Porsonify: A Portable System for Data Sonification

Tara M. Madhyastha*

Department of Computer Science
University of Illinois
1304 West Springfield Avenue
Urbana, Illinois 61801

October 1, 1992

Copyright © 1992 The University of Illinois Board of Trustees.
All Rights Reserved.

*This work was funded in part by NASA ICLASS Contract No. NAG–1–613, National Science Foundation grants NSF CCR86–57696, NSF CCR87–06653 and NSF CDA87–22836 (Tapestry), DARPA Contract No. DAVT63–91–C–0029, and by grants from the Digital Equipment Corporation External Research Program and Apple Computer.

Contents

Preface **iv**

1 Introduction **1**
1.1 Background 1
1.2 Sonification Goals 2
1.3 Overview of Porsonify 3
1.4 How To Use This Document 3

2 A Porsonify Demonstration **4**

3 Structure and Design of Porsonify **6**
3.1 The Network Audio Access Daemon 6
3.2 Sound Device Servers 7
3.3 Sonifications 10
3.4 Sound Control Files 10
3.5 Sonic Widgets 11
3.6 Putting It All Together 12

4 Getting Started: A Simple Example **15**
4.1 A Simple Example 15
4.2 Selecting a Data File 15
4.3 Running the Sonification 16
4.4 Reconfiguring the Sonification 17
4.5 Writing Widget Control Files 25

5 Porsonify Application Programs **26**
5.1 Interactive Sound Device Interface: `sdinterface` 26
5.2 Creating Sonifications: `porsonify` 27

6 Sound Control File Syntax **28**

7 Transformation Functions **30**

8 Widget Control File Syntax **32**

9 Writing New Transformation Functions **34**
9.1 Transformation Function Implementation 34

10 Supporting New Sound Devices **36**
10.1 Sonic Message Configuration File 36
10.2 Device Defaults File 37
10.3 Device Parameters File 38
10.4 An Example: Writing A New MIDI Instrument Server 38

A Distribution Contents **41**

B System Requirements **43**

C Installing Porsonify **44**
C.1 Building 44
C.2 Installation 44

D Obtaining Equipment and Software **46**

References **47**

List of Figures

1 The demonstration program. 4
2 A sine wave with errors. 5
3 A sound hardware configuration file. 7
4 Communication between application processes and sound servers. 8
5 A boolean sonic widget on the SPARC audio device. 12
6 Sound control file to map data to pitches on the SPARC audio device. 13
7 Sound control file to map data to pitches on a MIDI device. 14
8 A boolean sonic widget that drives a MIDI synthesizer. 14
9 The `porsonify` application main menu. 15
10 Selecting a data file with `porsonify`. 16
11 Configuration of the main transform function for `GettingStarted.wcf` 17
12 Configuration of the boolean transform function for `GettingStarted.wcf` 18
13 Configuration of the boolean transform function for `GettingStarted.wcf`, continued. 19
14 Configuration of the sonic messages in `GettingStarted.wcf`. 21
15 Configuration of **Pitch** in `GettingStarted.wcf`. 22
16 The linear transformation function menu used to configure **Pitch**. 23
17 Changing the linear transformation function for **Pitch** in `GettingStarted.wcf`. . . 24
18 Writing a widget control file after changing a sonification. 25
19 The sdinterface application. 26
20 SPARCserver audio device commands. 27
21 Sample sound control file with fixed, configurable, and variable arguments. 28
22 The widget control file for the `BooleanSonicWidget` used in §4. 32
23 Outline of widget control file syntax. 33
24 Sample sonic message configuration file. 36
25 Sample device defaults file. 37
26 Sample device parameters file. 38
27 Message configuration file for Yamaha TG33. 39
28 Device defaults for Yamaha TG33. 39
29 Device parameters file for Yamaha TG33. 40

Preface

The use of scientific data visualization to provide new insights is well established. However, visualization is but one way of presenting data — psychological studies show that users respond to audio cues more quickly than visual cues in certain instances. Just as visual elements (e.g., color, form, and line) are combined to display and analyze data visually, the elements of sound (e.g., duration, pitch, volume, timbre, and spatial location) can be combined to present data aurally. In short, sound represents a rich communication channel that has heretofore received little attention as a medium for scientific data presentation.

Porsonify is a portable system for data sonification, the aural equivalent of visualization. It is designed to facilitate sonification of data from various sources on a variety of sound hardware platforms. Access to sound devices is based on a client/server architecture, freeing applications from dealing with the low–level details of the sound device driver. All inquiries about Porsonify should be directed to `pablo@guitar.cs.uiuc.edu`.

This work could not have been accomplished without the help of many people. Primarily I wish to thank my research advisor, Dan Reed, for his guidance and support. I am also extremely grateful to Tom Birkett, who wrote the first version of the sound software for the SPARC audio device, for giving me a lot of help and inspiration. Thanks to the entire Picasso group for their friendship and support. Finally, I thank my friend David Zimmerman for being the East Coast Voice.

This work was funded in part by NASA ICLASS Contract No. NAG–1–613, National Science Foundation grants NSF CCR86–57696, NSF CCR87–06653 and NSF CDA87–22836 (Tapestry), DARPA Contract No. DAVT63–91–C–0029, and by grants from the Digital Equipment Corporation External Research Program and Apple Computer.

Tara Madhyastha
October 1, 1992

1 Introduction

Porsonify is a portable system for mapping data to sound. Sound is an interesting medium for presenting data because it can potentially highlight characteristics of data that cannot easily be seen. For example, a movie soundtrack or sound effects that accompany a vidio game convey information complementary to the imagery. The elements of sound (e.g., pitch, volume, duration and timbre) can be used in the same way that visual elements (such as color, form, and line) are manipulated to present and analyze data in visual displays. The use of sound to present data, the auditory equivalent of visualization, is called *sonification*.

Porsonify is designed to encourage experimentation with aural data presentation, or sonification, on a variety of sound devices.

1.1 Background

Humans use incidental sounds to assimilate information about the world around them. Movies, television, and video games rely on the ability of sound to augment the information we derive from the sense of vision. For example, the sound of footsteps growing louder from beyond the edge of our field of vision implies that someone is approaching. The careful listener might be able to deduce a person's appearance or identity from the tapping of shoes or the rustling of clothing. In general, accompanying sounds can either reinforce a visual display or provide an entirely different "view" of the data that can be heard but not seen.

Pitch, volume, duration, location, and timbre form the palette of an sonic display designer, and the canvas is time. The ambient sounds of everyday life are a continuous plot of different data points as a function of the time of day. Sound can serve as an interrupt, or alarm (the bell on the microwave letting you know that it is time for lunch), as a continuous status indicator (the unnoticed ticking of a clock until it chimes on the hour), or as a means of providing background information.

Experimentation with the use of sound to present data has been underway since the early 1980's, when technology made applied research in this area economical [1]. However, the sonifications have often been limited to simple mappings between data values and primitive sound characteristics, such as volume, pitch, and waveshape. In this way it is possible to represent discrete data points as notes. Currently, the sonic equivalents of scatter plots and bar graphs do not exist; there are no well–defined techniques for examining data and determining how it should sound.

Just as scientific visualization initially drew on the experience of the graphics community before developing it's own language and representations, sonification has drawn on the experience of the computer music community. However, like visualization and graphics, sonification and music differ in important ways; sonification idioms are only beginning to emerge. As examples, Scaletti [7], Rabenhorst *et al* [5], and Francioni [3] have begun exploring the use of new idioms and abstractions for mapping data to sound. Scaletti and Craig treat the data set as a stream of audio samples, transforming the new sound created this way, or using it to control other sounds. Rabenhorst and Francioni emphasize instead using the data stream to control parameters of discrete notes.

Much of sonification's embryonic state is attributable to the lack of software standards and common hardware interfaces (e.g., there is no sonic equivalent of the X Window System protocol, nor do all workstations include similar sound hardware). Ideally, we would like to accompany visualizations with real–time output of high quality, digital sound. However, general–purpose digital signal processing hardware, while extremely flexible, is also expensive. At the other extreme, the

audio chip in the Sun SPARCstation[1] provides a ubiquitous hardware platform; unfortunately, its sound fidelity is very low. The Sun SPARCstation audio device can play digitial audio data at a rate of 8000 samples per second with 12–bit precision, resulting in sound quality equivalent to standard telephone service. In contrast, CD quality sound is 44.1K samples per second at 16–bit precision per channel.

A better compromise is instruments that conform to the Musical Interface Digital Interface (MIDI) [4]. Using a variety of built–in synthesis techniques, most MIDI instruments can play notes in many voices (timbres), ranging from traditional voices like a piano or trumpet to interesting sound effects, such as rush–hour traffic in the city. MIDI commands are just a few bytes long, so it is easy to control output in real–time.[2] A wide range of sound mappings can be efficiently produced by connecting one or more of these instruments to a workstation. However, MIDI is a very general protocol, and most manufacturers use "system exclusive," or device–dependent messages to control the unique features of their instruments (such as the ability to create new voices). Furthermore, different instruments can interpret MIDI messages in different ways — a certain voice number might be reserved for the drum kit. This makes it almost impossible to play a composition written for one MIDI instrument on another.

1.2 Sonification Goals

In light of these software and hardware constraints, we have identified the following criteria that we believe a portable sonification system should satisfy:

- Sound hardware should be manipulated by a network server, so that the interface to local and remote sound devices is the same.
- The interface to a particular sound device cannot be too abstract — it must reflect the actual hardware functions. Sound devices vary greatly in their functionality, and attempts to create an all–encompassing set of commands for all types of sonic hardware are bound to be either restrictive or vague.
- The core of a sonic display, analogous to a visual graph or chart, should be both configurable and able to utilize all available capabilities of a sound device.
- It should be easy to create a wide range of sonic displays that operate in real–time, and that can be synchronized with visual displays.

Porsonify is an attempt to satisfy these goals. It provides an environment where it is easy to create very complicated sonic displays of data using whatever sound hardware is available. Portability is achieved by separating the sound hardware interface from the sonic "mapping" (i.e., the algorithm that creates sound to display the data).

Although this software was written to complement the Pablo performance analysis and visualization environment [6], it is general enough to be used independently by a variety of other applications, or by itself.[3]

[1] Sun SPARCstation is a trademark of Sun Microsystems Incorporated.

[2] In contrast, the Sun SPARCstation plays sampled audio data, which can require a long time to synthesize and a lot of software coordination to correlate.

[3] Pablo is a scalable, portable performance analysis tool which contains a set of data transformation and presentation modules, to be interconnected in user–specified ways.

1.3 Overview of Porsonify

The design of the Porsonify toolkit reflects the success of the X Window System client/server protocol. In Porsonify, each sound device is managed by its own network server, which accepts *sonic messages* that cause sounds to be played. An application program opens a connection to a sound device, receives a list of supported capabilities, and uses sonic messages thereafter to communicate with the sound device.

These sonic messages can be grouped to form a *sound control file*. *Transformation functions*, which apply simple mathematical operations to input data values, can be used to select a particular sound control file or to vary the parameters of individual sonic messages within a sound control file.

The highest level interface to sound control files and transformation functions is a *sonic widget*. Sonic widgets manipulate transformation functions and sound control files to create *sonifications*. A sonic widget can write a *widget control file*, specifying the transformation functions and sonic messages that comprise the sonification, so that it can be saved and recreated.

For example, suppose one wished to construct a sonification that mapped data values to musical notes, where the volume of each note represents the magnitude of the corresponding value. This sonification can be realized with a sonic widget that manipulates a single sound control file. In this example, the sound control file would contain commands that set the sound volume and play a note. The volume is a data–driven argument, controlled by a transformation function. If the sound device accepts volume settings in the range 0–100, and the data values lie in the range 0–1000, the transformation function could be configured to first scale each data value by 10 and then use the scaled value as the volume parameter.

1.4 How To Use This Document

This document can be read in two ways: as a guide to using the current software, or as a reference manual for those wishing to extend the system to support new features.

If you have not yet built and installed the system, before reading further you will probably want to consult Appendix C for installation instructions. The system software (e.g., `g++`, X11R5/Motif) necessary to build and run Porsonify is described in Appendix B.

Throughout this text, the following font conventions are used.

`typewriter` Indicates program names, code objects, and fragments of code or configuration files.

boldface Denotes menu entries.

A simple demonstration program, designed to illustrate some of the ways one can use sound to examine data, is described in §2. Those who wish to use Porsonify should see §3 for a comprehensive overview of the Porsonify system structure and design. It might be useful to look at this before reading §4, which is an introduction to using the standalone applications to produce sonifications. The standalone applications included in Porsonify, which allow you to create sonifications and manipulate sound devices, are described in greater detail in §5. At an intermediate level, see §6, §7, and §8 for more information about sound control files, transformation functions, and widget control files; these are useful references if you want to learn how to create new sonifications. Finally, if you are interested in expanding the system to support new transformation functions and sound devices, you should refer to §9 and §10. These contain low–level details that can also be found in the source code documentation.

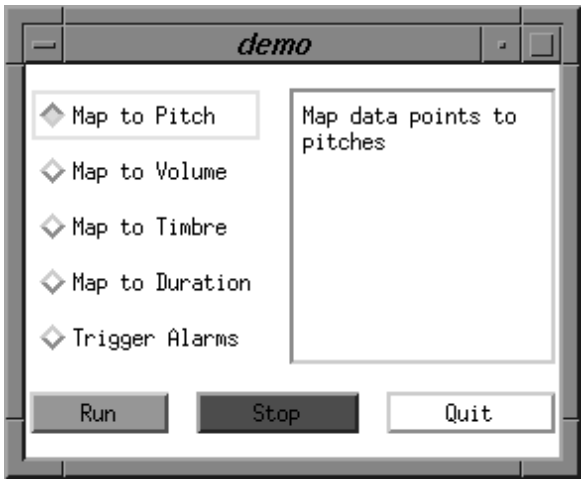

Figure 1: The demonstration program.

2 A Porsonify Demonstration

After building and installing the Porsonify software, as described in Appendix C, you can run the demonstration program. To do this, change to the `Demo` subdirectory in this distribution, and type `runDemo`. If the software has been built and installed correctly on a Sun SPARCstation, this will create the window shown in Figure 1. This program uses only the SPARC audio device.

This demonstration program will allow you to experiment with different sonifications of a sample data file, called `TestData.sine`, that contains samples of a simple sine curve ($y = sin(x)$) with x ranging from 0 to 10 radians.[4] A plot of this file is shown in Figure 2; it contains errors that are marked by an arrow.

Simply select one of the mapping options by choosing the appropriate radio button on the left and then push the **Run** button. The window on the right describes the selected sonification. To stop the sonification, push the **Stop** button.

The function of each of button is described briefly below.

Map To Pitch

For each data value, a note is played for 0.25 seconds with a pitch corresponding to the magnitude of the data. The higher the magnitude, the higher the pitch.

Map To Volume

For each data value, a note is played for 1 second whose volume is controlled by the magnitude of the data. Larger data values produce louder notes.

Map To Timbre

For each data value, a note is played for 0.5 seconds. Positive values correspond to notes of the "plucked string" timbre, and negative notes are played by the "flute" instrument. The pitch remains constant.

[4] We will refer to this example in §4.

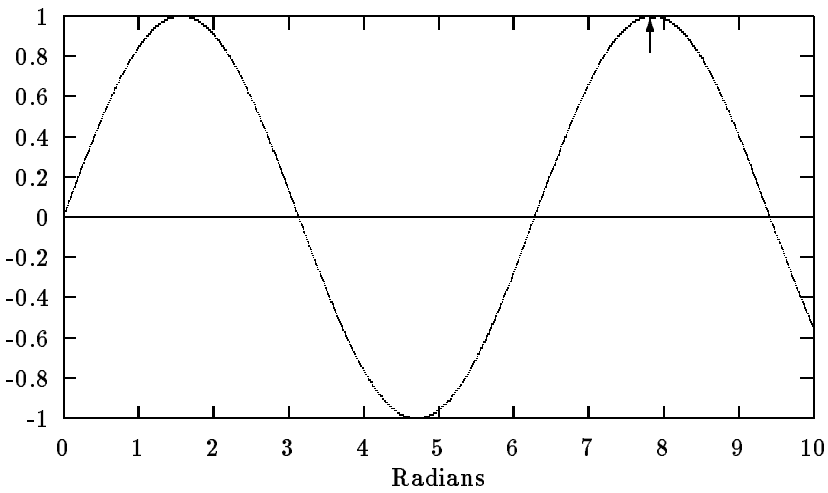

Figure 2: A sine wave with errors.

Map to Duration

For each positive data value, a note is played with a duration equivalent to twice the magnitude of the value, in seconds.

Trigger Alarms

Alarms are triggered at extreme values. A chime is played for data values in the range 0.98 to 1.0, and a chord is played for data values in the range -1.0 to -0.98.

3 Structure and Design of Porsonify

In Porsonify, access to sound devices is based on a server/client architecture. A *sound device server* specific to each sound device encapsulates all the device dependent code, and it accepts *sonic messages* that manipulate the device. Client applications interact with the sound device servers through a more general–purpose network audio access daemon, or *naad*.

This design was chosen for several reasons. The first and most practical reason was to allow remote access to sound devices. In an office or laboratory with three workstations and a sound device connected to only one of them, the sound device should be equally accessible to all. The second reason is portability — sound devices are manipulated exclusively through sonic messages, so that client application code needs only a textual representation of the sonic messages (saved in a *sound control file*) that it will send to the device. New sound hardware can be utilized by editing the sound control file to suit the new device. Finally, this scheme allows for concurrency. The *naad* allows multiple applications to access the same sound device server, ensuring that operations such as initialization functions are only performed the first time the sound device is accessed. In this way, for example, one application can provide an interface to control global sound device parameters, and another can send commands to the device.

When an application program wants to use a sound device on a particular host, either local or remote, it first contacts the *naad* on its local host. The *naad* determines if the sound device is free, and if so, establishes a connection between the application and the sound device server. The sound device server then sends the application a description of all the commands the sound device can support. Thereafter, the application simply manipulates this description to interact with the sound device; the location of the sound device and its idiosyncrasies are hidden.

3.1 The Network Audio Access Daemon

Because most applications gain access to a sound device through the network audio access daemon, or *naad*, each workstation that uses the Porsonify interface must start a *naad*, even if it supports no sound devices. Any application that wishes to access a local or remote sound device must first contact the *naad* on the local host. Specifically, there are two types of requests to which the *naad* can respond.

- Describe the sound devices that are available on the local workstation.
- Initiate a connection to a sound device on a local or remote workstation.

To determine what sound devices are available on remote workstations, an application can contact the *naad* on any remote workstation and ask what devices are available locally – much in the same way that one might call out–of–state directory assistance to obtain a phone number local to a remote city.

Locally available sound devices are described in a system configuration file that contains a textual description of the sound device and the TCP/IP port name listed in the file `/etc/services` that is associated with the specialized server; Figure 3 shows an example of a configuration file for a Sun SPARCstation, called `foobar.cs.uiuc.edu`, with a MIDI synthesizer connected to it. The configuration file specifies that that the SPARCstation audio device can be accessed through the `sparcaudio` port, and the MIDI synthesizer can be accessed through the `midiaudio` port.

```
# Sound hardware configuration file for foobar.cs.uiuc.edu
#
#    Description                port (from /etc/services)
"Sparc Audio Chip"                      sparcaudio
"Yamaha PortaSound MIDI synth"          midiaudio
```

Figure 3: A sound hardware configuration file.

A sound device server that accepts commands for the Yamaha PortaSound PSS–680 MusicStation[5] MIDI synthesizer would listen on port `midiaudio` on `foobar.cs.uiuc.edu`. In general, a host name and port name pair uniquely describes a particular sound device server. After a host name and a port name have been selected (e.g., by querying some host's *naad*) a connection can be initiated to that sound device server.

Clearly, there may be situations where *multiple* application processes on a particular host need to access a *single* sound device. For this reason, the *naad* accepts requests to initiate connections to sound devices. If a connection to the requested sound device is not already open, the *naad* creates a named FIFO pipe and forks. The child *naad* process establishes a connection to the port associated with the specified sound device. Thereafter, the child process simply reads from the pipe and sends the commands to the remote device. Given the pipe's name, the requesting application process can then open the pipe, locking it before writing commands and unlocking when finished. Future requests for this sound device from other application processes on the same host can be answered by the *naad* simply by returning the pipe name. The child *naad* process dies when no applications have the pipe open.

Figure 4 illustrates a scenario where two application processes on the local workstation (A and B) are accessing a MIDI synthesizer on a remote workstation. The *naad* on the local workstation has forked a child process that manages the communication between the remote sound server for the MIDI synthesizer and the two application processes.

While these two application processes are connected to the MIDI server, no other processes on any host except the local workstation can access the associated sound device. Although not strictly necessary, this restriction simplifies the sound device servers; they simply receive and execute messages. The *naad* provides the necessary coordination between application processes on the same host.

3.2 Sound Device Servers

The network interface, as described above, is the logical level for the abstraction between the application and the software that drives the sound device. This layer of abstraction is designed so that sonifications can take advantage of the capabilities unique to specific hardware with no device–specific code.

Each instance of a sound device has its own server that listens for connections on a TCP/IP port. When a program initiates a connection to a sound device server, it receives a set of configuration

[5]Yamaha PortaSound PSS–680 MusicStation is a trademark of Yamaha Corporation.

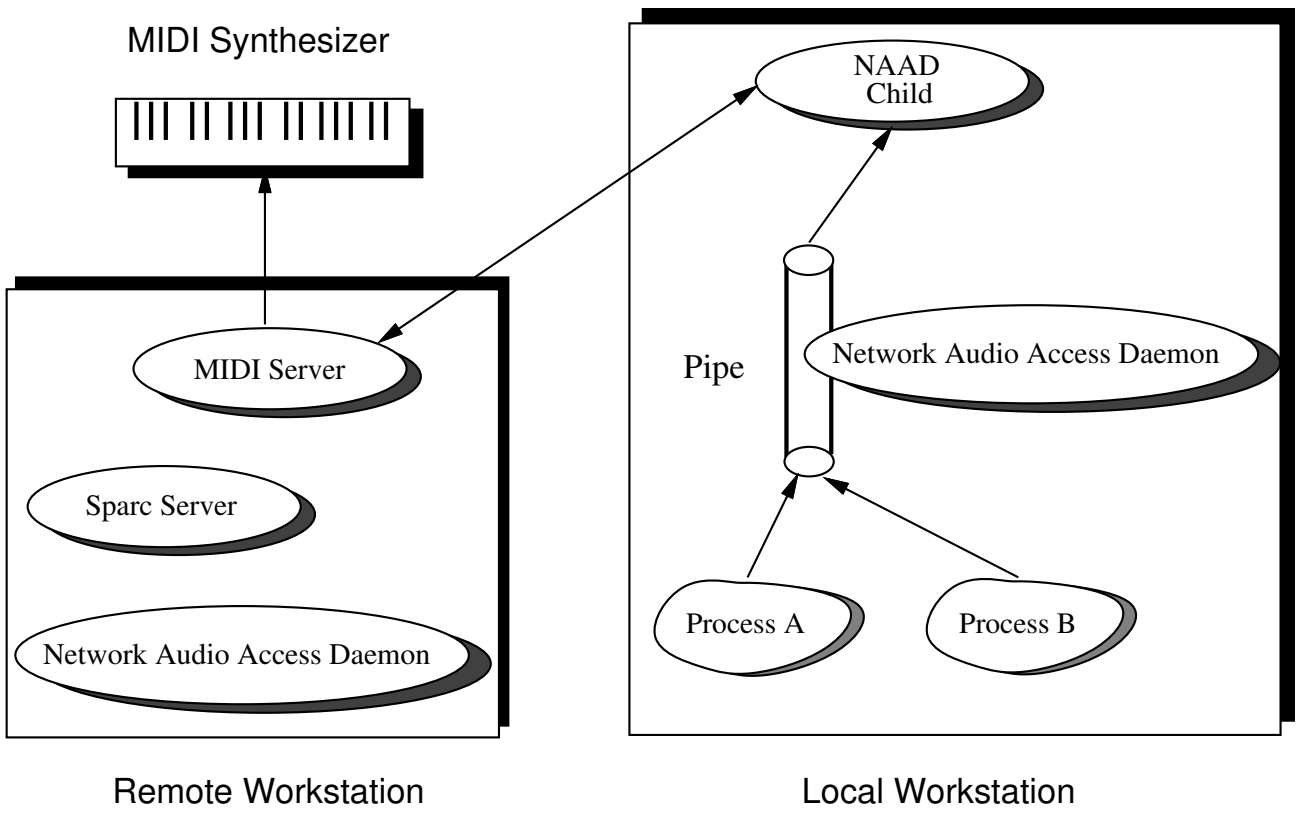

Figure 4: Communication between application processes and sound servers.

tables defining the commands the server accepts, the default arguments to those commands (or *device defaults*), and fixed *device parameters* that specify valid ranges for the arguments.

Imagine a sound device that supports only one function: it can beep for a specified duration of up to ten seconds. The textual representation of the command the server accepts(i.e., the *sonic message configuration table*, that describes the command to make it beep) is as follows:

```
"Beep"      doBeep        DURATION
```

The first field is the name of this command. The second field refers to the function which the sound device server calls to execute this command. The remaining field is an argument to the `Beep` command. Command arguments are described in the *device defaults* table, which for this simple device would appear as follows:

```
DURATION   INTEGER  "Duration in seconds"  1   MINDURATION  MAXDURATION
```

The first field above is the same key that appears in the sonic message configuration table. The second field is the type of the argument. This is followed by a textual description of this argument, its default value (1 second), and the keys of the device constants that bound the duration. These bounds are delineated in a *device parameters* table, shown below for the simple beep device.

```
MINDURATION    INTEGER    "Minimum Duration"    0
MAXDURATION    INTEGER    "Maximum Duration"    10
```

The first field in each line above is the same key that appears in the device defaults table. The second column is the type of these bounds, integer in this case. The third column is the description of the constant, and the fourth is the constant value.

When an application has received these tables, it knows what kinds of commands the sound device supports and the range of arguments for these commands. For example, in this example, an application can determine the beeping sound device supports only one command, **Beep**, which accepts one integer argument, **Duration in seconds**, that must lie between a **Minimum Duration** of 0 and a **Maximum Duration** of 10. An application can cause the device to beep for two seconds by sending it a *sonic message* of the form:

```
"Beep" 2
```

Sonic messages, synthesized using these tables, cause commands to be executed on a sound device and are the basic units of activity. Sonic messages may either cause a sound to occur (e.g., play a note), or change some sound parameter (such as volume) that affects current or future sounds.

Examples of some sonic messages for more sophisticated devices might include the following.

```
"Set Volume"        65
"Set Instrument"    piano
"Play Note"         60  1.0
```

These messages are not very descriptive without understanding the associated configuration tables, which specify the range of valid volumes for the **Set Volume** command, a list of instruments for the **Set Instrument** command, and the range of the pitch and duration arguments to the **Play Note** command.[6]

A sound device server listens on a network port unique to the interface for that device. If a host supports two sound devices of exactly the same type, their servers must run on different ports (e.g., **midi1** and **midi2**). However, the software assumes that all ports with the same name accept the same (or compatible) sets of commands. Two different port names might map to the same physical device, but manipulate it with completely different interfaces. For example, a workstation might be connected to a MIDI synthesizer via a serial to MIDI converter. The synthesizer might be accessed through the workstation in two ways: by writing MIDI control messages directly to the serial port, and by a UNIX kernel device driver, which can accurately play timestamped MIDI messages. In the first case, commands are executed as they are received. In the second, each command has an extra argument: a timestamp. Commands are executed only when the timestamp matches the time on the MIDI device driver clock. These two interfaces to the same physical hardware would be managed by two different sound device servers. They would synchronize to avoid conflicts, such as ensuring that there is only one active connection to the MIDI device and that the synthesizer is placed in a consistent state after each interaction.

This layer of abstraction can be used to create a general interface to any sound device, without requiring that each device support the same set of messages. Sound devices are so varied in their

[6] In reality, once the configuration tables have been communicated, messages are identified by number, not name (as shown above). These configuration tables are described in greater detail in §10.

capabilities that making them conform to one interface would be impossible. Instead, each device advertizes only the commands it can process.

3.3 Sonifications

A *sonification* is anything that aurally presents data. It is analogous to a visual display, except that when the display is updated, instead of changing without regard to time in a fixed two–dimensional screen space, it changes over a variable amount of time as a rhythmicized sequence of sounds is played.

Updating a sonification involves sending a stream of sonic messages to one or more sound devices for each data value that the sonification should "display." Data values may be scalars, or a sequence of scalars may be grouped as vectors.[7] Each scalar data value, or each element of a vector, can be used as a sonic message argument. For example, a simple sonification of points in a two–dimensional space might accept a stream of vectors of length two where the vector elements contain the x–coordinate (index 0) and the y–coordinate (index 1). For each input vector, a short note is played whose pitch is determined by the y–coordinate, and whose source location (balance) is determined by the x–coordinate. This kind of sonic display requires no memory of previous data and no knowledge of future inputs.

All sonifications must be constructed similarly, if the purpose of a sonification is to present data. The sonification must produce sounds or affect the sounds that are played every time it receives a new data value. Assuming we can preprocess the data so that values to be presented can trigger temporally ordered sonic events, writing a sonification can be reduced to determining what sonic messages are sent to what sound devices for each data value.

Creating sonifications is simplified by four major abstractions: *sound control files*, *transformation functions*, *sonic widgets* and *widget control files*. Sound control files are scripts written for a particular sound device that list sonic messages to be sent to that device's server. Transformation functions, which provide simple functional modifications of input data values, are used to choose between sound control files or to vary the parameters of individual sonic messages. Sonic widgets manipulate one or more sound control files and an optional transformation function to create a specific sonification. For example, a sound control file for the SPARCstation audio device might include a command to play a sampled chime. A time display called "Big Ben," which accepts hours and minutes as input, might be constructed using a sonic widget which executes the sound control file once for every hour in the current time if the minute field is zero. Finally, sonic widgets can save their complete configuration information in widget control files. In this way, sonifications can be saved, edited, and reused.

3.4 Sound Control Files

Sonic messages and their arguments can be encapsulated as a *sound control file*. The syntax of this file is described in greater depth in §6 — it is a list of messages and arguments specific to a particular sound device server. Arguments may be fixed (cannot be changed), configurable (can be changed at initialization), or variable (a function of values in an external data stream).

Sound control files can encapsulate one or more *transformation functions* that apply some simple mathematical operations to scalar data values. Transformation functions are most often used to

[7] Typically, a vector represents a point in a multi–dimensional data set.

provide the values for variable arguments in sound control files. However, they can also be used to select a particular sound control file to execute, based on the input data. This is described in more detail below.

3.5 Sonic Widgets

A *sonic widget* manipulates one or more sound control files and an optional transformation function without knowledge of either the underlying sounds produced by the manipulation or the characteristics of a sound device. If a transformation function is present, it can be used to select a sound control file or determine how many times to execute a single sound control file. The sound control files themselves may have variable parameters that are instantiated from the data stream. In short, the sonic widget provides a high–level means of manipulating groups of sonic messages with data.

For example, suppose one wishes to map incoming scalar data values to some characteristic of a note, such as pitch, timbre, duration, or volume. This would be a suitable task for the simplest sort of sonic widget: `SimpleSonicWidget`. This widget manages a single sound control file, which contains the commands to play a note on a particular sound device. The sound attribute of interest would be a variable argument in the sound control file, and a transformation function would determine how the argument's value is calculated from the values in the data stream.

Another common type of sonification is an alarm, where a sound is produced when certain conditions in the input data are satisfied. The `BooleanSonicWidget`, which manages one transformation function and one sound control file, can be used to implement an alarm. If the value produced by the transformation function is greater than zero, the commands in the sound control file are executed. As always, the sound control file may in turn contain variable arguments, so that like `SimpleSonicWidget`, it can map the input data to different sound characteristics.

In addition, the data value can select one of several sound control files to execute. `MultiSonicWidget` manages a transformation function and several sound control files to do just this. For example, suppose one wishes to sonify a stream of positive and negative integers. If the number is positive, a high note is played, and if it is negative, a low note is played. This sonification can be created with two sound control files, one to play each pitch, and a boolean transformation function that compares each input data value to zero. The result of the transformation function, true (1) or false (0) is used to select which sound control file is executed.

The input data can also be used to determine how many times a particular sound control file should be executed. This kind of widget, `LoopingSonicWidget`, is what we might use to implement the "Big Ben" sonic display described in §3.3. It has a transformation function and a sound control file; the transformation function operates on the input data to produce a number, and the sound control file is executed that number of times. In the "Big Ben" sonification, the input data is the number of hours in the current time, the transformation function passes the hour value through unchanged, and the sound control file plays a single chime.

The last kind of widget provided in Porsonify, `InitializationSonicWidget`, manages two sound control files. One sound control file is used strictly to initialize the sound device, and the other is used in the same way as `SimpleSonicWidget`. This kind of widget is invaluable for creating displays using MIDI synthesizers. Typically one selects a certain voice (and other characteristics) once, and reuses those settings thereafter.

Sonic widgets can create a file containing a textual representation of the objects they manage; this is called a *widget control file*. Sonic widgets can later be recreated from these files. Thus,

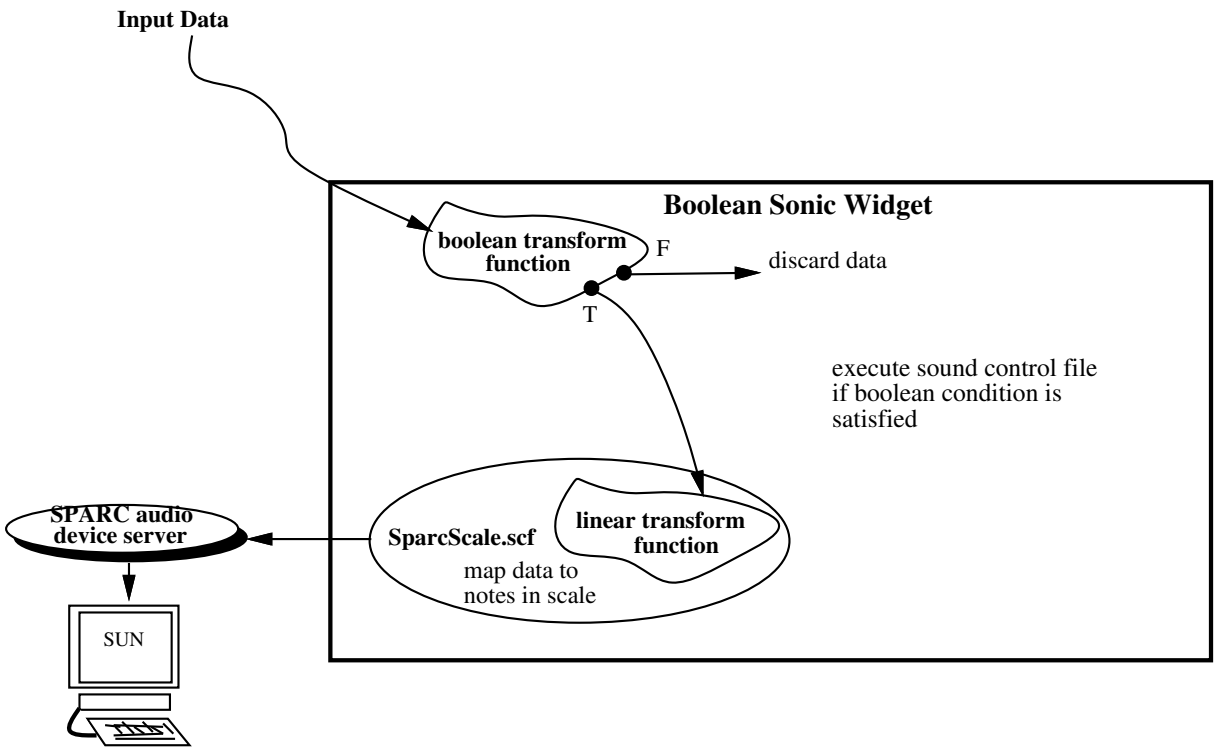

Figure 5: A boolean sonic widget on the SPARC audio device.

sonifications can be saved and reused.

3.6 Putting It All Together

Consider a sonification that accepts a stream of scalar data values and maps values that are greater than a certain threshold to notes in a musical scale. A single sound control file can map the data to pitches, and a transformation function can filter values that meet certain conditions. Therefore, a `BooleanSonicWidget`, which manages one sound control file and one transformation function, is best suited to this task. Figure 5 illustrates how sound control files and transformation functions are combined by a `BooleanSonicWidget` to form this sonification.

The boolean transform function in the `BooleanSonicWidget` checks to see if each input scalar value satisfies some boolean condition, and if so, the sonic widget executes the sound control file with the input data. The sonic widget recalculates the value of each of its variable parameters in the sound control file using its associated transform function. In this example, the sound control file has only one variable parameter, the pitch of each note, controlled by a linear transform function. The linear transform function evaluates the expression $a * x + b$ on the input data value x, where a and b are user–specified constants.

After the parameter value has been calculated, the widget sends the sonic messages to the Sun SPARCstation audio device server.

Figure 6 shows a sound control file that could be used with the `BooleanSonicWidget`.[8] This

[8] The syntax of this file is described in more detail in §6.

```
# Map input data to notes in a scale

# hostname    port name
localhost    sparcaudio

# sonic message name     message number      arguments
/* Set Output Port   */        4             ?1
/* Set Volume        */        0             ?60
/* Select Instrument */        3             ?3
/* Play Note         */        5             $0  ?1.0
```

Figure 6: Sound control file to map data to pitches on the SPARC audio device.

file contains sonic messages intended for the Sun SPARCstation audio device. In it is a `Play Note` message, which has two arguments, a pitch and a duration. The pitch is a variable argument (designated in the sound control file by `$0`).

The messages in this file are specific to the Sun SPARCstation; for example, "Set Output Port" determines whether the sound will be output to headphones or speakers. The sonic message most critical to the sonification is the one with the variable argument (designated in the sound control file by `$0`), "Play Note." It is this command that maps the input data to pitch. The transformation function that controls the pitch is not specified in this particular sound control file; when configuring the sonic widget, the user would select one that maps data from its natural range into suitable pitches in a chromatic scale.[9] When sonic widgets are assembled in this way, the sound–device specific information is isolated in the sound control file. In this example, the file can be replaced with any other sound control file that contains sonic messages to play a note and vary the pitch based on the input data. The entire software hierarchy can be modified to control a different sound device (for example, a MIDI device) by replacing the sound control file for a SPARCstation with an equivalent one for a MIDI device. Figure 7 is an example of an sound control file that would map data to pitches on a MIDI device. Note that there is only one sonic message, "Play Note on Channel," but it has a variable pitch argument as well. Any assortment of messages that makes sense could go here, as long as the basic mapping function is the same. Selection of sonic messages is left to the sonification designer. Figure 8 illustrates how the software hierarchy changes when the sound control file data in the `BooleanSonicWidget` is replaced with the sound control file data for the MIDI device.

[9] We are following the convention that 60 is C4 (middle C), and each integral value represents a half–step.

```
# Map input data to notes in a scale

# hostname    port name
localhost   midiaudio

# sonic message name         message number      arguments
/* Play Note on Channel */         18            $0  ?1.0   ?127  ?0
```

Figure 7: Sound control file to map data to pitches on a MIDI device.

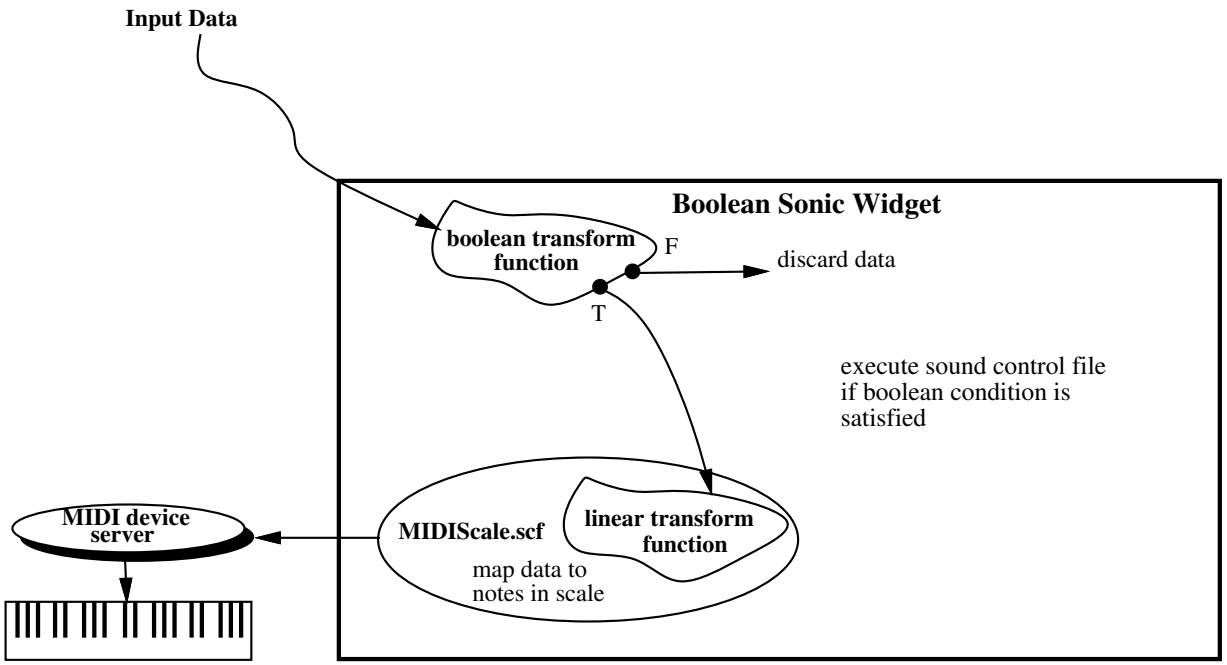

Figure 8: A boolean sonic widget that drives a MIDI synthesizer.

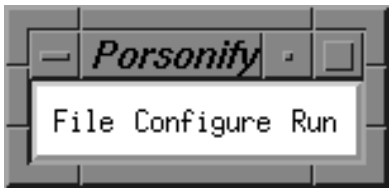

Figure 9: The `porsonify` application main menu.

4 Getting Started: A Simple Example

This section assumes you have built and installed the Porsonify software as described in Appendix C. It also assumes you have access to a Sun SPARCstation audio device; the examples below all rely on the `sparcaudio` sound device.

4.1 A Simple Example

The easiest way to create a sonification is to use the `porsonify` program. This program accepts a widget control file as a command–line argument.

To see how `porsonify` works, try the following command (assuming you have installed the software in `/usr/local/Porsonify`).

```
porsonify /usr/local/Porsonify/WCF/SPARC/GettingStarted.wcf
```

The window shown in Figure 9 will appear. The widget control file `GettingStarted.wcf` describes a sonification of the same structure as discussed in §3.6 (i.e., a boolean sonic widget that accepts values greater than a certain threshold value and executes commands in a sound control file for those values).

4.2 Selecting a Data File

Because the `porsonify` application maps a sequence of data values to sound, you must specify a data file containing the data. For this example, you should select the file

```
/usr/local/Porsonify/data/TestData.sine
```

This data file contains samples of a "defective" sine curve ($y = sin(x)$) with x ranging from 0 to 10 radians. The irregularity in this curve is difficult to detect visually, but it is easy to hear. Figure 2 shows a plot of this curve, with the errors marked by an arrow.

To use this file, select **New data file** from the **File** menu. This will create the file selection popup dialog, shown in Figure 10. You can choose the data file by typing the choice into the **Selection** text window, as shown, and pressing the **OK** button. It is also possible to move to the directory where the data file is located by selecting a directory from the **Directories** scroll box, and pressing the **Filter** button. This will change to that directory and list all files there in the **Files** scroll box. If the desired data file is printed in the **Files** listing, it can be selected by clicking the mouse on the file name from that list, and then clicking on the **OK** button.

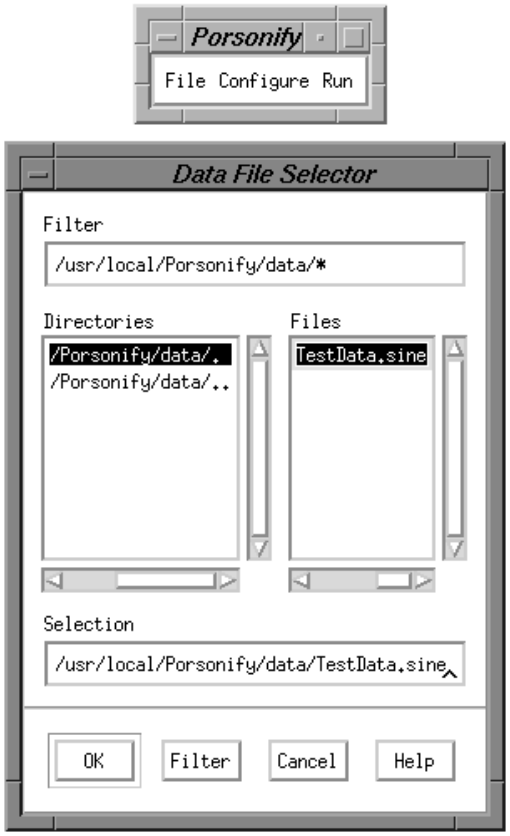

Figure 10: Selecting a data file with `porsonify`.

4.3 Running the Sonification

To listen to the sonification of the selected data file, choose **Start** from the **Run** menu. The notes that you hear are produced by substituting each data value x in the boolean expression $x > 0.8$. If the expression is true, a pitch determined by the formula $100 * x - 10$ is played.[10] This scaling formula produces notes that lie in an audible range.

You can stop the sonification by choosing the **Stop** menu selection from the **Run** menu. If you choose to start it again, it will start from where it was last stopped. When all the data values have been read from the data file, the sonification stops. Then you can either select another data file, or "rewind" the current file (moving back to the beginning) by selecting **Rewind** from the **Run** menu.

[10] The pitch numbered 60 is equivalent to middle C on a piano keyboard.

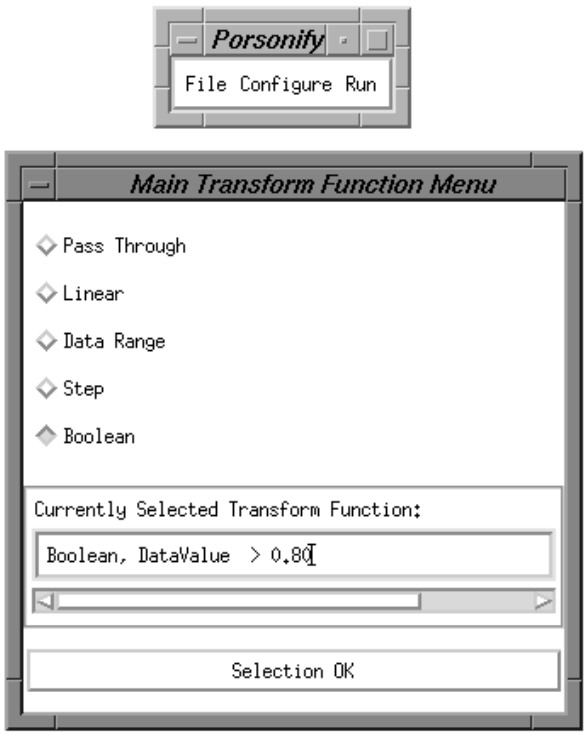

Figure 11: Configuration of the main transform function for `GettingStarted.wcf`

4.4 Reconfiguring the Sonification

All the information about how sounds are produced for each data value is contained in the widget control file `GettingStarted.wcf`. It is possible to interactively change this information to customize the sonification. `GettingStarted.wcf` describes the configuration of a `BooleanSonicWidget`, shown in Figure 5. First, we configure the transformation function that determines if any action is taken on the data value, and then we configuure the sonic messages that are sent.

To do this, select **Configure** from the `porsonify` main menu. This will produce the display shown in Figure 11. The commands in `GettingStarted.wcf` play a note for each data value that satisfies a certain boolean condition. The pitch of the note is controlled by the data value.

In this example, we sonify all points with values greater than 0.8 (the peaks of the curves in Figure 2). This is indicated by the label on the button at the bottom of Figure 11, which displays the currently selected transform function. Each data value x is tested in the expression $x > 0.8$; nothing happens unless this test returns true. To continue configuration of the sonification without changing this transformation function, press the button at the bottom of the screen.[11]

To change the currently selected transform function, select one of the options by clicking the mouse on the radio button to the left of the desired data transform.[12] As an example, suppose we want to change the boolean function so that it only plays notes when the data value x satisfies

[11] At any time, pressing the button indicating the currently selected transform function will close this window without making any changes.

[12] The available transform functions are described in more detail in §7.

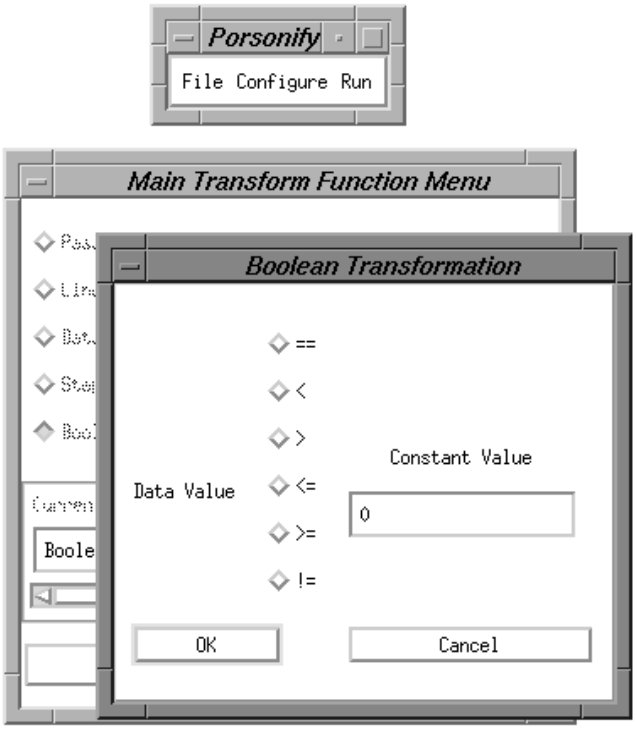

Figure 12: Configuration of the boolean transform function for `GettingStarted.wcf`

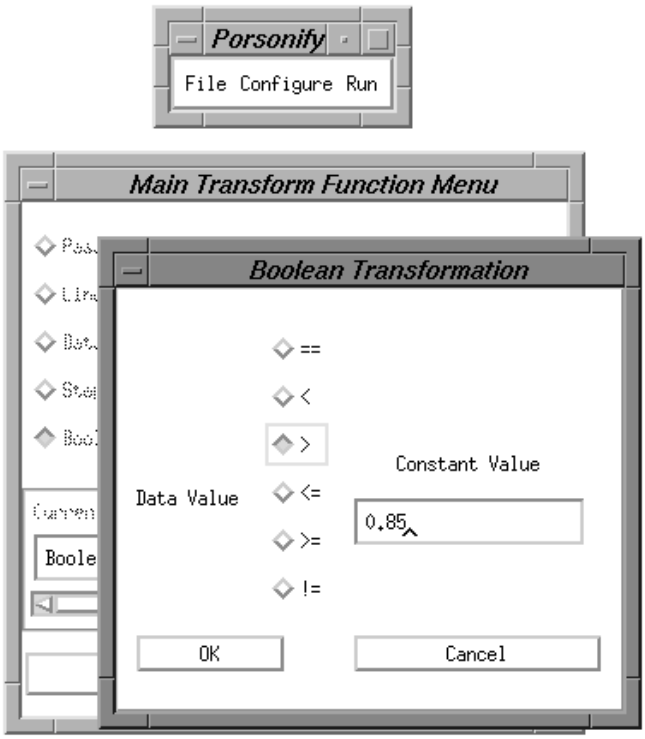

Figure 13: Configuration of the boolean transform function for `GettingStarted.wcf`, continued.

$x > 0.85$. Click the radio button marked **Boolean** to produce the screen shown in Figure 12. To set the condition for the boolean expression, select the $>$ radio button, and type in `0.85` in the **Constant Value** window; the windows should appear as shown in Figure 13. To cancel this change, press the button marked **Cancel**. This will return to the main transformation function menu. Otherwise, click on the **OK** button to dismiss all windows related to transformation function configuration and continue configuring the sonification.

The configuration menu for the commands to play a note for each data value will now appear, as shown in Figure 14. By default, audio output will be sent to the headphone port of the Sun SPARCstation. If you do not have headphones or officemates, change the output port to **Audio Speaker** by clicking on the desired item.

The pitches of the notes that are played for data values satisfying the previously specified condition are determined by the formula $100 * x - 10$. Like the boolean condition we selected earlier, this formula is specified by a transformation function. To change this formula, press the button above the **Transform for "Pitch"** label marked **Reconfigure**. This will create another transform function menu, shown in Figure 15. Notice that the label for the currently selected transform function is **Linear, A = 100, B = -10**. This transform function can be changed just like we changed the boolean transformation function earlier. For example, it is possible to play the same note (e.g., pitch 60) for each data value by selecting the linear transformation function $0 * x + 60$. To do this, click on the button marked **Linear**. The linear transformation configuration panel will pop up, as shown in Figure 16. Type the value 60 into the text box on the right. The screen will now look like in Figure 17. Press **OK** to continue.

When you have finished configuring the sonification, the main configuration panel can be dismissed by pressing the **Dismiss** button.

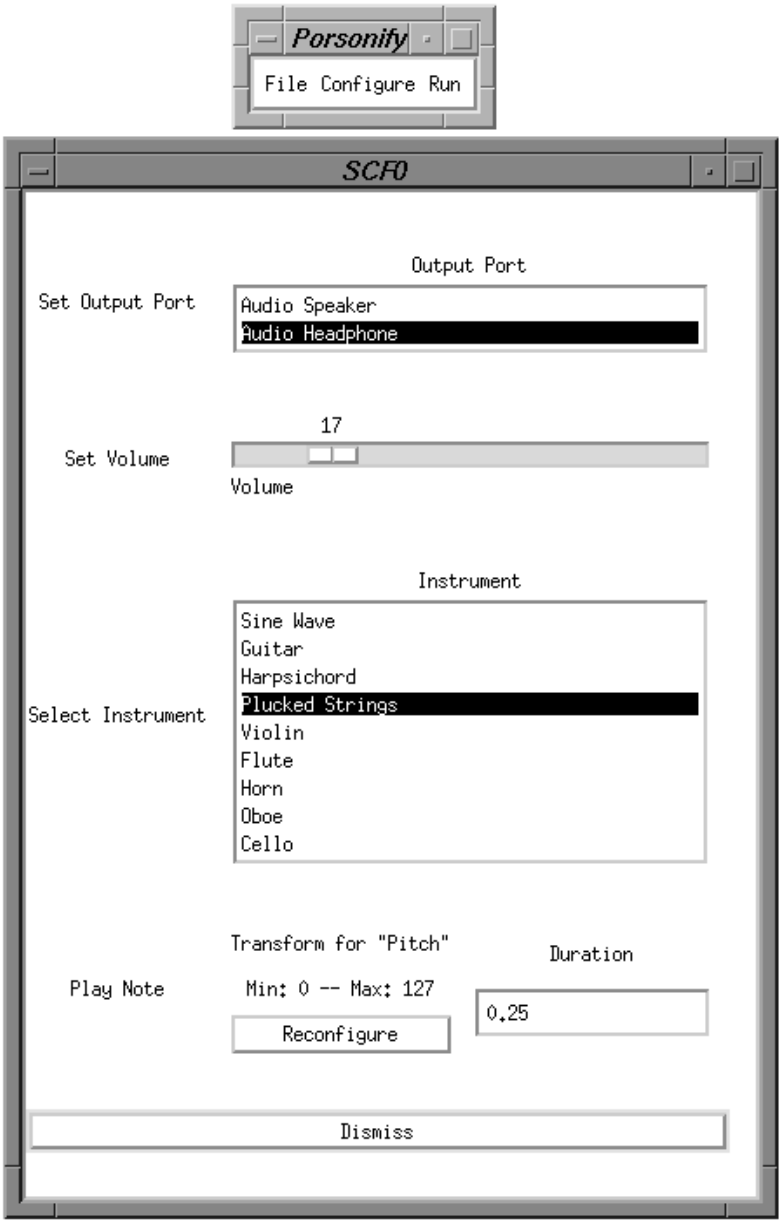

Figure 14: Configuration of the sonic messages in `GettingStarted.wcf`.

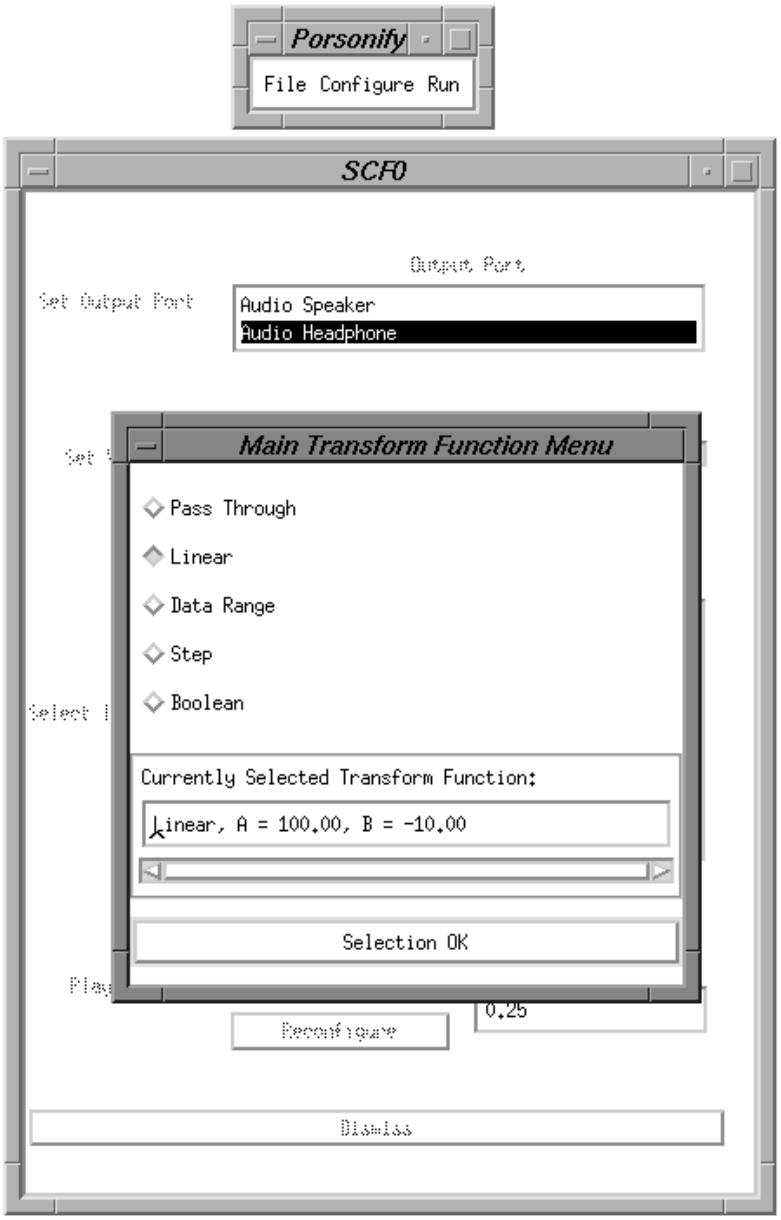

Figure 15: Configuration of **Pitch** in `GettingStarted.wcf`.

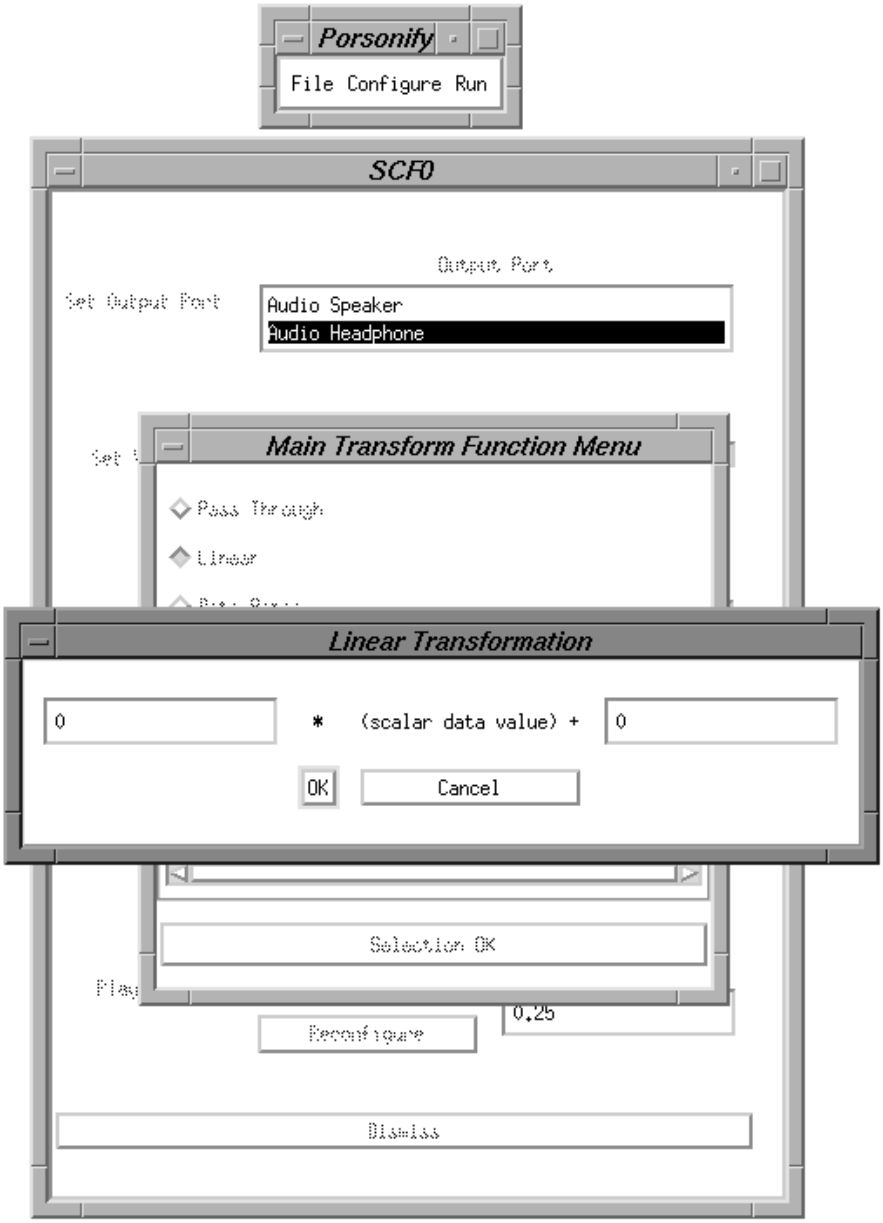

Figure 16: The linear transformation function menu used to configure **Pitch**.

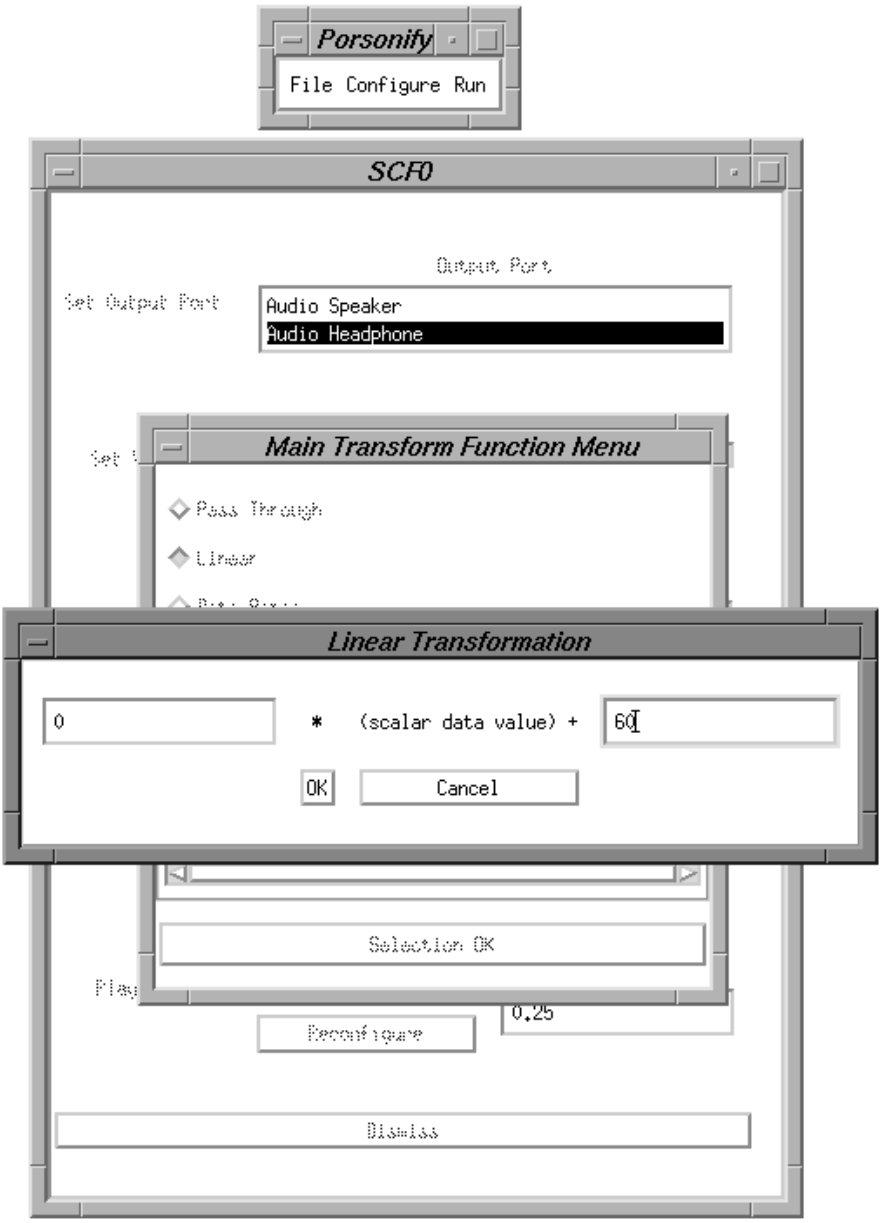

Figure 17: Changing the linear transformation function for **Pitch** in `GettingStarted.wcf`.

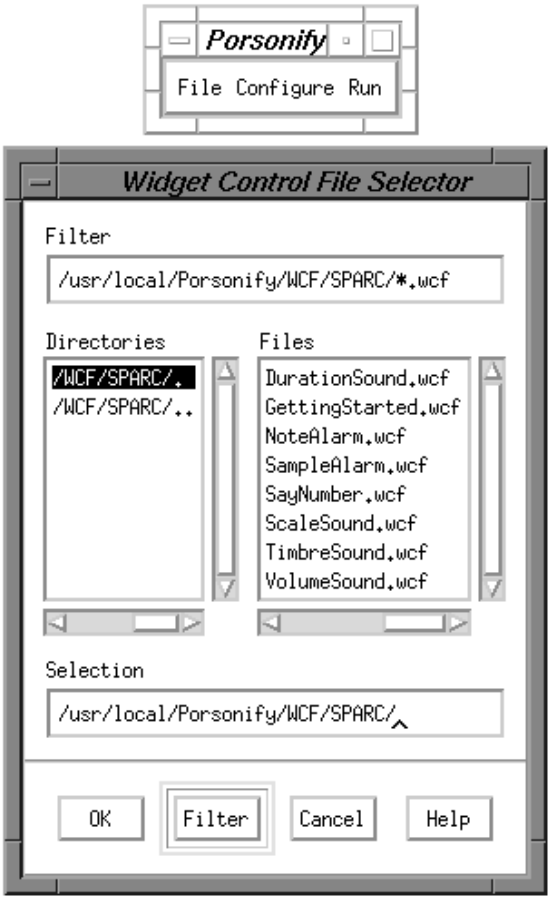

Figure 18: Writing a widget control file after changing a sonification.

4.5 Writing Widget Control Files

If you wish to save the changes to the sonification configuration, it is possible to write a new widget control file. This file can be used with `porsonify` to recreate the sonification with the new characteristics. To do this, select **Write WCF File** from the **File** menu. A file selection dialog box will appear, as shown in Figure 18.

You can type the name of a file in the **Selection** box, or click on a name in the list of files. When you have selected a file name, press the button marked **OK** to write the widget control file and close the file selection dialog. If you change your mind, the **Cancel** button will dismiss the file selection dialog without writing anything.

To run `porsonify` with the new widget control file, first quit the application by selecting **Quit** from the **File** menu. Then restart `porsonify` with the name of the new widget control file as a command line argument.

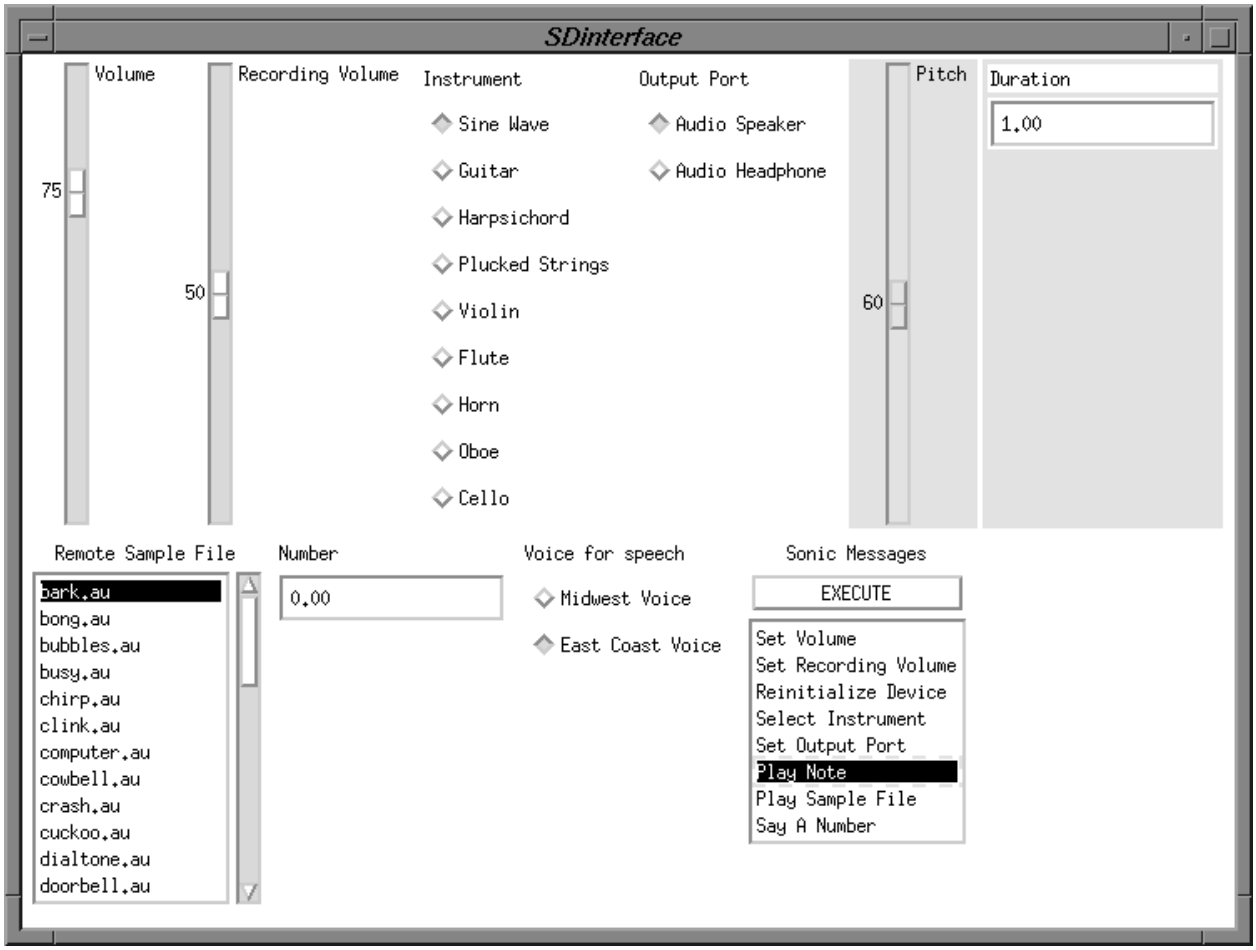

Figure 19: The sdinterface application.

5 Porsonify Application Programs

There are two application programs included in this distribution, intended as examples of toolkit use. They are **sdinterface**, which provides a simple interactive interface to a sound server, and **porsonify**, described in §4, which creates a sonification using a widget control file. This allows one to configure the sonfication and display data from a file.

5.1 Interactive Sound Device Interface: sdinterface

This simple application program connects to the host and port specified on the command line, and creates a window of sliders and buttons that can be used to manipulate the device defaults, and a menu of sonic messages that can be sent to the sound device. It can create an interface to any sound device that has an associated sound device server. Figure 19 shows the panel created by connecting to the port **sparcaudio** on the local workstation, with the following command.

```
sdinterface localhost sparcaudio
```

Command	Description
Set Volume	Sets the play volume of the device.
Set Recording Volume	Sets the recording volume of the device. (Currently this is not too useful. Recording is not yet implemented.)
Reinitialize Device	Reinitialize the device defaults.
Select Instrument	Select an instrument to be used when playing notes.
Set Output Port	Direct output to the speaker or headphone.
Play Note	Play a note for a certain duration. The pitch number 60 corresponds to middle C on a piano keyboard, and each whole number is one key up or down.
Play Sample File	Play a sample file which has been installed on the system.
Say a Number	Say a number (this is done by concatenating sampled spoken numerals).

Figure 20: SPARCserver audio device commands.

To send a command to the server, select a sonic message from the **Sonic Messages** menu (in the bottom right hand corner), and then click on the button directly above it, labelled **EXECUTE**. This action can also be accomplished by double–clicking on the menu selection. When a sonic message menu entry is selected, the valid arguments to that command are highlighted in the other panels. In Figure 19, the **Play Note** sonic message has been selected, highlighting the arguments to that message: **Pitch** and **Duration**. Clicking on the button labelled **EXECUTE** will now play a middle C (pitch number 60) for 1 second (the selected duration).

Adjusting parameters has no effect until the appropriate message is sent to the sound device. For example, if you move the volume slider, this change will not take effect until you send the **Set Volume** command from the **Sonic Messages** menu.

The eight commands the `sparcaudio` device server accepts are summarized in Figure 5.1.

5.2 Creating Sonifications: `porsonify`

As described in §4, the `porsonify` application creates a sonic widget that describes a sonification. It allows you to select a data file to sonify, and to configure the sonic messages that are sent to the sound device. In §4, we describe how to use `porsonify` with a widget control file. In addition to the file `GettingStarted.wcf`, which realizes the sonification in that section, the directory `/usr/local/Porsonify/WCF` contains many other widget control files that produce different kinds of sonifications. You can experiment with these to map data to volume, balance, timbre, duration, pitch, and combinations of these characteristics. For example, to play a note on a Sun SPARCstation as an alarm when a data value is out of range, you might try the following command.

```
porsonify /usr/local/Porsonify/WCF/SPARC/NoteAlarm.wcf
```

Each widget control file contains comments describing its function; browse the files and try different configurations.

```
foobar.cs.uiuc.edu   tg33midi

#   Command comment      MessageId    Arguments
/* Select Instrument */    3            75
/* Play Note         */    0            $0 ( /* Pass Through */ 0 )
                                         ?1.0
                                         $1 ( /* Linear Transform */  1
                                              /* A =               */  2.0
                                              /* B =               */  4.0 )
```

Figure 21: Sample sound control file with fixed, configurable, and variable arguments.

6 Sound Control File Syntax

A sound control file contains all the information necessary to configure the simplest kind of sonic widget. More precisely, it includes:

- the host/port pair needed to connect to the server for an audio device.
- a list of sonic messages and default message arguments (which are fixed, variable, or configurable), and
- optional default transformation functions, and initial configuration information, for variable message arguments.

Figure 21 shows a sample sound control file. The comment syntax is taken from the Bourne shell and C; a `#` indicates a comment that continues to the end of the line, and comments may also be delineated by `/*` and `*/`.

The first two strings (on line 1) specify the Internet hostname (`foobar.cs.uiuc.edu`) and the port name (`tg33midi`) for the target audio device. Because more than one host might support the server that monitors port `tg33midi`, if the hostname is given as `localhost`, the software will attempt to establish a connection to the server on the local host. This makes sound control files for a particular device portable among many hosts. The port name should not be changed, however, because the sonic messages in the sound control file are specific to that logical interface to the hardware (e.g., sonic messages intended for a MIDI synthezizer will generate errors if sent to a SPARC audio device).

Next come a sequence of one or more sonic messages (lines 4–9 in Figure 21). Each message consists of its identifying number, or "message id", and the arguments. Recall that the meaning, range, and default values for command arguments are sent to the application by the server upon connection. The arguments to a sonic message may span multiple lines, and are of three types:

fixed

required, cannot be changed

configurable

can be specified and changed only upon configuration of a sonic widget

variable

the value is recalculated each time the sonic widget executes. Variable arguments are instantiated by applying a transformation function to an input scalar data value, or the indicated element of a data vector, which the sonic widget is "displaying."

The sound device server also maintains a default value for each parameter, called the device default. This value is used in certain situations, described below.

The fixed arguments of sonic messages require no special syntax; they are simply specified in the sound control file. For example, on line 4 in Figure 21 there are two numbers. The first (`3`) is the message identifier for the `Select Instrument` command. The second number, `75`, is the selected instrument. In this sound control file, this value cannot be changed by user configuration. Fixed parameters are especially useful for situations where changing a particular parameter would render that sonification less aesthetically pleasing, or ineffective. For example, consider a sound device with both a master volume and a note volume control. If the note volume is a variable argument, it might be desirable to fix the master volume at some reasonable value so that the changes in note volume can be heard.

Configurable arguments are indicated by a question mark (`?`) preceding the default value. For example, on line 4, the `Play Note` command accepts three arguments: pitch, duration, and velocity (volume). Duration is a configurable argument, given a default value of `1.0` in the file.

Variable arguments do not have default values in the sound control file. They are indicated by a dollar sign (`$`) followed by an index. This index corresponds to an element of a vector. Sonic widgets may "display" scalar or vector data values. If the sonic widget that manages this sound control file maps a vector of data values to sound parameters, then the transformation will operate on the scalar data value corresponding to the specified vector element. If the index is out of bounds, no transformation is performed and the value of that message argument will be the device default, as provided by the sound device server (see §10.2). If the sonic widget displays a scalar data value, the scalar is treated as a single element vector, and all variable arguments except for those of the form `$0` are ignored. The remainder of the variable arguments default to the device default.

A transformation function and its initial configuration may be specified in parentheses directly after a variable argument. The first number in parentheses is the number of the transformation function; see §7 for a list of these. Therefore, on line 5 in Figure 21, the first argument of the `Play Note` sonic message is controlled by transform function 0, or **PassThrough**. This function passes the scalar data value through unchanged as the message argument.

Some transformation functions also have parameters that can be configured in the sound control file. For example, the transformation function for the third argument of the `Play Note` message (velocity, or volume) is **Linear**. This calculates $a * x + b$, where x is the input scalar data value. Values for a and b are read, in that order, from the sound control file, after the number of the transform function. Each transformation function reads arguments in whatever format it expects.[13] If no transformation function configuration information is specified in parenthesis after a variable argument, the device default for that argument is used until a transformation function is selected.

[13] See §7 for more information on configuration formats for the various transformation functions.

7 Transformation Functions

Transformation functions perform simple mathematical operations on scalar data values (that may be vector elements), so that transformed data can either be substituted for variable parameters in a sound control file (see §6), or used to control how other sonic widgets should present the data. Each transformation function has a unique identifying number that is used in sound control files and widget control files.

There are currently five transformation functions.

0. PassThrough

Does not modify the scalar data value, and does not take any arguments. Therefore, the following line, used in a sound control file or a widget control file where transformation configuration information is required, is enough to configure a PassThrough transformation function.

```
0 # identifier for PassThrough transformation
```

1. Linear

Computes $(a * x) + b$, where x is the data value, and a and b are user–specified constants. This is an easy way to scale a range of data values to an equivalent range of device defaults (e.g., to map a stream of numbers into audible musical notes). This transform can also be used to produce a constant value from a variable data stream, by setting $a = 0$ and b to the desired constant.

A sample configuration for a Linear transformation that computes $(10 * x) + 5$ is shown below.

```
1  # identifier for Linear transformation
10 # value of a
5  # value of b
```

2. DataRange

Returns a value based on the range in which the data value lies. Ranges are specified in a file that contains one or more lines of the format:

```
lower  upper   v
```

The values `lower` and `upper` may be integer or floating point, and `v` can be any sequence of non–whitespace characters. The value v is returned if the input data value (x) satisfies the conditions $x >$ `lower` and $x <=$ `upper`. If x does not satisfy these conditions for any line in the file, -1 is returned.

The name of the file must be user–specified. Given a file named `ranges.mf`, that contains a sequence of lines as just described, the following is a sample configuration for DataRange.

```
2         # identifier for DataRange
ranges.mf # the name of the file with data ranges
```

3. Step

Like DataRange, the Step transformation returns a value based on a data range given in a file. However, any values (`v`) present in the file are ignored. Instead, the returned value is calculated by multiplying the number of the line in the file that satisfies the conditions $x >$ `lower` and $x <=$ `upper` by *stepWidth*, and adding to that an *offset*. Both *stepWidth*, *offset*, and the name of the file with data ranges must be user–specified.

Step is useful when you want to choose between different sound events depending on the data. For example, consider the step file:

```
0   100
100 1000
```

If *stepWidth* is 1 and *offset* is 0, the Step transformation will return 0 if the scalar data value lies between 0 and 100, and 1 if the scalar data value lies between 100 and 1000. A MultiSonicWidget can play an alarm sound when the data is in the first range, and map it to a note in a musical scale otherwise.

The *stepWidth* and *offset* can be changed to provide greater flexibity than the DataRange transformation when mapping data to a sequence of non–regular ranges. Because actual parameter values do not need to be written in the data range file, the same file can be used to control variable parameters with different ranges (e.g. volume might range from 0–100 and pitch from 60–90).

A sample configuration for the Step transformation, given a file of mappings called `ranges.mf`, is shown below.

```
3         # identifier for the Step transformation
ranges.mf # the name of the file with data ranges
0         # offset
1         # stepWidth
```

4. Boolean

This uses the data value x and a constant to evaluate a simple boolean expression (x `op` *constant*) and return either true (1) or false (0). The supported boolean operations are: $=$, $<$, $>$, $\leq$, $\geq$ and $\neq$. These are numbered consecutively from 0 to 5. The boolean operation (as an integer) and the constant must be user–specified in a configuration as shown.

```
4     # identifier for Boolean transformation
2     # operation >, which is operation number 2
0     # constant
```

```
BooleanSonicWidget
# begin Boolean transformation function configuration
/* BooleanTransform */  4
/* operation is >   */  2
/* constant =       */  0.8
.
# end of Boolean transformation function Configuration


localhost sparcaudio
#                       MessageId       Arguments
/* Set Output Port   */     4             ?1
/* Set Volume        */     0             ?17
/* Select Instrument */     3             ?3
/* Play Note         */     5             $0 ( /* Linear */ 1
                                                /* A =    */ 100.00
                                                /* B =    */ -10.00 )
                                           ?0.25
.
```

Figure 22: The widget control file for the `BooleanSonicWidget` used in §4.

8 Widget Control File Syntax

Sonic widgets can manage one or more sound control files and an optional transformation function. They can also record the current configurations of these objects in a widget control file, which can later be used to reconstruct the widget.

Figure 22 shows the widget control file that was used in §4 (`GettingStarted.wcf`). As before, a `#` indicates a comment that contines to the end of the line; comments may also be delineated by `/*` and `*/`. A period (`.`) denotes the end of transformation function configuration information or sound control information. Sonic widgets that support multiple sound control files may list their contents one after another in the widget control file. Each sound control file listing must be separated from the next listing by a period. The first non–comment string of characters is the name of the class, or type, of sonic widget that the widget control file describes. The rest of the information in the file depends on the type of sonic widget. In this example, the widget is of type `BooleanSonicWidget`.

If the sonic widget manages a transformation function, the transformation function configuration information must appear following the name of the widget class in the widget control file, and it must be delineated by a period. As described in §3.5, a `BooleanSonicWidget` manages one transformation function and one sound control file. In Figure 22, the transformation configuration information is listed on lines 3–5. The transformation function number, on line 3, is `4`. The Boolean transformation function accepts as user–supplied parameters the identifier of an operation and a

<Sonic Widget Class Name>
[*<transformation function information>* .]
[*<sound control data>* .]+

Figure 23: Outline of widget control file syntax.

constant value. Lines 4–5 are the user–supplied parameters for the Boolean transformation function: the numerical identifier of an operation (`2`) and a constant value (`0.8`). As explained in §7, the operation identifier 2 corresponds to the boolean operation >. This transformation evaluates the expression $x > 0.8$ for each data value x.

The transform function's configuration data is followed by the sound control data describing the messages to be sent when the boolean condition is true. This sound control data is in the format described in §6. It contains the commands for the `sparcaudio` device on the `localhost` to play a note whose pitch (the first argument of the `Play Note` command) is controlled by a Linear transformation function.

Figure 23 outlines the syntax of widget control files. In summary, the name of the sonic widget class is followed by transformation function information (if necessary for configuration of the widget) and sound control data for each sound control file the widget manages. Transformation function information and sound control data must be in the format described in §7 and §6, respectively, and must be followed by a period.

9 Writing New Transformation Functions

Porsonify's existing set of transformation functions can easily be augmented. However, transformation functions are not intended to compute complicated mathematical operations on a single data value. They are necessary merely to promote interchangability of different sound devices and data files in a particular sonification.

For example, consider two data files that each contains information about eight different events. In the first file, these events are numbered 0–7. In the second file, they are numbered 8–15. If the event identifier is mapped to the pitch of a note with the **LinearTransform** function (which calculates $ax + b$, where x is the event number), the same sonification can be used for both files just by adjusting the constant b for each case.

Thus, the transformation functions are designed for simple scaling and range mapping. Like visual displays, sonifications include only those transformations necessary to display the data, not transformations that change or combine elements of the data stream. Manipulations of the second kind are best realized by a general purpose data analysis program.

This section describes how to add a transformation function to Porsonify. Knowledge of C++ and familiarity with WCL and the Motif widget set are asssumed.

9.1 Transformation Function Implementation

Each transformation function is implemented as its own C++ class, derived from the base transformation function class `DataTransform`, in the `CommonClasses` subdirectory. Transformation functions must support methods to read, set, and print their internal configurations (i.e., user–specified constants), and a method to calculate the new value of a variable parameter from a sound control file.

To add a new transformation function, one must do the following:

1. Write the new transformation function, say `NewTransform`, as a derived class of `DataTransform`. The methods in `DataTransform` must be redefined for `NewTransform` in a way that makes sense for the new transformation function.

   Transformation function configurations are always read by the `TransformFunctions` class. The first thing in the configuration printed by the `printConfiguration` method is a unique index that is used by the `TransformFunctions` class to create the appropriate transformation function. That transformation function then configures itself, using the `readConfiguration` method to read the appropriate parameters from the input stream. This index (in the above example, `NewTransform_`is just an enumerator that is defined in `DataTransform.h` as follows.

   ```
   enum TransformFunctions_ {PassThrough_,  Linear_,  DataRange_,
                             Step_, BooleanMenu_, NewTransform_ };
   ```

   Because `NewTransform` is, by assumption, the sixth transformation function, it will be assigned the integer value 5.

2. Edit `TransformFunctions.h` to include `NewTransform.h`.

3. Edit the `TransformFunctions` constructor in `TransformFunctions.cc` to add `NewTransform` as the last element of the `TransformFunctionList`.

4. Edit `TransformFunctions.cc` to return an object of that type when the `SelectFunction` method is called with the value of the enumerator `NewTransform_`.

5. If `NewTransform` has any configurable parameters, write a WCL resource file to prompt for those parameters. This file should be installed with the other such resource files in `/usr/local/Porsonify/app-defaults/Sound`, assuming you have installed the software in the standard location. The **UpdateTransformDefaultCB** callback is used to update transformation function parameters. The name of the X widget is used by the transformation function to determine which parameter to update. The **doneRunningCB** callback should be called when an **OK** button is selected. See, for example, `LinearTransform`, or any of the other transformation function menus.

6. Edit `app-defaults/Sound/TransformFunctionMenu` to offer `NewTransform` as a menu option, and create the new panel when it is selected. This file has comments explaining what changes need to be made.

```
"Set Volume"                setVolume          VOLUME
"Set Recording Volume"      setRecordVolume    RECORDVOLUME
"Reinitialize Device"       reinitialize
"Select Instrument"         selectInstrument   INSTRUMENT
"Set Output Port"           setOutputPort      OUTPUTPORT
"Play Note"                 playNote           NOTE DURATION
"Play Remote Sample File"   playSampleFile     FILENAME DURATION
"Say A Number"              playNumber         NUMBER VOICE
```

Figure 24: Sample sonic message configuration file.

10 Supporting New Sound Devices

To support a new sound device, one must write a sound device server to process requests specific to that device. Any sound device server has three basic layers.

1. Basic functions to access the physical hardware.
2. Interface functions that accept sonic message arguments and call the appropriate device–specific functions.
3. Sonic message configuration, device default, and device parameter files. These files describe sonic messages and their arguments, and they map each message to the corresponding interface functions from the second layer.

The key factor in sound server design is partitioning the functions of the sound device into autonomous units, or sonic messages, that do not mask its capabilities. For example, MIDI supports both note on and note off commands. Although it is convenient to encapsulate this functionality as a single "play note" sonic message that accepts a duration as an argument, it is vital to also support individual note on and note off commands. Otherwise, it would be impossible to gradually change the volume of a note, slowly change its pitch, or play chords.

In Porsonify, the interface functions of the second layer of the sound server belong to their own C++ class. In the current implementation, `MIDISoundDevice` and `SPARCSoundDevice` are examples of such classes, with methods that each accept a list of sonic message arguments, cast each argument into a basic type, and call lower level routines. Classes of this form are hereafter referred to as *sound device classes.*

All the rest of the server details are specified in the message configuration file, device default file, and device parameter file. These files are parsed by a script that generates code to initialize internal device–specific tables. This eliminates much of the tedious work involved in writing a new sound server, and minimizes the probability of errors. The format and meaning of these files is described below. The examples are the actual files used to build the server for the SPARC audio device.

10.1 Sonic Message Configuration File

```
VOLUME          CHAR        "Volume"                75    MINVOLUME   MAXVOLUME
RECORDVOLUME    CHAR        "Recording Volume"      50    MINVOLUME   MAXVOLUME
INSTRUMENT      ENUMVAL     "Instrument"             0    INSTRUMENTS
OUTPUTPORT      ENUMVAL     "Output Port"            0    OUTPUTPORTS
NOTE            CHAR        "Pitch"                 60    MINNOTE MAXNOTE
DURATION        FLOAT       "Duration"             1.0
FILENAME        REMOTEFILE  "Remote Sample File"    0    REMOTESAMPLEFILES
NUMBER          ARRAY       "Number"               "0"
VOICE           ENUMVAL     "Voice for speech"       1    VOICES
```

Figure 25: Sample device defaults file.

The sonic message configuration file, as the example in Figure 24 shows, describes all the messages that can manipulate the state of a sound device. The quoted text in the first column is a description of the message's function. The second column is the name of a C++ method in the sound device class for this particular device. This information is used by a script to generate the code that calls this method when message of this type is received. Case *is* significant. The remaining, optional columns are the expected arguments to the method. They are indexes into an associated device defaults file.

Internally, messages are numbered according to their order in this file. However, the message number is only significant when writing sound control files (see §6).

10.2 Device Defaults File

A device defaults file, as the example in Figure 25 shows, describes the valid arguments to sonic messages for the device. The first column is simply a tag to identify the default. It should correspond to a sonic message argument in the message configuration file. In the above example, `VOLUME` is an argument of the `"Set Volume"` message in Figure 24. The second column indicates the type of the default. The permitted types are shown below.

`INTEGER` integer

`BOOLEAN` 1 (true) or 0 (false)

`CHAR` an ASCII character, represented as a number in the range 0–255

`FLOAT` single precision floating point

`DOUBLE` double precision floating point

`ARRAY` character array, in quotes

`REMOTEFILE` the name of a remote file, which must be a valid UNIX path name

`ENUMVAL` an enumerator of `ENUMTYPE`, which is defined in the device parameters file

The character string in the third column is a description of the default. This is followed by the default value in the fourth column. If there are remaining columns, they are indexes into the device parameters file, and describe upper and lower bounds or a container enumerated type.

```
MINVOLUME           CHAR       "Minimum Volume"     0
MAXVOLUME           CHAR       "Maximum Volume"     100
INSTRUMENTS         ENUMTYPE   "Instrument List"    "SparcInstruments"
OUTPUTPORTS         ENUMTYPE   "Output Ports"       "SparcOutputs"
MINNOTE             CHAR       "Minimum Note"       0
MAXNOTE             CHAR       "Maximum Note"       127
REMOTESAMPLEFILES   ENUMTYPE   "Sample File Menu"   "SampleFileMenu"
VOICES              ENUMTYPE   "Spoken Voice List"  "Voices"
```

Figure 26: Sample device parameters file.

10.3 Device Parameters File

The device parameters file, an example of which is shown in Figure 26, describes static sound device parameters that cannot be changed under software control. This file has a format similar to the device defaults file, except that no bounds are specified, and only types **INTEGER**, **BOOLEAN**, **CHAR**, **FLOAT**, **DOUBLE**, and **ENUMTYPE** are permitted.

The first column is the name of the device parameter. It should correspond to some parameter that was listed as a bound or as an enumerated type in the device defaults file. The second column is the type of the parameter. The string in the third column is a description of the default. Finally, the value is given in the last column. The "value" of an **ENUMTYPE** is actually the name of a file that contains the names of the enumerators, one per line.

10.4 An Example: Writing A New MIDI Instrument Server

Although MIDI instruments have a wide variety of functions, writing a server for a new synthesizer is straightforward. Familiarity with MIDI programming is helpful for reading this section; see [2] and [4] for background.

Low–level functions to access a serial–to–MIDI interface already exist for the SPARCstation, eliminating the first step in writing a new sound device server. These routines are in the file **Shared/MIDIDevice.c**, in the directory where you placed the distribution source code.

The first step is to write a class whose methods accept sonic message arguments and call the appropriate low–level functions. There is a C++ class, **MIDISoundDevice**, that does this for simple MIDI messages. This class supports only primitive MIDI operations that should be accepted, if not understood, by all MIDI devices.

To "customize" this code for a more complicated MIDI device, first create a new class that is derived from **MIDISoundDevice**. An example of this, from this toolkit, is **TG33SoundDevice**. This implements a few specialized commands that the Yamaha TG33 Tone Generator[14] synthesizer supports, and redefines commands that are not implemented in the standard way. For example, this synthesizer has a vector control, which allows you to dynamically control the components of a synthesized sound; to experiment with the application of this feature to sonification, we implemented the methods **setVectorControlXaxis** and **setVectorControlYaxis**. The

[14]Yamaha TG33 Tone Generator is a trademark of Yamaha Corporation.

`selectInstrument` method has been redefined because the TG33 has two voice banks and the server must switch between them.

The next task is to write the message configuration, device default, and device parameter files.

```
"Play Note"                  playNote                PITCH  DURATION  VELOCITY
"Note On"                    noteOn                  PITCH  VELOCITY
"Note Off"                   noteOff                 PITCH  VELOCITY
"Select Instrument"          selectInstrument        INSTRUMENT
"Set Main Volume"            setVolume               VOLUME
"Set Modulation Wheel"       setModulationWheel      MODULATION
"Set Vector Control X-axis"  setVectorControlXaxis   VECTORX
"Set Vector Control Y-axis"  setVectorControlYaxis   VECTORY
"Select Channel"             selectChannel           CHANNEL
"Set Pitch Bend"             setPitchBend            PITCHBEND
"Set Channel Pressure"       setChannelPressure      CHANNELPRESSURE
"Set Sustain (Decay)"        setSustain              SUSTAIN
"All Notes Off"              allNotesOff
"Set Balance"                setBalance              BALANCE
"Pause"                      pause                   DURATION
"Reset All Controllers"      resetAllControllers
"Reinitialize Device"        reinitialize
"Set Pitch Bend Granularity" setPitchBendGranularity PITCHBENDGRANULARITY
"Play Note On Channel" playNoteOnChannel       PITCH DURATION VELOCITY CHANNEL
```

Figure 27: Message configuration file for Yamaha TG33.

```
PITCHBEND       CHAR         "Pitch Bend"        0  MINMIDI MAXMIDI
CHANNEL         CHAR         "Channel"           0  MINCHAN MAXCHAN
PITCH           CHAR         "Pitch"            60  MINMIDI MAXMIDI
VELOCITY        CHAR         "Note Volume"     120  MINMIDI MAXMIDI
INSTRUMENT      ENUMVAL      "Instrument"        0  INSTRUMENTS
SUSTAIN         CHAR         "Sustain (Decay)"   0  MINMIDI MAXMIDI
DURATION        FLOAT        "Duration"        1.0
VOLUME          CHAR         "Synth Volume"     50  MINMIDI MAXMIDI
MODULATION      CHAR         "Modulation Wheel"  0  MINMIDI MAXMIDI
VECTORX         CHAR         "Vector X-axis"     0  MINMIDI MAXMIDI
VECTORY         CHAR         "Vector Y-axis"     0  MINMIDI MAXMIDI
CHANNELPRESSURE CHAR         "Aftertouch"        0  MINMIDI MAXMIDI
BALANCE         CHAR         "Stereo Balance"    2  LEFT    RIGHT
PITCHBENDGRANULARITY CHAR    "Pitchbend Range"   0  MINPITCHBEND MAXPITCHBEND
```

Figure 28: Device defaults for Yamaha TG33.

```
MAXMIDI         CHAR       "Max MIDI Parameter"   127
MINMIDI         CHAR       "Min MIDI Parameter"     0
MINCHAN         CHAR       "Min Channel"            0
MAXCHAN         CHAR       "Max Channel"           15
INSTRUMENTS     ENUMTYPE   "Instrument List"       "YamahaTG33Instruments"
MINPITCHBEND    CHAR       "Minimum Pitch Bend"     0
MAXPITCHBEND    CHAR       "Maximum Pitch Bend"    12
LEFT            CHAR       "Stereo Balance Left"    0
RIGHT           CHAR       "Stereo Balance Right"   4
```

Figure 29: Device parameters file for Yamaha TG33.

Figure 27 is the current message configuration file for the Yamaha TG33. Note that the names in the second column correspond to methods in the `TG33SoundDevice` class (including those methods inherited from the `MIDISoundDevice` class). The arguments given in the file correspond to the parameters expected in the `ValList` which is passed to these methods. These arguments are names defined in the device defaults file shown in Figure 28, that contains the types, descriptions, and server default value, for these parameters. In the fifth and sixth columns of Figure 28 are upper and lower bounds, or container types, if the parameter can only assume some particular range of values. For example, `PITCHBEND` can assume values between `MINMIDI` and `MAXMIDI`, which are indexes into the device parameters file, but `DURATION` has no specific range. `INSTRUMENT` is an element of an enumerated type, which is defined by `INSTRUMENTS` in the device parameters file.

The device parameters file (Figure 29) is similar to the device defaults file, except that these are all constant values that cannot be changed under software control. Note the "value" of the `ENUMTYPE INSTRUMENTS`; it is the name of a file which lists all the available instruments. The rest of the parameters are all constant values of type `CHAR`.

Once the sound device class and device defaults, parameters, and message configuration files to describe the new device, have been written, the rest of the code necessary to generate a sound device server can be automatically generated. To do this, the `ConfigureDeviceServer` script must be executed with these three files and the name of the sound device class as arguments. See `Build/MIDIServer.TG33/Makefile` for an example of how to build the new sound server.

A Distribution Contents

This distribution contains the following files and directories:

app–defaults/

All the WCL (Widget Creation Library) resource files that describe the X/Motif interface to the Porsonify applications.

Clients/

Code for the applications `sdinterface` and `porsonify`.

CommonClasses/

All the code common to applications and network servers.

config

Contains the file that holds the local device menu.

ConfigureDeviceServer

A `perl` script that reads message configuration, device defaults, and device parameters files and generates C++code for a sound device server.

Demo/

Widget control files and a script to run a demonstration program.

Documentation/

The manual for this distribution.

Includes/

Symbolic links to all the include files in other places.

MIDIServer.TG33/

Code specific to the server for the Yamaha TG33 synthesizer.

NetworkAudioAccessDaemon/

Code for the network audio access daemon (*naad*).

SCF/

An assortment of sound control files.

SPARCServer/

Code specific to the Sun SPARCstation audio chip server.

Sample–Libraries/

Sampled digital audio data for the Sun SPARCstation audio chip server.

Shared/

Code shared among various servers.

Transform/

Code specific to the transformation functions.

WCF/

An assortment of sonic widget control files.

Build/

Where to do a make.

The `Build` subdirectory is a copy of the source level directory tree. All message configuration, device defaults, and device parameters files for each sound device are in the `Build` subdirectories for each sound device server, *not* with the source.

B System Requirements

To build the Porsonify package, you must have the following software installed on your UNIX[15] system:

- `g++`, version 1.40.3
- `gcc`, version 1.40
- `gnumake`, version 3.58
- X11R5
- Motif, version 1.1.4
- `perl`, version 3.0
- WCL (Widget Creation Library), version 1.05

This version of Porsonify supports two sound devices: the Sun SPARCstation audio device (under SunOS 4.1) and MIDI synthesizers that are connected to the serial port via a RS232–to–MIDI interface. Specifically, we have provided drivers for two synthesizers: the Yamaha TG33 Tone Generator, and the Yamaha PortaSound PSS–680 MusicStation. However, these will drive the basic functions of any MIDI instrument with only minor modifications. To date, the software has only been installed and tested with these devices on a Sun SPARCstation.

Appendix D describes where to obtain the software and the serial–to–MIDI interface described above.

[15] UNIX is a trademark of AT&T Bell Laboratories.

C Installing Porsonify

This section describes how to build Porsonify and install libraries and daemons.

C.1 Building

See Appendix B to determine if you have the software necessary to build Porsonify. You need approximately 20MB to build everything; a successful build should take about 10 minutes.

Change to the `Build` subdirectory. By default, a make will be performed in every subdirectory listed in the `Subdirs` file. If you do not support all the devices, you will probably want to remove the lines corresponding to those servers from the `Subdirs` file. For example, if you do not support MIDI, you can delete `MIDIServer.PortaSound` and `MIDIServer.TG33`.

You should edit `Makefile.defines` to select a pathname for the root of the installed Porsonify software hierarchy. By default, this path is `/usr/local/Porsonify`. It might be useful to look over the `Makefile` in each subdirectory that is listed in the `Subdirs` file after you have decided what to build. Each daemon should have a unique port name, which is defined in its `Makefile`; you might want to edit these definitions.

Finally, type `gnumake` in the `Build` subdirectory.

C.2 Installation

To install the Porsonify system in the directory specified in the `Makefile.defines` file, type `gnumake install` in the `Build` subdirectory.

To execute the examples in §2 or §4,

- the ports for all daemons should be known services,
- the network audio access daemon (*naad*) should be running on each machine from which you will be accessing a sound device, and
- the sound server for each sound device must be running on the machine which manages that device.

To ensure that the ports for all daemons (included in this release are *naad*, *sparcaudiod*, *tg33midid*, and *pssmidid*) are known, they must be registered as TCP/IP services. For example, if you have used the default port names, the following entries in the master `/etc/services` file will work. This will probably require the help of a system adminstrator.

```
naad            1666/tcp                        # network audio access
midiaudio       1667/tcp                        # midi audio server
sparcaudio      1668/tcp                        # sparc audio server
```

To start the sound servers, assuming you are using the default path `/usr/local/Porsonify`, execute the command

```
/usr/local/Porsonify/lib/startSoundServers
```

To verify that everything has been installed correctly, you can try the demonstration program (see §2). Note that the application programs must be able to locate the application defaults files necessary to create the X/Motif interface. These files describe the layout and appearance of the application interface. The easiest way to do this is by setting the environment variable `XAPPLRESDIR` to the directory where these files are kept on your system. For example, if you have installed the software in `/usr/local/Porsonify`, you can type, in `csh`,

```
setenv XAPPLRESDIR /usr/local/Porsonify/app-defaults
```

If this variable is not set, the Porsonify software will look for these defaults files in this directory. Problems will occur if it is set incorrectly.

D Obtaining Equipment and Software

The RS232–to–MIDI interface (the MIDIator[16] Model 114) used is from

Key Electronics
7515 Chapel Avenue
Fort Worth, TX

1-800-533-MIDI

At this writing, the MIDIator lists for $229.95.

The X Window system is copyrighted but freely distributed. It is available via anonymous ftp from several sites, including `ftp.uu.net` (Internet address 137.39.1.9). WCL (Widget Creation Library) is available as part of the X11R5 distribution.

If your hardware vendor is a member of the Open Software Foundation (OSF), they may be able to provide Motif binaries for your machine. Source licences for Motif must be obtained directly from OSF.

The `gcc` and `g++` compilers, as well as `gnumake` and `perl`, are available from the Free Software Foundation (FSF) by anonymous ftp on the host `prep.ai.mit.edu` (Internet address 18.71.0.38) in the `/pub/gnu` directory. Questions about FSF software should be addressed to `gnu@prep.ai.mit.edu`.

The Yamaha TG33 Tone Generator used lists for $595. It offers 128 preset voices, and an extensive library of sampled and FM waveforms which can be combined to create new voices. We selected the TG33 because of its wide variety of voices, and flexibility controlling and editing new sounds. Its excellent support for MIDI was also considered; all functions can be accessed with MIDI commands.

[16] MIDIator is a trademark of Key Electronics.

References

[1] Bly, S. *Sound and Computer Information Presentation.* PhD thesis, University of California, Davis, 1982.

[2] DeFuria, S., and Scacciaferro, J. *MIDI Programmer's Handbook.* M & T Books, Redwood City, CA, 1989.

[3] Francioni, J. M., Albright, L., and Jackson, J. A. The Sounds of Parallel Programs. In *Proceedings of the Sixth Distributed Memory Computing Conference* (1991), IEEE Computer Society Press, pp. 570 – 577.

[4] MIDI Manufacturers Association. *MIDI – Musical Instrument Digital Interface, Specification 1.0.* International MIDI Association, Los Angeles, CA, 1988.

[5] Rabenhorst, D. A., Farrell, E. J., Jameson, D. H., Linton, T. D., and Mandelman, J. A. Complementary visualization and sonification of multi–dimensional data. In *Proceedings of the SPIE, Conference 1259, Extracting Meaning from Complex Data: Processing, Display, Interaction* (1990), pp. 147 – 153.

[6] Reed, D. A., Olson, R. D., Aydt, R. A., Madhyastha, T. M., Birkett, T., Jensen, D. W., Nazief, B. A. A., and Totty, B. K. Scalable Performance Environments for Parallel Systems. In *Proceedings of the Sixth Distributed Memory Computing Conference* (1991), IEEE Computer Society Press, pp. 562 – 569.

[7] Scaletti, C., and Craig, A. B. Using Sound to Extract Meaning from Complex Data. In *Proceedings of the SPIE, Conference 1459, Extracting Meaning from Complex Data: Processing, Display, Interaction II* (San Jose, CA, 1991).