

Fedora Draft Documentation Musicians' Guide

Audio creation and music software in Fedora Linux.



Fedora Draft Documentation Musicians' Guide

Audio creation and music software in Fedora Linux.

Edition 14.9.2

Author

Christopher Antila

crantila@fedoraproject.org

Copyright © 2011 Red Hat, Inc. and others.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at <http://creativecommons.org/licenses/by-sa/3.0/>. The original authors of this document, and Red Hat, designate the Fedora Project as the "Attribution Party" for purposes of CC-BY-SA. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

For guidelines on the permitted uses of the Fedora trademarks, refer to https://fedoraproject.org/wiki/Legal:Trademark_guidelines.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

All other trademarks are the property of their respective owners.

This document explores some audio creation and music activities possible with Fedora Linux. Computer audio concepts are explained, and a selection of programs are demonstrated with tutorials that show their typical usage.

Preface	xi
1. Document Conventions	xi
1.1. Typographic Conventions	xi
1.2. Pull-quote Conventions	xii
1.3. Notes and Warnings	xiii
2. We Need Feedback!	xiii
 I. Linux Audio Basics	 1
1. Sound Cards and Digital Audio	3
1.1. Types of Sound Cards	3
1.1.1. Audio Interfaces	3
1.1.2. MIDI Interfaces	3
1.2. Sound Card Connections	3
1.2.1. Integrated into the Motherboard	4
1.2.2. Internal PCI Connection	4
1.2.3. External FireWire Connection	4
1.2.4. External USB Connection	4
1.2.5. Choosing a Connection Type	4
1.3. Sample, Sample Rate, Sample Format, and Bit Rate	4
1.3.1. Sample	5
1.3.2. Sample Format	5
1.3.3. Sample Rate	6
1.3.4. Bit Rate	6
1.3.5. Conclusions	6
1.4. Other Digital Audio Concepts	6
1.4.1. MIDI Sequencer	6
1.4.2. Busses, Master Bus, and Sub-Master Bus	7
1.4.3. Level (Volume/Loudness)	7
1.4.4. Panning and Balance	8
1.4.5. Time, Timeline, and Time-Shifting	9
1.4.6. Synchronization	10
1.4.7. Routing and Multiplexing	10
1.4.8. Multichannel Audio	11
2. Software for Sound Cards	13
2.1. How Linux Deals with Audio Hardware	13
2.2. Sound Servers	13
2.2.1. PulseAudio	13
2.2.2. JACK Audio Connection Kit	14
2.2.3. Phonon	14
2.3. Using the JACK Audio Connection Kit	14
2.3.1. Installing and Configuring JACK	14
2.3.2. Using QjackCtl	15
2.3.3. Integrating PulseAudio with JACK	15
3. Real-Time and Low Latency	17
3.1. Why Low Latency Is Desirable	17
3.2. Processor Scheduling	17
3.3. The Real-Time Linux Kernel	18
3.4. Hard and Soft Real-Time	18
3.5. Getting a Real-Time Kernel in Fedora Linux	18
4. Planet CCRMA at Home	21
4.1. About Planet CCRMA at Home	21

4.2. Deciding Whether to Use Planet CCRMA at Home	21
4.2.1. Exclusive Software	21
4.2.2. Security and Stability	21
4.2.3. A Possible "Best Practices" Solution	22
4.3. Using Software from Planet CCRMA at Home	22
4.3.1. Installing the Planet CCRMA at Home Repositories	23
4.3.2. Set Repository Priorities	23
4.3.3. Prevent a Package from Being Updated	24

II. Audio and Music Software 25

5. Audacity 27

5.1. Knowing When to Use Audacity	27
5.2. Requirements and Installation	27
5.2.1. Software Requirements	27
5.2.2. Hardware Requirements	27
5.2.3. Standard Installation	27
5.2.4. Installation with MP3 Support	28
5.2.5. Post-Installation Test: Playback	28
5.2.6. Post-Installation Test: Recording	28
5.3. Configuration	29
5.3.1. When You Run Audacity for the First Time	29
5.3.2. Configuring Audacity for Your Sound Card	29
5.3.3. Setting the Project's Sample Rate and Format	30
5.4. The Interface	31
5.5. Recording	32
5.5.1. Start to Record	32
5.5.2. Continue to Record	33
5.6. Creating a New Login Sound (Tutorial)	33
5.6.1. Files for the Tutorial	33
5.6.2. Scenario	34
5.6.3. Align Tracks	34
5.6.4. Stretching Tracks	34
5.6.5. Adjust the Volume Level	35
5.6.6. Remove Noise	35
5.6.7. Fade In or Out	35
5.6.8. Remove Some Audio	36
5.6.9. Repeat an Already-Recorded Segment	36
5.6.10. Add a Special Effect (the Phaser)	36
5.6.11. Conclusion	37
5.7. Save and Export	37
5.7.1. Export Part of a File	37
5.7.2. Export a Whole File	37

6. Digital Audio Workstations 39

6.1. Knowing Which DAW to Use	39
6.2. Stages of Recording	39
6.2.1. Recording	40
6.2.2. Mixing	40
6.2.3. Mastering	40
6.2.4. More Information	40
6.3. Interface Vocabulary	41
6.3.1. Session	41
6.3.2. Track and Multitrack	41

6.3.3. Region, Clip, or Segment	42
6.3.4. Transport and Playhead	43
6.3.5. Automation	43
6.4. User Interface	44
6.4.1. Messages Pane	44
6.4.2. Clock	45
6.4.3. Track Info Pane	46
6.4.4. Track Pane	47
6.4.5. Transport Controls	48
7. Ardour	51
7.1. Requirements and Installation	51
7.1.1. Knowledge Requirements	51
7.1.2. Software Requirements	51
7.1.3. Hardware Requirements	51
7.1.4. Installation	51
7.2. Recording a Session	51
7.2.1. Running Ardour	51
7.2.2. The Interface	51
7.2.3. Setting up the Timeline	55
7.2.4. Connecting Audio Sources to Ardour	56
7.2.5. Setting up the Busses and Tracks	56
7.2.6. Adjusting Recording Level (Volume)	56
7.2.7. Recording a Region	58
7.2.8. Recording More	59
7.2.9. Routing Audio and Managing JACK Connections	61
7.2.10. Importing Existing Audio	61
7.3. Files for the Tutorial	62
7.4. Editing a Song (Tutorial)	63
7.4.1. Add Tracks and Busses	63
7.4.2. Connect the Tracks and Busses	65
7.4.3. Add Regions to Tracks	67
7.4.4. Cut the Regions Down to Size	68
7.4.5. Compare Multiple Recordings of the Same Thing	70
7.4.6. Arrange Regions into the Right Places	74
7.4.7. Listen	77
7.5. Mixing a Song (Tutorial)	77
7.5.1. Setting the Session for Stereo Output and Disabling Edit Groups	77
7.5.2. Set Initial Levels	78
7.5.3. Set Initial Panning	79
7.5.4. Make Further Adjustments with an Automation Track	81
7.5.5. Other Things You Might Want to Do	82
7.5.6. Listen	82
7.6. Mastering a Session	82
7.6.1. Ways to Export Audio	82
7.6.2. Using the Export Window	83
8. Qtractor	85
8.1. Requirements and Installation	85
8.1.1. Knowledge Requirements	85
8.1.2. Software Requirements	85
8.1.3. Hardware Requirements	85
8.1.4. Other Requirements	85
8.1.5. Installation	85
8.2. Configuration	86

8.2.1. Audio Options	86
8.2.2. MIDI Options	86
8.2.3. Configuring MIDI Channel Names	87
8.3. Using Qtractor	87
8.3.1. Using the Blue Place-Markers	87
8.3.2. Using the MIDI Matrix Editor's Tools	88
8.3.3. Using JACK with Qtractor	89
8.3.4. Exporting a Whole File (Audio and MIDI Together)	90
8.3.5. Miscellaneous Tips	90
8.4. Creating a MIDI Composition (Tutorial)	91
8.4.1. Inspiration	91
8.4.2. Files for the Tutorial	91
8.4.3. Getting Qtractor Ready	92
8.4.4. Import the Audio File	92
8.4.5. Marking the First Formal Area	92
8.4.6. Creating our Theme	92
8.4.7. Repeat the Theme	94
8.4.8. Compose the Next Part	95
8.4.9. Qtractor's Measures 52 to 75	95
8.4.10. Qtractor's Measures 75 to 97	96
8.4.11. Qtractor's Measure 97	96
8.4.12. Qtractor's Measures 98 to 119	96
8.4.13. Qtractor's Measures 119 to 139	96
8.4.14. Qtractor's Measures 139 to 158	97
8.4.15. Qtractor's Measures 158 to 176	97
8.4.16. Qtractor's Measures 177 to the End	97
9. Rosegarden	99
9.1. Requirements and Installation	99
9.1.1. Knowledge Requirements	99
9.1.2. Software Requirements	99
9.1.3. Hardware Requirements	99
9.1.4. Other Requirements	99
9.1.5. Installation	99
9.2. Configuration	99
9.2.1. Setup JACK and Qsynth	99
9.2.2. Setup Rosegarden	100
9.3. Rosegarden and LilyPond	100
9.4. Write a Song in Rosegarden (Tutorial)	101
9.4.1. Start the Score with a Bass Line	101
9.4.2. Add a Percussion Track	102
9.4.3. Spice up the Percussion	103
9.4.4. Add a Melody	103
9.4.5. Possible Ways to Continue	104
10. FluidSynth	105
10.1. SoundFont Technology and MIDI	105
10.1.1. How to Get a SoundFont	105
10.1.2. MIDI Instruments, Banks, Programs, and Patches	105
10.1.3. MIDI Channels	106
10.2. Requirements and Installation	106
10.2.1. Software Requirements	106
10.2.2. There Are Two Ways to Install FluidSynth	106
10.2.3. Installation with Qsynth	106
10.2.4. Installation without Qsynth	107

10.2.5. Installation of SoundFont Files	107
10.3. Using FluidSynth in a Terminal	108
10.4. Configuring Qsynth	108
10.4.1. Starting FluidSynth	108
10.4.2. SoundFont Configuration	108
10.4.3. JACK Output Configuration	109
10.4.4. MIDI Input Configuration	109
10.4.5. Viewing all FluidSynth Settings	110
10.5. Assigning Programs to Channels with Qsynth	110
10.5.1. Changing the Number of MIDI Input Channels	110
10.5.2. Saving and Reusing Channel Assignments	111
10.6. Using Reverb and Chorus with Qsynth	111
10.7. Multiple FluidSynth Instances with Qsynth	112
11. SuperCollider	113
11.1. Requirements and Installation	113
11.1.1. Knowledge Requirements	113
11.1.2. Software Requirements	113
11.1.3. Hardware Requirements	114
11.1.4. Available SuperCollider Packages	114
11.1.5. Recommended Installation	115
11.2. Using GEdit to Write and Run SuperCollider Programs	115
11.2.1. Enable and Configure SCEd in GEdit	115
11.2.2. Enable SuperCollider Mode and Start a Server	116
11.2.3. Executing Code in GEdit	116
11.2.4. Other Tips for Using GEdit with SuperCollider	117
11.3. Basic Programming in SuperCollider	117
11.3.1. First Steps	117
11.3.2. Variables and Functions	120
11.3.3. Object-Oriented SuperCollider	126
11.3.4. Sound-Making Functions	129
11.3.5. Multichannel Audio	130
11.3.6. Collections	132
11.3.7. Repeated Execution	139
11.3.8. Conditional Execution	142
11.3.9. Combining Audio; the Mix Class	151
11.3.10. SynthDef and Synth	153
11.3.11. Busses	161
11.3.12. Ordering and Other Synth Features	166
11.3.13. Scheduling	169
11.3.14. How to Get Help	171
11.3.15. Legal Attribution	172
11.4. Composing with SuperCollider	172
11.4.1. Files for the Tutorial	172
11.4.2. Inspiration	173
11.4.3. Designing the First Part	173
11.4.4. Designing the Second Part	175
11.4.5. Creating Ten Pseudo-Random Tones	175
11.4.6. Scheduling the Tones	178
11.4.7. Optimizing the Code	180
11.4.8. Making a Useful Section out of the Second Part	183
11.4.9. Joining the Two Parts	187
11.5. Exporting Sound Files	188
11.5.1. Non-Real-Time Synthesis	188

11.5.2. Recording SuperCollider's Output (Tutorial)	188
12. LilyPond	191
12.1. How LilyPond Works	191
12.2. The LilyPond Approach	191
12.3. Requirements and Installation	192
12.4. LilyPond Basics	192
12.4.1. Letters Are Pitches	192
12.4.2. Numbers Are Durations	193
12.4.3. Articulations	193
12.4.4. Simultaneity	194
12.4.5. Chords	194
12.4.6. Commands	194
12.4.7. Source Files	196
12.4.8. How to Avoid Errors	198
12.5. Work on a Counterpoint Exercise (Tutorial)	200
12.5.1. Files for the Tutorial	200
12.5.2. Start the Score	200
12.5.3. Adjust Frescobaldi's Output	201
12.5.4. Input Notes	201
12.5.5. Format the Score	201
12.6. Work on an Orchestral Score (Tutorial)	203
12.6.1. Files for the Tutorial	203
12.6.2. Start the Score	203
12.6.3. Adjust Frescobaldi's Output	204
12.6.4. Input Notes	205
12.7. Work on a Piano Score (Tutorial)	210
12.7.1. Files for the Tutorial	210
12.7.2. Start the Score	210
12.7.3. Adjust Frescobaldi's Output	211
12.7.4. Input Notes	212
12.7.5. Troubleshoot Errors	214
12.7.6. Format the Score (Piano Dynamics)	215
13. Frescobaldi	217
13.1. Frescobaldi Makes LilyPond Easier	217
13.2. Requirements and Installation	217
13.3. Configuration	218
13.4. Using Frescobaldi	218
14. GNU Solfege	219
14.1. Requirements and Installation	219
14.1.1. Hardware and Software Requirements	219
14.1.2. Other Requirements	219
14.1.3. Required Installation	219
14.1.4. Optional Installation: Csound	219
14.1.5. Optional Installation: MMA	219
14.2. Configuration	219
14.2.1. When You Run Solfege for the First Time	220
14.2.2. Instruments	220
14.2.3. External Programs	220
14.2.4. Interface	221
14.2.5. Practise	221
14.2.6. Sound Setup	221
14.3. Training Yourself	221

14.3.1. Aural Skills and Musical Sensibility	222
14.3.2. Exercise Types	222
14.3.3. Making an Aural Skills Program	223
14.3.4. Supplementary References	224
14.4. Using the Exercises	225
14.4.1. Listening	225
14.4.2. Singing	226
14.4.3. Configure Yourself	227
14.4.4. Rhythm	227
14.4.5. Dictation	228
14.4.6. Harmonic Progressions	229
14.4.7. Intonation	229
A. Revision History	231
Index	233



DRAFT

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the [Liberation Fonts](https://fedorahosted.org/liberation-fonts/)¹ set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

Mono-spaced Bold

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

To see the contents of the file **my_next_bestselling_novel** in your current working directory, enter the **cat my_next_bestselling_novel** command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

Press **Enter** to execute the command.

Press **Ctrl+Alt+F2** to switch to the first virtual terminal. Press **Ctrl+Alt+F1** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **mono-spaced bold**. For example:

File-related classes include **filesystem** for file systems, **file** for files, and **dir** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click

¹ <https://fedorahosted.org/liberation-fonts/>

Close to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find...** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type **ssh *username@domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh *john@example.com***.

The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount */home***.

To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — *username*, *domain.name*, *file-system*, *package*, *version* and *release*. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **mono-spaced roman** and presented thus:

```
books      Desktop  documentation  drafts  mss    photos  stuff  svn
books_tests Desktop1  downloads      images  notes  scripts svgs
```

Source-code listings are also set in **mono-spaced roman** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;
```

```
public class ExClient
{
    public static void main(String args[])
        throws Exception
    {
        InitialContext iniCtx = new InitialContext();
        Object          ref    = iniCtx.lookup("EchoBean");
        EchoHome        home   = (EchoHome) ref;
        Echo             echo   = home.create();

        System.out.println("Created Echo");

        System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
    }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.



Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.



Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in Bugzilla: <http://bugzilla.redhat.com/bugzilla/> against the product **Fedora Documentation**.

When submitting a bug report, be sure to mention the manual's identifier: *musicians-guide*

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

DRAFT

Part I. Linux Audio Basics



DRAFT

Sound Cards and Digital Audio

This chapter introduces the technical vocabulary used for computer audio hardware.

1.1. Types of Sound Cards

A sound card is a hardware device which allows a computer to process sound. Most sound cards are either audio interfaces or MIDI interfaces. These two kinds of interfaces are described below.

1.1.1. Audio Interfaces

An audio interface is a hardware device that provides a connection between your computer and audio equipment, including microphones and speakers. Audio interfaces usually convert audio signals between analog and digital formats: signals entering the computer are passed through an analog-to-digital convertor, and signals leaving the computer are passed through a digital-to-analog convertor. Some audio interfaces have digital input and output ports, which means that other devices perform the conversion between analog and digital signal formats.

The conversion between analog and digital audio signal formats is the primary function of audio interfaces. Real sound has an infinite range of pitch, volume, and durational possibilities. Computers cannot process infinite information, and require sound to be converted to a digital format. Digital sound signals have a limited range of pitch, volume, and durational possibilities. High-quality analog-to-digital and digital-to-analog convertors change the signal format in a way that keeps the original, analog signal as closely as possible. These quality of the convertors is very important in determining the quality of an audio interface.

Audio interfaces also provide connectors for external audio equipment, like microphones, speakers, headphones, and electric instruments like electric guitars.

1.1.2. MIDI Interfaces

Musical Instrument Digital Interface (MIDI) is a standard used to control digital musical devices. Many people associate the term with low-quality imitations of acoustic instruments. This is unfortunate, because MIDI signals themselves do not have a sound. MIDI signals are instructions to control devices: they tell a synthesizer when to start and stop a note, how long the note should be, and what pitch it should have. The synthesizer follows these instructions and creates an audio signal. Many MIDI-controlled synthesizers are low-quality imitations of acoustic instruments, but many are high-quality imitations. MIDI-powered devices are used in many mainstream and non-mainstream musical situations, and can be nearly indistinguishable from actual acoustic instruments. MIDI interfaces only transmit MIDI signals, not audio signals. Some audio interfaces have built-in MIDI interfaces, allowing both interfaces to share the same physical device.

In order to create sound from MIDI signals, you need a "MIDI synthesizer." Some MIDI synthesizers have dedicated hardware, and some use only software. A software-only MIDI synthesizer, based on SoundFont technology, is discussed in [Chapter 10, FluidSynth](#)

You can use MIDI signals, synthesizers, and applications without a hardware-based MIDI interface. All of the MIDI-capable applications in the Musicians' Guide work well with software-based MIDI solutions, and are also compatible with hardware-based MIDI devices.

1.2. Sound Card Connections

Audio interfaces and MIDI interfaces can both use the following connection methods. In this section, "sound card" means "audio interface or MIDI interface."

1.2.1. Integrated into the Motherboard

Integrated sound cards are built into a computer's motherboard. The quality of audio produced by these sound cards has been increasing, and they are sufficient for most non-professional computer audio work. If you want a professional-sounding audio interface, or if you want to connect high-quality devices, then we recommend an additional audio interface.

MIDI interfaces are rarely integrated into a motherboard.

1.2.2. Internal PCI Connection

Sound cards connected to a motherboard by PCI or PCI-Express offer better performance and lower latency than USB or FireWire-connected sound cards. Professional-quality sound cards often include an external device, connected to the sound card, to which the audio equipment is connected. You cannot use these sound cards with a notebook or netbook computer.

1.2.3. External FireWire Connection

FireWire-connected sound cards are not as popular as USB-connected sound cards, but they are generally higher quality. This is partly because FireWire-connected sound cards use FireWire's "guaranteed bandwidth" and "bus-mastering" capabilities, which both reduce latency. High-speed FireWire connections are also available on older computers without a high-speed USB connection.

FireWire devices are sometimes incompatible with the standard Fedora Linux kernel. If you have a FireWire-connected sound card, you should use the kernel from Planet CCRMA at Home. Refer to [Section 3.5, "Getting a Real-Time Kernel in Fedora Linux"](#) for instructions to install the Planet CCRMA at Home kernel.

1.2.4. External USB Connection

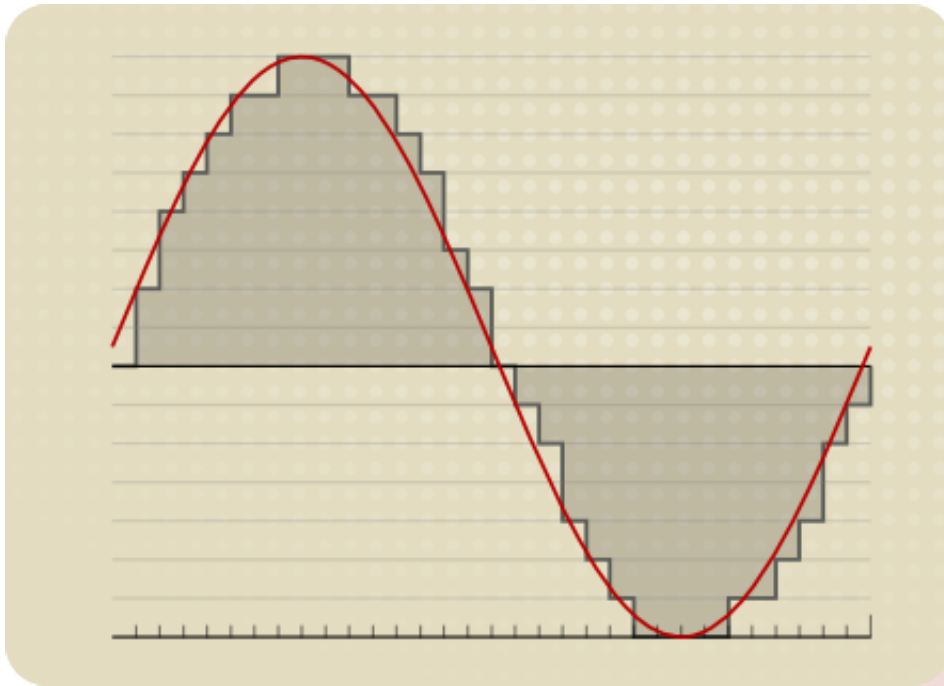
Sound cards connected by USB are becoming more popular, especially because notebook and netbook computer are becoming more popular. The quality can be as good as an internally-connected sound card, but the USB connection may add additional latency. USB-connected sound cards are generally the most affordable sound card for amateur musicians who want a high-quality sound card.

1.2.5. Choosing a Connection Type

The connection type is only one of the considerations when choosing a sound card. If you have a desktop computer, and you will not be using a notebook or netbook computer for audio, you should consider an internal PCI or PCI-Express connection. If you want an external sound card, you should consider a FireWire connection. If FireWire-connected sound cards are more too expensive, you should consider a USB connection. The connection type is not the most important consideration when choosing a sound card. The subjective quality of the analog-to-digital and digital-to-analog convertors is the most important consideration.

1.3. Sample, Sample Rate, Sample Format, and Bit Rate

The primary function of audio interfaces is to convert signals between analog and digital formats. As mentioned earlier, real sound has an infinite possibility of pitches, volumes, and durations. Computers cannot process infinite information, so the audio signal must be converted before they can use it.



source: **pcm.svg**, available from <http://commons.wikimedia.org/wiki/File:Pcm.svg>

Figure 1.1. A waveform approximated by computer

The diagram in *Figure 1.1, "A waveform approximated by computer"* illustrates the situation. The red wave shape represents a sound wave that could be produced by a singer or an acoustic instrument. The gradual change of the red wave cannot be processed by a computer, which must use an approximation, represented by the gray, shaded area of the diagram. This diagram is an exaggerated example, and it does not represent a real recording.

The conversion between analog and digital signals distinguishes low-quality and high-quality audio interfaces. The sample rate and sample format control the amount of audio information that is stored by the computer. The greater the amount of information stored, the better the audio interface can approximate the original signal from the microphone. The possible sample rates and sample formats only partially determine the quality of the sound captured or produced by an audio interface. For example, an audio interface integrated into a motherboard may be capable of a 24-bit sample format and 192 kHz sample rate, but a professional-level, FireWire-connected audio interface capable of a 16-bit sample format and 44.1 kHz sample rate may sound better.

1.3.1. Sample

A sample is a unit of audio data. Computers store video data as a series of still images (each called a "frame"), and displays them one after the other, changing at a pre-determined rate (called the "frame rate"). Computers store audio data as a series of still sound images (each called a "sample"), and plays them one after the other, changing at a pre-determined rate (called the "sample rate").

The frame format and frame rate used to store video data do not vary much. The sample format and sample rate used to store audio data vary widely.

1.3.2. Sample Format

The sample format is the number of bits used to describe each sample. The greater the number of bits, the more data will be stored in each sample. Common sample formats are 16 bits and 24 bits. 8 bit samples are low-quality, and not used often. 20 bit samples are not commonly used on computers. 32 bit samples are possible, but not supported by most audio interfaces.

1.3.3. Sample Rate

The sample rate is the number of samples played in each second. Sample rates are measured in "Hertz" (abbreviated "Hz"), which means "per second," or in "kilohertz" (abbreviated "kHz"), which means "per second, times one thousand." The sample rate used on audio CDs can be written as 44 100 Hz, or 44.1 kHz, which both have the same meaning. Common sample rates are 44.1 kHz, 48 kHz, and 96 kHz. Other possible sample rates include 22 kHz, 88.2 kHz, and 192 kHz.

1.3.4. Bit Rate

Bit rate is the number of bits in a given time period. Bit rate is usually measured in kilobits per second (abbreviated "kbps" or "kb/s"). This measurement is generally used to refer to amount of information stored in a lossy, compressed audio format.

In order to calculate the bit rate, multiply the sample rate and the sample format. For example, the bit rate of an audio CD (705.6 kb/s) is the sample rate (44.1 kHz) multiplied by the sample format (16 bits). MP3-format files are commonly encoded with a 128 kb/s bit rate.

1.3.5. Conclusions

Both sample rate and sample format have an impact on potential sound quality. The capabilities of your audio equipment, and your intended use of the audio signal will determine the settings you should use.

Here are some widely-used sample rates and sample formats. You can use these to help you decide which sample rate and sample format to use.

- 16-bit samples, 44.1 kHz sample rate. Used for audio CDs. Widely compatible. Bit rate of 705.6 kb/s.
- 24-bit samples, and 96 kHz sample rate. Audio CDs are recorded with these settings, and "down-mixed" later. Bit rate of 2304 kb/s.
- 24-bit samples, and 192 kHz sample rate. Maximum settings for DVD Audio, but not widely compatible. Bit rate of 4608 kb/s.
- 1-bit samples, and 2822.4 kHz sample rate. Used for SuperAudio CDs. Very rare elsewhere. Bit rate of 2822.4 kb/s.

Sample rate and sample format are only part of what determines overall sound quality. Sound quality is subjective, so you must experiment to find the audio interface and settings that work best for what you do.

1.4. Other Digital Audio Concepts

These terms are used in many different audio contexts. Understanding them is important to knowing how to operate audio equipment in general, whether computer-based or not.

1.4.1. MIDI Sequencer

A *sequencer* is a device or software program that produces signals that a synthesizer turns into sound. You can also use a sequencer to arrange MIDI signals into music. The Musicians' Guide covers two digital audio workstations (DAWs) that are primarily MIDI sequencers, Qtractor and Rosegarden. All three DAWs in this guide use MIDI signals to control other devices or effects.

1.4.2. Busses, Master Bus, and Sub-Master Bus

An *audio bus* sends audio signals from one place to another. Many different signals can be inputted to a bus simultaneously, and many different devices or applications can read from a bus simultaneously. Signals inputted to a bus are mixed together, and cannot be separated after entering a bus. All devices or applications reading from a bus receive the same signal.

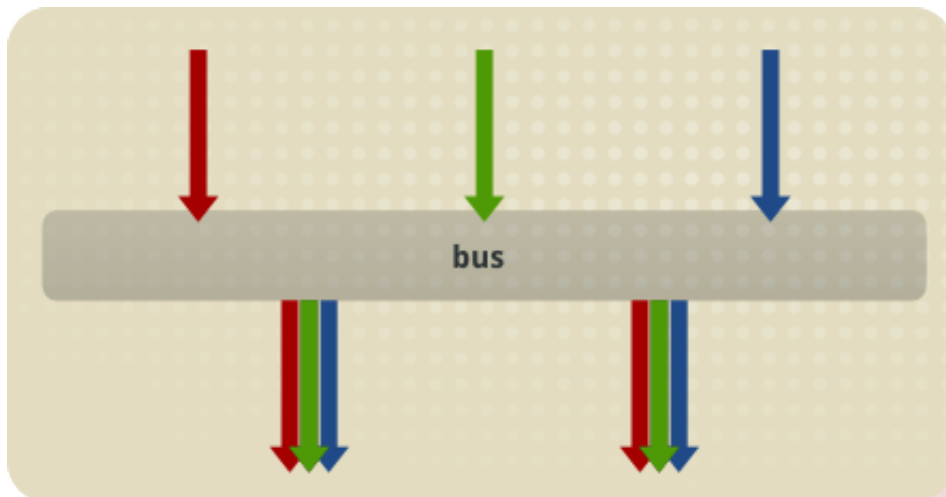


Figure 1.2. How audio busses work

All audio routed out of a program passes through the master bus. The *master bus* combines all audio tracks, allowing for final level adjustments and simpler mastering. The primary purpose of the master bus is to mix all of the tracks into two channels.

A *sub-master bus* combines audio signals before they reach the master bus. Using a sub-master bus is optional. They allow you to adjust more than one track in the same way, without affecting all the tracks.

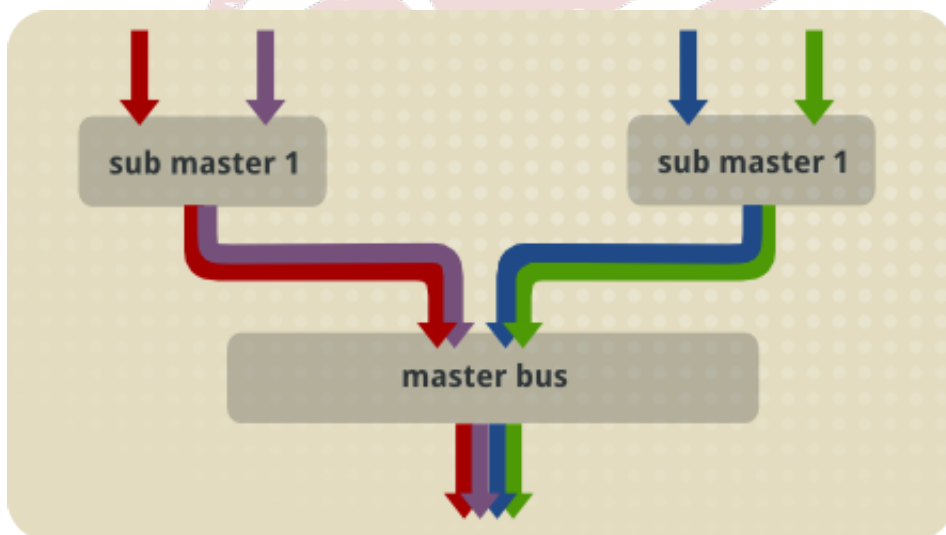


Figure 1.3. The relationship between the master bus and sub-master busses

Audio busses are also used to send audio into effects processors.

1.4.3. Level (Volume/Loudness)

The perceived *volume* or *loudness* of sound is a complex phenomenon, not entirely understood by experts. One widely-agreed method of assessing loudness is by measuring the sound pressure level (SPL), which is measured in decibels (dB) or bels (B, equal to ten decibels). In audio production

communities, this is called level. The *level* of an audio signal is one way of measuring the signal's perceived loudness. The level is part of the information stored in an audio file.

There are many different ways to monitor and adjust the level of an audio signal, and there is no widely-agreed practice. One reason for this situation is the technical limitations of recorded audio. Most level meters are designed so that the average level is -6 dB on the meter, and the maximum level is 0 dB. This practice was developed for analog audio. We recommend using an external meter and the "K-system," described in a link below. The K-system for level metering was developed for digital audio.

In the Musicians' Guide, this term is called "volume level," to avoid confusion with other levels, or with perceived volume or loudness.

- *Level Practices*, available at <http://www.digido.com/level-practices-part-2-includes-the-k-system.html>. The type of meter described here is available in the "jkmeter" package from Planet CCRMA at Home.
- *K-system (Wikipedia)*, available at <http://en.wikipedia.org/wiki/K-system>.
- *Headroom (Wikipedia)*, available at http://en.wikipedia.org/wiki/Headroom_%28audio_signal_processing%29.
- *Equal-Loudness Contour (Wikipedia)*, available at http://en.wikipedia.org/wiki/Equal-loudness_contour.
- *Sound Level Meter (Wikipedia)*, available at http://en.wikipedia.org/wiki/Sound_level_meter.
- *Listener Fatigue (Wikipedia)*, available at http://en.wikipedia.org/wiki/Listener_fatigue.
- *Dynamic Range Compression (Wikipedia)*, available at http://en.wikipedia.org/wiki/Dynamic_range_compression.
- *Alignment Level (Wikipedia)*, available at http://en.wikipedia.org/wiki/Alignment_level.

1.4.4. Panning and Balance

Panning adjusts the portion of a channel's signal that is sent to each output channel. In a stereophonic (two-channel) setup, the two channels represent the "left" and the "right" speakers. Two channels of recorded audio are available in the DAW, and the default setup sends all of the "left" recorded channel to the "left" output channel, and all of the "right" recorded channel to the "right" output channel. Panning sends some of the left recorded channel's level to the right output channel, or some of the right recorded channel's level to the left output channel. Each recorded channel has a constant total output level, which is divided between the two output channels.

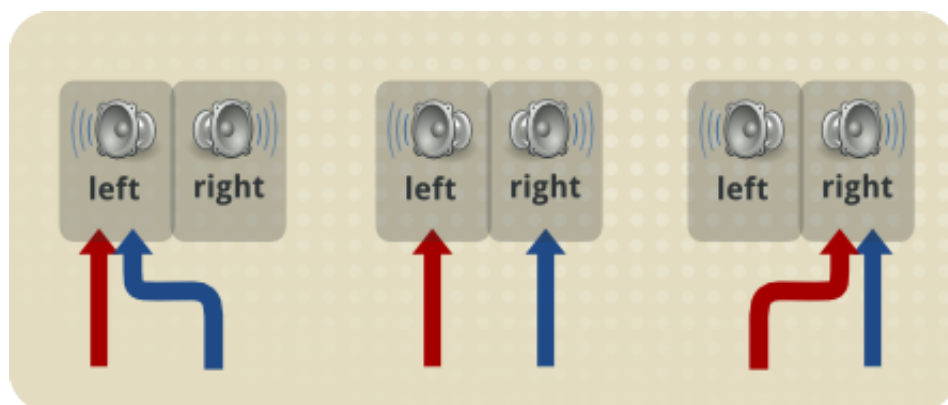


Figure 1.4. Panning

The default setup for a left recorded channel is for "full left" panning, meaning that 100% of the output level is output to the left output channel. An audio engineer might adjust this so that 80% of the recorded channel's level is output to the left output channel, and 20% of the level is output to the right output channel. An audio engineer might make the left recorded channel sound like it is in front of the listener by setting the panner to "center," meaning that 50% of the output level is output to both the left and right output channels.

Balance is sometimes confused with panning, even on commercially-available audio equipment. Adjusting the *balance* changes the volume level of the output channels, without redirecting the recorded signal. The default setting for balance is "center," meaning 0% change to the volume level. As you adjust the dial from "center" toward the "full left" setting, the volume level of the right output channel is decreased, and the volume level of the left output channel remains constant. As you adjust the dial from "center" toward the "full right" setting, the volume level of the left output channel is decreased, and the volume level of the right output channel remains constant. If you set the dial to "20% left," the audio equipment would reduce the volume level of the right output channel by 20%, increasing the perceived loudness of the left output channel by approximately 20%.

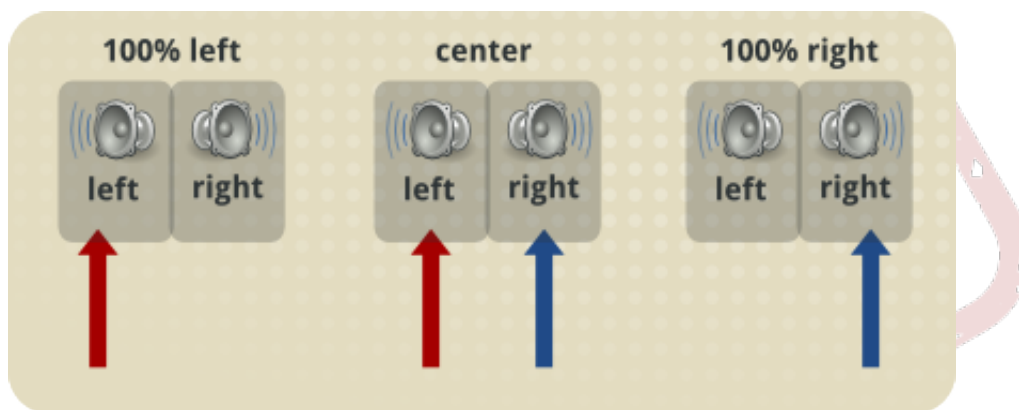


Figure 1.5. Balance

You should adjust the balance so that you perceive both speakers as equally loud. Balance compensates for poorly set up listening environments, where the speakers are not equal distances from the listener. If the left speaker is closer to you than the right speaker, you can adjust the balance to the right, which decreases the volume level of the left speaker. This is not an ideal solution, but sometimes it is impossible or impractical to set up your speakers correctly. You should adjust the balance only at final playback.

1.4.5. Time, Timeline, and Time-Shifting

There are many ways to measure musical time. The four most popular time scales for digital audio are:

- Bars and Beats: Usually used for MIDI work, and called "BBT," meaning "Bars, Beats, and Ticks." A tick is a partial beat.
- Minutes and Seconds: Usually used for audio work.
- SMPTE Timecode: Invented for high-precision coordination of audio and video, but can be used with audio alone.
- Samples: Relating directly to the format of the underlying audio file, a sample is the shortest possible length of time in an audio file. See [Section 1.3, "Sample, Sample Rate, Sample Format, and Bit Rate"](#) for more information.

Most audio software, particularly digital audio workstations (DAWs), allow the user to choose which scale they prefer. DAWs use a *timeline* to display the progression of time in a session, allowing you to do *time-shifting*; that is, adjust the time in the timeline when a region starts to be played.

Time is represented horizontally, where the leftmost point is the beginning of the session (zero, regardless of the unit of measurement), and the rightmost point is some distance after the end of the session.

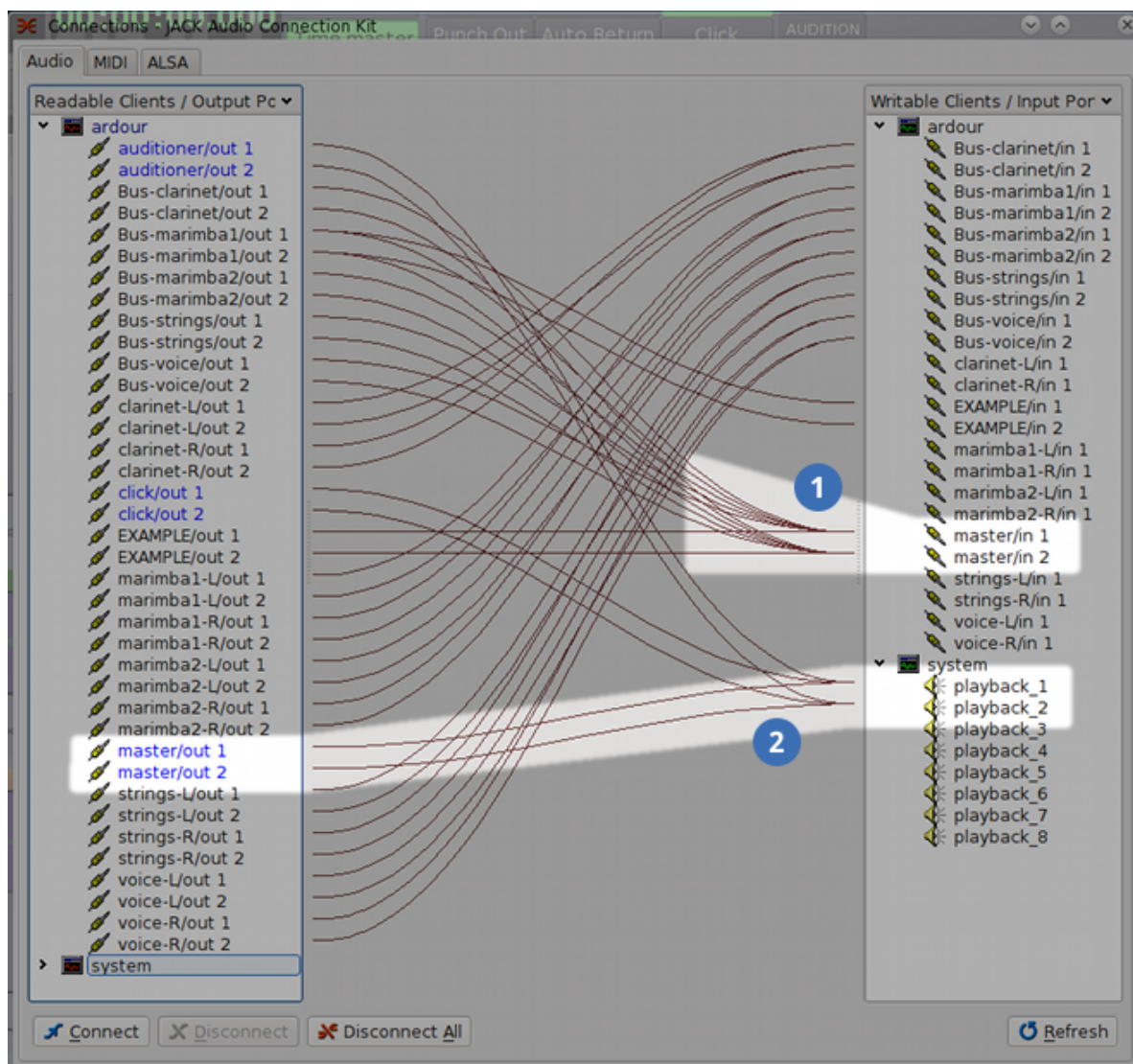
1.4.6. Synchronization

Synchronization coordinates the operation of multiple tools, most often the movement of the transport. Synchronization also controls automation across applications and devices. MIDI signals are usually used for synchronization.

1.4.7. Routing and Multiplexing

Routing audio transmits a signal from one place to another — between applications, between parts of applications, or between devices. On Linux systems, the JACK Audio Connection Kit is used for audio routing. JACK-aware applications (and PulseAudio ones, if so configured) provide inputs and outputs to the JACK server, depending on their configuration. The **QjackCtl** application can adjust the default connections. You can easily reroute the output of a program like FluidSynth so that it can be recorded by **Ardour**, for example, by using **QjackCtl**.

DRAFT



- Master bus inputs accept multiplexed audio from many sources.
- Master bus outputs routed to system playback inputs.

Figure 1.6. Routing and multiplexing

Multiplexing allows you to connect multiple devices and applications to a single input or output.

QjackCtl allows you to easily perform multiplexing. This may not seem important, but remember that only one connection is possible with a physical device like an audio interface. Before computers were used for music production, multiplexing required physical devices to split or combine the signals.

1.4.8. Multichannel Audio

An *audio channel* is a single path of audio data. *Multichannel audio* is any audio which uses more than one channel simultaneously, allowing the transmission of more audio data than single-channel audio.

Audio was originally recorded with only one channel, producing "monophonic," or "mono" recordings. Beginning in the 1950s, stereophonic recordings, with two independent channels, began replacing monophonic recordings. Since humans have two independent ears, it makes sense to record and reproduce audio with two independent channels, involving two speakers. Most sound recordings available today are stereophonic, and people have found this mostly satisfying.

There is a growing trend toward five- and seven-channel audio, driven primarily by "surround-sound" movies, and not widely available for music. Two "surround-sound" formats exist for music: DVD Audio (DVD-A) and Super Audio CD (SACD). The development of these formats, and the devices to use them, is held back by the proliferation of headphones with personal MP3 players, a general lack of desire for improvement in audio quality amongst consumers, and the copy-protection measures put in place by record labels. The result is that, while some consumers are willing to pay higher prices for DVD-A or SACD recordings, only a small number of recordings are available. Even if you buy a DVD-A or SACD-capable player, you would need to replace all of your audio equipment with models that support proprietary copy-protection software. Without this equipment, the player is often forbidden from outputting audio with a higher sample rate or sample format than a conventional audio CD. None of these factors, unfortunately, seem like they will change in the near future.



Software for Sound Cards

One of the techniques consistently used in computer science is abstraction. Abstraction is the process of creating a generic model for something (or some things) that are actually unique. The "driver" for a hardware device in a computer is one form of dealing with abstraction: the computer's software interacts with all sound cards in a similar way, and it is the driver which translates the universal instructions given by the software into specific instructions for operating that hardware device. Consider this real-world comparison: you know how to operate doors because of abstracted instructions. You don't know how to open and close every door that exists, but from the ones that you do know how to operate, your brain automatically creates abstracted instructions, like "turn the handle," and "push the door," which apply with all or most doors. When you see a new door, you have certain expectations about how it works, based on the abstract behaviour of doors, and you quickly figure out how to operate that specific door with a simple visual inspection. The principle is the same with computer hardware drivers: since the computer already knows how to operate "sound cards," it just needs a few simple instructions (the driver) in order to know how to operate any particular sound card.

2.1. How Linux Deals with Audio Hardware

In Linux, the core of the operating system provides hardware drivers for most audio hardware. The hardware drivers, and the instructions that other software can use to connect to those drivers, are collectively called "ALSA," which stands for "Advanced Linux Sound Architecture." ALSA is the most direct way that software applications can interact with audio and MIDI hardware, and it used to be the most common way. However, in order to include all of the features that a software application might want to use, ALSA is quite complex, and can be error-prone. For this and many other reasons, another level of abstraction is normally used, and this makes it easier for software applications to take advantage of the features they need.

2.2. Sound Servers

Sound servers are programs that run "in the background," meaning that they do not have a user interface. Sound servers provide a level of abstraction to automate some aspects of using ALSA, and to allow multiple applications to simultaneously access your audio hardware. The three sound servers discussed in this chapter have different goals and different features. The sound server you should use depends on what you are doing.

2.2.1. PulseAudio

PulseAudio is an advanced sound server, intended to make audio programming in Linux operating systems as easy as possible. The idea behind its design is that an audio application needs only to output audio to PulseAudio, and PulseAudio will take care of the rest: choosing and controlling a particular device, adjusting the volume, working with other applications, and so on. PulseAudio even has the ability to use "networked sound," which allows two computers using PulseAudio to communicate as though they were one computer - either computer can input from or output to either computer's audio hardware just as easily as its own audio hardware. This is all controlled within PulseAudio, so no further complication is added to the software.

The Fedora Project's integration of PulseAudio as a vital part of the operating system has helped to ensure that audio applications can "just work" for most people under most circumstances. This has made it much easier for users to carry out basic audio tasks.

2.2.2. JACK Audio Connection Kit

The JACK sound server offers fewer features than other sound servers, but they are tailor-made to allow the functionality required by audio creation applications. JACK also makes it easier for users to configure the options that are most important for such situations. The server supports only one sample rate and format at a time, and allows applications and hardware to easily connect and multiplex in ways that other sound servers do not (see [Section 1.4.7, “Routing and Multiplexing”](#) for information about routing and multiplexing). It is also optimized to run with consistently low latencies. Although using JACK requires a better understanding of the underlying hardware, the **QjackCtl** application provides a graphical user interface to ease the process.

2.2.3. Phonon

Phonon is a sound server built into the KDE Software Compilation, and is one of the core components of KDE. By default on Fedora Linux, Phonon feeds output to PulseAudio, but on other platforms (like Mac OS X, Windows, other versions of Linux, FreeBSD, and any other system that supports KDE), Phonon can be configured to feed its output anywhere. This is its greatest strength - that KDE applications like Amarok and Dragon Player need only be programmed to use Phonon, and they can rely on Phonon to take care of everything else. As KDE applications increasingly find their place in Windows and especially Mac OS X, this cross-platform capability is turning out to be very useful.

2.3. Using the JACK Audio Connection Kit

2.3.1. Installing and Configuring JACK

1. Use **PackageKit** or **KPackageKit** to install the *jack-audio-connection-kit* and *qjackctl* packages.
2. Review and approve the installation, making sure that it completes correctly.
3. Run **QjackCtl** from the KMenu or the Applications menu.
4. To start the JACK server, click **Start**. To stop the JACK server, click **Stop**.
5. Click **Messages** to see messages, which are usually errors or warnings.
6. Click **Status** to see various statistics about the currently-running server.
7. Click **Connections** button to see and adjust the connections between applications and audio hardware.



Important

JACK operates with special real-time privileges. You must add all users to the `jackuser` and `pulse-rt` groups so they can use JACK. For instructions to add users to groups, see Chapter 22, *Users and Groups* of the *Fedora Deployment Guide*, available at <http://docs.fedoraproject.org>. Do not add users to these groups if they will not use JACK.

With the default configuration of **QjackCtl**, it chooses the "default" sound card, which actually goes through the ALSA sound server. We can avoid this, and use the ALSA drivers without the sound server, which will help JACK to maintain accurately low latencies. The following procedure configures JACK to connect to the ALSA driver directly.

1. Open a terminal. In GNOME, choose **Applications** → **System** → **Terminal**. In KDE, click on the application launcher, then choose **System** → **Konsole**.
2. Execute this command: **cat /proc/asound/cards**
3. The **cat** program outputs a list of sound cards in your computer, which looks similar to this list:

```
0 [SB                ]: HDA-Intel - HDA ATI SB
                        HDA ATI SB at 0xf7ff4000 irq 16
1 [MobilePre         ]: USB-Audio - MobilePre
                        M Audio MobilePre at usb-0000:00:13.0-2
```

In this example output, the square brackets surround the name of the sound card. The names of the sound cards in this example output are **SB** and **MobilePre**.

4. Identify the name of the sound card that you want to use. If you do not see your sound card in the list outputted by **cat**, then your computer does not detect it.
5. Start **QjackCtl**.
6. Click **Setup** to open the "Setup" window.
7. In the 'Interface' text box, type the name of your preferred sound card with "hw:" in front. With the sound cards listed above, you might write **hw:MobilePre**.
8. Save your settings by exiting **QjackCtl**. If you want to use JACK, restart **QjackCtl**.

2.3.2. Using QjackCtl

The **QjackCtl** application offers many more features and configuration options. The patch bay is a notable feature, which lets users save configurations of the "Connections" window, and restore them later, to help avoid the lengthy set-up time that might be required in complicated routing and multiplexing situations.

For more information on **QjackCtl**, refer to *Jack Audio Connection Kit (64studio)* at <http://www.64studio.com/manual/audio/jack>.

2.3.3. Integrating PulseAudio with JACK

The default configuration of PulseAudio yields control of the audio equipment to JACK when the JACK server starts. PulseAudio will not be able to receive input or send output of any audio signals on the audio interface used by JACK. This is fine for occasional users of JACK, but many users will want to use JACK and PulseAudio simultaneously, or switch between the two frequently. The following instructions will configure PulseAudio so that its input and output is routed through JACK.

1. Use PackageKit or KPackageKit to install the *pulseaudio-module-jack* package.
2. Approve the installation and ensure that it is carried out properly.
3. You'll need to edit the PulseAudio configuration file to use the JACK module.
 - a. Be careful! You will be editing an important system file as the root user!
 - b. Run the following command in a terminal: **sudo -c 'gedit /etc/pulse/default.pa'**
 - c. Add the following lines, underneath the line that says `[code]#load-module module-alsa-sink[/
code]`:


```
load-module module-jack-sink
load-module module-jack-source
```

4. Restart PulseAudio by running the following command in a terminal: **killall pulseaudio**. PulseAudio restarts automatically.
5. Confirm that this has worked by opening **QjackCtl**. The display should confirm that JACK is "Active".
6. In the "Connect" window, on the "Audio" tab, there should be PulseAudio devices on each side, and they should be connected to "system" devices on the opposite sides.
7. Open **QjackCtl**'s "Setup" window, then click on the "Options" tab. Uncheck "Execute script after Shutdown: killall jackd". If you did not make this change, then **QjackCtl** would stop the JACK server from running every time the program quits. Since PulseAudio is still expecting to use JACK after that, you shouldn't do this any more.
8. When PulseAudio starts JACK, it uses the command found in the `~/ .jackdrc` file. **QjackCtl** automatically updates this file when you change settings, but you may have to restart both PulseAudio and JACK in order to get the new changes to take effect. If they refuse to take effect, you can edit that file yourself.
9. Be careful about using a very high sample rate with PulseAudio, since it will tend to use a lot of CPU power.

DRAFT

Real-Time and Low Latency

It is perhaps a common perception that computers can compute things instantaneously. Anybody who has ever waited for a web page to load has first-hand experience that this is not the case: computers take time to do things, even if the amount of time is often imperceptible to human observers.

Moreover, a computer doing one thing can seem like it's acting nearly instantaneously, but a computer doing fifteen things will have a more difficult time keeping up appearances.

3.1. Why Low Latency Is Desirable

When computer audio specialists talk about a computer acting in *real-time*, they mean that it is acting with only an imperceptible delay. A computer cannot act on something instantaneously, and the amount of waiting time between an input and its output is called *latency*. In order for the delay between input and output to be perceived as non-existent (in other words, for a computer to "react in real-time,") the latency must be low.

For periodic tasks, like processing audio (which has a consistently recurring amount of data per second), low latency is desirable, but *consistent* latency is usually more important. Think of it like this: years ago in North America, milk was delivered to homes by a dedicated delivery person. Imagine if the milk delivery person had a medium-latency, but consistent schedule, returning every seven days. You would be able to plan for how much milk to buy, and to limit your intake so that you don't run out too soon. Now imagine if the milk delivery person had a low-latency, but inconsistent schedule, returning every one to four days. You would never be sure how much milk to buy, and you wouldn't know how to limit yourself. Sometimes there would be too much milk, and sometimes you would run out. Audio-processing and synthesis software behaves in a similar way: if it has a consistent amount of latency, it can plan accordingly. If it has an inconsistent amount of latency - whether large or small - there will sometimes be too much data, and sometimes not enough. If your application runs out of audio data, there will be noise or silence in the audio signal - both bad things.

Relatively low latency is still important, so that your computer reacts imperceptibly quickly to what's going on. The point is that the difference between an 8 ms target latency and a 16 ms target latency is almost certainly imperceptible to humans, but the higher latency may help your computer to be more consistent - and that's more important.

3.2. Processor Scheduling

If you've ever opened the "System Monitor" application, you will probably have noticed that there are a lot of "processes" running all the time. Some of these processes need the processor, and some of them are just waiting around for something to happen. To help increase the number of processes that can run at the same time, many modern CPUs have more than one "core," which allows for more processes to be evaluated at the same time. Even with these improvements, there are usually more processes than available cores: my computer right now has 196 processes and only three cores. There has to be a way of deciding which process gets to run and when, and this task is left to the operating system.

In Linux systems like Fedora Linux, the core of the operating system (called the *kernel*) is responsible for deciding which process gets to execute at what time. This responsibility is called "scheduling." Scheduling access to the processor is called, *processor scheduling*. The kernel also manages scheduling for a number of other things, like memory access, video hardware access, audio hardware access, hard drive access, and so on. The algorithm (procedure) used for each of these scheduling tasks is different for each, and can be changed depending on the user's needs and the specific hardware being used. In a hard drive, for example, it makes sense to consider the physical location of data on a disk before deciding which process gets to read first. For a processor this is irrelevant, but there are many other things to consider.

There are a number of scheduling algorithms that are available with the standard Linux kernel, and for most uses, a "fair queueing" system is appropriate. This helps to ensure that all processes get an equal amount of time with the processor, and it's unacceptable for audio work. If you're recording a live concert, and the "PackageKit" update manager starts, you don't care if PackageKit gets a fair share of processing time - it's more important that the audio is recorded as accurately as possible. For that matter, if you're recording a live concert, and your computer isn't fast enough to update the monitor, keyboard, and mouse position while providing uninterrupted, high-quality audio, you want the audio instead of the monitor, keyboard, and mouse. After all, once you've missed even the smallest portion of audio, it's gone for good!

3.3. The Real-Time Linux Kernel

There is a "real-time patch" for the Linux kernel which enables the processor to unfairly schedule certain processes that ask for higher priority. Although the term "patch" may make it seem like this is just a partial solution, it really refers to the fact that the programming code used to enable this kind of unfair scheduling is not included in standard kernels; the standard kernel code must have this code "patched" into it.

The default behaviour of a real-time kernel is still to use the "fair queueing" system by default. This is good, because most processes don't need to have consistently low latencies. Only specific processes are designed to request high-priority scheduling. Each process is given (or asks for) a priority number, and the real-time kernel will always give processing time to the process with the highest priority number, even if that process uses up *all* of the available processing time. This puts regular applications at a disadvantage: when a high-priority process is running, the rest of the system may be unable to function properly. In extreme (and very rare!) cases, a real-time process can encounter an error, use up all the processing time, and disallow any other process from running - effectively locking you out of your computer. Security measures have been taken to help ensure this doesn't happen, but as with anything, there is no guarantee. If you use a real-time kernel, you are exposing yourself to a slightly higher risk of system crashes.

A real-time kernel should not be used on a computer that acts as a server, for these reasons.

3.4. Hard and Soft Real-Time

Finally, there are two different kinds of real-time scheduling. The Linux kernel, even at its most extreme, uses only *soft real-time*. This means that, while processor and other scheduling algorithms may be optimized to give preference to higher-priority processes, no absolute guarantee of performance can be made. A real-time kernel helps to greatly reduce the chance of an audio process running out of data, but sometimes it can still happen.

A *hard real-time* computer is designed for specialized purposes, where even the smallest amount of latency can make the difference between life and death. These systems are implemented in hardware as well as software. Example uses include triggering airbag deployment in automobile crashes, and monitoring the heart rate of a patient during an operation. These computers are not particularly multi-functional, which is part of their means to accomplishing a guaranteed low latency.

3.5. Getting a Real-Time Kernel in Fedora Linux

In Fedora Linux, the real-time kernel is provided by the Planet CCRMA at Home software repositories. Along with the warnings in the Planet CCRMA at Home chapter (see [Section 4.2.2, "Security and Stability"](#)), here is one more to consider: the real-time kernel is used by fewer people than the standard kernel, so it is less well-tested. The chances of something going wrong are relatively low, but be aware that using a real-time kernel increases the level of risk. Always leave a non-real-time option available, in case the real-time kernel stops working.

You can install the real-time kernel, along with other system optimizations, by following these instructions:

1. Install the Planet CCRMA at Home repositories by following the instructions in [Section 4.3.1, “Installing the Planet CCRMA at Home Repositories”](#).
2. Run the following command in a terminal: **su -c 'yum install planetccrma-core'** Note that this is a meta-package, which does not install anything by itself, but causes a number of other packages to be installed, which will themselves perform the desired installation and optimization.
3. Shut down and reboot your computer, to test the new kernel. If you decided to modify your GRUB configuration, be sure that you leave a non-real-time kernel available for use.



DRAFT

Planet CCRMA at Home

4.1. About Planet CCRMA at Home

As stated on the project's home page, it is the goal of Planet CCRMA at Home to provide packages which will transform a Fedora Linux-based computer into an audio workstation. What this means is that, while the Fedora Project does an excellent job of providing a general-purpose operating system, a general purpose operating system is insufficient for audio work of the highest quality. The contributors to Planet CCRMA at Home provide software packages which can tune your system specifically for audio work.

Users of GNU Solfege and LilyPond should not concern themselves with Planet CCRMA at Home, unless they also use other audio software. Neither Solfege nor LilyPond would benefit from a computer optimized for audio production.

CCRMA stands for "Center for Computer Research in Music and Acoustics," which is the name of an academic research initiative and music computing facility at Stanford University, located in Stanford, California. Its initiatives help scholars to understand the effects and possibilities of computers and technology in various musical contexts. They offer academic courses, hold workshops and concerts, and try to incorporate the work of many highly-specialized fields.

The Planet CCRMA at Home website suggests that they provide most of the software used on the computers in CCRMA's computing facilities. Much of this software is highly advanced and complex, and not intended for everyday use. More adventurous users are encouraged to explore Planet CCRMA's website, and investigate the software for themselves.

4.2. Deciding Whether to Use Planet CCRMA at Home

4.2.1. Exclusive Software

The only useful reason to install an additional repository is if you intend to install and use its software. The only software application covered in this guide, which is available exclusively from the Planet CCRMA at Home repository, is "SuperCollider". The Planet CCRMA repository also offers many other audio-related software applications, many of which are available from the default Fedora Project repositories.

Most of the audio software currently available in the default Fedora repositories was initially available in Fedora from the Planet CCRMA at Home repository. Sometimes, an updated version of an application is available from the Planet CCRMA repository before it is available from the Fedora Updates repository. If you need the newer software version, then you should install the Planet CCRMA repository.

This is also a potential security weakness, for users who install the Planet CCRMA repository, but do not install any of its software. When "yum" finds a newer version of an installed application, it will be installed, regardless of the repository. This may happen to you without you noticing, so that you begin using Planet CCRMA software without knowing it.

4.2.2. Security and Stability

The biggest reason that you should avoid installing the Planet CCRMA at Home repository unless you *need* its software is security. There are two main security issues with using the Planet CCRMA repositories:

1. Planet CCRMA is intended for specialized audio workstations. The software is packaged in such a way that creates potential (and unknown) security threats caused by the optimizations necessary to prepare a computer system for use in audio work. Furthermore, these optimizations may reveal software bugs present in non-Planet CCRMA software, and allow them to do more damage than on a non-optimized system. Finally, a computer system's "stability" (its ability to run without trouble) may be compromised by audio optimizations. Regular desktop applications may perform less well on audio-optimized systems, if the optimization process unintentionally un-optimized some other process.
2. CCRMA is not a large, Linux-focussed organization. It is an academic organization, and its primary intention with the Planet CCRMA at Home repository is to allow anybody with a computer to do the same kind of work that they do. The Fedora Project is a relatively large organization, backed by one of the world's largest commercial Linux providers, which is focussed on creating a stable and secure operating system for daily use. Furthermore, thousands of people around the world are working for the Fedora Project or its corporate sponsor, and it is their responsibility to proactively solve problems. CCRMA has the same responsibility, but they do not have the dedicated resources of the Fedora Project, it would be naive(???) to think that they would be capable of providing the same level of support.

4.2.3. A Possible "Best Practices" Solution

All Fedora Linux users should be grateful to the people working at CCRMA, who help to provide the Planet CCRMA at Home repository. Their work has been instrumental in allowing Fedora to provide the amount of high-quality audio software that it does. Furthermore, the availability of many of CCRMA's highly-specialized software applications through the Planet CCRMA at Home repository is an invaluable resource to audio and music enthusiasts.

On the other hand, Fedora users cannot expect that Planet CCRMA software is going to meet the same standards as Fedora software. While the Fedora Project's primary goal is to provide Linux software, CCRMA's main goal is to advance the state of knowledge of computer-based music and audio research and art.

Where do these two goals meet?

If you want to use your computer for both day-to-day desktop tasks and high-quality audio production, one good solution is to "dual-boot" your computer. This involves installing Fedora Linux twice on the same physical computer, but it will allow you to keep an entirely separate operating system environment for the Planet CCRMA at Home software. Not only will this allow you to safely and securely run Planet CCRMA applications in their most-optimized state, but you can help to further optimize your system by turning off and even removing some system services that you do not need for audio work. For example, a GNOME or KDE user might choose to install only "Openbox" for their audio-optimized installation.

Alternatively, there is the possibility of going half-way: installing only some Planet CCRMA applications, but not the fully-optimized kernel and system components. This would be more suitable for a computer used most often for typical day-to-day operations (email, word processing, web browsing, etc.) If you wanted to use SuperCollider, but did not require other audio software, for example, then this might be the best solution for you.

Ultimately, it is your responsibility to ensure that your computer and its data is kept safely and securely. You will need to find the best solution for your own work patterns and desires.

4.3. Using Software from Planet CCRMA at Home

The Planet CCRMA at Home software is hosted (stored) on a server at Stanford University. It is separate from the Fedora Linux servers, so **yum** (the command line utility used by PackageKit and

KPackageKit) must be made aware that you wish to use it. After installing the repository, Planet CCRMA at Home software can be installed through yum, PackageKit, or KPackageKit just as easily as any other software.

4.3.1. Installing the Planet CCRMA at Home Repositories

The following steps will install the Planet CCRMA at Home repository, intended only for Fedora Linux-based computers.

1. Update your computer with PackageKit, KPackageKit.
2. Run the following command in a terminal window:

```
su -c 'rpm -Uvh http://ccrma.stanford.edu/planetccrma/mirror/fedora/linux/\
planetccrma/13/i386/planetccrma-repo-1.1-2.fc13.ccrma.noarch.rpm'
```

This works for all versions of Fedora, whether 32-bit and 64-bit. Note that the command is a single line, broken here for presentation purposes.

3. Update your computer again.
4. You may receive a warning that the RPM database was altered outside of "yum". This is normal.
5. Your repository definition will automatically be updated.
6. Some packages are available from Fedora repositories in addition to other repositories (like Planet CCRMA at Home). If the Planet CCRMA repository has a newer version of something than the other repositories that you have installed, then the Planet CCRMA version will be installed at this point.

Although it is necessary to use the **rpm** program directly, all other Planet CCRMA software can be installed through **yum**, like all other applications. Here is an explanation of the command-line options used above:

- **-U** means "upgrade," which will install the specified package, and remove any previously-installed version
- **-v** means "verbose," which will print additional information messages
- **-h** means "hash," which will display hash marks (these: #) showing the progress of installation.

4.3.2. Set Repository Priorities

This is optional, and recommended only for advanced users. **yum** normally installs the latest version of a package, regardless of which repository provides it. Using this plugin will change this behaviour, so that **yum** will choose package versions primarily based on which repository provides it. If a newer version is available at a repository with lower priority, **yum** does not upgrade the package. If you simply wish to prevent a particular package from being updated, the instructions in [Section 4.3.3, "Prevent a Package from Being Updated"](#) are better-suited to your needs.

1. Install the *yum-plugin-priorities* package.
2. Use a text editor or the **cat** or **less** command to verify that **/etc/yum/pluginconf.d/priorities.conf** exists, and contains the following text:

```
[main]
```

```
enabled = 1
```

If you want to stop using the plugin, you can edit this file so that it contains **enabled = 0**. This allows you to keep the priorities as set in the repository configuration files.

3. You can set priorities for some or all repositories. To add a priority to a repository, edit its respective file in the `/etc/yum.repos.d/*` directory, adding a line like: **priority = N** where **N** is a number between **1** and **99**, inclusive. A priority of **1** is the highest setting, and **99** is the lowest. You will need to set priorities of at least two repositories before this becomes useful.

4.3.3. Prevent a Package from Being Updated

This is optional, and recommended only for advanced users. **yum** normally installs the latest version of packages. This plugin prevents certain packages from being updated. If you wish to prevent packages from a particular repository from being used, then [Section 4.3.2, “Set Repository Priorities”](#) is better-suited to your needs.

1. Install the **yum-plugin-versionlock** package.
2. Use a text editor or the **cat** or **less** command to verify that `/etc/yum/pluginconf.d/versionlock.conf` exists, and contains the following text: **enabled = 1**
3. Add the list of packages which you do not want to be updated to `/etc/yum/pluginconf.d/versionlock.list`. Each package should go on its own line. For example:

```
jack-audio-connect-kit-1.9.4
qjackctl-0.3.6
```

Part II. Audio and Music Software



DRAFT

Audacity

Audacity is a high-quality sound recording application, designed to be easy to use. We recommend Audacity to most computer users, because it is simple but it has many features and capabilities. You do not need to understand advanced computer audio concepts before using Audacity. If you can connect your microphone to your computer, you know enough to use Audacity.

5.1. Knowing When to Use Audacity

Audacity has a simple user interface, it is easy to use, and it has many advanced capabilities. Audacity does not require advanced knowledge of computers, music, or recording. Audacity is the right tool to use for editing a single audio file, and it can also coordinate multiple audio files simultaneously. Most users will prefer Audacity over the other applications in the Musicians' Guide which can record.

If you need to record quickly, and you do not have time to learn complicated software, you should use Audacity.

If you have professional-quality audio equipment, if you want to do highly advanced processing, or if you need fine-grained control over the recording, you should use Ardour, Qtractor, or Rosegarden. If you have not used any of these applications, we recommend learning Qtractor before Ardour or Rosegarden. Ardour and Rosegarden are more complicated than Qtractor, and you may not need their advanced features.

5.2. Requirements and Installation

5.2.1. Software Requirements

Audacity uses several "libraries." Libraries are incomplete programs that add capabilities to other programs or applications. Libraries can be shared between programs. The libraries needed by Audacity will be installed automatically.

The version of Audacity from the Fedora repository does not use an MP3 library. If you do not want to use MP3 files with Audacity, you should follow the instructions in [Section 5.2.3, "Standard Installation"](#). If you want to use MP3 files with Audacity, you should follow the instructions in [Section 5.2.4, "Installation with MP3 Support"](#).

Audacity can use the JACK Audio Connection Kit. You should install JACK before installing Audacity. Follow the instructions in [Section 2.3.1, "Installing and Configuring JACK"](#) to install JACK. We recommend using Audacity without JACK, but JACK is installed whether or not you use it.

5.2.2. Hardware Requirements

You need an audio interface to use Audacity. If you will record audio with Audacity, you must have at least one microphone connected to your audio interface. You do not need a microphone to edit existing audio files.

5.2.3. Standard Installation

This method installs Audacity from the Fedora repository. This version of Audacity does not use an MP3 library, and cannot process MP3 files.

1. Use PackageKit or KPackageKit to install the *audacity* package.

2. The proposed installation includes Audacity and all of the libraries that Audacity uses. Continue installing Audacity by reviewing and approving the proposed installation.
3. Audacity configures itself automatically, but it may not use the configuration you want. You need to test Audacity before recording, so that you know that it works. Follow the instructions in [Section 5.2.5, “Post-Installation Test: Playback”](#) and [Section 5.2.6, “Post-Installation Test: Recording”](#) to test Audacity.

5.2.4. Installation with MP3 Support

This method installs Audacity from the RPM Fusion repository. This version of Audacity uses an MP3 library, and can process MP3 files. The Fedora Project cannot provide support for this version of Audacity because it is not prepared by Fedora.

1. Run this command in a terminal:

```
su -c 'yum localinstall --nogpgcheck http://download1.rpmfusion.org/free/fedora/\
rpmfusion-free-release-stable.noarch.rpmhttp://download1.rpmfusion.org/nonfree/\
fedora/rpmfusion-nonfree-release-stable.noarch.rpm'
```

Note that this is a single command, broken into three lines here for presentation reasons.

2. Use PackageKit or KPackageKit to install the **audacity-freeworld** package.
3. The proposed installation includes Audacity and all of the libraries that Audacity uses. Continue installing Audacity by reviewing and approving the proposed installation.
4. Audacity configures itself automatically, but it may not use the configuration you want. You need to test Audacity before recording, so that you know that it works. Follow the instructions in [Section 5.2.5, “Post-Installation Test: Playback”](#) and [Section 5.2.6, “Post-Installation Test: Recording”](#) to test Audacity.

5.2.5. Post-Installation Test: Playback

1. Start Audacity.
2. Set the volume of your audio interface and speakers to a safe level.
3. Choose **File** → **Open** to open the Open File window.
4. Open the **/usr/share/sounds/alsa/Noise.wav** file. This file is designed for testing audio equipment.
5. Play the file as many times as you need. Adjust the volume of your audio interface and speakers while the file is playing.
6. If you cannot hear sound when the file is played, check that your audio interface and speakers are correctly connected and powered on.
7. If you still cannot hear sound when the file is played, see [Section 5.3.2, “Configuring Audacity for Your Sound Card”](#).

5.2.6. Post-Installation Test: Recording

1. Connect your microphones to your audio interface.

2. Start Audacity. Do not open a file.
3. Locate the volume level meters on the toolbar, to the right of the transport controls. If you do not see the meters, choose **View** → **Toolbars** → **Meter Toolbar**, which should have a check mark next to it.
4. Click on the arrow next to the microphone to open the input meter's pop-up menu. Choose **Start Monitoring**.
5. Sing, talk, or make noise into the microphone. The volume level meter should show moving red bars.
6. Adjust the volume of the recording inputs on your audio interface. When there is no noise, the moving red bars should be very small.
7. If you do not see moving red bars in the volume level meter, check that your audio interface and microphone are correctly connected and powered on. If your equipment was not properly connected, adjust the volume of the recording inputs on your audio interface now.
8. If you still do not see moving red bars, see [Section 5.3.2, “Configuring Audacity for Your Sound Card”](#).
9. Click the **Record** button to start a test recording. Sing, talk, or make noise into the microphone.
10. After a few seconds, click the **Stop** button to stop the test recording.
11. Click the **Play** button to hear the recorded audio.
12. If the recording sounds bad, you configure Audacity manually. See [Section 5.3.2, “Configuring Audacity for Your Sound Card”](#).

5.3. Configuration

5.3.1. When You Run Audacity for the First Time

When you run Audacity for the first time, you will be asked to select a language to use for the interface. Before you use Audacity, we encourage you to follow the post-installation test instructions above.

5.3.2. Configuring Audacity for Your Sound Card

Audacity configures itself automatically. If your computer has multiple sound cards or your sound cards have an unusual setup, Audacity guesses which input and output you will use. Audacity sometimes guesses incorrectly, so you can configure it yourself.

Audacity sometimes works, but has poor performance or low-quality audio. If Audacity runs poorly on your computer, you should configure the sound cards manually.

Follow these steps to configure Audacity for your sound card:

1. You need to know the Linux name of your sound card.
 - a. Open a terminal. In GNOME, choose **Applications** → **System** → **Terminal**. In KDE, open the application launcher, then choose **System** → **Konsole**.
 - b. Run this command: `cat /proc/asound/cards`.

- c. The **cat** program outputs a list of sound cards in your computer, which looks similar to this list:

```
0 [SB ]: HDA-Intel - HDA ATI SB
    HDA ATI SB at 0xf7ff4000 irq 16
1 [MobilePre ]: USB-Audio - MobilePre
    M Audio MobilePre at usb-0000:00:13.0-2
```

In this example output, the square brackets surround the name of the sound card. The names of the sound cards in this example output are **SB** and **MobilePre**.

- d. Identify the name of the sound card that you want to use. If you do not see your sound card in the list outputted by **cat**, then your Fedora does not detect it. You should also remember the number of the sound card, which is printed to the left of the name. You can use two different sound cards for recording and playback.

2. Start Audacity.
3. Open the "Preferences" window. Choose **File > Preferences**.
4. Set the "Host" to **ALSA**.
5. Set the recording and playback devices to the name of the sound card that you want to use. If there are many choices for your sound card, choose the one that ends with **(hw:0)**, where **0** is replaced by the number of the sound card that you want to use.
6. Follow the post-installation test procedures to confirm that the configuration is correct.

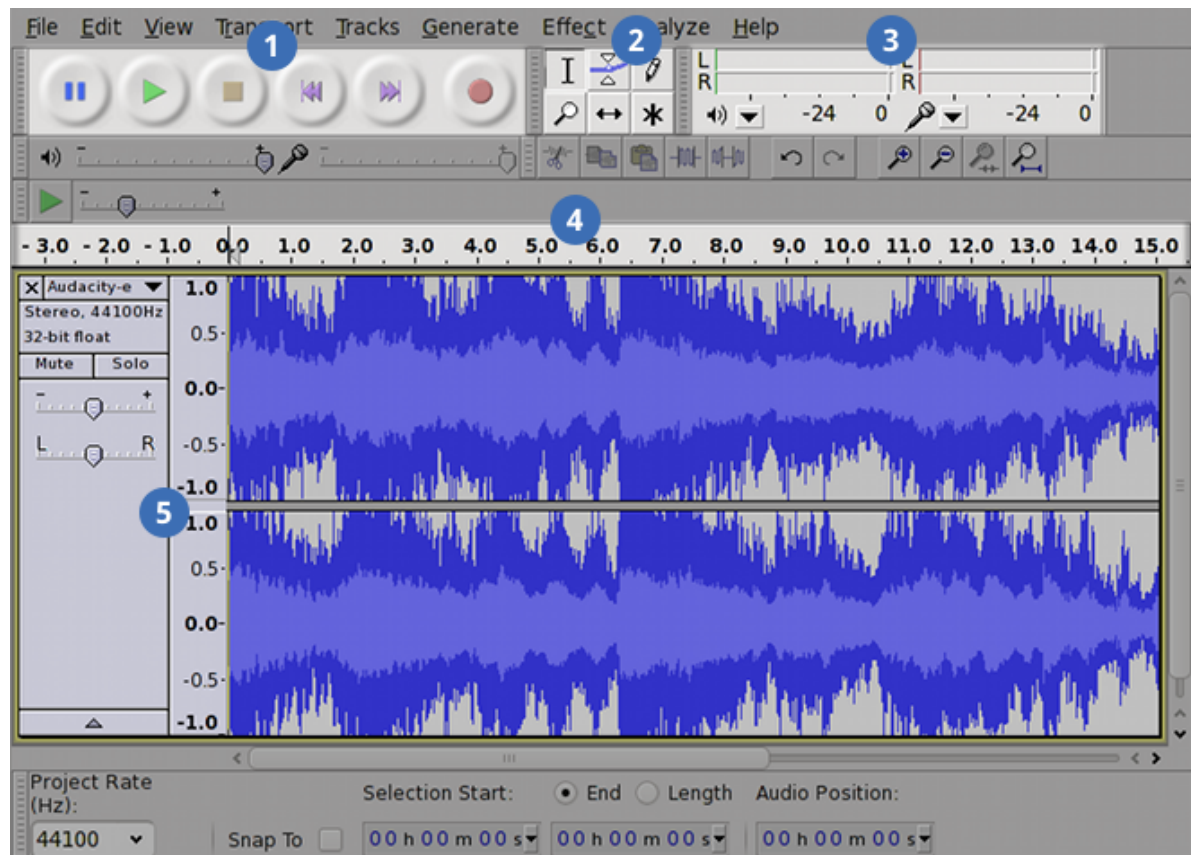
You will see many choices in the list of playback and recording devices. When configured with the **default** device, Audacity lets ALSA determine which sound card to use. When configured with the **pulse** device, Audacity lets PulseAudio determine which sound card to use. Audacity works most efficiently when configured with a specific sound card, so you should not use the **default** or **pulse** devices unless the other choices do not work.

5.3.3. Setting the Project's Sample Rate and Format

You can change the sample rate and sample format (see [Section 1.3, "Sample, Sample Rate, Sample Format, and Bit Rate"](#) for definitions). You should set the sample rate and sample format when you begin working on a project. You should not change the sample rate or sample format after you record audio. If you will use audio files that already exist, you should use the sample rate and sample format of the existing files.

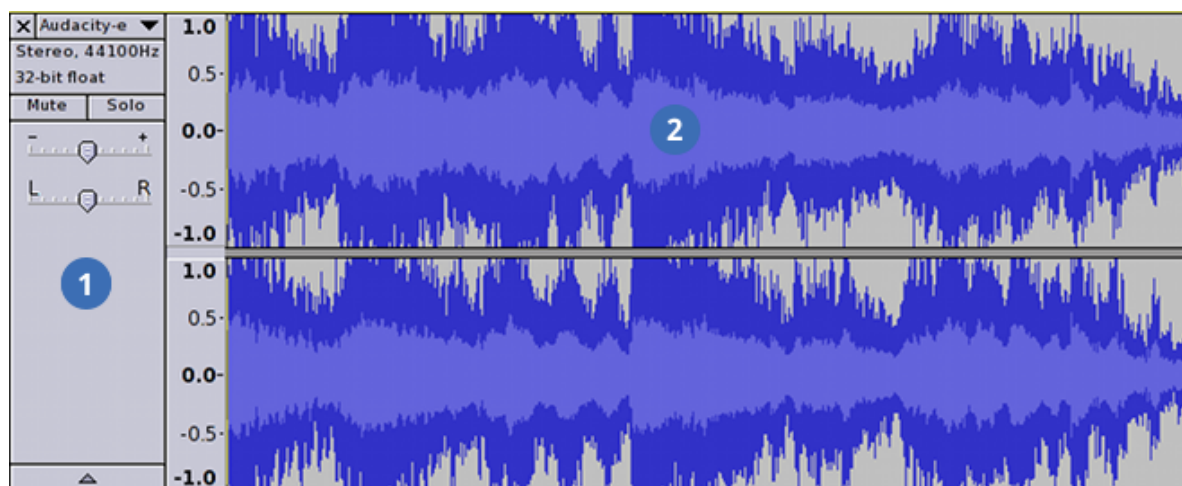
1. Choose **File → Preferences** to open the "Preferences" window.
2. Click **Quality** in the list on the left of the window.
3. Change the settings as you need. We recommend a sample rate of 44 100 Hz and a sample format of 16 bits for most projects.

5.4. The Interface



1. The *transport controls* play, stop, or pause playback of audio. The buttons to record and move quickly through a file are also located here.
2. The *tool-selection box* changes the cursor's function.
3. The two *volume level meters* display the volume level of stereo audio. The left meter displays the volume level of the output signal. The right meter displays the volume level of the input signal.
4. The *ruler* displays the time since the start of the file, in minutes and seconds.
5. Each *track* contains two channels of audio signal data. Audacity stacks tracks vertically in the main window. Audacity plays back all tracks simultaneously.

Figure 5.1. The Audacity interface



1. Each track has a *track info area*, which holds settings like the fader, panner, and **mute** and **solo** buttons.
2. The *timeline* is the main area of the main window of Audacity. The leftmost point is the beginning of the audio file.

Figure 5.2. A track in Audacity

Refer to the image above as you read about the user interface of Audacity.

- The "transport controls" play, stop, or pause playback of audio. The buttons to record and move quickly through a file are also located here.
- The "tool-selection box" changes the cursor's function.
- The two "volume level meters" display the volume level of stereo audio. The left meter displays the volume level of the output signal. The right meter displays the volume level of the input signal.
- The "ruler" displays the time since the start of the file, in minutes and seconds.
- The "timeline" is the main area of the main window of Audacity. The leftmost point is the beginning of the audio file.
- Each "track" contains two channels of audio signal data. Audacity stacks tracks vertically in the main window. Audacity plays back all tracks simultaneously.
- Each track has a "track info area," which holds settings like the fader, panner, and **mute** and **solo** buttons.

[Chapter 6, Digital Audio Workstations](#) contains more information about the Audacity user interface. That chapter defines the purpose and use of many user interface components.

5.5. Recording

This section explains possible ways to use Audacity to record.

5.5.1. Start to Record

This procedure can be used whether you want to record from microphones or from another application on your computer.

1. Prepare your computer to record. Connect the microphones you want to use, and test them. See [Section 5.2.5, “Post-Installation Test: Playback”](#) and [Section 5.2.6, “Post-Installation Test: Recording”](#) for instructions to test your equipment.
2. Check the sample rate and sample format. See [Section 5.3.3, “Setting the Project’s Sample Rate and Format”](#) for instructions to change the sample rate and sample format.
3. The **Record** button shows a red circle. When you are ready to record, click the **Record** button.
4. The **Stop** button shows an orange square. Click the **Stop** button to stop the recorder.
5. * Audacity displays the recorded audio in the timeline of a track, as a blue, sound-wave-like shape that represents the volume level. Audacity sometimes displays the blue shape immediately, and sometimes only after you click **Stop**. When Audacity does not show the blue shape immediately, it does not have enough processor power, and decides to process the audio instead of the blue shape.

5.5.2. Continue to Record

You can record more audio after the first recording in one of these ways:

1. You can continue to record from the end of the already-recorded audio.
 1. Press **Shift** then click **Record**.
 2. Audacity starts to record from the end of the already-recorded audio. The new audio is put in the previously-selected track.
2. You can record new audio that will play at the same time as the already-recorded audio.
 1. Choose **Transport** → **Skip to Start** or press **Home** to move the transport head to the start of the file.
 2. Click **Record**.
 3. Audacity starts to record from the beginning. The new audio is put in a new track, and does not erase existing audio.
3. You can record new audio that will play at the same time as the already-recorded audio, but start after the beginning.
 1. Find the timeline position when you want to record new audio. Click on an existing track at that time.
 2. Click **Record**.
 3. Audacity starts to record from the position of the transport head. The new audio is put in a new track, and does not erase existing audio.

5.6. Creating a New Login Sound (Tutorial)

5.6.1. Files for the Tutorial

Use the *Tutorial Start* file if you want to do the tutorial. The *Tutorial End* file and *Exported FLAC* file are completed examples. When you finish the tutorial, your project will probably not be the same as ours.

- *Tutorial Start* is available at http://docs.fedoraproject.org/en-US/Fedora/15/html/Musicians_Guide/files/Audacity/Audacity-start.tar.lzma

- *Tutorial End* is available at http://docs.fedoraproject.org/en-US/Fedora/15/html/Musicians_Guide/files/Audacity/Audacity-start.tar.lzma
- *Exported FLAC* is available at http://docs.fedoraproject.org/en-US/Fedora/15/html/Musicians_Guide/files/Audacity/Audacity-export.flac

5.6.2. Scenario

You tell a friend about some open-source audio applications in Fedora Linux. Your friend wants you to help them make a new sound that will play when they log in to their computer. You search your music CD collection, and find a favourite recording. You and your friend decide to use a few clips from the CD to start your new log-in sound.

5.6.3. Align Tracks

You adjust the starting time of the segments.

1. Click the **Time Shift** button in the tool-selection toolbar.
2. Select the track with the segment you want to move. Drag the segment to the left or right to change the time when it starts to play.
3. You can align the start of one segment with end of another segment. Drag the segment so that the beginning is near the end of the other segment. Drag the track back and forth slowly so a yellow line appears. The yellow line means that there is no silence between the segments, and that the segments do not overlap.
4. You can move a segment to start at the selection cursor. Click the **Selection Tool** button, then click on the track where you want the segment to start to play. Choose **Tracks** → **Align Tracks** → **Align with Cursor** to move the track.
5. You can move tracks in other ways by using the **Tools** → **Align Tracks** menu choices.
6. You can select multiple tracks at the same time. Press **Shift** while you click on each track.
7. You should listen to each adjustment to confirm that it is correct. Click **Selection Tool** in the tool-selection toolbar, then click on a track where you want to begin to listen. Press **Play** to hear the audio.

We aligned the tracks so that they sound nice to us. We tried to make the sound short, and interesting but not confusing. We changed the positions of the tracks many times after this step.

5.6.4. Stretching Tracks

You can stretch or compress tracks, so that they take up more or less time.

- The "Change Tempo" tool adjusts speed but not pitch. Sound quality is lowered significantly.
 - The "Change Speed" tool adjusts speed and pitch. Sound quality is less affected.
1. Click on the track info portion of the track that you want to adjust.
 2. Choose **Effect** → **Change Tempo** or **Effect** → **Change Speed**. If you input a positive number, the speed increases and the audio takes less time. If you input a negative number, the speed decreases and the audio takes more time.

3. Click **Preview** to hear some of the audio with the adjusted speed. Audacity does not change the audio when you preview the change.
4. Click **OK** to apply the change. Audacity processes the audio. Your change does not appear immediately.

We changed the speed of one of our tracks. We are not concerned about the sound quality, because this is a start-up sound for us. When you edit audio to which other people will listen, you should be careful to preserve the sound quality.

5.6.5. Adjust the Volume Level

You can adjust the volume level of tracks by adjusting the fader in the track info box. The fader is a horizontal line with a - sign on the left and a + sign on the right. The fader does not change the audio signal itself.

You can also adjust the volume level of a portion of audio, which does modify the audio signal itself. Follow these instructions to adjust the volume level of a portion of audio.

1. The tutorial's lowest track is very quiet. Click the track's **solo** button, then listen to the track alone. Most of the track is quiet humming.
2. Click **Selection Tool**, then hold down the mouse button to select some audio in one track.
3. Choose **Effect** → **Amplify**. When the "Amplify" window opens, it is set for the greatest volume level increase that will not decrease sound quality. You can enable "clipping," which allows you to increase the volume level even more, but will decrease the sound quality.
4. Use the Amplify tool to adjust the volume level as you wish.

5.6.6. Remove Noise

"Noise" is part of an audio signal that is not the intended signal. When you listen to music on the radio, you can hear the music, and usually also some noise. When computers record or play audio, the electronic components sometimes create extra noise. The lowest track of the tutorial file has noise created by electronic components in the old notebook computer used to play the CD. Follow these steps to use the "Noise Removal" tool.

1. Click **Selection Tool**, then select a small portion of the lowest track.
2. Choose **Effect** → **Noise Removal**. The "Noise Removal" window appears. Click **Get Profile**. The Noise Removal tool uses the selected audio as an example of the noise to remove.
3. Select the whole track.
4. Open the "Noise Removal" window. Click **OK**. The Noise Removal tool processes the audio. The Noise Removal tool is not effective in this example because most of the signal is noise.

5.6.7. Fade In or Out

Audacity has tools to fade in and out. When you "fade in," it means to gradually increase the volume level from silence to the original level. When you "fade out," it means to gradually decreased the volume level from the original level to silence. Professional recordings of concerts often fade out after a song, while the audience applauds. Fading out avoids an abrupt stop at the end of the recording, because the volume level is decreased gradually to silence.

You generally fade in at the beginning of an audio segment, and fade out at the end of an audio segment. You can use Audacity to create special effects if you fade in or fade out in the middle of an audio segment. You can make a surprising effect by setting Audacity to fade in or out over a few seconds, then adding the opposite fade inside the first fade.

1. Click **Selection Tool**, then select a portion of audio to fade in or out.
2. Choose **Effect** → **Fade In** or **Effect** → **Fade Out**.

We created a long fade out in one of our tracks, so that the track is quieter. We then adjusted the spacing of the other tracks, moving them closer.

5.6.8. Remove Some Audio

You can remove portions of a track. This procedure removes audio at the end of a track. You can use a similar method to remove audio in other ways.

1. Click **Selection Tool**, then place the editing cursor at the point where you want the track to end.
2. Choose **Edit** → **Select** → **Cursor to Track End**. There are many other choices in the **Edit** → **Select** menu.
3. Press **Delete**. The selected audio is deleted.
4. If you make a mistake, you can undo the previous action. Choose **Edit** → **Undo**.

We removed the end of the lowest track.

5.6.9. Repeat an Already-Recorded Segment

You can repeat a segment of audio that is already recorded.

1. Click **Selection Tool**, then select a portion of audio.
2. Choose **Edit** → **Copy** or press **Control+c**.
3. Click in a track near where you want the copy to start playing.
4. Choose **Edit** → **Paste** or press **Control+v**.
5. Click **Time Shift Tool**, then adjust the timeline position of the new audio clip.

We repeated the end of a segment twice, to make the end of the startup sound more interesting.

5.6.10. Add a Special Effect (the Phaser)

Audacity has many effect plugins, and you can add more. The Phaser adjusts reverberation, panning, and frequency to create a spinning-like sound.

1. Select a portion of any track when only one track is playing.
2. Choose **Effect** → **Phaser**, then experiment with the settings. Click **Preview** to hear the effect of your settings. Audacity does not modify the audio file when you preview your settings.
3. The tutorial uses the phaser at the end, with these settings:
 - Stages: 2

- Dry/Wet: 128
- LFO Frequency: 0.4
- LFO Start Phase: 0
- Depth: 128
- Feedback: 90

5.6.11. Conclusion

Your startup file sounds different than the one that we completed. The tutorial instructions are intentionally vague. You used your own creativity to complete the tutorial. You also learned how to use these tools for many different purposes.

5.7. Save and Export

Audacity saves its data in a format that only Audacity can use. Audacity saves more information than just the audio files, and this information cannot be stored in conventional audio files like OGG, FLAC, or AIFF.

When you want to share your work, or use it in another application, you must export it. When you "export" audio, Audacity converts the Audacity-only file into a conventional audio file. Audacity does not delete the original Audacity-only file, but the exported file cannot be converted into the Audacity-only format.

5.7.1. Export Part of a File

1. Select the portion of audio that you want to export.
2. Choose **File** → **Export Selection**.
3. Select the format, filename, and directory of the audio that will be exported. Click **Save** when you are finished.
4. Depending on which format you choose, the "Metadata" window appears. You can input this information if you wish.
5. Audacity processes the audio to export. This may take some time, depending on the audio.

5.7.2. Export a Whole File

1. Choose **File** → **Export**.
2. Select the format, filename, and directory of the audio that will be exported. Click **Save** when you are finished.
3. Depending on which format you choose, the "Metadata" window appears. You can input this information if you wish.
4. Audacity processes the audio to export. This may take some time, depending on the audio.

DRAFT

Digital Audio Workstations

The term *Digital Audio Workstation* (henceforth *DAW*) refers to the entire hardware and software setup used for professional (or professional-quality) audio recording, manipulation, synthesis, and production. It originally referred to devices purpose-built for the task, but as personal computers have become more powerful and wide-spread, certain specially-designed personal computers can also be thought of as DAWs. The software running on these computers, especially software capable of multi-track recording, playback, and synthesis, is simply called "DAW software," which is often shortened to "DAW." So, the term "DAW" and its usage are moderately ambiguous, but generally refer to one of the things mentioned.

For other terms related to digital audio, see [Chapter 1, Sound Cards and Digital Audio](#).

6.1. Knowing Which DAW to Use

The Musicians' Guide covers three widely-used DAWs: **Ardour**, **Qtractor**, and **Rosegarden**. All three use JACK extensively, are highly configurable, share a similar user interface, and allow users to work with both audio and MIDI signals. Many other DAWs exist, including a wide selection of commercially-available solutions. Here is a brief description of the programs documented in the Musicians' Guide:

- **Ardour** is the open-source standard for audio manipulation. Flexible and extensible.
- **Qtractor** is a relative new-comer, but easy to use; a "lean and mean," MIDI-focused DAW. Available from Planet CCRMA at Home or RPM Fusion.
- **Rosegarden** is a well-tested, feature-packed workhorse of Linux audio, especially MIDI. Includes a visual score editor for creating MIDI tracks.

If you are unsure of where to start, then you may not need a DAW at all:

- If you are looking for a high-quality recording application, or a tool for manipulating one audio file at a time, then you would probably be better off with **Audacity**. This will be the choice of most computer users, especially those new to computer audio, or looking for a quick solution requiring little specialized knowledge. **Audacity** is also a good way to get your first computer audio experiences, specifically because it is easier to use than most other audio software.
- To take full advantage of the features offered by **Ardour**, **Qtractor**, and **Rosegarden**, your computer should be equipped with professional-quality audio equipment, including an after-market audio interface and input devices like microphones. If you do not have access to such equipment, then Audacity may be a better choice for you.
- If you are simply hoping to create a "MIDI recording" of some sheet music, you are probably better off using **LilyPond**. This program is designed primarily to create printable sheet music, but it will produce a MIDI-format version of a score if you include the following command in the **score** section of your **LilyPond** source file: `\midi { }`. There are a selection of options that can be put in the **midi** section; refer to the **LilyPond** help files for a listing.

6.2. Stages of Recording

There are three main stages involved in the the process of recording something and preparing it for listeners: recording, mixing, and mastering. Each step of the process has distinct characteristics, yet they can sometimes be mixed together.

6.2.1. Recording

Recording is the process of capturing audio regions (also called "clips" or "segments") into the DAW software, for later processing. Recording is a complex process, involving a microphone that captures sound energy, translates it into electrical energy, and transmits it to an audio interface. The audio interface converts the electrical energy into digital signals, and sends it through the operating system to the DAW software. The DAW stores regions in memory and on the hard drive as required. Every time the musicians perform some (or all) of the performance to be recorded, while the DAW is recording, it is considered to be a *take*. A successful recording usually requires several takes, due to the inconsistencies of musical performance and of the related technological aspects.

6.2.2. Mixing

Mixing is the process through which recorded audio regions (also called "clips") are coordinated to produce an aesthetically-appealing musical output. This usually takes place after recording, but sometimes additional takes will be needed. Mixing often involves reducing audio from multiple tracks into two channels, for stereo audio - a process known as "down-mixing," because it decreases the amount of audio data.

Mixing includes the following procedures, among others:

- automating effects,
- adjusting levels,
- time-shifting,
- filtering,
- panning,
- adding special effects.

When the person performing the mixing decides that they have finished, their finalized production is called the *final mix*.

6.2.3. Mastering

Mastering is the process through which a version of the final mix is prepared for distribution and listening. Mastering can be performed for many target formats, including CD, tape, SuperAudio CD, or hard drive. Mastering often involves a reduction in the information available in an audio file: audio CDs are commonly recorded with 20- or 24-bit samples, for example, and reduced to 16-bit samples during mastering. While most physical formats (like CDs) also specify the audio signal's format, audio recordings mastered to hard drive can take on many formats, including OGG, FLAC, AIFF, MP3, and many others. This allows the person doing the mastering some flexibility in choosing the quality and file size of the resulting audio.

Even though they are both distinct activities, mixing and mastering sometimes use the same techniques. For example, a mastering technician might apply a specific equalization filter to optimize the audio for a particular physical medium.

6.2.4. More Information

It takes experience and practice to gain the skills involved in successful recording, mixing, and mastering. Further information about these procedures is available from many places, including these web pages:

- *Mastering Your Final Mix* (64studio), available at <http://www.64studio.com/howto-mastering>.

- *Audio Mixing (Wikipedia)*, available at http://en.wikipedia.org/wiki/Audio_mixing_%28recorded_music%29.
- *Multitrack Recording (Wikipedia)*, available at http://en.wikipedia.org/wiki/Multitrack_recording.

6.3. Interface Vocabulary

Understanding these concepts is essential to understanding how to use the DAW software's interface.

6.3.1. Session

A *session* is all of the tracks, regions, automation settings, and everything else that goes along with one "file" saved by the DAW software. Some software DAWs manage to hide the entire session within one file, but others instead create a new directory to hold the regions and other data.

Typically, one session is used to hold an entire recording session; it is broken up into individual songs or movements after recording. Sometimes, as in the tutorial examples with the Musicians' Guide, one session holds only one song or movement. There is no strict rule as to how much music should be held within one session, so your personal preference can determine what you do here.

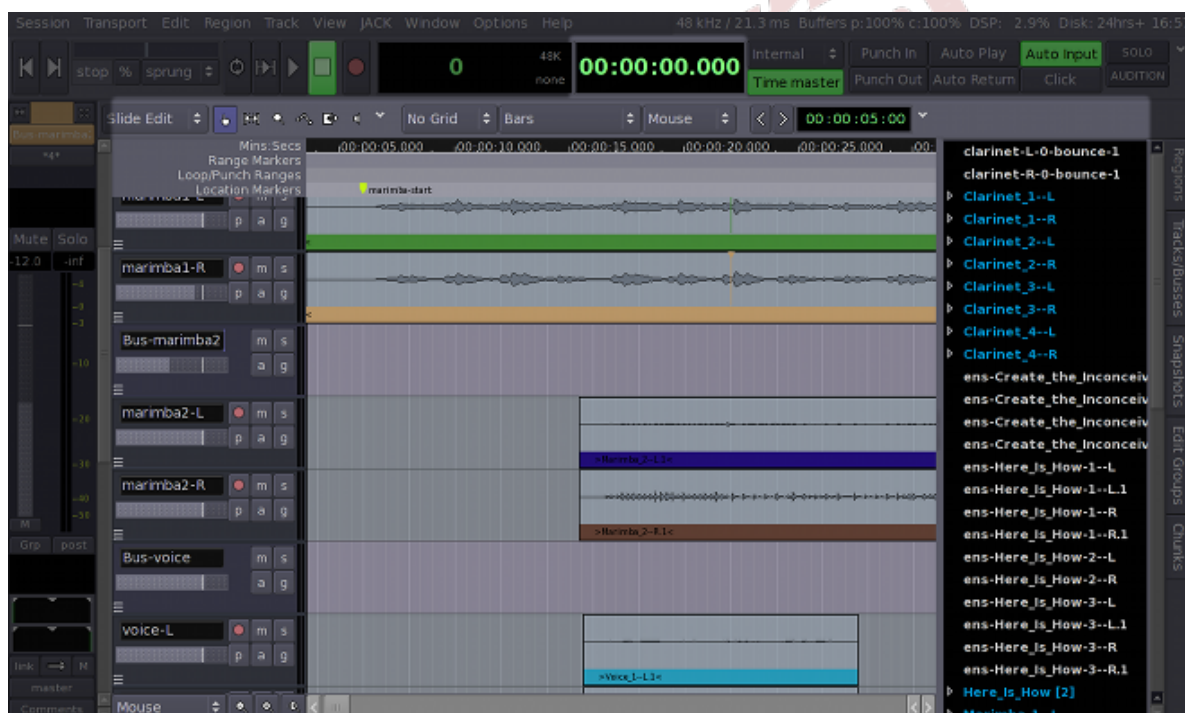


Figure 6.1. A session in Ardour

6.3.2. Track and Multitrack

A *track* represents one channel, or a predetermined collection of simultaneous, inseparable channels (as is often the case with stereo audio). In the DAW's main window, tracks are usually represented as rows, whereas time is represented by columns. A track may hold multiple regions, but usually only one of those regions can be heard at a time. The *multitrack* capability of modern software-based DAWs is one of the reasons for their success. Although each individual track can play only one region at a time, the use of multiple tracks allows the DAW's outputted audio to contain a virtually unlimited number of simultaneous regions. The most powerful aspect of this is that audio does not have to be recorded simultaneously in order to be played back simultaneously; you could sing a duet with yourself, for example.

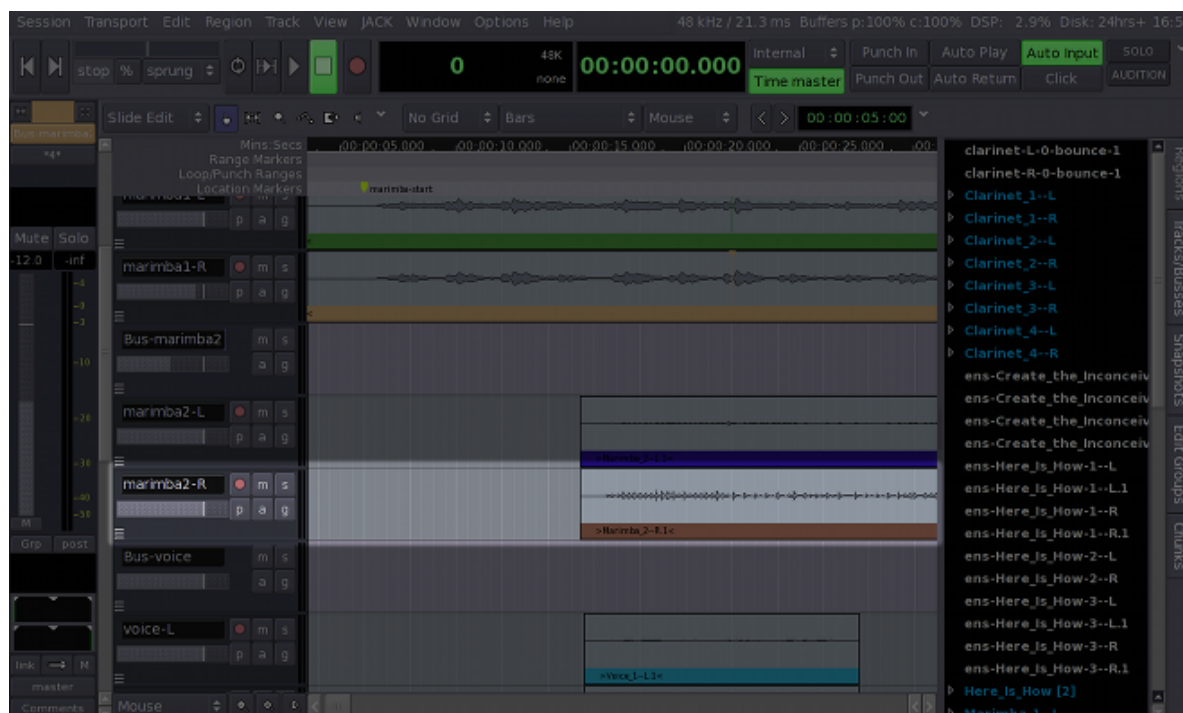


Figure 6.2. A track in Ardour

6.3.3. Region, Clip, or Segment

Region, clip, and segment are synonyms: different software uses a different word to refer to the same thing. A *xmultitrack* (or *clip* or *segment*) is the portion of audio recorded into one track during one take. Regions are represented in the main DAW interface window as a rectangle, usually coloured, and always contained in only one track. Regions containing audio signal data usually display a spectrographic representation of that data. Regions containing MIDI signal data usually displayed as matrix-based representation of that data.

For the three DAW applications in the Musicians' Guide:

- **Ardour** calls them "regions,"
- **Qtractor** calls them "clips," and,
- **Rosegarden** calls them "segments."

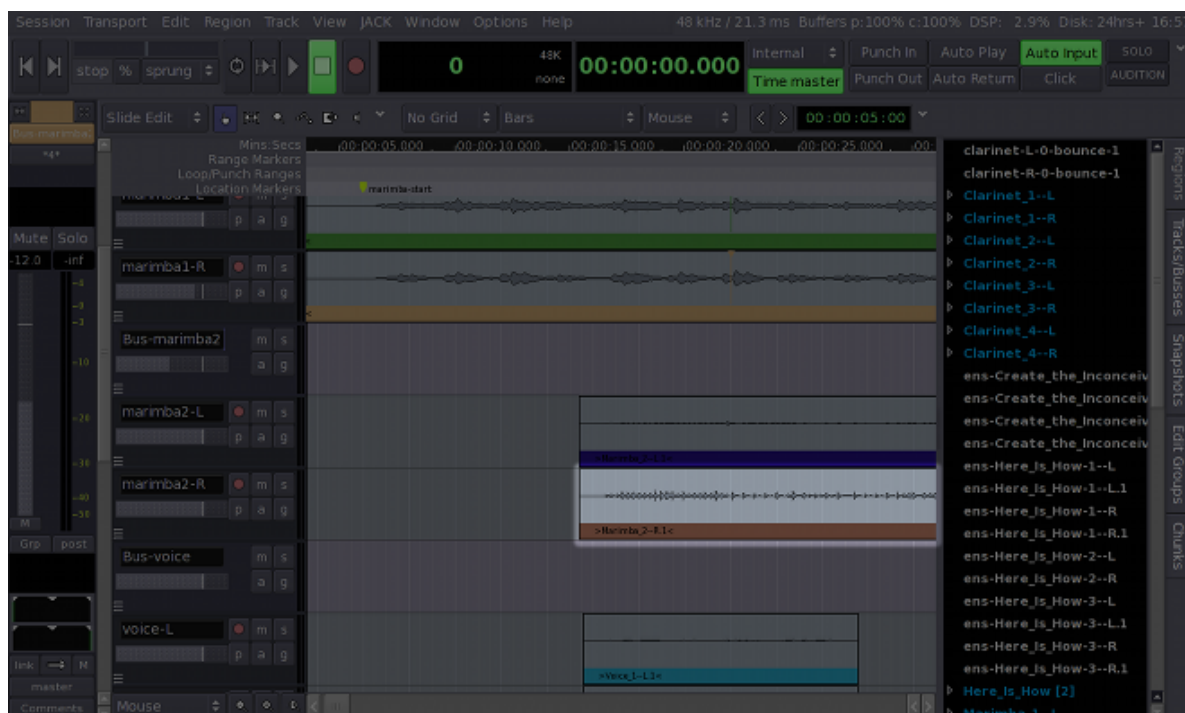


Figure 6.3. A region in Ardour

6.3.4. Transport and Playhead

The *transport* is responsible for managing the current time in a session, and with it the playhead. The *playhead* marks the point on the timeline from where audio would be played, or to where audio would be recorded. The transport controls the playhead, and whether it is set for recording or only playback. The transport can move the playhead forward or backward, in slow motion, fast motion, or real time. In most computer-based DAWs, the playhead can also be moved with the cursor. The playhead is represented on the DAW interface as a vertical line through all tracks. The transport's buttons and displays are usually located in a toolbar at the top of the DAW window, but some people prefer to have the transport controls detached from the main interface, and this is how they appear by default in Rosegarden.

6.3.5. Automation

Automation of the DAW sounds like it might be an advanced topic, or something used to replace decisions made by a human. This is absolutely not the case - *automation* allows the user to automatically make the same adjustments every time a session is played. This is superior to manual-only control because it allows very precise, gradual, and consistent adjustments, because it relieves you of having to remember the adjustments, and because it allows many more adjustments to be made simultaneously than you could make manually. The reality is that automation allows super-human control of a session. Most settings can be adjusted by means of automation; the most common are the fader and the panner.

The most common method of automating a setting is with a two-dimensional graph called an *envelope*, which is drawn on top of an audio track, or underneath it in an *automation track*. The user adds adjustment points by adding and moving points on the graph. This method allows for complex, gradual changes of the setting, as well as simple, one-time changes. Automation is often controlled by means of MIDI signals, for both audio and MIDI tracks. This allows for external devices to adjust settings in the DAW, and vice-versa - you can actually automate your own hardware from within a software-based DAW! Of course, not all hardware supports this, so refer to your device's user manual.

6.4. User Interface

This section describes various components of software-based DAW interfaces. Although the **Qtractor** application is visible in the images, both **Ardour** and **Rosegarden** (along with most other DAW software) have an interface that differs only in details, such as which buttons are located where.

6.4.1. Messages Pane

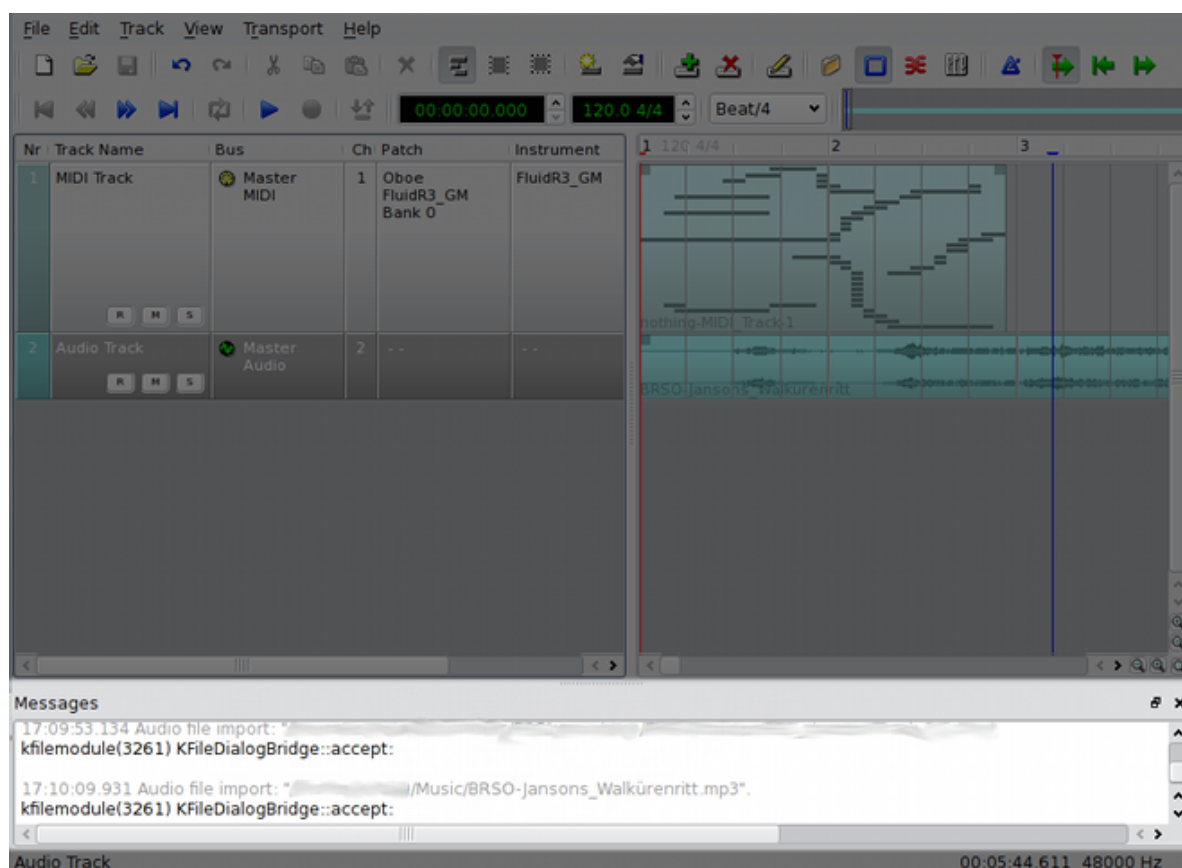


Figure 6.4. The Qtractor messages pane

The **messages** pane, shown in [Figure 6.4, “The Qtractor messages pane”](#), contains messages produced by the DAW, and sometimes messages produced by software used by the DAW, such as JACK. If an error occurs, or if the DAW does not perform as expected, you should check the **messages** pane for information that may help you to get the desired results. The **messages** pane can also be used to determine whether JACK and the DAW were started successfully, with the options you prefer.

6.4.2. Clock

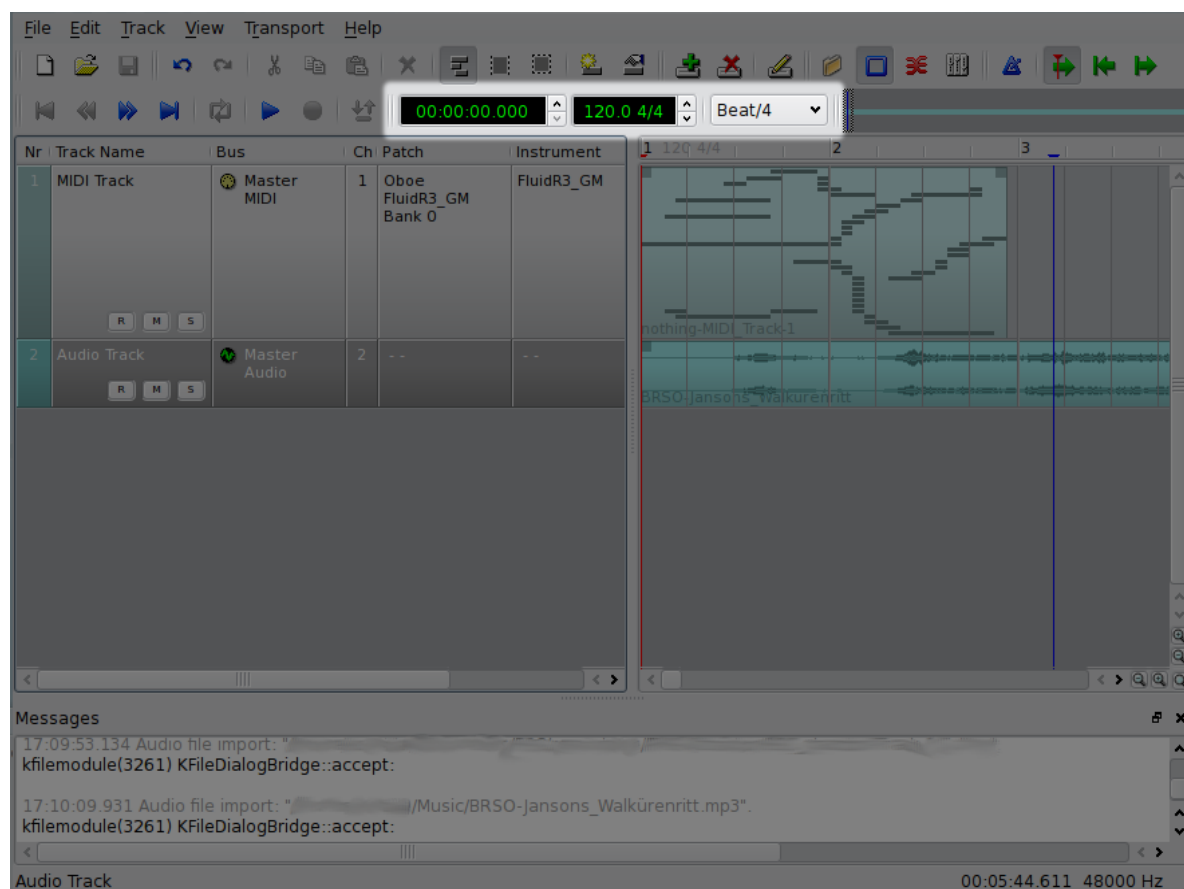
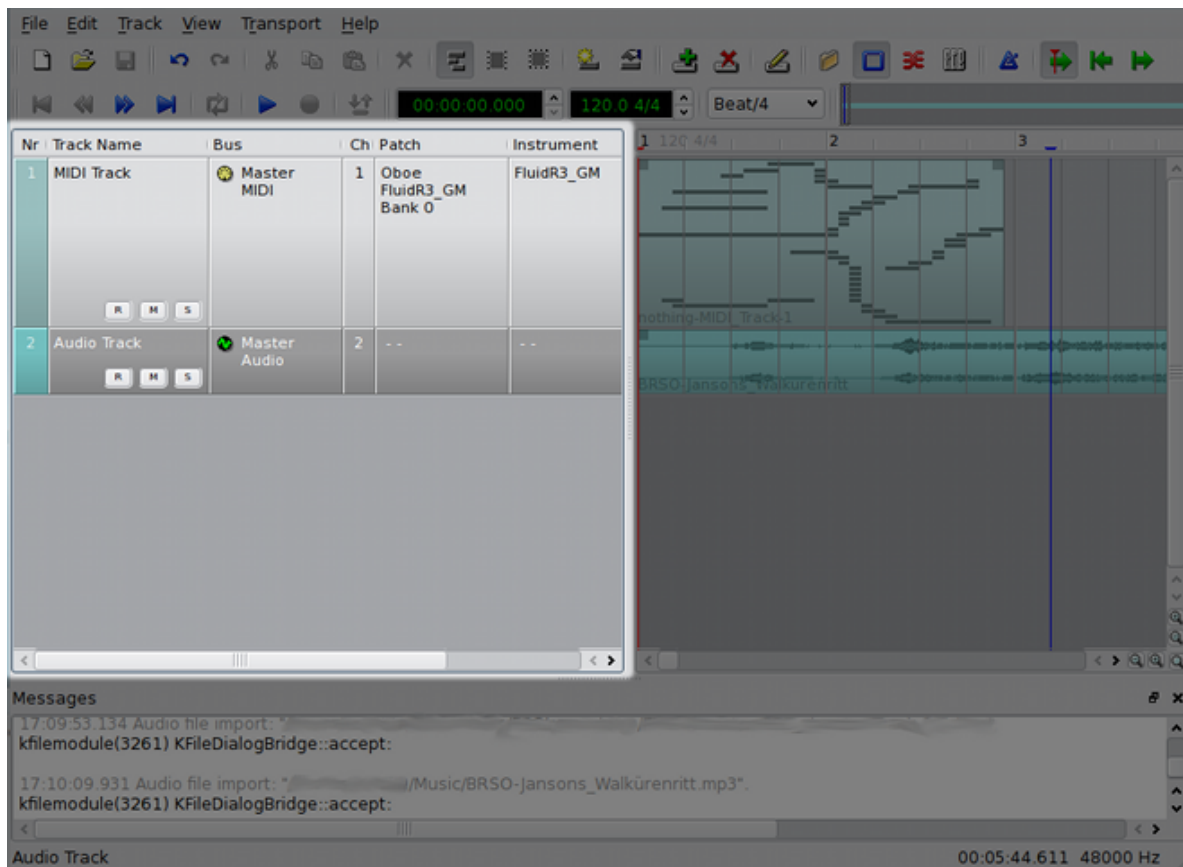


Figure 6.5. The Qtractor clock

The clock shows the current place in the file, as indicated by the transport. In [Figure 6.5, “The Qtractor clock”](#), you can see that the transport is at the beginning of the session, so the clock indicates **0**. This clock is configured to show time in minutes and seconds, so it is a *time clock*. Other possible settings for clocks are to show *BBT* (bars, beats, and ticks — a *MIDI clock*), samples (a *sample clock*), or an *SMPTE timecode* (used for high-precision synchronization, usually with video — a *timecode clock*). Some DAWs allow the use of multiple clocks simultaneously.

Note that this particular time clock in **Qtractor** also offers information about the MIDI tempo and metre (120.0 beats per minute, and 4/4 metre), along with a quantization setting for MIDI recording.

6.4.3. Track Info Pane



The track info pane: note a separate track info space for each of the two tracks that appear in the pane to the right.

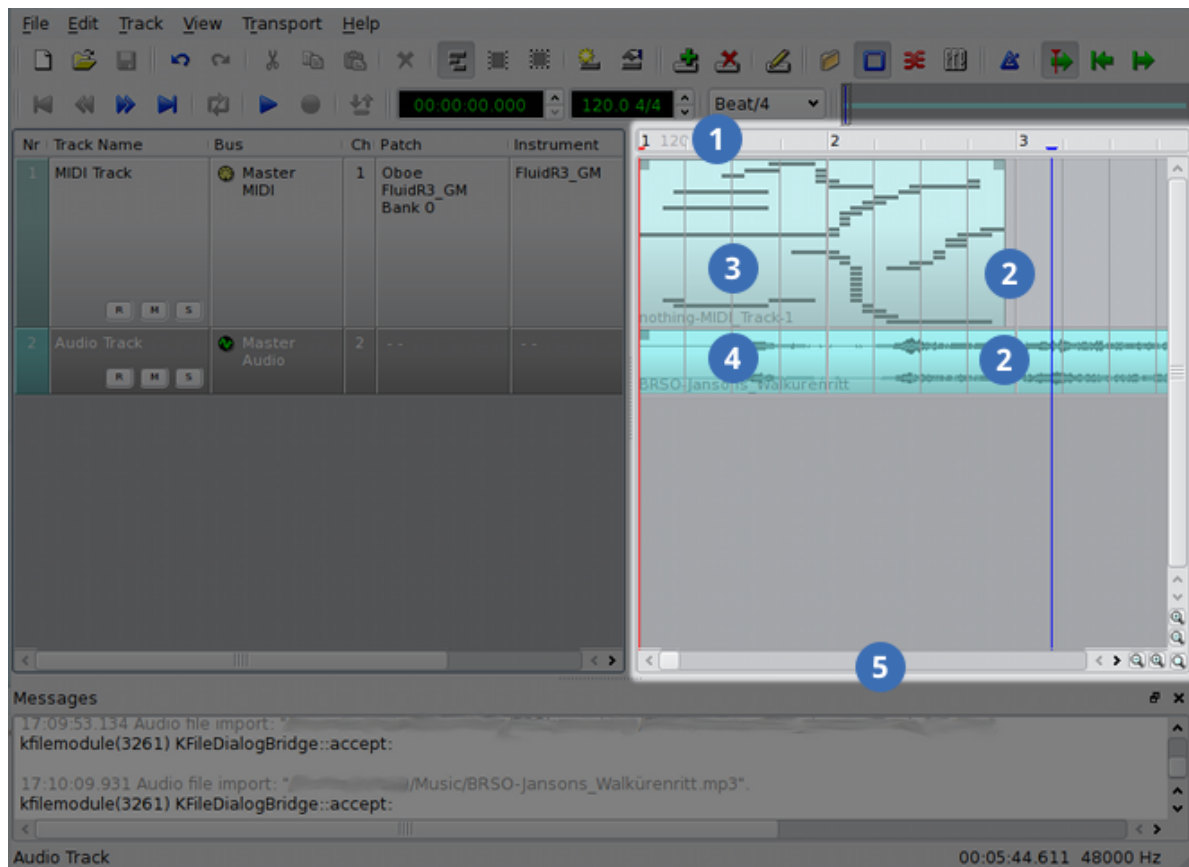
Figure 6.6. The Qtractor track info pane

The *track info* pane contains information and settings for each track and bus in the session. Here, you can usually adjust settings like the routing of a track's or bus' input and output routing, the instrument, bank, program, and channel of MIDI tracks, and the three buttons shown in [Figure 6.6, "The Qtractor track info pane"](#): **R** for "arm to record," **M** for "mute/silence track's output," and **S** for "solo mode," where only the selected tracks and busses are heard.

The information provided, and the layout of buttons, can change dramatically between DAWs, but they all offer the same basic functionality. Often, right-clicking on a track info box will give access to extended configuration options. Left-clicking on a portion of the track info box that is not a button allows you to select a track without selecting a particular moment in *track pane*.

The track info pane does not scroll out of view as the track pane is adjusted, but is independent.

6.4.4. Track Pane



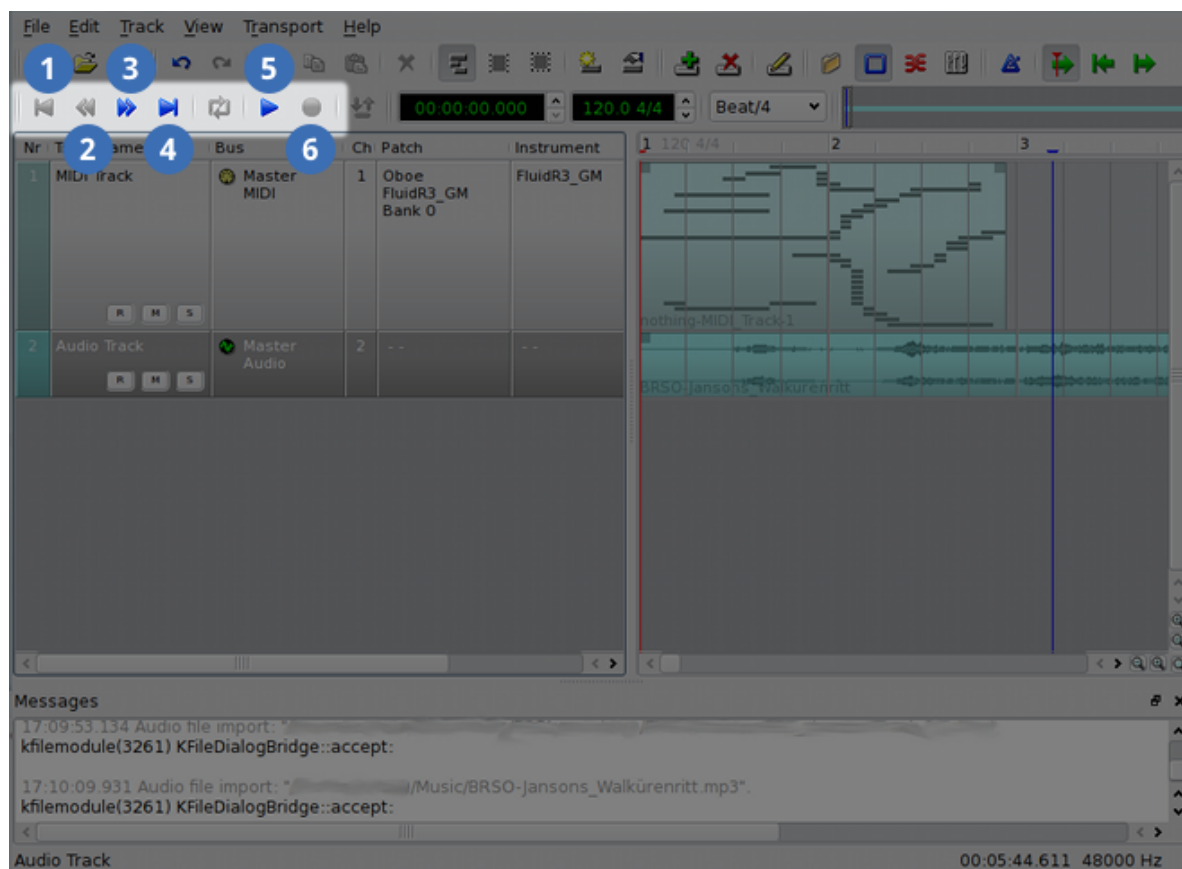
1. The *ruler*, set here to *BBT* (bars, beats, ticks).
2. Two tracks, presented as graphical representations. The horizontal axis represents time.
3. A MIDI *clip* (known as a *region* in other applications)
4. An audio *clip* (known as a *region* in other applications)
5. Scrollbar for the track pane; note that this pane scrolls independently.

Figure 6.7. The Qtractor track pane

The *track pane* is the main workspace in a DAW. It shows *regions* (also called *clips*) with a rough overview of the audio wave-form or MIDI notes, allows you to adjust the starting-time and length of regions, and also allows you to assign or re-assign a region to a track. The track pane shows the transport as a vertical line; in [Figure 6.7, “The Qtractor track pane”](#) it is the left-most red line in the track pane.

Scrolling the track pane horizontally allows you to view the regions throughout the session. The left-most point is the start of the session; the right-most point is after the end of the session. Most DAWs allow you to scroll well beyond the end of the session. Scrolling vertically in the track pane allows you to view the regions and tracks in a particular time range.

6.4.5. Transport Controls



1. Transport skip to beginning
2. Transport fast reverse
3. Transport fast forward
4. Transport skip to end
5. Transport forward at real time
6. Arm for recording

Figure 6.8. The Qtractor transport controls

The transport controls allow you to manipulate the transport in various ways. The shape of the buttons is somewhat standardized; a similar-looking button will usually perform the same function in all DAWs, as well as in consumer electronic devices like CD players and DVD players.

The single, left-pointing arrow with a vertical line will move the transport to the start of the session, without playing or recording any material. In **Qtractor**, if there is a blue place-marker between the transport and the start of the session, the transport will skip to the blue place-marker. Press the button again to the next blue place-marker or the beginning of the session.

The double left-pointing arrows move the transport in fast motion, towards the start of the session. The double right-pointing arrows move the transport in fast motion, towards the end of the session.

The single, right-pointing arrow with a vertical line will move the transport to the end of the last region currently in a session. In **Qtractor**, if there is a blue place-marker between the transport and the end of

the last region in the session, the transport will skip to the blue place-marker. Press the button again to skip to the next blue place-marker or the end of the last region in the session.

The single, right-pointing arrow is commonly called "play," but it actually moves the transport forward in real-time. When it does this, if the transport is armed for recording, any armed tracks will record. Whether or not the transport is armed, pressing the "play" button causes all un-armed tracks to play all existing regions.

The circular button arms the transport for recording. It is conventionally red in colour. In **Qtractor**, the transport can only be armed *after* at least one track has been armed; to show this, the transport's arm button only turns red if a track is armed.



DRAFT

Ardour

Ardour is a feature-rich application designed for multi-track recording situations.

7.1. Requirements and Installation

7.1.1. Knowledge Requirements

The **Ardour** user interface is similar to other DAWs. We recommend that you read [Section 6.4, “User Interface”](#) if you have not used a DAW before.

7.1.2. Software Requirements

Ardour uses the JACK Audio Connection Kit. You should install JACK before installing **Ardour**. Follow the instructions in [Section 2.3.1, “Installing and Configuring JACK”](#) to install JACK.

7.1.3. Hardware Requirements

You need an audio interface to use **Ardour**. If you will record audio with **Ardour**, you must have at least one microphone connected to your audio interface. You do not need a microphone to record audio signals from other JACK-aware programs like **FluidSynth** and **SuperCollider**.

7.1.4. Installation

Use **PackageKit** or **KPackageKit** to install the *ardour* package. Other required software is installed automatically.

7.2. Recording a Session

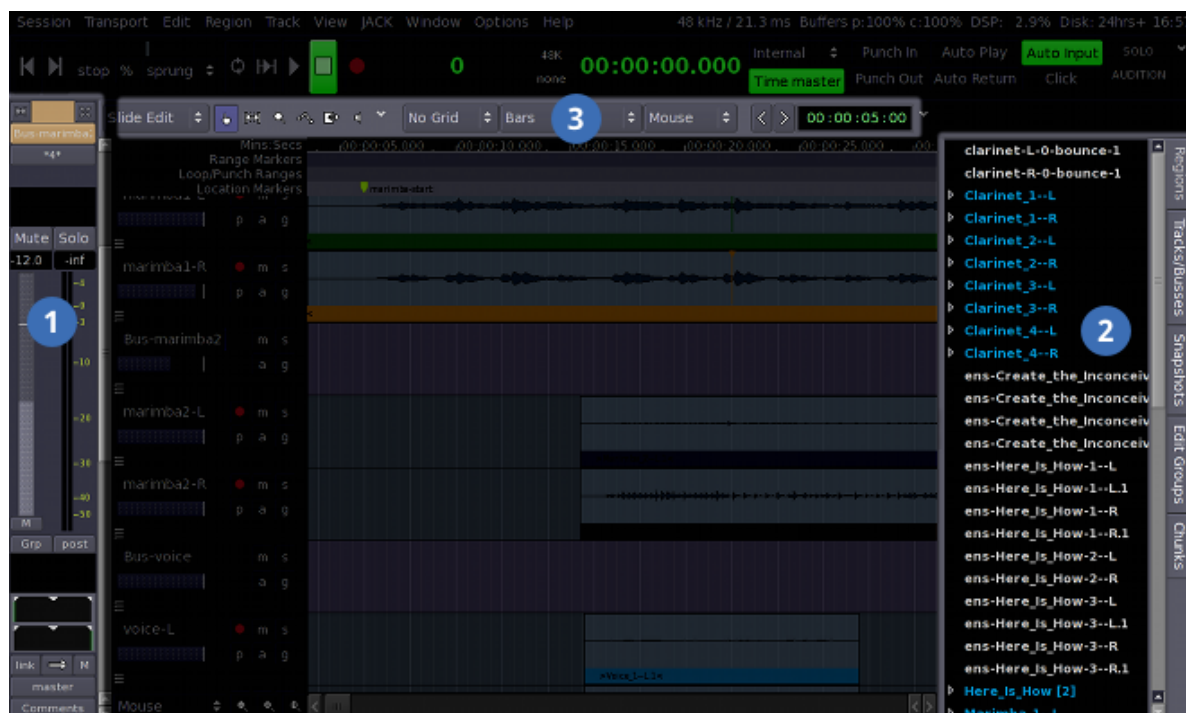
Recording a session usually happens all at once, but sometimes recording can happen over several days or even weeks. Mixing and mastering happen after a session has been recorded. Remember that JACK must have the same sample rate and sample format settings each time you open a session.

7.2.1. Running Ardour

1. **Ardour** uses the JACK sound server. Use **QjackCtl** to start JACK before **Ardour**, or **Ardour** starts JACK for you.
2. **Ardour** asks you to choose a location to save your new session. **Ardour** automatically creates a directory to store the session's files. You can also open an existing session.

7.2.2. The Interface

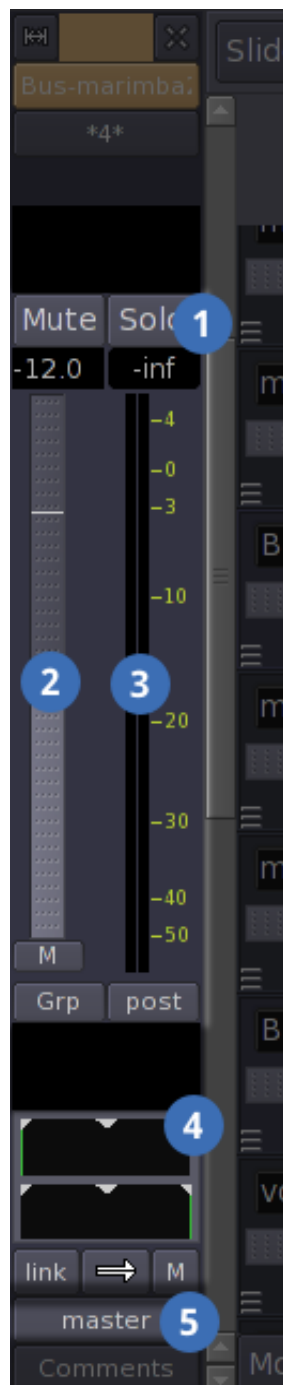
This section explains some of the graphical interface components that are unique to **Ardour**. Components that are consistent through most DAWs are explained in [Section 6.3, “Interface Vocabulary”](#).



1. The editor mixer
2. The session sidebar
3. The main toolbar

Figure 7.1. The Ardour interface

Figure 7.1, “The Ardour interface” illustrates three graphical interface components specific to the Ardour interface: the *editor mixer*, the *session sidebar*, and the main toolbar.

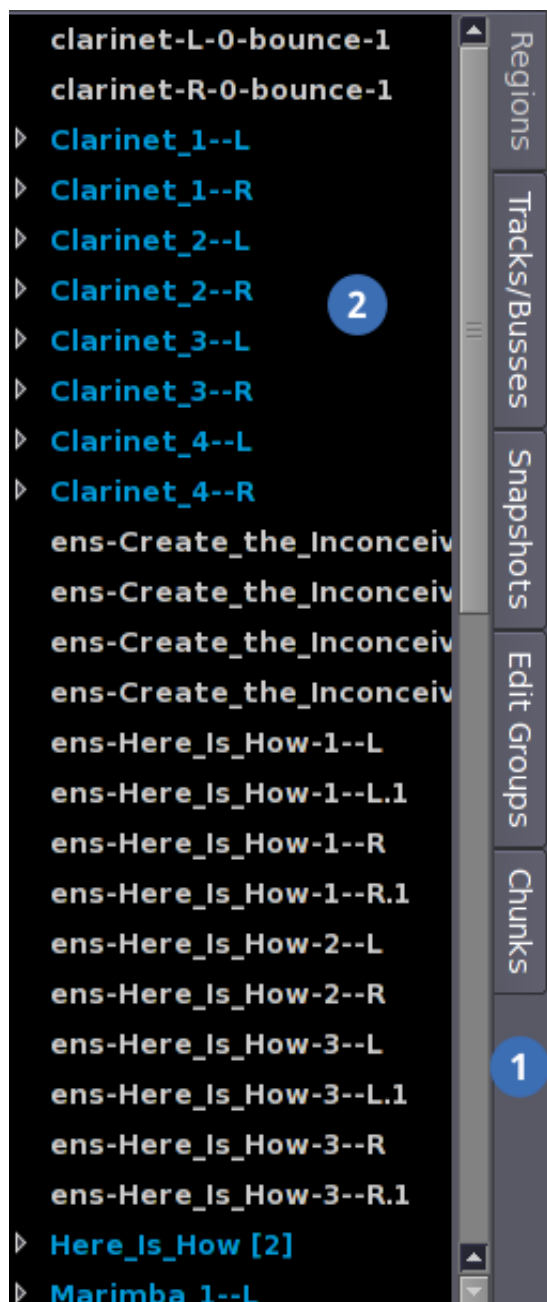


1. The fader
2. The fader control
3. The fader level meter
4. The panner
5. The output connection button

Figure 7.2. The Ardour editor mixer

Figure 7.2, “The Ardour editor mixer” shows the editor mixer, located at the left of the main **Ardour** window. The editor mixer shows only one mixer strip at a time. It shows the fader and its controls,

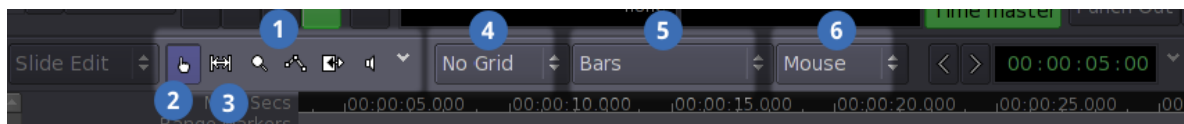
in the middle of the mixer strip, the panner and its controls, at the bottom of the mixer strip, and the **Comments** and outgoing connections buttons.



1. The tab strip
2. The region list

Figure 7.3. The Ardour session sidebar

Figure 7.3, “The Ardour session sidebar” shows the session sidebar, located at the right the main **Ardour** window. In this image, the **Regions** tab is selected, so the sidebar shows a list of regions currently in the session. You can see blue ones which were directly imported, white ones which were created from blue regions, and the arrows to the left of some blue regions, indicating that there are white-coloured sub-regions associated with those blue regions.



1. Tool selection buttons
2. The **select/edit object** button
3. The **select/edit range** button
4. The snap mode
5. The grid mode
6. The edit point

Figure 7.4. The main Ardour toolbar

Figure 7.4, “The main Ardour toolbar” shows the main toolbar, located underneath the transport controls, and above the timeline and its rulers. In the middle of the toolbar are three unlabeled, but highly useful multiple-choice menus: the **snap mode** menu (currently set to **No Grid**); the **grid mode** menu (currently set to **Bars**); and then **edit point** menu (currently set to **Mouse**). To the left of these menus are the tool-selection buttons, the most important of which are the two left-most buttons: **select/edit object**, and **select/edit range**.

7.2.3. Setting up the Timeline

At the top of the main **Ardour** window, to the right of the transport's toolbar, are two relatively large clocks. Right-click the clocks to choose what you want them to display:

- **Bars:Beats** displays the number of bars and beats
- **Minutes:Seconds** displays the time since beginning of track
- **Timecode** displays frames-per-second timecode (usually for work with films)
- **Samples** displays the samples since start

If you do not need both clocks, you can turn one of them off.

The **snap mode** menu is located between the timeline and the clocks. This menu controls where regions may move. You need to change these as you work with a session, depending on the current activity. The left menu contains:

- **No Grid**: regions can move freely
- **Grid**: regions must start on the nearest grid point
- **Magnetic** regions can move freely, but when they are near a grid point, they automatically snap to it

The middle menu controls where to place the grid lines; by timecode, by clock time, by beats and bars, or by regions.

The timeline (which contains many *rulers* showing different time-marking scales) is located at the top of the canvas area, underneath the toolbars. Use the right-click menu to select which rulers you want to display. The rulers you should choose depends on the clock settings and the snap mode.

7.2.4. Connecting Audio Sources to Ardour

The name of the track onto which you want to record should be the name of the input in JACK.

7.2.5. Setting up the Busses and Tracks

Refer to [Section 1.4.2, “Busses, Master Bus, and Sub-Master Bus”](#) for a general discussion of busses. By default, everything that you export from **Ardour** is sent to a master bus. Busses do not contain regions but function as a batch collecting zone, where you can subject the whole project to a particular filter or volume adjustment.

Procedure 7.1. Add a track for recording

1. Click **Track** → **Add Track/Bus**
2. ensure that **Tracks** is selected
3. set the number (probably **1**)
4. select the number of input channels (probably **Stereo**, meaning 2)
5. select the mode:
 - **Normal**: creates a new Region for each recording take
 - **Tape**: destructively records over whatever is already recorded (like a tape)
6. Click **Add** to create the track

Procedure 7.2. Rename tracks, to identify them

1. Click the existing track name in the label to the far left of the track area
2. Type over the existing name
3. press **Enter**

7.2.6. Adjusting Recording Level (Volume)

It is important to properly set the level of the inputs before recording.

The nature of audio equipment is such that it can only perceive sound pressures (perceived as volume) within a certain range. If a sound is too quiet, it will not be perceived, and if it is too loud, it will not be perceived accurately. Furthermore, and this is most important when thinking about your own ears — if a sound is far too loud, it may permanently damage the audio instrument.

The nature of digital audio is such that there is a distinct number of volume levels at which something can be recorded. If a sound is either below or above that range, then it will not be correctly recorded. When such an improperly-recorded sound is played back, whether too quiet or too loud, humans will usually perceive it as “nothing but noise.”

When **Ardour** records silence, it behaves no differently from when there is no input at all. When **Ardour** calculates that a portion of audio is too loud and therefore distorted, it outlines the wave-form representation in red, as shown in [Figure 7.5, “Audio that is too loud”](#).

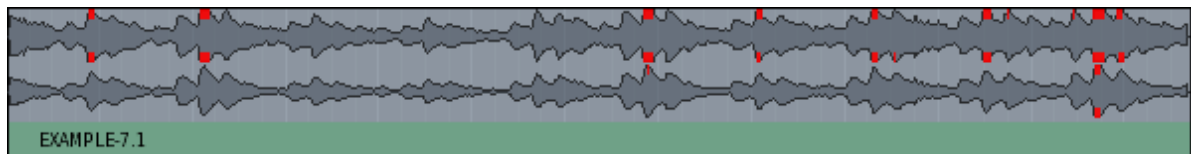


Figure 7.5. Audio that is too loud

There are three simple strategies that can be used to change the input level of an audio signal:

1. Move the microphone closer or farther from the source
2. Route the microphone through a mixer before it reaches the audio interface
3. Route the audio through a bus in **Ardour** before it gets recorded

Here are the advantages and disadvantages of each approach:

There are some circumstances where it is either impractical, impossible, or not advisable to move the microphone or route it through a hardware mixer. In these cases, you can use a bus in **Ardour** to modify the volume of the input signal before it is recorded.

1. Choose **Track** → **Add Track/Bus**.
2. Select "busses" in the window that pops up.
3. Choose the number of busses that you wish to add. You need one for every track that you are recording, and of which you want to adjust the volume. It is also possible to record at several different volumes.
4. Set the number of channels that you want in the bus.
5. Once you have the new bus, change its name by doing whatever. I suggest naming it something that makes it obvious you are using the bus for recording, rather than exporting, like "REC-Bus."
6. **Ardour** automatically sets up busses to be used with audio being outputted. Furthermore, the volume/level control only works on audio being outputted from a track or bus. This is why you cannot use the track's volume/level control to adjust the input volume for that track.
7. Use **QjackCtl** to reconnect like this (for help, refer to [Section 7.2.9, "Routing Audio and Managing JACK Connections"](#)):
 - a. Disconnect all of the connections to/from the bus you want to use for recording ("recording bus").
 - b. Ensure that nothing is connected to the input of the track onto which you want to record ("recording track").
 - c. Connect the microphone (the input source) to the recording bus' input
 - d. Connect the input bus' output to the recording track's input.
 - e. Ensure that the recording track's output is connected to the "master" input (this is the master output bus, which should be present in all projects, and through which all output audio should be routed).

Remember: only one track-to-be-recorded can be routed through a bus for this purpose, because a bus can only output one stream of audio.

Here is an algorithm to test whether your tracks are set at a good recording volume. This should be done before arming any tracks for recording. Unfortunately, you can never know that you have chosen

the best input level until after a region is recorded. It takes both instinct and experience to be able to choose good input levels reliably.

1. Set up all microphones as required.
2. Set up connections in JACK as required.
3. Set up any recording busses as required (see above).
4. On the audio tracks being recorded, set the "metering point" to "input" (here's how to do that).
5. Ask the performers to demonstrate the loudest passages they will be doing in the session. Adjust the input level so that the maximum level falls between -3 dB and -6 dB (by looking here). You can reset the maximum-level-seer by clicking on it.
6. Ask the performers to demonstrate the quietest passages they will be performing in the session. Adjust the input level so that this does not fall below -40 dB; it should probably be between -30 dB and -20 dB.
7. Ask the performers to demonstrate an average passage from what they will be performing in the session. This is usually less important than the previous two checks, but if most of the performance will be quieter, it may be worth risking a higher input level in order to capture more detail. Nevertheless, a "moderate" volume level should result in an input level reading of -20 dB to -10 dB.
8. When you are more experienced both with the kind of group you are recording, and the software and equipment being used to do it, you may not need to do these level-checks every time. It's better to be safe than sorry, however, because once a musical moment has passed, it is impossible to re-create.

7.2.7. Recording a Region

As you progressively record a session, you will create at least one region. Warning about audio being put out the "audition" output by default (use headphones)

1. Ensure that the inputs, timeline, and tracks are properly set up.
2. if there is nothing to the left of the editor window, press Ctrl+E or 'View > Show Editor Mixer'
3. Select the track you're recording onto
4. set the metering point to "input" and verify that it's working correctly and connected to the right thing (say what this does, and why you want to do it now)
5. See "Adjusting Recording Volume" below, and do it now
6. Arm the track for recording: either press "Record" in the track's mixer in the left, or press the small red record button on the track itself
7. the buttons will remain lighted to show that the tracks are armed
8. arm **Ardour** for recording by select the big red record button on the transport
9. start the transport in the normal way (big play button)
10. when you're done recording, stop the transport with the big stop button
11. each time you start and stop the transport, a new "region" is produced

12. each time you stop the transport, **Ardour** "un-arms" itself, but any tracks that you selected are still armed
13. When you've finished recording a region, use the "Regions" box-thing on the right of the interface to rename the region:
 - a. Find the region that you just recorded (by default they are named like "Audio 1-1" which is the name of the recording track followed by a hyphen, then a number in ascending sequence representing the "take"). Select it.
 - b. Click on the title, and a box should surround it.
 - c. Change the name to what you want.
 - d. Press enter to finish editing the name.

7.2.8. Recording More

After you have recorded one region, you will probably not have everything that you want. There are many ways to continue recording, depending on what still remains to be recorded.

7.2.8.1. To Continue the Same Session

This is what you'll want to do if, for example, you were recording a session and decided to take a ten-minute break. It will work for any situation where you want to continue a session that already started recording.

1. move the transport to somewhere after what you've already capture. You can do this either by using the forward/reverse and play/stop buttons on the transport, or by finding the point in the timeline where you want the transport to be, and then left-clicking somewhere in the time-line.
2. Verify that the connections and levels are still set correctly.
3. Verify that the recording tracks are still armed.
4. Arm the transport.
5. Start the transport when ready to record.

7.2.8.2. To Capture an Additional Part of Something That Is already Recorded

A technique often used for studio recordings is to separately record parts that would normally be played together, and which will later be made to sound together (see the "Preparing a Session" section, below). For example, consider a recording where one trumpeter wants to record both parts of a solo written for two trumpets. The orchestra could be brought into the studio, and would play the entire solo piece without any trumpet solo. **Ardour** will record this on one track. Then, the trumpet soloist goes to the studio, and uses **Ardour** to simultaneously listen to the previously-recorded orchestra track while playing one of the solo trumpet parts, which is recorded onto another track. The next day, the trumpeter returns to the studio, and uses **Ardour** to listen to the previously-recorded orchestra track and previously-recorded solo trumpet part while playing the other solo trumpet part, which is recorded onto a third track. The recording engineer uses Audacity's mixing and editing features to make it sound as though the trumpeter played both solo parts at the same time, while the orchestra was there.

Coordinating the timing of musicians across tracks recorded separately is difficult. A "click track" is a track with a consistent clicking noise at the desired tempo. Click tracks are played through

headphones to the musicians being recorded, or to a musician who leads the others. Click tracks are not included in the final mix.

To do this:

1. Record the first part. The order in which to record parts is up to the recording engineer (that means you). It will probably be easier to record whoever plays the most, or whoever plays the most rhythmically consistent part, before the others.
2. Add the track/s onto which you will next record.
3. Set up the connections for the new track.
4. Do a level check to ensure that the new track is neither too loud nor soft.
5. Set the transport to the beginning of the passage where you want to begin recording the next track. You do not need to set up the start of the track very precisely, since you can change that later. You will need to make sure that the next player has enough time after the transport is started to hear where they are supposed to enter, and at what tempo.
6. You will need to set up some way for the performers (or somebody conducting/leading them) to hear the already-recorded material. It is probably best to do this with headphones.
7. Arm the tracks that you want to record. Make sure that already-recorded tracks are no longer armed, especially if they are in "tape mode."
8. Arm the transport.
9. When you are ready to record, start the transport rolling.

7.2.8.3. To Capture a Better Recording of Something That Is already Recorded

If you have already recorded all or most of a session, you can re-record *part* of the session in order to "fix up" any issues. **Ardour** allows you to record onto the pre-existing tracks, keeping the first take, putting the newly-recorded region over it. Later, you will get to choose the exact points at which the outputted recording is to switch between regions/takes.

1. Record the session.
2. Ensure that you have the connections and levels set as they were during the first time you recorded the regions over which you're recording now.
3. You will need to set the transport location. Choose a place that is before the segment which you want to replace. The performers should probably also start playing before the section to be replaced, so you will need to start well enough in advance that they can pick up the tempo, get in the groove, and then start playing *all before* the part that needs replacement.
4. Click in the time-line to move the transport.
5. Ensure that the correct tracks are armed.
6. Arm the transport.
7. Start the transport and record the revised section of music.

At some point, you will have recorded everything that you need, and you will want to progress to mixing and editing.

7.2.9. Routing Audio and Managing JACK Connections

Ardour automatically saves the state of JACK connections when it saves a session.

Ardour offers the following output ports, assuming a stereo (two-channel) setup:

- two channels per track, called "track_name/out 1" and "track_name/out 2". These will usually be connected to the master bus, or to a sub-mixing bus, when you are using one.
- two channels per bus, called "bus_name/out 1" and "bus_name/out 2". These will usually be connected to the master bus, unless you are using two levels of sub-mixing busses.
- two channels for the auditioner, called "auditioner/out 1", which represents the channels used to audition a region; when you want to import it, for example, or in the "Regions" box on the right-side, when you select one and right-click and choose "Audition". These should not be connected to the master bus, but to an output device that you want to use when auditioning regions.
- two channels for the click-track, called "click/out 1", which represents the channels used to play the click-track when recording. These should not be connected to the master bus, but to an output device that you want to use for the click-track.
- two channels for the master bus, called "master/out 1", which represents the output used by the master output bus. These should be connected to an output device that you wish to use for listening to the session when the transport is moving.

Ardour offers the following input ports, for a stereo (two-channel) setup:

- two channels per track, called "track_name/in 1" and "track_name/in 2". These should both be connected to the same input device. If you are using a recording bus, then these should be connected to that bus.
- two channels per bus, called "bus_name/in 1" and "bus_name/in 2". These should be connected to whatever channels you want to be mixed into them. If you are using it as a recording bus, then these should be connected to the same input device.
- two channels for the master bus, called "master/in 1", which represents the input used for the master bus. These should be connected to all of the tracks. If you are using sub-bus mixing, then all of the tracks should connect to the master bus' input either directly or through a sub-bus.

In most setups, **Ardour** automatically sets the channel connections correctly. There are ways to change the connections from within **Ardour**, but they offer limited flexibility. For this reason, it is recommended that users use **QjackCtl** to monitor connections, since through **QjackCtl** it is also possible to monitor and change many other features of JACK.

Learning to make the right connections is a valuable trick for people using **Ardour**. The fact that **Ardour** uses JACK for both its internal and external connections allows tricks such as the earlier-mentioned recording bus (which adjusts the input level of a source), flipping the left and right audio channels, and creating a multi-channel audio output by combining many input channels. Undoubtedly, other tricks exist.

7.2.10. Importing Existing Audio

When you record audio, **Ardour** automatically save it to disk and adds a representation of that file in the program as a "region." You can also use pre-existing audio files as regions, which can then be added to any track.

To import an existing audio file:

1. Whip out the "regions" part of the panel on the right-hand side of the interface

2. Right-click anywhere in the box
3. Select "Import to Region List"
4. The "Add existing audio" window will be opened
5. You can use three different tabs to select an audio file to add:
 - "Browse Files" (does this) (covered here)
 - "Search Tags" (does this)
 - "Search Freesound" (does this)
6. Using "Browse Files," navigate to a sound that you want to add. Although certain other file formats are supported (like FLAC), it is probably best to add WAV or AIFF files.
7. Certain information about the audio file will be displayed on the right-hand side of the window. This portion of the window also allows you to "audition" the file before importing it (that is, you can hear it by using the "Play" and "Stop" buttons in the window, without affecting your current project.
8. If the file that you selected has a sample-rate that is not the same as that of the current project, then the sample-rate will be highlighted in red. You can choose to import it anyway, in which case **Ardour** warns you again. If you import a file in a different sample rate than that of the current project, it will be played back in the project's sample rate. This will result in incorrect speed and pitch.
9. There are a number of other options, displayed along the bottom of the window.
10. You can choose to add files:
 - "as new tracks," which puts each file in its own track, set to normal mode, then adds it to the region list
 - "as new tape tracks," which puts each file in its own track, set to tape mode, then adds it to the region list
 - "to region list," which puts each file in the region list, but does not automatically put it in any tracks.
 - Note that when you choose to automatically create new tracks, **Ardour** adds the region to the new track, with the region starting at the current location of the transport.
11. The other options in this list are self-explanatory. It is usually best to convert using the best quality, since quality can always be reduced later (which saves space).
12. If you chose not to automatically create tracks, then you will need to add the imported regions into a track before they will be played in your session. You can do this easily by selecting the region from the "Regions" box on the right, and dragging it to a track.

7.3. Files for the Tutorial

These tutorial files represent the material required to create a finished version of a song called "Here Is How," written by Esther Wheaton. The song was released as part of her first album, "Not Legendary," and she has released the source files for this song under !!!!! this licence (probably CC-BY-SA) !!!!! For more information on the artist, please refer to her *Esther Wheaton's MySpace Page*, available at <http://www.myspace.com/estherwheaton>.

The material presented for your use is a folder containing an **Ardour** file and the associated audio files required to start the tutorial. The tutorial itself comprises the following sections about editing, mixing, and mastering (or exporting). The program used to record the audio files split the left and right channels into separate files, so they are imported into **Ardour** as separate regions. Therefore, the setup is more complex than it would be if the song were originally recorded in **Ardour**, but this gives the opportunity to learn in greater detail about busses, creating and using the stereo image, and volume level adjustments.

The unique setup also means that none of the audio regions are in the right place on the timeline, and most of them require extensive editing. This would be bad if the objective of the tutorial were to create a finished version of the song as quickly as possible; but the objective is to learn how to use **Ardour**, and this is almost guaranteed.

!!EL!! Links to the files !!!!! I don't know where to put them!

7.4. Editing a Song (Tutorial)

This section covers the basics of preparing "Here Is How." The focus is on trimming the regions and placing them in the right position on the timeline. Since the goal is to replicate the form of the original song, there is little room for artistic freedom.

To get the most out of this section, you should use the tutorial files provided above. By following the instructions with the tutorial file, you will be able to use real editing, mixing, and mastering techniques to create a real song. Instructions to get the tutorial files are available in [Section 7.3, "Files for the Tutorial"](#).

7.4.1. Add Tracks and Busses

The program used to record these tracks was configured to record onto a separate track for the left and right channels, so **Ardour** will also have to be configured this way. It requires more setup, more memory, and more processing power, but it offers greater control over the stereo image and level balancing. We will use one track for vocals, clarinet, and strings, and two tracks for the marimba. This needs to be doubled to handle the stereo audio, so a total of ten tracks are needed. It might still be useful to manipulate the stereo tracks together, so we're going to combine them with five busses. This gives us the option of modifying both stereo channels or just one - you'll see how it works as the tutorial progresses. All of these actions take place within **Ardour**.

1. There is already a master bus, named "master". All audio being outputted should be fed through this bus.
2. Create five new busses:
 - a. From the menu, select 'Track > Add Track/Bus'
 - b. Adjust the options to add five stereo busses.
 - c. Click 'Add'
 - d. Five busses should appear in the canvas area, named "Bus 1" through "Bus 5", underneath the master bus.
3. Change the busses' names:
 - a. At the left-most side of the canvas area, each bus has a space with controls, including a box with the bus' name.
 - b. To rename a bus, use the mouse to left-click inside the box with the bus' name.
 - c. The box will turn into a text-editing box. Erase the contents, and write the new name.

- d. When you have entered the new name, press "Enter" on the keyboard to set it in **Ardour**.
 - e. The box will no longer be a text-editing box.
 - f.
 - Bus-marimba1
 - Bus-marimba2
 - Bus-voice
 - Bus-strings
 - Bus-clarinet
4. Create ten new tracks:
- a. From the menu, select 'Track > Add Track/Bus'
 - b. Adjust the options to add 10 normal mode mono tracks.
 - c. Click 'Add'
 - d. Ten tracks should appear in the canvas area, named "Audio 1" through "Audio 10", underneath the busses.
5. Change the tracks' names in the same way as you changed the busses' names. Remembering that each track here will hold only the left or right audio channel, each one should be pre- or post-fixed with a "O" or "L" for the left channel, or a "1" or "R" for the right channel. They should be called something like:
- marimba1-L
 - marimba1-R
 - marimba2-L
 - marimba2-R
 - voice-L
 - voice-R
 - strings-L
 - strings-R
 - clarinet-L
 - clarinet-R
6. Finally, we'll re-arrange the order of the tracks and busses. This isn't strictly necessary, and you can use whichever order you think makes the most sense. You might choose, for example, to put the marimba at the bottom, since it will be playing through most of the song.
- a. Find the session sidebar, to the right of the canvas area.
 - b. There are five tabs to choose: Regions, Tracks/Busses, Snapshots, Edit Groups, and Chunks. Select 'Tracks/Busses'

- c. All of the tracks and busses are shown in a list, along with a check-box that will show or hide that track or bus in the canvas area. Now you can see why it's a good idea to keep the word "bus" in the names of the busses.
- d. To change the ordering of tracks and busses, use the mouse to click and drag the name of the track or bus that you want to move.
- e. When you start dragging a track or bus, a line will appear in the list, marking where the track or bus would go. It can be helpful to move the track or bus that you are dragging to the side a bit, so that you can see the list itself.
- f. The interface makes it seem like you can move a track or bus on top of another track or bus. This is not the case. If it looks like a track or bus is going to be put on top of another track or bus, it will actually be placed into the list just above that track or bus.
- g. For editing, it is helpful to have each bus next to the tracks it will control. This can always be changed later.

7.4.2. Connect the Tracks and Busses

Although we have created a system of busses in our mind, we still have not told **Ardour** about it. You can use **QjackCtl** to confirm this: all of the additional tracks and busses are connected to output audio to the master bus. Worse still, the additional busses have no input signal at all. There are two approaches to letting **Ardour** know how we want to connect the tracks and busses. They will both be demonstrated, and you will be left to fill in the rest.

1. One way to connect tracks and busses is more suitable for small-scale connection changes.
 - a. Select the "marimba1-L" track by clicking in the track's control area, underneath the controls.
 - b. The editor mixer to the left of the canvas area should display the track's name near the top, and the track's colour (probably green in this case).
 - c. If you can't see the editor mixer, open it by using the menu. Click 'View > Show Editor Mixer' so that it is checked. You can also press 'Shift + E' on the keyboard to toggle its display.
 - d. After confirming that the editor mixer is showing the control for the "marimba1-L" track, look at the button on the bottom of the editor mixer, above 'Comments'. It should say "master", which means its output is connected to the master bus. This is not what we want, so click the "master" button.
 - e. When you click the 'master' button, a menu pops up, allowing you to choose a different output. We want to connect the track to the "Bus-marimba1" bus, which isn't in the list, so choose 'Edit' from the menu.
 - f. The connection window that appears looks confusing, but it isn't. Here's how it works:
 - The left side, labeled "Outputs," contains two output channels, "out 1" and "out 2," along with a list of everything to which those outputs are connected.
 - The 'Add' button adds an output channel. We're outputting the signal to a stereo bus, so two is enough.
 - The 'Remove' button removes an output channel.
 - The 'Disconnect All' button removes all of the track's output connections.
 - Clicking a connection in this list will remove it.

- The right side, labeled "Available connections," contains a list of all of the inputs offered by JACK.
 - Each JACK-aware application has a tab with its connections listed underneath.
 - Clicking a connection in this list will add it to the last-selected output channel.
- g. Click the 'Disconnect All' button.
- h. Click in the empty "out 1" list.
- i. From the "Available connections" list, click on the "Bus-marimba1/in 1" connection. It will be added to the "out 1" list.
- j. Then click on the "Bus-marimba1/in 2" connection. It will be added to the "out 2" list.
- k. The appearance of the connection lists will change to indicate that you've added a pair of connections.
- l. Click 'Close' to enable the connection change.
- m. Note that the "master" button now says something like "Bus-ma," because the track's output connection has changed.
2. The other way to change connections is much faster for large-scale changes like the ones required here.
- a. Choose **Window** → **Track/Bus Inspector**.
- The "Track/Bus Inspector" window will appear. It has a list of the tracks and busses on the left side, and four tabs of information on the right side.
 - The "Inputs" and "Outputs" tabs allow you to view and configure the input and output connections of the selected track or bus. The two "Redirects" tabs allow you to configure plug-in settings, which are not discussed in this tutorial.
- b. Select the "Bus-marimba1" bus from the list.
- You should recognize the "Input" and "Output" tabs.
 - Verify that the "marimba1-L" track is connected to this bus' input, and that the bus' output is connected to the "master" bus' inputs.
- c. Add both outputs of the "marimba1-R" track to the bus' input list.
- d. Check the outputs of the "marimba1-R" track. This isn't quite what we wanted, so remove the master bus connection.
- e. Adjust the remaining tracks so that they are connected as described in the table below.
- f. Verify the connections by viewing the busses' "Input" tabs.
- g. Verify that only the five busses are connected to the master bus' inputs.

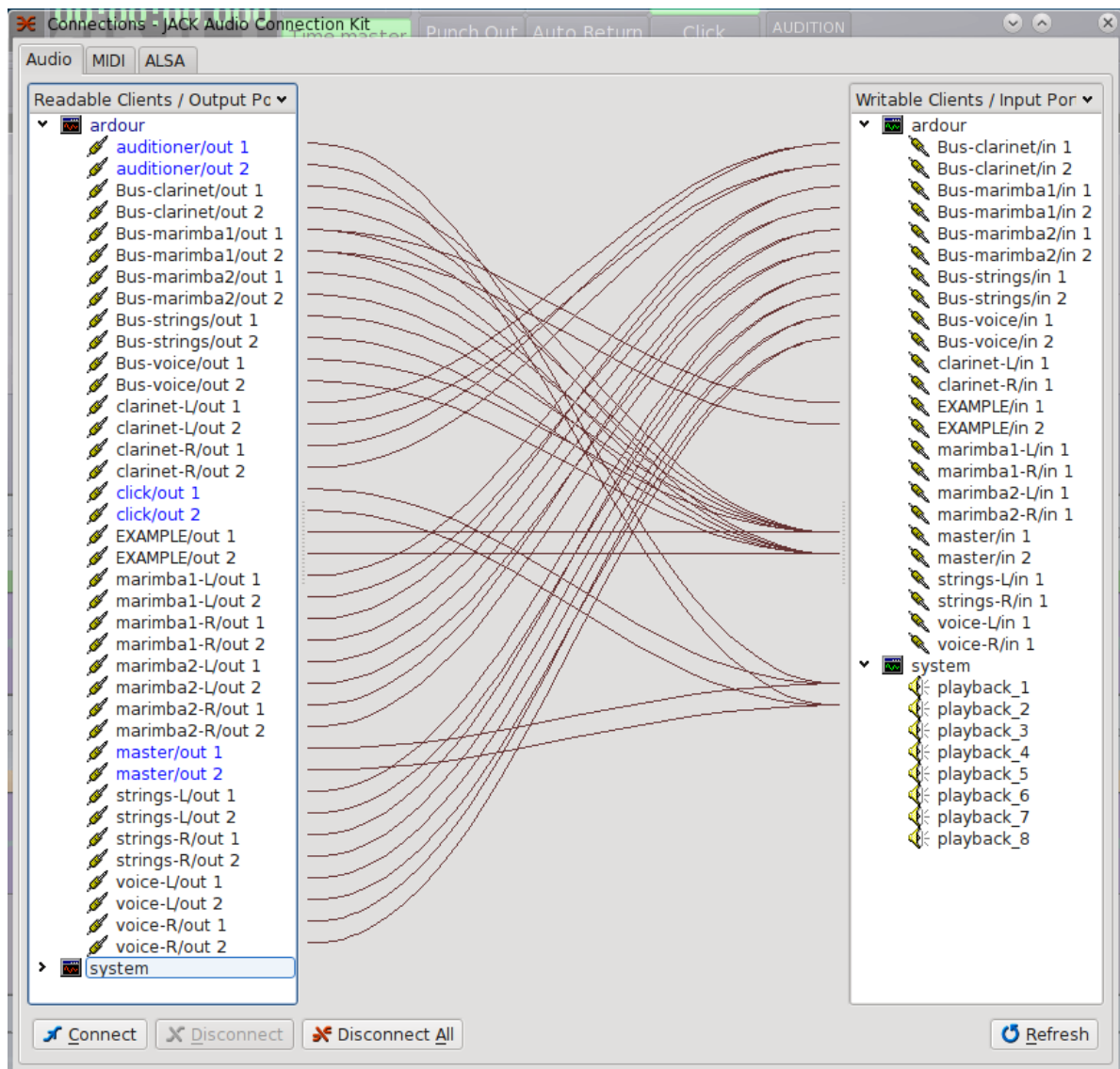


Figure 7.6. Connections in Ardour

7.4.3. Add Regions to Tracks

The next step is to add the regions into the tracks. It would be faster if we also cropped the regions at this point, but there are many reasons that it can be helpful to have longer regions, or rests (nearly silent space) within a region, so we'll keep them in tact for now.

1. In the session sidebar (to the right of the canvas area), select the "Regions" tab. This list contains all of the regions in the session. Each region is listed once, regardless of how many times it appears in the timeline, or if it's even used at all.
2. To add a region to the canvas area, simply click on the region's name, and drag it onto a track. The cursor will change as you do this, so that the vertical line of the cursor shows the point where the region will begin in the timeline.

Adding regions is just that easy!

You guessed it though - there's more to it than that, and it mostly has to do with the setup of this particular file. You will notice that the region list has many similarly-named regions, and that most of the names correspond to particular tracks and a bus. The files are named so that you know what's on them. They are given a number so that you know the sequence in which they're to be added

("Marimba_1" regions before "Marimba_2"), and a letter "L" or "R" at the end to signify whether the region is the left or the right channel. Furthermore, the regions that start with "ens-" belong on the "voice" tracks ("ens" is short for "ensemble," meaning that those regions contain a small vocal ensemble, whereas the "Voice..." regions contain just one singer). The "Here_Is_How" regions belong before the "Create_the_Inconceivable" regions. Remember: there is no technical reason that the regions are named as they are. The names are there to help you edit and mix the song. We don't need to use the "marimba2" tracks or bus yet, so just add all of the "Marimba_" regions to the "marimba1" tracks.

As you add the regions, you will learn a lot about manipulating regions in **Ardour**. Here are some tips to help:

- Use the mouse's scrollwheel (if you have one) to scroll vertically, seeing all the tracks.
- Hold down the 'Ctrl' button on the keyboard and use the mouse's scrollwheel to adjust the amount of time visible in the canvas window.
- Hold down the 'Alt' button on the keyboard and use the mouse's scrollwheel to scroll horizontally, moving along the timeline.
- After placing a region, you can move it around anywhere in the canvas area. You'll need to use the "Select/Move Objects" tool: click the pointing hand icon in the toolbar underneath the transport controls.
- If you are moving a region, be sure to click and drag from the large area above the coloured bar. If you click in the coloured bar at the bottom of the region, you will reduce the region's size.
- When you're just adding tracks like this, it's not important that they are synchronized perfectly. But you can use the "magnetic" snapping feature to automatically align the beginning of tracks to the second. As you slowly drag a region, when the start nears a second (indicated by vertical lines through the canvas area), the region will seem to "stick" to the second. Later, you may want to turn off this feature.
- Since we're just adding the regions imprecisely for now, put them into the appropriate track as tightly as possible, without overlapping.

7.4.4. Cut the Regions Down to Size

Now it's appropriate to cut some of the regions down in size. We're going to do this by removing some of the nearly-silent space before and after the material that was intended to be recorded. There are a few special cases, so first there will be specific instructions on how to do this to a region, and then general instructions for each region.

Let's start with the clarinet:

1. First, set the clarinet bus to "solo" mode by pressing the 's' button on the bus' control space. This will allow us to hear only the clarinet bus.
2. Listen to the first clarinet region by pressing "Home" on the keyboard to set the transport's playhead to the beginning, then pressing the space key to start the transport.
3. There is approximately 20 seconds of near-silence before the clarinet starts to play. If you listen carefully, you can hear the room, and somebody counting measure numbers.
4. If the channels aren't synchronized, you'll need to adjust their placement in the timeline. Use the yellow clock that appears when you drag regions - it's set to the same units as the secondary clock, and shows you the time of the beginning of the file. It's important to get it synchronized before the next step!

5. Choose either the "L" or "R" region. If you're using edit groups, it doesn't matter which you choose, because **Ardour** will realize that the regions in both tracks are "group equivalent" (that is, they're basically the same, so they probably belong together).
6. Use the mouse to click in the coloured bar of the region, close to where the clarinet starts.
7. **Ardour** will automatically move the start of the region in *both tracks*.
8. Move the playhead by clicking in the rulers at the point where you want the playhead to be, so that you can listen to the regions to ensure that you didn't cut out any of the useful audio.
9. If you want to adjust the beginning of the region, carefully move the mouse cursor to the start of the region, in the coloured bar. The cursor should turn into a double-ended left-and-right arrow. If you happened to accidentally remove some of the useful clarinet sound, you'll notice that it's still there. In fact, the beauty of trimming regions in this way is that it's "non-destructive," meaning that the entire original region is still there!
10. Notice that when you made the first adjustment, **Ardour** put an arrow beside the region name in the region list of the session sidebar. If you click on the arrow, you will see that there is another copy of the same region underneath, but it's white. **Ardour** wants you to know that the white-coloured region is a modification of the blue-coloured region. If you drag the white-coloured region into the canvas area, you'll notice that it starts at the same time as the region you just modified. It can also be dragged out to the full size of the original region, which would create another modified version of the original. While it seems like **Ardour** stores multiple copies of the region, it actually just stores one copy, and the information required to make it seem like there are many.
11. Adjust the end of the region so that there isn't too much silence after the clarinet. Be extra careful with this, so that you don't cut out any of the clarinet, which gets very quiet at the end of the region. There isn't much to cut off! Note that you cannot click in the coloured bar when adjusting the end of a region, so you'll have to click-and-drag.

Here are the instructions to edit the rest of the regions. As you trim the regions, you may find it helpful to move them all towards the start of the session. Remember to change the bus that's in "solo mode" when you move to different tracks, or else you won't be able to hear the tracks you're trying to edit. You may also notice that some of these regions contain identical or nearly-identical music, which we'll deal with later.

- Clarinet
 - Clarinet_2:
 - Starts with sound, but it's not useful, so cut it out, along with the silence after it.
 - End has a good chunk of silence to cut out.
 - Clarinet_3, Clarinet_4: the silence at the beginning and end can be removed, but leave the silence in the middle.
- Strings:
 - Strings_1: Starts with grumbling noise, which was not intended to be captured. You can keep it or discard as you please - Esther decided to keep it in, and so will I.
 - Strings_2, 3, 4: Silence at the beginning, but only a little at the end. You can cut out the talking, or deal with it later.
- Voice:
 - Voice_1, 2, 3, 4: Simply remove the silence from beginning and end, leaving the mistakes, which we'll take care of later.

- ens-Here_Is_How-1, 2, 3: It's too difficult for now to trim all of the extra noise, so just get most of it. The breathing and shuffling can be removed later.
- ens-Create_the_Inconceivable: For now, keep both of the attempts. Later, we'll choose which one we prefer.
- Marimba:
 - Marimba_1: Don't trim the beginning of this region; we'll use it to time the start of the session. You can trim the silence at the end, but be sure that you don't clip it while the sound of the marimba is still ringing.
 - Marimba_2, 3, 4, 5, 6, 7: Trim the silence around these as desired, still being careful not to clip the marimba while it's still ringing. This may require cautious listening at high volume settings.

Now that we have roughly trimmed the silence surrounding the portions of audio that we really want, we'll have an easier time putting them in the right order.

7.4.5. Compare Multiple Recordings of the Same Thing

Part of the power of recording with a DAW is that the same material can be capture multiple times. Mixing and matching like this allows us to seek the "perfect" performance of a piece of music. A few of the regions in this file are multiple takes of the same material. There are a few occasions where we can definitively say that one is better than the other, and there are a few occasions where it depends on your personal taste. This section covers techniques that can be used to further cut up the audio, in this case with the end goal of comparing and choosing preferred sections. Not all choices will be made yet.

7.4.5.1. Clarinet_1 and Clarinet_2 Regions

1. Listen to the Clarinet_1 and Clarinet_2 regions. They're the same musical material, and they're nearly identical, so it's too early to decide which one to use. But, let's label them so that we know they're the same.
 - a. Select the "Clarinet_1--L" region from the region list in the session toolbar by left-clicking on the triangle to the left of the blue name, so the white name appears, and left-clicking once on the white name. Remember that the white region was created when you trimmed the empty area out of the original (blue) region.
 - b. Then click again on the white name, and a text box will appear.
 - c. Change the textbox so it says "Clarinet_1A--L"
 - d. Press 'Enter' on the keyboard to set the name.
 - e. Rename the following regions as shown:
 - "Clarinet_1--R" becomes "Clarinet_1A--R"
 - "Clarinet_2--L" becomes "Clarinet_1B--L"
 - "Clarinet_2--R" becomes "Clarinet_1B--R"
 - f. Since Clarinet_1 and Clarinet_2 represent the same musical material, we've renamed them to show it. Now, they're both Clarinet_1, with two versions, "A" and "B."
2. There will be some naming inconsistencies at this point: the blue-coloured regions still have their original names, and the canvas area doesn't have any region called "Clarinet_2"! If this bothers you, you can rename the other regions to suit.

7.4.5.2. Clarinet_3 and Clarinet_4 Regions

Listen to the Clarinet_3 and Clarinet_4 regions. Clarinet_4 starts with the same material that's in Clarinet_3, and ends with the same material that's in Clarinet_1A and Clarinet_1B. First rename the "Clarinet_3" regions to "Clarinet_3A," and the "Clarinet_4" regions to "Clarinet_3B." Then, we'll extract the Clarinet_1-like portion from the Clarinet_3B regions.

1. Under the transport toolbar, select the "Select/Move Ranges" tool, which looks like a double-ended left-to-right arrow with vertical lines at the sides.
2. The cursor will change to look like a text-editing "I"
3. Scroll to the end of the "Clarinet_3B" regions, so you can see the part the sounds like the "Clarinet_1" regions.
4. Click and drag the mouse over the "Clarinet_1"-like region in one of the tracks, to select them.
5. Because of the edit group, **Ardour** will automatically select the same area of both tracks.
6. We have to be sure to select all of the "Clarinet_1"-like material, so after you've selected a range, right-click on the range, and select 'Play Range' from the menu.
7. If you want to adjust the selected range, use the darker squares at the top corners of the range-selection box. When you put the cursor over one of these boxes, it will change into a double-ended, left-to-right arrow.
8. Now create a new region from the range. Right-click on the selected range, and select 'Bounce range to region list' from the menu.
9. The range will appear as independent regions in the region list, called something like "clarinet-L-0-bounce-1". This isn't very helpful, so rename the regions to "Clarinet_1C--L" and "Clarinet_1C--R". Notice that the new regions are coloured white.
10. We no longer need the range tool, so select the "Select/Move Objects" tool, which is the hand-shaped icon just to the left of the range tool, underneath the transport toolbar.
11. The selected range will no longer be selected. Trim the end of the "Clarinet_3B" regions down to size, past the material that we just bounced to the region list, and past the near-silence before it.
12. Now move the "Clarinet_3" regions back, to make room for the newly-created "Clarinet_1C" regions.
13. Drag the new "Clarinet_1C" regions onto the canvas after the other "Clarinet_1" regions, and adjust the spacing of the other regions, if you wish.

7.4.5.3. Strings_1 Regions

1. These regions start with the grumbling noise that was accidentally recorded. If you decided to leave it in, you could bounce the grumbling to the region list, so it can be controlled independently of the strings that follow.
2. The new regions are probably called something like "strings-L-0-bounce-1". Because I know that the sound is chairs being moved across the floor in a room upstairs from the recording studio, I'm going to call the regions "Chairs--L" and "Chairs--R".
3. Then remove the noise of the chairs from the Strings_1 regions.
4. Since the marimba2 tracks aren't yet being used, we can put the Chairs regions there - at least for now - just to remember that we have them.

5. Listen to the Strings_1 region. You'll notice that the noise of the chairs continues throughout the region. We can remove it later.
6. You wouldn't know this without carefully listening to the song, or having the sheet music for "Here Is How," so I'll tell you: the musicians make a mistake near the end of this region, which is why the violinist says, "sorry," just after that.
7. We'll need to remove the error, so adjust the end of the track to make it about six seconds earlier. In the canvas view, you want to put the end of the region just before the second-last "blob."

7.4.5.4. Strings Regions

These four regions are all tied together, and the material overlaps between them.

- Strings_2 covers most of the same material as Strings_1, and goes for a bit longer. It doesn't have the noise of the chairs, but there is a playing mistake after about the first minute and 20 seconds (1:20), so we can't use the end.
 1. Trim the last 32 seconds or so.
 2. Rename this region to "Strings_1B"
 3. Rename the "Strings_1" regions to "Strings_1A," to match with 1B
 4. We'll decide which of these regions to use later.
- Strings_3 is also the same material, so rename it to "Strings_1C."
- Strings_4 starts with the place where Strings_1A went wrong, and goes on from there. Let's keep it as it is, for now.

7.4.5.5. Voice Regions

These regions contain some overlap, but it is relative simple to sort out.

- Voice_1 contains two chunks of audio. The first one is good, but the singer had a problem making the second one clear, so we're not going to use it.
- Voice_2 contains the second chunk of audio that was recorded poorly in Voice_1. She also has a bit of a problem in Voice_2. Let's fix this up now.
 1. Trim the Voice_1 regions to remove the second chunk of audio, and the near-silence that precedes it.
 2. One way to deal with the Voice_2 region is simply to cut off the first part of the region, which contains the words "I have your flax-" and some near-silence.
 3. The second time the singer sings, "I have your flax-," it sounds a bit rushed, so I'm going to combine the first "I have your flax-" with the following "golden tails to ... "
 - a. Use the "Select/Move Ranges" tool to select the first time the singer says "I have your flax-," being careful to capture all of the word "flax," but none of the near-silence that follows.
 - b. Use the loop function of the transport to ensure you've got the right range selected:
 - i. Select a range, then right-click on the range and select 'loop range'.
 - ii. If you want to make an adjustment, stop the transport, and adjust the range as desired.
 - iii. To listen again, right-click on the range and select 'loop range'.
 - iv. You may need to zoom in so that you can adjust the range with sufficient detail. Hold the 'Ctrl' key on the keyboard, and use the scrollwheel to zoom in.

- v. When you're done with the looping function, remove the looping markers from the timeline. They look like green triangles with the word "Loop" written beside. Move the cursor over a triangle, so that it changes colour. Then press the 'Delete' button on the keyboard.
- c. When you are happy with the range that you've selected, right-click on the range and choose 'Consolidate range' from the menu.
- d. **Ardour** will create a region from the range that you selected, leaving it in-place. It will also divide the space in the region before and after the new region, leaving you with many smaller regions, all conveniently collected in the session toolbar's Regions list, under the blue-coloured "Voice_2--L" and "Voice_2--R" regions.
- e. Trim the rest of the original Voice_2 region, so that it starts with "golden," and does not contain any of the previous word ("flax-"). You don't need to use the range tool, but you can if you wish.
- f. Then, push the two regions together, so that it sounds like "I have your flax-golden tails to..."
- g. This isn't going to sound perfect, but you might prefer it to simply trimming the beginning off the Voice_2 region.
- h. It's important to remember to move both regions together. If they are accidentally separated, then you can easily enough move them back into place.
- Voice_3 contains two chunks of audio. We'll leave it alone for now.
- Voice_4 contains the same two chunks of audio as Voice_3, but goes on to include more. We can't yet determine whether to use Voice_3 or Voice_4 for those two chunks, so we're going to leave them in both regions.

7.4.5.6. ens-Here_Is_How Regions

- ens-Here_Is_How-1 contains two chunks of similar audio, both correct.
- ens-Here_Is_How-2 contains two chunks of similar audio. It's different from ens-Here_Is_How-1, but the second of these chunks has incorrect pitches, but the timing between the two chunks is correct.
- * ens-Here_Is_How-3 contains the second chunk of audio from ens-Here_Is_How-2, with the correct pitches. Since we want to maintain the correct timing from ens-Here_Is_How-2, simply drag ens-Here_Is_How-3 over top of ens-Here_Is_How-2. Align the regions so that the wave-form shape of -3 is as close to covering that of -2 as possible. These regions will also have to be carefully moved together.

7.4.5.7. ens-Create_the_Inconceivable Regions

There are two takes of the same material in this region.

1. Listen to them both, and decide which you prefer - it's up to your preference. Remember, you can also reverse your choice later, because **Ardour** will not delete the material that you remove by trimming the region.
2. Use the range tool to select the range which includes the take that you prefer.
3. Use the transport's looping mechanism, if you wish, to be sure that you selected the right range.

4. Right-click on the range, then choose 'Crop region to range,' which will automatically trim the region for you.

7.4.5.8. Marimba Regions

The marimba regions do not need adjustment.

7.4.6. Arrange Regions into the Right Places

We're going to start by arranging the marimba, since it plays a relatively consistent rhythm throughout most of the song. It is a good idea to start with something like this, so that the following tracks and regions can be related to it.

All of the times here are given in minutes and seconds. The tutorial file is configured to use this unit by default, but if you have changed it, you will need to keep that in mind. Also, I have not cropped or trimmed the "Marimba_1" region, and it is located in the "marimba1" track, starting at 00:00:00.000. If you have modified that region, I suggest restoring it to the original size.

7.4.6.1. Start with the Marimba Regions

1. When you have made sure that the "Marimba_1" regions are not cropped or trimmed, and that they start at 00:00:00.000, we can lock it in place.
 - a. Right-click on the regions, and navigate to the 'Selected regions' menu, then click 'Lock' from that menu.
 - b. Notice that **Ardour** puts > and < around the name of the region, in the canvas area.
 - c. Also notice that you can no longer move the region with the "Select/Move Objects" tool.
2. Now place a marker to show the exact moment when it starts: six seconds into the session.
 - a. Zoom in appropriately so that you can see where the six-second mark on the ruler should go (but you needn't see it yet). Hold the 'Ctrl' button on the keyboard, and use the mouse's scrollwheel to zoom in, or press the '=' button on the keyboard to zoom in and the '-' button to zoom out.
 - b. Move the cursor to near the six-second mark (again, not important to be precise yet), and right-click in the "Location Markers" row. Select 'New location marker' from the menu.
 - c. Click-and-drag the yellow-green arrow that appears, so that the yellow clock shows 00:00:06.000, indicating six seconds precisely. Release the mouse.
 - d. Move the cursor over the marker, so it changes colours from yellow-green to red-tan (coral). Right-click and select 'Lock' from the menu, so that the marker will not be accidentally moved.
 - e. Again, right-click while the cursor is over the marker. Select 'Rename' from the menu.
 - f. A small window will appear. Write the name of the maker, "marimba-start," and click 'Rename' to set the new name.
3. Since we will be adjusting the placement of regions in the timeline very precisely, we will need to use different "Snap/Grid Mode." Each setting is useful for a different kind of task.
 - We can change the mode using the toolbar just above the canvas area. The pop-down menu probably says "Magnetic," indicating that it's in "Magnetic Snapping Mode," but it might also say "No Grid" or "Grid."
 - No Grid: This mode gives the user full control over where they will place a region. It is useful for doing highly-precise alignment, as we're about to do.

- Grid: This mode only allows the user to place regions where they will start on a grid-line. Unless you changed it, your grid is set to two seconds, so you can only start regions in two-second intervals - **Ardour** will not allow you to place a region so that it starts on an odd-numbered second, or anywhere in between.
 - Magnetic: This mode allows the user to place a region wherever they like, but when the start of the region is near a grid-line (an even-numbered second, in this session), the start of the region will automatically "snap" to that point. It behaves as if the start of regions were magnetically attracted to the grid lines.
4. Adjust the snap/grid mode to "No Grid."
 5. Move the "Marimba_2" regions so that they are in the "marimba2" tracks, so that the sound in "Marimba_2" starts at about 15 seconds (00:00:15.000) on the timeline. You'll have to move the grinding of the chairs out of the way, if you decided to keep it. Move it to the "strings" tracks, before the "Strings" regions.
 6. Ensure that both the "marimba1" and "marimba2" busses are on solo mode, so you will be able to hear them both.
 7. Now here's the difficult part: you'll have to align the two tracks, so that they start together.
 - The sound in the "Marimba_2" regions should start at the same time as the second pattern in the "Marimba_1" tracks, which is at about 15 seconds on the timeline.
 - You'll need to zoom in - it helps to be able to see the wave-form shapes of the regions. More importantly, zooming in allows you to adjust the placement of the regions with greater precision.
 - You may want to select a range, and use the loop mode of the transport. This will allow you to hear the start of the regions again and again, ensuring that they are aligned.
 - If you feel like you've got it close, but not quite together, then try moving it far away, listening, then adjusting it closer again.
 8. Once you have aligned the first few seconds of this pattern, it will eventually become unsynchronized, which is okay. The beginning will be the most noticeable part; when the listeners become accustomed to the sound of the two marimba tracks together, they will stop paying close attention. Furthermore, they are likely to be focussing on the words by the time the marimba tracks become unsynchronized.
 9. Remember to lock the "Marimba_2" region!
 10. Listen to the rest of the regions together. The end of the regions, at about 00:02:20.000, should be aligned, if you aligned the start correctly. The higher and lower marimbas will alternate.
 11. Move the "Marimba_3" regions so they start at approximately 00:03:20.000, with "Marimba_3" in the "marimba1" tracks. We will find a final alignment later.
 12. Move the "Marimba_4" regions so they start at approximately the same time as the "Marimba_3" regions, but in the "marimba2" tracks. We will find a final alignment later.

7.4.6.2. Continue with the Voice Regions

Throughout this section, you will need to move un-placed regions out of the way, farther down the session, so that they don't interfere with the alignment process. Remember to lock the regions once you put them in place. They can be unlocked and re-aligned later, if you choose. Finally, it will help if you place a marker (like the "marimba-start" marker that we placed earlier) where each region will

start. When you place a marker, you can click on it, and move the blue place-marker line. This will help you to align the start of sound in a region to the place where you want it to be.

1. Enable "solo" mode on the "voice" bus, so that you can hear it as well as the marimba busses.
2. Slide the "Voice_1" regions so that the singer starts at about the same time as the higher marimba.
3. Slide the "Voice_2" regions so that the singer starts at about **00:00:48.00**
4. Slide the "ens-Here_Is_How-1" regions so that they start singing at about **00:01:33.300**
5. Slide the "ens-Here_Is_How-2" (and the adjoined "ens-Here_Is_How-3") regions so that they start singing at about **00:02:11.500**
6. After playing closer attention to "Voice_3" and "Voice_4," you realize that the singer misses a word ("plan *in* you spider's ear") in "Voice_4." Because "Voice_3" doesn't contain the second part of "Voice_4," we'll need to trim the "Voice_4" region, and use both.
 - a. The singer should start singing in "Voice_3" at about **00:02:24.500**
 - b. The signer should start singing "and here is how" in "Voice_4" at about **00:02:43.000**
7. Slide the "ens-Create_the_Inconceivable" regions so that they start singing at about **00:02:59.000**

7.4.6.3. Align the Marimba at the End

Now that we have roughly placed all the singers' regions, we can figure out where to put the concluding marimba passage.

1. Listen to the "ens-Create_the_Inconceivable" regions. We're going to start the marimba's ending regions somewhere after the highest singer has said "if you know what I mean," but before she stops singing the word "mean."
2. It's up to you exactly where to re-start the marimba. Pick a place, and move the two "Marimba_3" region to that place in the "marimba1" bus, and the "Marimba_4" region in the "marimba2" bus.
3. You may need to set up a range and use the transport's looping function to get the alignment exact.
4. This particular entry is going to be difficult, because the low marimba enters quietly and blooms into sounding, while the higher marimba has a hard attack that dies. This means that, if you start both marimba tracks at the same time, the listener will perceive the lower track as starting after the higher track. If you don't believe me, try it out!
5. To solve this problem, the original editor (Esther) had the upper marimba start a bit later than the lower marimba.

If you were to listen to the session so far, you would notice that the marimba is way too loud compared to the singers, and that everything sounds very narrow. This is because we're only arranging regions, and we haven't done any of the mixing yet!

7.4.6.4. Align the Strings Regions

1. Slide the "Strings_1A" regions so that they start playing at about 00:00:28.00, which should leave plenty of room for the sound of the chairs, if you decided to keep it.

2. The "Strings_4" region begins in the same way that the "Strings_1A" regions end. Listen to both, and choose which you prefer, then use the position of the sounds in the "Strings_1A" region to guide your trimming and alignment of the "Strings_4" region.
3. The other two strings regions contain the same musical material as the "Strings_1A" region. We can't decide between them yet, so move them both past the end of the concluding marimba regions, so they won't be heard.

7.4.6.5. Align the Clarinet Regions

As with the Strings regions, we will simply pick a suitable clarinet region, and move it into place, leaving the choice between multiples until later. When you're moving regions a long distance like this, it helps to zoom out for a bit.

1. Slide the "Clarinet_1A" region so that the sound begins just after 00:01:06.200
2. Slide the "Clarinet_3A" region so that the sound begins just after 00:01:35.000

7.4.7. Listen

Before moving on to the mixing stage, listen to the whole song, to make sure that the ordering makes sense. When you're listening, remember that the volume levels and balances will sound off, and that the whole session will sound very "centred" in the stereo image.

7.5. Mixing a Song (Tutorial)

The next stage is called "mixing," and it primarily involves two tasks: setting volume levels, and adjusting the stereo pan settings. We'll use automation to store our fader and panning adjustments, and see how handy it can be to have left and right channels recorded on separate tracks, combined with sub-master busses.

In terms of producing a recording of a live musical performance, it is the mixing stage where the audio engineer (in this case, you) has the most creative influence. Careful adjustment and tuning of the tracks will greatly affect the listeners' experience.

Finally, it should be noted that, moreso than in the editing stage, the mixing stage should *not* be understood as progressing in a linear manner. This means you should not be following the tutorial from start to finish, but jumping between sections are desired. You should set up the tracks for stereo output first, and then read through all the sections and follow their advice as you wish, but sometimes returning to previous activities to re-tune those settings. When one setting is changed, it tends to have an effect on other settings, so if you set the level of a track once, then change its panning, you should check that the levels you set are still desirable - they'll probably need some tweaking, however minor it may be.

7.5.1. Setting the Session for Stereo Output and Disabling Edit Groups

Part of the reason that the session sounds so bad is that all of the audio has been routed through both the left and right channels equally, making it a "mono" recording, even though we have the material of a "stereo" recording. This could easily have been done sooner, but it wouldn't have made much of a difference until now. Whereas mixing was focussed on getting the regions assembled so that they are like the song, mixing is about fine-tuning the regions and tracks so that they make the song sound great.

Disabling the edit groups is also a good idea, because leaving them enabled actually *reduces* functionality in this stage of production. With edit groups enabled, any change that we make to one of

the tracks will automatically be made to the other track, too. We want to be able to adjust the tracks independently; for cases where both tracks need the same adjustment, we will use the sub-master bus to which they're attached.

These steps will disable the edit groups, and re-configure this session's tracks for stereo output.

1. We need to adjust tracks independently, so the edit groups must temporarily be disabled.
2. Flip to the "Edit Groups" tab of the session sidebar.
3. Uncheck the "Active" button for all the groups. If you want to re-enable an edit group later, simply re-check the "Active" button.
4. Open the mixer window by selecting from the menu 'Window > Show Mixer'. If you have a multiple-monitor setup, it can be very useful to keep the mixer window on a separate monitor from the main editor window. If you don't have a multiple-monitor setup, you can keep the mixer window on a separate virtual desktop. Of course, these are both optional steps.
5. Near the bottom of each track's mixer, above the buttons, is a small black rectangle with three grey triangles and a green vertical line. Each of the busses have two of these rectangles. This controls the panner, which adjusts a track's left/right position in the stereo image.
6. You can adjust the panner by click-and-dragging in the panner display. You don't need to click on the green line, but the line will show you the approximate placement of the track in the stereo image.
7. Each "left" track, ending with a capital "L," should have the green line set all the way to the left.
8. Each "right" track, ending with a capital "R," should have the green line set all the way to the right.
9. Each bus is probably already set correctly. The bus' upper window represents the left channel, and the green line should be all the way left. The bus' lower window represents the right channel, and the green line should be all the way right.

The mixer control located above the panner is called the "fader," and it allows you to adjust a track's level.

7.5.2. Set Initial Levels

As with editing, the point here is to get the levels set into the right general area, so they work for most of the track. When you start using an automation track later, the levels can be fine-tuned, and changed within the session. Here is one possible procedure to use for an initial level adjustment:

1. Open the mixer window with the menu, by choosing 'Window > Mixer'. As mentioned earlier, it can be convenient to put the mixer window on another monitor or virtual desktop.
2. Set all of the faders to 0 dB. They are probably already set to this level, unless you changed them earlier.
3. Take the quietest track, when set to 0 dB, as the limiting factor on how loud the other tracks should be. Since it's generally safer to avoid amplifying audio signals, if we use the quietest track as the "base-line," then we'll have to adjust the level of the other tracks *down* to suit. In this case, the voice tracks are the quietest.
4. At this point, it's best to stick with adjusting the busses' faders. If you adjust the faders on the tracks, this will affect the panning, and could lead to confusing problems later.

5. Play through the session, and adjust the faders of the busses so that all of the tracks can be heard equally well. Remember that you're just aiming for *most* of the session to be balanced at this point; a single fader setting is unlikely to be acceptable for the entire session.
6. You can adjust the fader setting in two ways:
 - a. Click-and-drag the vertical, dotted control strip to the left of the level meter (which lights up as a track is playing).
 - b. Use the indicator box as a text field: click in the box, erase the number that it shows, and write in a new number. Press 'enter' on the keyboard to set the new value.
7. You might wish to change the order of tracks and busses in the canvas area, which will change the order in the mixer window. Putting all of the busses together makes it easier to see them.
8. You could also choose to not display the tracks, again allowing you to focus on the busses that you will be changing. Do temporarily hide a track or bus in the mixer window, use the toolbox on the left side of the mixer window. Un-check the "Show" box for each of the tracks or busses that you want to temporarily hide.
9. The "maximum level" indicator on the fader tool might help you to judge how loud each track is. This indicator is located above the meter, underneath the "Solo" button. The indicator displays the highest level produced by the track since the indicator's last reset. You can reset the indicator by clicking on it.

7.5.3. Set Initial Panning

Setting up the initial panning takes quite a bit more thought than setting the initial levels. Different music will have different requirements, but the main purpose of adjusting the panning for this sort of recorded acoustic music is to ensure that each performer has a unique and unchanging position in the stereo image. When humans are listening to music, they implicitly ascribe a "location" to the sound - where their brain thinks it should be coming from. When listening to recorded music, we understand that the sound is actually coming from speakers or a set of headphones, and that the performers are not actually there. Even so, it can be difficult, tiring, and unpleasant to listen to music where the imagined position of a performer or sound is constantly changing - just as it's difficult and tiring to listen to music which has poorly balanced levels.

As if it weren't already difficult enough, the stereo image is created in our minds as a complex combination of many factors: quieter sounds and later sounds seem to be farther away than louder and earlier sounds. Although the DAW's panner can only put the signal somewhere in a straight line between "all the way left" and "all the way right," our brains process sound as existing in a three-dimensional world. A master audio engineer will be able to control these factors with relative ease, but for us it's going to involve much more trial and error.

A particular obstacle with this session is that the regions with the soloist put her in a different imagined position than the regions where the soloist is singing with other singers. Because these happen in the same tracks, we'll use automated panner and fader tracks to help solve this problem. Listen for yourself: start at about 00:02:40.000, and pay attention to where the soloist seems to be standing in the "Voice_4" regions and the "ens-Create_the_Inconceivable" regions. It seems to me like she moves from nearby on the right to a farther distance just to the left; somehow without bumping into the other people in the vocal ensemble, or the strings, which also seem to be in the way! You might argue that most listeners would not pick this up, and that's probably the case. Even so, I would counter that the drastic change of level and panning would be passively detected by those same people, even if they only consciously perceive it as being "not quite right."

Here's one way to start:

1. Listen to the session as needed, and see if you can place the location of the instruments/singers throughout most of the session. You'll need to remember this, so consider writing it down, or drawing a map.
2. Now, draw a map of where you think everything should be. Especially in non-standard ensembles like this, there is no pre-defined seating or standing arrangement. Some tracks will need very little adjustment, but others may need extensive adjustment. In general, the less tweaking required, the better the session will sound - so if something seems like a track already has a consistent location, and it doesn't conflict with other tracks, then it's probably better to leave it alone.
3. Here's what I hear. It may be different from what you hear, especially if you happened to do your initial level-setting differently:
 - Both of the marimba tracks are consistent throughout. The "marimba1" tracks seem to be about 5 metres in front of me, of to the left a bit. The "marimba2" tracks seem to be about the same distance away, but almost directly to my right.
 - All of the strings regions seem to be consistent, with the violin placed just left-of-centre, and the 'cello just right-of-centre. They seem to be a bit closer than the marimbas.
 - The clarinet seems to be on the opposite side of the higher marimba; about 5 metres away, half-way between in front and to the left.
 - The vocal ensemble seems to be standing in the same place as the strings, but extending a bit more to the right.
 - The solo vocalist seems to be standing in the same place as the male singers in the vocal ensemble.
4. Here's how I plan to fix it; directions are given assuming the listener is looking north:
 - Establish two rows of performers, surrounding the listener in a semi-circle.
 - The strings will be in the closer row, to the north-west. This requires moving them to the left a bit.
 - The vocal soloist will be in the closer row, just east of north (the middle). This requires moving her to the left *just* a little bit.
 - The vocal ensemble will be in the closer row, spread from north to north-east, allowing the soloist to remain in the same place. This will mostly require fader adjustment, to make the ensemble seem closer.
 - The lower marimba will be in the outer row, to the north-west. This may not require any adjustment, but perhaps a slight move to the left.
 - The higher marimba will be in the outer row, to the north-east. This requires a slight move to the left.
 - The clarinet will be in the outer row, to the north. This will require significant adjustment to the right.

I chose that particular layout because it requires relatively minimal adjustment, and it makes a certain amount of sense in terms of traditional instrumental ensemble seating patterns. Also, the notes played by the clarinet in this song seem suitable to appear as if from far away, and the passages are played with good expression, so I think it will be relatively easy for me to achieve that effect. The most important consideration was the placement of the vocal ensemble and the solo vocalist within it. Although the solo vocalist sings the highest part in the ensemble ("soprano"), the stereo recording seems to indicate that she was not standing at the left-most position in the ensemble (I

also know this because I was present during the recording). This adds an extra difficulty, in that the fader and panner settings for the whole voice track must be based on the moment in the "ens-Create_the_Inconceivable" region where the second-highest singer ("alto") sings just after the highest singer, who is the soloist.

Make rough adjustments to most of the tracks, to place them in approximately the right space in the stereo image. You may wish to adjust an individual track's panner setting, in addition to the busses' panner settings; they will have a slightly different effect. For the marimba tracks, you may wish to fine-tune things now, adjusting the fader settings. Because these tracks are so consistent, they will require relatively little automation, and therefore will benefit more from a more thorough initial set-up procedure. Remember that it's better to be turning down the fader than turning it up!

It's probably easier to avoid working with the voice tracks for now.

7.5.4. Make Further Adjustments with an Automation Track

So far, we've been crudely adjusting the fader and panner settings manually. This won't work if you want to change the settings while a session is playing; you would have to change all of the settings by yourself, every time you play the session. This quickly becomes complicated - not to mention difficult to remember. "Automation" allows effects (like the panner and fader) to be moved automatically during session playback. An automation track is simply a track that contains no audio, but rather instructions to adjust a particular effect. Automation tracks usually resemble audio tracks, but they hold lines and points, to show the settings changes. Automation tracks can, in effect, be "recorded," but we're going to use a more basic editing method. Automation tracks can be assigned to busses and tracks.

Here's how to create an automation track, and fill it in. We're going to adjust the fader on the lower marimba, so that it is louder in the introduction, and becomes quieter as the higher marimba and solo vocalist join in.

1. In the canvas area, click the 'a' button on the "Bus-marimba1" bus' control box, to open the "automation" menu.
2. Click 'Fader' in the automation menu.
3. An automation track, which controls the fader, will appear underneath the bus.
4. If you click in the automation track, a point will appear. Each point represents an absolute setting for the control. After the point appears, if you click-and-drag it, the yellow numbers by the cursor will tell you the fader's setting at that point.
5. If there are two or more points in the automation track, lines will appear to connect them. The fader will be moved gradually between absolute settings, as shown by the line connecting the points.
6. If you make a mistake and want to start over, you can press the 'clear' button on the automation track's control box. Unfortunately, you can't remove a single point. This isn't really necessary anyway; if you accidentally add too many points, simply use the extra one to keep a setting constant.
7. Add one point to the beginning of the automation track, with a setting of 0.0 dB
8. Add one point at about 00:00:15.000, with a setting of 0.0 dB
9. Add one point at about 00:00:16.500 (where the singer starts), with a setting of -10.0 dB, or whatever you set earlier.

10. Now you've set up an automation plan, but the fader is still in "Manual" mode, so the automation track will have no effect. Change the automation track's setting by clicking on the mode button in the track's control box. The button currently says "Manual."
11. From the menu, select "Play," which will cause the automation settings to be played. In "Manual" mode, you have to adjust all settings manually. In "Write" mode, changes that you make as the session plays will be recorded into the automation track, over-writing previous settings. In "Touch" mode, changes that you make as the session plays will be incorporated into the pre-existing automation settings.
12. Finally, listen to confirm that you like the automated panner change. If you don't, you can always adjust it now or later.

Now - here's the difficult part! Use automation to change the fader and panner settings throughout the session. In particular, ensure that the voice tracks are consistent.

7.5.5. Other Things You Might Want to Do

The mixing stage involves a lot of minor (and major) tweaking. Here are some things that you might want to do, which aren't adjusting the fader and panner settings:

- Re-align tracks to ensure that they're synchronized.
- Find a meaningful way to incorporate the region with the sound of the chairs.
- Compare the currently-unused clarinet regions with the in-use ones. Try different combinations of the regions, and remove the unused regions from the session.
- Compare the currently-unused strings regions with the in-use ones. These regions are much longer than the clarinet regions, so you might even want to pick and choose ranges of regions to switch back and forth.
- Have a friend - or at least somebody else - listen to the mix you're preparing. Get their opinion on difficulties that you may be having, or use them as a more generic listener.
- Listen to the mix on different kinds of reproduction equipment (speakers and amplifiers). The same audio signals will sound different when played on different equipment.

7.5.6. Listen

When you have finished mixing the song, you must listen to it. You should listen to it with as many different devices as possible: headphones, speakers, home theater systems, and so on. You should also ask your friends and colleagues to listen to your work. Other people hear things differently from you, and will give you different feedback.

7.6. Mastering a Session

To be a true master at mastering sessions requires years of experience and careful optimization for the target format. Knowing just the right equalization and filtering settings to apply is an art in itself, worth a full user guide. This section is concerned with getting the audio out of a session, to a useful format.

7.6.1. Ways to Export Audio

There are three ways to export audio from an **Ardour** session:

1. by region,

2. by range, or
3. by session.

To export a region:

1. Ensure the region is placed in the canvas area.
2. Right-click on the region.
3. Select the region-name's menu, then 'Export'.
4. Continue with the Export window.

To export all audio in a range on the timeline:

1. Select the range with the "Select/Move Ranges" tool. Regardless of which track you select, all tracks can be exported.
2. Right-click on the range.
3. Select 'Export' from the menu.
4. Continue with the Export window.

To export all audio in a session:

1. From the menu, select 'Session > Export > Export > Export session to audiofile', or on the keyboard, press 'Ctrl + Alt + e'
2. Continue with the Export window.

7.6.2. Using the Export Window

Regardless of which export method you choose, the "Export" window is similar. When you export a region, you do not get to choose which tracks to export (by definition you are only exporting that region's track).

7.6.2.1. Choose Which Tracks to Export

By default, **Ardour** will export all audio in the range or session being exported. What it actually exports is all audio routed through the master output bus. You can see the list of tracks to export on the right side of the "Export" window. If you click the 'Specific Tracks' button, you will be able to choose from a list of all the tracks and busses in a session. Choosing specific tracks only makes sense if you do not want to export the master bus' output, so you should probably de-select that first.

7.6.2.2. Choose the Export Format

Ardour offers quite a variety of output formats, and knowing which to choose can be baffling. Not all options are available with all file types. Fedora Linux does not support MP3 files by default, for legal reasons. For more information, refer to *MP3 (Fedora Project Wiki)* <http://fedoraproject.org/wiki/Multimedia/MP3>.

The tutorial's regions have 24-bit samples, recorded at a 48 kHz rate. Exporting any part of the session with a higher sample format or sample rate is likely to result in decreased audio quality.

Recommended File Types:

- WAV: An uncompressed format designed by Microsoft. Recommended only if further audio manipulation is intended. Carries only audio data, so information like title, artist, and composer will be lost. Playable with almost any device.

- **AIFF:** An uncompressed format designed by Apple. Recommended only if further audio manipulation is intended. Carries only audio data, so information like title, artist, and composer will be lost. Playable with almost any DAW and some audio players.
- **FLAC:** An open-source compressed format. A "lossless" format, meaning no audio information is lost during compression and decompression. Audio quality is equal to WAV or AIFF formats. Capable of carrying metadata, so information like title, artist, and composer will be preserved. Widely supported in Linux by default. For other popular operating systems, refer to *Download Extras (FLAC Website)* at <http://flac.sourceforge.net/download.html#extras> for a list of applications and programs capable of playing FLAC files. This is usually the best choice for distributing high-quality audio to listeners.
- **Ogg/Vorbis:** An open-source compressed format. A "lossy" format, meaning some audio information is lost during compression and decompression. Audio quality is less than WAV or AIFF formats, but usually better than MP3. Capable of carrying metadata, so information like title, artist, and composer will be preserved. Widely supported in Linux by default. For other popular operating systems, following the instructions on the *Vorbis Website* <http://www.vorbis.com/>. This is a good choice for distributing good-quality audio to listeners.

A higher setting for the sample format (explained in [Section 1.3.2, "Sample Format"](#)) allows a greater amount of audio information to be stored per sample. 32 bit support is virtually non-existent, but and you will probably not need to use this format in the near future. The "float" format stores samples in a different internal format, and you will need it only rarely.

If you are exporting audio for high-end equipment, or for further processing, choose the 24-bit format. Otherwise, choose the 16-bit format, which is the sample format of audio CDs.

"Sample endianness" is a difficult concept to understand, and it has no effect on the resulting audio - just how it is stored.. Unless you are using a rare PowerPC computer, choose the "Little-endian (Intel)" option.

A higher sample rate (explained in [Section 1.3.3, "Sample Rate"](#)) allows a greater amount of audio information to be stored, but increases the size of audio files.

"Conversion quality" and "dither type" are not available options for the file formats offered in Fedora Linux.

The "CD Marker File Type" allows you to export a CUE- or TOC-format list of CD tracks in the exported file. This is most useful when exporting a whole session, which contains a whole CD, that would be subsequently burned to disc.

Qtractor

Qtractor is a relatively new application, created and maintained by the same developers who are responsible for QjackCtl and Qsynth (both covered in other chapters of this Guide). It offers much more flexibility than Audacity, but is still easier to use than Ardour or Rosegarden. As such, it serves as the perfect starting-point for people first discovering software-based DAWs.

But Qtractor is much more than just a starting-point: its simplicity is its greatest strength. Ardour and Rosegarden, may offer more features, but Qtractor takes much less time to learn. After the initial learning-curve, you will be able to complete almost every audio or MIDI project with Qtractor. Its interface offers simple, intuitive, point-and-click interaction with clips, integrated control of JACK connections, MIDI control integration with external devices and other MIDI-aware software, and support for LADSPA, DSSI, native VSTi, and LV2 plug-ins. With development progressing very quickly, Qtractor is becoming more stable and usable by the minute. The simple interface allows you to focus on creating music to suit your creative needs.

Beginners and advanced users alike will be pleased to see how Qtractor can work for them.

8.1. Requirements and Installation

8.1.1. Knowledge Requirements

Qtractor is easy to use, and its user interface is similar to other DAWs. We recommend that you read [Section 6.4, “User Interface”](#) if you have not used a DAW before.

8.1.2. Software Requirements

Qtractor uses the JACK Audio Connection Kit. You should install JACK before installing Qtractor. See [Section 2.3.1, “Installing and Configuring JACK”](#) for instructions to install JACK.

8.1.3. Hardware Requirements

You need an audio interface to use Qtractor. If you will record audio with Qtractor, you must have at least one microphone connected to your audio interface. You do not need a microphone to record audio signals from other JACK-aware programs like **FluidSynth** and **SuperCollider**.

8.1.4. Other Requirements

You need a MIDI synthesizer to use Qtractor as a MIDI sequencer. You can use hardware-based and software-based synthesizers with Qtractor. We recommend using the software-based **FluidSynth** MIDI synthesizer. See [Chapter 10, FluidSynth](#) for information about **FluidSynth**.

8.1.5. Installation

Qtractor is not available from the Fedora software repositories. Qtractor is available from the "Planet CCRMA at Home" and "RPM Fusion" repositories. If you have already enabled one of those repositories, you should install Qtractor from that repository. If you have not already enabled one of those repositories, we recommend that you install Qtractor from the "Planet CCRMA at Home" repository. See [Section 4.3.1, “Installing the Planet CCRMA at Home Repositories”](#) for instructions to enable the "Planet CCRMA at Home" repository. The "Planet CCRMA at Home" repository contains a wide variety of music and audio applications.

After you enable the "RPM Fusion" or "Planet CCRMA at Home" repository, use PackageKit or KPackageKit to install the "qtractor" package. Other required software is installed automatically.

8.2. Configuration

Qtractor will work by itself, without further configuration. The options described here are for the settings you are most likely to want to discuss. Click on 'View > Options' to open the "Options" window.

8.2.1. Audio Options

The "Capture/Export" setting allows you to choose the format in which Qtractor stores its audio clips when recorded or exported. You will be able to choose a file type, such as "WAV Microsoft" for standard ".wav" files, "AIFF Apple-SGI" for standard ".aiff" files, or the preferable "FLAC Lossless Audio Codec," format. FLAC is an open-source, lossless, compressed format for storing audio signals and metadata. See the *FLAC Website* <http://flac.sourceforge.net/> for more information. You will also be asked to select a quality setting for lossy compressed formats, or a sample format for all lossless formats. If you do not know which sample format to choose, then "Signed 16-Bit" is a good choice for almost all uses, and will provide you with CD-quality audio. Most non-speciality hardware is incapable of making good use of higher sample formats. See *Section 1.3, "Sample, Sample Rate, Sample Format, and Bit Rate"* for more information about sample formats.

Setting the "Transport mode" will allow you to adjust the behaviour of the transport.

- None : allows Qtractor's transport to operate independently
- Slave : allows Qtractor's transport to accept instructions sent by JACK's transport, which can be controlled by QjackCtl or another application.
- Master : allows Qtractor's transport to send instructions to JACK's transport, which can be viewed by QjackCtl, or used by another application.
- Full : is equivalent to "Master" and "Slave" modes simultaneously; Qtractor's transport will both send and accept instructions.

If you are using Qtractor alone, or if you don't know which to choose, then "None" is a good choice. This setting can be adjusted at any time, if you later decide to link the transport in multiple applications.

The "Metronome" section allows you to use a (short) audio file as the metronome sound, rather than the standard, MIDI-based metronome. You can choose the same file for "beat," and "bar," if you prefer. The "Dedicated audio metronome outputs" option outputs the audio metronome's signal through separate outputs in JACK. This is Ardour's default behaviour.

8.2.2. MIDI Options

Adjusting the "File format" allows you to change how MIDI clips are stored. You will not need to adjust this unless required by an external application. Refer to *Musical Instrument Digital Interface: Standard MIDI (.mid or .smf)* (Wikipedia) at http://en.wikipedia.org/wiki/Musical_Instrument_Digital_Interface#Standard_MIDI_.28.mid_or_.smf.29 for more information about file types.

"MMC" stands for "MIDI Machine Control," and it allows multiple MIDI-connected devices to interact and control each other. Setting the "Transport mode" to a setting other than "None" allows it be controlled by MMC messages.

- None : will have Qtractor ignore incoming MMC messages, and not provide outgoing MMC messages
- Input : will have Qtractor follow incoming MMC instructions, but not provide outgoing MMC messages
- Output : will have Qtractor ignore incoming MMS messages, but provide outgoing MMC messages

- Duplex : will have Qtractor follow incoming MMC instructions, and provide outgoing MMC messages

You can also select a particular MIDI device number with which Qtractor will interact; if you do this, it will ignore MMC messages from other devices, and not send MMC messages to other devices. Enabled the "Dedicated MIDI control input/output" will provide JACK with MIDI inputs and outputs that will be used by Qtractor only for MMC messages. Qtractor will not send or receive MMC messages sent on other inputs or outputs if this option is enabled. "SPP" stands for "Song Position Pointer," and helps MIDI-connected applications to keep track of the current location in a session (in other words, where the transport is). This should probably be set to the same setting as "MMC." If you don't know which of these settings to use, then setting "MMC" to "None" is a good choice. This setting can be adjusted at any time, if you later decide to link applications with MMC messages.

The default metronome sound is provided by the "MIDI Metronome," and you can adjust its settings here. In particular, you may wish to provide a "Dedicated MIDI metronome output," to help you separate the metronome's signal.

8.2.3. Configuring MIDI Channel Names

If you're using "**FluidSynth**" with QSynth, you should tell Qtractor about the SoundFont that you're using. When you do this, you enable Qtractor to help you choose instruments ("patches").

1. Click on 'View > Instruments'
2. In the "Instruments" window, click "Import."
3. Navigate to `/usr/share/soundfonts`
4. Select the SoundFonts that you would like to use. The default is the FluidR3 GM SoundFont, but you can choose multiple SoundFonts to use simultaneously.
5. Close the "Instruments" window to return to the main Qtractor window.

8.3. Using Qtractor

The "Tutorial" section teaches you how to use Qtractor by example. This section is designed to serve as a reference while you complete the tutorial, and for refreshing your memory afterwards.

8.3.1. Using the Blue Place-Markers

In addition to the transport, Qtractor provides two other place-markers. They are blue. Here are some things you can do with the blue place-markers:

- * Mark a range:
 1. Move the cursor to the place where you want the left-most marker.
 2. Left-click and hold.
 3. Drag the cursor to the location where you want the right-most marker.
 4. Release the button.
 5. The blue markers should mark the particular range.
- * Mark one particular place:
 1. Move the cursor to the place you want to mark.
 2. Left-click and hold on the location.
 3. Drag the cursor to the right just a little bit.

4. Two blue markers will appear.
 5. Instead of leaving them separate, move the second blue marker over top the first one.
 6. Release the mouse button.
 7. The arrows should be converged.
- * Move one of the markers:
 1. Place the cursor over the triangular top of the marker, located in ruler at the top of the track pane.
 2. The cursor should change to show a double-ended, horizontal arrow.
 3. Left-click and drag the mouse.
 4. The blue place-marker should follow the mouse. If it doesn't, then try again. It is sometimes difficult to select the place-marker.

8.3.2. Using the MIDI Matrix Editor's Tools

Qtractor's matrix editor for MIDI clips offers a collection of useful tools, available from the "Tools" menu in the matrix editor window. These tools only operate on MIDI notes selected in the matrix editor window before the tool window is opened. Regardless of which tool you select from the menu, the same options are available to you each time the window is opened. It is possible to use multiple tools at a time.

Be careful: the names of the tools may be misleading.

Here is an explanation of what the tools do:

- Quantize: This tool can be used to regulate how often MIDI notes are allowed to begin, and how long they are allowed to last. They are marked in divisions of a beat.
 - Time: The default setting, "Beat/4" will allow notes to begin only on every sixteenth note subdivision. If a note begins somewhere else, it will be moved to the nearest sixteenth note subdivision. For example, a note that begins on the first 32nd note of a measure would be moved to begin on the first beat instead.
- Duration: The default setting, "Beat/2" will allow notes to last only in multiples of eighth notes. If a note lasts for shorter or longer than an eighth note multiple, it will be adjusted to the closest duration that is an eighth note multiple. For example, a sixteenth note would be adjusted to an eighth note.
- Swing: This regularizes a "swing" effect on the rhythm. You will have to experiment with the settings to find one that works for you.
- Common settings, described in simple meter where a quarter note gets the beat:
 - Beat : quarter-note duration
 - Beat/2 : eighth-note duration
 - Beat/3 : eighth-note triplet duration
 - Beat/4 : sixteenth-note duration
 - Beat/6 : sixteenth-note sextuplet duration
 - Beat/8 : thirty-second note duration

- **Transpose:** This tool adjusts either the pitch (vertical axis) or the time (horizontal axis) of the selected MIDI notes.
 - **Pitch:** Transposes the notes by this many semitones. Positive values transpose to a higher pitch-level, while negative values transpose to a lower pitch-level.
 - **Time:** Adjusts the starting time of MIDI notes, without affecting duration or pitch. This is most usefully measured as "BBT" (meaning "Bars, Beats, and Ticks" - each is separated by a decimal point), but can also be measured as time or frames.
- **Normalize:** This adjusts the loudness of the selected MIDI notes, called "velocity." There are three ways to use the tool. The value specified in the "MIDI Tools" window is used as the highest allowable velocity; all other velocity values are adjusted proportionally.
 - **Adjust "Percent" only:** This allows you to set the velocity to a percentage of the maximum velocity.
 - **Adjust "Value" only:** This allows you to supply a specific value for the velocity. Possible values range from 0 to 127.
 - **Adjust "Percent" and "Value" together:** This allows you to specify a percentage of the indicated value. If you set "50%" and "32," for example, the resulting velocity will be "16," which is 50% of 32.
- **Randomize:** This tool adjusts the selected parameters to pseudo-random values. The values are only pseudo-random for two reasons: computers cannot produce truly random numbers, only numbers that seem random to humans; the percentage value allows you to specify how widely-varied the results will be. A lower percentage setting will result in MIDI notes that are more similar to the pre-randomized state than if the MIDI notes were randomized with a higher percentage setting. The following parameters can be randomized:
 - **Note**, which means pitch.
 - **Time**, which means the time of the beginning of the note.
 - **Duration**, which means duration.
 - **Value**, which means loudness (velocity).
- **Resize:** This tool allows you to explicitly specify the duration or velocity of some MIDI notes. Setting the "Value" field will set the velocity (loudness) of all selected notes to that setting. Valid settings are from 0 (quietest) to 127 (loudest). Setting the "Duration" field will set the duration (length) of all selected notes to that setting. Duration is most usefully measured as "BBT" (meaning "Bars, Beats, and Ticks" - each is separated by a decimal point), but can also be measured as time or frames.

8.3.3. Using JACK with Qtractor

Qtractor and QjackCtl are programmed and maintained by the same developers. For this reason, Qtractor offers a QjackCtl-like interface to modify JACK's connections. Furthermore, Qtractor preserves all of the connections with every Qtractor file, so there is no need to use QjackCtl's "patch bay" feature (which does the same thing). However, if you wish to change JACK's settings, you will need to use QjackCtl.

To view the JACK connections window within Qtractor, press the F8 key, or click 'View > Connections'. Pressing F8 again will hide the window. Qtractor's "MIDI" tab displays the devices on QjackCtl's "ALSA" tab.

Qtractor automatically starts JACK, unless it is already running.

8.3.4. Exporting a Whole File (Audio and MIDI Together)

Qtractor can export all of a session's audio clips as one audio file, but it cannot export the MIDI clips directly into that audio file. This is because Qtractor does not synthesize audio from MIDI signals, but uses an external MIDI synthesizer to do this. Thankfully, there is a relatively simple way to overcome this, allowing both audio and MIDI to be exported in the same audio file: use Qtractor to record the audio signal produced by the MIDI synthesizer. This procedure only works if you use a MIDI synthesizer (like **FluidSynth**) which outputs its audio signal to JACK.

1. Create a new audio track in Qtractor by clicking on 'Track > Add Track'.
2. Ensure that your MIDI synthesizer is set up correctly to produce output in the usual method (through your speakers).
3. Use Qtractor's "Connections" window (press F8 on the keyboard) to ensure that the output from your MIDI synthesizer is routed to the input for the newly-created audio track.
4. Move Qtractor's transport to the start of the session.
5. Ensure that only the newly-created audio track is armed for recording.
6. Arm Qtractor's transport for recording.
7. Press "Play" on the transport, and wait as the session plays through.
8. When you have reached the end of the session, stop the transport. Qtractor will not automatically stop the transport.
9. Export the file as you normally would by clicking 'Track > Export Tracks > Audio'.

After the audio version of the MIDI signal is recorded, and you have exported the session, you may wish to delete the new audio track.

8.3.5. Miscellaneous Tips

If you wish to scroll horizontally in Qtractor's tracks pane or the matrix editor, hold down the Control (Ctrl) key as you adjust your mouse's scrollwheel.

Qtractor files should be saved in an otherwise-empty folder. Unlike Ardour, Audacity, and Rosegarden, Qtractor will not create a new sub-folder automatically, but will instead store all of a session's clips in the same folder as the the Qtractor file itself.

8.3.5.1. Transport

You can move Qtractor's transport to a particular point in a session by holding Shift as you use the mouse to click on that point.

You can choose whether or not you want the track pane to automatically scroll as the transport moves by clicking 'Transport > Follow Playhead'.

8.3.5.2. MIDI

When creating a MIDI track, you can use the "omni" check-box to allow the track to respond to input from any MIDI channel. If the check-box is unselected, the track will respond only to signals on its assigned MIDI channel.

In the matrix editor window, you can adjust the "velocity" (loudness) of a note by using the "Resize" MIDI Tool (see [Section 8.3.2, "Using the MIDI Matrix Editor's Tools"](#) above)

If you find it difficult to work with Qtractor's matrix editor, but you find it easy to work with LilyPond, you can use this to your advantage. LilyPond will output a MIDI-format representation of your score if you include a "midi" section in the "score" section. It should look something like this:

```
\score
{
  ...
  \midi { }
}
```

You can import LilyPond's MIDI output by clicking 'Track > Import Tracks > MIDI' in Qtractor.

8.4. Creating a MIDI Composition (Tutorial)

We've created a demonstration of what a first-time user might try for their first project with Qtractor. The following sequences demonstrate the decision-making, and the various features that could be learned. This does not attempt to show a generic method for creation, but rather the specific way that I created a new composition with the inspiration stated below.

8.4.1. Inspiration

The goal of this demonstration is to illustrate a particular strength of Qtractor: combining audio and MIDI tracks. I decided to start with a portion of one of my favourite compositions, and to compose a MIDI-based "alter-ego" to go along with it. The piece is listed below in "Requirements."

Since that particular movement is a "theme and variations" movement, it starts with a theme, then continues with varied versions of that theme. The theme is in two parts, each of which is repeated once. Beethoven uses several compositional techniques that are typical of his time period, and achieves a consistently similar, but consistently new, movement.

We are no longer bound by the aesthetic rules of Beethoven's time. We are also using a very different style of notation with Qtractor - the matrix editor does not even resemble standard Western musical notation.

Another interesting aspect of this piece is that, unless you have access to the same audio recording that I used, you will not be able to experience the piece as I do. Playing the MIDI alone gives a completely different experience, and it is one that I knew would happen. This sort of "mix-and-match" approach to music-listening is more common than you might think, but rarely is it done in such an active way; normally, the "extra sound" of listening to music is provided by traffic, machines like furnaces and microwave ovens, and even people in a concert hall or auditorium with you. The fact that my audio files cannot be legally re-distributed forced me to add a conscious creative decision into every listening of the piece.

8.4.2. Files for the Tutorial

- A recording of the second movement from Beethoven's Piano Sonata No.23, "Appassionata," either:
 - *Mutopia* at <http://www.mutopiaproject.org/cgi-bin/make-table.cgi?searchingfor=appassionata> (MIDI synthesizer recording with LilyPond sheet music)
 - *MusOpen* at <http://www.musopen.com/music.php?type=piece&id=309> (live recording)
 - The recording I used, played by Rudolf Serkin, available on the "Sony" label.
- You need to use FluidSynth, covered in [Chapter 10, FluidSynth](#).

8.4.3. Getting Qtractor Ready

1. Open QjackCtl, and start JACK.
2. Open Qsynth, and configure it with one instance, using the default FluidR3 SoundFont.
3. Open Qtractor
4. Configure the MIDI tracks to cooperate with the default FluidR3 SoundFont.
5. Switch to QjackCtl and ensure the proper connections:
 - Qtractor's MIDI out (on "ALSA" page) is connected to Qsynth's MIDI in
 - Qsynth's audio out is connected to system in (the speakers)
 - Qtractor's audio out is connected to system in (the speakers)
 - No other connections are necessary.

8.4.4. Import the Audio File

1. Create a new audio track.
2. Right-click on the audio track, and go to "Clip" then "Import"
3. Locate the audio file that you want to import (in this case, I imported a recording of the second movement of Beethoven's Op.57 piano sonata, "Appassionata."
4. If the clip doesn't start at the beginning of the track, then click and drag it to the beginning.

8.4.5. Marking the First Formal Area

In addition to the transport, Qtractor has two blue place-markers, which sometimes merge into one. The best way to learn the behaviour of the blue place-markers is by using them. They are intended to mark a range in the work area ("main screen").

1. We're keeping the standard tempo of 120 beats per minute, and the metre is 4/4
2. Start from the beginning, and listen until the end of the first formal section, which I've decided is about the fourth beat of measure 12 (use the ruler to see).
3. Mark that point with the blue arrow.
4. Mark the beginning of the formal area by left-clicking at the beginning of the session. If the transport is at the beginning, then it will hide a blue marker placed at the beginning.
5. Create a new clip by clicking on "Edit > Clip > New"
6. The clip editor will appear.

8.4.6. Creating our Theme

I want something simple, to match the simple-sounding chorale at the beginning that is the theme of this movement. What could be simpler than a moment of sound, followed by some moments of silence?

1. In the MIDI matrix editor window, click on the "Edit mode" tool, which looks like a pencil with no other markings.

2. I decided to place a note every three beats (one beat is marked by one vertical line), on the beat, lasting for one sixteenth note.
 - a. Click the pencil where you want the note to appear. A box will appear. If you drag the box to the right, then the note will sound for longer.
 - b. I put all of the notes on the same pitch, but it doesn't matter whether you put them on the same pitches or not - they will be changed later.
 - c. I also made a mistake when I was inputting the notes, so there's one place where they are only two beats apart instead of three. This didn't matter to me, but it might matter to you.
 - d. Continue inputting notes until you have filled the whole pre-selected region (between the blue markers). Qtractor will let you continue beyond that point, so you need to keep an eye on the marker yourself.
 - e. To scroll sideways, you can hold down either Shift or Ctrl and use your mouse's scroll wheel.
3. Move to transport just before the end of the segment you added: use Shift-click.
4. Listen to the end to ensure that your segment ends with or before the end of the first formal area.
5. Close the matrix editor window.
6. Use the main window to view the MIDI segment which you just inputted. The vertical lines represent barlines, and the darker rectangles represent notes.
7. If the MIDI segment extends beyond the last note that you inputted, click-and-drag the end so that there isn't much over-hang. If you accidentally adjust it too far and remove notes, then simply drag the segment back out - the notes should still be there.
8. Return to the matrix editor by double-clicking on the MIDI segment.
9. Select all of the notes that you have inputted so far:
 - Press Control-a
 - Click-and-drag to select, or
 - Click on 'Edit > Select > Select All'
10. Randomize the pitches:
 - a. Click on 'Tools > Randomize'
 - b. Ensure that "Randomize" is checked
 - c. Ensure that "Note" is checked (this means "pitch")
 - d. Choose a percentage.
 - e. Click "OK" to apply.
 - f. You may need to experiment with the percent of randomization that you allow. Greater randomization means a lower chance of repetition, but it also means that the pitches will be spread within a smaller range.
 - g. If you want to re-try the randomization, click on 'Edit > Undo Randomization', then use the "Randomize" tool again.

- h. If you like what happens to most of the pitches, you can select and move a few of them either individually or together. To adjust pitches as a group, select the ones that you want to move (either by click-and-drag select or by Control-click select), and Control-drag them to the desired new location.
11. Now you need to adjust the volume of the pitches. There are two ways to do this:
 - You can select all of the pitches, then use the "Resize MIDI" tool, adjusting the "Value" property.
 - You can select all of the pitches, then use the value editor portion of the matrix editor window. This is at the bottom of the matrix editor window, and the height of each pitch shown here tells you the volume at which it will play. To adjust all of the pitches at once, Control-drag to adjust the height as desired. Be careful when doing this that you don't change the horizontal position, which will change the time that the notes sound.
 - I would suggest a volume of approximately "32," but this depends on your taste. Also, I adjust the volume of some pitches to be louder when the audio file is louder.
12. When you are satisfied with your pitches and volumes, start the transport from the beginning, and listen to the entire segment that you just created. You can change the section again or move on to the next step.

8.4.7. Repeat the Theme

The beginning of this composition opens with a thirteen-measure (in this Qtractor session) segment that is immediately repeated. If you chose to create a thirteen-measure theme, like we did, you will either need to create a second MIDI segment to cover Beethoven's repeat, or you can do what I did, and copy-and-paste to get an exact repetition of your theme.

To repeat your theme exactly:

1. Click on the first MIDI segment to select it, then copy it by clicking on 'Edit > Copy' or Control-c
2. Paste it by clicking on 'Edit > Paste' or Control-v
3. The cursor will turn into a clipboard icon, and a rectangle will appear to its right. This rectangle represents the clip that is going to be pasted, but first you must select a place to put it.
 - a. Move the cursor so that the rectangle appears just after the end (right-most edge) of the first MIDI clip.
 - b. You can use the scrollbar arrows to scroll the main window, but it can be difficult, because the cursor has changed.
 - You can also scroll the main window by pressing Control and using your mouse's scroll wheel.
 - You can also scroll by carefully moving the mouse cursor to the edge of the main-part-thing.
 - c. It is not important to get the rectangle exactly where it will stay, but just near where it needs to go.
 - d. Left-click when you have placed the rectangle where you want it.
4. Position the transport so that you can listen to the transition from the end of the first clip into the beginning of the second clip. Press **Play** on the transport control to listen, then stop it when you are done.
5. If you need to adjust the position of the second clip, then click-and-drag it into the desired position. Re-listen to verify that you placed the clip where you want it to stay.

8.4.8. Compose the Next Part

It's difficult to explain, but this part feels more chord-focussed to me, even though it's very similar to the first part. I decided to show this by using the same generative idea as the first part, but with two simultaneous pitches instead of one. At the end of the segment, I included a brief gathering of "randomized" pitches, with longer durations than before. There is no particular reason that I included this chord-like incident, but it felt like the right thing to do.

1. Listen to the next portion of the audio file, and mark off the next formal section that you want to use (mine is from Qtractor's measure 25 to beat 2 of measure 38). The portion that I chose is also repeated, like the first part.
2. Place the blue markers at the beginning and end of the segment that you chose.
3. Create a new MIDI clip.
4. Create the notes separated by three beats, as in the last segment. This time, be sure to add two notes at the same time, by ensure that they are aligned vertically. Again, it doesn't matter which pitches you choose, because they will be randomized.
5. Select all of the pitches, and randomize them by using the "Randomize MIDI" tool.
6. Depending on how the pitches are randomized, each pair of notes will probably end up in one of the following situations:
 - They are too close or share a particular intervallic relationship that makes them sound like one note.
 - They are too far or share a particular intervallic relationship that makes them sound like two notes.
 - They share a particular intervallic relationship that makes them sound like one chord built of two equal chords.Depending on your aesthetic preferences, you may wish to change some of the notes so that one or some of these situations are avoided.
7. I created the tone cluster at the end by clicking arbitrarily across bar 37. It happened to create six notes with the pitches G, E, C-sharp, F-sharp, G-sharp, and B. I could have used the "Randomize MIDI" tool, but chose not to.
8. Then I carefully click-and-dragged the right-most end-point of each pitch, so that they all ended at the same time: the first beat of measure 38.
9. When you're done, you may need to copy-and-paste the segment.

8.4.9. Qtractor's Measures 52 to 75

You already know everything that you need to create this segment, so I will simply explain the artistic reasoning.

This corresponds to the "first variation" in the audio file. Since variations are based on the theme, the rest of my sections are all somehow based on my theme section. Here, I derived inspiration from the music again: there is a note (generally) every three beats like the theme, but I extended it to take up two beats, at the end of which another note briefly sounds. This is like Beethoven's technique in the first variation. Although I ignored them in the theme, there are small transitions between the inner-sections of Beethoven's theme, and I chose to add them into my first variation (you can see it in Qtractor's measure 69).

8.4.10. Qtractor's Measures 75 to 97

You already know everything that you need to create this segment, so I will simply explain the artistic reasoning.

This section corresponds to the part that we created in the "Compose the Next Part" section above. I decided to combine the idea of this first variation with the idea of that "Next Part." As you see, the result here is much like measures 52 to 75, but with more simultaneous pitches, as in the "Next Part."

At this point, my MIDI accompaniment really begins to take on its own rhythm and personality, competing with the audio file representing Beethoven's idea. Compared to the Beethoven, the randomized pitches of the MIDI part sound child-like and trivial. This might send listeners the message that MIDI is simply trivial and child-like, when compared to "real classical music," and this is a perfectly valid interpretation.

However, what I intended to communicate was this: Beethoven wrote a lot of piano music, much of which is still enjoyed by people today. Nobody will ever be able to re-create the magic of Beethoven, and I feel that it would be silly to try; this is why I let the music sound silly, rather than attempting to make it sound serious. I also feel that taking inspiration from composers such as Beethoven is an excellent way to create new art for ourselves, which is why I am deriving certain cues directly from the music (mostly vague stylistic ones), but ignoring others (like the idea that pitches should be somehow organized).

8.4.11. Qtractor's Measure 97

This is a three-beat transitional passage, which I added for no particular reason but to fill a pause in the audio track.

8.4.12. Qtractor's Measures 98 to 119

I used one new technique while composing this section: copy-and-paste within the matrix editor. You can see this around the beginning of measure 103, where the same pitch-classes are heard simultaneously in a high and low octave. I created the upper register first, then selected the notes that I wanted to copy. I used Control-c and Control-v to create the copy. Like when copy-and-pasting clips in the main window, the cursor icon changes to a clipboard, and an outline of the to-be-pasted material is shown so that you can position it as desired. As you will see, you can paste the copy onto any pitch level, and at any point in the measure. What is kept the same is the pitch intervals between notes and the rhythms between notes.

I also used the copy-and-paste technique with the three stepwise-descending-notes figure in this passage. After building the initial note of each set of four, I randomized those, and copy-and-pasted the three descending notes after. This way, I was able to randomize part of the melody, but avoid randomizing another part.

In this passage, I kept the "a note followed by three beats of rest" idea, then added onto the melody by taking two cues from the audio file. The first was the increasing surface rhythm of the upper part, which gave rise to the "three-descending-notes" figures. The second was the fact that the chords are still going on underneath that melody, so I added a second randomized set of notes underneath my upper part. At the end of the passage I continued the trend that I started with a finishing flourish that picks up sustained notes.

8.4.13. Qtractor's Measures 119 to 139

This passage does not introduce new techniques, but uses some trick manipulation of volume that are explained at the end of the section.

This passage sounds much busier because I increased the space between primary notes from three beats to two. I divided the octaves into four approximate ranges. The lowest has randomized pitches lasting one beat, which begin on beats that don't have a "primary note." There is no parallel in Beethoven's music at this point.

The next higher range is meant to mirror the melody in this part of the audio file, which is slightly lower than it was before. The highest range is connected to this, because Beethoven wrote some parts of the melody much higher than the other parts.

The second-highest range reflects the quickly-moving accompaniment part in the upper register of the piano.

Sorting out the volumes for this passage posed a small challenge, because of the much greater number of notes than in previous passages. Thankfully, the parts are mostly well-separated from each other in the matrix editor. I was able to use click-and-drag selection to select each range separately, and adjust its volume using both the "Resize MIDI" tool and Control-drag in the volume adjustment space at the bottom of the matrix editor window.

My sustained-note flourish at the end of this passage was a feeble attempt to establish A Major tonality in the highest register that I used.

8.4.14. Qtractor's Measures 139 to 158

There are no new techniques used in this section.

We maintained two-beat spacing of primary notes, and began with only two/three registers. The lowest register is just an occasional reinforcement of the uppermost, as in the audio file at this point. We used copy-and-pasting to create the quickly-moving middle line.

As the section progresses, the middle line gains a simultaneous addition. This eventually becomes more adventurous, at first jumping into a very high register, then leading downwards towards its place in the next section, in the lowest register.

8.4.15. Qtractor's Measures 158 to 176

There are no new techniques in this section. I made extensive use of copy-and-pasting, and especially of partial randomization: adding the first note of a flourish, randomizing it, then copy-and-pasting the rest of the flourish into place at the appropriate pitch-level.

At this point, I had basically dropped any obvious reference to my theme, as happens in the Beethoven score. Of course, its influence is still there: every four beats, my middle voice repeats the same pitches, and sustains them for the next four beats. Also, the upper voice in my part shares the same sort of "single repeated pitch" idea that makes up Beethoven's upper voice. There is also a weak rhythmic similarity between the two.

Near the end of the first sub-section (measures 164-166 inclusive), I included a long, downward 12-tone scale, which was inspired by the much smaller downward scale in Beethoven's piece.

The next sub-section is an exact repeat of ideas, but with different pitches, a smaller pitch range, and a slightly different ending.

8.4.16. Qtractor's Measures 177 to the End

There are no new techniques used in this section.

This part of piece was intended to mirror Beethoven's score quite obviously. The only real bit of trickery that I played was looking at Beethoven's score, and incorporating particular notes: the chord in

measure 212 is composed of the same pitches that are used in the chord in the audio file in measure 210. It sounds very different because of the "real piano vs. MIDI piano" issue, and because the tuning of the piano in the recording is different than the tuning of the MIDI piano. Also, the chord in the second beat of measure 213 is the first chord of the movement following the one recorded in the audio file. By including this (then "resolving" it, then re-introducing it), I intend to play with the expectations of a listener that may already be familiar with Beethoven's sonata.

DRAFT

Rosegarden

9.1. Requirements and Installation

9.1.1. Knowledge Requirements

Rosegarden's user interface is similar to other DAWs. We recommend that you read [Section 6.4, "User Interface"](#) if you have not used a DAW before.

9.1.2. Software Requirements

Rosegarden uses the **JACK Audio Connection Kit**. You should install **JACK** before installing **Rosegarden**. See [Section 2.3.1, "Installing and Configuring JACK"](#) for instructions to install **JACK**.

9.1.3. Hardware Requirements

You need an audio interface to use **Rosegarden**. If you will record audio with **Rosegarden**, you must have at least one microphone connected to your audio interface. You do not need a microphone to record audio signals from other **JACK**-aware programs like **FluidSynth** and **SuperCollider**.

9.1.4. Other Requirements

You need a MIDI synthesizer to use **Rosegarden** as a MIDI sequencer. You can use hardware-based and software-based synthesizers with **Rosegarden**. We recommend using the software-based **FluidSynth** MIDI synthesizer. See [Chapter 10, FluidSynth](#) for information about **FluidSynth**.

9.1.5. Installation

Use **PackageKit** or **KPackageKit** to install the *rosegarden4* package. Other required software is installed automatically.

9.2. Configuration

9.2.1. Setup JACK and Qsynth

1. Start **QjackCtl** to control **JACK**.
2. Start **Qsynth** to control **FluidSynth**.
3. In order to receive MIDI input from **Rosegarden**, **Qsynth** will need to be configured to use the "alsa_seq" MIDI Driver. Instructions for doing this can be found in [Section 10.4.4, "MIDI Input Configuration"](#).
4. You may want to disconnect all **JACK** connections except for those that you want to use with **Rosegarden**. Open **QjackCtl**'s "Connect" window, and verify the following:
 - On the "Audio" tab:
 - Qsynth's output ports are connected to the "system" input ports.
 - If you plan to use audio in **Rosegarden** (in addition to MIDI), then you will need to connect its output ports to the "system" input ports, too. The ports labeled "record monitor" are to be used to monitor audio while it is being recorded. The ports labeled "master out" will be used during

regular file playback. **Rosegarden** does not need to be connected directly to the system output ports if you are only using MIDI.

- If you plan to record audio in **Rosegarden**, then you will need to connect an output port (probably from "system") to **Rosegarden**'s input port. Be aware that **Rosegarden** can record from two independent sources ("1" and "2"), with two channels ("L" for left and "R" for right) from each, to produce a stereo recording.
- On the "MIDI" tab:
 - Nothing.
- On the "ALSA" tab:
 - a. Rosegarden's output ports must be connected to the "FLUID synth" input port:
 - 1:sync out (sends MIDI control messages)
 - 2:external controller (for the first set of MIDI instruments in the session)
 - 3:out 2 - General MIDI Device (for the second set of MIDI instruments in the session)
 - b. To make **Rosegarden** take commands from another MIDI device, you'll need to connect its output ports to **Rosegarden**'s input ports. I don't know what they are for yet:
 - 0:record in
 - 2:external controller

If a connection is not being used, it is better to leave it disconnected, to avoid making mistakes.

9.2.2. Setup Rosegarden

1. 'Edit > Preferences'
2. Setup "General" as desired.
 - On "Behaviour" tab maybe "Use **JACK** Transport"
 - On "Extrenal Applications" tab maybe change those to match what's installed (GNOME users)
3. Setup "MIDI" as desired.
 - On "MIDI Sync" tab maybe set to "Send MIDI Clock, Start and Stop" if **Rosegarden** is the ultimate controller, or "Accept Start, Stop and Continue" if it's being controlled. Otherwise "Off" is safe.
4. Setup "Audio" as desired.
 - The preview scale will not affect the audio, just its appearance.
 - Reducing quality from 32-bit to 16-bit may help low-power systems keep up.
 - Changing the external audio editor only affects when you choose to use an extrenal editor.

9.3. Rosegarden and LilyPond

Rosegarden and LilyPond can be used together, which can greatly enhance your productivity. LilyPond will output a MIDI-format representation of your score if you include a "midi" section in the "score" section. It should look something like this:

```
\score
{
```

```
...  
\midi { }  
}
```

This MIDI file can then be imported into **Rosegarden** by selecting from the menu 'File > Import > Import MIDI File'. Unfortunately, this will erase the current session, so it can be done only once.

It is also possible to export MIDI tracks to LilyPond format, which can then be edited further or simply turned into a score. To do this, from the menu select 'File > Export > Export LilyPond File'. After you select a location to save the file, **Rosegarden** allows you to control some score settings while exporting. After exporting a LilyPond file, you should inspect it with Frescobaldi before relying on it to be correct - computers sometimes make mistakes!

9.4. Write a Song in Rosegarden (Tutorial)

When using **Rosegarden**'s notation editor, the application may accidentally leave some notes playing longer than it should. You can fix this, if it bothers you, by pressing the "panic" button in **Qsynth**. If you are using another synthesizer with a panic button, it should server the same function.

9.4.1. Start the Score with a Bass Line

1. Start **QjackCtl**, then **Qsynth**, then **Rosegarden**.
2. For this tutorial, we'll be using the default "Fluid R3" SoundFont, but you can use any General MIDI SoundFont.
3. Setup **Rosegarden** in **JACK** for MIDI use only.
4. From the **Rosegarden** menu, select 'Edit > Preferences'. Click the "MIDI" tab, then the 'General' tab, then select "Send all MIDI controllers at start of each playback". This will ensure that the MIDI synthesizer (**FluidSynth** for this tutorial) uses the right patches.
5. Create a new segment.
 - a. Click on the "Draw" tool on the toolbar (it looks like a red pencil), or press 'F3' on the keyboard.
 - b. In the first track, click-and-drag to select the area over the first two bars.
 - c. When you release the mouse, there should be a rectangle that says, "Acoustic Grand Piano," or something similar.
6. Double-click on the rectangle to open the default MIDI segment editor. It should be the notation editor.
 - a. Change the clef. Click on the bass clef on the toolbar at the left side of the editor window. Then click on the existing clef to replace it.
 - b. Switch to the note-inputting tools as required. They're located next to the clef tools, in the toolbar at the left of the notation editor window.
 - c. Input three quarter-notes on c, two eighth notes on c and g, two quarter-notes on e-flat, and four eighth notes on d, f, d, and g.
7. You'll probably make a few mistakes. The easiest way to fix a mistake is to erase the note/s and re-input the correct note/s. You can use the eraser tool for this. It's located on the top toolbar of the score editing window, to the left of the capital "T," and to the right of the red pencil. It looks like a red and white rectangle, which represents a standard white eraser.

8. Listen to your creation by clicking on the 'play' button in the transport window. The playhead won't stop until you stop it, even though the visible playhead (the vertical line) will stop.
9. Close the notation editor - you don't need to click the 'save' button, because **Rosegarden** will automatically keep your changes.
10. You should save the file, though.
11. Click the "Select and Edit" tool, which looks like a mouse cursor on the toolbar. You can also select that tool by pressing 'F2' on the keyboard.
12. Select the segment you just created by clicking on it.
13. Create a copy of it by holding the 'Ctrl' key on the keyboard, as you click-and-drag the segment.
14. Place the copy in the same track, immediately after the first segment.
15. Create a few more copies like this.
16. Use the transport to move the playhead to the start (click the button that looks like a vertical bar with a single-direction arrowhead pointing towards it).
17. Name the newly-created track by double-clicking on its name. A window will pop up, allowing you to rename the track. If you don't know what to call it, try "Bass line."

9.4.2. Add a Percussion Track

1. Select the second track, and rename it to "Percussion" (or something else, if you prefer).
2. In the "Instrument Parameters" portion of the toolbox on the left side of the editor window, check the "Percussion" check-box.
3. Play the transport for just a second, so that **Rosegarden** synchronized the re-assignment with FluidSynth.
4. Press 'F3' on the keyboard, or click on the "Draw" tool, which looks like a pencil.
5. Create a one-measure segment in the second track, then right-click on the segment, and select 'Open in Percussion Matrix Editor'.
 - a. The percussion matrix editor is like the matrix editor, but each pitch is labeled with the instrument it triggers.
 - b. Experiment with the different instruments and pitches to find a pattern of four quarter notes that you want to repeat in each measure.
 - c. When you're happy with your pattern, close the editor.
6. Select the newly-created percussion segment, then from the menu select 'Edit > Copy'.
7. Move the transport's playhead to the end of the first measure. First move it to the beginning, then press the "fast forward" button to advance it to the end of the first measure.
8. Be sure that the second track is still selected.
9. Press 'Ctrl + v' or from the menu choose 'Edit > Paste' a few times, so that the percussion track fills up the same space as the bassline.

9.4.3. Spice up the Percussion

1. Four quarter notes isn't really a suitable percussion accompaniment, so let's make it more interesting.
2. Open the first percussion segment in the standard notation editor by double-clicking it.
3. From the menu select 'Edit > Select Whole Staff'.
4. Then from the menu select 'Adjust > Notes > Eighth Note'.
5. This compressed the notes, which isn't what you wanted. Undo the change with 'Edit > Undo'.
6. Again make sure that the whole staff is selected.
7. From the menu select 'Adjust > Notes > Eighth Note (Without Duration Change)'.
8. Now there are eighth-note rests between the original rhythm. You can add off-beat accents as you wish, in the space where the rests are. You can switch to the percussion matrix editor, or stick with the notation editor - it can be fun guessing which pitch represents which instrument.
9. Now that you've made a change, you'll have to delete all the copies and re-copy them. We'll use a better solution instead, but you'll still have to start by deleting all of the copies of the segment. Remember to keep the first one!
10. After removing all but the first segment, from the "Segment Parameters" box in the left toolbox, check the "Repeat" checkbox.
11. Now the segment repeats forever. This can be useful, but eventually you'll want to stop the song. When you want to change the repeats into copies, from the menu select 'Segment > Turn Repeats Into Copies'.
12. Adjust the first track so that it's repeated, too.

9.4.4. Add a Melody

1. Rename the third track to "Melody," or something like that.
2. Change the track's program.
 - a. Look at the "Instrument Parameters" box in the "Special Parameters" toolbox on the left side of the main editing window.
 - b. Make sure that you have the "Melody" track selected.
 - c. Select the "General MIDI" bank, which is probably already selected.
 - d. For the program, select whatever you prefer. I decided to use program 51, called "Synth Strings 1." It reminds me of music from the 1980s.
 - e. After setting the program, press 'play' on the transport toolbar, and let it go for just a second. This will allow **Rosegarden** to send the program-change message to FluidSynth. It isn't strictly necessary, and it would have been done later anyway, but doing it now helps to avoid confusion later.
3. Use the "Select and Edit" tool to create a segment in the "Melody" track, of four measures long.
4. Edit the segment in the default editor by double-clicking on it.
5. Create four measures of a melody. It doesn't have to be complicated, or even interesting.

6. Close the notation editor, and listen to the the three tracks. Don't forget to reset the playhead to the beginning of the session!
7. It sounds a bit silly to have the melody enter at the very beginning, so add an introduction. With the "Select and Edit" tool (press 'F2' on the keyboard to engage it), click-and-drag the melody segment to a few bars or beats later. Note that, even if you move it to start on beat 2, 3, or 4, the view in the notation editor will always start the segment on beat 1.

9.4.5. Possible Ways to Continue

You're on your way to a full MIDI composition. All you need is some inspiration to continue, and some willingness to experiment with more advanced features and tools. If you don't know how to continue, try these suggestions. Remember: you're just starting out, so your first song doesn't have to be interesting or particularly good. Once you learn how to use MIDI composition tools, you'll naturally learn how to create better music with them!

- Make a simple ternary form (ABA): you've already created the first part (called "A"); so make a transition and a second, different part (called "B"), then another transition and repeat the first part. It doesn't need to be long, but it can be.
- Make a "variations" form: repeat the part that you've already created several times, but make it slightly different every time. Try several different ways of modifying it: add another melody to go along with the one you have; add extra notes to the existing melody; change the percussion track; write a new bassline; expand the melody so it takes twice as long; combinations of these. Making a variation of an existing section is a common way of making it take more time.
- Make an expanded version of the existing material, by following your intuition to add more music.
- Instead of adding a different melody of a repeating bassline and percussion segment, try repeating the melody over and over, creating a new bassline and percussion segments.

FluidSynth

FluidSynth is a software-based MIDI synthesizer. **FluidSynth** accepts MIDI input from programs like Qtractor and Rosegarden, and uses SoundFont technology to create audio signals. This makes **FluidSynth** a very flexible tool; it can be used even on low-power computers, doesn't require specialized hardware, and can take advantage of a wide selection of high-quality MIDI instruments. When used with the **Qsynth** graphical interface, **FluidSynth** becomes even more powerful: users can easily control basic effects like chorus and reverb, and they can start multiple **FluidSynth** synthesizers, each with their own settings and MIDI instrument assignments. Finally, because **Qsynth** was created and is maintained by the same developers as Qtractor and QjackCtl, it provides a familiar interface, and integrates well with these other applications.

10.1. SoundFont Technology and MIDI

SoundFont technology was developed in the early 1990s, and comprises a file format and certain hardware technologies designed to allow the creation of MIDI instruments that sound like acoustic instruments. It would be virtually impossible to make an electronically-synthesized instrument sound identical to an acoustic counterpart, but SoundFont technology enables the gap to narrow considerably. Heard in the right context, most people would not notice that music was recorded by a SoundFont-capable MIDI synthesizer, but results can vary widely.

What **FluidSynth** enables users to do is eliminate the hardware component of using SoundFonts, so that any computer becomes capable of synthesizing from SoundFont files, which are often simply referred to as "a SoundFont." As fonts change the look of text characters, SoundFonts change the sound of MIDI notes - the overall meaning is the same when conveyed by any font (or SoundFont), but the particular nuance is changed.

Fedora offers a few SoundFonts in the default repositories. By default, **FluidSynth** installs the FluidR3 General MIDI ("GM") SoundFont, which contains a wide array of conventional (and some non-conventional) "patches." To see the other options that are available, use PackageKit, KPackageKit, or yum to search for "soundfont".

10.1.1. How to Get a SoundFont

There is a large selection of SoundFonts available for free on the internet, and some are also available for purchase, including a few very high quality SoundFonts. The following three websites have links to SoundFont resources, and some SoundFonts available for paid or free download. No guarantee is made of the quality of the material provided, or of the quality and security of the websites.

- *S. Christian Collins' "General User" SoundFont*, available from <http://www.schristiancollins.com/generaluser.php>.
- *HammerSound SoundFont Library*, available at <http://www.hammersound.net/cgi-bin/soundlink.pl>.
- *homemusician.net SoundFont Library*, available at <http://soundfonts.homemusician.net/>.
- *Synth Zone*, available at <http://www.synthzone.com/soundfont.htm>.

See the "Optional Installation: SoundFont ..." below for installation instructions.

10.1.2. MIDI Instruments, Banks, Programs, and Patches

A "MIDI instrument" is the synthesizer itself. If the synthesizer uses SoundFonts, then the SoundFont also constitutes part of the instrument. Each instrument can be thought of as a library, which stores books.

Each instrument offers at least one, but possibly several "banks," which store programs. If a MIDI instrument is a library, then a bank is like a particular shelf. You must first select a shelf before choosing a book.

Each bank offers between one and one hundred and twenty seven "programs," (also called "patches") which are the sounds themselves. If a MIDI instrument is a library and a bank is a shelf, then a program is a book. Programs need not necessarily be related, but banks with a large number of programs (like the "General MIDI" bank) usually follow some sort of order. It is the program alone which determines the sound of the synthesized audio; the bank and instrument simply limit the possible choices of program.

10.1.3. MIDI Channels

A MIDI synthesizer will accept input on multiple channels. Although each "instance" of the synthesizer can only have one MIDI instrument assigned to it, each channel can be assigned a program independently. This allows the synthesis of a virtual instrumental ensemble.

The General MIDI ("GM") standard, used partially by the default FluidR3 SoundFont and by **FluidSynth** itself, further specifies that there will be 16 channels, and that channel 10 will be used for (mostly unpitched) percussion instruments. Any program change message sent to channel 10 will be ignored, and although **FluidSynth** can be configured to use a non-percussion program on channel 10, this use is discouraged.

For cases where **FluidSynth** does not adhere to the General MIDI standard, it is adding functionality, rather than removing it.

10.2. Requirements and Installation

10.2.1. Software Requirements

FluidSynth requires the **JACK** Audio Connection Kit. If you have not already installed the **JACK** packages from the Planet CCRMA at Home repository, then it is recommended that you do so *before* installing **FluidSynth**. See [Section 2.3.1, "Installing and Configuring JACK"](#) for instructions.

10.2.2. There Are Two Ways to Install FluidSynth

There are two ways to install **FluidSynth**. The first, to install **FluidSynth** with **Qsynth**, allows **FluidSynth** to be used with the **Qsynth** graphical interface. It also installs a default SoundFont, which can be used by any SoundFont-aware application (like **timidity++**). This installation method does not - by default - allow **FluidSynth** to be used from a terminal, although this ability can be easily added later. This is the installation method recommended for most users.

The second way to install **FluidSynth** is without the **Qsynth** graphical interface. This method allows **FluidSynth** to be run from a terminal, and does not install a default SoundFont. This installation is recommended only for advanced users.

10.2.3. Installation with Qsynth

This installation method is recommended for most users, and will install everything you need to start using **FluidSynth**.

1. Use "PackageKit" or "KPackageKit" to install the "qsynth" package.
2. Review and approve the proposed installation:

-
- The installation may include the "fluid-soundfont-gm" package, which is quite large.

If you wish to use **FluidSynth** from a terminal, without the **Qsynth** graphical interface, you can enable this capability by installing the *fluidsynth* package. This package is not needed if you only intend to run **FluidSynth** with **Qsynth**, because **Qsynth** only uses files in the *fluidsynth-libs* package, which is automatically installed with **Qsynth**. If you are unsure of whether you should install the *fluidsynth* package, you can safely install it, even if you never use it. It only uses a small amount of hard drive space.

10.2.4. Installation without **Qsynth**

This installation method is recommended only for advanced users. You will have to use **FluidSynth** from a terminal. You will also have to install a SoundFont file before using **FluidSynth**.

Use **PackageKit** or **KPackageKit** to install the *fluidsynth* package.

10.2.5. Installation of SoundFont Files

Qsynth automatically installs a SoundFont for use with **FluidSynth**, but if you did not install **Qsynth**, or if you want to add additional SoundFont files with additional programs, you will need to install them separately. The Fedora package repositories offer a small selection of SoundFont files, which you can find by searching for "soundfont" with PackageKit, KPackageKit, or yum. These files will automatically be installed correctly. If you wish to install additional SoundFont files, it is recommended that you install them in the same location - and with the same security settings - as the ones available from the Fedora repositories. If you do this, then you enable all users of the computer system to access the files, you will not "lose" them if you forget where they are stored, and you help to minimize the potential security risk of using software downloaded from the internet.

The following steps move a SoundFont file called **myFont.sf2** to the default folder (**/usr/share/soundfonts**), and correctly set the security settings. Note that you will need the system administrator's password (belonging to the "root" account) to complete this operation. If you do not have this password, it is best to ask the system administrator to install the files for you. Alternately, you may simply use the SoundFont file from your a sub-folder in your home folder.

1. Start a shell or terminal and navigate to the folder where the SoundFont file is currently stored.
2. Run **su -c 'cp myFont.sf2 /usr/share/soundfonts'**
 - a. Modify the command as necessary to copy your SoundFont file, rather than **myFont.sf2**.
 - b. You will be asked for the password to the **root** account.
3. Run **cd /usr/share/soundfonts** to change to the directory of the SoundFont
4. Run **su -c 'chmod 644 myFont.sf2'**
 - a. Modify the command as necessary to refer to your SoundFont file, rather than **myFont.sf2**.
 - b. This will set the file-system permissions to "read-write" for the owner (the "root" user, in this case), and "read-only" for all other users. This way, only the system administrator should be able to change the file, but all users will be able to use it.
5. Run **ll myFont.sf2** to verify that the permissions were set correctly.
 - a. Modify the command as necessary to refer to your SoundFont file, rather than **myFont.sf2**.

- b. The output should resemble this:

```
-rw-r--r--. 1 root root 9 2010-06-23 02:28 myFont.sf2
```

but with a different date, time, and filename.

6. Highly-observant users may notice that the SELinux context of the new file is different from that of any Fedora-installed SoundFont file. As long as the type is **usr_t**, which it should be by default, then there is no practical difference (no difference in enforcement) between this and a Fedora-installed SoundFont file. If you don't know what this means, or if you hadn't noticed it, then it means that this additional SoundFont file should not create a new potential security problem.

10.3. Using FluidSynth in a Terminal

This is not the recommended way to use **FluidSynth**, because the **Qsynth** graphical interface is much easier to use. **Qsynth** automatically configures most of **FluidSynth**'s settings by default, allowing you to avoid focus on how you want to use **FluidSynth**, rather than on how to use **FluidSynth**.

If you want to use **FluidSynth** in a terminal, you can use the **fluidsynth** command. The default sample-rate is 44.1 kHz, so if you want to use **JACK** at a different sample rate, you need to use the **-r** flag, like this: **fluidsynth -r 48000**

When you start **FluidSynth** from a terminal, it will normally start a shell of its own. How to use this shell is beyond the scope of the Musicians' Guide, but you can get basic help by running the "help" command from the **FluidSynth** command line.

10.4. Configuring Qsynth

When you quit **Qsynth**, all settings are preserved, and re-used when **Qsynth** is re-started. This includes settings for additional instances of **Qsynth** (described below), which are also re-created when **Qsynth** is re-started.

10.4.1. Starting FluidSynth

1. Start **Qsynth** from the Applications menu, or the K Menu
2. The **FluidSynth** engine will be started automatically.
3. The row of buttons at the right of the **Qsynth** window control **Qsynth**. The other settings control **FluidSynth**.
4. You can use the "Messages" button to display a window containing **FluidSynth**'s output. If **FluidSynth** doesn't work as expected, you can use this window to view any error message that might have been produced.

10.4.2. SoundFont Configuration

The default "FluidR3" SoundFont, installed with **Qsynth**, is automatically configured.

To configure an additional Soundfont:

1. Click on the 'Open' button, and navigate to the path of the SoundFont you wish to add. This should be **/usr/share/soundfonts**, if installed to the standard location specified in [Section 10.2.5, "Installation of SoundFont Files"](#).

2. Select the additional SoundFont, then click the 'Open' button.
3. To change the SoundFont ID (SFID), use the 'Up' and 'Down' buttons to change the position of the SoundFonts, as desired. This does not directly change the function of **FluidSynth** - any SoundFont should work with any SoundFont ID number.

10.4.3. JACK Output Configuration

It is possible to configure **FluidSynth** to output synthesized audio either to **JACK** or to ALSA. The default, and recommended, method is to output synthesized audio to **JACK**. This allows the greatest control over audio quality, and the greatest flexibility in terms of routing and multiplexing (for definition see [Section 1.4.7, "Routing and Multiplexing"](#)), which allows you to simultaneously record the synthesized audio signal and listen to it.

If you are having problems, you may wish to confirm that **Qsynth** is configured correctly to use **JACK**.

1. Open **Qsynth**'s "Setup" window.
2. Select the 'Audio' tab, and ensure that "Audio Driver" is set to "jack".
3. You should also ensure that the other settings are correct - especially that the "sample rate" is set to the same sample rate as **JACK** (through **QjackCtl**).
4. The default settings for most things should work. If you changed a default setting, they are these:
 - Buffer Size: 1024
 - Buffer Count: 2
 - Audio Channels: 1
 - Audio Groups: 1
 - Polyphony: 256

If you are having problems with audio cut-outs, you may wish to increase the buffer settings. The size should be increased in multiples of 1024, and the buffer count should not be increased much. The default setting of one "Audio Channel" provides stereo output, and each additional channel produces another set of stereo outputs. Increasing the "polyphony" setting will allow a higher number of simultaneous notes ("MIDI events," really), which will be useful in extremely complex situations.

10.4.4. MIDI Input Configuration

FluidSynth will only produce sound as instructed by a connected (software- or hardware-based) MIDI device. If you are having problems configuring **FluidSynth** to accept MIDI input, verify the following options.

1. Open **Qsynth**'s "Setup" window.
2. The "Enable MIDI Input" setting must be enabled.
3. There are two settings for "MIDI Driver" that you would likely want.
 - When set to "alsa_seq", the input will appear on **QjackCtl**'s "ALSA" tab in the "Connect" window. This is useful if the MIDI generator device that you are using, such as a MIDI-enabled keyboard, is connected directly to ALSA.
 - When set to "jack", the input will appear on **QjackCtl**'s "MIDI" tab in the "Connect" window. This is useful if the MIDI generator device that you are using, such as **Rosegarden**, is connected directly to **JACK**.

4. You can set the number of MIDI input channels provided by **FluidSynth**. Refer to [Section 10.5.1](#), “*Changing the Number of MIDI Input Channels*” below.

10.4.5. Viewing all FluidSynth Settings

1. Open the "Setup" window
2. Select the "Settings" tab
3. Scroll through the list to the setting you wish to see.
4. The settings in this tab are not editable in this tab.

10.5. Assigning Programs to Channels with Qsynth

The best way to do this is through your MIDI sequencer, like **Qtractor** or **Rosegarden**. If you are using an application which doesn't allow you to send program-change messages, or if you need to configure programs for another reason, you can follow these instructions.

1. In the main **Qsynth** window, click "Channels" to open the "Channels" window.
2. In the Channels window, each channel will, by default, look like

1	-	-	-
---	---	---	---

3. To assign a program to a channel, click on the row of the channel that you want to assign.
4. Select the bank and program number that you want, using the name, SFID, and Soundfont columns to guide you.
5. Repeat this process for all of the channels that you wish to assign.
6. If you are not going to use a channel, there is no harm in assigning it a program anyway. However, assigning a program to a channel that you will not use can lead to confusion, especially if you later forget that you intended to not use that channel.
7. Remember that channel 10 is reserved for percussion instruments, and should not be changed from a General MIDI percussion program.

10.5.1. Changing the Number of MIDI Input Channels

You can increase the number of MIDI input channels offered by **FluidSynth**. Although **QSynth** will let you set any number between 1 and 256, channels will be added and removed only in sets of 16. **Qsynth** will automatically create the lowest number of channels that will allow as many channels as you need. Each set of 16 channels will be indicated in **JACK** with an additional MIDI input device. The name of the device indicates which channels are listening: channels 1 through 16 listen to the device ending in 0; channels 17 through 32 listen to the device ending in 1; channels 33 through 58 listen to the device ending in 2; and so on.

To change the number of MIDI input channels, follow these instructions.

1. Open the **Qsynth** "Setup" window.
2. The "Enable MIDI Input" setting must be enabled.
3. Input the number of channels that you wish to use, between 1 and 256.

10.5.2. Saving and Reusing Channel Assignments

QSynth allows you to save multiple sets of program assignments, which you can restore later. This means that you will not have to re-configure all of the channels every time you want to change them.

To save the settings for later:

1. Ensure that **Qsynth** is correctly configured, and working with the program assignments as desired.
2. In **Qsynth**'s "Channels" window, erase the contents of the "Preset Name" text field, and replace it with whatever name you would like.
3. Click "Save" to preserve the settings under that name.

To restore settings from earlier:

1. Open **Qsynth**'s "Channels" window.
2. Search through the "Preset Name" drop-down box to find the name of the preset that you wish to restore.
3. Select it, and verify that the correct channel assignments were restored.

10.6. Using Reverb and Chorus with Qsynth

While "reverb" (meaning "reverberation") and "chorus" effects are not part of the General MIDI standard, they are offered by most MIDI synthesizers. **FluidSynth** is no exception, and the **Qsynth** interface provides a convenient and easy way to adjust settings of the reverb and chorus effect-generators. Experimentation is the only way to know whether you have chosen the right settings, and they will probably change depending on the music you are working on, and even during a piece. Most MIDI sequencers and players (including Qtractor and Rosegarden) allow you to send MIDI messages changing the reverb and chorus settings while a session is playing.

The reverb and chorus effects can be turned off temporarily if you do not plan to use them. To do this in **Qsynth**, uncheck the "Active" check-box underneath the respective effect generator's settings dials.

The *reverb* effect generator creates artificial reverberation (like an "echo," but more complex), which occurs naturally in almost all performing environments. The effect generator works by creating a virtual room, and pretending that the user is listening within it. These are the settings offered by **Qsynth**:

- **Room:** Adjusts the size of the virtual room. Settings higher than 100 can cause a situation that escalates in volume to dangerously high levels, even with the lowest possible "level" settings. This is an interesting effect, but caution is advised so that listener do not incur accidental hearing damage. Remember also that the reverb effect can accumulate as time goes on, so it may even take many minutes for the volume level to build to a dangerously high level, depending on the settings.
- **Damp:** Adjusts the amount of "sound damping" in the virtual room, affecting not only the time it takes for the sound to die, but the frequencies which are most affected. The greater the damping, the shorter the higher frequencies last.
- **Width:** Adjusts the perceived "width" of the stereo image of the virtual room, and also has a large effect on the volume level.
- **Level:** Adjusts the volume level of the reverberation.

All of the settings interact in ways that make it difficult to describe any sort of recommended settings. Users are strongly encouraged to experiment with the various settings to find ones that suit their needs. A wide variety of listening environments can be simulated, and the settings required do not

always reflect the most logical choice - it is possible to emulate a concert hall with a relatively small "room" setting, for example. Effective use of the reverb effect can greatly enhance the MIDI listening experience.

The *chorus* effect generator creates the impression that more than one real-world instrument is playing each MIDI line. The effect generator works by slightly responding to each MIDI note several times, with slightly inaccuracies in pitch and time in each response. Although it may seem overly negative to say so, humans intuitively recognize that this inaccuracy is part of real-world, acoustic music, so the chorus effect helps to add a realistic sound to some performances. These are the settings offered by **Qsynth**:

- **N**: Adjusts the number of effect "stages," which allows you to control the approximate number of discrete instruments perceived.
- **Level**: Adjusts the volume level of the effect.
- **Speed**: Adjusts the speed of the chorus, and so the range of time over which the notes are spread.
- **Depth**: Adjusts the perceived "depth" of the stereo image.

10.7. Multiple FluidSynth Instances with Qsynth

Rarely will you need more than one instance of **FluidSynth**, because **QSynth/FluidSynth** together offer up to 256 independent input channels, and the ability to control the reverb and chorus effects independently by channel. The most common use of multiple **FluidSynth** instances is if you want to use multiple MIDI instruments. In other words, if you want to use multiple SoundFonts at the same time, you will need to use one instance of **FluidSynth** for each SoundFont.

Thankfully, **Qsynth** allows us to do this almost effortlessly! Each "instance" of **FluidSynth** is created and controlled by, in effect, running the **FluidSynth** synthesis application multiple times. The reality is that, since **Qsynth** controls the **FluidSynth** engine directly, it simply starts the synthesis engine directly, creating multiple instance of that engine. This is much more efficient than actually running **Qsynth** multiple time, or even than running **FluidSynth** directly from a terminal. Each such instance is represented in **Qsynth** by a "tab," displayed at the bottom of the **Qsynth** window.

To create an additional instance of **FluidSynth**:

1. Press the green "+" button in the bottom-left corner of **Qsynth**'s main window
2. Adjust the settings as desired, by using the Setup window that pops up.
3. Press "OK" to start the additional instance.
4. To close an additional instance, use the red "X" near the lower-right corner.

Each instance of the **FluidSynth** engine has its own settings in the "Setup" window. **Qsynth** supports a theoretically unlimited number of **FluidSynth** instances, but your computer's memory will probably not allow many more than ten, depending on the SoundFonts used.

SuperCollider

SuperCollider is many things, but above all:

- An audio synthesis engine,
- A flexible programming language, and
- An interpreter to transform the programming language into synthesis instructions.

11.1. Requirements and Installation

11.1.1. Knowledge Requirements

SuperCollider is by far the most difficult program described in the Fedora Musicians' Guide. The **SuperCollider** applications themselves are easy to use, and they work very well, but they are merely tools to help you accomplish something useful. **SuperCollider** has an extremely powerful and flexible programming language, with libraries designed primarily for audio processing. As often happens with computers, however, this added flexibility and power comes at the cost of requiring greater understanding and learning on the part of the user. Because **SuperCollider** involves actual programming, a rudimentary understanding of some principles and concepts of computer science will provide huge benefits to somebody learning the language. The following articles from Wikipedia are not mandatory reading, but you should refer to them as necessary while learning the language.

- *Computer Programming* at http://en.wikipedia.org/wiki/Computer_programming: You probably know what this is; it's what you'll be doing.
- *Programming Language* at http://en.wikipedia.org/wiki/Programming_language: **SuperCollider** is a programming language.
- *Interpreter* at http://en.wikipedia.org/wiki/Interpreter_%28computing%29: This reads your code, and sends commands to the server, which causes it to produce sound.
- *Server* at http://en.wikipedia.org/wiki/Server_%28computing%29: **SuperCollider** has a 'server' component, which is operated by the interpreter.
- *Functional Programming* at http://en.wikipedia.org/wiki/Functional_programming: **SuperCollider** can be treated as a "functional" language.
- *Imperative Programming* at http://en.wikipedia.org/wiki/Imperative_programming: **SuperCollider** can be treated as an "imperative" language.
- *Object-Oriented Programming* at http://en.wikipedia.org/wiki/Object-oriented_programming: **SuperCollider** can be treated as an "object-oriented" language.

11.1.2. Software Requirements

SuperCollider uses the JACK Audio Connection Kit. You should install JACK before installing **SuperCollider**. Refer to [Section 2.3.1, "Installing and Configuring JACK"](#) for instructions to install JACK.

SuperCollider is not available from the Fedora software repositories. You must enable the "Planet CCRMA at Home" repository to install **SuperCollider**. See [Section 4.3.1, "Installing the Planet CCRMA at Home Repositories"](#) for instructions to enable the "Planet CCRMA at Home" repository. The "Planet CCRMA at Home" repository contains a wide variety of music and audio applications.

11.1.3. Hardware Requirements

You need an audio interface to use **SuperCollider**. You do not need a microphone to use **SuperCollider**.

11.1.4. Available SuperCollider Packages

The **SuperCollider** packages are all held in the Planet CCRMA at Home repository, and there are a lot of them. Many of them have standard Fedora suffixes, but many are other kinds of optional components. Most of the optional features add libraries to **SuperCollider**, allowing you to use them in your audio programs. The specific features available in each additional package are not described here.

- *supercollider-ambiem* : Optional Library ("Ambisonics classes for SC").
- *supercollider-debuginfo* : Decodes the debugging information provided by **SuperCollider**.
- *supercollider-devel* : Contains files needed for development with **SuperCollider**.
- *supercollider-dewdrop* : Optional Library ("DewDrop external collection for SC").
- *supercollider-emacs* : Adds **SuperCollider** support to the **emacs** text editor.
- *supercollider-extras* : Optional Library ("Extra plugins and classes for SC").
- *supercollider-gedit* : Adds **SuperCollider** support to the **GEdit** text editor.
- *supercollider-libscsynth* : "**SuperCollider** synthesis library."
- *supercollider-quarks* : Optional Library ("Local quarks repository for **SuperCollider**").
- *supercollider-sclang* : Help files, examples, the class library, and language interpreter.
- *supercollider-vim* : Adds **SuperCollider** support to the **vim** text editor.
- *supercollider* : Installs the "minimum requirements" to run **SuperCollider**.
 - *supercollider-sclang*
 - *supercollider-libscsynth*
 - *fftw*
 - *w3m-el*
 - *emacs*
- *supercollider-bbcut2* : Optional Library ("Beat tracking of audio streams").
- *supercollider-bbcut2-debuginfo* : Decodes the debugging information provided by *bbcut2*.
- *supercollider-mathlib* : Optional Library ("Useful classes for SC").
- *supercollider-redclasses* : Optional Library ("Frederik Olofsson Red SC classes").
- *supercollider-redclasses-debuginfo* : Decodes the debugging information provided by *redclasses*.
- *supercollider-world* : Installs most **SuperCollider** packages.
 - *supercollider*
 - *abmiem*

- *supercollider-redclasses*
 - *supercollider-dewdrop*
 - *supercollider-emacs*
 - *supercollider-mathlib*
 - *supercollider-midifile*
 - *supercollider-extras*
 - *supercollider-bbcut2*
 - *supercollider-reduniverse*
- *supercollider-midifile* : Optional Library ("MIDI file reader for **SuperCollider**").
 - *supercollider-reduniverse* : Optional Library ("Sonification and visualization of dynamic systems").

11.1.5. Recommended Installation

If you have never used **SuperCollider** before, then we recommend installing the smallest number of packages possible. This will allow you to start learning with the core classes, available on all **SuperCollider** installations. Installing the bare minimum requirements will not prevent you from installing optional libraries in the future, of course.

The recommended installation also avoids installing the **emacs** or **vim** components, which - unless you are already a programmer - you probably don't know how to use. The **emacs** and **vim** text editors are extremely powerful and extensible, but they can be difficult to learn. Furthermore, there's no reason to learn them just for **SuperCollider**, because the component for **GEdit** is more than sufficient.

To install the minimum recommended installation for **SuperCollider**:

1. In a terminal, run `su -c 'yum install supercollider supercollider-gegit'`
2. Review the proposed installation carefully. The list may be quite long, and require a large download.

11.2. Using GEdit to Write and Run SuperCollider Programs

The *supercollider-gegit* package installs an extension for **GEdit** which allows editing and running **SuperCollider** code from within **GEdit**. There are also **SuperCollider** extensions for the **emacs** and **vim** text editors. This tutorial uses the **GEdit** extension, because it is easier to learn how to use **GEdit** than **vim** or "emacs."

11.2.1. Enable and Configure SCed in GEdit

These steps should be followed the first time that you use **GEdit**'s **SuperCollider** extension.

1. Start **GEdit**
2. Open the Preferences window (from the menu, choose **Edit** → **Preferences**)
3. Choose the 'Plugins' tab, and scroll down to *Sced*, then make sure that it is selected.

4. Click on the **Configure Plugin** button, then select a runtime folder where the **SuperCollider** server will store any synth sent to it during program execution. The safest place for this folder could be a sub-folder of the location where you will store your **SuperCollider** code.

11.2.2. Enable SuperCollider Mode and Start a Server

These steps should be followed every time you open **GEdit**, and wish to use the **SuperCollider** extension.

1. Choose **Tools** → **SuperCollider Mode**
2. A **SuperCollider** menu should appear, and a window at the bottom which says, "**SuperCollider** output".
3. If you cannot see the window at the bottom, then select **View** → **Bottom Pane** from the menu, so that it shows up. It is sometimes important to see the information that **SuperCollider** provides in this window.
4. After enabling **SuperCollider** mode, the window should display a series of notices. Near the end should be something like this:

```
RESULT = 0
Welcome to SuperCollider, for help type ctrl-c ctrl-h (Emacs) or :Schelp (vim) or ctrl-U
(sced/gegit)
```

If this window gives a non-zero value for "RESULT," then an error has probably occurred, and you should scroll up to see what it is, and try to fix it. If you receive the following warning: "The GUI scheme 'swing' is not installed" then you will not be able to run any **SuperCollider** programs that use a GUI (graphical user interface). The GUI components are not used anywhere in this Guide, and they are highly optional.

5. You will probably also want to start a server at this point, so from the menu select '**SuperCollider** > Start Server'.
6. After the server starts, you should see messages from "JackDriver". If a JACK server is not already started, then **SuperCollider** will start one automatically.
7. If the **SuperCollider** server started successfully, you should see a message similar to this:

```
SuperCollider 3 server ready..
JackDriver: max output latency 46.4 ms
notification is on
```

11.2.3. Executing Code in GEdit

You can execute code directly from **GEdit**, without having to use **sc1ang** from the command-line.

1. Ensure that **SuperCollider** mode is enabled, and that a server has been started.
2. Select the code that you wish to execute. A single line of code may be executed simply by placing the text-input cursor on that line.
3. Press 'Ctrl+E' on the keyboard, or from the menu select '**SuperCollider** > Evaluate'

4. To stop all sound on the server, press 'Esc' on the keyboard, or from the menu select '**SuperCollider** > Stop Sound'
5. If the server successfully executes the code, then it will output something to the "**SuperCollider** output" pane. The output will be different, depending on what **SuperCollider** asked the server to do, but will usually either look like this:

```
Synth("temp_0": 1000)
```

or this:

```
RESULT = 0
```

11.2.4. Other Tips for Using **GEdit** with **SuperCollider**

- If you close **GEdit** while the **SuperCollider** server is running, then **GEdit** will automatically shut down the server.
- If JACK is started by **SuperCollider**, then it will automatically terminate when the **SuperCollider** server terminates.
- **SuperCollider** will automatically attempt to connect its outputs to the system's outputs. If your audio output doesn't work, then you should use **QjackCtl** to verify that it is correctly connected.
- Other functions available in the **SuperCollider** menu include:
 - Find Help (Opens the **SuperCollider** help file for currently-selected object).
 - Find Definition (Opens the **SuperCollider** source file for the currently-selected object).
 - Restart Interpreter (Restarts the **SuperCollider** interpreter; also closes running servers, but does not restart them).
 - Clear output (Clears all output from the "**SuperCollider** output" pane).

11.3. Basic Programming in **SuperCollider**

As with any programming language, you will start learning **SuperCollider** with the basic commands, that are of little use by themselves. However, since the language is so flexible, even the most basic commands can be combined in ways that create highly complex behaviours. The example program, "Method One," was written with the goal of illustrating how a single sound-generating object can be used to create an entire composition. This tutorial does not begin with audio-generating code, which helps to emphasize that **SuperCollider** is primarily a programming language.

This portion of the Guide is designed as a "reference textbook," which you can use both to learn the **SuperCollider** language in the first place, and to remind yourself about the language's features afterwards.

The section is most effective when read in small portions.

11.3.1. First Steps

11.3.1.1. The Different Parts of **SuperCollider**

As you discovered when installing **SuperCollider**, there are actually many different components involved with **SuperCollider**. Here is a list of some of them, with brief descriptions of their purpose:

- Programming language: this is an abstract set of rules and guidelines that allow you to write down instructions for producing sounds.
- Interpreter: this is what is run in **GEdit**; it transforms the programming language instructions written by you into useful instructions for the server; also called the "client."
- Server: this is what synthesizes the sound, according to instructions sent to it by the interpreter.
- Library: these contain commands and the instructions to be executed when you call the commands; the interpreter looks up commands in the library when you call them.

This modular design allows for several advanced capabilities and features. Any particular element could theoretically be replaced without affecting other elements, as long as the methods of communication remain the same. As long as the programming language is the same, portions of the library can be modified, removed, or added at will; this happens often, and Planet CCRMA at Home provides a collection of library extensions. One of the most exciting capabilities is the ability to run the interpreter and server on different physical computers. The networking component is built into these components - they always communicate by UDP or TCP, even when run on the same computer! Although this ability is not used in this Guide, it is not difficult.

The most important thing to remember is that the **SuperCollider** interpreter is what deals with the programs you write. The **SuperCollider** server is controlled by the interpreter, but is an independent program. For simple things, like the Hello World Programs below, the server is not even used - after all, there is no audio for it to synthesize.

11.3.1.2. "Hello, World!"

The first program that one traditionally makes when learning a new programming language is called "The Hello World Program." This is a simple and trivial application that simply prints out the phrase, **Hello, World!** (or a variation of it). It might seem useless at first, but the ability to provide feedback to an application's user is very important, and this is essentially what the Hello World Program does.

Here is the program in **SuperCollider**:

```
"Hello, World!".postln;
```

Here is an extension to that program:

```
"Hello, World!".postln;  
"Hello, SC!".postln;
```

As with all examples in this section, you should paste these programs into **GEdit**, and execute them with **SuperCollider**. Look at the output produced by the programs, but don't worry about it for now.

These programs are very small, but it highlights some key concepts of the **SuperCollider** language, described below.

11.3.1.3. Return Values

Every **SuperCollider** program must provide the interpreter with a value (some information) when it has carried out all of its instructions. This value is called a "return value," because it is the value given by a program when it "returns" control to the interpreter. In a **SuperCollider** program, it is the last value stated in a program that automatically becomes the return value - no special command is required. When program execution ends, and control is returned to the **SuperCollider** interpreter, the interpreter outputs the return value in the "SuperCollider output" pane.

In the single-line Hello World Program above, the program produces the following output: **Hello, World! Hello, World!** The program appears to have been executed twice, but that is not the case. The first **Hello, World!** is printed by the program. The second **Hello, World!** appears because **"Hello, World!".postln** is the last (in this case, the only) value of the program. It is "returned" by the program, and the interpreter prints it.

In the two-line Hello World Program above, the program produces the following output: **Hello, World! Hello, SC! Hello, SC!** This makes it more clear that the program is not being executed twice, and that it is the last value of a program that is returned to the interpreter.

Try executing the following single-line programs. Look at the output produced by each, and determine whether it is printed by the program itself, the interpreter, or both.

- **"Hello, World!".postln;**
- **"Hello, World!";**
- **5.postln;**
- **5;**

Can you modify the two-line Hello World Program so that each line is printed only once?

In reality, every "function" must return a value. Functions are described in [Section 11.3.2.3, "Functions"](#), but the difference is not yet important.

11.3.1.4. Statements

A "statement" is a single instruction, which always ends with a semicolon. Exactly what constitutes a statement will become clear as you gain experience, and you will eventually automatically remember the semicolon.

In the Hello World Programs above, all of the statements contain the single instruction to post a line to the output screen. What happens when you remove the first semicolon, which marks the end of the first statement? The **SuperCollider** interpreter produces an unhelpful error message, and tells you that an error occurred *after* the forgotten semicolon. This is why it is important to always remember statement-concluding semicolons.

11.3.1.5. Data Types: Numbers and Strings

In many programming languages, it is the programmer's responsibility to determine the type of data that is being used, and how it should be stored. The **SuperCollider** interpreter takes advantage of the power of modern computers, and deals with this on our behalf. This greatly simplifies basic tasks, because there are only two kinds of data to worry about, and they make perfect sense:

- **Numbers:** These are numbers, written simply as numbers. Anything that can be done with real-world numbers can also be done with **SuperCollider**'s numbers. They can be as large or small, positive or negative as you want. They can have any number of digits on either side of the decimal point.
- **Strings:** These are a string of characters, written between two double-quote characters like "this." The double-quote characters are required so that **SuperCollider** knows where to begin and end the string of characters. A string of character can contain as many characters as you like, including one character and no characters. If you want to include a double-quote character in a string, you should put a backslash before it. The following is interpreted by **SuperCollider** as a string with only a double-quote character: **"\""**

Here are some examples of numbers and strings:

- **5**

- 18920982341
- 0.000000000000001
- "characters"
- "@"
- ""
- "6"

Is the last example a number or a string? You and I recognize that it is a number inside a string, but **SuperCollider** treats it as a string. You can do string things with it, but you cannot do number things with it. You cannot add "6" to something, for example.

Try executing the following single-line programs. Think about why the **SuperCollider** interpreter produces the output that it does.

- 6 + 3;
- "6" + 3;
- "six" + 3;

11.3.1.6. Consecutive Execution

Complex **SuperCollider** programs contain many parts, which all do different things. Sometimes, executing all of these together doesn't make sense, and it can be difficult to know which portions of the program are supposed to be executed when. To help with this, the interpreter allows you to mark portions of your program between (and) so that you will know to execute them together.

Here is an example:

```
(  
  "Hello, Fred!".postln;  
  "Hello, Wilma!".postln;  
)  
(  
  "Goodbye, Fred!".postln;  
  "Goodbye, Wilma!".postln;  
)
```

It doesn't make sense to say "hello" and "goodbye" at the same time, so separating these sections with parentheses will serve as a reminder. In case we try to execute all of the code at once, the **SuperCollider** interpreter will give us an error.

11.3.2. Variables and Functions

The concepts in this section are related to the mathematical terms with the same names. This is a modern-day result of the first uses of computers and programming languages: the calculation of complex mathematical problems.

11.3.2.1. Variables

A variable is a symbol that can be assigned an arbitrary value. A "symbol" is a series of alphabetic and numeric characters, separated by whitespace (a space, a line-break, or the end of the file). When a

variable is "assigned" a value, the variable name (the symbol) is understood to be a substitute for the assigned value.

Consider a traffic light, which has three possible symbols: green, yellow, and red. When you are driving, and you encounter a traffic light, you might see that its red symbol is activated (the red light is illuminated). What you see is a red light, but you understand that it means you should stop your car. Red lights in general do not make you stop - it is specifically red traffic lights, because we know that it is a symbol meaning to stop.

SuperCollider's variables work in the same way: you tell the interpreter that you want to use a symbol, like **cheese**. Then you assign **cheese** a value, like **5**. After that point, whenever you use **cheese**, the interpreter will automatically know that what you really mean is **5**.

Run the following two programs. They should result in the same output.

```
(
  5 + 5;
)
(
  var x;
  x = 5;
  x + x;
)
```

In the first example, the program calculates the value of **5 + 5**, which is **10**, and returns that to the interpreter, which prints it out. In the second example, the program tells the interpreter that it wants to use a variable called **x** then it assigns **cheese** the value **5**. Finally, the program calculates **cheese + cheese**, which it understands as meaning **5 + 5**, and returns **10** to the interpreter, which prints it out.

This trivial use of a variable does nothing but complicate the process of adding 5 to itself. Soon you will see that variables can greatly simplify your programs.

11.3.2.2. Using Variables

There are three words that describe the key stages of using a variable: declaration, initialization, and assignment.

A variable must be declared before use, so that the interpreter knows that you want to use that symbol as a variable. All variables must be declared before any statement that does not declare a variable; in other words, you should declare your variables before doing anything else. Variable names are declared like this:

```
var variableName;
```

Variables can also be declared in lists, like this:

```
var variableName, variableOtherName;
```

Variables can be assigned a value at any time after they have been declared. Any single object can be assigned to a variable. If a variable is already assigned a value, any subsequent assignment will erase the previous assignment; the previously-assigned value will not be retrievable.

The first assignment to a variable is said to "initialize" the variable. Initialization is a special kind of assignment, because a variable cannot be used before it is initialized. If a program attempts to use an un-initialized variable, the **SuperCollider** interpreter will cause an error. For this reason, you should always initialize a variable when you declare it. There is a special way to do this:

```
var variableName = nil;
```

Since you can't always assign a useful value, you can pick an arbitrary one. Assigning "nil" is common practice, because it means "nothing," but without actually being nothing (this avoids *some* errors). Assigning zero is another possibility; it is standard practice in many programming languages, and will avoid most errors, even if the variable is eventually supposed to hold another kind of object. Initialization and declaration of multiple variables can also be done as a list:

```
var variableName = 0, variableOtherName = 0;
```

Single-letter variable names have a special purpose in **SuperCollider**. They are already declared, so you don't have to declare them. They are also already initialized to "nil", so you don't have to do that either. These variable names are intended to be used as a quick fix, while you're experimenting with how to make a program work. You should not use them in good-quality programs.

The single-letter variable "s" is automatically assigned to the server on the computer running the interpreter. You should avoid re-assigning that variable.

Variable names must always begin with a lower-case letter.

Use variables to write programs that do the following tasks:

1. Perform arithmetic with an uninitialized variable. An error should appear when the program is executed.
2. Calculate the value of y , if all other values are known, for the quadratic equation: $y = a * x * x + b * x + c$
3. Re-write the Hello World Program so that it will say "Hello" to a name stored in a variable. Remember that you can use the interpreter to automatically output the last line of a function.

11.3.2.3. Functions

A Function is a statement, or a series of statements, that we want to use many times. When a Function is assigned to a variable, you can execute the Function as many times as you wish. Any statements that happen between braces { like this; } are treated as a Function. Functions are executed by passing them the "value" message, as in the following example.

Here is a Function that is not assigned to a variable, and is executed once.

```
{ "Hello, World!".postln; }.value;
```

Notice that there are two semicolons: one after the statement within the Function, and one after the "value" message that tells the Function to execute.

Here is a Function with identical function, assigned to a variable, and executed twice.

```
var myFunction = { "Hello, World!".postln; }; // note two semicolons
myFunction.value;
myFunction.value;
```

11.3.2.4. Function Arguments

The most useful aspect of Functions is that they can produce varying results, depending on their input. For whatever reason, the input accepted by a Function is called an "argument." **SuperCollider's**

Functions can accept any number of arguments - zero, one, or many. Argument values (called "parameters") are provided to a Function by adding them in parentheses after the name of the Function, separated with commas, like this: **exampleFunction(5, 7, 9);** Argument variables are declared as the first statement in a Function, like this: **arg oneNumber, twoNumber;**

This program is significantly more complicated than previous examples, but it shows how useful Functions can be. Notice how the braces in that example are on different lines than the rest of the Function, which gives us more space within the Function to complete some useful work.

```
(
  var greeter =
  {
    arg name;
    ( "Hello" + name ).postln;
  };

  greeter.value( "Samantha" );
  greeter.value( "Jermain" );
  nil;
)
```

Here is how the program works:

1. A variable named **greeter** is declared, and assigned a Function.
2. The Function contains an argument called **name**, and outputs "Hello" plus the name given to it.
3. The parentheses here (**"Hello" + name**) ensure that the two strings are added together *before* the `postln` message prints them out.
4. The **greeter** variable is used to call the Function with two different names.
5. The **nil**; statement is optional, and does not affect the operation of the program. What it does is return a "nothing" value to the interpreter after program execution completes, so that the last message is not repeated.

Since every argument has a name, **SuperCollider** allows you to use that name when executing the function. This example executes the `greeter` function from the last example:

```
greeter.value( name:"Myung-Whun" );
```

This is more useful if there are many arguments, and you do not remember the order that they appear in the Function's definition.

SuperCollider also allows you to specify default values for arguments, so that they do not need to be specified. This allows optional customization of a Function's behaviour, and is therefore very powerful.

This example modifies the one above by adding default-value arguments, and by calling arguments with their name. As you can see, I've been tricking you a bit: `postln` is actually a Function, but a special kind, explained later.

```
(
  var greeter =
  {
    arg name, greeting = "Hello";
    postln( greeting + name );
  };
)
```

```
greeter.value( "Samantha" );
greeter.value( "Jermain", "Goodbye" );
greeter.value( name:"Myung-Whun" );
greeter.value( greeting:"Bienvenue", name:"Marcel" );
nil;
)
```

Any value can be used as a parameter, as long as the Function expects it. In fact, even Functions can be used as parameters for Functions!

11.3.2.5. Function Return Values

All **SuperCollider** Functions return a value to the interpreter when they have finished execution. As with programs, the value returned is the value of the last statement in the Function. The return value of a Function can be captured, assigned to a variable, and used again later.

This example assigns the result of a Function to a variable.

```
(
  var mysticalMath =
  {
    arg input = 0;
    input * 23;
  };
  var someNumber = 9;

  someNumber = mysticalMath.value( someNumber );
  someNumber.postln;
  nil;
)
```

Here is how the program works:

1. A Function and variable are created, and assigned values.
2. This line **someNumber = mysticalMath.value(someNumber);** executes the **mysticalMath** function, which multiplies its argument by **23** and returns the value. Then, it assigns the return value of the Function to **someNumber**. In any statement that contains an assignment, the assignment is always done last. In other words, the Function in this example will *always* be given an argument of **9**, and only *after* the Function completes execution and returns a value will that value be assigned to **someNumber**.
3. The new value of **someNumber** is displayed.

The program could have been shortened like this:

```
(
  var mysticalMath =
  {
    arg input = 0;
    input * 23;
  };

  var someNumber = mysticalMath.value( 9 );
  someNumber.postln;
  nil;
)
```

It could have been shortened even more like this:

```
(
  var mysticalMath =
  {
    arg input = 0;
    input * 23;
  };

  mysticalMath.value( 9 ).println;
  nil;
)
```

Experiment with the shortened versions of the program, ensuring that you know why they work.

11.3.2.6. Variable Scope

A variable is only valid within its "scope." A variable's scope is determined by where it is declared. It will always last between either (and) or { and }, and applies to all statements within that block of code. Variable names can be re-declared in some contexts, which can be confusing.

Consider the scope of the variables in this example:

```
(
  var zero = 0;
  var function =
  {
    var zero = 8;
    var sixteen = 16;
    zero.println; // always prints 8
  };

  function.value;
  zero.println; // always prints 0
  sixteen.println; // always causes an error
)
```

Because **function** declares its own copy of **zero**, it is modified independently of the variable **zero** declared before the Function. Every time **function** is executed, it re-declares its own **zero**, and the interpreter keeps it separate from any other variables with the same name. When **function** has finished executing, the interpreter destroys its variables. Variables declared inside any Function are only ever accessible from within that Function. This is why, when we try to execute **sixteen.println**;, the interpreter encounters an error: **sixteen** exists only within **function**, and is not accessible outside the function. By the way, in order to excute this example, you will need to remove the error-causing reference to **sixteen**.

Now consider the scope of the variables in this example:

```
(
  var zero = 0;
  var function =
  {
    var sixteen = 16;
    zero = 8;
    zero.println; // always prints 8
    sixteen.println;
  };

  function.value;
```

```
    zero.postln; // always prints 8
  )
```

Why does the last line always print **8**? It's because **zero** was set to **8** within **function**. More importantly, **function** did not declare its own copy of **zero**, so it simply accesses the one declared in the next "highest" block of code, which exists between **(** and **)** in this example.

This is why it is important to pay attention to a variable's scope, and to make sure that you declare your variables in the right place. Unexpected and difficult-to-find programming mistakes can occur when you forget to declare a variable, but it is declared elsewhere in your program: you will be allowed to use the variable, but it will be modified unexpectedly. On the other hand, it can be greatly advantageous to be able to access variables declared "outside the local scope" (meaning variables that are not declared in the same code block in which they are used), but careful thought and planning is required.

Astute readers will notice that it is possible to re-declare the single-letter variable names, allowing you to control their scope. Consider the following program:

```
(
  var a = 0;
  b =
  {
    var c = 16;
    a = 8;
    a.postln;
    c.postln;
  };

  b.value;
  a.postln;
)
```

This example requires careful examination. What is the scope of **a**, **b**, and **c**? The answers may be surprising.

- **a** is declared just after the **(** character, so the interpreter destroys it upon reaching the **)** character.
- **c** is declared just after the **{** character, so the interpreter destroys it upon reaching the **}** character.
- **b** is *not* declared in this program, so it refers to the automatically-declared variable with that name. The interpreter does not destroy it until it is restarted or stopped. This means that the Function assigned to **b** is still available *after* the program finishes execution. Try it! Execute the program above, and then execute this single-line program alone: **b.value**;

11.3.3. Object-Oriented SuperCollider

SuperCollider is difficult to describe precisely, because its syntax allows great flexibility. There are many different ways to accomplish the same task. Each one is subtly different, and gives you a different set of possibilities, but there is often no "best solution." One of the advantages to this is that it easily allows three "programming paradigms," although one is used much more often than the others.

11.3.3.1. Imperative Programming

Imperative programming is easy to understand: it is simply a list of commands, like this:

```
(
  var a, b, c;
```

```
a = 12;  
b = 25;  
c = a + b;  
a.postln;  
)
```

Declare the variables, set the variables, do a calculation, and print the result of the calculation. This is a simple example, and a simple model, but it is very difficult to escape completely. After all, humans think of large problems in terms of algorithms (the instructions needed to do something). Computers solve large problems, so being able to program them with a series of instructions makes sense.

11.3.3.2. Functional Programming

Functional programming is also easy to understand, but it can be a little bit more difficult to think about complex tasks. Functional programs use Functions to complete all of their work. This is not strictly possible in **SuperCollider**: it is more imperative than functional, but the creative use of Functions can easily solve some problems that are difficult to write with an imperative approach.

The following example is an extension of the "Imperative" example. Pretend that the following Functions exist, and do the following tasks:

- `getinput` : allows the user to enter a number, and returns that number
- `add` : adds together the numbers given as arguments, returning the sum

```
(  
  postln( add( getinput, getinput ) );  
)
```

SuperCollider will always execute the inner-most Functions first. This is how the interpreter executes the single-line program above:

1. Execute the left call of `getinput`
2. Execute the right call of `getinput`
3. Execute `add` with the two numbers returned by `getinput`
4. Execute `postln` with the number returned by `add`

Both imperative and functional programming have advantages and disadvantages. **SuperCollider** will allow you to use either approach, or a mix of both, when solving problems.

11.3.3.3. Object-Oriented Programming

Object-oriented programming is more difficult to think about than imperative or functional. When using this paradigm (mode of thought), almost everything in **SuperCollider** is thought of as an abstract Object. In this way, it allows programmers to make compelling comparisons to the real world, where all tangible things are objects, and where it is not hard to conceive of most intangible things as objects, too. With object-oriented programming, computer science takes a break from mathematics, and is influenced by philosophy.

Class names always begin with an uppercase letter.

Anything can be represented as an Object - like a bicycle, for instance. Let's pretend that we have an Object called a Bicycle. We don't yet have a particular bicycle - just the abstract *class* containing everything that is true about all bicycles. If a Bicycle class exists in **SuperCollider**, you generate

a specific instance like this: `var bike = Bicycle.new;` All **SuperCollider** Objects can be *instantiated* in this way: you get a specific Bicycle from the generic class, and you can then modify and work with your own Object as you choose. The specific properties associated with a particular instance of a class are called *instance variables*.

There are certain things that Bicycles are designed to do: turn the wheels, turn the handlebar, raise and lower the seat, and so on. You can cause these things to happen by providing a certain input to a real-world Bicycle: if you want to turn the wheels, you might push the pedals. In **SuperCollider**, you cause things to happen by *sending a message* to the Object that tells it what you want: if you have a Bicycle, you might turn the wheels like this: `bike.turnTheWheels;` When you do this, you actually execute the `turnTheWheels` function, which is defined by the abstract Bicycle class. Because it doesn't make sense to turn the wheels of all bicycles in existence, you don't call the *method* (a synonym for "Function") from the Bicycle class itself, but from the particular instance whose wheels you want to turn. The proper way to access instance variables is by using instance methods.

If this kind of programming is new to you, it might seem extremely difficult. It can be intimidating at first, but it is actually not too difficult to understand once you start to use it. In fact, you have already been using it! Remember the `postln` command that was described earlier as a special kind of Function? It's actually a Function defined by **SuperCollider's** abstract class **Object**, which defines a set of messages that can be passed to *any* **SuperCollider** object. Because most things in **SuperCollider** are objects, we can send them the `postln` message, and they will understand that it means to print themselves in the "**SuperCollider** output" pane.

Why is it that all Objects respond to the `postln` message? **SuperCollider** classes are allowed to belong to other **SuperCollider** classes, of which they are a part. Consider the Bicycle class again. It is a kind of vehicle, and philosophers might say that "things that are members of the bicycle class are also members of the vehicle class." That is, real-world bicycles share certain characteristics with other real-world objects that are classified as "vehicles." The bicycle class is a "sub-class" of the vehicle class, and it *inherits* certain properties from the vehicles class. **SuperCollider** allows this behaviour too, and calls it *inheritance*. In **SuperCollider**, since all classes define Objects, they are all automatically considered to be a sub-class of the class called **Object**. All classes therefore inherit certain characteristics from the **Object** class, like knowing how to respond to the `postln` message.

equivalent notation: `5.postln` versus `postln(5)`

You still don't know how to write new Classes and Objects in **SuperCollider**, but knowing how to use them is more than enough for now. By the time you need to write your own Classes, you will probably prefer to use the official **SuperCollider** help files, anyway.

11.3.3.4. Choosing a Paradigm

At this point you may begin worrying about which programming paradigm you should choose, and when. The answer is unhelpful: "Whichever seems best for the task."

Let's expand on this. If you are primarily a programmer, then you probably already know how to choose the best paradigm and algorithm for the task. If you are a musician, then you probably just want your program to produce the output that you want (in this case, a particular set of sounds). Part of the beauty of **SuperCollider's** flexibility is that it allows you to produce the same output in different ways. As a musician this means that, as long as your program works as you want it to work, it doesn't matter how you write it. Experience will teach you more and less effective ways of doing things, but there is no need for rules.

Even so, here are some guidelines that will help you to start thinking about music programs:

- Programs of all sorts often follow a simple, four-step flow. Not all parts are always present.
 1. Declare variables.

2. Get input from the user.
 3. Calculate something with the input.
 4. Provide output to the user (i.e. "make noise").
- Repetition is the enemy of correctness, so if you're going to execute some code more than once, try writing a Function. If it's slightly different every time, try using arguments. Arguments with default values are a great way to expand a Function's usefulness.
 - If it looks too complicated, then it probably is. The more difficult it is to understand something, the greater the chance of making a mistake.
 - Don't forget the semicolons at the end of every statement.

11.3.4. Sound-Making Functions

It's finally time to start thinking about Functions that produce sound!

This example is discussed below in the following sections. Remember, when running **SuperCollider** code in **GEdit**, you can stop the sound by pressing **Esc**

```
{ SinOsc.ar( 440, 0, 0.2 ); }.play;
```

11.3.4.1. UGens

"UGen" stands for "unit generator." UGens are special Objects that generate either an audio or a control signal.

The UGen that will be used for most of the experimentation in this Guide, and which was primarily used to generate the "Method One" program that goes with this Guide, is called **SinOsc**, which generates a sine wave. The class' name, **SinOsc** means "sine oscillator."

The example at the beginning of this chapter, **SinOsc.ar(440, 0, 0.2);** produces an "instance" of the **SinOsc** class, which continuously outputs a signal, based on the parameters given in parentheses. This instance produces an "audio rate" signal, which means that it is of sufficient quality to eventually become sound.

A slightly modified version of that code will give us a "control rate" signal: **SinOsc.kr(440, 0, 0.2);** There is only one small difference between the two examples - for us - but for **SuperCollider**, the difference is huge. A control rate signal will not be of sufficient quality to become sound; it is used to control other UGens that do become sound.

Unlike other Classes, UGen Classes should not be instantiated with the new message. They should always be instantiated as either audio-rate (by passing the **ar** message), or control-rate (by passing the **kr** message). Control-rate signals are calculated much less often than audio-rate signals, which allows the **SuperCollider** interpreter and server to save processing power where it wouldn't be noticed.

11.3.4.2. The "play" Function

The **play** function does exactly what it says: it plays its input. The input must be a function with an audio-rate signal generator as the return value.

The following two examples produce the same output:

```
{ SinOsc.ar( 440, 0, 0.2 ); }.play;
```

```
play( { SinOsc.ar( 440, 0, 0.2 ); } );
```

The first example is written from an object-oriented perspective. Functions know how to play their return value, when passed the `play` message. This is true of all Functions whose return value is an audio-rate UGen. The second example is written from a functional perspective. The Function called `play` will play its input, which must be a Function whose return value is an audio-rate UGen. Whether you should write `play` in the functional or object-oriented way depends on which makes more sense to you.

Try to re-write the above example so that the `play` function operates on a variable-defined Function.

The arguments to **SinOsc**, whether the audio- or control-rate generator, are these:

- The first is called **freq**; it sets the frequency.
- The second is called **add**; it is added to all values produced by the UGen.
- The third is called **mul**; all values produced by the UGen are multiplied by this.

You now know enough to spend hours with the sine oscillator UGen. Try combining audio- and control-rate UGens, and try to figure out what happens when each of the arguments is adjusted. Be careful that your audio interface's volume isn't set too high! Experiment with this one:

```
(
  var myFrequency = SinOsc.kr( freq:1, mul:200, add:400 );
  var sound = { SinOsc.ar( myFrequency, 0, 0.2 ); };

  play( sound );
)
```

11.3.5. Multichannel Audio

By now you must be growing tired of the left-side-only sounds being produced by the examples.

11.3.5.1. Stereo Array

The easiest way to output multichannel audio in **SuperCollider** is to use a kind of "Collection" (defined later) called an "Array." **SuperCollider** will theoretically handle any number of audio output channels, but by default is usually only configured for two-channel stereo audio. Since humans have only two ears, this is sufficient for most tasks! A multichannel array is notated like this:

[LeftChannel.ar(x), RightChannel.ar(y)]

Here is our simple sine oscillator expanded to produce stereo audio:

```
{ [ SinOsc.ar( 440, 0, 0.2 ), SinOsc.ar( 440, 0, 0.2 ) ]; }.play;
```

Not much has changed, except that the audio we hear is now being emitted from both the left and right channels. Change the frequency of one of the sine oscillators to **450** and the difference will become much more apparent.

Multichannel arrays can also be combined with each other, like this:

```
{
  var one = [ x, y, z ];
  var two = [ a, b, c ];
  [ one, two ];
}
```

If **a**, **b**, **c**, **x**, **y**, and **z** were all audio-rate UGens, this function could be play'ed. It would produce stereo audio, and each channel would have three independent UGens.

11.3.5.2. Multichannel Expansion

You can automatically create multiple UGens by providing an Array as one of the parameters. The **SuperCollider** interpreter will automatically create multichannel UGens as a result.

The following two examples produce equivalent output:

```
{ [ SinOsc.ar( 440, 0, 0.2 ), SinOsc.ar( 440, 0, 0.2 ) ]; }.play;
```

```
{ SinOsc.ar( [440, 440], 0, 0.2 ); }.play;
```

The second example can be easier to read, because it is obvious that only the frequency is changing - or in this case, that nothing is changing. This technique is more useful in a situation like the following:

```
{ SinOsc.ar( [[440, 445, 450, 455, 460, 465],
              [440, 445, 450, 455, 460, 465]],
            0,
            0.2 ); }.play;
```

That's not exactly easy to read, but it's easier to figure out than the most obvious alternative:

```
{
  [[ SinOsc.ar( 440, 0, 0.2 ), SinOsc.ar( 445, 0, 0.2 ), SinOsc.ar( 450, 0, 0.2 ),
    SinOsc.ar( 455, 0, 0.2 ), SinOsc.ar( 460, 0, 0.2 ), SinOsc.ar( 465, 0, 0.2 ) ],
    [ SinOsc.ar( 440, 0, 0.2 ), SinOsc.ar( 445, 0, 0.2 ), SinOsc.ar( 450, 0, 0.2 ),
    SinOsc.ar( 455, 0, 0.2 ), SinOsc.ar( 460, 0, 0.2 ), SinOsc.ar( 465, 0, 0.2 ) ]];
}.play;
```

More importantly, multichannel expansion gives us another tool to avoid repetition. Repetition is the enemy of correctness - it's so much more difficult to find a mistake in the second example than in the first!

11.3.5.3. Method One

Believe it or not, you now know enough to understand a slightly-modified version of the first part of "Method One," a **SuperCollider** program written and heavily commented specifically for use with this guide. You should play this example, and experiment with changing the frequencies, volumes, and so on. The fully-commented version provides a full explanation of how the function works.

```
{
  // sets up the frequencies of both channels
  var frequencyL = SinOsc.kr( freq:10, mul:200, add:400 ); // oscillating
  var frequencyR = SinOsc.kr( freq:1, mul:50, add:150 ); // oscillating
  var frequencyL_drone = SinOsc.kr( freq:0.03, mul:20, add:100 ); // drone
  var frequencyR_drone = SinOsc.kr( freq:0.01, mul:20, add:210 ); // drone
```

```

// changes the volume of the oscillating part in the left channel
var volumeL = SinOsc.kr( freq:0.5, mul:0.02, add:0.03 );

// left channel
var left = [ SinOsc.ar( freq:frequencyL, mul:volumeL ), // this is the oscillating part
            SinOsc.ar( freq:[frequencyL_drone,2*frequencyL_drone], mul:0.02 ), // the
rest make up the drone
            SinOsc.ar( freq:[5*frequencyL_drone,7*frequencyL_drone], mul:0.005 ),
            SinOsc.ar( freq:[13*frequencyL_drone,28*frequencyL_drone], mul:0.001 ) ];

// right channel
var right = [ SinOsc.ar( freq:frequencyR, mul:0.1 ), // this is the oscillating part
            SinOsc.ar( freq:[frequencyR_drone,2*frequencyR_drone], mul:0.02 ), // the
rest make up the drone
            SinOsc.ar( freq:4*frequencyR_drone, mul:0.005 ),
            SinOsc.ar( freq:[64*frequencyR_drone,128*frequencyR_drone], mul:0.01 ) ]; //
high frequencies!

[ left, right ];
}

```

11.3.6. Collections

A "collection" is just that - a collection of Objects. Collections are simply a means of organizing a large amount of data, without having to assign a variable name for each portion of data. Compared to other programming languages, **SuperCollider** provides a relatively large number of Collections in the standard library.

We have already seen an example of a Collection as multichannel audio arrays. An Array is a kind of Collection - in object-oriented terminology, the Array Class is a *sub-class* of the Collection Class, and inherits its behaviours. Conversely, the Collection Class is the *super-class* of the Array Class. The Collection Class itself is not to be used; it is designed to provide common features so that it is easier to write Classes for collections.

As with all the chapters from this point on, it is not necessary to read this in sequence. If you prefer, you can skip it and return later when you need to manage a large set of data.

11.3.6.1. Array

Arrays have been traditionally been very popular with programmers. In **SuperCollider**, they are capable of storing a large number of Objects, and they provide advanced behaviours that are normally not associated with Arrays. They are not as indispensable as they used to be. Most programming languages now provide (or can easily be extended to add) Lists, Trees, and other kinds of data storage structures, which offer more capabilities, and are easier to use and to think about. Users new to programming might find the various kinds of Lists to be more helpful.

11.3.6.1.1. Building an Array

An Array is a Collection with a finite maximum size, determined at declaration time. It is the programmer's responsibility to maintain a meaningful order, and to remember the meaning of the data. Data in an Array is called "elements," each of which is assigned a specific "index number." Index numbers begin at 0. Any mix of Objects can be stored in an Array, including an Array.

This example declares an Array, adds some elements, then prints them out.

```
(
```

```

var tA = Array.new( 2 ); // "tA" stands for "testArray"

tA = tA.add( 5 );
tA = tA.add( 3 );
tA = tA.add( 17 );

tA.postln;
nil;
)

```

Notice that `Array` is a Class, and it must be instantiated before use. Here, the variable `tA` is assigned an `Array` with enough space for two objects. Notice that the elements are printed out in the order that you add them to the `Array`. They are not sorted or shuffled (unless you send a message like `scramble`). But why did I write `tA = tA.add(17);` instead of `tA.add(17);`? Shouldn't the second method be sufficient for adding an `Object` to an `Array`, thereby making the re-assignment unnecessary? It does, but let's see what happens when we take it away:

```

(
  var tA = Array.new( 2 ); // "tA" stands for "testArray"

  tA.add( 5 );
  tA.add( 3 );
  tA.add( 17 );

  tA.postln;
  nil;
)

```

The **17** is missing - it doesn't get added into the `Array`! This is because the `Array` was only declared with two slots, and you can't add three `Objects` into two slots. So why did this work the first time? **SuperCollider** was programmed to help us fit additional items into an `Array`. If an `Array` has reached its capacity, **SuperCollider** will automatically make a new, larger `Array` for us, and returns that from the `add` method. Therefore, any time you add an element to an `Array`, you should always re-assign the result, so that you don't have to worry about whether you exceeded the `Array`'s capacity.

11.3.6.1.2. Accessing an Array's Elements

There are two ways to access individual elements within an `Array`. One way is object-oriented, and one way is more traditional, inspired by programming languages such as the wildly popular "C" language. The object-oriented style uses the `at` and `put` methods. The traditional style uses square brackets with an index number.

The following examples produce equivalent output. The first uses the object-oriented style, and the second uses the traditional style.

```

(
  var tA = Array.new( 3 );

  tA = tA.add( 5 );
  tA = tA.add( 3 );
  tA = tA.add( 17 );

  tA.at( 0 ).postln; // outputs 5
  tA.at( 1 ).postln; // outputs 3
  tA.at( 2 ).postln; // outputs 17

  tA.put( 0, 24 ); // assigns 24 to element 0
)

```

```
tA.at( 0 ).postln; // outputs 24

nil;
)
```

```
(
  var tA = Array.new( 3 );

  tA = tA.add( 5 );
  tA = tA.add( 3 );
  tA = tA.add( 17 );

  tA[0].postln; // outputs 5
  tA[1].postln; // outputs 3
  tA[2].postln; // outputs 17

  tA[0] = 24 ; // assigns 24 to element 0

  tA[0].postln; // outputs 24

  nil;
)
```

Different people prefer different styles of accessing Arrays.

11.3.6.2. List

An List is a Collection with an infinite maximum size. It is the programmer's responsibility to maintain a meaningful order, and to remember the meaning of the data. Data in a List is called "elements," each of which is assigned a specific "index number." Index numbers begin at 0. Any mix of Objects can be stored in a List, including a List. Lists and Arrays are very similar, but **SuperCollider** manages some of the dirty work for you, when you use the List Class.

11.3.6.2.1. Building a List

There are four methods which instantiate a List. These are all "Class methods," meaning they do not operate on a specific List, but can be used to make any List.

- `List.new` creates a List. You can also specify the initial number of elements as an argument, if you choose.
- `List.newClear(x)` creates a List with **x** number of slots, filled with **nil**.
- `List.copyInstance(aList)` creates a List which is a copy of **aList**.
- `List.newUsing(anArray)` creates a List with the same elements as **anArray**.

11.3.6.2.2. Adding to an Existing List

These are "instance methods," meaning that they operate on a specific list.

- `put(index, item)` adds **item** into the List at index number **index**.
- `add(item)` adds **item** to the end of a List.
- `addFirst(item)` adds **item** to the beginning of a List.

11.3.6.2.3. Accessing a List

These are "instance methods," meaning that they operate on a specific list.

- `at(index)` returns the Object assigned to the **index** index number. If **index** is greater than the last element in the List, returns **nil**.
- `clipAt(index)` returns the Object assigned to the **index** index number. If **index** is greater than the last element in the List, returns the last element in the List.
- `wrapAt(index)` returns the Object assigned to the **index** index number. If **index** is greater than the last element in the List, returns an element based on a "wrap-around" index number. For a three-element List, **0** will return element **0**, **1** returns **1**, **2** returns **2**, **3** returns **0**, **4** returns **1**, **5** returns **2**, **6** returns **0**, and so on.
- `foldAt(index)` returns the Object assigned to the **index** index number. If **index** is greater than the last element in the List, returns an element based on a "fold-back" index number. Whereas `wrapAt()` always continues from the lowest to the highest index number, `foldAt()` changes every time: low to high, high to low, low to high, and so on.

11.3.6.2.4. Removing from a List

One way to remove an element from a List is to re-assign that element's index number the value **nil**. These two Functions also remove elements from a List. They are "instance methods," meaning that they operate on a specific list.

- `pop` returns the last element in a List, and removes it from the List.
- `removeAt(index)` removes the element assigned to **index** index number, removing it from the List and shrinking the List. `removeAt()` does not leave a **nil** element in the List.

11.3.6.2.5. Examples

The following examples show different ways to use List's.

```
(
  var tL = List.new;

  tL.add( 42 );
  tL.add( 820 );

  postln( tL.pop ); // outputs 820

  tL.add( 7 );
  tL.add( 19 );
  tL.add( 23 );

  postln( tL.pop ); // outputs 23
  postln( tL.pop ); // outputs 19
  postln( tL.pop ); // outputs 7
  postln( tL.pop ); // outputs 42

  postln( tL.pop ); // List is empty, so we get "nil"

  nil;
)
```

This code adds numbers to the end of a List, then removes them from the end of the List.

```
(
  var tL = List.new;
```

```

tL.addFirst( 42 );
tL.addFirst( 820 );

postln( tL.pop ); // outputs 42

tL.addFirst( 7 );
tL.addFirst ( 19 );
tL.addFirst ( 23 );

postln( tL.pop ); // outputs 820
postln( tL.pop ); // outputs 7
postln( tL.pop ); // outputs 19
postln( tL.pop ); // outputs 23

postln( tL.pop ); // list is empty, so we get "nil"

nil;
)

```

This modification of the first example adds numbers to the beginning of a List, then removes them from the end of the List. This is one way to ensure that the List elements are removed in the same order that they are added.

```

(
  var tL = List.new;

  tL.add( 42 );
  tL.add( 820 );

  postln( tL.removeAt( 0 ) ); // outputs 42

  tL.add( 7 );
  tL.add( 19 );
  tL.add( 23 );

  postln( tL.removeAt( 0 ) ); // outputs 820
  postln( tL.removeAt( 0 ) ); // outputs 7
  postln( tL.removeAt( 0 ) ); // outputs 19
  postln( tL.removeAt( 0 ) ); // outputs 23

  // postln( tL.removeAt( 0 ) ); // causes an error!

  nil;
)

```

This modification of the first example adds numbers to the end of a List, then removes from the beginning of the List. This is another way to ensure that the List elements are removed in the same order that they're added. Note that, when the List is empty, using the "removeAt()" Function causes an error, because you try to access a List index which doesn't exist.

```

(
  var tL = List.new;

  tL = [42,820,7,19,23];

  tL.at( 0 ).postln; // outputs 42
  tL.at( 1 ).postln; // outputs 820
  tL.at( 2 ).postln; // outputs 7
  tL.at( 3 ).postln; // outputs 19
  tL.at( 4 ).postln; // outputs 23
  tL.at( 5 ).postln; // outputs nil
)

```

```
    tL.at( 6 ).postln; // outputs nil  
    nil;  
  )
```

This example shows another way to add elements to an empty List, which also works for Arrays. Then it shows what happens when you try to access elements beyond the end of a List with the "at()" Function.

```
(  
  var tL = List.new;  
  
  tL = [42,820,7,19,23];  
  
  tL.clipAt( 0 ).postln; // outputs 42  
  tL.clipAt( 1 ).postln; // outputs 820  
  tL.clipAt( 2 ).postln; // outputs 7  
  tL.clipAt( 3 ).postln; // outputs 19  
  tL.clipAt( 4 ).postln; // outputs 23  
  tL.clipAt( 5 ).postln; // outputs 23  
  tL.clipAt( 6 ).postln; // outputs 23  
  
  nil;  
)
```

This example shows what happens when you try to access elements beyond the end of a List with the "clipAt()" Function. For index numbers beyond the end of the List, the interpreter will simply return the last element.

```
(  
  var tL = List.new;  
  
  tL = [42,820,7,19,23];  
  
  tL.foldAt( 0 ).postln; // outputs 42  
  tL.foldAt( 1 ).postln; // outputs 820  
  tL.foldAt( 2 ).postln; // outputs 7  
  tL.foldAt( 3 ).postln; // outputs 19  
  tL.foldAt( 4 ).postln; // outputs 23  
  tL.foldAt( 5 ).postln; // outputs 19  
  tL.foldAt( 6 ).postln; // outputs 7  
  
  nil;  
)
```

This example shows what happens when you try to access elements beyond the end of a List with the "foldAt()" Function. For index numbers beyond the end of the List, the interpreter will start moving back through the List, towards the first element, "folding" through the List.

```
(  
  var tL = List.new;  
  
  tL = [42,820,7,19,23];  
  
  tL.wrapAt( 0 ).postln; // outputs 42  
  tL.wrapAt( 1 ).postln; // outputs 820  
  tL.wrapAt( 2 ).postln; // outputs 7  
  tL.wrapAt( 3 ).postln; // outputs 19  
  tL.wrapAt( 4 ).postln; // outputs 23
```

```
tL.wrapAt( 5 ).postln; // outputs 42
tL.wrapAt( 6 ).postln; // outputs 820

nil;
)
```

This example shows what happens when you try to access elements beyond the end of a List with the "wrapAt()" Function. For index numbers beyond the end of the List, the interpreter will start again at the beginning of the List, "wrapping" around to the beginning.

11.3.6.3. LinkedList

Linked lists are very common structures for data management in computer science. They are more efficient than arrays for many tasks, particularly when it's impossible to know how many elements will be required in an array until the program is run. **SuperCollider's** List Class is implemented with arrays, and it offers nearly the same functionality as the LinkedList class.

A true linked list is accessed most efficiently from the start (called the "head" of the list) or the end (called the "tail"). Each element is linked to the one before it, the one after it, or both. **SuperCollider's** LinkedList Class has elements which are linked both to the preceding and following elements, so it is called a "doubly linked list."

Knowing when to use a LinkedList over a List is a question of efficiency, and for small collections of information, it isn't going to make a big difference - you might as well use a basic List. When you plan to store hundreds or thousands of elements, choosing the right Class becomes more important, and can save a lot of processor time. Here is how to know which Class you should use:

- If you're going to be adding elements to the start or end of the list, and accessing from the start or end of the list, the LinkedList Class will be more efficient.
- If you're going to be adding elements at arbitrary index numbers *inside* the list, and accessing elements at arbitrary index numbers inside the list, the List Class will be more efficient.
- If you're going to be adding elements to the start or end, but accessing specific indices, or adding elements at specific indices, but accessing from the start or end, then you get to choose where to save computation time. In one of these cases, it might not matter which one you choose.

11.3.6.3.1. Efficient Functions

These Functions make use of the LinkedList Class in an efficient way. They are efficient because they access only the first or last element in the LinkedList.

- `add(obj)` adds **obj** to a LinkedList as the last item.
- `addFirst(obj)` adds **obj** to a LinkedList as the first item.
- `pop` removes the last item in a LinkedList and returns it.
- `popFirst` removes the first item in a LinkedList and returns it.
- `first` returns a copy of the first item in a LinkedList.
- `last` returns a copy of the last item in a LinkedList.

11.3.6.3.2. Inefficient Functions

These Functions make use of the LinkedList Class in an inefficient way, but they can be useful. They are inefficient because they may potentially have to review all of the elements in a LinkedList before completing.

- `at(index)` pretends the `LinkedList` is an `Array`, and returns a copy of what would be the element at the given index number.
- `put(index, obj)` pretends the `LinkedList` is an `Array`, and changes the element at the given index number to be **obj**.
- `remove(obj)` searches through a `LinkedList` and removes the element judged to be equal to **obj**, regardless of its index number.
- `removeAt(index)` pretends the `LinkedList` is an `Array`, and removes the element located at the given index number.

11.3.6.3.3. Example

This example uses a `LinkedList` as a queue, adding numbers to the tail, and removing and printing from the head.

```
(  
  var tL = LinkedList.new;  
  
  tL.add( 42 );  
  tL.add( 89 );  
  
  tL.popFirst.postln; // prints 42  
  
  tL.add( 256 );  
  
  tL.popFirst.postln; // prints 89  
  
  tL.add( 900 );  
  
  tL.popFirst.postln; // prints 256  
  tL.popFirst.postln; // prints 900  
  
  nil;  
)
```

11.3.6.4. Other Collections

As mentioned previously, the **SuperCollider** language provides for many more kinds of data structures. The following Collections are useful, but much more complex than those listed above. For usage instructions, refer to the **SuperCollider** documentation.

- Dictionary: stores and allows retrieval of data by arbitrary Objects (for example, by symbols, rather than by index numbers).
- Library: a type of Dictionary. Objects inserted can be used by any Object in the program, like books in a real-world library can be used by anybody who walks in.
- Set: an unordered Collection of like Objects, where no two elements are identical.
- SortedList: a List where all elements are kept in a sorted order, regardless of how they are added. The inserted Objects should have a useful ordering method, numerical or lexicographic (alphabetic, for example).

11.3.7. Repeated Execution

Repeating boring tasks is one of the main uses of computers, which don't mind doing the same thing over and over again. More importantly, writing code once and using it many times is much more

intelligent than writing the same code many times. Repetition of the same code is often problematic, and repetition with subtle differences is even worse. Errors in this kind of code are difficult to find in the first place, and more difficult to solve effectively. Thankfully, as with most other things, **SuperCollider** offers a wide variety of ways to repeat code without re-writing it.

The code structure used to create repetition is normally called a *loop*. "Do" loops are **SuperCollider**'s most versatile and useful repetition structure, and there are a few different ways to think about and write it. The "while" loop is a standard of most programming languages.

11.3.7.1. "Do This to Everything in This Collection"

One way to write a "do" loop is basically the same as telling the interpreter to "do this Function to every element in this Collection." The syntax looks like this:

```
do(aCollection, aFunction(number, number));
```

... or like this:

```
aCollection.do(aFunction(number, number));
```

This causes `aFunction` to be executed once for each element in `aCollection`, which can be any kind of Collection. Each time `aFunction` is run, it is given two arguments, in this order: an element of `aCollection`, and the element's index number. For Collections that don't have index numbers, it returns what the element's index number would have been. The loop always begins at the start of the Collection, and progresses with each element in order to the end. The second argument, really, is the integers from zero to one less than the number of elements in the Collection, increasing by one each time the loop executes `aFunction`.

11.3.7.2. "Do This, This Many Times"

Another way to write a "do" loop takes advantage of **SuperCollider**'s flexibility, and is really the same as one of the methods above. It's basically equivalent to telling the interpreter to "run this Function this many times." The syntax looks like this,

```
aNumber.do(aFunction(number));
```

This causes `aFunction` to be executed `aNumber` times. The interpreter still provides two arguments to `aFunction`, but they are the same: it is the integers from zero to one less than `aNumber`. You might also think of it as the number of times that `aFunction` has been executed prior to this particular execution.

11.3.7.3. Example "Do" Loops

These examples illustrate different ways to use "do" loops for trivial tasks.

```
(
  var tL = List.new;
  tL = [27, 46, 102, 81, 34, 0, 39, 26, 203, 62];

  do( tL, { arg item, rep; [rep, item].postln; } );

  nil;
)
```

This example is of the first syntax shown. For each element in `tL`, the interpreter executes the function once, giving it *first* the corresponding element of the Collection, and *then* the iteration counter, which happens to be equal to the element's List index number.

```
(
  var tL = List.new;
  var myFunc =
  {
    arg item;
    item.postln;
  };

  tL = [27, 46, 102, 81, 34, 0, 39, 26, 203, 62];

  tL.do( myFunc; );

  nil;
)
```

This example does several things differently, but maintains the same basic functionality as the previous example. In this case, the Function only uses the first argument that the interpreter provides, and completely ignores the iteration counter. The syntax here also puts the Collection outside the parentheses, which perhaps makes it more clear that **tL** is not part of the function.

```
(
  10.do( { "repeat".postln; }; );

  nil;
)
```

This example simply prints the string "repeat" ten times. If the Function accepted one argument, it would receive the integers zero through nine. If it accepted two arguments, both of the arguments would be equal.

11.3.7.4. "Do This While"

"While" loops execute continuously while their "test condition" is evaluated to be "true". Upon reaching the loop, the **SuperCollider** interpreter executes the test condition. If it is "false", the interpreter does not execute the loop, and continues with the code after the loop. If it is "true", the interpreter executes the code in the loop once, then re-executes the test condition. If the test condition is "true", the loop is executed, the test condition re-executed, and so on. Until the test condition returns "false", the interpreter will never leave the loop.

Here is the format of a "while" loop in **SuperCollider**:

```
while(boolean testFunction(number), bodyFunction(number));
```

or like this:

```
testFunction.while(bodyFunction(number));
```

The test condition, called testFunc, is a function which returns a boolean value - either **true** or **false**. The loop's body, called bodyFunc, is a function which can do anything. The loop body function is not provided any arguments by the interpreter. You will have to use comparison operators and boolean expressions when writing the Function for the test condition. For information on how these work in **SuperCollider**, see [Section 11.3.8.1, "Boolean Operators"](#) and [Section 11.3.8.2, "Boolean Expressions"](#).

The following three code blocks are equivalent:


```
(
  10.do( { "repeat".postln; }; );
)
```

and

```
(
  var counter = 0;
  while( { counter < 10; }, { "repeat".postln; counter = counter + 1; } );
)
```

and

```
(
  var counter = 0;
  { counter < 10; }.while( { "repeat".postln; counter = counter + 1; } );
)
```

You can see how it's easier to write this particular activity as a "do" loop. It's often the case that a "do" loop better reflects what you want to do, but not always.

Contemplate a situation where you are waiting for the user to input some information, which you're going to use to calculate the rest of the composition. The following example isn't real code. It's intended to simplify a complex situation, so you can see where a "while" loop makes more sense than a "do" loop.

```
play( some background music );
while( { is the user still inputting information? }, { keep playing music } );
stop( some background music );
```

The background music is begun, and then the interpreter would enter the loop. For as long as the user is still inputting information, the interpreter will then "keep playing music." When the user is not still inputting information, the interpreter will move on to the next command, which stops the music. An equivalent "do" loop would be very difficult to write, if not impossible. This is because we won't know when the user has finished inputting their information until *after* they've finished, so we can't plan in advance for how long to play background music.

Thus, the most appropriate use of a "while" loop is for cases where you cannot know in advance how many times something should be executed. For most other cases of repeated execution, a "do" loop is the most appropriate choice.

11.3.7.5. Other Loops

The default language provides two other loop structures, both of which are designed to iterate over a series of integer values: "for" loops and "forBy" loops. Their use is more limited than "do" loops. They are explained in the **SuperCollider** documentation.

11.3.8. Conditional Execution

Conditional execution tells the **SuperCollider** interpreter to execute code on the condition that something is true. **SuperCollider** offers three conditional execution structures, "if", "switch", and "case" statements. Each of these structures is controlled by one or a series of "boolean

expressions" (sometimes called "conditional expressions"), which are composed of "boolean operators".

11.3.8.1. Boolean Operators

Boolean operators evaluate to **true** or **false**, and are most useful in boolean expressions, where they help to determine which portion of a program to execute.

The following table lists binary boolean operators that take two arguments: one on the left and one on the right. These operators produce either **true** or **false**.

Table 11.1. Binary Boolean Operators in SuperCollider

Operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equivalent
!=	not equivalent
===	identical (the same object)
!==	not identical (not the same object)
&&	logical And
	logical Or

The following table lists unary boolean operators that take one arguments. These operators produce either **true** or **false**.

Table 11.2. Unary Boolean Operators in SuperCollider

Operator	Meaning
isPositive	true if the argument is greater than or equal to 0
isStrictlyPositive	true if the argument is greater than 0
isNegative	true if isPositive is false

Unary operators are actually functions, and must be used as such.

```
(
  var x = 5;
  x.isPositive; // returns "true"
  isNegative( x ); // returns "false"
)
```

The use of these operators is explained below in the "Boolean Expressions" section.

11.3.8.2. Boolean Expressions

Boolean expressions are expressions which, when executed, result in either "true" or "false". Boolean expressions must use at least one boolean operator (as listed above), or a Function which returns a boolean value. Boolean expressions can also use other operators and Functions.

11.3.8.2.1. Simple Expressions

Here are a few simple boolean expressions. Of course, variables can be used in place of constant numbers:

```
5 < 60; // evaluates to "false"
42 != 42; // evaluates to "false"
42 == 42; // evaluates to "true"
0.isPositive; // evaluates to "true"
isNegative( -256 ); // evaluates to "true"
```

11.3.8.2.2. Assignment/Equality Mistake

Beware the following pitfall, common to a large number of programming languages:

```
a == 42; // evaluates to "true" or "false", depending on the value in "a"
a = 42; // assigns the value 42 to "a", over-writing the previously-stored value
```

One possible work-around is to write the number first.

```
42 == a; // evaluates to "true" or "false", depending on the value in "a"
42 = a; // causes an error, because you can't assign a value to a constant number
```

This way, if you accidentally leave out one of the = signs, the interpreter will stop execution and produce an error, rather than continuing with an unexpected assignment.

11.3.8.2.3. Equality versus Identity

The identity-equivalence operators are not usually needed.

```
(
  var a = [12,24,48];
  var b = [12,24,48];

  a == b; // evaluates to "true"
  a === b; // evaluates to "false"
)
```

The == operator evaluates to **true** because **a** and **b** represent equivalent Objects - they are equal. The === operator evaluates to **false** because **a** and **b** represent different instances of the Objects - they are not identical.

```
(
  var a = [12,24,48];
  var b = a;

  a == b; // evaluates to "true"
  a === b; // evaluates to "true"
)
```

In this case, the == operator still evaluates to **true**. The === operator also evaluates to **true**, because **a** and **b** both represent the same Object. When the interpreter evaluates **var b = a;** in the example above, it actually assigns **b** the same value that **a** stores, not a copy of it.

11.3.8.2.4. Logical And and Or

The logical And and Or operators must receive two boolean arguments. Logical And returns **true** if both of its arguments are **true**. Logical Or returns **true** if one of its arguments are **true**.

The following table illustrates how the **SuperCollider** interpreter will evaluate each of the following situations.

Table 11.3. Truth Table

If the left sub-expression is...	... and the right sub-expression is...	... then logical And evaluates to...	... and logical Or evaluates to...
true	false	false	true
false	true	false	true
true	true	true	true
false	false	false	false

The interpreter evaluates the expression on the left first, and then the expression on the right *only* if it will influence the outcome. This means that, if the left-side expression of a logical Or operator evaluates to "true", the interpreter will not test the right-side expression, because the result will be "true" anyway. Similarly, if the left-side expression of a logical And operator evaluates to "false", the interpreter will not test the right-side expression, because the result will be "false" anyway.

This can be exploited to help avoid errors like division-by-zero.

```
(
  var x = 5.rand; // assigns a pseudo-random number between 0 and 5

  ( x != 0 ) && ( { x = 17 / x; } ); // doesn't divide by x if it would cause division-by-zero

  x; // the interpreter automatically prints this value after execution
)
```

If the left-side expression of the logical And operator is "false", the interpreter doesn't evaluate the right-side expression; it simply moves on to the next expression. If the left-side expression is "true" (meaning that x is not zero), then the right-side expression is evaluated. The right-side expression happens to be a Function which assigns "x" the result of dividing 17 by its previous value. The result of the logical And operation is simply discarded in this case - it doesn't really matter to us. This isn't the most straight-forward code, and there are other ways to avoid division-by-zero. If you use this, it's probably best to include a brief explanation of what the code does, as a comment.

If you run this code many times, you will see that it gives many different results - one of which is zero, which proves that the code works as intended. If **SuperCollider** divides by zero, the result is "inf", representing infinity. Try modifying the code so that it will divide by zero, and see what happens.

11.3.8.2.5. Order of Precedence

In complicated boolean expressions, it's important to clarify the order in which you want sub-expressions to be executed. This order is called the "order of precedence," or "order of operations" (see *Order of Operations (Wikipedia)*, available at http://en.wikipedia.org/wiki/Order_of_operations for more information). In computer science, different programming languages enforce different orders of precedence, so you should use parentheses to clarify your intended order, to proactively avoid later confusion. The interpreter will evaluate an expression from left to right, and always fully evaluate parentheses before continuing.

Even simple expression can benefit from parentheses. These produce the same results:

```
(
  var a = 5 == 5 && 17 != 5;
  var b = ( 5 == 5 ) && ( 17 != 5 ); // parentheses help to clarify

  a == b; // evaluates to "true"
)
```

```
(
  var a = 5.isPositive && isNegative( 6 ) || 12 + 5 * 42 - 1 > 18 ;
  var b = ( 5.isPositive && isNegative( 6 ) ) || ( ((12 + 5) * 42 - 1) > 18 ); //
  parentheses help to clarify

  a && b; // evaluates to "true"
)
```

And perhaps even more surprisingly...

```
( 12 + 5 * 42 - 1 ) != ( (12 + 5) * 42 - 1 );
```

... evaluates to "false"! They're equal - the interpreter doesn't follow the standard mathematical order of precedence rules! **SuperCollider** evaluates from left to right, so it's important to clarify to the interpreter what you mean. Where would you put parentheses so that **SuperCollider** evaluates the expression as per the standard mathematical order of precedence rules, with multiplication before addition and subtraction?

11.3.8.3. "If This Is True, Then... "

The "if" structure is provided a boolean expression and two Functions. If the expression evaluates to "true", it executes one Function. If the expression evaluates to "false", it executes the other Function.

Here are the two ways to write an "if" structure:

```
if ( booleanExpression, trueFunction, falseFunction );
```

and

```
booleanExpression.if( trueFunction, falseFunction );
```

It's possible to exclude the `falseFunction`, which is like telling the interpreter, "If the boolean expression is true, then execute this Function. Otherwise, don't execute it."

```
(
  var test = [true,false].choose; // pseudo-randomly chooses one of the elements in the List

  if ( ( true == test ), { "It's true!".postln; }, { "It's false!".postln; } );
  nil;
)
```

This example prints out a nice message, saying whether **test** is **true** or **false**. Because **test** is already a boolean value, we don't need to include it in an expression. The "if" statement could have been shortened like this: `if (test, { "It's true!".postln; }, { "It's false!".postln; });`

Suppose we only wanted to be alerted if **test** is **true**.

```
(
  var test = [true,false].choose; // pseudo-randomly chooses one of the elements in the List

  test.if( { "It's true!".postln; } );
  nil;
)
```

In this example, the alternate "if" syntax is used, where the boolean expression is placed before the parentheses.

"If" structures can also be "nested," which is like telling the interpreter, "If this is true, do this; otherwise if this is true, do this; otherwise if this is true, do this." In this relatively simple example of nesting, the interpreter evaluates each "if" structure only if the previous one was "false".

```
(
  var test = [1,2,3].choose; // pseudo-randomly chooses one of the elements in the List

  if( 1 == test, { "It's one!".postln; },
    if( 2 == test, { "It's two!".postln; },
      if( 3 == test, { "It's three!".postln; } ) ) );

  nil;
)
```

This is a more complex example of nesting:

```
(
  var testA = [1,2,3].choose; // pseudo-randomly chooses one of the elements in the List
  var testB = [1,2,3].choose;

  if( 1 == testA, { ( 1 == testB ).if( { "It's one and one!".postln; },
    ( 2 == testB ).if( { "It's one and two!".postln; },
      ( 3 == testB ).if( { "It's one and three!".postln; } ) ) ); },
    if( 2 == testA, { ( 1 == testB ).if( { "It's two and one!".postln; },
      ( 2 == testB ).if( { "It's two and two!".postln; },
        ( 3 == testB ).if( { "It's two and three!".postln; } ) ) ); },
        if( 3 == testA, { ( 1 == testB ).if( { "It's three and one!".postln; },
          ( 2 == testB ).if( { "It's three and two!".postln; },
            ( 3 == testB ).if( { "It's three and
three!".postln; } ) ) ); } ) ) ); } ) );

  nil;
)
```

As you can see, this type of nesting is not easy to figure out - from the standpoint of the original programmer or somebody else who wishes to use your code. In writing this example, it took me several attempts before getting the parentheses and braces right. Usually, if you have a long list of possibilities to test (like the nine in this example), it is better to use a "case" or "switch" structure. Not only does this help to make the code easier to understand, but the **SuperCollider** interpreter can apply optimizations that make the code run marginally faster.

11.3.8.4. "Switch Execution to This Path"

A "switch" structure can be most easily understood by comparison to a switch in a railway line. As a train approaches a railway switch, an operator inspects the train, and decides whether it should be going to the passenger station, the freight station, or the garage for storage. A "switch" structure - like a railways switch, can only act on one Object at a time, but the Object can be a Collection, allowing

you to compare multiple things at once. Each case is tested with the boolean equality operator before being executed.

Here is the syntax of a "switch" statement:

```
case( compareThis,
    toThis1, { doThis; },
    toThis2, { doThis; },
    toThis3, { doThis; }
);
```

You can include any number of cases. Notice that there is no comma after the last case, and that I've put the concluding ")," on a separate line with the same indentation as the word "case", so that it's easy to see.

The following example shows a simple switch.

```
(
    var grade = 11.rand + 1; // pseudo-randomly chooses 0 to 11, then adds 1 to give 1 to 12

    grade =
    switch( grade,
        1, { "D-" },
        2, { "D" },
        3, { "D+" },
        4, { "C-" },
        5, { "C" },
        6, { "C+" },
        7, { "B-" },
        8, { "B" },
        9, { "B+" },
        10, { "A-" },
        11, { "A" },
        12, { "A+" }
    );

    ("Your grade is" + grade).postln;
    nil;
)
```

The code picks a pseudo-random number between 1 and 12, then uses a "switch" structure to convert that number into a letter-grade, assigning it to the same **grade** variable. Then, it adds the "Your grade is" string to the value of **grade** (with a space between), and prints that result.

This example avoids the complex nested "if" structure from above.

```
(
    var testA = [1,2,3].choose; // pseudo-randomly chooses one of the elements in the List
    var testB = [1,2,3].choose;

    switch( [testA,testB],
        [1,1], { "It's one and one!".postln; },
        [1,2], { "It's one and two!".postln; },
        [1,3], { "It's one and three!".postln; },
        [2,1], { "It's two and one!".postln; },
        [2,2], { "It's two and two!".postln; },
        [2,3], { "It's two and thre!".postln; },
        [3,1], { "It's three and one!".postln; },
        [3,2], { "It's three and two!".postln; },
        [3,3], { "It's three and three!".postln; }
    );
```



```
);
    nil;
)
```

This is an elegant way to inspect two otherwise-separate variables. Remember that the first argument to "switch" (in this case, it's **[testA, testB]**) is compared to the first argument of possible result with the equality operator: **==**

When evaluating which switch to use, the **SuperCollider** interpreter will always apply the *last* one that evaluates to **true**.

```
(
  switch( 5,
    5, { "one".postln; },
    5, { "two".postln; },
    5, { "three".postln; }
  );
  nil;
)
```

All of these cases are true, but this will always result in "three" being printed.

11.3.8.5. "In This Case, Do This"

"Case" and "switch" structures look similar, but work in subtly different way. A "switch" structure is like a railway switch, allowing one train to be routed onto the right track, according to qualities of the train. A "case" structure, on the other hand, works like somebody trying to decide how to get to work. The person might ask themselves how far they are going, how long they have to get to work, how fast the available options are, what the available options cost, and so on. While in a "switch" structure, the path of execution is determined by examining only one Object, a "case" structure determines the path of execution based on any number of things.

Here is the syntax of a "case" structure:

```
case
  <replaceable>booleanFunction</replaceable> <replaceable>resultFunction</replaceable>
  <replaceable>booleanFunction</replaceable> <replaceable>resultFunction</replaceable>
  <replaceable>booleanFunction</replaceable> <replaceable>resultFunction</replaceable>
  <replaceable>booleanFunction</replaceable> <replaceable>resultFunction</replaceable>
;
```

Contemplate the following pseudo-code example, which represents a possible musical situation, and a good use of the "case" structure.

```
(
  var coolFunction =
  {
    case
      { is there no music playing?
        AND people are in the room } { play music }
      { has the same song been playing for too long?
        OR is the song boring? } { change the song }
      { has everybody left the room? } { turn off the music }
      { has a song been requested? } { change to that song }
      { is the music too loud? } { lower the music's volume }
  }
)
```

```

    { is the music too quiet? } { raise the music's volume }
    { is the music too fast? } { lower the music's tempo }
    { is the music too slow? } { raise the music's tempo }
    { is everything okay? } { wait for 10 seconds }
  ;
};

( 5 == 5 ).while( coolFunction ); // executes coolFunction continuously
)

```

It might seem like this example doesn't relate to a real **SuperCollider** programming situation, but in fact it might. If you could program Function's which determined all of those questions and all of the answers, this sort of "case" structure would be very helpful in a situation where a computer running **SuperCollider** were left in a room by itself, and expected to play music whenever anybody entered the room. Since five is always equal to five, the interpreter will run coolFunction forever. If the music needs adjustment in some way, the Function will adjust the music. If everything is okay, then the interpreter will wait for 10 seconds, and then the loop will cause the Function to be re-evaluated. Because many different criteria are evaluated in the "case" structure, this represents an efficient use of the structure.

"Case" structures can be used to do the same thing as "switch" structures, but it's usually less elegant solution. Also, it doesn't allow the interpreter to use an speed optimization that it would have used in an equivalent "switch" structure.

```

(
  var grade = 11.rand + 1; // pseudo-randomly chooses 0 to 11, then adds 1 to give 1 to 12

  grade =
  case
    { 1 == grade; } { "D-" }
    { 2 == grade; } { "D" }
    { 3 == grade; } { "D+" }
    { 4 == grade; } { "C-" }
    { 5 == grade; } { "C" }
    { 6 == grade; } { "C+" }
    { 7 == grade; } { "B-" }
    { 8 == grade; } { "B" }
    { 9 == grade; } { "B+" }
    { 10 == grade; } { "A-" }
    { 11 == grade; } { "A" }
    { 12 == grade; } { "A+" }
  ;

  ("Your grade is" + grade).postln;
  nil;
)

```

This example is equivalent to one of the "switch" structure examples. This is not a good use of the "case" structure, because it requires a lot of code repetition.

Unlike a "switch" structure, a "case" structure will always follow the *first* case that evaluates to "true".

```

(
  case
    { 5 == 5; } { "one".postln; }
    { 5 == 5; } { "two".postln; }
    { 5 == 5; } { "three".postln; }
  ;

  nil;
)

```

```
)
```

This example will always result in "one" being printed.

11.3.9. Combining Audio; the Mix Class

One of the requirements of multi-channel audio is the ability to combine a large number of UGen's into a small number of channels - normally just two. The **SuperCollider** interpreter allows you to accomplish this in a number of ways, which are explained here.

11.3.9.1. The "Mix" Class

The "Mix" Class allows you to combine a mutli-channel Array into one channel. It's just that simple: you put in an Array of UGens, and out comes a single-channel combination of them.

Here are the two possible syntaxes for the "Mix" Class:

```
Mix.new(ArrayOfUGens);
```

and

```
Mix(ArrayOfUGens);
```

The second form is simply a short-hand version of the first. The **Mix** Class doesn't really create "Mix" Objects either - it's just a Function that combines many UGen's into one.

Here's an example of the "Mix" Class in action:

```
{
  Mix( [SinOsc.ar(220, 0, 0.1),
        SinOsc.ar(440, 0, 0.1),
        SinOsc.ar(660, 0, 0.1),
        SinOsc.ar(880, 0, 0.1),
        SinOsc.ar(850, 0, 0.1),
        SinOsc.ar(870, 0, 0.1),
        SinOsc.ar(880, 0, 0.1),
        SinOsc.ar(885, 0, 0.1),
        SinOsc.ar(890, 0, 0.1),
        SinOsc.ar(1000, 0, 0.1)] );
}.play;
```

Notice how all of these **SinOscs** are heard through the left channel only. The **Mix** class mixes all the UGen's together into one. You could use a bus to send the audio to both the left and right channels. What happens if we don't use the **Mix** class? Try to remove the function, and find out. You only hear some of the **SinOsc**'s. Which ones? The first two, representing the left and right channels. If your audio interface has more than two channels, you may be able to hear more than those first two channels.

There is another function offered by the **Mix** class, and it is a kind of loop. The function is called **Fill**, and it takes two arguments: the number of times to run a function, and the function to run. The function is provided with one argument (like in a "do" loop), which is the number of times the function has *already* been run.

```
(
  var n = 8;
  var sineFunc =
  {
```

```

    arg iteration;

    var freq = 440 + iteration;
    SinOsc.ar( freq:freq, mul:1/n );
  };

  { Mix.fill( n, sineFunc ); }.play;
)

```

As you can see, the `fill` function itself is quite simple: you provide the number of UGen's to create, and a function that creates UGen's. It's the **sineFunc** function that is a little confusing. The argument is called "iteration", because it holds how many times the function has already been run - how many iterations have happened already. It uses this value to help calculate the frequency (stored in a variable called **freq**), and then creates a **SinOsc** UGen. The **mul** argument helps to automatically control the volume level. Since the total volume should be no more than 1.0, the `sineFunc` function calculates UGen's' volume by dividing 1, the maximum level, by the number of UGen's that will be created. The slowly pulsating volume is part of the acoustic result of this many frequencies being so close together - it is not a hidden effect by **SuperCollider**.

11.3.9.2. Arrays of Arrays

There is another way to combine many UGen's into two channels for stereo output: rather than sending the Array's to the Mix class, combine them into a two-element Array.

```

{
  [
    [ SinOsc.ar(440, 0, 0.1), SinOsc.ar( 880, 0, 0.1 ), SinOsc.ar( 1660, 0, 0.1 ) ],
    [ SinOsc.ar(440, 0, 0.1), SinOsc.ar( 880, 0, 0.1 ), SinOsc.ar( 1660, 0, 0.1 ) ]
  ];
}.play;

```

Here, a two-element Array is the result of a Function, which gets sent the "play" message. Each of the elements is an equivalent, three-element Array where each element is a UGen.

This representation also offers another benefit: each UGen can have a different "mul" value, which will be preserved.

```

{
  [
    [ SinOsc.ar(440, 0, 0.2), SinOsc.ar( 880, 0, 0.1 ), SinOsc.ar( 1660, 0, 0.05 ) ],
    [ SinOsc.ar(440, 0, 0.2), SinOsc.ar( 880, 0, 0.1 ), SinOsc.ar( 1660, 0, 0.05 ) ]
  ];
}.play;

```

This sounds much less harsh than the first example. Try it with the Mix Class. Even with the different "mul" values, it sounds the same as the first example! This helps Mix to ensure that the total level doesn't exceed 1.0, but it has the disadvantage that careful level-balancing on your part will be erased.

11.3.9.3. Addition

This method of combine UGen's into two channels uses the addition operator: +

```

{
  [
    ( SinOsc.ar(440, 0, 0.1) + SinOsc.ar( 880, 0, 0.1 ) + SinOsc.ar( 1660, 0, 0.1 ) ),
    ( SinOsc.ar(440, 0, 0.1) + SinOsc.ar( 880, 0, 0.1 ) + SinOsc.ar( 1660, 0, 0.1 ) )
  ];
}

```

```
];
}.play;
```

Notice that, like with the Mix Class, independent "mul" levels are not preserved.

11.3.10. SynthDef and Synth

The preceding sections of this "Basic Programming" guide have only created sound with Function's. The truth is that Function's are very useful for creating sound, but they represent a simplification of the actual commands and Functions that must be run by the interpreter in order to create sound.

```
// When you write this...
(
  { SinOsc.ar( freq:440, mul:0.2 ); }.play;
)

// The interpreter actually does this...
(
  SynthDef.new( "temp__963", { Out.ar( 0, SinOsc.ar( freq:440, mul:0.2 ); } ).play;
)
```

Yikes! Don't despair - it's easy to understand - it just looks scary!

11.3.10.1. "Out" UGen

The "Out" UGen is one of the bits of magic automatically taken care of by the interpreter. It routes an audio signal from another UGen into a specific output (actually, into a specific bus - see [Section 11.3.11, "Busses"](#)).

The following examples have the same effect:

```
{ SinOsc.ar( freq:500, mul:0.2 ); }.play;
```

and

```
{ Out.ar( 0, SinOsc.ar( freq:500, mul:0.2 ); } ).play;
```

The first argument to "Out.ar" is the bus number for where you want to place the second argument, which is either a UGen or a multi-channel Array of UGen's. If the second argument is an Array, then the first element is sent to the first argument's bus number, the second argument is sent to one bus number higher, the third to two bus numbers higher, and so on. This issues is explained fully in [Section 11.3.11, "Busses"](#), but here is what you need to know for now, working with stereo (two-channel) audio:

- If the second argument is a two-element Array, use bus number 0.
- If the second argument is a single UGen, and you want it to be heard through the left channel, use bus number 0.
- If the second argument is a single UGen, and you want it to be heard through the right channel, use bus number 1.

If you're still struggling with exactly what the "Out" UGen does, think of it like this: when you create an audio-rate UGen, it starts creating an audio signal; the "Out" UGen effectively connects the audio-rate UGen into your audio interface's output port, so it can be heard through the speakers. In [Section 11.3.11, "Busses"](#), it becomes clear that there are, in fact, other useful places to connect an audio-rate UGen (through an effect processor, for example), and the "Out" UGen can help you do that.

11.3.10.2. SynthDef

A SynthDef is what we use to tell the server how to create sound. In order to truly understand what SynthDef accomplishes, we need to recall the disconnect between the interpreter and the server. In reality, the interpreter has no idea how to make sound or work with audio hardware. The server, likewise, has no understanding at all of the **SuperCollider** language. The interpreter takes the code that we write, and does one of a number of things, depending on the nature of the code:

- executes it completely,
- executes it partially, makes choices, and then does something else
- send the server information about how to synthesize sound,
- etc.

For simple code like **2.postln;** the interpreter just executes it. For code like **{ SincOsc.ar; }.play;** the interpreter expands it a bit, then sends instructions to the server, which deals with the rest of the synthesis process.

A SynthDef is part of this last process; SynthDef Objects represent the synthesis information that is sent to the server before (or at the same time as) telling the server to play the sound.

There are two steps to creating a useful SynthDef: making an interpreter Object, and sending the actual synthesis information to the server. There are two ways to write this, as follows:

```
someVariable = SynthDef.new( nameOfSynthDef, FunctionContainingOutUGen );
someVariable.send( nameOfServer );
```

and

```
SynthDef.new( nameOfSynthDef, FunctionContainingOutUGen ).send( nameOfServer );
```

The FunctionContainingOutUGen is simply that - a function that, when executed, returns an **Out** UGen (meaning that the **Out** UGen must be the last expression in the function). The **nameOfSynthDef** should be a symbol (as described in [Section 11.3.10.4, "Symbols"](#)), but can also be a string. The **nameOfServer** is a variable that represents the server to which you want to send the SynthDef's information; unless you know that you need to use a different variable for this, it's probably just the letter "s", which the interpreter automatically assigns to the default server.

Here is a demonstration of both methods:

```
(
  var playMe =
  {
    Out.ar( 0, SinOsc.ar( freq:440, mul:0.2 ) );
  };

  var playMeSynthDef = SynthDef.new( \playMe, playMe );

  playMeSynthDef.send( s );

  nil;
)
```

and

```
(
  var playMe =
  {
    Out.ar( 0, SinOsc.ar( freq:440, mul:0.2 ) );
  };

  SynthDef.new( \playMe, playMe ).send( s );

  nil;
)
```

The only advantage to assigning something to a variable is the ability to refer to it later. If you use the first method, then you can send the SynthDef to more than one server. Since it's rare that you will want to use more than one server, it's usually better to use the second style. In fact, if you won't be using the "playMe" Function again, you don't need to assign it to a variable!

```
SynthDef.new( \playMe, { Out.ar( 0, SinOsc.ar( freq:440, mul:0.2 ) ); } ).send( s );
```

This is all that's really needed to create and send a synthesis definition to the server. It looks long and frightening, but now at least you understand what all of the parts do.

11.3.10.3. Load

There is another way to send a SynthDef to the server: the "load" Function. The "send" Function sends the synthesis information to the server, which stores it in memory. When the server stops running, all synthesis information given with "send" is lost. The "load" Function, on the other hand, sends the synthesis information to the server, which stores it on disk and in memory. Every time the server is started, it loads all of the synthesis information previously sent to it with the "load" Function. The definition remains until you delete it specifically. This is most useful for a SynthDef that takes up a lot of memory, and which would use considerable network time to transfer to the server whenever the server is run. It is also useful to use the "load" Function instead of "send", when there are a lot of SynthDef's, regardless of the size of each one. The idea is the same: avoid sending the SynthDef in order to save time.

The syntax and usage for "load" is the same as for "send".

11.3.10.4. Symbols

As stated in the section about variables, a symbol is simply something which represents something else. When used in the context of a SynthDef, a symbol is a string of characters that refers to a SynthDef that we've already sent to the server. What wasn't mentioned in the section about variables is that, in addition to the symbols that can be used as variable names, the **SuperCollider** language provides a distinct data-type (like numbers or strings) for symbols. Many programming languages don't provide a "symbol" data-type, so many programmers do not use them extensively, but they are very handy for situations like this. As local variable names are symbols representing data stored by the interpreter, here we are using the symbol data-type to refer to data stored on the server.

Symbols are a better way to name SynthDef's than strings. Not only do symbols take up less memory, they aren't actually the interpreter doesn't actually think of them as Objects, and neither should you. Symbols are universally unique; only one instance of a symbol with the same characters can exist. On the other hand, an infinite number of strings with the same characters can exist. When we use a symbol, we are defining it universally. When we use a string, the server pretends that all strings with the same characters are the same philosophical object, even though they aren't. This isn't a technical problem, but it can be difficult to think about, and is cognitively dissonant.

If all this seems a little abstract and ontological, that's because it is.

11.3.10.4.1. Symbols: the Easy Way

Symbols are things that you should use to identify a SynthDef sent to the server.

11.3.10.4.2. Writing Symbols

There are two ways to write out a symbol: between single-quotation marks, and after a back-slash. Symbols given between single-quotation marks can contain any characters but a single-quotation mark. Symbols given after a back-slash can contain any characters but a space. Neither type of symbol name can cross onto a new line.

The following example contains some valid symbols, and some invalid symbols.

```
\stopSign \\ this is a symbol
\stop sign \\ this is a symbol called 'stop' followed by the unrelated word 'sign'
'stopSign' \\ this is a symbol
'stop sign' \\ this is a symbol
'stop
sign' \\ these lines are not a symbol, and will cause an error
```

The following example illustrates the differences between strings and symbols.

```
var a = "stop sign" \\ a string
var b = "stop sign" \\ a string with the same letters as the first string
a == b; \\ returns "true" because the strings are equivalent
a === b; \\ returns "false" because the strings are separate copies with the same characters

var c = 'stop sign' \\ a symbol
var d = 'stop sign' \\ the same symbol
c == d; \\ returns "true" because the symbols are equivalent
c === d; \\ returns "true" because the symbols are identical
```

11.3.10.5. SynthDef Becomes Synth

After you send a SynthDef to the server, you can put it into action. When the interpreter tells the server to play a synthesis definition (which the interpreter holds in a SynthDef Object), the server creates a synth from the definition, and starts generating sound. The interpreter gives us a Synth Object to represent each synth on the server, so that we can control the synth.

This is the syntax used to create a new synth, after its definition has been sent to the server.

Synth.new(nameOfSynthDef);

The name will be a symbol or a string - whatever you supplied when you ran the SynthDef.new() function.

Because we're creating a new Synth Object, we should assign it to a variable, for later reference.

```
SynthDef.new( \playMe, { Out.ar( 0, SinOsc.ar( freq:440, mul:0.2 ) ); } ).send( s );
var mySynth = Synth.new( \playMe );
```

Recall that the interpreter automatically uses the Synth and SynthDef Classes when we send the "play" message to a Function. We can actually capture and use the Synth Object created from "playing" a Function, too. This example is almost the same as the previous one.

```
var mySynth = { SinOsc.ar( freq:443, mul:0.2 ); }.play;
```

The difference is subtle: after the second example, we have no control over what name the interpreter gives to the SynthDef that it sends to the server, so we can't re-use the SynthDef. On the other hand, because we assign the name `\sillyTutorialSD` to the SynthDef in the first example, we know what it's called, and we can re-use it. Theoretically, we can make an infinite number of synths from this single definition. Realistically, it's limited by the amount of memory the server can use; for most modern computers, this number is so high that we don't ever need to worry about it.

As usual, the interpreter provides us with an optional short-cut:

```
var mySynth = SynthDef.new( \playMe, { Out.ar( 0, SinOsc.ar( freq:440, mul:0.2 ) ); } ).play;
```

This automatically sends the synthesis information to the server, creates a synth, and plays it. What minor functionality is lost when we use this shortcut?

11.3.10.6. Shortcomings of a SynthDef

Consider the following program:

```
(
  var myRandFunc =
  {
    var frequency = 440.rand + 440; // produces an integer between 440 and 880
    SinOsc.ar( freq:frequency, mul:0.025 );
  };

  10.do( { myRandFunc.play; } );
)
```

Execute the program a few times. The result will be different each time: ten different SinOsc's with ten different frequencies.

What if we convert the program to use a SynthDef and multiple Synth's instead? This program will probably cause an error the first time - this is explained below.

```
(
  var myRandFunc =
  {
    var frequency = 440.rand + 440; // produces an integer between 440 and 880
    Out.ar( 0, SinOsc.ar( freq:frequency, mul:0.025 ) );
  };

  SynthDef.new( \myRandFunc, myRandFunc ).send( s );

  10.do( { Synth.new( \myRandFunc ); } );
)
```

Execute the program a few times. The result is still different each time, but it's the *same* ten SinOsc's, all with the same frequency. This is the nature of a SynthDef: once it's sent to the server, you can create a synth from the same instructions without resending them.

This program causes an error the first time you run it. Inspect the error messages, and see if you can determine why. It's because the server is processing commands *asynchronously*: things don't happen right when the interpreter asks, but very shortly thereafter. The result is that the server is asked to make a new synth before it deals with the synth definition. There are ways to get around this, but they're too complex for this section - for now (to simplify this text's examples), just accept that the error may happen the first time you run a Synth.

11.3.10.7. Creating Change Anyway

The way to create change and pseudo-randomness anyway is to incorporate another UGen to do it for you. Remember: when you send synthesis information to the server, that information can't change unless you replace it. This doesn't mean that the output produced by the synth can't change!

The way to do this is with control-rate UGen's. The following example uses a control-rate SinOsc to set the frequency of an audio-rate SinOsc.

```
(
  var myRandFunc =
  {
    var frequency = SinOsc.kr( freq:0.5, add:660, mul:220 ); // oscillates between 440 and
    880, hitting each extreme every 2 seconds
    Out.ar( 0, SinOsc.ar( freq:frequency, mul:0.2 ) );
  };

  SynthDef.new( \myRandFunc, myRandFunc ).send( s );

  Synth.new( \myRandFunc );
)
```

When you use a UGen as a control-rate UGen, you have to think about its arguments quite differently than when using it as an audio-rate UGen. This table shows how the same argument gives a different result for an audio-rate vs. control-rate UGen used for pitch:

Table 11.4. Parameters in Audio-Rate and Control-Rate SinOsc UGens

Parameter	In an audio-rate UGen...	In a control-rate UGen...
freq	controls the pitch	controls the speed of oscillation
add	??	sets the middle point of the sine wave by adding this to the output
mul	controls volume level	sets the deviation from "add"

For an audio-rate SinOsc UGen, you set the frequency and the volume level. For a control-rate UGen, you set the mid-point of oscillation with **add**, the extremes of oscillation which will be **add - mul** and **add + mul**, and the speed of oscillation with **freq**. The end result is very different numbers.

There is a handy UGen designed specifically for replacing pseudo-randomness in Functions. The following example restores the "ten different pitches" to the example from the last section.

```
(
  var myRandFunc =
  {
    var frequency = Rand( 440, 880 ); // produces an integer between 440 and 880
    Out.ar( 0, SinOsc.ar( freq:frequency, mul:0.025 ) );
  };

  SynthDef.new( \myRandFunc, myRandFunc ).send( s );

  10.do( { Synth.new( \myRandFunc ); } );
)
```

If you run this multiple times, you will again hear ten different pitches. Depending on audio hardware, previous musical experience, and other factors, some people may have difficulty hearing that the pitches are different. Try reducing the number of synths created in the loop.

11.3.10.8. SynthDef with Arguments

There are some situations where you simply cannot pre-determine all of the material that you're going to use when creating a synth. It might be easier to resort to using a Function rather than a SynthDef, but there is yet another solution - creating an argument in your SynthDef Function.

With only a subtle change to our Function, we can add the possibility of passing arguments on synth creation:

```
var myRandFunc =
{
  arg frequency = Rand( 440, 880 ); // default value between 440 and 880
  Out.ar( 0, SinOsc.ar( freq:frequency, mul:0.025 ) );
};
```

I've decided to use the Rand UGen anyway, so that supplying a frequency is optional. This adds functionality while making the added complexity optional:

```
(
  var myRandFunc =
  {
    arg frequency = 440;
    Out.ar( 0, SinOsc.ar( freq:frequency, mul:0.025 ) );
  };

  SynthDef.new( \myRandFunc, myRandFunc ).send( s );

  10.do( { Synth.new( \myRandFunc ); } );
)
```

If you use the SynthDef in the old way, as in the example, you'll get the expected result: ten Synth's, all with the same frequency. But, if you add a "rand" Function call into the loop, you can get ten different frequencies!

```
(
  var myRandFunc =
  {
    arg frequency = 440;
    Out.ar( 0, SinOsc.ar( freq:frequency, mul:0.025 ) );
  };

  SynthDef.new( \myRandFunc, myRandFunc ).send( s );

  10.do( { Synth.new( \myRandFunc, [\frequency,(440.rand + 440)] ); } );
)
```

Notice how we supply arguments: an Array with elements alternating between a string-quoted parameter name, and the value of the argument itself. If we "parameterized" all three main fields of the SinOsc, we could supply them like this:

```
Synth.new( \mySinOsc, [\freq,440,\add,0,\mul,0.2] );
```

11.3.10.9. Things to Do with a SynthDef: Set and Free

Once you send a synth definition to the server, and make some synths, you've collected a few Synth Objects, and you wonder what to do with them next. Of course, you could listen to them, but you can also change the arguments that you used, and stop it.

To change the arguments used by a synth, send it the "set" message, with a list of arguments:

```
variableHoldingSynth.set( [\argument1,value,\argument2,value,...] );
```

This helps to save even more time and memory: rather than destroying and creating synths all the time, you can simply change pre-existing ones.

This modification of the ten-pseudo-random-tones example includes an extra line that lets you change the tones without destroying and re-creating the synths.

```
// run this first
h = List.new;

// run this second
(
  var myRandFunc =
  {
    arg frequency = 440;
    Out.ar( 0, SinOsc.ar( freq:frequency, mul:0.025 ) );
  };

  SynthDef.new( \myRandFunc, myRandFunc ).send( s );
)

// run this third
10.do( { h.add( Synth.new( \myRandFunc, [\frequency,(440.rand + 440)] ) ); } );

// run this fourth, as many times as you please
h.do( { arg item; item.set( \frequency, (440.rand + 440) ); } );
```

The reason that you have to run each of those segments separately is two-fold: we need to store the List of Synth's in a single-letter variable because, for this simple demonstration, this is the most efficient way; second, for the asynchronous behaviour of the server that was previously noted as causing an error.

The only aspect of that example that's a little tricky to understand is the "do" loop. Remember that when you run a "do" loop on a List, the interpreter automatically loops over each of the elements in the List, running the Function that you provide. Each time the Function is run, it receives the current List item, and its index number in the List, in that order. So the Function in this loop simply uses "set" to change the "frequency" argument.

Take special note that the arguments in this case are not identical to those given with the "new" message. Compare the two forms below:

```
SynthDef.new( \SynthName, [\parameter1,value,\parameter2,value] );
```

and

```
existingSynth.set( \parameter1, value, \parameter2, value );
```

To get rid of one synth without stopping all sound, send its corresponding Synth the "free" message:

```
variableHoldingSynth.free;
```

This stops the synth and frees the associated memory - on the server. Your Synth Object still exists in the interpreter, but you can't use it any more. A Synth Object represents a synth on the server; since you got rid of the synth on the server, the Synth Object represents something that doesn't exist. If you

attempt to send the "free" message again, you'll get an error. For this reason, it's a good idea to get rid of the Synth Object at the same time:

```
<replaceable>variableHoldingSynth</replaceable>.free;
<replaceable>variableHoldingSynth</replaceable> = nil;
```

If you accidentally send "free" to an already-freed Synth, the interpreter will cause an error, and program execution will stop. If you accidentally send "free" to a variable set to "nil", nothing will happen. Proactively avoiding mistakes like this is good programming practice.

11.3.11. Busses

SuperCollider busses work just like busses in other audio creation contexts, which work similarly to busses used to transport humans. Busses are used to send audio from one place to another, and in **SuperCollider** they can also be used to send control-rate signals. Each **SuperCollider** bus is given an index number, which are integers starting at 0. Audio-rate busses and control-rate busses are independent, and are given an independent set of index numbers. Any number of unique signals can be routed into a bus, and any number of receivers can take signals from a bus - but the signal will be the sum of all the input signals. In other words, if you want to send two different sets of signals, you need two different busses with different index numbers.

11.3.11.1. Audio-Rate Bus Numbers

There are special audio-rate busses reserved automatically by the server. These interact with the audio interface, allowing you to get sound from its inputs, and send sound to its outputs. The lowest audio-rate bus numbers are reserved for audio interface outputs, each channel receiving an independent bus. The bus numbers just above those are reserved for audio interface inputs, each channel receiving an independent bus.

For a simple audio interface with two channels each for input and output, the pre-reserved audio-rate bus numbers would be these:

- 0 for left channel output
- 1 for right channel output
- 2 for left channel input
- 3 for right channel input

Now you can change the first argument of the "Out" UGen as desired!

11.3.11.2. Out and In UGens

The "Out" UGen is discussed in [Section 11.3.10.1, "Out UGen"](#). What it does is take a signal and route it to the specified bus number. The "In" UGen performs a similar action: take a signal from a bus number, and make it available for use.

This is the syntax to use for the **Out** UGen:

```
Out.ar(busNumber, audioRateUGen);
```

or

```
Out.kr(busNumber, controlRateUGen);
```

The UGen enclosed here should not be enclosed in a Function. If the UGen provides multi-channel output, "Out" will automatically route the lowest channel to the specified bus number, the next channel

to the next highest bus number, and so on. This way, you can achieve stereo output with one "Out" UGen.

This is the syntax to use for the **In** UGen:

```
In.ar(busNumber, numberOfChannels);
```

or

```
In.kr(busNumber, numberOfChannels);
```

Whereas "Out" automatically outputs the right number of channels based on how many you provide, "In" requires that you specify how many channels you want it to fetch for you.

When created with this form, both of these UGens automatically connect to the default server, stored in the single-letter "s" variable by the interpreter.

11.3.11.3. Bus Objects

As SynthDef and Synth represent things stored on the server, the interpreter provides us with an instantiable "Bus" Class that represents the server's busses. In many programs that you write, you won't need to use a Bus Object - particularly when you're doing simple input and output with the automatically-reserved bus numbers. But, like the SynthDef and Synth Classes make it easier to deal with synthdefs and synths on the server (which are very difficult to deal with directly), the Bus Class makes it easier to deal with busses on the server, and provides some extra functionality to boot!

The primary advantage of using Bus Objects is that you don't have to keep track of bus numbers, whether they're being used, and how many channels are being routed. For simple input and output of audio-rate signals, you're better off simply remembering the bus numbers

The **new** message creates a new Bus object. This is the syntax:

```
Bus.audio(serverName, numberOfChannels);
```

or

```
Bus.control(serverName, numberOfChannels);
```

The interpreter takes "numberOfChannels" busses on the "serverName" server, and groups them together for multi-channel use in one Bus Object, which it returns to you. The "numberOfChannels" argument is optional; if you leave it out, the Bus Object will have only one bus, for single-channel signals. You should always assign the object to a variable, or else you have no way to use the bus later:

```
var myBus = Bus.audio( s, 2 );
```

The interpreter also keeps track of which bus numbers are used for which Bus Objects, so the signals will never get confused. Of course, you can still route signals through those bus numbers without using the Bus Object, but the Bus Class helps us to keep things straight.

The following messages/functions can also be used with Bus Objects:

Table 11.5. Some Functions of the **Bus** Class

Message/Function	Example	Return Value
index	b.index;	The lowest bus number used by this Bus object.
numChannels	b.numChannels;	The number of busses used by this Bus object.
rate	b.rate;	Either audio or control .

Message/Function	Example	Return Value
server	b.server;	The name of the server used by the Bus object. The default server is localhost

When you are done with a Bus, you can release the channels for use by other Bus Objects:

```
<replaceable>myBusVariable</replaceable>.free;
<replaceable>myBusVariable</replaceable> = nil;
```

Like when sending the "free" message to a Synth Object, you should set the variable name of a "free'd" Bus to "nil". This will prevent you from accidentally sending audio there after the Bus is released.

11.3.11.4. Using Busses: Control-Rate Example

The best way to understand how and when to use a bus is to see them in action.

```
( // execute first: prepare the server
  var busAudioSynth =
  {
    arg bus, freqOffset = 0;

    Out.ar( 0, SinOsc.ar( freq:( In.kr(bus) + freqOffset ), mul:0.1 ) );
  };

  var busControlSynth =
  {
    arg bus, freq = 400;

    Out.kr( bus, SinOsc.kr( freq:1, mul:( freq/40 ), add:freq ) );
  };

  SynthDef( \tutorialAudioBus, busAudioSynth ).send( s );
  SynthDef( \tutorialControlBus, busControlSynth ).send( s );

  b = Bus.control( s );
)

( // execute second: create synths
  x = Synth.new( \tutorialControlBus, [\bus, b] ); // control synth
  y = Synth.after( x, \tutorialAudioBus, [\bus, b] ); // low audio synth
  z = Synth.after( x, \tutorialAudioBus, [\bus, b, \freqOffset, 200] ); // high audio synth
)

( // commands to free each Object
  x.free; x = nil; // control synth
  y.free; y = nil; // low audio synth
  z.free; z = nil; // high audio synth
  b.free; b = nil; // control bus
)
```

This example contains three stages: prepare the server, create the synths, destroy the synths. These three stages will become familiar as you program in **SuperCollider**, whether or not you use busses frequently. The example is fairly complicated, so the code is explained here:

- **busAudioSynth** Function: Accepts two arguments, and creates an audio-rate **SinOsc**, routed to the left output channel. The frequency is determined by a control-rate bus given as an argument, and optionally with an offset that can be supplied as an argument.

- `busControlSynth` Function: Accepts two arguments, and creates a control-rate **SinOsc**, routed to the bus given as an argument. Can also be given a frequency; the value produced by the synth is intended to be used to control pitch. The centre pitch of the oscillation is **freq**, and the range of oscillation is one-twentieth the size of **freq** (one-fourtieth both higher and lower than **freq**).
- `SynthDef`: These commands are straight-forward. They send the synthesis definitions to the server.
- **b = Bus.control(s);** : This should also be straight-forward. A single-channel control bus is created, and assigned to the pre-declared variable **b**.
- For synth creation, **x** is assigned a control-rate synth, while **y** and **z** are assigned audio-rate synths. Each synth is given the variable **b**, which refers to our control-rate bus. **z** is also given an argument for `\freqOffset`, which makes its frequency 200 Hz higher than the synth assigned to **y**.
- Don't worry about the `after` message for now. It's explained in [Section 11.3.12.1, "Ordering"](#).

11.3.11.4.1. Why Use Global Variables

Since this is just an example, and not an actual program, the program uses four automatically-declared global variables: **b**, **x**, **y**, and **z**. Because these variables are shared with everything, it's especially important to set them to **nil** when you're done. If this were going to be written into a real program, it would be a good idea to change the variables to something which can't be accessed by other programs.

11.3.11.4.2. Why Use a Bus

The control-rate bus in this example might seem trivial and pointless to you, especially since the use of a UGen to control frequency has already been illustrated in other examples. For this particular program, a control-rate UGen would probably have been a better choice, but remember that this is just an example.

Here are some advantages to using a control-rate Bus over a UGen:

- The signal can be changed without sending the `set` message to the audio-rate UGen, simply by changing the input to the bus.
- Input to the bus can be produced by any number of control-rate UGen's.
- The signal in the bus can be received by more than one UGen, as it is in this example. One thousand audio-rate UGen's powered by 25 control-rate UGen's is a much better solution than if each audio-rate UGen were powered by its own control-rate UGen.
- Busses can be accessed quickly and efficiently from any place in the program that has access to the variable holding the Bus. It's easier and safer (less error-prone) than making all of your UGen's equally accessible.

Some of these advantages could be seen as disadvantages. Whether you should use a Bus or a UGen depends on the particular application. The simpler solution is usually the better one, as long as you remember to avoid repetition!

11.3.11.4.3. Special Note about Control-Rate Busses

Control-rate Bus'ses are a great way to enhance the flexibility of your program. The best part is that, in order to use a control-rate Bus, the UGen doesn't even need to have been written to accomodate it.

```
{ SinOsc.ar( freq:In.kr( controlRateBus, 1 ), mul:0.2 ); }.play;
```

Now you've managed to spice up an otherwise-boring synth!

Also, control-rate Busses don't need to be constantly changing. Unlike an audio-rate Bus, a control-rate Bus will hold the last-inputted value until another value is provided. You can set the value of a control-rate Bus with the `set` message (and a single argument, which is the value). You can also get the current value, whether created by `set` or a `UGen`, by using the `get` message, and sending a Function with one argument.

```
(
  var bus = Bus.control( s );
  bus.set( 12 );
  bus.get( { arg val; val.postln; } );
  bus.free; bus = nil;
)
```

When running this example, you'll notice that the **12** doesn't get posted until *after* the program finishes with **nil**. This is because of the *latency* between when the interpreter asks the server to do something, and when the server does it. The amount of time it takes for the server to complete a command is usually very small, but as you can see, it can make an important difference to your program. This latency is also the reason that you can't call **SynthDef.new(...)** and **Synth.new(...)** at the exact same time.

This latency is also the reason that we have to provide a single-argument function as an argument to the `get` function. Since the function won't immediately be able to get the value of the bus from the server, we can't expect the value to be returned by the function. Instead, when "get" gets the value of the bus from the server, it runs the function that you gave it.

11.3.11.5. Using Busses: Audio-Rate Example

```
(
  var tutorialDecayPink =
  {
    arg outBus = 0, effectBus,
    direct = 0.5; // controls proportion of "direct" / "processed" sound
    var source;

    // Decaying pulses of PinkNoise.
    source = Decay2.ar( in:Impulse.ar( freq:1, phase:0.25 ),
                       attackTime:0.01,
                       decayTime:0.2,
                       mul:PinkNoise.ar
                       );

    Out.ar( outBus, (source*direct) ); // main output
    Out.ar( effectBus, (source*(1-direct)) ); // effects output
  };

  var tutorialDecaySine =
  {
    arg outBus = 0, effectBus,
    direct = 0.5; // controls proportion of "direct" / "processed" sound
    var source;

    // Decaying pulses of a modulating Sine wave.
    source = Decay2.ar( in:Impulse.ar( freq:0.3, phase: 0.25),
                       attackTime:0.3,
                       decayTime:1,
                       mul:SinOsc.ar( freq:SinOsc.kr( freq:0.2, mul:110, add:440 ) )
                       );

    Out.ar(outBus, (source*direct) ); // main output
  }
)
```

```

    Out.ar(effectBus, (source*(1-direct)) ); // effects output
};

var tutorialReverb =
{
  arg outBus = 0, inBus; // default outBus is audio interface
  var input;

  input = In.ar( inBus, 1 );

  16.do( { input = AllpassC.ar( in:input,
                               maxdelaytime:0.04,
                               delaytime:{ Rand(0.001,0.04) }.dup,
                               decaytime:3
                             )
        }
    );

  Out.ar( outBus, input );
};

// send synthesis information to the server
SynthDef( \tutorialReverb, tutorialReverb ).send( s );
SynthDef( \tutorialDecayPink, tutorialDecayPink ).send( s );
SynthDef( \tutorialDecaySine, tutorialDecaySine ).send( s );

// reserve an effects Bus
b = Bus.audio( s );
)

(
  x = Synth.new( \tutorialReverb, [\inBus, b] );
  y = Synth.before( x, \tutorialDecayPink, [\effectBus, b] );
  z = Synth.before( x, \tutorialDecaySine, [\effectBus, b, \outBus, 1] );
)

// Change the balance of "wet" to "dry"
y.set( \direct, 1 ); // only direct PinkNoise
z.set( \direct, 1 ); // only direct Sine wave
y.set( \direct, 0 ); // only reverberated PinkNoise
z.set( \direct, 0 ); // only reverberated Sine wave
y.set( \direct, 0.5 ); // original PinkNoise
z.set( \direct, 0.5 ); // original Sine wave

( // commands to free each Object
  x.free; x = nil;
  y.free; y = nil;
  z.free; z = nil;
  b.free; b = nil;
)

```

I'm not going to explain this example as extensively as the previous one. It's definitely the most complex example so far. It's better if you figure out what the parts do by playing with them yourself. The bus works by routing audio from the **\tutorialDecayPink** and **\tutorialDecaySine** synths into the **\tutorialReverb** synth. The first two synths can be controlled to put all, none, or some of their signal into the bus (so that it goes through the **\tutorialReverb** synth), or straight out the audio interface (bypassing the **\tutorialReverb** synth). Notice that the *same* effects processor is operating on two different input sources.

11.3.12. Ordering and Other Synth Features

This section discusses the important topic of creating and enforcing an "order" on the server. Because this is done with Functions (or methods) from the Synth Class, other useful Functions from the Class are discussed here.

11.3.12.1. Ordering

Ordering is instructing the server to calculate in a particular order. The audio synthesized by the server takes the same form as any other digital audio: a series of samples are played at a particular speed (called sample rate), each with a set number of bits per sample (called sample format). For each sample, the server calculates the signal at that point in a pre-determined order. Each sample is calculated from scratch, so if a particular UGen depends on the output of another UGen, the other one had better be calculated first. For more information on samples, sample rate, and sample format, see [Section 1.3, "Sample, Sample Rate, Sample Format, and Bit Rate"](#).

Consider the following example:

```
{ SinOsc.ar( freq:SinOsc.kr( freq:1, add:500, mul:10 ), mul:0.2 ); }.play;
```

What happens if the server calculates the audio-rate UGen first? It wouldn't have a frequency. This is another one of those things which the interpreter takes care of automatically when we run Function rather than create a Synth. Since it's often preferable to use a synth instead of a Function, we need some way to control the order of execution. The interpreter and the server are only so good at guessing what we need, after all.

There are two methods in the **Synth** Class that we can use to inform the server about our desired order of execution: "before" and "after". They represent a small extension to the "new" method, and they work like this:

Synth.before(*variableHoldingSynth*, *nameOfSynthDef*, *ListOfArguments*);

and

Synth.after(*variableHoldingSynth*, *nameOfSynthDef*, *ListOfArguments*);

And it works just as it looks, too: the server creates a new synth, adds it before or after the synth represented by "variableHoldingSynth" (depending on which Function you use), and uses "nameOfSynthDef" and "ListOfArguments" just as in the "add" method. This example, from [Section 11.3.11.4, "Using Busses: Control-Rate Example"](#), uses the "after" Function to ensure that the control-rate synth is calculated before the audio-rate synths that depend on it.

```
( // execute first: prepare the server
  var busAudioSynth =
  {
    arg bus, freqOffset = 0;

    Out.ar( 0, SinOsc.ar( freq:( In.kr(bus) + freqOffset ), mul:0.1 ) );
  };

  var busControlSynth =
  {
    arg bus, freq = 400;

    Out.kr( bus, SinOsc.kr( freq:1, mul:( freq/40 ), add:freq ) );
  };

  SynthDef( \tutorialAudioBus, busAudioSynth ).send( s );
  SynthDef( \tutorialControlBus, busControlSynth ).send( s );

  b = Bus.control( s );
)
```

```
( // execute second: create synths
  x = Synth.new( \tutorialControlBus, [\bus, b] ); // control synth
  y = Synth.after( x, \tutorialAudioBus, [\bus, b] ); // low audio synth
  z = Synth.after( x, \tutorialAudioBus, [\bus, b, \freqOffset, 200] ); // high audio synth
)

( // commands to free each Object
  x.free; x = nil; // control synth
  y.free; y = nil; // low audio synth
  z.free; z = nil; // high audio synth
  b.free; b = nil; // control bus
)
```

In this case, the control-rate synth is created before the audio-rate synths - probably the easier way to think about it. Even so, it's possible to add them in the opposite order with a little extra thought.

The other example from [Section 11.3.11, "Busses"](#) use the "before" Function to ensure that the "pink noise" and "sine wave" UGen's were calculated before the "reverberation" UGen. Especially since these are all audio-rate UGen's, the server would not reasonably know which to calculate first, so you need to let it know.

11.3.12.2. Changing the Order

SuperCollider offers equally easy-to-use methods to change the order of execution.

To move a synth's execution after another:

```
variableHoldingSynthmoveAfter(variableHoldingAnotherSynth);
```

To move a synth's execution before another:

```
variableHoldingSynthmoveBefore(variableHoldingAnotherSynth);
```

11.3.12.3. Replace a Running Synth

The server allows you to replace a running synth with a newly-created one, maintaining all of the ordering relationships.

This is the syntax:

```
variableHoldingNewSynth = Synth.replace( variableHoldingSynthToReplace, nameOfSynthDef,
  ListOfArguments );
```

The "variableHoldingNewSynth" will often be the same as the "variableHoldingSynthToReplace," but not always. When you use this Function, the synth being replaced is freed from the server (equivalent to running "free"), so that variable should always be assigned something.

11.3.12.4. Pausing and Restarting a Synth

The server allows you to temporarily pause and later re-start a synth, without freeing and re-creating it.

To pause a synth:

```
variableHoldingSynth.run( false );
```

To re-start a synth:

```
variableHoldingSynth.run( true );
```

11.3.13. Scheduling

The practice of scheduling allows you to making things happen in a pre-determined amount of time. Scheduling is very different from ordering: ordering is a primarily technical consideration to ensure that the server synthesizes the sound in the right order; scheduling is a primarily musical consideration that allows you to control the perceived time that things happen.

11.3.13.1. Clocks

SuperCollider's clocks have two main functions: they know what time it is, and they know what time things are supposed to happen.

There are three types of clocks, which each do slightly different things:

- **TempoClock**: These clocks are aware of metre (time signature) changes, and have an adjustable tempo. They are to be used for scheduling musical events, and they run with a high priority.
- **SystemClock**: This clock always runs in seconds. It can be used to schedule musical events, since it runs with a high priority. There is only one **SystemClock**.
- **AppClock**: These clocks always run in seconds. They are to be used for graphic events and other non-musical things not discussed in this guide. These clocks do not run with a high priority, so they can be temporarily "side-lined" by a **TempoClock** or the **SystemClock**, if one of those has something to do urgently.

11.3.13.2. Default Clocks

The **SuperCollider** interpreter provides two default clocks, and one default pseudo-clock.

The **SystemClock** always operates in seconds, and it can be used to schedule musical events, but usually this isn't necessary.

The **TempoClock.default** runs at 60 beats-per-minute by default (equal to one beat per second). Since it's accessible from anywhere within a program, any tempo changes will have an effect on the scheduling of the entire program - so be careful! If you don't want something to be effected by tempo changes, you can create a new **TempoClock** just for that part of the program. If you will be using this clock frequently, you can assign it to a variable like this:

```
var t = TempoClock.default;
```

The **thisThread.clock** is not really a clock in itself, but refers to the clock which is responsible for scheduling the part of the program where the command is written. It can be a little bit tricky working with this clock, since it may be either the **SystemClock** or a **TempoClock**.

11.3.13.3. Finding the Current Time

Using the "beats" method on a clock will return that clock's current time. Try running each of the following:

```
SystemClock.beats;  
TempoClock.default.beats;  
thisThread.clock.beats;
```

This can be useful for scheduling events in an absolute way, or for a number of other things.

11.3.13.4. Relative Scheduling

The standard way to schedule things is in a certain number of beats from now. If you're scheduling on a `SystemClock`, one beat is equal to one second. If you're scheduling on a `TempoClock`, one beat is equal to whatever the current setting is.

To schedule things on a clock, use the "sched" Function:

```
nameOfClocksched(beatsFromNow, FunctionToExecute);
```

The interpreter will let you schedule just about anything, but there's no point in scheduling something other than a Function: scheduling a five won't have any effect - try it!

```
SystemClock.sched( 5, 5 );
```

It looks like nothing happens. The **5** does happen, but... well... it doesn't do anything. Scheduling a Function *will* do something:

```
SystemClock.sched( 5, { 5.postln; } );
```

When you run this, there are two things to notice:

- The interpreter prints out "SystemClock" first. This is to let you know that it did the scheduling as requested.
- The five prints out endlessly, in five-second intervals. For an explanation, see "Repeated Scheduling"

11.3.13.5. Repeated Scheduling

If you schedule a Function that returns a number, the interpreter will schedule the Function to re-run in that many beats.

This will print "5" every five seconds, until you press [Esc] to stop execution.

```
SystemClock.sched( 5, { 5.postln; } );
```

To avoid this, you can end your Function with **nil**, which has been done sometimes through this guide.

```
SystemClock.sched( 5, { 5.postln; nil; } );
```

This will print **5** in five seconds, and then stop.

11.3.13.6. Working with the TempoClock Class

Here is a brief explanation of some Functions available with the `TempoClock` Class. Throughout this section, the variable "t" is used to represent any particular `TempoClock`.

```
var t = TempoClock.new( tempo, beats );
```

This creates a new `TempoClock`. The arguments are optional, and have the following meanings:

- tempo: tempo of the clock, given in beats per second. To input a value in beats-per-minute, divide it by 60. Defaults to 60 beats per minute, or one per second.
- beats: starts the clock at this time. Default is zero.

```
t.stop;  
t = nil;
```

Equivalent to the "free" method for a Synth of Bus Object. This stops the clock, discards all scheduled events, and releases the resources used to run the clock. Setting the variable to "nil" afterwards is optional, but recommended, to avoid later programming mistakes.

```
t.clear;
```

Discards all scheduled events, but keeps the clock running.

```
t.tempo;
```

Returns the current tempo in beats-per-second.

```
t.tempo_( newTempo );
```

Allows you to change the clock's tempo. The new tempo should be in beats-per-second. To input a tempo in beats-per-minute, divide the value by 60.

```
t.play( aFunction );
```

Schedules the Function to begin execution on the next beat.

There are many other Functions in the TempoClock Class, related to absolute scheduling, scheduling with bars, and conversion of beats to and from seconds.

11.3.14. How to Get Help

Knowing how to get help in **SuperCollider** is going to play a large part in determining whether you have a productive or frustrating relationship with the language and its components. There are a large number of ways to get help, but here are some of the most helpful.

11.3.14.1. Use the SuperCollider Help Files

SuperCollider comes with an extensive collection of help files, which contain the answers to most of your problems. The difficulty will be in finding the solution - it's not always located where you think it is, because it often isn't the solution you think it will be.

On Fedora Linux systems, the main help file is located at [file:///usr/share/SuperCollider/Help/Help.html this URL], and it can be viewed in any web browser. It may also be helpful to browse the directory structure of the help files, located at [file:///usr/share/SuperCollider/Help this URL], which can also be viewed in your web browser.

If you're looking for further explanations of material in this tutorial, you could start by reviewing the [file:///usr/share/SuperCollider/Help/Tutorials/Getting-Started/Getting%20Started%20With%20SC.html Getting Started With **SuperCollider**] tutorial, on which this document is based. The sections in that tutorial roughly correspond to the sections in this guide.

11.3.14.2. Internet Relay Chat (IRC)

If you know how to use Internet Relay Chat (IRC), you can join the #supercollider channel on the Freenode network. The channel does not usually have a large number of participants or a lot of activity, but the users are some of the most polite and helpful on the internet.

11.3.14.3. Email

If you feel comfortable sending an email to a mailing list, you can use the *sc-users Mailing List*, available at http://www.beast.bham.ac.uk/research/sc_mailing_lists.shtml. If you decide to subscribe to this list, be aware that it receives a large amount of mail every day.

11.3.14.4. The SuperCollider Website

The **SuperCollider Website** at SourceForge (<http://supercollider.sourceforge.net/>) offers links to many resources.

11.3.15. Legal Attribution

This portion of the Fedora Musicians' Guide, called "Basic Programming with **SuperCollider**," is a derivative work of the, *Getting Started With SuperCollider* tutorial. The original work was created by Scott Wilson, James Harkins, and the **SuperCollider** development team. It is available on the internet from <http://supercollider.svn.sourceforge.net/viewvc/supercollider/trunk/common/build/Help/Tutorials/Getting-Started/Getting%20Started%20With%20SC.html>.

The original document, like all **SuperCollider** documentation, is licenced under the Creative Commons' *Attribution Share-Alike 3.0 Unported Licence*, accessible on the internet at <http://creativecommons.org/licenses/by-sa/3.0/>.

This usage should in no way be construed as an endorsement of the Fedora Project, the Musicians' Guide, or any other party by the **SuperCollider** development team.

11.4. Composing with SuperCollider

This section is an explanation of the creative thought-process that went into creating the SuperCollider composition that we've called "Method One," for which the source and exported audio files are available below.

It is our hope that, in illustrating how we developed this composition from a single SinOsc command, you will learn about SuperCollider and its abilities, about how to be creative with SuperCollider, and how a simple idea can turn into something of greater and greater complexity.

As musicians, our goal is to learn enough SuperCollider to make music; we don't want to have to memorize which parameters do what for which functions, and in which order to call them. We want to know what they do for us musically. Explicitly calling parameters, and making comments about what does what, so that we can return later and change musical things, are going to help our musical productivity, at the expense of slowing down our typing.

11.4.1. Files for the Tutorial

The following files represent complete versions of the program. You should try to complete the program yourself before reviewing these versions:

- *Method One* is available at http://docs.fedoraproject.org/en-US/Fedora/15/html/Musicians_Guide/files/SuperCollider/Method_One.sc
- *Method One (Optimized)* is available at http://docs.fedoraproject.org/en-US/Fedora/15/html/Musicians_Guide/files/SuperCollider/Method_One-optimized.sc
- A FLAC-format recording of *Method One* is available at http://docs.fedoraproject.org/en-US/Fedora/15/html/Musicians_Guide/files/SuperCollider/Method_One.flac

Method_One.sc is an extensively-commented version of the source code. The comments not only describe the way the code works, but pose some problems and questions that you may wish to work

on, to increase your knowledge of SuperCollider. The problem with the verbosity of the comments is that it can be difficult to read the code itself, as it would be written in a real program.

Method_One-optimized.sc is a less-commented version of the source code. I've also re-written part of the code, to make it more flexible for use in other programs. The differences between this, and code that I would have written for myself only, are trivial.

Method_One.flac is a recording that I produced of the program, which I produced in **Ardour**.

11.4.2. Inspiration

The intention of this program is to represent one way to write a SuperCollider program. I decided to take one class, SinOsc, and use it for "everything." Here, "everything" means any function that returns a sound, or any function that directly controls a SinOsc.

In order to fill up time, I decided to employ a three-part "rounded binary" form: ABA' or "something, something new, then the first thing again." This is kind of like a sine oscillation, too!

11.4.3. Designing the First Part

1. I started with something simple: a single SinOsc: `{ SinOsc.ar(); }.play;`
2. This is not exciting: it just stays the same forever, and it only uses one channel! So, I added another SinOsc to the right channel, using the `[,]` array notation. The result is `{ [SinOsc.ar(), SinOsc.ar()] }.play;`
3. Now it sounds balanced, at least, like it's coming from the middle. But it's still boring, so I added a frequency-changing SinOsc to the right channel, resulting in `{ [SinOsc.ar(), SinOsc.ar(SinOsc.kr(1, 50, 300))] }.play;`
4. Since that's difficult to read, and since I know that I'm just going to keep adding things, I expand the code a little bit to make it more legible. This gives me

```
{
  var left = SinOsc.ar();
  var right = SinOsc.ar( SinOsc.kr( 1, 50, 300 ) );

  [ left, right ]

}.play;
```

I define a variable holding everything I want in the left channel, then the same for the right. I still use the `[,]` array notation to create a stereo array. Remember that SuperCollider functions return the last value stated, so it might look like the stereo array is ignored, but because this array is what is returned by the function contained between `{` and `}`, it is this array that gets played by the following `".play;"`

5. I also added a frequency controller to the left SinOsc, and realized that it's getting a bit difficult to read again, especially if I wanted to add another parameter to the SinOsc.ar objects. So I placed the SinOsc.kr's into their own variables: frequencyL and frequencyR. This results in

```
{
  var frequencyL = SinOsc.kr( freq:10, mul:200, add:400 );
  var frequencyR = SinOsc.kr( freq:1, mul:50, add:150 );

  var left = SinOsc.ar( frequencyL );
```

```

    var right = SinOsc.ar( frequencyR );

    [ left, right ]

  }.play;

```

6. Now I can experiment with the frequency-changing SinOsc's, to make sure that I get things just right. When I realize what the parameters do, I make a note for myself (see "FSC-method-1-.sc"), so that it will be easy to adjust it later. I also explicitly call the parameters. This isn't necessary, but it also helps to avoid future confusion. Most programmers would not explicitly call the parameters, but we're musicians, not programmers.
7. The left channel has something like a "melody," so I decided to add a drone-like SinOsc to it. This is easy, of course, because any SinOsc left alone is automatically a drone! But, where should it be added? Into the "left" variable, of course. We'll create an array using [,] array notation. There are two things that I would do at this point to help with future readability:
 - a. Align all of the left-channel SinOsc's vertically (using tabs and spaces), so that each line is one sound-generating UGen.
 - b. At the end of each line, write a small comment describing what the UGen on that line doesn't.
8. Now the volume is a problem. For most sound-producing UGen's, the "mul" argument controls the volume. For most of those, the default is "1.0," and anything greater will create distorted output. The physics and computer science factors that wind up creating distortion are rather complicated, and it isn't necessary to understand them. What we need to know is that, if the output of a UGen (or some UGen's) sounds distorted, then we should probably adjust the "mul" argument. Sometimes, of course, you may prefer that distorted output.
 - It seems that, when you're using multiple SinOsc's in one output channel, the "mul" of all of them must not add to more than 1.0
 - We're using two output channels (left and right). We'll leave the right channel alone for now, because it has only one output UGen.
 - So, I'll change add a "mul" argument to each of the left-channel UGen's, to 0.5
9. Now we can't hear the left channel, because the right channel is too loud! Playing with volumes (sometimes called "adjusting levels" for computers) is a constant aesthetic concern for all musicians. Add a "mul" argument to the right channel, and set it to what seems an appropriate volume for the moment. It will probably change later, but that's okay.
10. But let's add another dimension to this: there's no reason to keep the volume static, because we can use a SinOsc to change it periodically! I added a SinOsc variable called "volumeL," which I used as the argument to "mul" for the "frequencyL" SinOsc in the left channel.
11. And now the sheer boredom of the drone in the left channel becomes obvious. I decide to make it more interesting by adding a series of overtones (an overtone is...). I decide to add six, then experiment with which frequencies to add. But, every time I adjust one frequency, I have to recalculate and change all the others. So I decide to add a variable for the drone's frequency: "frequencyL_drone". This way, after finding the right intervals, I can easily adjust all of them just by changing the variable. I've decided on drone*1, 2, 5, 13, and 28. These are more or less arbitrary, and I arrived on them through experimentation. Of course, the drone will be way too loud.
12. Writing

```
SinOsc.ar( [frequencyL_drone, 2*frequencyL_drone, 5*frequencyL_drone, 13*frequencyL_drone, 28*frequencyL_drone],
mul:0.1 )
```

in your program is not easy to read, and actually it doesn't work out volume-balance-wise (for me, at least): the high frequencies are too loud, and the lower ones are not loud enough. In retrospect, I should have created a variable for the "mul" of these drones, so I could adjust them easily in proportion. But, I didn't.

13. A constant drone isn't as much fun as one that slowly changes over time. So, I changed the "frequencyL_drone" value to a SinOsc.kr UGen. Because it's supposed to be a "drone," it should change only very gradually, so I used a very small frequency argument. It still moves quite quickly, but people won't want to listen to this too long, anyway!
14. I did something similar with the right channel, adding a slowly-changing drone and overtones above it.
15. After some final volume adjustments, I feel that I have completed the first part. There is no way to know for sure that you've finished until it happens. Even then, you may want to change your program later.

11.4.4. Designing the Second Part

The next thing that I did was to design the second part. This will not join them together yet, and I'm going to focus on something completely different, so I decided to do this in a separate file.

My inspiration for this part came from experimenting with the drones of the first part. There are a virtually unlimited number of combinations of sets of overtones that could be created, and the combinations of discrete frequencies into complex sounds is something that has fascinated me for a long time. Moreover, when thousands of discrete frequencies combine in such a way as to create what we think of as "a violin playing one note," it seems like a magical moment.

I'm going to build up a set of pseudo-random tones, adding them one at a time, in set increments. As you will see, this introduces a number of problems, primarily because of the scheduling involved with the one-by-one introduction of tones, and keeping track of those tones.

The fact that there are ten tones also poses a problem, because it might require a lot of typing. We'll see solutions to that, which use SuperCollider's programming features to greatly increase the efficiency.

Although we've already solved the musical problems (that is, we know what we want this part to sound like), the computer science (programming) problems will have to be solved the old-fashioned way: start with something simple, and build it into a complex solution.

First I will develop the version used in FSC-method-1.sc, then the version used in FSC-method-1-short.sc

11.4.5. Creating Ten Pseudo-Random Tones

1. We'll start again with something simple, that we know how to do.

```
{
  SinOsc.ar();
}.play;
```



```
[ SinOsc.ar( freq:frequency8, mul:0.01 ), SinOsc.ar( freq:frequency8, mul:0.01 ) ]
var frequency9 = 200 + 600.rand;
[ SinOsc.ar( freq:frequency9, mul:0.01 ), SinOsc.ar( freq:frequency9, mul:0.01 ) ]
var frequency0 = 200 + 600.rand;
[ SinOsc.ar( freq:frequency0, mul:0.01 ), SinOsc.ar( freq:frequency0, mul:0.01 ) ]
}.play;
```

7. It still doesn't work! The error given in the "SuperCollider output" window is not easy to understand, but it means "You have to put all of your variable declarations before everything else."

```
{
  var frequency1 = 200 + 600.rand;
  var frequency2 = 200 + 600.rand;
  var frequency3 = 200 + 600.rand;
  var frequency4 = 200 + 600.rand;
  var frequency5 = 200 + 600.rand;
  var frequency6 = 200 + 600.rand;
  var frequency7 = 200 + 600.rand;
  var frequency8 = 200 + 600.rand;
  var frequency9 = 200 + 600.rand;
  var frequency0 = 200 + 600.rand;

  [ SinOsc.ar( freq:frequency1, mul:0.01 ), SinOsc.ar( freq:frequency1, mul:0.01 ) ]
  [ SinOsc.ar( freq:frequency2, mul:0.01 ), SinOsc.ar( freq:frequency2, mul:0.01 ) ]
  [ SinOsc.ar( freq:frequency3, mul:0.01 ), SinOsc.ar( freq:frequency3, mul:0.01 ) ]
  [ SinOsc.ar( freq:frequency4, mul:0.01 ), SinOsc.ar( freq:frequency4, mul:0.01 ) ]
  [ SinOsc.ar( freq:frequency5, mul:0.01 ), SinOsc.ar( freq:frequency5, mul:0.01 ) ]
  [ SinOsc.ar( freq:frequency6, mul:0.01 ), SinOsc.ar( freq:frequency6, mul:0.01 ) ]
  [ SinOsc.ar( freq:frequency7, mul:0.01 ), SinOsc.ar( freq:frequency7, mul:0.01 ) ]
  [ SinOsc.ar( freq:frequency8, mul:0.01 ), SinOsc.ar( freq:frequency8, mul:0.01 ) ]
  [ SinOsc.ar( freq:frequency9, mul:0.01 ), SinOsc.ar( freq:frequency9, mul:0.01 ) ]
  [ SinOsc.ar( freq:frequency0, mul:0.01 ), SinOsc.ar( freq:frequency0, mul:0.01 ) ]
}.play;
```

8. It still doesn't work! SuperCollider is confused because I was been lazy and didn't include enough semicolons. The error we get is, "Index not an Integer," which is a clue as to what SuperCollider is trying to do (but it's irrelevant). The real problem is that SuperCollider interprets our ten stereo arrays as all being part of the same statement. We don't want them to be the same statement, however, because we want ten *different* stereo arrays to be played. Fix this problem by putting a semicolon at the end of each stereo array. You do not *need* to include one at the end of the last statement, because SuperCollider assumes the end of the statement when it encounters a } (end-of-function marker) after it. Since we're still building our code, we might move these around or add something afterwards, so it's better to include a semicolon at the end of each stereo array.
9. Now the file plays successfully, but with a disappointing result. If you can't already see the problem, try to think of it before continuing to read.
10. Only one SinOsc array gets played, and it's the last one. This is because the last statement is returned by the function that ends at } and it is that result which gets sent to the following .play
11. To fix this, and ensure that all of the stereo arrays are played, you should remove the .play from the end of the function, and add a .play to each stereo array statement. You end up with

```
{
  var frequency1 = 200 + 600.rand;
  var frequency2 = 200 + 600.rand;
  var frequency3 = 200 + 600.rand;
  var frequency4 = 200 + 600.rand;
```

```

var frequency5 = 200 + 600.rand;
var frequency6 = 200 + 600.rand;
var frequency7 = 200 + 600.rand;
var frequency8 = 200 + 600.rand;
var frequency9 = 200 + 600.rand;
var frequency0 = 200 + 600.rand;

[ SinOsc.ar( freq:frequency1, mul:0.01 ), SinOsc.ar( freq:frequency1,
mul:0.01 ) ].play;
[ SinOsc.ar( freq:frequency2, mul:0.01 ), SinOsc.ar( freq:frequency2,
mul:0.01 ) ].play;
[ SinOsc.ar( freq:frequency3, mul:0.01 ), SinOsc.ar( freq:frequency3,
mul:0.01 ) ].play;
[ SinOsc.ar( freq:frequency4, mul:0.01 ), SinOsc.ar( freq:frequency4,
mul:0.01 ) ].play;
[ SinOsc.ar( freq:frequency5, mul:0.01 ), SinOsc.ar( freq:frequency5,
mul:0.01 ) ].play;
[ SinOsc.ar( freq:frequency6, mul:0.01 ), SinOsc.ar( freq:frequency6,
mul:0.01 ) ].play;
[ SinOsc.ar( freq:frequency7, mul:0.01 ), SinOsc.ar( freq:frequency7,
mul:0.01 ) ].play;
[ SinOsc.ar( freq:frequency8, mul:0.01 ), SinOsc.ar( freq:frequency8,
mul:0.01 ) ].play;
[ SinOsc.ar( freq:frequency9, mul:0.01 ), SinOsc.ar( freq:frequency9,
mul:0.01 ) ].play;
[ SinOsc.ar( freq:frequency0, mul:0.01 ), SinOsc.ar( freq:frequency0,
mul:0.01 ) ].play;
}

```

12. When you execute this, no sound is produced, but SuperCollider outputs "a Function." Can you think of why this happens? It's because you wrote a function, but never told SuperCollider to evaluate it! At the end of execution, SuperCollider just throws away the function, because it's never used. This is the same thing that happened to the first nine stereo arrays - they were created, but you never said to do anything with them, so they were just thrown out. We need to execute the function. Because it doesn't produce a UGen, we can't use "play," so we have to use "value" instead. You can choose to do either of these:

```
{ ... }.value;
```

or

```
var myFunction = { ... };
myFunction.value;
```

13. This gives us yet another error, as if we can't play the stereo arrays! In fact, we can't - and we didn't do it in the first part, either. We play'ed the result of returning a stereo array from a function. The subtle difference isn't important yet - we're just trying to make this work! Use { and } to build a function for .play to .play
14. Now make the correction nine more times.
15. When you play execute the resulting code, you probably get something that sounds quite "space-age." Execute it a few times, to see the kind of results you get.

11.4.6. Scheduling the Tones

1. The next step is to get these started consecutively, with 5-second pauses after each addition. For this we will use a TempoClock, and since this is the only thing that we're doing, we'll just use the

default one called `TempoClock.default`. I don't feel like typing that, however, so we're going to define an alias variable: **`var t_c = TempoClock.default`**; You could put that in the main function, but I suggest putting it before the main function. This way, if we want to write another function later, then it can also access `t_c`.

2. The default `TempoClock` has a default tempo of one beat per second (1 Hz). This will be good enough for us. If you wanted to change the tempo, remember that you can enter a metronome setting (which is "beats per minute") by dividing the metronome setting by 60. So a metronome's 120 beats per minute would be given to a new `TempoClock` as **`TempoClock.new(120/60)`**. Even though you could do that ahead of time and just write "2," inputting it as "120/60" makes it clearer what tempo you intend to set.
3. You can schedule something on a `TempoClock` by using **`t_c.sched(x, f)`**; where **`f`** is a function to execute, and **`x`** is when it should be done, measured as the number of beats from now. So we can schedule our `SinOsc` like this:

```
t_c.sched( 1, {[ SinOsc.ar( freq:frequency1, mul:0.01 ), SinOsc.ar( freq:frequency1,
mul:0.01 ) ]}.play; } );
```

4. Schedule the rest, in intervals of five beats (which is five seconds). They will all be scheduled virtually instantaneously (that is, the computer will notice the slight delay between when each one is scheduled, but humans will not). I started at one beat from now, to insert a slight pause before the sound begins.
5. If you've done this correctly, then we should get a build-up of ten pitches. But they never stop! This is going to take some more ingenuity to solve, because we can't just make a stereo array, play it, then throw it away. We need to hold onto the stereo array, so that we can stop it. The first step here is to store the stereo arrays in variables, and subsequently schedule them. You will end up with something like this:

```
var sinosc1 = { [ SinOsc.ar( freq:frequency1, mul:0.01 ), SinOsc.ar( freq:frequency1,
mul:0.01 ) ] };
// the other nine...

t_c.sched( 1, { sinosc1.play; } );
// the other nine...
```

6. It should still work, but we after all that cutting-and-pasting, we still haven't managed to turn off the `SinOsc`'s. We need to "free" the object that was returned when we used the "play" function. We need to declare yet more variables: **`var so1, so2, so3, so4, so5, so6, so7, so8, so9, so0`**; should appear anywhere before the scheduler.
7. Now adjust all the scheduling commands so they look like this: **`t_c.sched(1, { so1 = sinosc1.play; });`**
8. Now you can add ten of these, after the existing scheduling commands: **`t_c.sched(51, { so1.free; });`**. Be sure to schedule each one for 51 beats, so that they all turn off simultaneously, 5 beats after the last pitch is added.
9. It should work successfully. If it doesn't, then compare what you have to this, which does work:

```
var t_c = TempoClock.default;

{
  var frequency1 = 200 + 600.rand;
```

```

var frequency2 = 200 + 600.rand;
var frequency3 = 200 + 600.rand;
var frequency4 = 200 + 600.rand;
var frequency5 = 200 + 600.rand;
var frequency6 = 200 + 600.rand;
var frequency7 = 200 + 600.rand;
var frequency8 = 200 + 600.rand;
var frequency9 = 200 + 600.rand;
var frequency0 = 200 + 600.rand;

var sinosc1 = { [ SinOsc.ar( freq:frequency1, mul:0.01 ), SinOsc.ar( freq:frequency1,
mul:0.01 ) ] };
var sinosc2 = { [ SinOsc.ar( freq:frequency2, mul:0.01 ), SinOsc.ar( freq:frequency2,
mul:0.01 ) ] };
var sinosc3 = { [ SinOsc.ar( freq:frequency3, mul:0.01 ), SinOsc.ar( freq:frequency3,
mul:0.01 ) ] };
var sinosc4 = { [ SinOsc.ar( freq:frequency4, mul:0.01 ), SinOsc.ar( freq:frequency4,
mul:0.01 ) ] };
var sinosc5 = { [ SinOsc.ar( freq:frequency5, mul:0.01 ), SinOsc.ar( freq:frequency5,
mul:0.01 ) ] };
var sinosc6 = { [ SinOsc.ar( freq:frequency6, mul:0.01 ), SinOsc.ar( freq:frequency6,
mul:0.01 ) ] };
var sinosc7 = { [ SinOsc.ar( freq:frequency7, mul:0.01 ), SinOsc.ar( freq:frequency7,
mul:0.01 ) ] };
var sinosc8 = { [ SinOsc.ar( freq:frequency8, mul:0.01 ), SinOsc.ar( freq:frequency8,
mul:0.01 ) ] };
var sinosc9 = { [ SinOsc.ar( freq:frequency9, mul:0.01 ), SinOsc.ar( freq:frequency9,
mul:0.01 ) ] };
var sinosc0 = { [ SinOsc.ar( freq:frequency0, mul:0.01 ), SinOsc.ar( freq:frequency0,
mul:0.01 ) ] };

var so1, so2, so3, so4, so5, so6, so7, so8, so9, so0;

t_c.sched( 1, { so1 = sinosc1.play; } );
t_c.sched( 6, { so2 = sinosc2.play; } );
t_c.sched( 11, { so3 = sinosc3.play; } );
t_c.sched( 16, { so4 = sinosc4.play; } );
t_c.sched( 21, { so5 = sinosc5.play; } );
t_c.sched( 26, { so6 = sinosc6.play; } );
t_c.sched( 31, { so7 = sinosc7.play; } );
t_c.sched( 36, { so8 = sinosc8.play; } );
t_c.sched( 41, { so9 = sinosc9.play; } );
t_c.sched( 46, { so0 = sinosc0.play; } );

t_c.sched( 51, { so1.free; } );
t_c.sched( 51, { so2.free; } );
t_c.sched( 51, { so3.free; } );
t_c.sched( 51, { so4.free; } );
t_c.sched( 51, { so5.free; } );
t_c.sched( 51, { so6.free; } );
t_c.sched( 51, { so7.free; } );
t_c.sched( 51, { so8.free; } );
t_c.sched( 51, { so9.free; } );
t_c.sched( 51, { so0.free; } );

}.value;

```

11.4.7. Optimizing the Code

Hopefully, while working through the previous sections, you got an idea of how tedious, boring, difficult-to-read, and error-prone this sort of copy-and-paste programming can be. It's ridiculous, and it's poor programming:

- We're using a lot of variables and variable names. They're all just used once or twice, too.
- When you copy-and-paste code, but change it a little, you might make a mistake in that little change.

- When you copy-and-paste code, when you make a mistake, you have to copy-and-paste to fix it everywhere.
- Repetition is the enemy of high-quality code. It is much better to write something once and re-use that same code.

Thankfully, SuperCollider provides three things that will greatly help to solve these problems - at least for our current situation:

- Arrays can be used to hold multiple instances of the same thing, all referred to with essentially the same name. We're already doing something similar, (sinosc1, sinosc2, etc.) but arrays are more flexible.
- Functions can be written once, and executed as many times as desired.
- Loops also provide a means to write code once, and execute it many times. As you will see, they are useful in situations different from functions.

It should be noted that, while it is good practise to program like this, it is also optional. You will probably find, though, that writing your programs well in the first place ends up saving huge headaches in the future.

1. The first thing we'll do is write a function to deal with generating the stereo arrays of SinOsc's.
2. Take the code required to generate one stereo array of SinOsc's with a pseudo-random frequency. Put it in a function, and declare a variable for it (I used the name "func").
3. Now remove the frequency1 (etc.) variables, and change the sinosc1 (etc.) variables to use the new function. Make sure that the code still works in the same way. It's much easier to troubleshoot problems when you make only one change at a time!
4. At this point, we've eliminated ten lines of code, and made ten more lines easier to read by eliminating the subtle copy-and-paste changes. If you can't manage to work it out, refer to the FSC_method_1.sc file for tips.
5. We can eliminate ten more lines of code by using a loop with an array. Let's change only one thing at a time, to make it easier to find a problem, if it should arise. Start by commenting out the lines which declare and initialize sinosc1, sinosc2, and so on.
6. Then declare a ten-element array in the same place: **var sinosc = Array.new(10);**
7. The next part is to write code to get ten func.value's into the array. To add something to an array in SuperCollider, we use the "add" method: **sinosc.add(thing_to_add);** There is a small wrinkle to this, described in the SuperCollider documentation. It's not important to understand (for musical reasons, that is - it is explained on this help page), but when you add an element to an array, you *should* re-assign the array to the variable-name: **sinosc = sinosc.add(thing_to_add)** Basically it works out like this: if you don't re-assign, then there is a chance that the array name only includes the elements that were in the array before the "add" command was run.
8. With this, we are able to eliminate a further level of redundancy in the code. Ten exact copies of **sinosc = sinosc.add({ func.value; });** Now, ten lines that look almost identical actually are identical. Furthermore, we don't have to worry about assigning unique names, or even about index numbers, as in other programming languages. SuperCollider does this for us!
9. This still won't work, because we need to adjust the rest of the function to work with this array. The scheduling commands be changed to look something like this: **t_c.sched(1, { so1 =**

`sinosc[0].play; });` Since arrays are indexed from 0 to 9, those are the index numbers of the first ten objects in the array.

10. Remember that you need to put all of your variable declarations *before* anything else.
11. It *should* still work. Let's use a loop to get rid of the ten identical lines.
12. In SuperCollider, `x.do(f);` will send the **value** message to the function **f** *x* times. So, to do this ten times, we should write `10.do({ sinosc = sinosc.add({ func.value; }); });` and get rid of the other ones. This is very powerful for simple things that must be done multiple times, because you are definitely not going to make a copy-and-paste error, because it's easy to see what is being executed, and because it's easy to see how many times it is being executed.
13. Now let's reduce the repetitiveness of the scheduling. First, replace `so1`, `so2`, etc. with a ten-element array. Test it to ensure that the code still works.
14. Getting the next two loops working is a little bit more complicated. We know how to run the exact same code in a loop, but we don't know how to change it subtly (by supplying different index numbers for the array, for example). Thankfully, SuperCollider provides a way to keep track of how many times the function in a loop has already been run. The first argument given to a function in a loop is the number of times that the function has *already* been executed. The first time it is run, the function receives a 0; if we're using a `10.do(something);` loop, then the last time the function is run, it receives a 9 because the function has already been executed 9 times. Since our ten-element array is indexed from 0 to 9, this works perfectly for us.
15. The code to free is shorter: `10.do({ arg index; t_c.sched(51, { so[index].free; }); });` This can look confusing, especially written in one line, like it is. If it helps, you might want to write it like this instead:

```
10.do
({ arg index;
  t_c.sched( 51, { so[index].free; } );
});
```

Now it looks more like a typical function.

16. The next step is to simplify the original scheduling calls in a similar way, but it's slightly more complicated because we have to schedule a different number of measures for each call. With a little math, this is also not a problem - it's just a simple linear equation: **number_of_measures = 5 * array_index + 1** Try to write this loop by yourself, before going to the next step.
17. If you missed it, my solution is

```
10.do( { arg index; t_c.sched( ((5*index)+1), { so =
  so.add( sinosc[index].play; ); } ); } );
```

which includes some extra parentheses to ensure that the math is computed in the right order.

18. The code is already much shorter, easier to understand, and easier to expand or change. There is one further optimization that we can easily make: get rid of the `sinosc` array. This simply involves replacing `sinosc[index]` with what all of its elements are: `{ func.value; }`
19. The resulting program is a little different from what ended up in `FSC_method_1.sc`, but produces the same output. What I have is this:

```

var t_c = TempoClock.default;

{
  var so = Array.new( 10 );

  var func =
  {
    var frequency = 200 + 600.rand;
    [ SinOsc.ar( freq:frequency, mul:0.01 ), SinOsc.ar( freq:frequency, mul:0.01 ) ];
  };

  10.do( { arg index; t_c.sched( ((5*index)+1), { so =
so.add( {func.value;}.play; ); } ); } );
  10.do( { arg index; t_c.sched( 51, { so[index].free; } ); } );

}.value;

```

20. Finally, assign this Function to a variable (called "secondPart", perhaps), and remove the "value" Function-call. If we leave that in, the Function will execute before the rest of the program begins!

11.4.8. Making a Useful Section out of the Second Part

This section describes the reasons for the differences between the second part's Function that was just created, and the Function that appears in "FSC_method_1-short.sc". It all comes down to this: the current solution is tailor-made for this particular program, and would require significant adaptation to be used anywhere else; I want to re-design the Function so that it can be used anywhere to begin with, while still defaulting to the behaviour desired for this program.

You can skip this section, and return later. The actions for the rest of the tutorial remain unchanged whether you do or do not make the modifications in this section.

Here's what I have from the previous step:

```

var t_c = TempoClock.default;

var secondPart =
{
  var so = Array.new( 10 );

  var func =
  {
    var frequency = 200 + 600.rand;
    [ SinOsc.ar( freq:frequency, mul:0.01 ), SinOsc.ar( freq:frequency, mul:0.01 ) ];
  };

  10.do( { arg index; t_c.sched( ((5*index)+1), { so =
so.add( {func.value;}.play; ); } ); } );
  10.do( { arg index; t_c.sched( 51, { so[index].free; } ); } );

};

```

This Function is the perfect solution if you want ten pseudo-random pitches between 200 Hz and 800 Hz, and a five-second pause between each one. If you want nine or eleven pitches, if you want them to be between 60 Hz and 80Hz, if you want a six-second pause between each - you would have to modify the Function. If you don't remember how it works, or if you give it to a friend, you're going to have to figure out how it works before you modify it. This is not an ideal solution.

Let's solve these problems one at a time, starting with allowing a different number of SinOsc synths to be created. We know that we'll have to create an argument, and that it will have to be used wherever we need the number of SinOsc's. Also, to preserve functionality, we'll make a default assignment of 10. Try to accomplish this yourself, making sure to test your Function so that you know it works. Here's what I did:

```
var t_c = TempoClock.default;

var secondPart =
{
  arg number_of_SinOscs = 10;

  var so = Array.new( number_of_SinOscs );

  var func =
  {
    var frequency = 200 + 600.rand;
    [ SinOsc.ar( freq:frequency, mul:0.01 ), SinOsc.ar( freq:frequency, mul:0.01 ) ];
  };

  number_of_SinOscs.do( { arg index; t_c.sched( ((5*index)+1), { so =
so.add( {func.value;}.play; ); } ); } );
  number_of_SinOscs.do( { arg index; t_c.sched( 51, { so[index].free; } ); } );
};
```

The "do" loop doesn't need a constant number; it's fine with a variable. What happens when you pass a bad argument, like a string? This would be an easy way to sabotage your program, and in almost any other programming context it would concern us, but this is just audio programming. If somebody is going to try to create "cheese" SinOsc's, it's their own fault for mis-using the Function.

Now let's modify the Function so that we can adjust the range of frequencies that the Function will generate. We know that we'll need two more arguments, and that they'll have to be used in the equation to calculate the frequency. But we'll also need to do a bit of arithmetic, because of the way the "rand" Function works (actually we don't - see the "rand" Function's help file). Also, to preserve functionality, we'll make default assignments of 200 and 800. Try to accomplish this yourself, making sure that you test the Function so you know it works. Here's what I did:

```
var t_c = TempoClock.default;

var secondPart =
{
  arg number_of_SinOscs = 10,
    pitch_low = 200,
    pitch_high = 800;

  var so = Array.new( number_of_SinOscs );

  var func =
  {
    var freq = pitch_low + (pitch_high - pitch_low).rand;
    [ SinOsc.ar( freq:freq, mul:0.01),
      SinOsc.ar( freq:freq, mul:0.01) ];
  };

  number_of_SinOscs.do( { arg index; t_c.sched( ((5*index)+1), { so =
so.add( {func.value;}.play; ); } ); } );
  number_of_SinOscs.do( { arg index; t_c.sched( 51, { so[index].free; } ); } );
};
```

Notice that I changed the name of the variables, and the indentation in the "func" sub-Function, to make it easier to read. This isn't a particularly difficult change.

Now let's allow the user to set the length of time between each SinOsc appears. We will need one more argument, used in the scheduling command. Try to accomplish this yourself, and if you run into difficulty, the next paragraph contains some tips.

The change to the "do" loop which schedules the SinOsc's to play is almost trivial. My new argument is called "pause_length", (meaning "the length of the pause, in seconds, between adding each SinOsc"), so I get this modification: number_of_SinOscs.do(

```
{
  arg time;
  secondPart_clock.sched( (1+(time*5)), { sounds = sounds.add( func.play ); } );
});
```

Again, I changed the indentation, and the names of the variables in this sub-Function. Recall that the "1+" portion is designed to add a one-second pause to the start of the Function's execution. The problem comes in the next "do" loop, where we have to know how the number of beats from now will be five seconds after the last SinOsc is added. We'll have to calculate it, so I added a variable to store the value after it's calculated. This also allows us to return it, as a convenience to the Function that called this one, so that it knows how long until this Function is finished. Try adding this yourself, then testing the Function to ensure that it works. I got this:

```
var t_c = TempoClock.default;

var secondPart =
{
  arg number_of_SinOscs = 10,
    pitch_low = 200,
    pitch_high = 800,
    pause_length = 5;

  var so = Array.new( number_of_SinOscs );

  var when_to_stop = ( 1 + ( pause_length * number_of_SinOscs ) );

  var func =
  {
    var freq = pitch_low + (pitch_high - pitch_low).rand;
    [ SinOsc.ar( freq:freq, mul:0.01),
      SinOsc.ar( freq:freq, mul:0.01) ];
  };

  number_of_SinOscs.do(
  {
    arg time;
    t_c.sched( (1+(time*5)), { so = so.add( func.play ); } );
  });

  t_c.sched( when_to_stop,
    {
      number_of_SinOscs.do( { arg index; so[index].free; } );
      nil;
    });

  when_to_stop;
};
```

I decided to "invert" the "free-ing" of the SinOsc's. Rather than scheduling `number_of_SinOscs` Function-calls at some point in the future, I decided to schedule one thing: a "do" loop that does the work. The indentation looks strange, but sometimes there's not much you can do about that. The "when_to_stop" variable must be the last thing in the Function, so that the interpreter returns it to the Function's caller.

In order to retain the "bare minimum" robustness to be used elsewhere, we can't rely on the "TempoClock.default" clock having the tempo we expect, and we certainly can't rely on it being declared as `"t_c"`. The solution is quite easy: create a new `TempoClock` within the Function.

```
var t_c = TempoClock.new; // default tempo is one beat per second
```

We could hypothetically use the "SystemClock", since we're measuring time strictly in seconds. But, using a `TempoClock` is preferred for two reasons:

1. It has the word "tempo" in its name, and it's designed for scheduling musical events; the "SystemClock" is for system events.
2. We can easily extend this in the future to use a "TempoClock" set to a different tempo.

There are some further ways to improve this Function, making it more robust (meaning that it will work consistently in a greater range of circumstances). Here are things that could be done to improve the Function, with an explanation of why it would make the Function more robust:

- Made the clock an argument, allowing this Function to schedule events on a clock belonging to some other Function. Since all clocks respond to the "sched" message, we could even accept the "SystemClock" or "AppClock". The default value would still be **TempoClock.new**.
- Use absolute scheduling rather than relative scheduling. Depending on how long the server and interpreter take to process the commands, it could lead to significant delays if the Function is asked to create a lot of SinOsc's.
- Create one SynthDef (with an argument) for all of the synths. Especially when asked to create a large number of SinOsc's, this will lead to faster processing and lower memory consumption. On the other hand, it increases the complexity of the code a little bit, requiring more testing.
- Each SinOsc is currently created with the same "mul" argument, regardless of how many SinOsc's are created. Set as it is, when asked to create 51 SinOsc's, the signal would become distorted. If you're puzzled about why 51, remember that for each SinOsc the Function is asked to create, it currently creates two: one for the left and one for the right audio channel.
- Allow the SynthDef to be passed in as an argument, with the requirement that such a SynthDef would need to accept the "freq" and "mul" arguments. This is going out on a limb a bit, and requires careful explanation in comments to ensure that the Function is used correctly. You will also need to test what happens if the Function is used incorrectly. Crashing the server application is a bad thing to do, especially without a warning.
- Use a Bus to cut the number of synths in half, so that one synth will be sent both to the left and right channels. Alternatively, you could add special stereo effects.

As you can see, there are a lot of ways to improve this Function even further; there are almost certainly more ways than listed here. Before you distribute your Function, you would want to be sure to test it thoroughly, and add helpful comments so that the Function's users know how to make the Function do what they want. These are both large topics in themselves, so I won't give them any more attention here.

11.4.9. Joining the Two Parts

Now it is time to join the two parts, and ensure a clean transition between them. My reasons for building the first part as a SynthDef, but the second part as a function are explained in the FSC_part_1.sc file. Additional reasons include my desire to illustrate the use of both possibilities, and because the second part stops itself (so it can be a function which is executed and forgotten), whereas the first part does not stop itself (so we'll need to hold onto the synth, to stop it ourselves).

1. I copy-and-pasted both parts into a new file, leaving the other original code in tact, in case I want to build on them in the future. Be sure to copy over the `var t_c = TempoClock.default;` definition from the second part.
2. By default, the two parts would both start playing at the same time (give it a try!) This isn't what we want, however, so you'll need to erase the "play" command from both parts' functions. We'll also need some way to refer to them, so declare the second part as a variable (I've used the name, "secondPart,"), but don't worry about the first part yet. Don't forget the semicolon at the end of the function declaration!
3. To join the two parts, I'm going to use function that does all the scheduling. This is similar to a "main" function, which are used in most programming languages. Although they are optional in SuperCollider, it just makes sense to use one function that does all the scheduling, and nothing else: that way, when you have problems with the scheduling, or you want to make an adjustment or addition to the program, you can easily find the place where the scheduling happens. If your scheduling commands were spread out through the source file, it would be much more difficult to find and modify the scheduling commands.
4. Our first job is to determine which variables we'll need to use: just one, which will be assigned the currently-running \FirstPart Synth. Also, if you didn't previously assign "TempoClock.default" to the variable "t_c", then it makes sense to do this now.
5. The next thing our function must do is guarantee that we're going to have the right tempo. Use the "tempo_" Function with an argument in beats-per-second, to assign "TempoClock.default" a tempo of one beat per second.
6. The next and last thing will be to schedule our sounds. First, we need to determine which events will need to be scheduled, and then at what times.
 - a. Since \FirstPart is a SynthDef, we'll need to start it and stop it ourselves. Since it happens two times in the intended program, we'll need to do it twice.
 - b. secondPart is a Function, and it stops itself when it's finished. We'll need to start it once and let it go.
 - c. Just in case something takes a while to process, we'll start the first \FirstPart on beat one, rather than beat zero. We'll let it play for 60 seconds the first time, and 30 seconds the second time.
 - d. In order to schedule the second appearance of \FirstPart, we need to know how long secondPart will take. Let's inspect the function and calculate how many beats it will take.
 - 1 beat of silence at the beginning,
 - 5 beats between the entrance of each SinOsc,
 - 10 SinOsc's,
 - 5 beats after the last SinOsc until the function stops.

- This gives us $1 + (5 * 9) + 5 = 51$. Why $5 * 9$? Because although there are ten SinOsc's, there are only nine spaces between them; the last five-second space happens *after* the last SinOsc.
- e. This gives us the following schedule:
- 1 beat: start \FirstPart
 - 61 beats: stop \FirstPart
 - 61 beats: start secondPart
 - 113 beats: start \FirstPart
 - 143 beats: stop \FirstPart
7. Try to schedule the events for yourself, then test your program to make sure that it works as you intended. Here's what I wrote:

```
t_c.sched( 1, { sound = Synth.new( \FirstPart ); } );
t_c.sched( 61, { sound.free; } );
t_c.sched( 61, { secondPart.value; nil; } );
t_c.sched( 113, { sound = Synth.new( \FirstPart ); } );
t_c.sched( 143, { sound.free; } );
```

Why is the "nil" required after "secondPart"? Because that function returns a number. As you know, any scheduled function which returns a number will re-schedule itself to run that many beats after the previous execution began. Since "secondPart" returns the number of seconds it takes to finish, it will always be re-started as soon as it finishes. Including "nil" disallows this repetition.

11.5. Exporting Sound Files

This section explains one way to record your SuperCollider programs, so that you can share them with friends who don't have SuperCollider on their computer.

11.5.1. Non-Real-Time Synthesis

SuperCollider allows you to synthesize audio output to an audio file. Doing this requires using OSC commands on the server, the **DiskOut** UGen, the **Buffer** UGen, and other relatively advanced concepts. The built-in *DiskOut* help file, available from <file:///usr/share/SuperCollider/Help/UGens/Playback%20and%20Recording/DiskOut.html> on Fedora Linux systems, contains some help with the **DiskOut** UGen, and links to other useful help files. This method is not further discussed here.

11.5.2. Recording SuperCollider's Output (Tutorial)

Since SuperCollider outputs its audio signals to the JACK sound server, any other JACK-aware program has the opportunity to record, process, and use them. This portion of the tutorial will help you to record SuperCollider's output in Ardour. Due to the advanced nature of SuperCollider, the text assumes that you have a basic knowledge of how to work with Ardour. If not, you may find it helpful to refer to [Chapter 7, Ardour](#).

This procedure will help you to use **Ardour** to record the **SuperCollider** output.

1. Close unnecessary applications and stop unnecessary processes, which will help to reduce the risk of a buffer overrun or underrun, which cause an audible break in audio. If you are viewing this

document in a web browser, you may want to copy-and-paste it into a simple text editor, or **GEdit**, if you are already using that.

2. Use **QjackCtl** to set up JACK with the right audio interface and configuration options.
3. In order to get a clean start, restart the **SuperCollider** interpreter in **GEdit**, then start the server.
4. Open **Ardour** with a new session, and set up the rulers and timeline as desired. Seconds is usually the most appropriate unit with which to measure a **SuperCollider** recording.
5. Add a stereo track (or however many channels desired), and rename it it "SuperCollider."
6. Use Ardour (the "Track/Bus Inspector" window) or **QjackCtl** to connect the "SuperCollider" track to SuperCollider's outputs.
7. You'll want to make sure that the **SuperCollider** output is also connected to your audio interface, so that you can hear the program as you progress. This is an example of multi-plexing. Changes to your audio interface's volume control will not affect the recording in Ardour.
8. Arm the track and transport in **Ardour**. When you are ready, start the transport. It is not important to start SuperCollider as quickly as possible, since you can cut out the silence after the recording is made.
9. Switch to **GEdit** and play the program that you want to record. If you make a mistake while starting the program, that's okay. We can always edit the recording after it's recorded.
10. Listen to the recording as it goes along. Use **QjackCtl** to make sure that you don't encounter a buffer underrun, and **Ardour** to make sure that you do not record a distorted signal.
11. When **SuperCollider** has finished playing your program, switch to **Ardour**, and stop the transport.
12. When you are ready to export, use the **Ardour** menu. **Choose Session → Export → Export session to audio file**
13. The audio file will be created in the "export" sub-directory of the session's directory.

DRAFT

LilyPond

LilyPond is a notation engraving program, with a focus on creating a visually appealing product.

LilyPond is text-based, and allows you to focus on the (semantic?) content of your musical scores, rather than on their visual appearance. Conventional commercial notation engraving programs allow users to edit the score visually. While this approach has its benefits, especially because it's very easy to see exactly what the printed score will look like, it also has disadvantages - chief among these is the fact that users of those programs are constantly worrying about what their score looks like.

This is where **LilyPond** comes in - users don't need to worry about how their score will work, because they know that the expertly-crafted methods of **LilyPond** will automatically configure the objects on the score so that they look good, and are easy to read. **LilyPond**'s users focus on *what* needs to be displayed, rather than on *how* it is going to be displayed.

As with any particular approach, the **LilyPond** approach is not for everybody. However, once you have become accustomed to working with the software, and once you have learned methods to help deal with problems and organize your scores' source-files, you will probably realize that **LilyPond** is both much faster, and much more flexible, than traditional, commercially-available music engraving programs.

LilyPond offers many other features, too. Some of these features include:

- Putting scores into LaTeX or HTML documents.
- Putting scores into **OpenOffice.org** documents, with the **oolilypond** program.
- Being compatible with all major operating systems.
- Managing parts and full scores for large compositions.
- Allowing new musical symbols with the Scheme programming language.

It is the goal of this guide to help users more quickly overcome the initial learning handicap incurred because of the text-based approach. Making use of tools such as the **Frescobaldi** text-editor will help to increase productivity, to make trouble-shooting easier, and to ease the memory burden associated with the text-based approach.

12.1. How LilyPond Works

Think of **LilyPond** as an automobile mechanic. When your car breaks down, the mechanic knows which tools to use. You can buy tools to fix your car by yourself, but the mechanic is specialized. The mechanic knows what tools to use, how to prepare the tools, and how to fix your car faster than you can fix it. **LilyPond** uses many programs that you can use by yourself, but **LilyPond** is specialized. **LilyPond** knows what programs to use, what settings to use, and most importantly, **LilyPond** takes much less time than if you use the programs directly.

We give instructions to **LilyPond** in specially-formed text files. **LilyPond** input files describe the music to notate. **LilyPond** decides how the music will look, then creates an output file. The input file does not contain instructions about how the music looks. Sometimes you must make an adjustment to how the output file looks, so **LilyPond** lets you change the settings of its internal tools.

12.2. The LilyPond Approach

For an extensive explanation of the following section, please see the **LilyPond Website** at <http://www.lilypond.org/about/automated-engraving/>, from where this was sourced.

LilyPond works by separating the tasks of what to put, and where to put it. Each aspect of an object's position is controlled by a specific plug-in. You can think of the plug-ins as tools. **LilyPond** knows how and when to use each tool; if it doesn't know enough about a tool, then it isn't used.

Before **LilyPond** places an object, it first considers many different possibilities for the specific alignment and layout of that object. Then it evaluates the possibilities according to aesthetic criteria set out to reflect those used in hand-engraved notation. After assigning each possibility a score representing how closely it resembles to the aesthetic ideal, **LilyPond** then chooses the better possibility.

12.3. Requirements and Installation

1. Use PackageKit or KPackageKit to install the **lilypond** package.
2. Review the dependencies. Many packages called **lilypond- *-fonts** are installed.
3. **LilyPond** is run from the command line, with the command **lilypond**.

We recommend that you use the **Frescobaldi** text editor, which is designed specifically for **LilyPond**. It has many features that enhance productivity when editing **LilyPond** files, and that greatly speed up the learning process. Refer to [Chapter 13, Frescobaldi](#) for more information.

12.4. LilyPond Basics

The syntax of **LilyPond** files offers the most flexibility through the most diverse musical conditions. Over time, you will realize that features which seem too complex at first are really a very powerful and simple way to solve complex problems that commercial programs cannot.

12.4.1. Letters Are Pitches

One letter is all that's required to create a note in **LilyPond**. There are additional symbols and letters that are added to indicate further details, like the register, and whether the note is "sharp" or "flat."

Although it can be changed, the default (and recommended) way to indicate "sharp" or "flat" is by using Dutch note-names: "-is" to indicate a sharp, and "-es" to indicate a flat. For example, the following command would create b-double-flat, b-flat, b, b-sharp, and b-double-sharp: **beses bes b bis bisis** Getting used to these names happens quickly, and they take less time to input than the English alternative. Furthermore, with these names, it becomes possible to sing note-names when you are ear training!

Pitch can be entered either absolutely, or relative to the preceding notes. Usually (for music without frequent large leaps) it is more convenient to use the "relative" mode. The symbols `,` and `'` (comma and apostrophe) are used to indicate register.

When entering absolute pitches, the register is indicated mostly as in the Helmholtz system (see *Helmholtz Pitch Notation* (Wikipedia) at http://en.wikipedia.org/wiki/Helmholtz_pitch_notation: octaves begin on the pitch "C," and end eleven tones higher on the pitch "B." The octave below "middle C" (octave 3 in scientific pitch notation - see *Scientific Pitch Notation* (Wikipedia) at http://en.wikipedia.org/wiki/Scientific_pitch_notation *Scientific Pitch Notation*) has no commas or apostrophes. The octave starting on "middle C" (octave 4) has one apostrophe; the octave above that (octave 5) has two apostrophes, and so on. Octave 2 (starting two octaves below "middle C") has one comma, the octave below that has two commas, and so on. It is usually not necessary to understand how to use this in **LilyPond**, or to be able to use it quickly, because most scores will use "relative mode."

When entering pitches relative to the previous one, the register is still indicated with commas or apostrophes, but usually none are needed. When using this input mode, the octave of each note is guessed based on the octave of the previous note. Think of it this way: the next note will always be placed so it produces the smaller interval. For example, after a C, an E could be placed as a major third, a minor sixth, a major tenth, a minor thirteenth, and so on. In relative mode, **LilyPond** will always choose the "major third" option. If you wanted **LilyPond** to notate the E so that it's a minor sixth, you would tell **LilyPond** with a comma appended: **c e,** so that **LilyPond** knows what you want. It's the same case if you were to input **c aes** (meaning "C then A-flat"): the A-flat will be notated so that it is a major third from the C; if you wanted **LilyPond** to notate it so that the A-flat is a minor sixth higher than the C, you would need to append an apostrophe: **c aes'**

The only possible ambiguity with this method is with a tritone. **LilyPond** solves this by not recognizing "tritones," per se, and thinking of them as "augmented fourth" or "diminished fifth." Unless instructed otherwise (with a comma or apostrophe), **LilyPond** will always notate the interval as an augmented fourth.

You must always indicate a sharp or flat, even if it is already in a key signature. This ultimately helps to reduce the number of errors.

Letters used to indicate pitch are always in lower-case.

12.4.2. Numbers Are Durations

A number appended to a letter is understood by **LilyPond** to indicate that particular note's note-value. A whole note is indicated with a 1, and all other durations are indicated with a number representing the fraction of a whole note that they occupy: half notes are 2 (like "1/2 note"); quarter notes are 4 (like "1/4 note"); eighth notes are 8 (like "1/8 note") and so on.

To add a "dot" to a note (thereby increasing its length by one half), you simply include a period after the number. For example, **e4.** means "dotted quarter note on E."

To add a "tie" from one note to the next (thereby continuing its duration across a measure-line), add a tilde (~) after the pitch and duration.

After indicating a duration, it is assumed that all subsequent notes have the same duration, until indicated otherwise.

12.4.3. Articulations

Many different symbols are used to tell **LilyPond** to add articulation signs to a note. They are all appended after the pitch and (if included) duration, and many have a position indicator, too.

A full list of articulation markings is available in the **LilyPond** manual, and **Frescobaldi** remembers most of them for you (they are stored in the left-side panel).

These are some of the most common articulation marks, which use a position indicator unless specified otherwise:

- (to begin a slur (no position indicator)
-) to end a slur (no position indicator)
- ~ to begin a tie (which needs no end; no position indicator)
- . for a "staccato" mark
- > for an "accent"
- - for a "tenuto" mark

- ^ for a "marcato" mark
- _ for a "portato" mark (dot and line)

There are three position indicators:

- - which means to put the articulation mark wherever **LilyPond** thinks it makes sense
- _ which means to put the articulation mark *below* the note-head
- ^ which means to put the articulation mark *above* the note-head

These position indicators will sometimes result in notes like: **g4 - -**, **g4_**, and **g4^^**, but although this may look incorrect, it is perfectly acceptable.

12.4.4. Simultaneity

Simply put, anything enclosed inside **<<** and **>>** is considered by **LilyPond** to happen simultaneously. This can be used in any context (and any Context - see above/below). It is possible to tell **LilyPond** that you want two notes to happen at the same time in the same voice (yielding a chord), at the same time on the same staff (yielding polyphony), and so on. Moreover, any score with multiple staves will use **<<** and **>>** to tell **LilyPond** that the parts should begin at the same time, and creative use of **<<** and **>>** is one of the keys to advanced notation.

It is not important to understand simultaneity at first. By observing how **Frescobaldi** creates scores for you, and how examples on the internet take advantage of these symbols, you will eventually understand how to use them.

12.4.5. Chords

Making use of the **<<** and **>>** idea to indicate simultaneity, if a note has multiple pitches indicated between **>** and **<** then **LilyPond** assumes that they are in a chord together. Notating a single chord with single **<** **>** brackets has two advantages: firstly, it is easier to see that they are a chord and not something more complex; secondly, it allows you to enter information more clearly.

Consider the following examples, which should produce equivalent output: **<<g'4->-5 b d>>** and **<g' b d>4->-5** With the first example, it is more difficult to see the chord notes, the duration, and what the **5** means. With the second example, it is easy to see that the chord notes are a **G**, a **B**, and a **D**, that they have quarter-note duration, and that the **5** is actually a fingering indication.

There is another advantage to using **<** and **>** for notation of simple chords: they preserve logical continuity in "relative" mode. The following note will always be notated as relative to the lowest note in the chord, regardless of how many octaves the chord covers. This is not true with **<<** and **>>**, where following notes will be notated as relative to the last note between the **<<** and **>>**

12.4.6. Commands

There are a wide variety of commands available in **LilyPond**, some of them quite simple, and other quite complex. They all begin with a backslash, followed by the name of the command, and subsequent "arguments" that give the command further information about what you want it to do. Just like using a letter to indicate a note, commands are simply another way for you to tell **LilyPond** how you want your score to be rendered.

For example, **\time 2/4** and **\clef bass** are two commands that you are likely to use quite often. They happen to do precisely what it seems like they should: **\time** changes the time signature and metre, and **\clef** changes the clef. they belong to different contexts (**\time** applies for the whole Score, but **\clef** for only one Staff).

It can take some time to remember even these basic commands and the way you should format their input, and this is where **Frescobaldi**'s built-in documentation viewer can help out. All of the official **LilyPond** documentation is made available in **Frescobaldi**, which makes it easy for you to view the documentation and make changes to your score in the same window of the same application.

12.4.6.1. Customization

It is rarely necessary to customize the output in a way that is very specific, and not allowed for in the standard **LilyPond** syntax. As a beginner, this can happen quite often when you are trying to exactly emulate the look of a pre-existing score. Remember that **LilyPond** provides a content-focussed way to express music, and that it will usually produce meaningful output without advanced interference. If in doubt about whether a customization is really necessary, ask yourself this: will it change the music that is played from the score?

If you really must customize some setting, then keep in mind these two points:

1. Tinkering with **LilyPond** can become as complex as you want.
2. Ultimately all tinkering takes the form of commands.

Searching the internet for **LilyPond** tips and tricks can be a life-saver, but it can also lead to needlessly complex solutions. Sometimes this is the result of poor solutions being posted on the internet, but more often it is the result of the ongoing development of **LilyPond**, which makes better solutions available regularly. For this reason, it is recommended to search the official **LilyPond** documentation first, then the "**LilyPond** Snippet Repository" (LSR - [link here](#)), and then Google.

12.4.6.2. Contexts

Another aspect of **LilyPond** that often confuses beginners is the idea of contexts. It is an essential concept for modifying default behaviour. Like everything in **LilyPond**, it makes perfect sense: "context" means "context." The three primary contexts are "Voice," "Staff," and "Score." Everything that happens in a score happens in a context - it's just that simple - everything happens in a context! Everything that happens in a score happens in Score context, everything that happens on a staff happens in a Staff context, and everything that happens in a voice happens in a Voice context.

To help clear things up a little, here are three examples:

1. Where does the title of a composition belong?
 - Does it belong on the score? Yes, so it belongs in the Score context.
 - Does it belong on a staff? No, so it doesn't belong in a Staff context.
 - Does it belong in a voice? No, since it would have to be on a staff to be in a voice.

The composition's title doesn't belong on a staff or in a voice, but it does belong on the score, so we say that it happens in the Score context.

2. Where does a clef belong?
 - Does it belong on the score? Yes, because it is a part of notation.
 - Does it belong on a staff? Yes, because it wouldn't make sense to have a clef off a staff.
 - Does it belong in a voice? No, because it's not related to a specific "line" or voice.

Clefs usually belong on the staff, at the beginning of every line, so we say that they happen in a Staff context.

3. Where does a note-head belong?
 - Does it belong on the score? Yes, because it is a part of notation.

- Does it belong on a staff? Yes - even if it's on a ledger line, note-heads are meaningless unless they're on a staff.
- Does it belong in a voice? Yes, because one particular musical line is indicated primarily with note-heads.

Note-heads belong to a particular voice, so we say they happen in a Voice context.

To help further clarify, consider the following:

As in the real world, objects in **LilyPond** can potentially happen in any context...

- (like a doorknob on a door)
- (like a bus driver sitting in a bus' drivers' seat)
- (like a person buying groceries at a supermarket)
- (like a flat-sign next to a note-head)

... some contexts are unusual, but make sense...

- (like a doorknob on a wall, if the wall is designed to look like a door)
- (like a bus driver sitting in a passenger seat, if they are going to the garage)
- (like a person buying groceries from a corner store, if the supermarkets are closed)
- (like a flat sign in a paragraph of text, if that paragraph describes what flat signs do)

... and some contexts do not make sense, but can still happen...

- (like a doorknob on a television)
- (like a bus driver serving patrons at a restaurant)
- (like a person buying groceries at a governmental office)
- (like a flat sign in the top corner of a score, where the page number should be)

The **LilyPond** designers wisely decided that they could not think of all possible uses for their software, so **LilyPond** allows most engravers to be used in most contexts. Furthermore, a context can happen within an other context, which makes sense - a **Voice** context only makes sense if it appears on a **Staff**, and a **Staff** only makes sense if it appears in a **Score**.

So, when trying to sort out the context in which something should apply, ask yourself exactly that: "In what context does this apply?" Beams and flags happen in one voice, accidentals apply to a whole staff, and tempo markings apply to the whole score. Although it may take some careful thought to get used to the idea, contexts ultimately make perfect sense.

12.4.7. Source Files

Source files are the text files prepared with instructions telling **LilyPond** the content of the score you want it to create. They are so called because these files are the "source" of what you wish to create. As with programming languages, the text inside these files is often referred to as "source code." It sounds scary to think that you must edit code in order to use **LilyPond**, but "code" just means that it isn't normal English (or insert-language-here).

The particular formatting (the placement of tabs, spaces, and newlines) is not determined by **LilyPond**, but by individual users. This can be a headache when you encounter somebody else's

formatting, but it is one of the keys to the application's flexibility. This guide uses a very specific style of formatting, but it should be understood that this is simply the style of one user, affected by their experiences, and the tasks they usually perform in **LilyPond**.

You will eventually develop your own style, better-suited to the kinds of tasks that you accomplish. When you do this, there is only one rule to keep in mind: be consistent within source files. When source files are programmed in a consistent way, it means that anybody who wants to use those files (like yourself, in the future) will easily be able to determine how they are organized.

12.4.7.1. Organizing Source Files

LilyPond files are constructed as a series of commands. For better or worse, the **LilyPond** interpreter allows a great deal of flexibility when it comes to source file setup. This can lead to confusion about where things should be done, especially when using automated score-setup tools like **Frescobaldi**.

The generic structure of a **LilyPond** source file is this:

```
\version version_number

\header { things like title, composer, and so on }

\score
{
  \new Staff
  {
    notes go here
  }

  \layout
  {
  }

  and so on
}
```

Confusion arises here: for maximum flexibility, **LilyPond** allows source files to create its own commands. On hearing this, you may think, "Okay that's it - I don't need advanced commands or any of this stuff, so I'm packing it in and just using Finale!" There's no need to do that just yet - commands are easy! Think of commands - whether you wrote them or they were included with **LilyPond** - as a means of text-substitution.

It works like this:

1. You "define" (create) a command with the form **commandName = { lots of commands }**
2. You can then use the command anywhere below that, as many times as you want, by writing **\commandName** in your source file. When **LilyPond** processes that portion of text, it will instead see whatever you wrote in the definition.

It's as easy as 1-2!

Frescobaldi (along with most **LilyPond** users) take advantage of this functionality to provide well-organized, easy-to-use source files.

Here is a good template source file, that might be created for you by **Frescobaldi**:

```
\version "2.12.2"
```



```

\header
{
  title = "Empty Example"
}

violin = \relative c''
{
  \key c \major
  \time 4/4
  % Music follows here.

  Put your music here
}

\score
{
  \new Staff \with
  {
    instrumentName = "Violin"
  }
  \violin
  \layout { }
}

```

This example uses many built-in commands (like `\relative` and `\key`), and it defines the **violin** command. **Frescobaldi** places a `% Music follows here.` comment where you should input your notes. Notice that your notes belong in the **violin** section. This means that you write part of the **violin** command.

Frescobaldi separates the violin's notes from the `\score` section so that the source file is easier to read. You can fix errors more easily in well-organized source files. If you complete the following tutorials, you will learn three strategies to organize sources files.

The `\layout` section is empty in this example. Like the `\score` section, `\layout` is a command. The `\layout` command tells LilyPond to output a musical score. Advanced users sometimes remove the `\layout` command, but most source files use it.

12.4.8. How to Avoid Errors

The **LilyPond** syntax has two built-in symbols for helping to avoid musical errors. The octave-check symbol ensures that your notes are in the octave you intend. The bar-check symbols ensures that your barlines are where you intend.

12.4.8.1. The Octave-Check Symbol

The octave-check symbol compares a relative pitch with its absolute pitch equivalent. **LilyPond** prints a warning if the pitches do not match, then continues with the absolute pitch. You should correct the relative pitch when **LilyPond** prints a warning.

The octave-check symbols is `=`. The symbol appears after the note, and is followed by a comma (,), apostrophe ('), or neither, depending on the intended relative octave of the pitch.

Here is an example use of the octave-check symbol: `c' = ''`. In this example, the absolute pitch is `c''`, which **LilyPond** knows because of the `c` at the left, and the `''` after the `=` symbol.

How does this example work: `c' = 4` ? There is a **4** after the `=` symbol instead of a comma or apostrophe. The absolute pitch is `c`, which **LilyPond** knows because of the `c` to the left of the `=` symbol, and because there is no comma or apostrophe to the right of the `=` symbol. **LilyPond** understands the **4** as "quarter note."

You should use the octave-check symbol when you need it. New users use the octave-check symbol more frequently than experienced users.

12.4.8.2. The Bar-Check Symbol

The bar-check symbol tests whether a measure line is printed at that point in the music. **LilyPond** prints a warning if the bar-check symbol does not line up with a measure line in the score. **LilyPond** continues to process the music, and does not automatically change any durations.

The bar-check symbol is the pipe character, `|`. We recommend that you put a bar-check symbol at the end of every measure, and that you input only one measure on every line. If you follow this recommendation, you will make fewer mistakes.

This example is correct, so **LilyPond** will not print a warning:

```
\time 4/4
c4 c c c |
c4 c c c |
```

There are four quarter notes between each bar-check symbol, which is the right number of beats in the indicated metre.

This example is incorrect, so **LilyPond** will print a warning.

```
\time 4/4
c4 c2 c4 |
c4 c2 c |
c4 c2 c4 |
```

This example shows three measures of the same pattern: **c4 c2 c4**. The second measure is missing the last quarter-note symbol (a **4**), so **LilyPond** outputs one quarter note followed by two half notes. The second measure is five beats, so the bar-check symbol does not coincide with the measure line after the fourth beat. **LilyPond** prints a warning when it reaches the bar-check symbol at the end of the second measure.

Consider this example.

```
\time 4/4
c2 c c c |
c4 c c c |
```

The first bar has four half-notes, which is twice as many beats as are allowed in one measure. A bar-check symbol only triggers a warning if it is placed where there is no bar line. Therefore, **LilyPond** will not print a warning for this example, even though only two of the three bar lines has a bar-line check symbol.

One mistake in a file will sometimes cause **LilyPond** to print multiple warnings. A bar-check symbol does not force **LilyPond** to insert a bar line. **LilyPond** continues to insert notes with the duration shown in the file. Look at the following example, and the **LilyPond** output.

```
\time 4/4
c4 c2 c4 |
```

```
c4 c2 c |  
c4 c2 c4 |
```

```
Interpreting music...  
example.ly:3:10: warning: barcheck failed at: 1/4  
c4 c2 c  
|
```

The mistake in the second measure causes a warning at the bar-check symbol. **LilyPond** does not modify note durations, so the half note at the end of the second measure occupies the fourth beat of the second measure *and* the first beat of the third measure. The third measure is correct, with four beats, but the bar-check symbol causes a warning to be printed because **LilyPond** expects a bar line before the last quarter note, not after it.

You can avoid confusion if you fix the first warning, then reprocess the file before you fix other warnings. One mistake may cause many warnings.

12.5. Work on a Counterpoint Exercise (Tutorial)

Imagine you're in Counterpoint class, and you've been asked to submit a very clean copy of your next assignment. Since you don't want to pay \$450,000 for a commercially-available engraving solution and a fruity computer to use it, you decide that LilyPond is the solution for you.

12.5.1. Files for the Tutorial

You do not need these files to do the tutorial. They are example completions.

- *LilyPond Input File* at http://docs.fedoraproject.org/en-US/Fedora/15/html/Musicians_Guide/files/LilyPond/Counterpoint-source.ly
- *PDF Output File* at http://docs.fedoraproject.org/en-US/Fedora/15/html/Musicians_Guide/files/files/LilyPond/Counterpoint-result.pdf

12.5.2. Start the Score

1. Open Frescobaldi into a blank document
2. Use the 'LilyPond > Setup New Score' dialogue
3. Set the title and composer (and any other field) on the 'Titles and Headers' tab
4. You need a two staves, so you go to 'Parts' then 'Keyboard' then 'Piano'
5. You need to label the intervals between the notes, so you choose 'Special' then 'Figured Bass' (we'll put it in the right spot later)
6. You go to the 'Score settings' tab
 - a. since you've already played through this example, uncheck "Create MIDI output" and save processing time
 - b. your example is in F Major
 - c. You want only whole notes, with no barlines; choose a 4/4 time signature, since we'll change this later.

7. Press 'Try' to get a preview of the score setup - the notes are demonstrative and will not be there
8. If the setup looks like you want it, then press 'OK' to generate the template

12.5.3. Adjust Frescobaldi's Output

For this tutorial, Frescobaldi's default output is good enough.

12.5.4. Input Notes

1. Look over the template. It's not important to understand what any of this means, but it's good if you can figure it out.
2. The piano's upper staff will contain the notes between the { and } following the **right = \relative c''** portion
3. The piano's lower ... left
4. The figured bass will say what you put in figBass
5. start with the piano's upper part. Input "f1 g a f d e f c a f g f" (explain)
6. Check that it's right. Preview by press the "LilyPond" button on the toolbar
7. The exercise starts too high, and it ends too low. Change the starting pitch by "right = \relative c' "
8. Preview again; of course, it still ends too low (these simple exercises usually start and end on the same pitch).
9. Put a ' right after the c so that it goes to the upper C
10. Preview again, and this time it's right.
11. Now, enter the notes that you've written for the piano's lower staff: "f1 e d c bes c bes f d bes e f"
12. Preview the output again, and see that it starts too high, but ends on the right pitch.
13. Fix the start-too-high by changing to "left = \relative c { "
14. Preview again, and decide where to fix the other change (NB: I end up with "f1 e d c bes c bes f d bes e f"
15. Enter the interval number between the top and bottom between the { and } preceded by "figBass = \figuremode "
16. You have to put each interval between < > brackets. The note-length of the figure's duration goes after the >, so I end up with "<1>1 <3> <5> <4> <3> <3> <5> <5> <5> <5> <3> <1>"
17. Now you realize that there are some significant errors in your work. The top staff is the cantus, and cannot change. You have to correct the lower staff.
18. I've ended up with a lower part that is "f1 e d a bes c d e f d c f", which gives figures that are "<1>1 <3> <5> <6> <3> <3> <3> <6> <3> <3> <5> <1>"

12.5.5. Format the Score

We recommend formatting the score after the notes have been inputting. Formatting after inputting helps you avoid making formatting errors. LilyPond works effectively with this approach.

12.5.5.1. Move the Figured Bass

Normally, figured bass parts are below the staves. This example is using the figured-bass feature of LilyPond to do something else, so we're going to move the figured bass so that it's where it needs to be.

1. the `\score{ }` section contains everything in the score; notice that the figured bass has the identifier `\bassFiguresPart`; remove this
2. scroll up to where it says `"bassFiguresPart = ... "` and comment it with `%`
3. scroll up to where it says `"pianoPart = ... "` and enter the `"\new FiguredBass \figBass"` line that you just commented out, on a line between the "right" and "left" staves
4. now erase the part that you commented in (2) (or leave it there)
5. Preview the file, and see that the figures now appear between the piano staves

12.5.5.2. Remove the Word "Piano" at the Start

1. Find the part that begins `"pianoPart = ... "` and erase the `"instrumentName ... "` line (or comment it)
2. Preview the file, and see that it no longer says, "Piano"

12.5.5.3. Make some Elements Transparent

This involves some more advanced tweaking. Explain this sort of command.

Every layout object has a "stencil" property. By default, this is set to whatever function draws the object. If you set that property to `#f`, which means "false", then the drawing function is not called, and the object will not appear on the score.

Every layout object also has a "transparent" property. By default, this is set to `#f` ("false"). Setting it to `#t` ("true") will make the object transparent.

You can use `"\once \override ..."` or `\revert ...` too

1. Find the "global" section
 2. After `\key` and `\time`, put `" \override Score.BarLine #stencil = ##f "`
 3. Preview the file, and see that this doesn't work quite as intended.
 4. It so happens that, while measure lines within a staff are handled by `Staff.BarLine`, measure lines between staves are handled by `Staff.SpanBar`; so you'll need to set its 'transparent' symbol to `#t` also
 5. But there's still no measure-line at the end! You want a barline at the end, so pick one of the staves (right, left, `figBass` - it doesn't matter in this case) and use the `\revert` command (don't know if I should put this in, but: `"\revert Score.BarLine #transparent and \revert Score.SpanBar #transparent"`)
 6. But even this isn't quite right. You want a double-barline at the end. So, put the cursor after the `\revert` lines, and then from the menu, 'LilyPond > Bar Lines > Ending Bar Line'. It knows what you want, remembers the symbol for you, so you don't have to!
- explain in there the difference between a `Staff.*` and `Score.*` override

- Unlike with some other elements, if you simply remove the "time 4/4" indicator, it will still print the default 4/4 time signature.
- This example is musically simple, but it includes some advanced concepts, and importantly helps to get over a couple of common (and understandable) fears, especially for beginners, and especially for musically-simple things like this

12.6. Work on an Orchestral Score (Tutorial)

Scenario: You volunteer at a community orchestra, and the conductor decides to play a Haydn symphony. The orchestra does not own any Haydn symphonies, so the conductor asks you if you can help to find a full score and parts. You find a book with the conductor's score, but no parts. You decide to input the score in LilyPond, which will allow you to easily create the missing parts.

This tutorial uses the first movement of Joseph Haydn's "Sinfonia No. 92: Oxford."

When following this tutorial, it is recommended that the reader follows along, creating their replica of this tutorial's PDF.

12.6.1. Files for the Tutorial

You do not need the **LilyPond** input file to do the tutorial. You do need the PDF output file to do the tutorial. Use the input file if you encounter problems, or to check your result.

- *LilyPond Input File* at http://docs.fedoraproject.org/en-US/Fedora/15/html/Musicians_Guide/files/LilyPond/Orchestra-source.ly
- *PDF Output File* at http://docs.fedoraproject.org/en-US/Fedora/15/html/Musicians_Guide/files/LilyPond/Orchestra-result.pdf

12.6.2. Start the Score

1. 'LilyPond > Setup New Score'
2. "Titles and Headers" Tab:
 - Dedication: Comte d'Ogny and Prince Oettingen-Wallerstein
 - Title: Sinfonia No. 92
 - Subtitle: "Oxford"
 - Composer: Haydn, Joseph
 - all the rest are blank
3. "Parts" Tab:
 - Flute, 2 of Oboe, Bassoon
 - Horn in F, Trumpet in C
 - Timpani
 - Violin, Violin, Viola, Cello, Contrabass
4. "Score settings" Tab:

- Key signature: G Major
 - Time signature: 3/4
 - Tempo indication: Adagio
 - other settings as desired; I recommend not removing bar numbers, and removing the default tagline
5. Click "Try" to see if it works. The instrument names will not be the same as the PDF score. This will be fixed later.
 6. Click "OK" to generate the score.

12.6.3. Adjust Frescobaldi's Output

These steps are useful in establishing a consistent input style for LilyPond. The things suggested here are also useful for getting used to working with large scores, which can be a challenge in any text editor. Thankfully, careful (and consistent!) code organization goes a long way in helping you to quickly find your way around your files. Setting up files the right way to begin with makes this much easier in the end.

When you first setup the score, Frescobaldi will have created many sections for you by default. The program avoids making too many stylistic choices for you, which allows you to create your own style. It also sets up the default sections in a logical way:

```
version
header
widely-used settings like tempoMark and "global"
individual parts
score formatting
```

The specific ordering will become more obvious to you as you get used to LilyPond.

Here are some of the things that I do before inputting notes:

- Use section-dividers (like %%%%%%%%% **NOTES** %%%%%%%%% for individual parts) to demarcate sections.
- Use different levels of dividers (like %%%% **OBOE** %%%% for the oboe) to show sub-sections
- Add blank lines between large sections, to separate them more obviously
- Begin braces on new lines, like

```
\header
{
  title = "Symphony"
  ...
}
```

instead of on the same line, like

```
\header {
  title = "Symphony"
  ...
}
```

This is simply a matter of personal taste, resulting from prior experience with C and C-like programming languages.

- Familiarize myself with the sections and commands created by Frescobaldi, getting a sense of what the section/command does (even if I don't understand what each specific command does). This makes it easier to sort out problems and customization down the road. Sometimes, when the setup is quite complex, I make comments about what seems to be going on.
- Change the instrument names to replicate what my score indicates. You don't necessarily need to do this, so long as the people using your score will understand what the instruments are. Since my goal is to replicate this Dover edition score, I will change the names.
 1. In the "PARTS" section, each xPart section has an "instrumentName = "something"" field.
 2. We'll be changing those to the following Italian names:
 - Flute --> Flauto
 - Oboe I --> I Oboe
 - Oboe II --> II Oboe
 - Bassoon --> 2 Fagotti
 - Horn in F --> 2 Corni in Sol (NB: these would have been "natural horns")
 - Trumpet in C --> 2 Clarini in Do (NB: a clarino is a kind of trumpet)
 - Violin I --> Violino I
 - Violin II --> Violino II
 - Cello --> Violoncello obbligato
 - Contrabass --> Basso
 3. Now we have to change the horns' transposition to match the name. This is because Frescobaldi added a "Horn in F" part, which is the most common horn transposition. However, Haydn's score uses horns in G (or "Sol" in Italian).
 4. Scroll up to the "hornF" section under the NOTES marker. Change the line `\transposition f` to `\transposition g`.
 5. As it turns out, transposition can be a little more complicated than that. We'll deal with that when we get there.

12.6.4. Input Notes

This tutorial offers step-by-step instructions representing one way to input the score. Only the part before "Allegretto" will be inputted. The full first movement is included in the PDF file, so you can input it yourself.

12.6.4.1. Start with the Easy Part

The best way to get started on large scores is to just start with something easy. Nothing can be easier than doing nothing, so let's first input the multi-measure rests in the wind sections and timpani.

1. Count the number of measures of rests. In this case, all of the winds and the timpani have thirteen or more measures of rests before their first entrance.
2. Let's put in those thirteen measures first.

1. In the NOTES section, find the flute section, and put in **R2. *13 |**.
2. The **R** symbol means "full-bar rest" (or "full-measure rest"). LilyPond draws full-bar rests differently than rests that simply take up the length of a whole bar.
3. The **|** symbol is a bar-check symbol. See [Section 12.4.8.2, "The Bar-Check Symbol"](#) for more information.
4. Copy and past that into the rest of the winds and the timpani.
3. The timpani and trumpet/clarini parts have six further full measures of rest, before the measure with a fermata. The fermata can't be included in the multi-measure rest, so we'll treat it separately. Add **R2. *6 |** to the timpani and trumpets parts.
4. The horns, bassoons, and second oboe have one further measure of rests, so add **R2. |** to those parts. Full-measure rests should always have a capital letter R. This tells LilyPond to properly center the rest in the bar.

12.6.4.2. Continue with the Wind and Timpani Parts

1. Now we need to start adding notes. We'll start with the parts that have only rests:
 - a. The trumpets and timpani have no notes in the slow introduction, so all they need is the three-beat measure with a fermata on the third beat.
 - b. Rests are notated as though they were a pitch called "r". That is to say, if you want to input a rest, then you input the pitch r. This three-quarter-rest measure will be notated as **r4 r r |**. Put this into the trumpets and timpani parts. Remember: it's always a good idea to indicate the duration at the beginning of a measure, but it's not necessary to indicate a repeated duration.
 - c. Frescobaldi allows you to insert common musical signs with the "Quick Insert" tab on the left of the screen. It may be hidden; to show it, click "Quick Insert" on the left-most edge of the Frescobaldi window.
 - d. We need a regular fermata on the third quarter rest of this measure. So, place the text-input caret just after the third quarter rest, and click the fermata symbol in the "Quick Insert" toolbar-thing.
 - e. You should end up with **r4 r r\fermata |**.
 - f. While the word for a fermata symbol is easy to remember, other symbols have less-obvious LilyPond notations, so the "Quick Insert" toolbar-thing is very handy.
 - g. Copy this measure to the other part.
2. The next easiest part is the bassoons:
 - a. We've already input the bassoons' full-measure rests, so we can start with the first measure in which they play. It should be notated as **r8 d g bes d bes |**. Input this, and preview the score to see what happens.
 - b. Success! When first starting an instrument, it's important to check that the notes begin in the right register. If they don't, it can be adjusted easily by changing the pitch indication that follows the **\relative** declaration. In this case, the bassoon happened to be in the right register for us by default.

- c. The next measure is **g8 cis cis4 r** |. Remember to indicate "cis" twice. Put it in and check that it's correct.
- d. It is, but we're still missing some formatting. Use the "Quick Insert" toolbar-thing to add staccato markings to the first measure. You can add a staccato to the eighth-rest, but this doesn't make sense, so you shouldn't.
- e. Slurs begin at (and end at). Add a slur from the g to c-sharp.
- f. Preview the score to make sure that you entered these articulations correctly. Your code should be:

```
r8 d-. g-. bes-. d-. bes-. |
g8( cis) cis4 r |
```

3. Now to add the "forte" marking. You can add text (or any object, for that matter) onto a note (or rest, etc.) with one of these three symbols:

- ^ meaning "put this above the object"
- - meaning "put this above or below, as you think is best"
- _ meaning "put this below the object"

As you saw earlier, Frescobaldi attached our staccato markings with the "as you think best" symbol, which is almost always the right choice for articulations. By convention, dynamic markings always go below the staff to which they apply, so we won't want to give LilyPond so much freedom, this time.

4. The easiest way to add (unformatted) text to a note is to simply attach it in quotation marks. For the "forte" marking, put this on the eighth-rest: **r8_"f" and so on**
5. When you preview this, you'll notice that the result is thoroughly underwhelming. It looks quite unlike a "forte" marking, and people reading the score would probably be confused, if just momentarily.
6. Thankfully, LilyPond provides a very easy and elegant way to input well-formatted dynamic markings. Change the eighth-rest to this: **r8\f and so on**
7. When you preview this, you will see that it now looks exactly like a typical "forte" marking. Not all dynamic markings have these short-forms, but most do.
8. The "a 2" marking, meaning "to be played by two players," does need the text-in-quotes format, however. Put that marking *above* the d following the eighth rest.
9. Those two measures should now look like this:

```
r8\f d-.^"a 2" g-. bes-. d-. bes-. |
g8( cis) cis4 r |
```

Note that **d- . ^"a 2"** gives the same result as **d^"a 2" - .**

12.6.4.3. Oboe and Horn Parts

1. You can complete the oboe parts and the flute part. If you get stuck, read these tips.

- You will need to adjust the range of the flute and oboe, to read **flute = \relative c'''** and **oboeI = \relative c'''**
 - You may want to use [and] to control eighth-note beaming in the Oboe I and Flauto parts. You may not
 - The Flauto will need more multi-measure rests with R, *after* some of the notes are inputted.
 - All of the parts will end with the same three-quarter-note-rests-with-fermata measure.
2. And now for the horns part. Transposing instruments pose a small problem for LilyPond, as with any human or computerized engraving tool. These steps first ensure that the transposition is set correctly.
 3. The "hornF" section should already have a **\transposition g** segment from earlier. This tells LilyPond that the following notes are not in concert pitch, but rather are "in G." A transposition statement tells LilyPond, in absolute pitch, which pitch actually sounds when the player plays a written c' . In this case, the sound pitch is a perfect fourth below c' . If we wanted it to be a perfect fifth higher, then we would need to write **\transposition g'**, but that's not accurate for this case.
 4. Our next obstacle is not actually a problem with LilyPond, but with how Frescobaldi set up the score for us. The score that we wish to notate does not have a written key signature for the horn, but the "global" section (near the top of the file) includes one: G Major. If this score were particularly complicated, or if it contained a large number of transposing instruments, then it would be best to remove the **\key g** declaration from the "global" section (and including it in every instrument as necessary). However, since there is only one transposing instrument, we might be better off simply removing the global from the horn.
 5. do the rest of this stuff to get that done right
 6. make a note in the global section of which instruments don't use it
 7. While you're at it, do the same for the trumpets and timpani parts, which also do not use a printed key signature.
 8. The other issue with the horn part is that two pitches are to be played at once, and they are both notated in the same voice. This is solved in the piano example like this: **<g g'>**. You can copy-and-paste this as needed.
 9. You can now finish inputting the horn part.

12.6.4.4. Continue with the Strings

After correctly finishing all of the wind and timpani parts, you can move on to the strings.

- Input all of the pitches and note lengths first, then return to fill in the other markings.
- If you get stuck, then put in some filler notes, and finish the rest of that part. Make a note for yourself, as a comment in the source file, so that you know to return later to finish the part.
- If twenty measures at once is too many, then break it into smaller chunks and input those.
- Here are some tips to help you through this passage. Use the LilyPond help files if you need further assistance.
 - Tuplets (Triplets):
 - To write any tuplet, use the formula **\times x/y { notes in here }**, where x is the number of notes actually space to use, and y is the number of notes to let you display.

- For an eighth-note triplet that takes up one beat, you might use this: `\times 2/3 { c8 d e }`, because in the space of 2 eighth notes you want to fit three instead.
- This is much more powerful than it seems at first. You might want to make it seem as though the measures have stopped being marked, and write something like `\times 4/40 { ... }`, which will allow you to fit forty of something into the number of beat usually occupied by four of them. This is especially useful with cadenzas, and cadenza-like passages.
- Short-term Polyphonic Input (Divisi):
 - Anything between `<<` and `>>` is interpreted by LilyPond as happening together. If you take a look at the "score" section at the bottom of the file, you will see that all of the parts are listed in that kind of bracket. This ensures that they all happen simultaneously.
 - For short-term polyphonic input, use the formula `<< { upper-voice notes } \\
{ lower-voice notes } >>`. Remember that the "upper voice" has upward-pointing stems, and the "lower voice" has downward-pointing stems.
- Ties: These can be written by adding `~` to the end of the note beginning the tie: `c4~ c8`
- Grace Notes: These take up no logical time, and are smaller than ordinary notes. Any notes appearing `\grace { in here }` would be considered grace notes.
- Crescendo and Diminuendo Markings:
 - These are added like other dynamic markings, attached with a backslash to the note where they begin. A crescendo is triggered with `\<` and a diminuendo with `\>`.
 - The left-most point of the marking (its beginning) is indicated by where you put `\<` or `\>`
 - The right-most (or end-)point is indicated by either another dynamic marking of any sort, or the special cancellation character `\!`
- Italic "staccato" Text: `\markup { \italic { staccato } }`
- Source File Measure Numbers: One of the techniques that can be used when inputting larger scores is writing measure numbers into the source file as comments. I usually write measure numbers every five measures, but it depends on the speed of the music, and what seems like it's useful.

12.6.4.5. Continue to the Fast Section

If you want to finish inputting the first movement, as an exercise, then you will also need to know how to write a tempo-change in an orchestral score:

1. In order to continue, we'll need to include a special measure-line (barline) and a change-of-tempo indicator. This would be easy, and would display correctly, if we simply inputted the change in one particular voice. However, if we did that, the change would only appear in one of the orchestral parts exported from the score.
2. We're going to define a new whatever-thing, like the "global" section created for us by Frescobaldi, and include it in all of the parts.
3. The code needed for this is a little bit complicated, but you don't need to write it yourself: just take it from the "tempoMark" section created by Frescobaldi. All you need to do is change "Adagio" to "Allegro spiritoso," and add the barline indicator. This is also easy because of Frescobaldi: 'LilyPond > Bar Lines > Repeat start'

4. You end up with

```
startExposition =
{
  \once \override Score.RehearsalMark #'self-alignment-X = #LEFT
  \once \override Score.RehearsalMark #'break-align-symbols = #(time-signature key-signature)
  \mark \markup \bold "Allegro spiritoso"
  \bar "|:"
}
```

5. Add the reference to this in all of your parts. Because of how I named it, this also serves as a handy way to find your way through the LilyPond markup file.

```
r4 r rfermata |

\startExposition

R2.*4 |
```

6. The barline and tempo-change will not appear unless you write some music after them, so put in some or all of the rests that follow, just to test it.

12.7. Work on a Piano Score (Tutorial)

Scenario: At a used music store, you encounter a 60-year-old, hand-engraved copy of a Schubert "Impromptu," but it's been badly damaged by a flood. The store's owner says that, if you can use the score, you can keep it for free. The score is barely legible, so you decide to prepare a copy with a computer notation program.

I'm using Schubert's "Impromptu" Op.90 (D.899) Nr.4, in A-flat major. Published by Edition Peters: 10463, edited by Walter Niemann. We'll be setting the "A" section (from the beginning to the "Trio").

When following this tutorial, it is recommended that the reader follows along, creating their replica of this tutorial's PDF.

12.7.1. Files for the Tutorial

You do not need the **LilyPond** input file to do the tutorial. You should use the input file if you encounter problems, and to compare your completion. You do need the PDF output file to do the tutorial.

- **LilyPond** Input File at http://docs.fedoraproject.org/en-US/Fedora/15/html/Musicians_Guide/files/LilyPond/Piano-source.ly
- **PDF** Output File at http://docs.fedoraproject.org/en-US/Fedora/15/html/Musicians_Guide/files/LilyPond/Piano-result.pdf.

12.7.2. Start the Score

1. Open Frescobaldi with a new document.
2. Start the "Setup New Score" tool, by clicking '**LilyPond** > Setup New Score' in the menu.
3. Fill in the following fields on the "Titles and Headers" tab:
 - Title: Impromptu
 - Composer: Schubert, Franz

- Opus: Opus 90/4
4. Switch to the "Parts" tab:
 - a. From the "Available parts" list, select "Keyboard instruments"
 - b. Select "Piano"
 - c. Click "Add"
 - d. On the right-hand side of the window, it's possible to add multiple voices from the start. We won't be using this feature, because most of the score is not polyphonic. It is especially convenient in fugues.
 5. Switch to the "Score settings" tab, and adjust the following settings:
 - Key signature: As Major
 - Time signature: 3/4
 - Tempo indication: Allegretto
 - Remove default tagline: checked
 - Adjust other settings as desired - it is recommended to keep the default pitch name language.

12.7.3. Adjust Frescobaldi's Output

These steps are useful in establishing a consistent input style for **LilyPond**. The things suggested here are also useful for getting used to working with large scores, which can be a challenge in any text editor. Thankfully, careful (and consistent!) code organization goes a long way in helping you to quickly find your way around your files. Setting up files the right way to begin with makes this much easier in the end.

When you first setup the score, Frescobaldi will have created many sections for you by default. The program avoids making too many stylistic choices for you, which allows you to create your own style. It also sets up the default sections in a logical way:

1. version
2. header
3. widely-used settings like tempoMark and "global"
4. individual parts
5. score formatting

The specific ordering will become more obvious to you as you get used to **LilyPond**.

Here are some of the things that I do before inputting notes:

- Use section-dividers (like %%%%%%%%% **NOTES** %%%%%%%%% for individual parts) to demarcate sections.
- Use different levels of dividers (like %%%% **RIGHT HAND** %%%% for the right hand) to show sub-sections
- Add blank lines between large sections, to separate them more obviously
- Begin braces on new lines (as


```
\header
{
  title = "Impromptu"
  ...
}
```

instead of on the same line (as

```
\header {
  title = "Impromptu"
  ...
}
```

This is simply a matter of personal taste, resulting from prior experience with C and C-like programming languages.

- Familiarize myself with the sections and commands created by Frescobaldi, getting a sense of what the section/command does (even if I don't understand what each specific command does). This makes it easier to sort out problems and customization down the road. Sometimes, when the setup is quite complex, I make comments about what seems to be going on.
- At this point, I also added the "Dynamics Context" commands, as described below in the "Piano Dynamics" section

12.7.4. Input Notes

Piano scores present some unique challenges with **LilyPond**, but they're easy to overcome with some careful thought. This tutorial will avoid step-by-step instructions on how to input particular notes, instead focussing on those unique piano challenges presented in this particular composition. The **LilyPond** "Notation Reference" provides a section dedicated to keyboard and piano music. Most of the situations described there are not present or discussed in this score, which gives this tutorial unique material.

12.7.4.1. Order of Input

Choosing the right order to input your scores can make things much easier to troubleshoot. Here are some suggestions:

1. Input music in small sections at a time. Two, four, or eight measures is usually a good size, but it depends on the size of the composition, the size of its sections, and the form of the music. It doesn't make sense to input a passage of 9 whole-notes in stages of two measures, but two measures may be too long for passages composed primarily of 128th-notes.
2. Input one staff at a time, then check its accuracy using Frescobaldi's preview function (press the "LilyPond" button on the toolbar)
 - Input the pitch and rhythms first, then slurs, articulations, ties, and fingerings. It's easier to correct errors with pitch and rhythm (and register!) before the extra markings are added.
 - To help make sixteenth-note passages easier to read, you can use double- or triple-spaces to separate beats. Such passages often feature repeated patterns; before copy-and-pasting, make sure that the pattern repetition is truly exact!

3. When you progress to a new four-measure section (or two- or six-, etc.), input the less difficult staff first (if there is one). This way, you will have a better visual reference when verifying the more difficult part. It's easier to see differences if your score looks closer to the original.
4. The same idea applies for multi-measure polyphonic passages in the same staff: input the easier part first, so you have a visual reference.
5. To help ensure that you don't get lost, write measure numbers in the source file every five measures or so. See the example source file to see how this is done. Even if you don't want to have measure numbers in the final score, it can be helpful to include them during inputting and error-checking.
6. Save the dynamics and pedal markings for last! Sometimes, they can help you to keep your place in the score while double-checking that it's correct, but I don't usually put them in with the rest of the notes, for reasons described below in the "Piano Dynamics" section.

Most importantly, remember that these are just suggestions! The order in which you do things should change depending on what suits you best. Different kinds of scores will require different strategies.

12.7.4.2. Chords

There are two ways to input chords, but one will be used much more often.

This style of chord notation is more common: `<as ces>4- _` Notice how only the pitches are notated inside the `< >` brackets, and everything else attached to the end. There is one exception to this: fingering should be indicated on the pitch associated with that finger: `<as-1 ces-3>4- _` Not only does this help you to sort out what was probably intended, but it allows **LilyPond** to stack the fingering in the right order. When using "relative" entry mode, it is the lowest note of the chord that is taken into consideration when placing the next note.

This style of chord notation is less common: `<<as4-> ces>>` Notice how everything must be notated inside the `<< >>` brackets. This can make it more difficult to read the chord in the source file, but it also allows much greater flexibility: only some chord members can have ties; certain chord members can last for longer than others; certain chord members can "break out" into or out of polyphonic passages. This notation is rarely *needed*, but you may be inclined to over-use it if you are trying to exactly copy the look of a hand-engraved score. Like the "times" command for tuplets, this is one of **LilyPond**'s deceptively powerful techniques. When using "relative" entry mode, it is the last note of the chord that is taken into consideration when placing the next note.

12.7.4.3. Fingering

LilyPond allows you to indicate fingering by attaching the digit number to a note as an articulation mark: `a16-5` will show a "5" as a fingering mark. As with all other articulation marks indicated in this way, you can use `^` or `_` to instruct **LilyPond** to put the mark above or below the note, respectively. It is usually better to let **LilyPond** decide for itself by using a `-`.

When entering chords, it is recommended that you enter the fingering with the note to which it is attached, like `<as-1 ces-4>4- _`. It is possible to enter this as `<as ces>4-1-4->`, but this not only looks confusing, it may confuse **LilyPond** as to which digit is intended for which note.

Because the extra digits look like they indicate note-lengths, it is recommended to mark them consistently. For this same reason, it is also recommended that fingering marks be added to source files only after the pitch and rhythm have been double-checked. The source file included with this tutorial puts fingering marks after any other articulation and length marking.

12.7.4.4. Cautionary Accidentals

Musical Definition: This score makes some use of *cautionary accidentals*. These are accidentals which don't change the pitches to be played, but rather are used as a precaution against forgetting that they are there. This usually happens when an accidental in the written key signature is changed for a significant number of measures, and then suddenly changes back. The cautionary accidental would be applied when the standard key signature returns, to remind the musician of the key signature.

LilyPond Notation: These are notated in **LilyPond** with an exclamation mark placed before the note-value: **ces!16**

12.7.4.5. Change the Style of Crescendo

Sometimes the composer or editor prefers to use the words *crescendo* or its abbreviation, *cresc.*, instead of the `\<` style of crescendo. In **LilyPond** these are handled by the same source-file notation (`\<` to start and `\!` to end explicitly). However, if you want to use text and a "spanner" (dotted or dashed line, for example) instead of the `<` sign, you need to tell **LilyPond**. This can be accomplished with the following command: `\crescTextCresc.`

After changing to this style of crescendo, you can revert to the standard `<` style with the following command: `\crescHairpin`.

The `\dimTextDim` and `\dimHairpin` commands do the same for a diminuendo.

12.7.4.6. Polyphonic Sections of Homophonic Music

Sometimes, especially in piano music, a passage of some measures will require polyphonic (multi-Voice) notation in **LilyPond**, even though most of the music does not. In this case, you would use the following format: `<< { % upper voice notes go here } \ { % lower voice notes go here } >>` This is used a few times in both hands in the example score file.

When writing these sections in "relative" entry mode, it is a good idea to use the "octave-check" mechanism, at least at the beginning of the lower voice. This is because, when judging the relative starting pitch of the first note of the lower voice, **LilyPond** judges from the last note of the upper voice - *not* the last note before the polyphonic section began.

12.7.4.7. Octave-Change Spanners ("8ve" Signs, "Ottava Brackets")

Musical Definition: In order to most conveniently notate over large ranges, composers and editors sometimes use text spanners to indicate that a passage should be played one or two octaves higher or lower than written. This allows the notes to stay mostly within the staff lines, thereby decreasing the number of ledger lines required.

LilyPond Notation: The `\ottava` command is used to notate all five states of transposition:

- 15va (play two octaves higher than written) is engaged with `\ottava #2`
- 8va (play one octave higher than written) is engaged with `\ottava #1`
- loco (play in the octave written) is engaged with `\ottava #0`, which also cancels another sign.
- 8vb (play one octave lower than written) is engaged with `\ottava #-1`
- 15vb (play two octaves lower than written) is engaged with `\ottava #-2`

12.7.5. Troubleshoot Errors

It can be difficult to troubleshoot inputting errors, especially when you find them days or weeks after originally inputting that section of a score. The best way to fix errors is to input scores in a way that

doesn't allow them in the first place. As they say, "an ounce of prevention is worth a pound of cure," which means "if you input your **LilyPond** files carefully, then you will encounter fewer problems." Such practices as proper spacing, and regular use of the octave- and bar-check features will deal with many common problems.

However, when searching for an error in your score, Frescobaldi does offer some features to help you find it:

- If you have written something in the text-input window that is obviously incorrect, then Frescobaldi will sometimes catch your mistake and underline and highlight the erroneous code in red. This is most useful for catching typographical errors ("typos").
- If **LilyPond** encounters a processing error, it will show up in the "**LilyPond** Log" window below the text-input window. You can click on the blue, underlined text to move the text-selection caret to the error.
- If you can see an error in the PDF preview, but you can't find it in the source file, you can click on the problematic note or object in the PDF preview, and Frescobaldi will automatically move the text-selection caret to that object.

12.7.6. Format the Score (Piano Dynamics)

Keyboard instruments use a unique notation when it comes to dynamics. Most instruments use only one staff per player, so the dynamics are, by convention, notated underneath that staff. Keyboard instruments usually use two staves (organs and complex piano music may use three). Because the dynamics are usually meant to apply to both staves, they are usually notated between the two staves. This is similar to notation beneath the upper staff, but in truth, piano dynamics tend to be placed in the middle between the staves - entering the dynamics as belonging to the upper staff, in **LilyPond**, will not produce that effect.

There is a way to notate dynamics between two staves in **LilyPond**, and it involves a little bit of thought to get it right. It also requires the addition of a significant number of commands, and the creation of a new context.

This process looks difficult, and may seem daunting. It's not necessary to understand all of the commands in the "PianoDynamics" Context in order to use the context, so there is no need to worry!

12.7.6.1. Prepare the "PianoDynamics" Context

It is probably easier to add these commands before inputting most of the score, but there is no reason why this context cannot be added to any score at any time.

Follow these steps to create a "PianoDynamics" Context:

1. Between the left and right staves of the PianoStaff, add "**\new PianoDynamics = \dynamics \dynamics**". For the Schubert score, this looks like:

```
\new PianoStaff \with
{
  instrumentName = "Piano"
}
<<
\new Staff = "right" \right
\new PianoDynamics = "dynamics" \dynamics
\new Staff = "left" { \clef bass \left }
>>
```

2. To the layout section, add the following:

```
% Everything below here is for the piano dynamics.
% Define "PianoDynamics" context.
\context
{
  \type "Engraver_group"
  \name PianoDynamics
  \alias Voice
  \consists "Output_property_engraver"
  \consists "Script_engraver"
  \consists "New_dynamic_engraver"
  \consists "Dynamic_align_engraver"
  \consists "Text_engraver"
  \consists "Skip_event_swallow_translator"
  \consists "Axis_group_engraver"

  \override DynamicLineSpanner #'Y-offset = #0
  \override TextScript #'font-size = #2
  \override TextScript #'font-shape = #'italic
  \override VerticalAxisGroup #'minimum-Y-extent = #'(-1 . 1)
}

% Modify "PianoStaff" context to accept Dynamics context.
\context
{
  \PianoStaff
  \accepts PianoDynamics
}
% End of PianoDynamics code.
```

This creates a "PianoDynamics" context, and modifies the "PianoStaff" context so that it will accept a "PianoDynamics" context.

- Before the "\score" section, add a section called "dynamics," like this:

```
dynamics =
{
  % Dynamics go here.
}
```

This is where you will input the dynamics.

12.7.6.2. Input Dynamics

Now you can input the dynamic markings. These are inputted with a special note called a "spacer," that uses the letter "s" rather than a note name. You can also use rests (both partial- and multi-measure, *r* and *R*), but dynamic markings cannot be assigned to them.

For example, if you want a piano marking on the first beat, and a diminuendo from the third to the fourth beat, you could write this: **s4\p r s\> s\!**, or this: **s2\p s4\> s\!**

That's all there is to it! Think of the dynamics part as an invisible, pitch-free line between the two staves, for the sole purpose of dynamics (and other expression markings).

Frescobaldi

Frescobaldi is an advanced text editor, designed specifically for use with LilyPond source files. Its interface has been crafted in such a way that it aids the average workflow of creating and editing musical scores in LilyPond. Frescobaldi's tight integration with various system tools is similar to the way LilyPond itself is tightly integrated with various other software programs.

Frescobaldi is designed to take advantage of several features of the KDE 4 desktop system. Regular KDE users will immediately recognize Frescobaldi's components as being identical to several other key KDE applications - specifically Kate, Okular, and Konqueror. The key advantage to this approach is that KDE users will already know how to use most of the features of Frescobaldi, because those other applications are not just replicated, but actually used by Frescobaldi. There are many other advantages to this development approach.

For non-KDE users - especially those with older computers, this KDE-based approach may be more of a hassle than a help. An unfortunate side-effect of this development choice is that Frescobaldi has many "dependencies" in KDE packages, meaning that some KDE software will need to be installed in order to use Frescobaldi. This is a significant disadvantage for some users, but the benefits of using Frescobaldi far outweigh the extra hard drive space that will be required.

13.1. Frescobaldi Makes LilyPond Easier

Using LilyPond with Frescobaldi is the recommended method for this User Guide. All three case studies in the "LilyPond" chapter (link? maybe?) use Frescobaldi. This is because the writers feel that Frescobaldi truly makes using LilyPond an easier thing to do.

Here are some of the benefits:

- The text-editing component (from Kate), PDF-preview component (from Okular), and help file component (from Konqueror) are from applications that you may already know how to use.
- The one-window setup of the application allows you to view either the source file and its output, or the source file and a help document, side-by-side. This is perfect for today's widescreen monitors.
- The "Setup new score" tool greatly increases the speed of creating a new score, and offers some advanced configuration options that might otherwise take hours to discover.
- The "Quick Insert" sidebar on the left-hand side of the window makes it easy to insert articulation, trill, and other marks. These would often otherwise require checking a reference document.
- The 'LilyPond' menu provides several further tools to help adjust and troubleshoot your scores.
- Careful integration with system tools allows you to input notes with a MIDI keyboard, listen to the MIDI output of your LilyPond score, and print the PDF file of your score, all from within Frescobaldi. These features are not covered in this Guide.

For more information refer to the *Frescobaldi Website* at <http://www.frescobaldi.org/>.

13.2. Requirements and Installation

1. Install the *frescobaldi* package with **PackageKit** or **KPackageKit**.
2. There are a lot of dependencies, including *perl*-* packages, *kde*-* packages, and **timidity++** including *fluid-soundfont-gm* (which is 114 MB)
3. Review and approve the list of dependencies.

4. The application can be found in the K Menu under 'Multimedia' or somewhere in the GNOME 'Applications' menu (probably under 'Multimedia'... for some reason)

13.3. Configuration

Changing these default settings is not necessary, but it may result in a more productive experience.

1. Chose **Settings** → **Configure Frescobaldi**
2. In 'General Preferences' check 'Save document when LilyPond is run', or else you will have to save it manually
3. Review the other options on this page
4. In 'Paths', only some of the fields will be filled; this is okay.
 - GNOME users might want to set "PDF Viewer" to 'evince'; this will not change the embedded PDF viewer, but it will change what opens when you click "Open PDF" in the bottom pane's 'LilyPond Log'. If left blank, this will open KDE's "Okular".
 - You can fill in the "MIDI Player" path, so that the "Play MIDI" button will work
 - The program "pmidi" is available from the Planet CCRMA at Home repositories, and is a command-line MIDI player; install it with yum; run 'pmidi -l' to see which ports are available; then put "pmidi -p 14:0" in the "MIDI Player" field where "14:0" is instead whatever port you want to use; you'll need a MIDI synthesizer (like FluidSynth) listening
 - or you can use timidity++
 - You can change the "Default Directory" to the place where you store your LilyPond files
 - You can change "LilyPond documentation" if you don't want the right-side-bar "LilyPond Documentation" thing to internet it every time
 - install the 'lilypond-doc' package (it's 44 MB)
 - put "/usr/share/doc/lilypond-doc-2.12.2/index.html" in the field (or in another language if that's what you want)
 - this will need to be updated when LilyPond is updated; if you don't update it, then the built-in "LilyPond Documentation" tab will simply show up blank, instead of with the user guide - so it's harmless if you forget to update
5. options for the "Editor Component" are the same as those for KWrite
 - I recommend going to the Editing tab, then "Auto Completion" and ensuring that both boxes are un-checked
 - You may want to change other settings here; highly-flexible, customizable, powerful; applies only in Frescobaldi

13.4. Using Frescobaldi

The practical use of Frescobaldi for editing LilyPond source files is described in [Chapter 12, LilyPond](#).

GNU Solfege

14.1. Requirements and Installation

14.1.1. Hardware and Software Requirements

It is assumed that, prior to using GNU Solfege, users have already correctly configured their audio equipment.

In addition, the **timidity++** package is required by Solfege, which requires the installation of a large (approximately 140 MB) SoundFont library. This library is shared with the **FluidSynth** application, which has its own section in this guide, and is used by several other software packages. **timidity++** also requires the installation of the JACK Audio Connection Kit. If you have installed the Planet CCRMA at Home repository, and have not yet followed the instructions to correctly install and configure its version of JACK, then it is recommended that you do so before installing GNU Solfege. Refer to [Section 2.3.1, "Installing and Configuring JACK"](#) for instructions to install JACK.

14.1.2. Other Requirements

Solfege requires knowledge of Western musical notation, and basic music theory terms and concepts.

14.1.3. Required Installation

Please review the "Requirements" section above, before installation.

Use PackageKit, KPackageKit to install the *solfege* package.

14.1.4. Optional Installation: Csound

Csound is a sound-synthesis program, similar to SuperCollider. It is older, quite well-developed, and has a broader range of features.

To install Csound, first close Solfege, then use PackageKit or KPackageKit to install the *csound* package. When Solfege is restarted, you will be able to use the "Intonation" exercises.

14.1.5. Optional Installation: MMA

MMA stands for "Musical MIDI Accompaniment," and it is not available for Fedora in a prepackaged format. The software can be found on the *MMA Homepage* at <http://www.mellowood.ca/mma/>, where you can download the source code and compile it if desired. MMA is only used by some of the harmonic dictation questions, so its installation is not required.

14.2. Configuration

It is not necessary to configure Solfege. The simple first-time configuration will help to customize the application for your needs, but it is not required. The other configuration options are explained for your reference.

These configuration options are not required, but you may wish to change them. They are all available from the "Preferences" window, accessible through Solfege's main menu: 'File > Preferences'.

14.2.1. When You Run Solfege for the First Time

These steps allow Solfege to automatically customize some questions for you.

1. Start Solfege.
2. From the menu, select 'File > Preferences'.
3. Select the "User" tab.
4. Input your vocal range, if you know it. Solfege uses this information to assign questions in a comfortable pitch range.
5. Input your biological gender in the "Sex" field. Solfege uses this information to assign questions in the correct octave.

14.2.2. Instruments

- **Tempo:** Changes the speed at which examples are played.
 - **Default:** Applies to everything that isn't an "Arpeggio."
 - **Arpeggio:** Applies to arpeggios, which are, by default, played at three times the speed of the default tempo. This is standard practice.
- **Preferred Instrument:** This MIDI instrument will be used when playing examples.
- **Chord Instruments:** For polyphonic voices, these MIDI instruments will be used for the highest, middle, and lowest voices above the bass voice (which takes the "preferred instrument.")
- **Percussion instruments:**
 - **Count in:** This MIDI instrument will be used to give "count in" beats that precede musical examples involving rhythm.
 - **Rhythm:** This MIDI instrument will be used to dictate rhythm-only examples.

14.2.3. External Programs

Solfege uses external programs to perform many tasks. On this tab, you can provide the command to be used, as though being run from a terminal.

Converters:

- MIDI to WAV
- WAV to MP3
- WAV to OGG
- If you use %(in)s and %(out)s, Solfege will substitute the filename it wants inputted and outputted.

Audio File Players, each with "Test" buttons:

- WAV
- OGG
- MP3
- MIDI
- Solfege will substitute %s with the name of the file to be played.

Miscellaneous:

- CSound: Solfege uses CSound for intonation exercises. It is an optional component. See the "Optional Installation" section above.
- MMA: Solfege uses MMA for certain harmonic dictation exercises. It is an optional component, and not available in Fedora through standard means. See [Section 14.1.5, "Optional Installation: MMA"](#)
- Lilypond-book: Solfege uses this for generating print-outs of ear training exercise progress. See the "Optional Installation" section above.
- Latex: Solfege uses this for generating *.dvi format progress reports, rather than the default HTML format.
- Latex: Solfege uses this for generating *.dvi format progress reports, rather than the default HTML format.

14.2.4. Interface

- Resizeable main window: This allows users to resize the main Solfege window.
- Select language
- Identify tone keyboard accelerators: In exercises requiring the input of particular pitches, this will allow a conventional keyboard to be used as a MIDI keyboard - letters on the keyboard will be associated with particular pitches.

14.2.5. Practise

These options apply to exercises done in "Practise" mode.

- "Not allow new question before the old is solved": Solfege will not provide a new question until you have solved the current question.
- "Repeat question if the answer was wrong": Solfege will repeat a question if you provide an incorrect answer.
- "Expert mode": Solfege will provide advanced configuration options for individual exercises.

14.2.6. Sound Setup

- "No sound": This mode is used for testing and debugging Solfege.
- "Use device": This mode lets you specify which audio interface Solfege will use.
- "User external MIDI player": This mode lets you choose a MIDI synthesizer.
- The button, "Test": allows you to ensure that you've correctly configured Solfege.

14.3. Training Yourself

There are three kinds of exercises available in Solfege:

- "Listen-and-identify" exercises will play some sort of musical structure, and ask you to identify, classify, or label it according to widely-used conventions.
- "Identify" exercises will show you some sort of musical structure (usually in Western classical notation), and ask you to identify, classify, or label it according to widely-used conventions.

- "Sing" exercises will provide a specific musical structure, sometimes partially-completed, and ask you to sing the completion of the structure.

Unlike many commercially-available aural skills computer applications, Solfège often relies on the user to know whether they performed the task correctly - and how correctly. This is especially true of the "Sing"-type exercises, since Solfège lacks the capability to receive sung input from the user. This requires at least two things to be kept in mind: firstly, that it is to your own benefit to honestly tell Solfège when you correctly and incorrectly provide a solution, since this helps Solfège to focus on your weaknesses, and to more accurately track your progress; secondly, there are many degrees of correctness when it comes to music, and harmonic and melodic dictation are particularly prone to these forms of partial correctness.

When you encounter a rough spot with your aural skills development, remember that it takes a significant amount of time and effort to build your musical sensibility. It is easier for some people than for others, and most people will have an easier time with some exercises than with others. Although the prevailing cultural thought about musical sensibility (and aural skills) still suggests that an individual either possesses musical ability or cannot acquire it, recent research has suggested that any hearing person with enough determination, dedication, and the right instruction can develop their musical sensibility and aural skills to a very high level.

With that in mind, the following sections aim to help you incorporate Solfège as part of a complete aural skills development program.

14.3.1. Aural Skills and Musical Sensibility

When somebody decides to receive musical training, what they are really doing is developing skills and acquiring stylistic knowledge required for participation in a particular kind of music. There are many different kinds of training, and the time spent in a classroom is not as important to musical development as time spent elsewhere, taking part in real, musical situations. Many different kinds of skills are useful for musicians, depending on the kind of music in which they intend to participate. A folk singer who plays guitar might wish to memorize the chord progressions, melodies, and words for thousands of different songs. An oboe player in an orchestra might wish to make their own reeds from cane tree bark. Most musicians need to be able to listen to music and perceive certain structures that other musicians use to describe their work. These structures are explained by "music theory," and the skill set used to hear these things in music is called "aural skills." Musicians train their aural skills by a set of procedures known as "ear training," or "aural skills training."

Musical sensibility is developed when aural skills are used to help a musician gain an understanding of how and why their music (and other people's music) works and sounds the way it does. This understanding is key to having a sense of the procedures and conventions that an audience will expect of a performer, and therefore to the performer's ability to produce aesthetically pleasing music.

Having a well-developed musical sensibility and set of aural skills are both important aspects of being a musician, but they are by no means the only aspects. More than anything, a musician (or an aspiring musician) should be talking and listening to other musicians.

14.3.2. Exercise Types

Solfège's exercises are arranged in six broad categories:

- Intervals, which tests your ability to perceive and identify melodic and harmonic intervals.
- Rhythm, which tests your ability to perceive and notate rhythms played with a single pitch, and to perform such a rhythm by means of tapping on the keyboard.
- Theory, which tests your ability to name written intervals and scales, and to correctly use solfa syllables.

- Chords, which tests your ability to perceive and identify chords built from various intervals, played at various pitch levels, and to sing such a chord.
- Scales, which tests your ability to perceive and identify scales and their modes.
- Miscellaneous, which includes a variety of exercises, including:
 - Intonation, which tests your ability to perceive and identify whether a second pitch is lower or higher than it should be.
 - Dictation, which tests your ability to perceive and notate melodies.
 - Identify Tone, which tests your ability to use relative pitch (see *Relative Pitch (Wikipedia)* at http://en.wikipedia.org/wiki/Relative_pitch for more information).
 - Sing Twelve Random Tones, which tests many skills.
 - Beats per Minute, which tests your ability to determine a tempo.
 - Harmonic Progressions, which tests your ability to perceive and apply Roman numeral chord symbols for a series of chords played together.
 - Hear Tones, which helps you to train for relative pitch.
 - Cadences, which tests your ability to perceive and apply Roman numeral chord symbols for two chords played together.

All of these exercises require the use of your aural skills outside an actual musical situation. This may seem fruitless, but it has long been recognized as an important part of the eventual ability to hear them within a musical context. Neither ability will suddenly appear; it will take dedicated practice.

14.3.3. Making an Aural Skills Program

Aural skills training - like eating - requires a regular, daily commitment of various kinds of input. As far as food is concerned, you should eat at least three meals a day, with a large portion of fruits and vegetables, and a good balance of meats and alternatives, grains, and other kinds of foods. Ear training also requires diverse inputs at various times throughout the day.

There is no solution that will work for everybody. You will need to choose and modify the time of day, number and length of sessions, and content to suit your needs. The following suggestion can be considered a starting point.

- Training for 30 minutes daily, in three 10-minute segments.
- Segment 1
 - When you wake up, before breakfast.
 - Warm up your voice and body (good activity anyway - *not* part of the ten minutes!)
 - Sing and perfect some excerpts from a book designed for sight-singing.
- Segment 2
 - After lunch, before getting back to work.
 - Listen to some music, then try to transcribe it.
 - To make the task more manageable, take a small portion of music, and focus on one aspect: melody, harmony, or rhythm.

- You can test correctness either by using a piano (or fixed pitch reference), or comparing with a published score, if available.
- Segment 3
 - Some time after supper.
- Use GNU Solfege to test your ability to perceive musical rudiments.
- Spend only a few minutes on a few different kinds of exercises.

Three ten-minute segments is not a lot of time, and indeed it may take additional time to plan and to find and set up materials. Even so, the point is that training your aural skills does not have to take an inordinate amount of time or effort. What's important is that your effort is consistent and well-planned.

14.3.4. Supplementary References

GNU Solfege offers a relatively wide variety of exercises, but no one source can possibly offer all of the exercises and training required to develop a well-rounded set of aural skills. Some of the following books and activities should be used to supplement the exercises available in "Solfege." Note that melodic and harmonic dictation are not yet "Solfege's" strong points, mostly due to a lack of different exercises. This may improve in the future, as the developers improve the software.

Activities:

- Taking dictation from real music. Try focussing on one element, rather than writing a complete score. Verify your solution by checking a published score if possible. Remember that some orchestral instruments may not be written at concert pitch.
- Using your sight-singing skills to sing a melody or rhythm to a friend, who can use it as an example for dictation.
- Use Bach chorales as sight-singing examples for a group of four or more people. It helps to have friends in all vocal ranges, but it isn't necessary. Remember to use solfa syllables, and avoid the chorale's text.
- If a mechanical device, like a fan, is emitting a constant pitch, you can practice singing harmonic intervals with that device.
- Perform exercises for your friends - long and short - on different instruments. This will help to build competence in musical contexts more realistic than a computer can provide, but still in an isolated situation.

Aural Skills Books:

- Crowell's *Eyes and Ears* is an open-source sight-singing text, available for free from the URL listed above. This book contains about 400 melodies from the public domain, along with some helpful instructions.
- Hall's *Studying Rhythm* is a small and expensive - but useful - book, containing a collection of rhythms to be spoken or spoken-and-clapped. This book also contains some three-part and four-part rhythms. The exercises increase in difficulty. The book offers some performance tips.
- Hindemith's *Elementary Training* is a classic aural skills text, but it was originally published in 1949, and a lot of research has taken place since then about how people learn aural skills. The book offers a wide variety of exercises, especially to develop coordination when performing multiple independent musical lines. We recommend that you ignore Hindemith's instructions, and use *tonic solfa* syllables. For more information on tonic solfa, refer to *Tonic sol-fa* at http://en.wikipedia.org/wiki/Tonic_sol-fa

- Hoffman's *The Rhythm Book* was developed specifically for the takadimi rhythm system. Like Hall's text, *The Rhythm Book* progresses from easy to difficult exercises, and offers helpful instructions for performance.
- Karpinski's two texts are the result of two decades of research about how musicians acquire aural skills. The *Manual* contains instructions, exercises, and tips on training yourself to hear musical elements and real music, and how to perform and imagine music. The *Anthology* is organized in chapters that coincide with the topics discussed in the *Manual*, and contains very few instructions. The *Anthology* contains a wide variety of musical excerpts, some from the public domain and others under copyright, taken from the "classical music" canon and from national repertoires. There are some three-part and four-part exercises, which should be performed by a group of people learning aural skills together.

When you practise sight-singing from any book, we recommend hearing the tonic pitch from a fixed-pitch instrument (like a piano or keyboard synthesizer) before you sing an excerpt. With the tonic pitch in your mind, and without singing aloud, find the starting note of the excerpt, and sing the excerpt *in your mind* several times, until you are sure that you are singing the excerpt correctly. When you have the melody in your mind, sing the excerpt out loud, as many times as you need to be sure that you are singing it correctly. Only *after* you sing the excerpt perfectly should you play it on a fixed-pitch instrument to confirm that you are correct.

You should build your ear with as little help from external sources (pianos, and so on) as possible. A significant amount of research shows that this gives you a more flexible musical mind, and that, while the initial learning curve is very steep, you will ultimately be able to learn new concepts faster.

Aural Skills Books

Ben Crowell. *Eyes and Ears: an Anthology of Melodies for Sight-Singing*. 2004. Ben Crowell . <http://www.lightandmatter.com/sight/sight.html> .

Anne Hall. *Studying Rhythm*. 2004. Prentice Hall .

Paul Hindemith. *Elementary Training for Musicians*. 1984. Schott Music Corp. .

Hoffman. *The Rhythm Book*. 2009. Smith Creek Music .

Gary Karpinski. *Manual for Ear Training and Sight Singing*. 2007. Norton .

Gary Karpinski. *Anthology for Sight Singing*. 2006. Norton .

14.4. Using the Exercises

14.4.1. Listening

1. Open the software
2. It is at the "Front Page"
3. Decide which type of exercise to do
4. Decide which sub-section to focus on
5. Click "New" or "New Interval" or whatever to get the first question
6. On some exercises, you need to click "new" whenever you want a new one

7. Some exercises can be configured to automatically provide a new question when you correctly answer the previous one
8. After hearing each exercise, try to make a correct identification.
9. If you need to hear the exercise again, do it.
10. It is good to limit the number of times you listen.
11. Select what you think is the correct choice.
12. Go to the next question, which may be automatic for some questions. You may want to pre-select a number of seconds to wait before progressing to the next question.

14.4.2. Singing

These are: "Sing intervals"

1. Select "Sing intervals"
2. Choose which ones you want to focus on
3. The exercise will begin, playing the first of the tones you are to sing
4. You must sing the first and the second tone, or to make it harder, only the second tone (tip: use sol-fa syllables!)
5. Solfege does not know whether you sang the interval correctly, so you must tell it.

"Tap generated rhythm"

1. Select "Tap generated rhythm"
2. Choose a subcategory (they correspond to those in the dictation, but there is no compound metre available). See below.
3. Click "New"
4. It will play you a rhythm; listen carefully, and conduct the beat if you can.
5. as with rhythmic dictation, you will be given an intro
6. You must repeat the rhythm by click on the "Tap here" button
7. Best to use the space bar to tap in; here's how.
8. The "accuracy" may be set too high; I like 0.30
9. On "Config," change "Number of beats in question" to adjust the difficulty

"Sing chord"

1. Select "Sing chord"
2. Choose the type of chords you want to sing
3. Click "New"
4. Solfege will automatically play an "A" for you, and you can hear it again by clicking, "440hz"

5. Sing the chord ascending
6. Verify that you sang correctly by clicking "Play answer" and hearing whether the pitches are the same.
7. Click "New" for another question
8. On the "Config" tab, it allows you to change how far it will transpose the built-in models; best to leave this as it is [**'key'**, **-5**, **5**]
9. Solfege does not know whether you sang the interval correctly, so you must tell it.

"Sing chord tone"

1. Select "Sing chord tone"
2. Select which chordal member you want to practise singing
3. Click "New"
4. Solfege will display and play a chord in blocked form, and you must sing the chord member that it tells you to sing.
5. You can repeat the chord in blocked form ("Repeat") or in arpeggio form ("Repeat arpeggio"). It is much easier to hear a chord played in arpeggio form, so we recommend that you practice both ways.
6. When you are sure that you correctly sang the chord member, click "Play answer"
7. For the next question, click "New"

14.4.3. Configure Yourself

These exercises allow you to choose the focus of your training, rather than using a Solfege preset. When you enter an exercise, you are given a default setup, which is then customized on the "Config" tab in the activity. The following things are customizable in "Configure Yourself" exercises, but not in the other counterparts:

"Harmonic intervals" allows you to de/select specific intervals between m2 and M10

"Melodic intervals" and "Sing intervals" allow you to de/select specific intervals between m2 and M10, and whether to test them up, down, or both.

"Compare intervals" allows you to select specific intervals between minor second and major tenth. Also allows you to switch between harmonic or melodic intervals independently for the first and second interval.

"Id tone": allows you to choose a "weighting" for each pitch, to concentrate on specific ones. Also allows you to adjust octave displacement to higher or lower octaves.

For all of the rhythm exercises, "binary time" means "simple metre," and "ternary time" means "compound metre." All sections allow you to choose which single-beat rhythms to use when creating the question.

14.4.4. Rhythm

This is dictation or play-back. The rhythms described in this section use the "takadimi" rhythm system, which is explained in *The Takadimi Article*, available at <http://www.takadimi.net/takadimiArticle.html>. Use the rhythm system you prefer.

For Rhythmic Dictation:

1. Click "Rhythm"
2. Choose which subcategory:
 - Rhythms (easy) is: quarter, 2x eighths, 4x sixteenths
 - Rhythms is: those plus ka-di-mi, ta-ka-mi, ta-ka-di, ta-mi, and ta-ka
 - Rhythms (difficult) is: those plus rests and triplets
 - Rhythms in 3/4 is: compound metre everything
3. Click "New" to get a new question
4. Click the buttons above the "Play" button to input the rhythm-units, in order from start to finish
5. Use paper to work it out
6. If you make a mistake inputting, use the "Backspace" button
7. You can "Repeat" to hear it again - not too many times!
8. You can change the difficulty by increasing the number of beats per question, on "Config" tab
9. If you get a question wrong, you will have a chance to correct it; the incorrect parts are underlined for you in red

For Rhythmic Tap-Back, see above section "Singing Exercises."

14.4.5. Dictation

These dictation exercises are for melodic dictation. There is not a great variety of examples here, and they are either easy or difficult, with no middle-ground.

1. Click "Dictation"
2. Choose a level:
 - Volkslieder 1: German folk songs (easy)
 - Volkslieder 2: German folk songs (easy)
 - Parts of 2 Bach inventions: only 2; the hardest of the four categories
 - Norwegian children songs: only 3 (easy)
3. The clef, key and time signatures are given for you, along with the starting note, and title.
4. The quarter-note buttons allow you to play only part of the melody.
5. "Play the whole music" plays both parts of the music.
6. "Back" and "Forward" shifts through the excerpts for dictation.
7. It's best to "Play the whole music" as many times as needed (5 - 7 or less maximum, depending on the excerpt).
8. It's best to avoid playing only part of the music.

9. Write down the excerpt on paper, then when you're sure that you've finished it correctly, click "Show."
10. This exercise is self-policing, and does not track progress.

14.4.6. Harmonic Progressions

These dictation exercises are for harmonic dictation. You will be asked to guess the harmonic progression, but users should also notate at least the outer voices (lowest and highest). It should be noted that these progressions do not follow Common Practice Period harmonic procedures.

1. Click "Harmonic progressions"
2. Some harmonic progressions require MMA (as indicated). See below for instructions on installation.
3. The Non-MMA Categories contain the following chords:
 - "Easy harmonic progressions": I, II, IV, V, VI
 - with Inversions: add IV6 and V6
 - "Three chords, root position": I, II, III, IV, V, VI
4. Choose a category.
5. Click "New" to get a question.
6. The passage will automatically play once, but you will not get a key signature. If you are notating the dictation, and you do not know which pitches are being used, then you may wish to guess the key, using sol-fa equivalents to know if you're correct.
7. Click the chord-buttons to input the series of chords that you hear, in order from first to last.
8. To hear the example again, click "Repeat."
9. If you make a mistake, click the "Backspace" button to erase the last-input chord.
10. When you are sure that your answer is correct, click "Guess answer."
11. To get another question, click "New".

14.4.7. Intonation

In order to use the Intonation exercises, you must install the "Csound" application. See [Section 14.1.4, "Optional Installation: Csound"](#) for instructions to install Csound.

1. Click on "Intonation"
2. All of the exercises test an ascending perfect fifth. The closer the number is to 1.0, the less the difference when it is out-of-tune.
3. Click "New" to get a new question.
4. The interval will automatically play.
5. Click "Repeat" to repeat the interval.
6. You must choose whether the second tone is flat (the interval is too small), in tune (the interval is the right size), or sharp (the interval is too large).

7. When you are sure of your answer, click the corresponding button.
8. To get the next question, click "New."

DRAFT

Appendix A. Revision History

Revision 0.0 27 July 2010

Christopher Antila
crantila@fedoraproject.org

Initial creation of book by publican

Revision 0.1 2 August 2010

Christopher Antila
crantila@fedoraproject.org

First publically-available draft in DocBook format.

Revision 0.2 3 August 2010

Christopher Antila
crantila@fedoraproject.org

Revised section headings.

Improved and consistent-ified "Requirements and Installation" for DAWs

Revision 0.3 4 August 2010

Christopher Antila
crantila@fedoraproject.org

Significantly revised Audacity chapter.

Updated JACK installation slightly.

Revision 0.4 5 August 2010

Christopher Antila
crantila@fedoraproject.org

Added all external and internal links.

Added all images.

Revision 0.5 6 August 2010

Christopher Antila
crantila@fedoraproject.org

Re-formatted ex-<pre> tags.

Changed images to <inlinemediaobject>, where appropriate.

Revision 0.6 7 August 2010

Christopher Antila
crantila@fedoraproject.org

Added tables to SuperCollider chapter.

Reviewed <code> tags, all but Basic Programming in SC.

Revision 0.7 8 August 2010

Christopher Antila
crantila@fedoraproject.org

Reviewed use of <code> tags in "Basic Programming in SC" chapter.

Searched code for " or '" and replaced them with <emphasis> where appropriate.

Formatted bibliography/book-list in the Solfege chapter.

Revision 0.8 27 August 2010

Christopher Antila
crantila@fedoraproject.org

Re-wrote the JACK installation instructions, to incorporate the fact that jack2 will be in F14
Made other minor revisions to the sound servers section.

Revision 0.9 6 September 2010**Christopher Antila**crantila@fedoraproject.org

Added links to tutorial files.
Incorporated some comments from external sources.
Changed numbering of Revision History.
Added list of links that need to be updated with every release, to Musicians_Guide.xml
Updated Audacity to make it publish-able.
Replaced instances of "Fedora 14" with "Fedora \$PRODVER;"

Revision 14.9 23 December 2010**Christopher Antila**crantila@fedoraproject.org

Changed version numbering to correspond more closely to the intended Fedora release version.
Removed "asdf" part introductions.
Updated entity (.ent) file for Fedora 15.
Updated hard-coded links to supplemental files for Fedora 15.
Changed subtitle to remove possessive ("Fedora Linux's").

Revision 14.9.1 26 December 2010**Christopher Antila**crantila@fedoraproject.org

Fixed Bug 665663 ("URLs to all Tutorial Files Are Broken") for Fedora 15.
Re-wrote Section 12.4.7.1 to fix the first error in Bug 654187.
Renamed some section in Chapter 12, making them easier to translate (I hope).

Revision 14.9.2 2 January 2011**Christopher Antila**crantila@fedoraproject.org

Reformatted dates in Revision History
Re-wrote Section 12.4.8.2 to fix the second error in Bug 654187.

Index

F

feedback

contact information for this manual, xiii

DRAFT

DRAFT