
    ShogunReader.h \
        shogun_qt.h \
    midiselectdialog.h \
    keypresslistener.h \
    miditestdialog.h \
    mainpage.h
SOURCES += RtMidi.cpp \
    ShogunMidiController.cpp \
    ShogunBank.cpp \
    ShogunShow.cpp \
    ShogunStep.cpp \
    ShogunVoice.cpp \
    ShogunReader.cpp \
    main.cpp \
        shogun_qt.cpp \
    midiselectdialog.cpp \
    keypresslistener.cpp \
    miditestdialog.cpp
FORMS += shogun_qt.ui \
    midiselectdialog.ui \
    miditestdialog.ui
RESOURCES +=
INCLUDEPATH +=
DEFINES += __WINDOWS_MM__ \
        __RTMIDI_DEBUG__
LIBS += -lwinmm
```

When it comes to compilation, of course, the Windows-specific MinGW tools are not cross-platform compatible. Since these serve as Windows ports of the `g++` compiler and related tools, however, any equivalent C++ compiler, linker, etc. may be used. (Qt's `qmake` also helps standardize the compilation process across different platforms.)

If SHOGUN is eventually compiled and tested on other platforms, there will no doubt be parts of the codebase found to be less platform-agnostic than they were intended to be. Nonetheless, the adjustments necessary to make these portions of the code truly cross-platform should still be fairly minimal, and will promote a more stable SHOGUN overall.

Chapter 7

Future Work

A number of avenues exist for continued development of the SHOGUN system. Some of these tasks involve the completion of functionality originally intended for this release, while others represent possible additions of functionality.

7.1 Application Functionality

7.1.1 Diagnostic Mode

An interface for testing MIDI network communication was originally intended for this release, but time constraints prevented this feature from being completed. A tentative GUI design for this feature exists in the current SHOGUN codebase, and making that interface functional should be a relatively straightforward task.

For each MIDI channel, the diagnostic interface would allow the user to perform three functions:

- to turn a single MIDI note on or off, testing whether MIDI communication in general is possible on the specified channel, and that the appropriate device is responding to MIDI data on that channel
- to send a patch-change command, ensuring that the target device is configured to listen for and respond to patch-change messages[1]
- to send a bank-change command, ensuring that the target device is configured to listen for and respond to bank-change messages, and that the

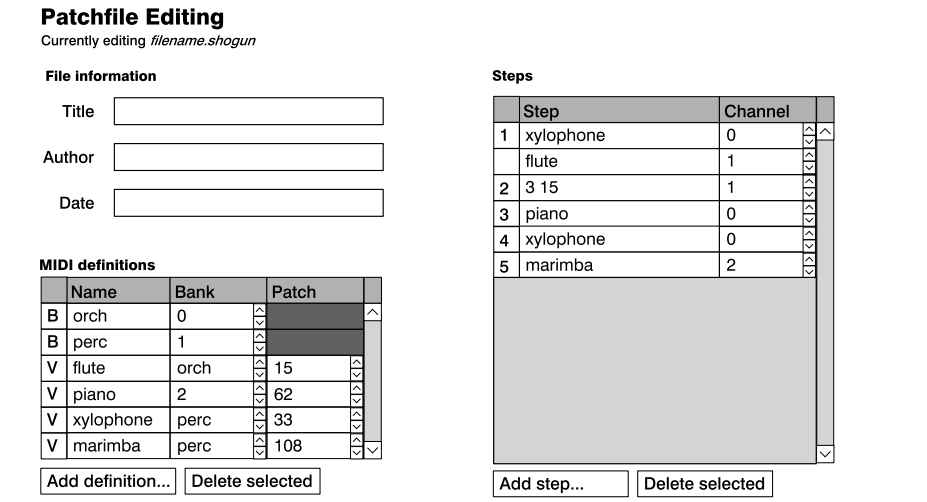
[1] Many MIDI-capable devices allow various filters to be applied to incoming or outgoing MIDI data — for example, a synthesizer set up to respond to externally-generated *note* data may filter out all MIDI control messages to avoid inadvertent modifications to its settings.

particular MIDI message sent by SHOGUN has the desired effect

7.1.2 Patchfile Mode

Because SHOGUN uses a custom syntax for its patchfiles, it may be convenient for users to be able to edit patchfiles within the application. This could be as simple as a text-editing environment with, perhaps, some manner of syntax highlighting. If it was decided that a more structured editing environment would be helpful, users could be presented with an interface like the wire-frame shown in Figure 7.1. Such an interface could make it easier for new users to construct a patchfile, while still allowing the patchfiles themselves to be saved in plain text. (This last point is significant primarily because it is the author's belief that configuration files should always be manually editable.)

Figure 7.1: Possible layout of a structured patchfile-editing interface.



Additionally, the ShogunReader class could be expanded to support two parsing modes: a strict *syntax checking* mode in addition to the current fault-tolerant *performance* mode. In the former mode, errors in syntax would be indicated to the user, who would also be provided with information about the correct syntax for the situation in question. SHOGUN could even an existing spelling-checking library, possibly with a custom dictionary of SHOGUN

"syntax words," to assist users to correct syntax errors that are merely typographical. The performance parsing mode, however, must still exist, so that if a user loads a syntactically-incorrect patchfile, SHOGUN will deal with the errors as gracefully as possible.

7.2 Syntax Extensions

7.2.1 Custom MIDI Commands

In Chapter 2 it was mentioned that the MIDI standard provides a second bank-change controller (#32), which is rarely used and therefore currently unsupported by SHOGUN. Since rare use is nonetheless some use, future development for SHOGUN could include the ability for users to specify custom bank-change commands — or indeed, custom MIDI commands in general — for their devices. The MIDI standard also supports *system-exclusive* (SysEx) messages, which allow specific devices or manufacturers to send and receive arbitrary data over the MIDI line. It is not inconceivable that users may wish to include SysEx messages in a SHOGUN patchfile, particularly if they are working with older equipment that may require unusual control messages to change voices.

The degree to which the sending of arbitrary MIDI data is supported should be carefully considered, however, lest the SHOGUN interface or patchfile syntax become prohibitively complex. For example, should users be allowed to define a single alternate bank-change command per MIDI channel, or should syntax exist for the definition and use of an arbitrary number of MIDI commands at any point in the patchfile? Intuition suggests that any extensions to SHOGUN's patchfile syntax be as small as possible but, to paraphrase that common misquotation of Einstein, no smaller.

7.2.2 Bookmarks

Early in the development of SHOGUN's patchfile syntax, the idea of patchfile *bookmarks* was considered. These would be syntactic elements that marked particular locations in the sequence of patch-change steps, and to which users could navigate quickly through SHOGUN's GUI. (The hypothetical syntax would have been something like `@⟨bookmark name⟩`.) This syntax was discarded, however, for two reasons. First, the SHOGUN application already allows users to jump to an arbitrary step by entering its sequence number, making "bookmarking" possible simply by noting a few numbers on a scrap of paper. Second, even if the patchfile syntax were to be modified to include

bookmarks, it was not evident how the SHOGUN GUI could accommodate a method of accessing an arbitrary number of these bookmarks. An array of buttons could become prohibitively crowded if each button stretched to fit the available area, or else the user would need to scroll through a list of bookmarks if each entry was assigned some minimum amount of space regardless of screen size. If a method of interaction for accessing bookmarks could be developed that was both unobtrusive and easy to enact under SHOGUN's environmental constraints as described in Chapter 1, then perhaps the syntax could be extended to include bookmarks.

Appendix A

Shogun Patchfile Context-Free Grammar

This document represents a formal specification of the syntax defining a valid SHOGUN patchfile. Patchfiles obeying this syntax should be tagged with `#version 1.0` (or `#version 1`, if preferred). The ShogunReader class is able to parse any patchfile obeying this syntax correctly. ShogunReader makes an attempt to deal as gracefully as possible with syntactically-invalid patchfiles by ignoring invalid elements and parsing correct ones, but is not guaranteed to deliver correct results for *any* part of a patchfile containing invalid syntax.

A.1 Notes

1. Text set in `monospaced type` represent literal characters to be included in the patchfile. (All quotation marks in the grammar are set in this manner.) *Italicized text* represents a textual description of a data member that is clearer to express in this manner than with a formal grammar.
2. Elements with the "OPT" subscript are optional. Rules containing optional elements could be expanded into multiple rules to produce a more formal grammar, but are here condensed for clarity.
3. Unless otherwise specified, any amount of whitespace may separate defined elements without a change in meaning. At least one whitespace character must separate elements. Elements in ⟪double brackets⟫ must appear on separate lines.

4. The ⟪MetaDate⟫ element does not enforce any sort of date format — this field is intended for human reference, not machine parsing.

5. The elements ⟨VoiceName⟩ and ⟨BankName⟩ are used to promote syntactic clarity within this document. All ⟨Name⟩ elements share a single namespace and follow the same rules of construction.

6. Because different MIDI devices implement the bank-change method inconsistently, it is the responsibility of the user to ensure that patchfiles contain valid bank numbers. The SHOGUN application may provide some assistance to this end in its troubleshooting mode.

7. The current definition of a ⟨String⟩ may present problems for users who employ non-Latin character sets. This shortcoming may be resolved in a future version of the standard.

8. Because they are immediately discarded in the parsing process, both blank lines and comments have been eliminated from this grammar. Comments must begin with `//`; there are no block comments.

9. There is currently no restriction in this grammar to prevent the construction of a PatchChangeStep of the form `2 14, 5 120, 2 36`, in which each individual PatchChange is addressed to the same channel (or to no channel at all). This action should trigger a warning when a patchfile is being validated, but is not explicitly prevented because such an action will not produce undesirable operation. Each patch change will be executed, in the order specified, as a single step; the end result will simply be that the target device has loaded the final specified voice.

A.2 Shogun Patchfile Syntax, version 1.0

⟨**Patchfile**⟩ → ⟨Metadata⟩$_{\text{OPT}}$ ⟨Declarations⟩$_{\text{OPT}}$ ⟨PatchChangeSteps⟩$_{\text{OPT}}$
⟨Metadata⟩ → ⟪MetaTitle⟫$_{\text{OPT}}$ ⟪MetaAuthor⟫$_{\text{OPT}}$ ⟪MetaDate⟫$_{\text{OPT}}$ ⟪MetaVersion⟫
⟨Declarations⟩ → ⟪Declaration⟫ | ⟨Declarations⟩ ⟪Declaration⟫
⟨PatchChangeSteps⟩ → ⟪PatchChangeStep⟫ | ⟨PatchChangeSteps⟩ ⟪PatchChangeStep⟫

⟪MetaTitle⟫ → `#title "`⟨String⟩`"`
⟪MetaAuthor⟫ → `#author "`⟨String⟩`"`
⟪MetaDate⟫ → `#date "`⟨String⟩`"`
⟪MetaVersion⟫ → `#version "`⟨VersionNumber⟩`"`

⟪Declaration⟫ → ⟪VoiceDeclaration⟫ | ⟪BankDeclaration⟫ | ⟪DefaultChannelDeclaration⟫

⟪PatchChangeStep⟫ → ⟨PatchChange⟩ | ⟨PatchChange⟩ `,` ⟨PatchChangeStep⟩
⟨PatchChange⟩ → ⟪VoiceName⟫ | ⟨VoiceName⟩ ⟨Channel⟩ | ⟨BankName⟩ ⟨PatchNumber⟩ | ⟨BankName⟩ ⟨PatchNumber⟩ ⟨Channel⟩ | ⟨BankNumber⟩ ⟨PatchNumber⟩ | ⟨BankNumber⟩ ⟨PatchNumber⟩ ⟨Channel⟩

⟨String⟩ → *any string of alphanumeric characters, optionally including underscores or spaces*
⟨VersionNumber⟩ → ⟨Integer⟩ `.` ⟨Digit⟩ | ⟨Integer⟩
⟨Integer⟩ → *any string of characters that represents a nonnegative integer*
⟨Digit⟩ → *a single numeric digit*

⟪VoiceDeclaration⟫ → `#voice` ⟨VoiceName⟩ ⟨BankNumber⟩ ⟨PatchNumber⟩ ⟨Channel⟩$_{\texttt{opt}}$ | `#voice` ⟨VoiceName⟩ ⟨BankName⟩ ⟨PatchNumber⟩ ⟨Channel⟩$_{\texttt{opt}}$
⟪BankDeclaration⟫ → `#bank` ⟨BankName⟩ ⟨BankNumber⟩
⟪DefaultChannelDeclaration⟫ → `#defaultchannel` ⟨ChannelNumber⟩

⟨VoiceName⟩ → ⟨Name⟩
⟨BankName⟩ → ⟨Name⟩
⟨Name⟩ → *a string of non-whitespace characters without leading digits*
⟨PatchNumber⟩ → *a number in the range* $[0, 127]$ *as per the MIDI standard*
⟨BankNumber⟩ → ⟨Integer⟩
⟨Channel⟩ → `c`⟨ChannelNumber⟩
⟨ChannelNumber⟩ → *a number in the range* $[0, 15]$ *as per the MIDI standard*

Appendix B

Sample Patchfiles

The patchfiles in this section represent valid SHOGUN patchfiles (according to the grammar presented in Appendix A) that demonstrate a variety of features of the syntax. Listing B.1 is a simple test case, exhibiting most syntactic elements and variants, including an explicit listing of all possible types of patch-change steps. Listing B.2 is designed for testing SHOGUN's dynamic HUD-layout system, by requiring thirteen channels in an otherwise-minimal patchfile. Each possible HUD layout can be demonstrated by adding or removing lines to this patchfile and loading it into SHOGUN.

Listing B.1: test.shogun

```
1  // test.shogun
2  // simple test case for shogun
3
4  #title   "My Big Shogun Musical!  "
5  #author  "Ben LaVerriere"
6  #date    "8 December 2010"
7  #version "1.0"
8
9  #defaultchannel 0
10
11 #bank orch 0
12 #bank synth 1
13
14 #voice cello 2 24
15 #voice piano 3 15
16 #voice flute 5 120 c2
```

```
17
18 // just a voice name
19 cello
20
21 // voice name and internally-specified channel
22 flute
23
24 // voice name and channel number
25 piano c3
26
27 // bank name and patch number
28 orch 15
29
30 // bank name, patch number, and channel
31 orch 16 c2
32
33 // bank number and patch number
34 1 52
35
36 // bank number, patch number, and channel
37 2 49 c2
38
39 // multi-channel step
40 2 15 c1, 3 27 c2
41
42 // "false positive" multi-channel step
43 3 58 c2,
44
45 // end test.shogun
```

Listing B.2: channels-13.shogun

```
1 // channels-13.shogun
2 // a very silly patchfile for demonstrating the dynamic layout
3 // of a large number of channel-HUDs
4
5 #title   "My Big Shogun Musical!  "
6 #author  "Ben LaVerriere"
7 #date    "8 December 2010"
8 #version "1.0"
```

```
9
10 #defaultchannel 0
11
12 #voice sine 2 24
13
14 sine c0
15 sine c1
16 sine c2
17 sine c3
18 sine c4
19 sine c5
20 sine c6
21 sine c7
22 sine c8
23 sine c9
24 sine c10
25 sine c11
26 sine c12
```

Bibliography

BariAngel. 2009. Monitor dei netbook: dimensioni a confronto. http://www.eeepc.it/monitor-dei-netbook-dimensioni-a-confronto/.

Best, F. 2011. MIDI library. http://www.arduino.cc/playground/Main/MIDILibrary.

Capocasa, C. 2006. Portable C++ MIDI libraries review. http://lists.linuxaudio.org/pipermail/linux-audio-dev/2006-February/014829.html.

Dannenberg, R. 93. CMU MIDI toolkit. http://www.cs.cmu.edu/~music/cmt/.

enoki. 2009. portmidizero. http://gitorious.org/portmidizero/.

Gallant, M. 2008. David Rosenthal — movin' out, movin' up. *Keyboard Magazine*.

Genovation 2010. *Genovation MIDI Patch Changer User Guide* Preliminary v0.79 Ed. Genovation.

Harrison, J. 2010. pyPortMidi. http://alumni.media.mit.edu/~harrison/pyportmidi.html.

Klose, T. 2011. Non-commercial DIY projects for MIDI hardware geeks. http://www.ucapps.de/.

Koftinoff, J. D. 2011. jdksmimdi. https://github.com/jdkoftinoff/jdksmidi.

Lokovic, T. 2009. midi-dot-net. http://code.google.com/p/midi-dot-net/.

Max M. 2005. Python midi package. http://www.mxm.dk/products/public/pythonmidi.

Nokia Corporation 2010a. *qmake Manual*. Nokia Corporation.

Nokia Corporation. 2010b. Qt — cross-platform application and UI framework. http://qt.nokia.com/.

Oracle. 2010. javax.sound.midi. http://download.oracle.com/javase/1.4.2/docs/api/javax/sound/midi/package-summary.html.

Palem, G. 2010. MusicNoteLib, the C++ music programming library. http://gpalem.web.officelive.com/CFugue.html.

Peck, D., SantiBanks, and fraz. 2007. Keyboard patch management live. http://www.gearslutz.com/board/so-much-gear-so-little-time/140452-keyboard-patch-management-live.html.

PortAudio Developers. 2009. PortMidi portable real-time MIDI library. http://portmedia.svn.sourceforge.net/viewvc/portmedia/portmidi/trunk/pm_common/portmidi.h?view=markup. PortMidi is part of the PortAudio project <http://www.portaudio.com/>.

PyGame Developers. 2010. pygame.midi. http://www.pygame.org/docs/ref/midi.html.

Roland Corporation 2007. *FC-300 MIDI Foot Controller Owner's Manual*. Roland Corporation.

Rumsey, F. 1990. *MIDI Systems and Control*. Butterworth & Co. (Publishers) Ltd, Newton, MA, USA.

Sanford, L. 2007. C# MIDI toolkit. http://www.codeproject.com/KB/audio-video/MIDIToolkit.aspx.

Scavone, G. P. 2011. The RtMidi tutorial. http://www.music.mcgill.ca/~gary/rtmidi/.

Stinson, P. 2010. pyrtmidi. http://trac2.assembla.com/pkaudio/wiki/pyrtmidi.

Stroustrup, B. 2005. *The Design and Evolution of C++*. Addison-Wesley, Boston, MA, USA, 1–32. Extended foreword "C++ in 2005" from Japanese translation. <http://www2.research.att.com/~bs/DnE2005.pdf>.

Tiedemann, E. S. 2000. PyMidi. http://hyperreal.org/~est/python/MIDI/.

Appendix C

Codebase Documentation

The documentation on the following pages has been generated automatically from the SHOGUN codebase (and its comments) with the Doxygen utility.

Shogun

Generated by Doxygen 1.7.2

Thu Apr 28 2011 03:46:15

Contents

Chapter 1

The Shogun MIDI Control System

Author

Ben LaVerriere

This project stems from the author's experience as a keyboardist playing for musical theatre productions. Keyboardists in musical theatre often make use of a variety of synthesized sounds, or "patches", over the course of a performance or even a single song. Since the development of the modern synthesizer, there have been various ways to access these patches, whether it be a series of buttons (appropriate for a small number of options) or a number-pad interface (for systems with a wide variety of patches). For the musical theatre keyboardist, however, patch changes need to be executed quickly and without error; relying on the musician's ability to push the right button or type the right number is risky in this context.

Shogun addresses these concerns by providing a simple, compact interface that allows a keyboardist (or any other person using MIDI devices) to send a series of patch-change commands easily and in any order. Users may compose their own sequences of patch changes, and navigate these lists sequentially or in random order, as desired.

For more information about the Shogun project, please contact the WPI Gordon Library (`http://wpi.edu/+library`) for a copy of the report detailing Shogun's development.

1.1 Compiling Shogun

Compiling Shogun is most easily accomplished in the Qt Creator IDE. After ensuring that a C++ compiler/linker and the Qt framework are installed on your system, open the `shogun_qt.pro` project file in Qt Creator. If Qt Creator can find (or has been told where to find) your compiler and the Qt binaries (particularly `qmake`), you should be

able to build Shogun straight away.

1.2 License

Chapter 2

Class Documentation

2.1 KeyPressListener Class Reference

Protected Member Functions

- bool eventFilter (QObject ∗obj, QEvent ∗event)

2.1.1 Detailed Description

Definition at line 9 of file keypresslistener.h.

2.1.2 Member Function Documentation

2.1.2.1 bool KeyPressListener::eventFilter (QObject ∗ *obj,* QEvent ∗ *event*) [protected]

Defines a custom event listener (specifically, the eventFilter() method) to be assigned to all elements of the main Shogun interface. This listener ensures that left- and right-arrow keypresses are received and the corresponding patch-change events are sent.

Definition at line 11 of file keypresslistener.cpp.

The documentation for this class was generated from the following files:

- keypresslistener.h
- keypresslistener.cpp

2.2 RtMidiIn::MidiMessage Struct Reference

Public Attributes

- std::vector< unsigned char > **bytes**

- double **timeStamp**

2.2.1 Detailed Description

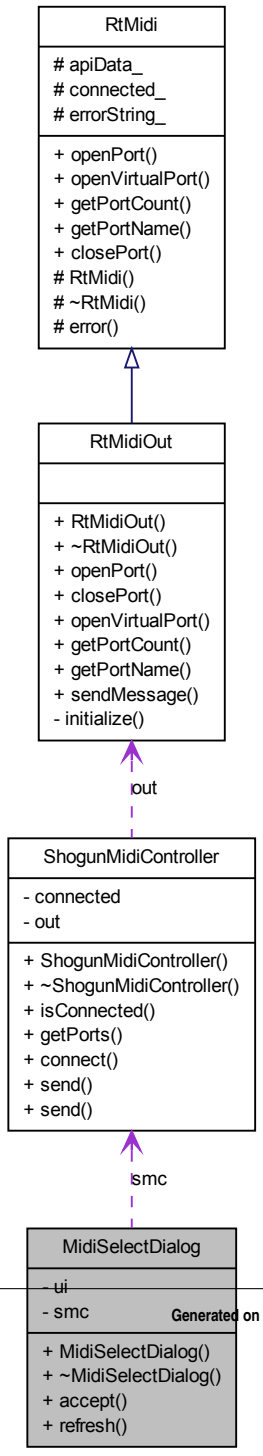
Definition at line 194 of file RtMidi.h.

The documentation for this struct was generated from the following file:

- RtMidi.h

2.3 MidiSelectDialog Class Reference

Collaboration diagram for MidiSelectDialog:



Public Slots

- void **accept** ()
- void **refresh** ()

Signals

- void **MidiDeviceChanged** (int device)

Public Member Functions

- **MidiSelectDialog** (QWidget ∗parent=0)

Private Attributes

- Ui::MidiSelectDialog ∗ **ui**
- ShogunMidiController ∗ **smc**

2.3.1 Detailed Description

Definition at line 12 of file midiselectdialog.h.

The documentation for this class was generated from the following files:

- midiselectdialog.h
- midiselectdialog.cpp

2.4 MidiTestDialog Class Reference

`#include <miditestdialog.h>`

Public Member Functions

- **MidiTestDialog** (QWidget ∗parent=0)

Private Attributes

- Ui::MidiTestDialog ∗ **ui**

2.4.1 Detailed Description

Intended to provide a simple MIDI-network-testing interface.

Warning

This class and its user interface are currently non-functional. MIDI communication does not occur, and there are some UI elements missing.

Definition at line 16 of file miditestdialog.h.

The documentation for this class was generated from the following files:

- miditestdialog.h
- miditestdialog.cpp

2.5 RtError Class Reference

Exception handling class for RtAudio & RtMidi.

`#include <RtError.h>`

Public Types

- enum Type {
  WARNING, DEBUG_WARNING, UNSPECIFIED, NO_DEVICES_FOUND,
  INVALID_DEVICE, INVALID_STREAM, MEMORY_ERROR, INVALID_PARAMETER,
  DRIVER_ERROR, SYSTEM_ERROR, THREAD_ERROR }

  Defined RtError types.

Public Member Functions

- RtError (const std::string &message, Type type=RtError::UNSPECIFIED)

  The constructor.

- virtual ~RtError (void)

  The destructor.

- virtual void printMessage (void)

  Prints thrown error message to stderr.

- virtual const Type & getType (void)

  Returns the thrown error message type.

- virtual const std::string & getMessage (void)

  Returns the thrown error message string.

- virtual const char * getMessageString (void)

  Returns the thrown error message as a C string.

Protected Attributes

- std::string **message_**
- Type **type_**

2.5.1 Detailed Description

Exception handling class for RtAudio & RtMidi. The RtError class is quite simple but it does allow errors to be "caught" by RtError::Type. See the RtAudio and RtMidi documentation to know which methods can throw an RtError.

Definition at line 18 of file RtError.h.

2.5.2 Member Enumeration Documentation

2.5.2.1 enum RtError::Type

Defined RtError types.

Enumerator:

WARNING A non-critical error.

DEBUG_WARNING A non-critical error which might be useful for debugging.

UNSPECIFIED The default, unspecified error type.

NO_DEVICES_FOUND No devices found on system.

INVALID_DEVICE An invalid device ID was specified.

INVALID_STREAM An invalid stream ID was specified.

MEMORY_ERROR An error occured during memory allocation.

INVALID_PARAMETER An invalid parameter was specified to a function.

DRIVER_ERROR A system driver error occured.

SYSTEM_ERROR A system error occured.

THREAD_ERROR A thread error occured.

Definition at line 22 of file RtError.h.

The documentation for this class was generated from the following file:

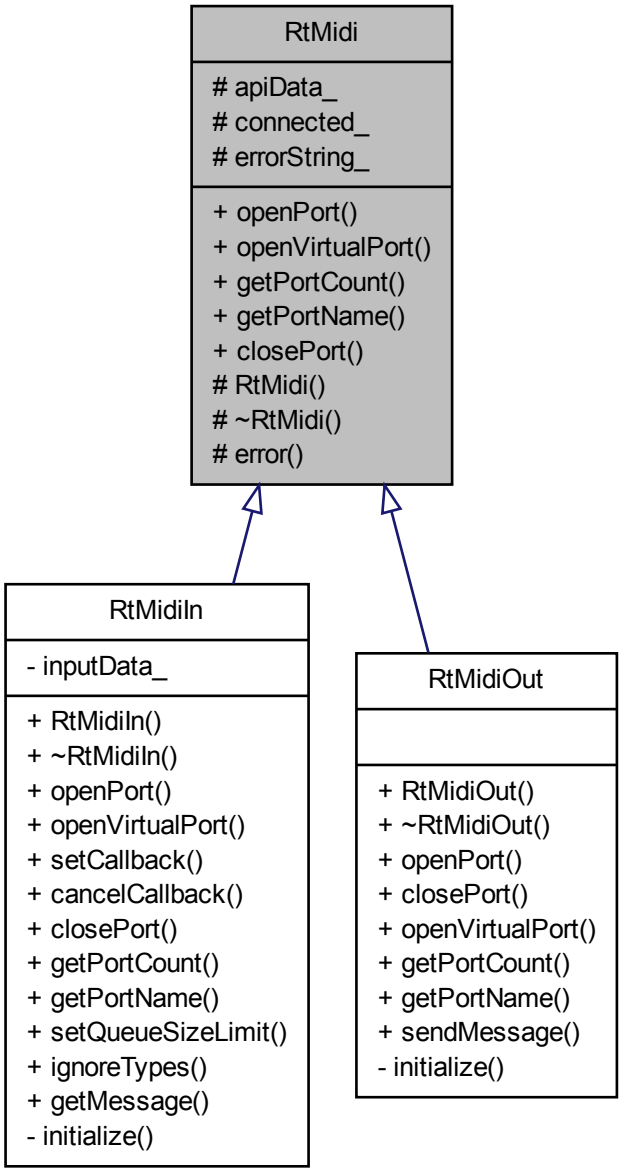
- RtError.h

2.6 RtMidi Class Reference

An abstract base class for realtime MIDI input/output.

`#include <RtMidi.h>`

Inheritance diagram for RtMidi:



Public Member Functions

- virtual void openPort (unsigned int portNumber=0, const std::string portName=std::string("RtMidi"))=0

  Pure virtual openPort() function.

- virtual void openVirtualPort (const std::string portName=std::string("RtMidi"))=0

  Pure virtual openVirtualPort() function.

- virtual unsigned int getPortCount ()=0

  Pure virtual getPortCount() function.

- virtual std::string getPortName (unsigned int portNumber=0)=0

  Pure virtual getPortName() function.

- virtual void closePort (void)=0

  Pure virtual closePort() function.

Protected Member Functions

- void **error** (RtError::Type type)

Protected Attributes

- void * **apiData_**
- bool **connected_**
- std::string **errorString_**

2.6.1 Detailed Description

An abstract base class for realtime MIDI input/output. This class implements some common functionality for the realtime MIDI input/output subclasses RtMidiIn and RtMidiOut.

RtMidi WWW site: http://music.mcgill.ca/~gary/rtmidi/

RtMidi: realtime MIDI i/o C++ classes Copyright (c) 2003-2010 Gary P. Scavone

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

Any person wishing to distribute modifications to the Software is requested to send the modifications to the original developer so that they can be incorporated into the canonical version.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Definition at line 46 of file RtMidi.h.

The documentation for this class was generated from the following files:

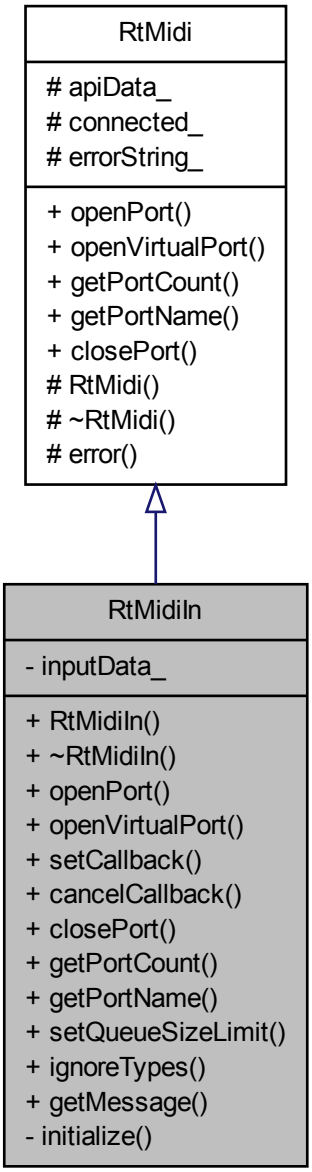
- RtMidi.h
- RtMidi.cpp

2.7 RtMidiIn Class Reference

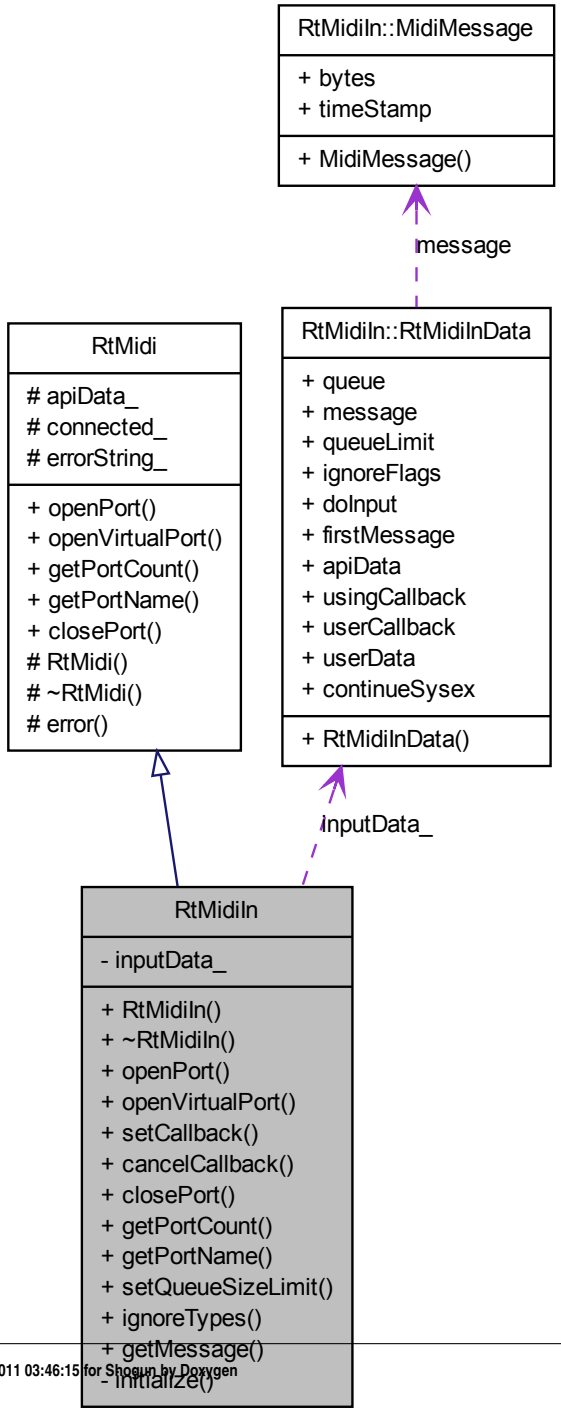
A realtime MIDI input class.

`#include <RtMidi.h>`

Inheritance diagram for RtMidiIn:



Collaboration diagram for RtMidiIn:



Classes

- struct MidiMessage
- struct RtMidiInData

Public Types

- typedef void(∗ RtMidiCallback)(double timeStamp, std::vector< unsigned char > ∗message, void ∗userData)

  User callback function type definition.

Public Member Functions

- RtMidiIn (const std::string clientName=std::string("RtMidi Input Client"))

  Default constructor that allows an optional client name.

- ∼RtMidiIn ()

  If a MIDI connection is still open, it will be closed by the destructor.

- void openPort (unsigned int portNumber=0, const std::string Portname=std::string("RtMidi Input"))

  Open a MIDI input connection.

- void openVirtualPort (const std::string portName=std::string("RtMidi Input"))

  Create a virtual input port, with optional name, to allow software connections (OS X and ALSA only).

- void setCallback (RtMidiCallback callback, void ∗userData=0)

  Set a callback function to be invoked for incoming MIDI messages.

- void cancelCallback ()

  Cancel use of the current callback function (if one exists).

- void closePort (void)

  Close an open MIDI connection (if one exists).

- unsigned int getPortCount ()

  Return the number of available MIDI input ports.

- std::string getPortName (unsigned int portNumber=0)

  Return a string identifier for the specified MIDI input port number.

- void setQueueSizeLimit (unsigned int queueSize)

  Set the maximum number of MIDI messages to be saved in the queue.

- void ignoreTypes (bool midiSysex=true, bool midiTime=true, bool midiSense=true)

  Specify whether certain MIDI message types should be queued or ignored during input.

- double getMessage (std::vector< unsigned char > *message)

  Fill the user-provided vector with the data bytes for the next available MIDI message in the input queue and return the event delta-time in seconds.

Private Member Functions

- void **initialize** (const std::string &clientName)

Private Attributes

- RtMidiInData **inputData_**

2.7.1 Detailed Description

A realtime MIDI input class. This class provides a common, platform-independent API for realtime MIDI input. It allows access to a single MIDI input port. Incoming MIDI messages are either saved to a queue for retrieval using the getMessage() function or immediately passed to a user-specified callback function. Create multiple instances of this class to connect to more than one MIDI device at the same time. With the OS-X and Linux ALSA MIDI APIs, it is also possible to open a virtual input port to which other MIDI software clients can connect.

by Gary P. Scavone, 2003-2008.

Definition at line 102 of file RtMidi.h.

2.7.2 Constructor & Destructor Documentation

2.7.2.1 RtMidiIn::RtMidiIn (const std::string *clientName* = `std::string("RtMidi Input Client")`)

Default constructor that allows an optional client name.

An exception will be thrown if a MIDI system initialization error occurs.

Definition at line 72 of file RtMidi.cpp.

2.7.3 Member Function Documentation

2.7.3.1 void RtMidiIn::cancelCallback ( )

Cancel use of the current callback function (if one exists).

Subsequent incoming MIDI messages will be written to the queue and can be retrieved with the *getMessage* function.

Definition at line 96 of file RtMidi.cpp.

2.7.3.2 double RtMidiIn::getMessage (std::vector< unsigned char > * *message*)

Fill the user-provided vector with the data bytes for the next available MIDI message in the input queue and return the event delta-time in seconds.

This function returns immediately whether a new message is available or not. A valid message is indicated by a non-zero vector size. An exception is thrown if an error occurs during message retrieval or an input connection was not previously established.

Definition at line 122 of file RtMidi.cpp.

2.7.3.3 std::string RtMidiIn::getPortName (unsigned int *portNumber* = `0`) [virtual]

Return a string identifier for the specified MIDI input port number.

An exception is thrown if an invalid port specifier is provided.

Implements RtMidi.

2.7.3.4 void RtMidiIn::ignoreTypes (bool *midiSysex* = `true`, bool *midiTime* = `true`, bool *midiSense* = `true`)

Specify whether certain MIDI message types should be queued or ignored during input.

By default, MIDI timing and active sensing messages are ignored during message input because of their relative high data rates. MIDI sysex messages are ignored by default as well. Variable values of "true" imply that the respective message type will be ignored.

Definition at line 114 of file RtMidi.cpp.

2.7.3.5 void RtMidiIn::openPort (unsigned int *portNumber* = 0, const std::string *Portname* = std::string("RtMidi Input")) [virtual]

Open a MIDI input connection.

An optional port number greater than 0 can be specified. Otherwise, the default or first port found is opened.

Implements RtMidi.

2.7.3.6 void RtMidiIn::openVirtualPort (const std::string *portName* = std::string("RtMidi Input")) [virtual]

Create a virtual input port, with optional name, to allow software connections (OS X and ALSA only).

This function creates a virtual MIDI input port to which other software applications can connect. This type of functionality is currently only supported by the Macintosh OS-X and Linux ALSA APIs (the function does nothing for the other APIs).

Implements RtMidi.

2.7.3.7 void RtMidiIn::setCallback (RtMidiCallback *callback*, void * *userData* = 0)

Set a callback function to be invoked for incoming MIDI messages.

The callback function will be called whenever an incoming MIDI message is received. While not absolutely necessary, it is best to set the callback function before opening a MIDI port to avoid leaving some messages in the queue.

Definition at line 77 of file RtMidi.cpp.

2.7.3.8 void RtMidiIn::setQueueSizeLimit (unsigned int *queueSize*)

Set the maximum number of MIDI messages to be saved in the queue.

If the queue size limit is reached, incoming messages will be ignored. The default limit is 1024.

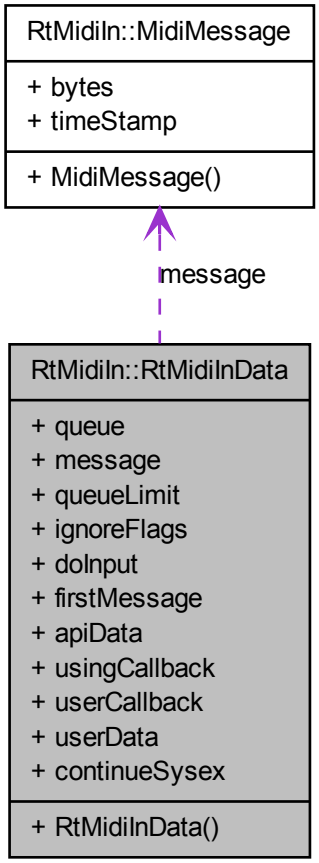
Definition at line 109 of file RtMidi.cpp.

The documentation for this class was generated from the following files:

- RtMidi.h
- RtMidi.cpp

2.8 RtMidiIn::RtMidiInData Struct Reference

Collaboration diagram for RtMidiIn::RtMidiInData:



Public Attributes

- std::queue< MidiMessage > **queue**
- MidiMessage **message**
- unsigned int **queueLimit**

- unsigned char **ignoreFlags**
- bool **doInput**
- bool **firstMessage**
- void * **apiData**
- bool **usingCallback**
- void * **userCallback**
- void * **userData**
- bool **continueSysex**

2.8.1 Detailed Description

Definition at line 206 of file RtMidi.h.

The documentation for this struct was generated from the following file:

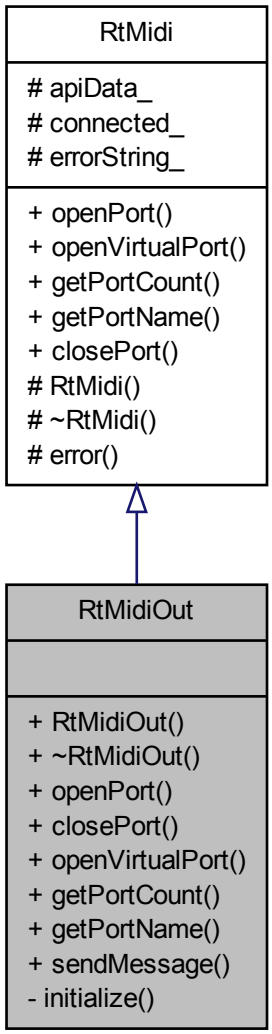
- RtMidi.h

2.9 RtMidiOut Class Reference

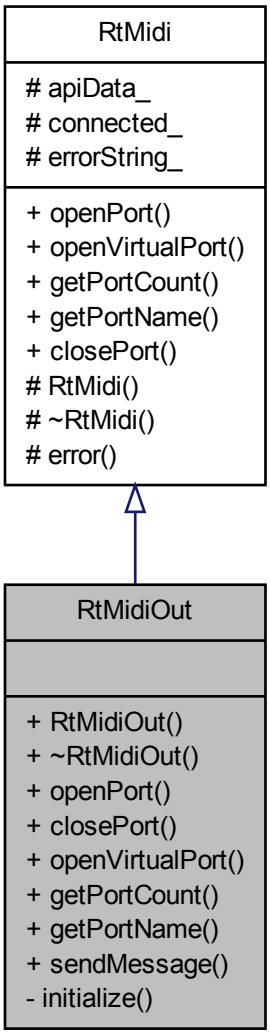
A realtime MIDI output class.

`#include <RtMidi.h>`

Inheritance diagram for RtMidiOut:



Collaboration diagram for RtMidiOut:



Public Member Functions

- RtMidiOut (const std::string clientName=std::string("RtMidi Output Client"))

  Default constructor that allows an optional client name.

- ~RtMidiOut ()

  The destructor closes any open MIDI connections.

- void openPort (unsigned int portNumber=0, const std::string portName=std::string("RtMidi Output"))

  Open a MIDI output connection.

- void closePort ()

  Close an open MIDI connection (if one exists).

- void openVirtualPort (const std::string portName=std::string("RtMidi Output"))

  Create a virtual output port, with optional name, to allow software connections (OS X and ALSA only).

- unsigned int getPortCount ()

  Return the number of available MIDI output ports.

- std::string getPortName (unsigned int portNumber=0)

  Return a string identifier for the specified MIDI port type and number.

- void sendMessage (std::vector< unsigned char > *message)

  Immediately send a single message out an open MIDI output port.

Private Member Functions

- void **initialize** (const std::string &clientName)

2.9.1 Detailed Description

A realtime MIDI output class. This class provides a common, platform-independent API for MIDI output. It allows one to probe available MIDI output ports, to connect to one such port, and to send MIDI bytes immediately over the connection. Create multiple instances of this class to connect to more than one MIDI device at the same time.

by Gary P. Scavone, 2003-2008.

Definition at line 248 of file RtMidi.h.

2.9.2 Constructor & Destructor Documentation

2.9.2.1 RtMidiOut::RtMidiOut (const std::string *clientName* = std::string("RtMidi Output Client"))

Default constructor that allows an optional client name.

An exception will be thrown if a MIDI system initialization error occurs.

Definition at line 147 of file RtMidi.cpp.

2.9.3 Member Function Documentation

2.9.3.1 std::string RtMidiOut::getPortName (unsigned int *portNumber* = 0) [virtual]

Return a string identifier for the specified MIDI port type and number.

An exception is thrown if an invalid port specifier is provided.

Implements RtMidi.

2.9.3.2 void RtMidiOut::openPort (unsigned int *portNumber* = 0, const std::string *portName* = std::string("RtMidi Output")) [virtual]

Open a MIDI output connection.

An optional port number greater than 0 can be specified. Otherwise, the default or first port found is opened. An exception is thrown if an error occurs while attempting to make the port connection.

Implements RtMidi.

2.9.3.3 void RtMidiOut::openVirtualPort (const std::string *portName* = std::string("RtMidi Output")) [virtual]

Create a virtual output port, with optional name, to allow software connections (OS X and ALSA only).

This function creates a virtual MIDI output port to which other software applications can connect. This type of functionality is currently only supported by the Macintosh OS-X and Linux ALSA APIs (the function does nothing with the other APIs). An exception is thrown if an error occurs while attempting to create the virtual port.

Implements RtMidi.

2.9.3.4 void RtMidiOut::sendMessage (std::vector< unsigned char > * *message*)

Immediately send a single message out an open MIDI output port.

An exception is thrown if an error occurs during output or an output connection was not previously established.

The documentation for this class was generated from the following files:

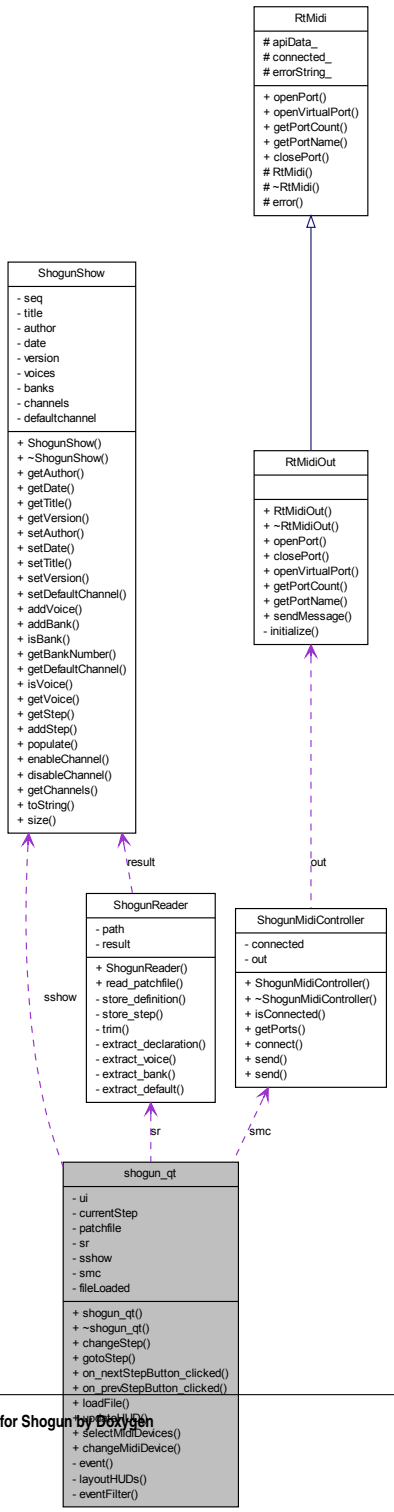
- RtMidi.h

- RtMidi.cpp

2.10 shogun_qt Class Reference

```
#include <shogun_qt.h>
```

Collaboration diagram for shogun_qt:



Public Slots

- void on_nextStepButton_clicked ()
- void on_prevStepButton_clicked ()
- void loadFile ()
- void **updateHUD** (int step)
- void **selectMidiDevices** ()
- void **changeMidiDevice** (int device)

Signals

- void **stepChanged** (int newStep)
- void **stepChanged** (QString newStep)
- void **fileChosen** ()

Public Member Functions

- shogun_qt (QWidget *parent=0)
- void changeStep (int increment)
- void gotoStep (int step)

Private Member Functions

- bool event (QEvent *event)
- void layoutHUDs (QVector< int > channels)
- bool **eventFilter** (QObject *obj, QEvent *event)

Private Attributes

- Ui::mainWindow **ui**
- int **currentStep**
- QString **patchfile**
- ShogunReader **sr**
- ShogunShow **sshow**
- ShogunMidiController **smc**
- bool **fileLoaded**

2.10.1 Detailed Description

Defines the main user interface for Shogun, including the dynamic heads-up-display, patchfile parsing, etc.

Definition at line 31 of file shogun_qt.h.

2.10.2 Constructor & Destructor Documentation

2.10.2.1 shogun_qt::shogun_qt (QWidget * *parent* = 0)

Constructor for the Shogun Qt interface. Also calls shogun_qt::selectMidiDevices to prompt the user to select a MIDI output device immediately.

Definition at line 8 of file shogun_qt.cpp.

2.10.3 Member Function Documentation

2.10.3.1 void shogun_qt::changeStep (int *increment*)

Moves an arbitrary number of steps forwards or backwards in the patch list, calling shogun_qt::gotoStep() with `currentStep + increment`.

Parameters

increment	The number of steps to advance. May be negative (to move backwards) or zero (which will have no effect).

Definition at line 66 of file shogun_qt.cpp.

2.10.3.2 bool shogun_qt::event (QEvent * *event*) [private]

Ensures that all keypress events are captured by Shogun, so that navigation commands (arrow keys, etc.) are process properly.

Definition at line 120 of file shogun_qt.cpp.

2.10.3.3 void shogun_qt::gotoStep (int *step*)

Set a particular patchfile step as the "current" step, and emit the appropriate signals so that UI elements are updated and MIDI data are sent.

Parameters

step	The step (zero-indexed) to be set as the current step.

Definition at line 76 of file shogun_qt.cpp.

2.10.3.4 void shogun_qt::layoutHUDs (QVector< int > *channels*) [private]

Given a QVector containing the MIDI channels used in the loaded patchfile, lay out the "heads-up display" UI elements corresponding to each channel. Uses a specific pre-defined layout for each possible number of channels (one to sixteen, inclusive). If fewer than one or more than sixteen channels are specified, the minimum or maximum number of HUDs, respectively, will be included. (For the minimum case, a HUD corresponding to the non-existant channel 0 is generated.)

Definition at line 140 of file shogun_qt.cpp.

2.10.3.5 void shogun_qt::loadFile ( ) [slot]

Prompts for a Shogun patchfile (using the native file browser) and calls ShogunReader::read_patchfile() on the chosen file. Also emits the appropriate singals to update the current step and associated UI elements.

Definition at line 95 of file shogun_qt.cpp.

2.10.3.6 void shogun_qt::on_nextStepButton_clicked ( ) [slot]

Moves one step forwards in the patch list, calling shogun_qt::changeStep() with +1 as argument.

Definition at line 53 of file shogun_qt.cpp.

2.10.3.7 void shogun_qt::on_prevStepButton_clicked ( ) [slot]

Moves one step backwards in the patch list, calling shogun_qt::changeStep() with -1 as argument.

Definition at line 43 of file shogun_qt.cpp.

The documentation for this class was generated from the following files:

- shogun_qt.h
- shogun_qt.cpp

2.11 ShogunBank Class Reference

Public Member Functions

- **ShogunBank** (QString name, int number)
- QString **getName** () const
- int **getNumber** () const
- void **setName** (QString name)
- void **setNumber** (int number)
- const QString **toString** ()

Private Attributes

- QString **name**
- int **number**

2.11.1 Detailed Description

Definition at line 7 of file ShogunBank.h.

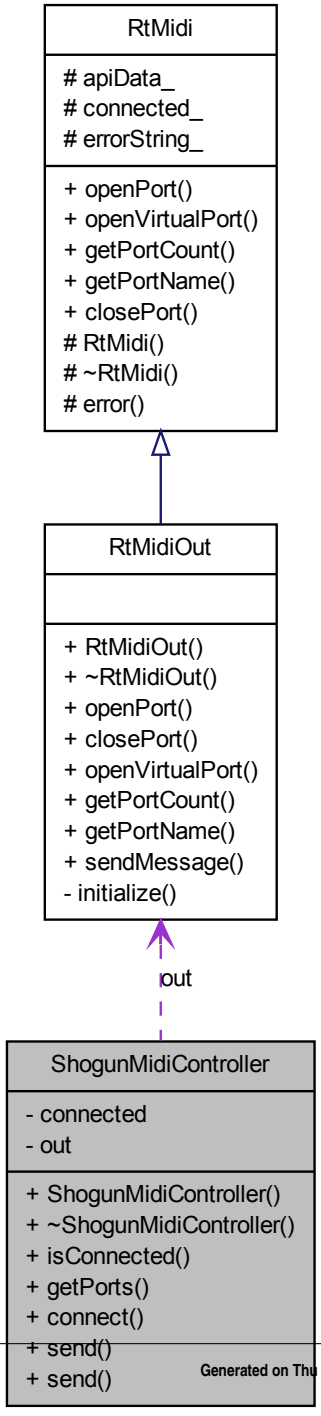
The documentation for this class was generated from the following files:

- ShogunBank.h
- ShogunBank.cpp

2.12 ShogunMidiController Class Reference

```
#include <ShogunMidiController.h>
```

Collaboration diagram for ShogunMidiController:



Public Member Functions

- ShogunMidiController ()
- ∼ShogunMidiController ()
- bool **isConnected** ()
- std::map< int, QString > getPorts ()
- void connect (int port)
- void send (ShogunStep s)
- void send (ShogunVoice v)

Private Attributes

- bool **connected**
- RtMidiOut ∗ **out**

2.12.1 Detailed Description

Provides a subset of RtMidi's functionality (taken primarily from the RtMidiOut class) that represents those MIDI-related actions needed by the Shogun system.

Definition at line 18 of file ShogunMidiController.h.

2.12.2 Constructor & Destructor Documentation

2.12.2.1 ShogunMidiController::ShogunMidiController ( )

Creates a new ShogunMidiController, which will not initially be connected to any MIDI devices.

Exceptions

RtError	

Definition at line 9 of file ShogunMidiController.cpp.

2.12.2.2 ShogunMidiController::∼ShogunMidiController ( )

See also

RtMidiOut::∼RtMidiOut()

Note

By calling RtMidiOut's destructor, this method ensures that any open MIDI device

connections will be properly closed.

Definition at line 25 of file ShogunMidiController.cpp.

2.12.3 Member Function Documentation

2.12.3.1 void ShogunMidiController::connect (int *port*)

Connects to the specified MIDI output port.

Note

This method assumes that the calling method has an up-to-date listing of MIDI devices (as provided by ShogunMidiController::getPorts()) so that the port name-to-number correspondence is consistent.

Attention

If this ShogunMidiController is already connected to a MIDI device, this method ends the existing connection and then attempts to create the new connection.

Parameters

port	Integer representing a MIDI output port.

Definition at line 73 of file ShogunMidiController.cpp.

2.12.3.2 std::map< int, QString > ShogunMidiController::getPorts ( )

Generates and returns a listing of available MIDI output devices. The number assigned to each port may then be used as an argument to ShogunMidiController::connect().

Returns

A mapping of port numbers to their names (as supplied by the operating system)

Definition at line 41 of file ShogunMidiController.cpp.

2.12.3.3 void ShogunMidiController::send (ShogunVoice *v*)

Sends data to the currently-connected MIDI device to effect a specified patch change. A bank-change message and a program-change message will always both be sent, since the MIDI standard specifies that devices should only change banks when a program-change message is received after a bank-change message.

If this ShogunMidiController is not currently connected to a MIDI output device, this method has no effect.

Parameters

v	A ShogunVoices to be sent to the currently-connected MIDI device.

Definition at line 111 of file ShogunMidiController.cpp.

2.12.3.4 void ShogunMidiController::send (ShogunStep *s*)

Sends all voices within the specified ShogunStep to the currently-connected MIDI device using ShogunMidiController::send().

If this ShogunMidiController is not currently connected to a MIDI output device, this method has no effect.

Parameters

s	A ShogunStep containing one or more ShogunVoices, to be sent to the currently-connected MIDI device.

Definition at line 91 of file ShogunMidiController.cpp.

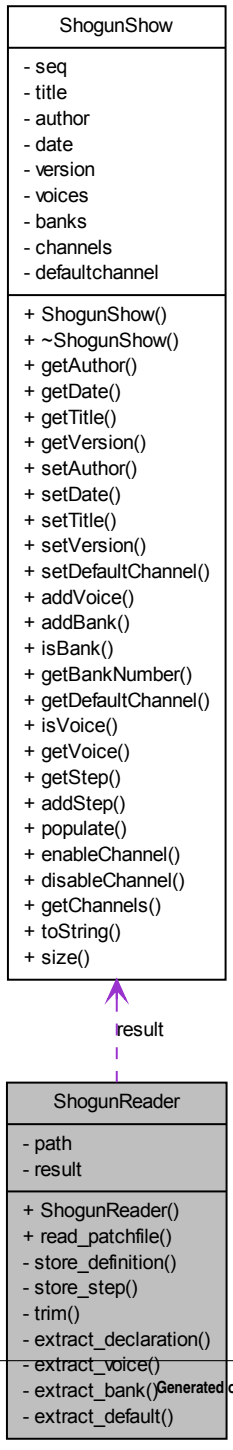
The documentation for this class was generated from the following files:

- ShogunMidiController.h
- ShogunMidiController.cpp

2.13 ShogunReader Class Reference

```
#include <ShogunReader.h>
```

Collaboration diagram for ShogunReader:



Public Slots

- ShogunShow read_patchfile (QString file)

Private Member Functions

- void store_definition (QString line)
- void store_step (QString line)
- QString **trim** (QString s)
- QString extract_declaration (QString line)
- ShogunVoice extract_voice (QString line)
- ShogunBank extract_bank (QString line)
- int extract_default (QString line)

Private Attributes

- QString **path**
- ShogunShow **result**

2.13.1 Detailed Description

Reads a `.shogun` patchfile, and parses the file into Shogun's internal data representation.

See also

ShogunShow, ShogunStep, ShogunVoice, ShogunBank

Definition at line 26 of file ShogunReader.h.

2.13.2 Member Function Documentation

2.13.2.1 ShogunBank ShogunReader::extract_bank (QString *line*) [private]

Given a bank line from a patchfile, returns a ShogunBank containing the name and number of the specified bank.

Parameters

line	

Returns

Definition at line 272 of file ShogunReader.cpp.

2.13.2.2 QString ShogunReader::extract_declaration (QString *line*) [private]

Given a declaration line from a patchfile, extracts the value enclosed in quotation marks.

Parameters

line	A line of the form declaration "this is the value"

Returns

In the example above, return 'this is the value' (without quotation marks)

Definition at line 213 of file ShogunReader.cpp.

2.13.2.3 int ShogunReader::extract_default (QString *line*) [private]

Given a defaultchannel line from a patchfile, returns the value supplied as the patchfile's default MIDI channel.

Parameters

line	

Returns

The specified channel, or 0 if no valid channel number is specified.

Definition at line 295 of file ShogunReader.cpp.

2.13.2.4 ShogunVoice ShogunReader::extract_voice (QString *line*) [private]

Given a voice line from a patchfile, constructs a ShogunVoice containing the bank, program, and (if specified) channel of the voice.

Parameters

line	

Returns

Definition at line 230 of file ShogunReader.cpp.

2.13.2.5 ShogunShow ShogunReader::read_patchfile (QString *file*) [slot]

Reads in and parses a Shogun-format patchfile, and returns a ShogunShow representing the contents of that file. If the specified file path is not valid, an empty ShogunShow will be returned.

Note

Because the file path passed to this message will always come from an operating-system-native file selection dialog box, it is unlikely (but never impossible!) that the specified path will be invalid.

Parameters

file	A string representing a local path to a Shogun patchfile.

Returns

A ShogunShow containing the data specified by the patchfile.

Definition at line 19 of file ShogunReader.cpp.

2.13.2.6 void ShogunReader::store_definition (QString *line*) [private]

Given a line that contains a definition, stores the data in the appropriate place (either as metadata, a bank definition, or a voice definition). If the line does not represent a valid definition, no action will be taken.

Parameters

line	A QString containing a `#[definition]` statement

Definition at line 70 of file ShogunReader.cpp.

2.13.2.7 void ShogunReader::store_step (QString *line*) [private]

Extracts one or more ShogunVoice entries from a patchfile "step" line, wraps these in a ShogunStep, and stores this in the ShogunShow that will be returned by this ShogunReader.

This method will parse lines containing patch-change specifications as defined by the Shogun patchfile grammar, which means the following formats will be accepted for each patch change:

```
<voice name>
<voice name>  <channel>
<bank name>   <patch number>
<bank name>   <patch number> <channel>
<bank number> <patch number>
<bank number> <patch number> <channel>
```

One or more such patch change may exist on each line, separated by commas. If the line does not represent one or more valid patch changes, the invalid portions of the line will be discarded and no action taken based on those declarations. (In other words, if a line contains one valid patch-change specification and one invalid specification, the valid specification will be stored, and no other action will be taken.)

Parameters

line	The line from which a ShogunStep is to be constructed.

Definition at line 132 of file ShogunReader.cpp.

The documentation for this class was generated from the following files:

- ShogunReader.h
- ShogunReader.cpp

2.14 ShogunSequence Class Reference

`#include <ShogunSequence.h>`

2.14.1 Detailed Description

Represents a sequence of patch changes, which normally will have been read in from a .shogun patchfile.

Definition at line 8 of file ShogunSequence.h.

The documentation for this class was generated from the following files:

- ShogunSequence.h
- ShogunSequence.cpp

2.15 ShogunShow Class Reference

`#include <ShogunShow.h>`

Public Member Functions

- QString **getAuthor** () const
- QString **getDate** () const
- QString **getTitle** () const
- QString **getVersion** () const
- void **setAuthor** (QString author)
- void **setDate** (QString date)
- void **setTitle** (QString title)
- void **setVersion** (QString version)
- void **setDefaultChannel** (int channel)
- void **addVoice** (ShogunVoice v)
- void **addBank** (ShogunBank b)
- bool **isBank** (QString bankname)
- int **getBankNumber** (QString bankname)
- int **getDefaultChannel** ()
- bool **isVoice** (QString voicename)
- ShogunVoice **getVoice** (QString voicename)
- ShogunStep **getStep** (int step)
- void **addStep** (ShogunStep s)
- void populate ()
- void **enableChannel** (int ch)
- void **disableChannel** (int ch)
- QVector< int > **getChannels** ()
- const QString **toString** ()
- int **size** ()

Private Attributes

- std::deque< ShogunStep > seq
- QString **title**
- QString **author**
- QString **date**
- QString **version**
- QMap< QString, ShogunVoice > **voices**
- QMap< QString, ShogunBank > **banks**
- bool **channels** [16]
- int **defaultchannel**

2.15.1 Detailed Description

A ShogunShow contains all information present in a Shogun patchfile, and provides access to this information to the main Shogun application. This includes sequential and random access to steps in the sequence of patch changes, as well as information about the banks, voices, and channels used in those steps.

Definition at line 21 of file ShogunShow.h.

2.15.2 Member Function Documentation

2.15.2.1 void ShogunShow::populate ( )

Examines the entire ShogunShow and fills in each ShogunStep therein so that each step contains data for every MIDI channel used in the show up to that point.

Definition at line 92 of file ShogunShow.cpp.

2.15.3 Member Data Documentation

2.15.3.1 std::deque<ShogunStep> ShogunShow::seq [private]

So, intuitively, I would want this to have getNextPatch, getPrevPatch, and so on. But should this actually be sendNextPatch, etc? Probably not, or at least not exclusively, since I might want to just get the patch and then get its name for use in the UI. Also, this is just a data object - it shouldn't touch MIDI.

However, dealing with multi-channel patches might be interesting. May need another class, like ShogunStep, that contains a bunch of voices. (Is there such a thing as throwing too many classes at the problem?)

Definition at line 66 of file ShogunShow.h.

The documentation for this class was generated from the following files:

- ShogunShow.h
- ShogunShow.cpp

2.16 ShogunStep Class Reference

Public Member Functions

- void **addVoice** (ShogunVoice v)
- bool **usesChannel** (int ch)

- ShogunVoice **getVoiceByChannel** (int ch)
- const QString **toString** ()

Private Attributes

- QVector< ShogunVoice > **voices**
- bool **channels** [16]

2.16.1 Detailed Description

Definition at line 9 of file ShogunStep.h.

The documentation for this class was generated from the following files:

- ShogunStep.h
- ShogunStep.cpp

2.17 ShogunVoice Class Reference

Public Member Functions

- **ShogunVoice** (QString name, int channel, int bank, int program)
- **ShogunVoice** (QString name, int bank, int program)
- int **getBank** () const
- int **getChannel** () const
- int **getProgram** () const
- void **setBank** (int bank)
- void **setChannel** (int channel)
- void **setProgram** (int program)
- QString **getName** ()
- QString **getOriginalName** ()
- void **setName** (QString name)
- void **setOriginalName** (QString name)
- const QString **toString** ()
- const QString **toStringShort** ()
- bool **isGenerated** ()
- void **markGenerated** ()
- void **markNotGenerated** ()

Private Attributes

- QString **name**
- QString **originalName**
- int **channel**
- int **bank**
- int **program**
- bool **generated**

2.17.1 Detailed Description

Definition at line 7 of file ShogunVoice.h.

The documentation for this class was generated from the following files:

- ShogunVoice.h
- ShogunVoice.cpp

Index