VOX Audio Server

Olivetti Research California

Version 1.0
January 1990

Revision History:

Author(s)	Date	Rev.	Comments
Arons	Dec 89	1.0	Integration of documents.

Contents

Chapter 1

Documentation, Release, and Installation Notes

Revision History:

Author(s)	Date	Rev.	Comments
Arons	Oct. 17 89	0.9	Created.
Arons	Dec. 27 89	1.0	Clean-up, additions...

1.1 Introduction

This document details the current release of the VOX Audio Server, related support and client software, etc. The VOX Audio Server software is an experimental prototyp
risk! The VOX software and this documentation are believed to be the correct
however there will certainly be errors, oversights, and omissions. Keepin
synchronized with the actual VOX Audio Server is a particularly difficult ta

1.2 The Documentation

Each of the chapters in this manual was originally a separate document, th
may have its own introduction, appendices, acknowledgements, etc. There
index, table of contents, and bibliography for the entire manual. The indivi
been combined here to give more coherence to the documentation. The bull
written by the person(s) first named on the revision history page each chap
no better way to give credit to the original authors given the current docum
we are using).

There is a single index for the entire document, however the bulk of the inde
from a single chapter. The style of the chapters on Mercury, IPC package, e
consistent with the rest of the manual, as these chapters are due to be comp

All documentation is currently written in LaTeX format, and can be found in `$VOX_DIR/doc`. Printable postscript and text versions of all documents can be found in `$VOX_DIR/doc/ps` and `$VOX_DIR/doc/text`.

In addition to a single combined document, the chapters can be made as individual documents. A common latex header file called "voxheader.tex" is included by all the documents. When making the single document, this file is set to zero length. When making individual chapters, this file is copied from "voxheader.template". Note that the environment variable `$TEXINPUTS` must point to the `src/common` directory so that the header files and bibliography files can be found.

1.3 Workstation Environment

The VOX Audio Server is intended to be a device-independent software platform, running on a variety of hosts and with a range of speech and audio peripherals.

The system was developed on, and currently runs on:

1. Olivetti M380 workstations running an experimental version of MACH/BSD4.3 Unix/i386[1]
2. Sun 3/60 workstations running SunOS 4.0
3. Sun 3/60 workstations running MACH (4.3 BSD)
4. Compatibility with Sun 386i workstations running SunOS 4.0 is expected for this release, but has not been tested.

All current prototyping work is proceeding on the M380's because of PC-BUS devices we wish to use with VOX.

The prototype applications (sound editor, answering machine, mixer control) are all built upon the X Window System and use the Xt toolkit and the Xaw Athena interface widgets.

1.4 Audio Hardware Environment

VOX currently supports the following audio hardware:

1. VBX/I board from Natural Microsystems. A single audio channel PC-BUS board that provides play, record, and telephone capabilities.
2. VideoTelecom Voice System 100. A PC-BUS board providing full-duplex speaker-phone capabilities.
3. Yamaha DMP11 Digital Mixing Processor. A 8in 2out audio mixer. Note that this is controlled via a Hinton MIDIC.
4. Akai DP2000 Audio Video Digital Matrix Patch Bay. A 16 16 audio+video crossbar switch. Note that this is controlled via a Hinton MIDIC.
5. Akai DP3200 Audio Digital Matrix Patch Bay. A 32 32 audio crossbar switch. Note that this is controlled via a Hinton MIDIC.
6. Miscellaneous microphones, speakers (we are using small powered speakers from Fostex), cables, etc.

In many ways, the VBX board is the key component of the system. Many useful working prototypes can be built just using the features of the board. The other devices are useful, but are not necessary.

[1] An earlier version of this software ran on Olivetti M380 workstations running System V Unix. Current or future compatibility with System V is not assured.

We are investigating hardware platforms and software extensions for speech recognition and text-to-speech synthesis.

Note that several items on the list are MIDI controlled devices. We interface the workstations to these devices via a serial port with a Hinton MIDIC RS-232 to MIDI converter. (See section 1.11.1 for information regarding controlling the Akai crossbar switches.)

While the configuration of VOX is quite flexible, we have adopted the following "standard" configuration for interconnections to each workstation's crossbar switch:

Low crossbar connections (1/4 inch):

	0	1	2	3	4	5	6	7
In	Mic-0	Mic-1				VBX	CD-L	CD-R
Out	DMP-0	DMP-1	DMP-2	DMP-3		VBX	skprL	spkr-R

High crossbar connections (XLR):

	8	9	10	11	12	13	14	15
In	tie	tie	tie		EC	EC	stereoL	stereoR
Out	tie	tie	tie		EC	EC		

1.5 ORC Standard Audio Workstation

The standard ORC audio workstation consists of an Olivetti M380 workstation (XP1, XP3, XP5, etc.) running the i386 version of the MACH operating system.

In addition to the audio hardware listed in section 1.4, the M380 workstation should be typically equipped with:

minimum 150MB hard disc drive (300MB drive is strongly recommended)

As much memory as possible. Preferably 16MB, with a minimum of at least 8 MB (12MB is current maximum on systems running the X Window System)

Quad port serial board (for Hinton boxes)

BLIT card, monitor, serial mouse (optional to run the X Window System)

Please check with the Operating Systems Project at ORC to see exactly which devices are supported by the MACH port (heed this warning!).

1.6 Directory Layout

All files are stored under the `$VOX_DIR` directory. The major directories in this release include:

```
./bin                   binaries/executables
./client/answmach       answering machine source
./client/sedit          audio editor source
./config                configuration files
./doc                   documentation
./lib                   libraries
./server                server source
./support/emacs
./support/environ
./support/make          makefile utilities
./support/utils
./tools/imake           imakefile utilities
./tools/makedepend      imakefile utilities
./tools/mercury         RPC generator source
./unixdev/vbx           VBX/I driver source
./unixdev/vtc           VideoTelecom driver source
./voxdev                VOX devices source
```

Under each of the major directories, there are typically one or more of the following subdirectories. *Note* that there is an exception in the `tools/makedepend` directory, the `obj` directory is a sybolic link to the parent directory (i.e. `tools/makedepend/obj` is a link to verb+tools/makedepend+).

```
./RCS                   RCS files
./asm                   assembler
./bin                   executables
./include               .h files
./lib                   libraries
./obj                   .o files
./sintf                 Mercury files
./src                   .c files
```

The documentation directories have a similar structure, but are not detailed here.

1.7 Installation Procedures

You will need to be logged in as "root" for most of these operations.

On all systems, there must be temporary IPC directory. Note that the current location of the `.ipc` directory is in `/tmp/.ipc`. On some systems, `/tmp` is cleared automatically when the machine is rebooted, so this directory may inadvertently vanish.

Create a new user "vox", `chown vox ~vox`.

For execute-only systems, install configuration files in `~vox/config`. Copy a `.login` file to get appropriate environment variables set.

```
mkdir /tmp/.ipc
chmod 777 /tmp/.ipc
```

The VOX server must be registered in `/etc/services`. We are currently using the following entries:

```
vox                3131/tcp
vox                3131/ip
sound-edit         3132/tcp
sound-edit         3132/ip
answer_machine     3135/tcp
answer_machine     3135/ip
```

Note that currently the DMP11's assume that they are talked to via a Hinton MIDIC box on specific tty ports.

1.7.1 M380 installation

Installing the VBX board:

- Set the VBX Board to IRQ3, by installing the jumper on pins W6
- The following commands configure MACH for the VBX board:

  ```
  cd /dev
  mknod ln0 c 8 0
  chmod 666 ln0
  ```

- Install `dial.bin` and `vox.bin` in `~vox/bin`.

Installing the Quad Port serial board:

- Set switches on quad port board to:
  dn dn dn, dn dn up, dn up dn up
- The following commands configure MACH for the quad port board:

```
cd /dev
mknod qd00 c 13 0
mknod qd01 c 13 1
mknod qd02 c 13 2
mknod qd03 c 13 3
ln -s qd00 tty00
ln -s qd01 tty01
ln -s qd02 tty02
ln -s qd03 tty03
chmod 0666 qd??
```

Installing the VTC board:

1.7.2 Sun 3/60 Installation

```
cd /dev
mknod ttyd0 c ? ?
mknod ttyd1 c ? ?
chmod 0666 ttyd0 ttyd1
```

1.8 Environment Variables

The following environment variables are used by VOX and client applications:

1. `$VOX_DIR` points to the root of the VOX source code tree. The default is `~vox`.
2. `$VOX_BIN_DIR` is a directory for binary files needed by individual devices and drivers. For example, the VBX board needs two binary files that are stored under this directory. The default is `$VOX_DIR/bin`.
3. `$VOX_CONFIG_DIR` is where VOX looks to find its run-time configuration files. The default is `$VOX_DIR/config`, though typically it will point to a machine or user dependent directory such as `$HOME/config`.
4. `$VOX_SOUND_DIR` where sound files are stored by the server. The default is `$VOX_DIR/sounds`, though it will typically point to a user specific directory, such as `$HOME/sounds`.
5. `$VOX_HOST` is the hostname of the desired VOX server. This variable is *not* used by the server, but should be used by client applications. Ideally the use of this variable should be transparent to the application writer and VOX startup code that uses the variable should be part of library or toolkit.

On the server machine it is only necessary to set `$VOX_DIR`, though it may be helpful to set `$VOX_CONFIG_DIR`. On a client machine it is necessary to set `$VOX_HOST`, but `$VOX_SOUND_DIR` may also be set.

1.9 Things That May Change

The VOX software is still undergoing continued refinement and development. It is our intent that the interfaces described in the documentation will remain unchanged, even though various pieces of VOX software may change underneath. The following modules are particularly susceptible to being replaced or significantly modified:

1. Binding's IPC
2. Binding's Mercury RPC generator
3. Binding's Threads
4. Everything else in the `utils` directory (bit and df set routines, memory allocator, etc.)

1.10 Reference Files

Much of the server code is quite complicated and is undergoing revision and cleanup. Many of the core functions such as play and record are complex due to the handling of multiple buffers, a large interface to the driver, etc. In adding a new type of device to the server, it would be wise to use a simpler device for reference, such as the fader, source, or filter LAUDs.

1.11 Hardware Modifications

This section details simple hardware modifications that are needed, or possibly helpful, in using the audio hardware outlined above.

1.11.1 AKAI Crossbar MIDI Interface

The Akai crossbar switches are not *really* MIDI devices in that they do not follow the MIDI specification with regard to data format or electrical connections. To use Akai hardware with a MIDI controller, you must make a special cable[2] as shown in Figure 1.1. The connector at the top is a

[2]This information was supplied by Bill Buxton of EuroPARC.

4-pin XLR, the connectors at the bottom are standard MIDI. The Radio Shack 5-pin DIN cables that we have used have had pin 4 red and pin 5 black.

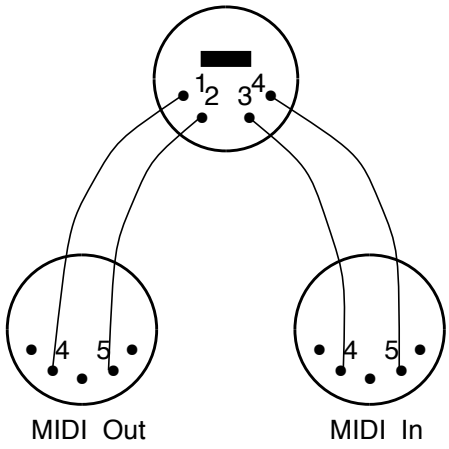


Figure 1.1: Akai to MIDI wiring.

1.11.2 Line Level Interface to NMS Board

The VBX/I board does *not* have a line level audio in connector. You can get line level in functionality by building a small circuit[3] as shown in Figure 1.2 and plugging into the phone connection of the VBX board.

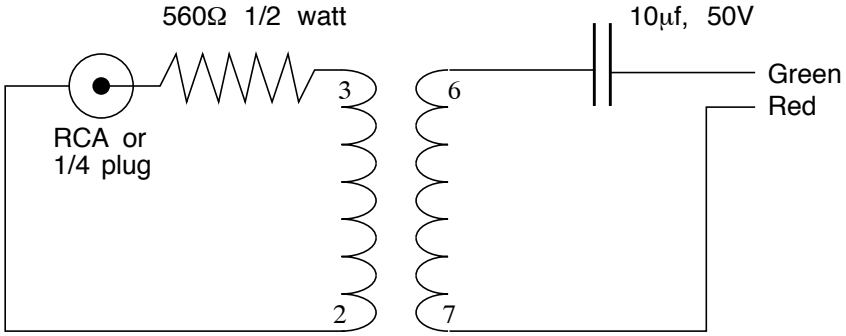


Figure 1.2: VBX line level input hardware.

Note that the capacitor is unpolarized (e.g. Radio Shack 272-999). The transformer is a 600-600 "wet" telephone transformer. Sample part numbers include: Midland-Ross Midcom 671-0346, Pan-Magnetics TTC-122, Prem SPT-109, Stancor TTPC-15, James Electronics Model 13645. When looking at the bottom of the transformer, with the keyway in the upper right hand corner, the pins are numbered 1–8 starting at the keyway going clockwise.

[3]This is a simplified version of information that was supplied by Natural Microsystems. The information provided by NMS also included line level output.

1.12 Acknowledgements

The VOX Audio Server was designed by Barry Arons, Carl Binding, Keith Lantz, and Chris Schmandt. Chris Lauwers provided much useful input in early design stages. The bulk of the server software was written by Carl Binding. Pieces of drivers, applications, and extensions to VOX were written by Barry Arons, Pehong Chen, Keith Edwards, Duane Northcutt, and Sanjay Manandhar. Duane Northcutt has undertaken the herculean task of integrating and cleaning up the code from all these parties and engineering the the first major software release.

Chapter 2

Design Specification

Revision History:

Author(s)	Date	Rev.	Comments
Arons, Binding, Lantz, Schmandt	Aug. 1 88		Original
Binding/Arons	89	0.8	Revision
Arons	Oct. 17 89	0.9	Reformat, cleaned up, rewrote.

2.1 Introduction

During the last decade we have experienced an increased availability of window systems based on interactive pointing devices and high-resolution bitmap screens. As a result, applications have started to become graphical and highly interactive in nature, relegating the character-oriented, teletype-style of interaction to history. Faced with the success of introducing a new medium, namely graphics, into human-computer communications, we now have the opportunity to incorporate yet another medium: the world of audio.

Much research and development work has been performed in the area of low-level speech and sound processing. Sophisticated encoding algorithms reduce communications bandwidth and still produce intelligible output. Text-to-speech synthesis has reached a level of maturity that makes it acceptable for a wider audience. Recognition of human speech can be performed reliably enough in certain situations to make the technique interesting for further investigation and applications.

However, it still is an open question how these techniques are best used in a day-to-day computing environment. The purpose of our work is therefore to specifically address the integration of voice and audio processing techniques into a wider range of applications. To that effect we propose a software architecture[1] that enables the application programmer to efficiently, conveniently, and flexibly use the audio-related resources provided by a network-transparent audio[2] server, called the VOX Audio Server.

Sample applications that our architecture must support include:

Answering machine: A simple telephone answering machine that plays and records sounds and connects to the telephone network. The answering machine should also have the ability to provide a graphical interface as well as remote access to messages over the phone using Touch-Tones or speech recognition.

Teleconferencing: In a teleconferencing environment our architecture must support access to the telephone network and the transmission of voice in real-time. Future extensions may include support for digital transmission of voice and audio, sychronization of audio, video, and data, etc.

Audio editor: An audio editor that supports coarse-grained speech editing. By that we mean cutting and pasting of speech at the phrase level, not at the word, syllable, or sample level. We believe that such a voice editor with a relatively simple audio user interface is flexible enough for use in an office environment.

Voice annotation: A text editor supporting voice annotations that are iconically represented and playable upon request.

[1] We shall use the term *architecture* as a synonym of *model*; since any architecture is the realization of an underlying model, it necessarily reflects that model and thus can be considered synonymous.

[2] Audio is meant to include voice and sounds, at least at toll line quality.

Office automation: In addition to voice annotation mentioned above, our architecture can accommodate various office applications such as proof-reading using text-to-speech synthesis, computer-assisted training using text-to-speech synthesis and speech recognition, or presentation support tools which are voice controlled or involve pre-recorded audio.

Acoustic effects: We wish, for instance, to play one sound on the left speaker, then play another sound on the right speaker, and finally play both sounds simultaneously on the right and left speaker.

Based on these prototypical applications, we can summarize our design constraints:

Sharing: The model must support the sharing of audio resources by multiple applications. Additionally, applications must be able to gain *exclusive* access to critical audio resources for limited periods of time.

Routing: The model has to enable applications to create *dynamic* routings between audio components. For example, in a conversational telephone answering machine it is desirable to rapidly and conveniently switch from a speech recognizing configuration to a sound recording configuration since both activities are useful to gracefully handle an incoming call.

Real-time behavior: The architecture must address the issue of real-time handling of audio events. The model must support a mechanism that minimizes the overhead of processing audio requests and events at time-critical moments, which can be achieved by decoupling the "real-time server" from the client.

Device independence: The architecture should shield the clients from the idiosyncrasies of particular audio devices. However, the great variety of audio equipment implies that this goal can only partially be achieved, in particular when considering the possibility of flexible audio-routings.

Expandable architecture: The architecture should leave room for unforeseen uses of audio, new types of audio, speech, or telephony devices, and possibly the integration of video.

It should be obvious from the applications and constraints listed above that our intention is not to perform ground-breaking research in the areas of speech encoding, text-to-speech synthesis, or speech recognition. Rather, we intend to incorporate existing technology into an integrated and easy-to-access software environment in order to study the potential of audio and voice in advanced multimedia user interfaces.

The remainder of this document details the software architecture of an audio, voice, and telephony server that satisfies the above design constraints. We call this the VOX Audio Server, or simply VOX. We propose a set of functions that support a broad range of voice, audio, and telephony applications, including some simple file system abstractions for storage of digitally recorded sounds. The architecture attempts to decouple the client from the hard real-time constraints of audio processing by pre-allocating critical resources, by pre-fetching data, and by the use of a fast sound

storage sub-system. Although the total amount of processing remains the same, this increased "preparedness" reduces the execution latency at time-critical moments.

An important aspect of our proposal is resource management of audio devices shared between a set of clients. Audio resource management is quite distinct from resource management in a graphics environment, primarily due to the temporal behavior of audio, and the server tries to address these issues.

We also have paid special attention to audio routing and mixing. Research environments often have a wide variety of audio devices typically interconnected via a large manual patch bay, or set of patch bays. VOX attempts to unify this complex hardware configuration into a coherent software environment. As a software architecture the VOX Audio Server is not a panacea that will miraculously solve all audio control and interconnection problems. However, *if* the devices can be interconnected under software control and they support a small set of common functions, *then* VOX will be able to breadboard the devices together into a useful circuit for a clients' use.

Finally, we believe that our model can easily be adapted to yet another medium, namely video. This issue is discussed further at the end of this proposal.

The remainder of this document describes our model in detail and is organized as follows. Sections 2.2 and 2.3 provides a brief overview of the intended target environment and general overview of VOX. Section 2.4 defines the terms and concepts of the model. Sections 2.5, 2.6, 2.7, and 2.8 describe the individual VOX concepts in greater detail. A collection of examples to illustrate the architecture is given in section 2.10. Section 2.11 describes related work in the areas of audio, voice, and telephony, and user interface software and shows how these are related to our proposal.

2.2 Target Environment

Imagine an environment consisting of powerful workstations with high-resolution bitmapped displays *and* a variety of special peripherals (such as speech-processing units, digital scanners, transparent tablets, real-time video processors, and videodisc players), all employing shared file servers, back-end computation engines, and other specialized facilities—all workstations are interconnected by high-speed digital networks.

From the point of view of a user sitting in front of one of these workstations, the following functionality is available (beyond what exists in contemporary workstation environments):

> Voice can be used to interface to standard services such as electronic mail, voice mail, and telephone management. It can also be used to interact with hands- or eyes-busy applications such as CAD tools, instrument control, or document annotation. More generally, voice can be used for application invocation and menu selection.

> Video can be used for mail, real-time teleconferencing, tutorials, and display of scientific data. It can be displayed coincident with conventional windows and graphics.

From an architectural point of view, this functionality is made possible by several key hardware and software components. With respect to hardware, each workstation is equipped with an "audio device" capable of playing and digitally recording sounds stored in the standard workstation file system. This audio hardware also interfaces to one or more conventional telephone lines, providing the user with wide area telecommunication access. Similarly, each workstation is equipped with a video camera and a display/video board capable of supporting multiple live video windows—in addition to its support for graphics.

Functions that can not effectively be implemented on a per-workstation basis are available as back-end services. Such services may include speaker-independent continuous speech recognition, text-to-speech synthesis, audio/video routing and mixing, access to videodisc players and VCRs, and standard broadcast television.

Regardless of where they reside, these new services are implemented in a manner similar to contemporary window systems, that is, as "server processes". Whether the server resides on a workstation or a back-end server, the interface to it remains the same. Thus, access to servers is *network-transparent*.

The remainder of this document focuses on one of these servers, namely the "audio server" which we call VOX. The goal is to provide a consistent software interface to a broad range of audio devices. While we assume that, typically, there will be a separate instance of VOX on each workstation controlling the local audio hardware. This is similar to contemporary window servers controlling the local graphics device as well as the keyboard and mouse. The architecture will also support centralized services by using audio hardware on a dedicated server machine, or a portion of the hardware on a workstation.

2.3 Overview

Our model is based on a network-transparent audio server with an architecture analogous to modern window systems. Multiple client processes are connected to a server process that provides various services to perform audio operations such as recording and playback of sounds, speech recognition, and text-to-speech synthesis. Our envisioned computing environment consists of workstations with each runing one instance of the audio server process to control the workstation's audio resources. Additional servers may reside on dedicated server machines. Client processes can be running on any host in the network.

To support the possibility of dynamic routing between audio devices, we use the paradigm of assembling audio circuits in a manner similar to electronic hardware. That is, lower-level components are grouped into higher-level components of increased functionality. We call the lowest

level building block a *LAUD*[3] (**L**ogical **AU**dio **D**evice) which are assembled into *composite* LAUDs, called *CLAUD*s. Each LAUD has a set of *audio ports* that are connected ("soldered") together when creating a CLAUD and is associated with a *device* that implements the actual audio activity. In essence, clients assemble LAUDs to serve a specific purpose and CLAUDs can be considered as a single logical entity. All of these abstractions are provided by the VOX server and accessed from clients distributed throughout a network.

All LAUDs can be considered to be extensions of the base type *LAUD*. That is, every LAUD supports a common set of operations. Examples of primitive LAUDs include abstractions for playback and recording, a phone abstraction, as well as loud-speakers and microphones.

Each composite LAUD (CLAUD) specifies which audio devices shall be used at run-time for the execution of audio requests. Resource sharing is achieved through the concept of mapping and unmapping CLAUDs to devices. It is only while a CLAUD is mapped that it can actually execute audio activity. For example, if the VOX server receives a request from CLAUD-2 to use a device that is currently mapped by CLAUD-1, CLAUD-1 is preempted (suspended) and unmapped, then CLAUD-2 is mapped to the device. However, a client may lock a CLAUD's devices for its exclusive (non-preemptable) use.

We envision a more sophisticated device arbitration scheme involving an *audio resource manager* (the *Monitor CLAUD*), a specialized VOX client with greater privileges than "average" clients. In that scenario, only the audio manager client will be allowed to perform map and unmap calls, while regular clients may only express their desire to be mapped. We already provide a set of operations for CLAUD management that are of use to a possible audio manager. In particular, there are calls to map and unmap a CLAUD to its devices and calls to temporarily suspend or resume a CLAUD's operation. There are also calls to lock or unlock the devices being used by the CLAUD (see section 2.5.3).

The audio resource manager can be seen as the equivalent of a window manager for a window server. In fact, both managing entities can be considered part of the more general *workstation manager*. Figure 2.1 represents the overall view of such an environment. The window server and audio server are used by a set of clients. The workstation manager (WSM) is merely a client with greater privileges and uses the same communication model with the servers as a regular client. The *Monitor* CLAUD in figure 2.1 plays a similar role to the *root* window in a window server. That is, it keeps track of the server's internal state and notifies interested clients about events occurring within the server.

Input and output from CLAUDs is based on an event-driven scheme. The client submits *output requests* to a CLAUD and may receive *input events* from a CLAUD[4]. For example, the client may submit play and record requests, and receive tokens from a speech recognizer.

Output requests can be *prepared* in advance. To that effect, a CLAUD embodies a *queuing* abstrac-

[3] Pronounced *loud*.

[4] *Output* is data sent from the client to the server; *input* is data sent from the server to the client, similar to I/O handling in window systems.

voxandx.ps

Figure 2.1: Audio and Window Server Architecture

tion, which serves as a buffer for client requests. When enqueuing requests, the server attempts to prepare the request as much as possible. This may involve the opening of a sound file, prefetching an already recorded sound, or establishing the state of a speech recognizer. All these activities can be executed *before* the actual servicing of the request takes place and reduce the execution latency of a request at time-critical moments. (This scheme, providing real-time behavior in time-shared operating systems, is based on previous audio and telephony systems built by two of the authors.)

Amongst the advantages of our model is the possibility to flexibly interconnect various audio components, the feasibility of an interactive graphical editor to create various audio configurations, and the extensibility of the model to incorporate video. We also achieve effective decoupling between application and server through the use of the queuing and preparing of audio requests. And, lastly, the model accommodates an environment of applications distributed over a local area network.

A major disadvantage of the model is its low level. The flexibility of the low-level LAUD and audio-ports abstractions are thus offset by an increased tediousness of client software. We hope, however, to provide a toolkit that will hide the idiosyncrasies of the VOX server[5].

2.4 Terminology

In this section we define the concepts that have been introduced in section 2.3 more precisely.

The principal element of abstraction in our architecture is the **L**ogical **AU**dio **D**evice (LAUD) which is a virtual audio building block or component. A LAUD can contain *audio ports*, a *control port*, and a *data port*. It may represent a single audio device (e.g. speaker, microphone, amplifier), or may be a part of some larger audio device (e.g. the play or record function on an audio

[5]This is similar to the roles of window systems and toolkits for user interface software.

card). The number and type of audio ports and a minimal set of software messages supported by a LAUD are predefined (i.e. known at compile time), so that the logical devices can be controlled in a device independent manner.

Flow of audio data between LAUDs is established through the inter-connection of *audio port*s. An audio port on a LAUD can be connected to another audio port with compatible electrical characteristics (e.g. microphone level, line level, etc.) thereby creating a *connection*. These connections can be dynamically enabled or disabled by a client providing a primitive switching capability.

A *composite* LAUD or *CLAUD* is a hierarchical composition of LAUDs. Note that the name CLAUD is not really needed (it is of type LAUD), but we will maintain the distinction such that a LAUD is a low level building block, and a CLAUD is a higher-level abstraction built by a client. A CLAUD is dynamically configurable; its internal routing can be changed by rerouting connections between its audio ports. At the hardware level, LAUDs are mapped by the server onto a set of *devices*. The server performs the necessary mapping of LAUDs to devices at run-time.

One output and one input control *queue* are associated with each LAUD hierarchy (CLAUD). Each audio queue is processed in a first-in, first-out fashion. Thus, for two output requests to be serviced simultaneously, two LAUDs with separate audio queues are required. The output queue, in particular, serves as a decoupling mechanism between client and server. When output requests are enqueued to a suspended queue, the server prepares these requests which can then be executed with low latency once the audio queue's execution is resumed.

Audio data can be stored digitally in *sounds* residing on some storage medium and accessible through a hierarchical name space. Sounds can be recorded and played using the appropriate audio devices of the server[6].

Figure 2.2 graphically shows some of these abstractions. The most primitive layer is the raw device layer, upon which we build logical audio devices, i.e. LAUDs. These, in turn, are composed into more specialized CLAUD abstractions. Clients then use these abstractions to service their audio needs.

In figure 2.2, LAUDs L3 and L4 share device D3. CLAUD C1 contains CLAUD C2 and (indirectly) uses LAUDs L1, L2, and L3. Similarly, CLAUD C3 uses LAUDS L4 and L5. Output requests to the LAUDs are synchronized by the use of the single output queus of the top-level CLAUD.

2.5 LAUDs

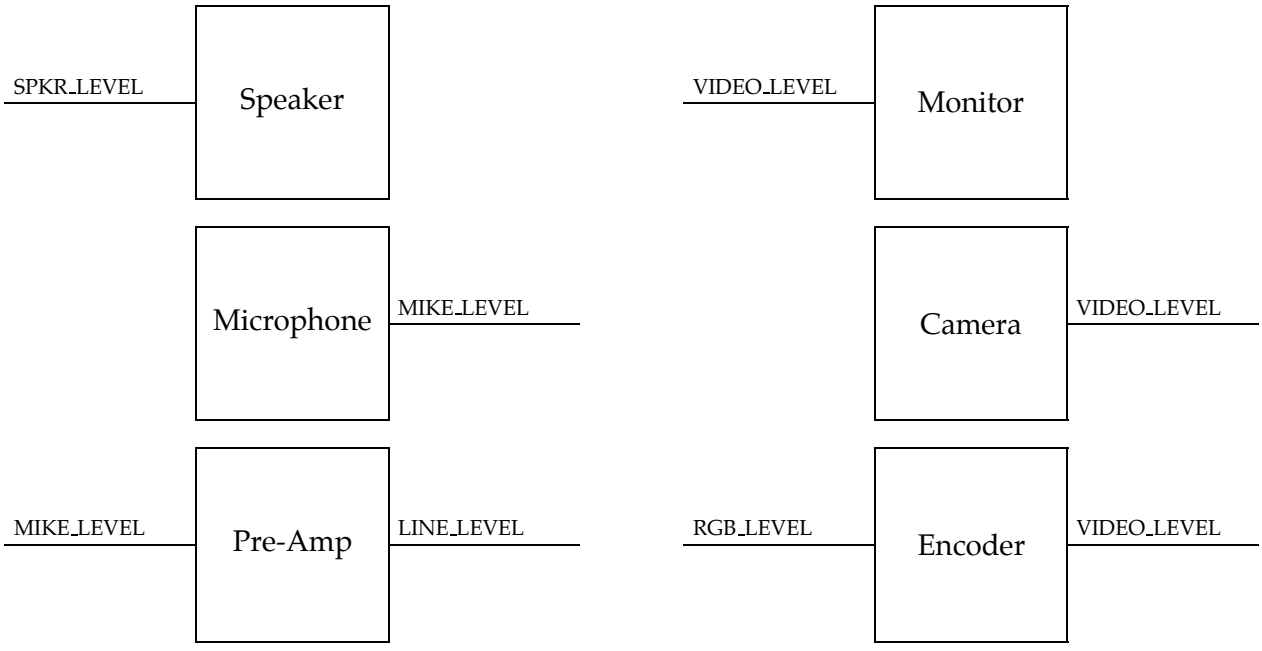
The LAUD is the most important concept of the VOX architecture. Each LAUD represents a simple logical building block as shown in the figures below. The left column represents the audio LAUDs that we will initially be concerned with, while the right column shows equivalent abstractions in

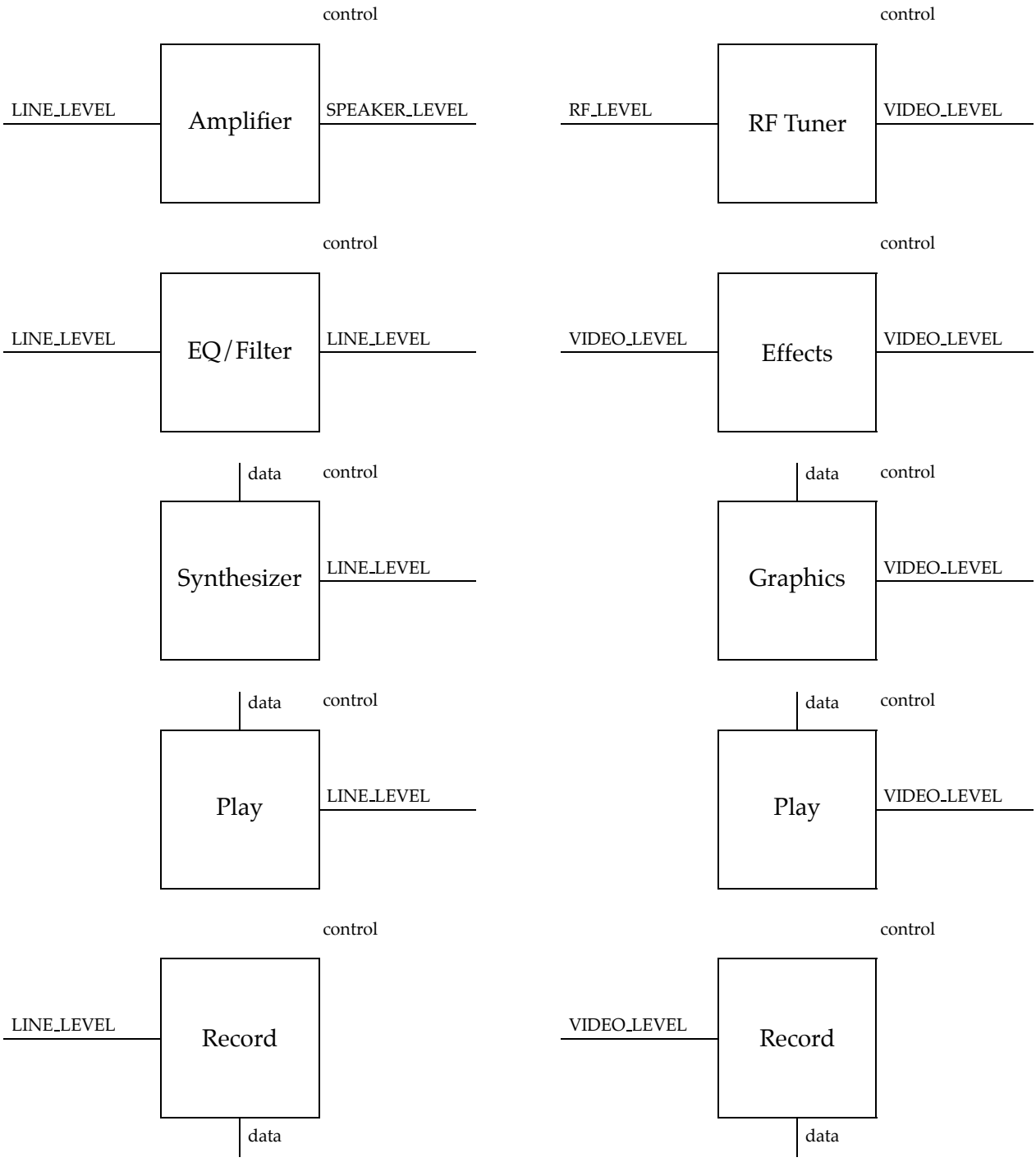
[6]The sound filing system is not necessarily part of the VOX server, although the server reflects the underlying mechanisms to name and access sounds.

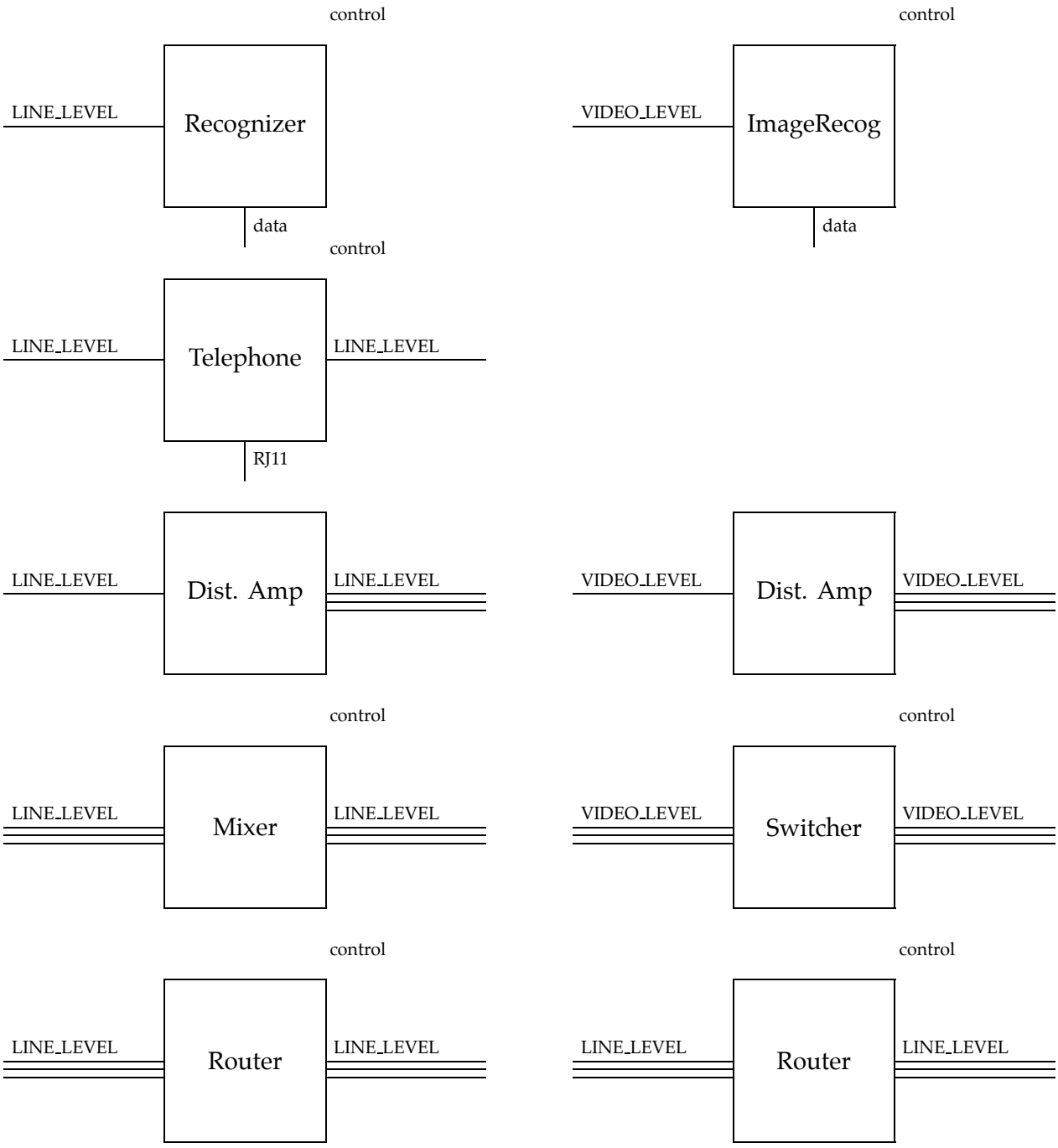
voxlayers.ps

Figure 2.2: Internal VOX Server Architecture

the video domain (see section 2.12.3). The list is sorted in increasing order of complexity. Note that some devices (e.g. text-to-speech synthesizer and playback) have the same I/O behavior.




control
LINE_LEVEL
Amplifier
SPEAKER_LEVEL
control
RF_LEVEL
RF Tuner
VIDEO_LEVEL
control
LINE_LEVEL
EQ/Filter
LINE_LEVEL
control
VIDEO_LEVEL
Effects
VIDEO_LEVEL
data
control
Synthesizer
LINE_LEVEL
data
control
Graphics
VIDEO_LEVEL
data
control
Play
LINE_LEVEL
data
control
Play
VIDEO_LEVEL
control
LINE_LEVEL
Record
data
control
VIDEO_LEVEL
Record
data



A client instantiates a LAUD and receives a handle to that LAUD instance. LAUDs are strongly typed and clients create instances of type extensions of type LAUD. For instance, VOX shall provide a *Phone* LAUD, a *Play* LAUD, and a *Record* LAUD. A LAUD instance can be known to several clients by sharing the LAUD's handle. In particular, the workstation manager needs to be able to refer to arbitrary LAUDs independent of their creators.

2.5.1 CLAUDs

The LAUDs listed above are the primitives to build composite LAUDs (CLAUDs) that reflect a client's specific audio needs. The process for building and using CLAUDs is demonstrated by the following code fragment and is very similar to the creation and use of segment hierarchies in graphic systems such as GKS or PHIGS[7].

```
<create a new CLAUD>;
<open a LAUD; insert it into CLAUD>;
<open a LAUD; insert it into CLAUD>;
<solder LAUD audio ports together>;
<terminate CLAUD building>;
<map CLAUD onto devices>;
<submit output requests to, receive input events from, top-level CLAUD>;
<destroy CLAUD>;
```

Notice that the CLAUD construction must explicitly be terminated. This is necessary to inform other VOX clients of the presence of a newly built CLAUD (in particular the workstation manager, which is a VOX client). We intend to minimize the overhead associated with CLAUD creation, manipulation, and destruction. Thus, clients can use CLAUDs in a rather dynamic style, not unlike the use of windows in current generation window systems.

2.5.2 Input and Output

CLAUDs accept I/O and control requests from the client. *I/O* requests produce audio for the user or generate data from the user's speech; *control* requests modify a CLAUD's audio state. For example, output requests include requests to play a sound, change a volume, or dial a phone number. Input events are generated by the external world, such as speech recognition, telephone ring-detection, touch-tone detection, etc.

With each CLAUD we associate *one* thread of control that processes I/O requests. Requests are submitted to the top-level CLAUD. However, to support the presence of several LAUDs of the same functionality within a CLAUD, the client can indicate which LAUD instance should service a request (see also section 2.8). Clients always read input events from the top-level CLAUD; thus input events are serialized from the client's viewpoint. (Notice that control requests, i.e. requests to establish audio status, do *not* transit through the I/O queues.)

When output requests are created, clients may indicate what external events should terminate the given output request by indicating a logical disjunction of possible termination events. Notice that

[7]However, unlike in graphics, the nesting of LAUDs does not imply any kind of transformation, but expresses only a containment relationship.

the interpretation of these termination conditions are on a per-CLAUD basis, i.e. an asynchronous input event on CLAUD $_1$ does not affect output requests of CLAUD $_2$.

A special output request is the *delay event*. Its effect is to temporarily suspend execution of output requests. Delays can also be terminated upon occurrence of external, asynchronous events.

Clients may query the server on the status of output events (how long is the current recording; how much of a sound has been played.). This communication is out-of-band and does *not* transit through the normal I/O audio queues. Instead, a status query for a given I/O request is synchronous from the client's viewpoint, i.e. the call returns the current status "immediately".

2.5.3 Management

CLAUDs accept a series of managing requests. In principle, these calls can be performed by any client, but we shall rely on a particular client, the *audio resource manager* to manage the VOX server. (See also section 2.3).

The server provides the following operations to perform CLAUD management[8]:

Map: This call maps a CLAUD onto the set of devices it needs to operate and thereby gains access to these devices. The mapping call is only advisory in that the VOX server can map and unmap a CLAUD at any time to allow resource sharing.

Unmap: This call frees the devices that are mapped by a CLAUD.

Lock: If a client needs to gain exclusive non-preemptable access to the devices it may lock them with the *Lock* call. (For an example, see section 2.10.2.)

Unlock: To become preemptable, a client can unlock the devices used by a CLAUD with the *Unlock* call.

Pause: This call temporarily suspends processing of the audio queues. Clients may still submit output requests to suspended CLAUDs; their processing is merely delayed or the output is "thrown-away".

Resume: This is the inverse of the *Pause* call and resumes I/O processing.

Other CLAUD controlling calls are CLAUD-specific. For example, we shall have calls to set the number of rings for the phone to be answered, to indicate the length of pauses for silence detection during recording, etc.

[8]Note that some of these requests can be combined into a single function call. For example, when a CLAUD is mapped there is a flag indicating that the devices are also to be locked. This capability prevents possible deadlock situations.

Currently, we force clients to perform explicit mapping and unmapping of CLAUDS onto devices. One could, however, envision implicit, non-locking, mapping of CLAUDS onto devices at the time the CLAUD actually starts processing output requests or starts accepting input events. It is not entirely established which VOX calls would need to perform implicit mapping and therefore we have—for now—decided to maintain explicit mapping and unmapping. Also, notice that mapping and unmapping may be rather expensive operations and it thus might be judicious to let the client perform these explicitly.

Clients can preempt each other for the usage of devices, unless these are locked. Once the preempting client releases its devices via an unmapping call, the previously mapped client may re-acquire its devices if these are available. However, when attempting to remap a previously mapped CLAUD, the server does not preempt other CLAUDS. Hence, there is no guarantee that the preempted CLAUD becomes remapped transparently to the client. The server attempts to remap clients in the reversed order of their preemption.

The audio resource manager described above relies on the specialized *Monitor* LAUD[9] which provides clients with asynchronous events signaling the creation, destruction, mapping, and unmapping of CLAUDs. The role of the *Monitor* LAUD is thus essentially to monitor CLAUD activity within the server and keep interested clients informed. As shown in figure 2.1, the *Monitor* LAUD plays a role similar to the *root* window in a window server (the X server is a prime example). However, CLAUDs are not nested within the *Monitor* as are windows inside the root. Instead, the *Monitor* acts more like a background daemon managing other CLAUDs and notifying interested clients about the server's state.

2.6 Audio Ports and Connections

Audio ports model the electrical inputs and outputs of LAUDs[10]. They have a *type* and a *direction* corresponding to their electrical signal level and direction of data flow associated with them.

```
TYPE Level = {Mike, Speaker, Line, RJ11, Video, RF_Video};

TYPE Direction = {In, Out, InOut};
```

The direction is self-relative to the LAUD; *in* always means *into* the LAUD. For example, the *line* level *out* port of a pre-amplifier can get connected to the *line* level *in* of a speech recognizer.

Audio ports are referenced by either an abstract *handle* or by their *string name*. Primitive LAUDs have a predefined set of audio ports, such as "out" on a microphone or "in" on a speaker.

[9] We are aware of the overloading of the term *monitor*, but this is the most appropriate name based upon functionality.
[10] There is typically a one-to-one mapping of audio ports to the jacks on the rear panel of an audio device

When building CLAUDs, clients can rename the audio ports of the CLAUD for greater ease of reference[11].

Connections are created by soldering together two audio ports and support dynamic audio routings. Once a connection has been established, the following operations can be performed:

Make: dynamically enables the connection for audio throughput.

Break: dynamically disables the connection for audio throughput.

Destroy: the connection is destroyed and can no longer be used.

2.7 Devices

LAUDs are logical abstractions. In order for them to become truly useful, they must be associated with *devices*[12] that perform the actual audio activity.

2.7.1 Naming

The client must be able to uniquely identify devices that the server supports. For example, a server may control several microphones, several speakers, and several playing and recording devices. We shall use the following device naming scheme:

Major: the major device describes its functionality. For example "recorder", "player", "mic", or "speaker".

Minor: the minor device number specifies the instance of the device the client desires to use. We shall simply use a straight numbering scheme. Thus, we refer to ("mic", 0) or ("speaker", 3). The client will however be able to use the *Any* wildcard in naming a minor device. In this case, the server uses any available device that provides necessary functionality for a given LAUD.

The clients may query the server to find out what devices are accessible via that particular server. This is necessary since an audio environment may offer a considerably greater variety of devices than most graphics environments.

[11] A renamed audio port is not a new port, we simply create aliases to the audio ports of the lowest level LAUDs.

[12] Devices may be incarnated by either software or hardware. For example, a speech recognizer device is mainly implemented in software.

2.7.2 Connectivity

Each server may limit the possibility of device interconnections. The device connectivity matrix describes what devices can be connected together. The matrix elements indicate whether or not audio ports of the given devices can be interconnected. As an example, consider a voice and speech board that implements the following devices:

1. Microphone: has one output audio port; at line level.
2. Speaker: has one input audio port; at speaker level.
3. Line-In: is an audio input port; at line level.
4. Line-Out: is an audio output port; at line level.
5. In-Mux: is an input selector; has two input audio ports and one output audio port; all at line level.
6. Out-Mux: is an output selector; has one input audio port and two output audio ports; at line level and speaker level.
7. Play: has one output port; at line level. (Plus a control and data connection.)
8. Record: has one input port; at line level. (Plus a control and data connection.)

The device connectivity matrix of this board is shown in table 2.1.

Outputs	Inputs				
	Speaker	Line-Out	In-Mux	Out-Mux	Record
Mic			X		
Line-In			X		
Play				X	
In-Mux					X
Out-Mux	X	X			

Table 2.1: Device Connectivity on the Audio Board

Notice that in table 2.1 the In-Mux and the Out-Mux accept only one incoming (respectively outgoing) connection at a time[13].

We also model the case where several audio ports of a device can feed into one input port or where one output port can drive several input ports on other devices. Therefore, a connectivity matrix element can also be *S* (for shared, simultaneous) instead of *X* (for exclusive). As an example, consider a modified version of the voice and speech board, where it is physically possible to connect

[13]This is given by the hardware specifications.

the Mic and Line-In audio out-ports simultaneously into the In-Mux. Similarly, it is possible to drive both the speaker and the line-out audio in-ports from the Out-Mux multiplexer. Table 2.2 reflects the modified situation.

Outputs	Inputs				
	Speaker	Line-Out	In-Mux	Out-Mux	Record
Mic			S		
Line-In			S		
Play				X	
In-Mux					X
Out-Mux	S	S			

Table 2.2: Modified Device Connectivity on the Audio Board

The client need never to see the connectivity matrix; requests to establish illegal connections simply fail.

2.7.3 Exclusivity

Given the above device connectivity matrix of the voice and speech board (see table 2.1), one could theoretically create two CLAUDs: one implementing a recording function and the other a (simultaneous) playback function. However, the board's software doesn't currently support that. Thus, the server needs another matrix establishing which devices can be used concurrently.

For our sample board, the device exclusivity matrix is shown in table 2.3.

	Speaker	Mic	Line-In	Line-Out	In-Mux	Out-Mux	Record	Play
Speaker								
Mic								
Line-In								
Line-Out								
In-Mux						X		
Out-Mux					X			
Record								X
Play							X	

Table 2.3: Device Exclusivity on Audio Board

Based on the matrix of table 2.3, it is obvious that play and record devices can not operate simultaneously. Thus, if a client builds two CLAUDs which incorporate LAUDs mapped onto the *Play* and the *Record* device there is potential conflict (which can be detected at CLAUD creation time). To arbitrate such conflicts, clients have to use the locking and unlocking calls of section 2.5.3.

2.8 Audio Queues

All input and output to CLAUDs transit via *audio queues*. The client may read from a audio queue (*get*) and write to a audio queue (*put*). *Output* queues are used by the client to submit requests for output to the VOX server. Examples are play requests, record requests, and text-to-speech synthesize requests. *Input* queues are used by the client to receive information from a LAUD. Examples are a speech recognizer or the "phone has rung" event coming from a CLAUD incorporating the *Phone* LAUD.

When enqueuing output requests, the server can prepare the request in-advance. For instance, a sound file can be opened and its contents loaded into core memory. Preparation can also be performed on input requests by opening sound files and allocating in-core memory for recording purposes or allocate memory space for recognized speech tokens. This preparation then reduces the latency of the actual servicing of the request during the time-critical phase of real-time audio processing. (VOX borrows these ideas of request preparation from the work done at the MIT Media Lab by two of the authors.)

Prepared output requests are enqueued in the audio queues of a CLAUD. Since all events have been prepared, the queue's request can be serviced in order with almost no delay between subsequent, enqueued service requests. This decouples the client from the server improving the real-time characteristics of audio request processing by reducing the delays between individual requests, independent of the client's behavior. With respect to input, audio queues also play a decoupling role. That is, a recognized speech input token is buffered in the input audio queue until the client dequeues it from there.

Individual LAUDs support the creation of LAUD-specific output events that are then enqueued in the LAUD's output queue either implicitly or explicitly via the *Queue.Put()* operation to be processed sequentially. The audio queue-managing operations of the low-level *Queue* interface can influence the scheduling to a limited extent. For instance, we can implement a *play-one-out-of- several* behavior by preparing and enqueuing a set of play requests and removing all but one before actually executing the requests in the queue (see also section 2.10).

Input audio queues are used by a client to receive data from the server. Input events are asynchronous in nature. For instance, the client signals the presence of an incoming ring event on the phone LAUD via the generation of an appropriate event.

The example below demonstrates how output events are queued explicitly with a CLAUD's output audio queue and how this can be used to delay the processing of the output request depending on an input event.

```
Recognize.Prepare(recognizer, ...., ev1);
(* create two play events; don't queue them ! *)
Player.Prepare(player, ...., enq:= false, sound:= sound1, ev:= ev2);
Player.Prepare(player, ...., enq:= false, sound:= sound2, ev:= ev3);
```

```
.....
(* wait for the recognized token *)
LAUD.Get(recognizer, ev1);
(* and start playing the right sound depending on input token*)
IF (<ev1 == "yes"> ) THEN LAUD.Put(player, ev2);
ELSE LAUD.Put(queue, ev3);
END
```

Processing of the events transiting through audio queues is performed by the top-most CLAUD in a LAUD hierarchy. This is ensured by returning the handle to the top-most CLAUD's queue when performing a *LAUD.Queue()* call on a nested LAUD. To deal with the case where a CLAUD has two LAUDs of similar functionality, the event-creating call *must* name the LAUD on which the request should be serviced. For example, a CLAUD dealing with two playback LAUDs necessitates the following code:

```
Player.Prepare(player:= player1, ..., enq:= true, sound:= sound1; ev:= ev);
 Player.Prepare(player:= player2, ..., enq:= true, sound:= sound2; ev:= ev);
```

Notice that due to our definition of CLAUDs these requests would be serviced sequentially.

2.9 Sound Management

Audio data is stored in *sounds*. We envision these to be stored in files provided by the underlying operating system and do not propose a specialized sound storage sub-system. Access to sounds is through a hierarchical name space and sounds are referred to by their string name. Each CLAUD maintains the notion of its current directory and sound names are interpreted relative to these.

Sound management operations include the standard set of operations related to a storage medium: creation, deletion, copy, rename, etc. Other operations on sound permit to retrieve the sound header that contains detailed information of the named sound. For example, the encoding algorithm, the length of the sound, and the date on which the sound was created are contained in the header.

To support editing, the server exports an interface to *sound buffers*. This data type implements operations to record a sound buffer, play a sound buffer, cut material from a sound buffer, and paste data into a sound buffer. Use of a temporary, in-core data structure for editing purposes avoids the need of incorporating editing functionality into the sound storage system. This contrasts with the Etherphone system [22] where it is the sound storage system that implements the editing primitives.

2.10 Examples

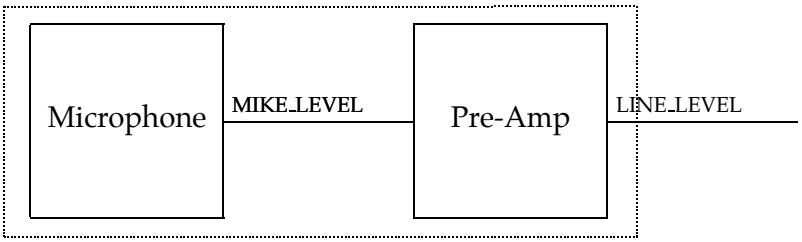
This section demonstrates the VOX model for some simple examples.

2.10.1 Connecting a Microphone and Pre-Amp

This example runs the output from a microphone into a pre-amplifier by creating a new building block. The LAUDs are first inserted into the CLAUD, then the devices are connected together.

```
IMPORT Text, VOX, LAUD, Mic, PreAmp;
VAR sv: VOX.Server;
    speechOut, mic, pre: LAUD.T;
    p1, p2: Port.T;
    connection: Connection.T;

LAUD.Create(speechOut);              (* create new building block *)
Mic.Open("mike", 0, mic);            (* open mic 0 on server *)
PreAmp.Open("pre", 0, pre);          (* open pre-amp 0 on server *)

LAUD.Insert(mic, speechOut);    (* insert mic instance into CLAUD *)
LAUD.Insert(pre, speechOut);   (* insert pre-amp instance into CLAUD *)

(* create connection between LAUDs *)
LAUD.PortOf(pre, "in", p1); LAUD.PortOf(mic, "out", p2);
Port.Solder(p1, p2, enabled:= false; connection);

LAUD.Assemble(speechOut);    (* notify server that CLAUD is complete *)

(* rename audio port *)
Port.Rename(p2, speechOut, "mike_out");
```

The newly created CLAUD looks like this:



Note that the last command adds a new audio port named `"mike_out"` to the LAUD `speechOut`. This new audio port has the same characteristics as the output port of the pre-

amplifier, and *Port.PortOf(pre, "out", ...) Port.PortOf(speechOut, "mike_out", ...)*.

2.10.2 Simple Answering Machine

In this example a simple answering machine is implemented. The sample code shows a situation where devices are made non-preemptable, due to the interactive and real-time behavior of a telephone conversation. Notice that the phone device only becomes locked *after* the incoming ring has been detected; this allows outgoing calls to be initiated while the phone is not actively in use.

```
IMPORT VOX, LAUD, Port, Event, Phone, Recorder, Player;
VAR sv: VOX.Server;
    s: VOX.Status;
    am, phone, rec, play: LAUD.T;
    ev, recEv, playEv: Event.T;
    p1, p2: Port.T;
    c1, c2: Connection.T;
    phStatus: Phone.Status;

(* create new answering machine *)
LAUD.Create(am);
Phone.Open("phone", 0, phone);
Recorder.Open("record", Any, rec);
Player.Open("play", Any, play);

(* insert LAUDs into CLAUD *)
LAUD.Insert(tel, am); LAUD.Insert(rec, am); LAUD.Insert(play, am);

(* establish connections between LAUD ports *)
Port.PortOf(rec, "in", p1); Port.PortOf(tel, "out", p2);
Port.Solder(p1, p2, c1);
Port.PortOf(tel, "in", p1); Port.PortOf(play, "out", p2);
Port.Solder(p1, p2, c2);

(* signal termination of CLAUD and map it onto devices *)
LAUD.Assemble(am);
LAUD.Map(am, lock:= false, block:= false);

(* prepare I/O requests without submitting for execution *)
Player.Prepare(play, sound, enqueue:= false, ..., ev:= playEv);
Recorder.Prepare(rec, sound2, enqueue:= false, ..., ev:= recEv);

(* wait for an input event *)
```

```
LOOP
  LAUD.Get(am, ev);
  TYPECASE Event.BodyOf(ev) OF
    REF Phone.RingEv =>  (* phone is ringing *)
        (* make LAUD non-preemptable *)
        IF (LAUD.Lock(am, block:= false) = VOX.Success) THEN
          (* start answering phone call *)
          phStatus.set = Off;
          Phone.SetStatus(tel, {Phone.Set}, phStatus,...);
          (* submit I/O requests for execution *)
          LAUD.Put(am, playEv);
          LAUD.Put(am, recEv);
          (* wait until recording terminates *)
          Event.WaitForState(recEv, {Done});
          (* make LAUD preemptable *)
          LAUD.Unlock(am);
        END;
    ....
  END; (* TYPECASE *)
END; (* LOOP *)
```

Schematically we can represent the phone answering CLAUD as in figure 2.3. Note that the individual audio queues of the component LAUDs are subsumed by the top level CLAUD queue.

ansmach.ps

Figure 2.3: Answering Machine CLAUD

2.10.3 Extended Audio Queue Example

This example demonstrates our LAUD model in a setting where the client wants to record and recognize parts of the voice input.

```
IMPORT VOX, LAUD, Event, Phone, Recorder, Player, Port;
VAR sv: VOX.Server;
    recognizer, record, mic, switch, rr: LAUD.T;
    s: VOX.Status;
    p1, p2: Port.T;
    c1, c2, c3: Connection.T;
    ev: Event.T;

(* create the CLAUD hierarchy *)
LAUD.Create(rr);
Recognizer.Open("recognizer", 0, recognizer);
Recorder.Open("recorder", 0, record);
Mic.Open("mic", 0, mic);
(* open a switch with one in-audio-port; two out-audio-ports *)
Switch.Open(sv, "switch/1/2", Any, switch);

(* insert LAUDs into CLAUD *)
LAUD.Insert(recognizer, rr);  LAUD.Insert(record, rr);
LAUD.Insert(mic, rr);  LAUD.Insert(switch, rr);

(* create port connections between LAUDs *)
Port.PortOf(switch, "in", p1); Port.PortOf(mic, "out", p2);
Port.Solder(p1, p2, c1);
Port.PortOf(record, "in", p1); Port.PortOf(switch, "out1", p2);
Port.Solder(p1, p2, c2);
Port.PortOf(recognizer, "in", p1); Port.PortOf(switch, "out2", p2);
Port.Solder(p1, p2, c3);

(* signal completion of CLAUD and map it to its devices *)
LAUD.Assemble(rr);
LAUD.Map(rr, lock:= true, block:= true);

(* submit I/O requests and execute them *)
Switch.Prepare(switch, "in", "out1", On, ev);
Record.Prepare(rr, ..., ev);
(* now switch to use recognizer; assume toggle switch *)
Switch.Prepare(rr, "in", "out2", On, ev);
Recognizer.Prepare(rr, ..., ev);
LAUD.Get(rr, ev);   (* from recognizer *)
```

The *Switch* LAUD is a LAUD that enables the switching of two *audio* lines. The *Recognizer* LAUD provides functionality for speech recognition.

Notice that a CLAUD services prepared requests in order. Thus, the first request switches the *in* audio port to the output audio port *out1* which directs the microphone's input to the recording LAUD. Then the actual recording takes place, after which the switch is "flipped" to direct the switch's output to the recognizer LAUD.

2.11 Related Work

The VOX server architecture relates to work in two distinct areas: one area is, of course, related to voice and telephony, while the other area relates to window systems and user interfaces.

VOX attempts to integrate the functionality provided by systems built at IBM Research [16] and at MIT's Media Lab [18, 20, 19]. These include support for telephony as well as for text-to-speech synthesis and speech recognition. The work done at the Media Lab pioneered the notion of request preparation in order to have low latency execution of audio requests which VOX incorporates in its design. As extensions to these systems, however, VOX takes a more dynamic approach to audio routing, provides increased flexibility and operates in an environment supporting several, possibly distributed, clients at once.

In contrast with the Etherphone system developed at Xerox PARC [21, 22, 23], VOX does not rely on a centralized voice storage server. Instead, sounds are stored on the workstation on which the server is running and VOX provides features to migrate sounds over the network. Moreover, VOX does not support sound editing as primitives of the sound storage system, but shall support the concept of in-core sound buffers to support a voice editor. A further difference with the Etherphone system is VOX's flexibility in establishing audio routings on the server to support a varied class of applications.

The influence of window systems and user interface software on VOX appears in several aspects. First, like the NeWS and the X window systems [11, 17], VOX is network transparent. That is, a client can reside on any machine in a local area network and access the audio server. Secondly, VOX inherits the concepts of logical hierarchies from graphics systems such as VGTS or Eureka [6, 13]. Instead of building display hierarchies, however, we build hierarchies of audio abstractions.

VOX also incorporates the idea of a privileged client that is used to manage the server and its resources. In that respect VOX is similar to window systems with an associated window manager [6, 11, 17]. Another similarity with user interface software is the layering of the VOX software: the architecture largely follows the model of [14]. Indeed, the VOX architecture can be roughly partitioned in the device layer, an abstraction of the device layer, and the client or application layer with the possibility of an intermediate toolkit layer. (See also figures 2.1 and 2.2).

Unlike the NeWS window system [11], VOX does not provide a full-fledged interpreter within the server. The processing of audio I/O requests is purely sequential; all the control logic is within the VOX client. It is an open question whether an equivalent to the NeWS model, using the PostScript programming language [24], can be established for voice. In particular, it is unclear whether or

not a simple framework similar to two-dimensional graphics exists for audio.

2.12 Open Issues

2.12.1 Sharing Input and Output Devices

As presented thus far VOX disallows concurrent access to input or output devices by more than one client. However, consider the case of an answering-machine client that is waiting to seize a telephone line while a separate call-accounting client is passively waiting to track telephone use. They both want to receive the "incoming ring" event, even though only the answering machine is actually going to take the phone off hook. Unfortunately, the current model does not allow for both clients to receive the event notification. What is needed is the software analog of a hardware splitter or distribution amplifier.

Similarly, consider the case of a calendar program that wishes to announce an appointment during an existing teleconference. Both clients need access to a loudspeaker; if only one is available, the audio output from both clients should be mixed, with the workstation manager determining relative volumes based on input focus. Again, the current model does not support such sharing. It is necessary for such clients to be *implicitly* connected to a variLAUD when accessing an audio output device such as a speaker.

We are addressing these issues but a consensus was not reached by the time this document went to press.

2.12.2 Toolkit Extensions

As detailed in this document, the VOX server implements the minimal functionality necessary to create useful multi-media applications. The current level of granularity forces the client to request specific devices and connect specific audio ports. Hence, the application writer must know ahead of time what devices exist on the server and how they are configured. Like the X Window System, VOX implements the low-level mechanism for dealing with devices, but it does not enforce any policy on how the hardware is used. VOX allows the application writer to "roll his own" audio system, but does not give him many tools to assist him.

We hope to build a high level toolkit that relieves the client of many of the details of building and controlling user-level circuits. With such a toolkit the client would be shielded from having to know what devices were present on the server and need not specify the CLAUD creation, LAUD instantiation, and interconnection in such excruciating detail. In the case of often used configurations there could be a library of "audio widgets" that can be called upon if, for example, a simple digital recorder, telephone-based teleconferencing circuit, or answering machine is needed.

2.12.3 From Audio to Video

Most of our applications experience and the scenarios discussed in preparation of this document have focused on audio. However, as illustrated by the diagrams in section 2.5, from an abstract point of view most of the audio LAUDs have analogous logical devices in the video domain. Moreover, most of the design constraints—sharing, routing, etc.—discussed with respect to audio in section 2.1 also apply to video.

Therefore, we believe that the basic concepts and general architecture of the VOX server can be applied to video. It is too early to tell if the video server should be an extension to VOX (making it a multi-media server), or an extension to X (in the spirit of the X Video Working Group), or if it should be completely separate "VIX" server. We plan on working on such a video server in the future and incorporating it into the multi-media workstation envisaged in section 2.2.

2.13 Acknowledgments

Chris Lauwers' observations forced us to revise earlier models of the VOX server. Device arbitration via late binding is based on suggestions by Derek Oppen. Comments by Pehong Chen, Tommy Joseph, Richard Koo, Allyn Romanow, and Wayne Yamamoto have also influenced the VOX design.

Chapter 3

Configuration

Revision History:

Author(s)	Date	Rev.	Comments
Edwards	Jul. 10 89		Original manual
Edwards	Aug. 8 89		Add xbar and interconnects
Arons	Oct. 17 89	0.9	Reformat, cleaned up

3.1 Introduction

This document describes the procedures involved in setting up and changing a VOX system configuration for the hardware configuration for a particular workstation. The configuration code allows users to modify their machine-dependent configurations without having to recompile any VOX server source code. Simple human-readable ASCII files are used to store all startup information.

3.2 The Files

There are six files that VOX uses to determine its startup state. These are called `vox.dev`, `vox.xbar`, `vox.config`, `vox.conflict`, `vox.connect`, and `vox.inter`. The files are parsed in the order they are listed here. The VOX server looks at the environment variable `$CONFIG_DIR` for the directory in which these files reside.

In all the files, lines which start with a hash-mark (#) in the first column are treated as comments. The files are parsed line-per-line, with several fields per line. There should not be any extraneous linefeeds in meaningful lines in the file, but totally blank lines are permitted. Any whitespace characters except for newlines can be used to delimit fields. If there is an error in any field, or not enough fields, an error is reported and the line is ignored.

3.2.1 `vox.dev`

The `vox.dev` file is a database of the capabilities of various devices. This file should remain relatively static within a site using the VOX Audio Server, and should contain entries for all the devices which may possibly be connected to a workstation.

A fragment of a sample `vox.dev` file is shown in figure 3.2.1.

Each `TYPE` roughly corresponds to a LAUD. These types need to have been compiled into the server (see Adding New Devices, section 3.4).

The `MAKE` and `MODEL` fields are simply ASCII strings which specify the vendor name and model of the equipment.

`PORTS` gives the number of ports associated with a particular device type. Note that in the case of crossbar switches, the number of ports reflects the total input and output ports (thus a 16 16 crossbar has 32 ports).

The last field is the `PORT_INFO` field. This field consists of a list of entries specifying the `LEVEL` and `DIRECTION` of the port. Except in the case of *switch* devices, there should be one entry per port in

```
#      sample vox.dev file
#
# TYPE       MAKE         MODEL      PORTS    PORT_INFO
#-------------------------------------------------------------------
OATCP         Olivetti     oatc         1      LINE/OUT
OATCR         Olivetti     oatc         1      LINE/IN
DMP11I0       Yamaha       dmp11        2      LINE/IN LINE/OUT
DMP11I1       Yamaha       dmp11        2      LINE/IN LINE/OUT
DMP11M0       Yamaha       dmp11        2      LINE/IN LINE/OUT
VBXP          NMS          VBX/I        1      LINE/OUT
VBXR          NMS          VBX/I        1      LINE/IN
```

Figure 3.1: `vox.dev` file.

this field. Each `PORT_INFO` entry should be of the form `LEVEL/DIRECTION`, where `LEVEL` may be one of either `{MIKE, SPEAKER, LINE, RJ11, VIDEO, RF_VIDEO}`. `DIRECTION` may be one of either `{IN, OUT, INOUT}`. These values must be separated by a slash. *Switch* devices are assumed to have the same port characteristics for each port, and therefore have only one entry in the `PORT_INFO` field. Users may assume that this port characteristic is replicated for all ports on the switch device.

3.2.2 `vox.xbar`

`vox.xbar` contains a description of the various crossbar switches which may be connected to a workstation. Crossbar switches differ from most other audio devices which VOX knows about in that crossbars do not correspond to LAUDs in the server. Clients generally have no knowledge of crossbars and instead rely on the VOX server to manipulate crossbar connections. See figure **??**.

```
#      sample vox.xbar file
#
# NAME       TYPE         DEVICE          PORTS
#-----------------------------------------------------------
primary       DP2000       /dev/tty00      32
secondary     DP2000       /dev/tty01      32
#
# this workstation is also controlling the central xbar
central       DP3200       /dev/tty02      64
```

Figure 3.2: `vox.xbar` file.

The `NAME` field is the logical name of the crossbar. This name is used in other files to reference a particular port on a given crossbar. NAME may be any printable ASCII string (except for whites-

pace characters).

TYPE is one of the crossbar types. These types must be compiled into the server (see Adding New Devices section 3.4), but needn't be listed in vox.dev.

DEVICE is the Unix device name at which the crossbar resides (usually a serial port).

PORTS is the *total* number of ports available on the crossbar (thus a 16 16 crossbar has 32 ports).

3.2.3 vox.config

vox.config contains the machine-specific configuration information. This is the information which is not device-dependent but rather varies from machine to machine. The format of the file is shown in Figure 3.2.3.

The first field is for the TYPE of the device. Note that there are "generic" audio types present in the vox.config file. There are three generic types: SOURCE, SINK, and FILTER. Each of these generic types is assumed to have one port, line level, and direction either In, Out, or InOut (depending on the type).

All the types listed here must be known to the VOX server (the device interface routines for the type must be compiled into the server). In addition, if the type is not one of the generic types, the TYPE must be listed in the vox.dev file. Generic types need not be listed in the vox.dev file since their characteristics are known *a priori*. In fact, any attempt to declare a generic type in the vox.dev database will result in an error informing you that you are attempting to redeclare one of the generic types.

The TYPE field is used to produce the *major* name of the device (see What the Configuration Code Produces, section 3.3).

The next field is the INSTANCE field. The INSTANCE field is simply a minor device number for the attached piece of audio equipment. For example, there may be two CD players attached to a particular workstation. If they are both of the same type, one would be specified as INSTANCE 0, the other as INSTANCE 1.

DEVICE is the logical device filename to which the equipment is attached (if any). Any equipment which is accessed through its device name should have that name listed here. If the equipment is not accessed through its device name, the value NONE should be given.

The next field lists information necessary to locate the device's logical ports on actual physical hardware. Whereas most audio components (including most standard rack-mount audio-gear) exchange analog audio data through the crossbar switch, there may be devices which do not reside on the crossbar. Devices not on a crossbar should have the value NONE specified in this field.

For those devices which do connect to a crossbar switch, the next field consists of a list of strings

```
#      sample vox.config file
#
# TYPE(MAJ)  INST(MIN) DEVICE     XBAR_NAME:XBAR_PORT:PORT_NAME
#-------------------------------------------------------------------------
#
# there is a DMP11 mixer on tty03 (I/O goes to the primary crossbar)
#
DMP11I0          0      /dev/tty03  primary:0:dmpi0.0 primary:1:dmpi0.1
DMP11I1          0      /dev/tty03  primary:2:dmpi0.0 primary:3:dmpi0.1
DMP11M0          0      /dev/tty03  primary:4:dmpi0.0 primary:5:dmpi0.1
#
# the rest of this stuff is on the secondary crossbar
OATCP            0      NONE        secondary:0:oatc-play
OATCR            0      NONE        secondary:1:oatc-record
VBXP             0      /dev/vbx    secondary:2:vbx-play
VBXR             0      /dev/vbx    secondary:3:vbx-record
VBXPH            0      /dev/vbx    secondary:4:vbx-phone
#
# finally, here are the generic (dumb) audio sources and sinks
#      from Carl's office
SOURCE           0      NONE        primary:6:source.0
#      from Barry's office
SOURCE           1      NONE        primary:7:source.1
#      my local mike
SOURCE           2      NONE        primary:8:source.2
#      to Carl's office
SINK             0      NONE        primary:9:sink.0
#      to Barry's office
SINK             1      NONE        primary:10:sink.1
#      my local speaker
SINK             2      NONE        primary:11:sink.2
```

Figure 3.3: `vox.config` file.

(separated by white space) which list the logical location and name of the ports on the devices. There must be exactly one of these strings for each port present on the device (as determined by looking up the device in the `vox.dev` database). `SOURCE`, `SINK`, and `FILTER` types each have one port.

The strings have three components, delimited by colons. The first component is the name of the crossbar on which the port resides. This name must have been previously declared in the `vox.xbar` table. The next component is the number of the wire on the named crossbar on which the logical port of the device resides. Finally, the third component is the logical name of the port. Users are free to come up with their own naming scheme for ports.

Note that a `NONE` in both the `DEVICE` and `PORT_INFO` fields indicates that the device is not attached to the computer.

3.2.4 `vox.conflict`

The `vox.conflict` file lists those devices which cannot be operated at the same time. For example, because of hardware limitations some boards may not be able to perform the record and playback functions at the same time. See Figure 3.2.4.

```
#      sample vox.conflict file
#
# TYPE        CONFLICTS (TYPE/MODE ...)
#-------------------------------------------
VBXR          VBXP/E OATCP/S
VBXP          VBXR/E
```

Figure 3.4: `vox.conflict` file.

Each line of the `vox.conflict` file has two fields: a `TYPE` field, and then a list of `CONFLICTS` denoting which devices may not be used in conjunction with that `TYPE`. Each entry in the `CONFLICT` list consists of a `TYPE`, followed by a slash, followed by the *exclusivity mode* for that type. The `TYPE` must be one of those types which are listed in `vox.dev`. The `MODE` for the conflict is one of either `{N, E, S}` for non-exclusive, exclusive, and shareable.

A non-exclusive mode means that there is no conflict for the particular device (and thus there is really no need for it to be listed in the `vox.conflict` file). An exclusive mode means that the types always conflict within a given instance and may never be used at the same time (although types may be used at the same time if they are on different instances). Shareable mode means that the types may be used at the same time only if they are mapped into the same CLAUD.

This file is used to generate the `CF_conflictTable` internal table (see What the Configuration Code Produces, section 3.3).

3.2.5 vox.connect

The vox.connect file lists any special connections (because of hardware dependencies or local configuration) which may be possible for the VOX hardware. Any devices which do their own internal routing (for example, the play and record functions on a particular board may be connected independently of the crossbar) should be listed here. Any device types which do not connect directly to a port on the crossbar should also be listed here. It is assumed that any devices which connect directly to the crossbar may connect to any other devices on the crossbar, within the constraints of port direction and level. Thus, most "normal" devices (those which are simply connected to the crossbar) do not need to be listed here. See Figure 3.2.5.

```
#      sample vox.connect file
#
# TYPE/PORT        CONNECTION_LIST (TYPE/PORT/MODE ...)
#-----------------------------------------------------------------
OATCR/0            OATCP/0/P
OATCP/0            OATCR/0/P
DMP11I0/1          DMP11M0/0/A
DMP11I1/0          DMP11M0/0/A
DMP11I1/1          DMP11M0/0/A
```

Figure 3.5: vox.connect file.

The file lists which ports on which devices may connect to other ports on other devices. The first field, the TYPE/PORT field, gives a device TYPE, followed by a slash, followed by a PORT number. Next is the CONNECTION_LIST field. This consists of one or more entries of the form TYPE followed by a slash, followed by a PORT number followed by a slash, followed by a *connection mode*. There are three MODES which may be specified here. A MODE of N means that the devices may never be connected. A MODE of A means that the devices are always connected (they cannot be disconnected). A MODE of P means that they are potentially connectable (even though they may not be on a crossbar, there is some mechanism which allows the logical devices to be connected).

3.2.6 vox.inter

The file vox.inter specifies the connections between different crossbars (both on the same workstation and on various workstations). The configuration code parses tables of the format shown in Figure 3.2.6.

What this file precisely means hasn't been determined yet. There will probably be further changes to the format of this file before all the inter-server communication code works.

```
#     sample vox.inter file
#
# SERVER:WIRE        DIR          XBAR:WIRE
#-------------------------------------------------
torino:0              IN           primary:6
torino:1              OUT          primary:7
#
napoli:8              IN           primary:8
napoli:14             OUT          primary:0
```

Figure 3.6: `vox.inter` file.

3.3 What the Configuration Code Produces

The four configuration files are used to generate several internal tables that VOX uses. These tables are used by the server to store information about the various devices which are connected to the workstation. During the parsing process, the configuration module also creates several internal tables which are not externally visible and are not discussed here.

3.3.1 `CF_devTable`

The `CF_devTable` is an array of `CF_Device` types. This structure has the following declaration:

```
typedef struct {
   char *maj;                 /* major device name */
   unsigned short min;        /* minor device name */
   char *unixDev;             /* unix /dev filename */
   CF_DevType typ;            /* type id for device */
   int fd;                    /* file descriptor for device */
   unsigned short nbrOfPorts; /* number of  I/O ports on device */
   int (*map)();              /* map routine */
   int (*unmap)();            /* unmap routine */
   int (*ioctl)();            /* io control one arg */
   int (*read)();             /* read routine */
   int (*write)();            /* write routine */
   int (*start)();            /* activate device */
   int (*stop)();             /* halt device */
   int (*ioctl2)();           /* io control two args */
} CF_Device
```

The configuration code gets the major device name `maj` by searching for the device's `TYPE` in a table called `typeList`. This table, which is compiled into the configuration module, contains all the allowable `TYPES` and the corresponding device major names (usually the device major name is just a lowercase ASCII representation of the `TYPE` identifier).

The minor device number `min` comes from the `INSTANCE` field in the `vox.config` table. The `typ` member comes from the `TYPE` field in the same file. `nbrOfPorts` is determined by looking for the `TYPE` entry in `vox.dev`. The other structure members are pointers to various device control routines. These members are bound to function pointers by a switch statement in the configuration code.

Crossbar devices are also entered into the `CF_devTable` by the configuration module. The major device name is obtained similarly to the major device name for other types. In the case of crossbars, the crossbar type is searched for in an internal table called `xbarList`. All of the device control routines are set to point to `NULL`.

The total size of the `CF_devTable` is kept in a globally-visible variable called `CF_devTableSize`.

3.3.2 `CF_portTable`

The `CF_portTable` consists of an array of `CF_Port` structures. For the most part, each of these structures corresponds to one port in the local VOX configuration. Devices which have their `SWITCH` field in `vox.dev` specified as Y are not entered into the port table (these devices are only used by the server for routing—it is the devices which connect to the switcher that are required in the `CF_portTable`). The structure contains information on the characteristics of the port.

The `CF_Port` structure is declared as:

```
typedef struct {
   unsigned short dev; /* index of device on which port resides */
   unsigned short id;  /* index nbr of port on device */
   char *name;         /* port name */
   P_Level level;      /* signal level of port */
   P_Direction dir;    /* signal direction of port */
   short xbarIndex;    /* index into the xbar table */
   short wire;         /* physical wiring to the crossbar */
} CF_Port;
```

The `dev` member is the index of the device on which the port resides in the `CF_devTable` structure. `id` is the port number within that device instance. `name` is the obtained from the name

field of the `vox.config` file. `level` and `dir` are obtained from the port characteristics field of `vox.dev`.

Since the configuration module allows for multiple crossbars, it is necessary to indicate which crossbar the port resides on. The `xbarIndex` field gives the index into the crossbar table `CF_xbarTable` for the crossbar to which the port is attached. The `wire` member is the crossbar port number to which this port is connected (a -1 is stored here if the port is not connected to the crossbar).

The globally visible variable `CF_portTableSize` gives the size (in number of elements of the port table.

3.3.3 CF_conflicts

The `CF_conflicts` table is simply a array which lists the conflicts among the various devices. The values which may be present in the table are `VOX_NoConflict`, `VOX_Exclusive`, and `VOX_Shared`. These values (corresponding to the exclusivity modes `S`, `N`, and `E` in the `vox.conflict` table) are stored in row–column addresses to indicate whether or not there is a conflict between two devices. This table is always symmetrical (if A cannot be used at the same time as B implies that B cannot be used at the same time as A).

The table is constructed from the `vox.conflict` file. Conflicts are entered into `CF_conflicts` on a per-instance basis.

3.3.4 CF_connectivity

The `CF_connectivity` table is an array, _ _ , that indicates which devices may be connected. The table is created by reading in the `vox.conflict` file and filling in the various row–column addresses with one of the modes `CF_NotConnectable`, `CF_IsConnected`, `CF_UnConnected`, or `CF_AlwaysConnected`.

Three of these states (`CF_NotConnectable`, `CF_AlwaysConnected`, and `CF_UnConnected`) may be entered into the `vox.connect` table by specifying the modes as `N`, `A`, or `P`, respectively. The remaining mode, `CF_IsConnected`, is used to represent ports that may possibly be either connected or unconnected that happen to currently be in the connected state. Since all of these connections are made by the server, two ports may not be initially set into this state by the configuration tables.

The `vox.connect` table is always symmetrical.

Note that devices which are connected to the crossbar are considered to always be connectable and are not entered into the table.

3.3.5 CF_xbarTable

The CF_xbarTable is used primarily by the routines for crossbar control. The table is an array of struct CF_Xbar:

```
typedef struct {
   char    *name;                  /* logical name of this xbar */
   CF_XbarType    xbarType;        /* type identifier */
   short          index;           /* index of xbar in devTable */
   char           *dev;            /* unix device name */
   int            (*xbar_open)();  /* open routine */
   int            (*xbar_set)();   /* connect two ports */
} CF_Xbar;
```

All of the information is obtained from vox.xbar. The open and set routines are bound to the pointers by a switch statement in the configuration module.

This table is used by the routine xbar_init() which opens and initializes all the crossbars present on the workstation. xbar_init() provides a layer of abstraction between the configuration code and the device-dependent intricacies of the various crossbar types.

The size of the crossbar table is available from the global variable CF_xbarTableSize.

3.3.6 CF_interTable

Currently, CF_interTable is stubbed and contains no data. Eventually, when all the semantics of inter-server communication are known, this table will be constructed by the configuration module. The module currently parses the file vox.inter and records the number of entries in the global variable interTableSize, but builds no data in the CF_interTable itself.

3.4 Adding New Devices

It will be necessary to recompile the VOX server as new types of devices are added. There are several steps that must be taken to add new devices to the server. First, the enumerated type called CF_DevType located in the config.h file must be expanded to list the type names of the new types. (Make the changes to CF_XbarType if the type is a new crossbar.)

The table typeList in config.c gives the mappings between type identifiers and major device names. It is also used to validate the types entered into the various tables. This table should

be changed when new devices are added (make the changes to `xbarList` when adding new crossbars).

Also in `config.c`, a switch statement in the routine `BuildDevTable` binds the device control function pointers to their appropriate routines. A new case in the switch must be added so that new types may be bound to their device routines. (The switch to bind crossbar types to their device control routines is in the procedure `BuildXbarTable`.)

Certain devices may be easily "compiled out" of a particular server even though the server source may reference the device types. For each device type, there is a "HAVE" `#define` which specifies whether the device should be compiled into the server. For example, to leave the DMP11 device routines out of the server (and thus avoid having to link with them), specify `#define HAVE_DMP11 0`. If the value is defined to be 0, the server will not declare the DMP11's device control routines as being `extern`. Instead, they will be `#define`'d to `NULL`. This feature is useful when a particular device interface has not been implemented for a certain hardware platform. For example, the VBX board will not work on a Sun. By doing `#define HAVE_VBX 0` the server will not try to link with the (non-existent) VBX control routines. The various device header files should implement this feature properly for their particular devices.

The interface generator file `vox.m` should be extended to included the interfaces for your new device. After this file has been modified it will be necessary to run *mercury* again. *Mercury* will generate the libraries which you must link the server and clients with. You need to make sure that both the server and the clients are linked with the same version of these libraries.

Finally, in the file `obj.c` there is a switch statement in the routine `OBJ_Validate` in which you must add your new devices (e.g., `VOX_Phone`).

Chapter 4

Crossbar Management

Revision History:

Author(s)	Date	Rev.	Comments
Edwards	Aug. 9 89		Original Manual
Arons	Oct. 17 89	0.9	Reformat, cleaned up, rewrote.

4.1 Introduction

Warning: This document is incomplete.

The control of audio crossbar switches is one of the most important capabilities of the VOX Audio Server. This document describes in detail how to set up, configure, and control crossbar switches. Information is also provided to those wishing to add new crossbar types to the VOX Audio Server. The subject of crossbar mangement is treated at three levels: the user level (primarily concerned with setting up the crossbar configuration on a particular workstation), the server level (what routines and data structures does the server use to manipulate the crossbars), and the device layer (what is involved with controlling crossbars at the lowest levels of abstraction).

Most of the preliminary work on crossbar control was done by Pehong Chen. Keith Edwards later reworked that code to make it dynamically configurable and to support multiple crossbar instances and multiple crossbar types.

4.2 Crossbar Configuration

Crossbar configuration is performed by `config.c`, the configuration module in the server . The configuration module reads in several files when the server starts (see [10]). The configuration file `vox.xbar` sets up the crossbar configuration on a particular workstation.

Here is a sample `vox.xbar` file:

```
#
#      sample vox.xbar file
#
# NAME       TYPE         DEVICE          PORTS
#-------------------------------------------------------------
primary       DP2000       /dev/tty00      32
secondary     DP2000       /dev/tty01      32
#
# this workstation is also controlling the central xbar
#
central       DP3200       /dev/tty02      64
```

There are four fields per line in this file. The first gives the logical name of the crossbar. This name is used in other configuration tables to refer to specific crossbars. Each crossbar should have a unique name.

The next field gives the type of the crossbar device. It is the device type that determines which device control routines will be used to manipulate the crossbar. Currently there are two supported

crossbar types: the Akai DP2000 (16 16), and the Akai DP3200 (32 32).

The third field specifies the Unix device to which the crossbar is attached. All MIDI control information for the crossbar will go through this device.

Finally, the last field specifies the number of ports available on the crossbar. Note that this is the *total* number of ports (thus, a 16 16 crossbar has 32 ports).

This file may be hand-edited at any time. The server should be killed and restarted to force a re-read of the configuration files.

The configuration code builds a globally-available table called `CF_xbarTable` which contains the pertinent information about the crossbars attached to the system. The size of this table (in number of elements) is stored in `CF_xbarTableSize`.

4.3 Adding New Crossbar Types

There are four places in the server code which must be changed to add a new crossbar type to VOX.

1. The enumerated type called `CF_XbarType` located in the `config.h` file must be expanded to list the type name of the new crossbar type. This enumerated type is used throughout the server to specify crossbar types.
2. The table `xbarList` in `config.c` gives the mappings between crossbar type identifiers and major device names. It is also used to validate the crossbar types entered into the various tables. This table should be changed when new devices are added.
3. In `config.c`, a switch statement in the routine `BuildXbarTable()` binds the device control function pointers to their appropriate routines. A new case in the switch must be added so that a new crossbar type may be bound to the device routines.
4. Certain devices may be easily "compiled out" of a particular server even though the server source may reference the device types. For each device type, there is a "HAVE" `#define` which specifies whether the device should be compiled into the server. For example, to leave the DP2000 device routines out of the server (and thus avoid having to link with them), specify `#define HAVE_DP2000 0`. If the value is defined to be 0, the server will not declare the DP2000's device control routines as being `extern`. Instead, they will be defined to be `NULL`. This feature is useful when a particular device interface has not been implemented for a certain hardware platform. The various device header files for the specific crossbar types should implement this feature properly for their particular devices.

4.4 Controlling the Crossbars

As with most other VOX devices, crossbar control is achieved through a two-layer mechanism. However, crossbars differ from other VOX devices in at least one important way. Crossbars do *not* correspond to LAUDs, as do most other devices. Thus, there is no "LAUD-layer" in the crossbar control software. The two layers of the crossbar software are called the *generic* layer and the *device* layer. The code for the generic layer is found in the VOX `dev/xbar_generic` directory, while the code for the various device-specific routines are found under the `dev` directory in subdirectories named after the particular crossbar device (e.g., `dev/dp2000`).

4.4.1 The Generic Layer

The generic layer is the device independent layer of the crossbar software. Three routines are provided by the generic layer: `xbar_init()`, `xbar_connect()`, and `xbar_set()`.

`xbar_init()` uses the globally-accessible `CF_xbarTable` to open and initialize all the crossbars and associated data structures. See section 4.5, Crossbar Initialization.

`xbar_connect()` takes as parameters a `name`, and two `wires`. The routine connects the specified wires on the named crossbar (if it exists). The routine ensures that the appropriate device dependent connection procedure is called.

Finally, the `xbar_set()` call takes two parameters which are indices into the server's port table. The routine calls the appropriate connection routine (this is dependent on the device type of the crossbar). Currently, the two ports must be on the same crossbar for this call to work. The call returns `VOX_Status`.

4.4.2 The Device Layer

The device layer contains the low-level routines for handling specific types of crossbars. These routines are linked with and called by the higher layers (usually the generic layer).

Six routines are provided for each crossbar type by the device layer:

`reset()` takes a file descriptor as a parameter and resets the crossbar attached to the device specified by the descriptor. The routine returns a 0 on success, or `XBAR_FAILED` on a failure.

`open()` takes a device name as a parameter and calls `midi_open()` on the device. If the call succeeds, the file descriptor is returned. Otherwise, `XBAR_FAILED` is returned.

`close()` takes a file descriptor and closes the device.

`set()` takes a file descriptor, and an in and an out channel, and a state byte (which may be either `XBAR_ON` or `XBAR_OFF`). `set()` then connects the wires on the crossbar.

`out_query_dev()`

`out_query()`

`in_query()`

4.5 Crossbar Initialization

4.6 Crossbar Interconnectivity

Chapter 5

Programmer Interface

Revision History:

Author(s)	Date	Rev.	Comments
Binding	Dec. 12 88		Original Manual
Arons	Oct. 17 89	0.9	Reformat, cleaned up, rewrote.

5.1 Introduction

This document describes the programmer's interface to the VOX Audio Server. A description of the general architecture of the server can be found in [1].

The server is implemented in a single UNIX address space with clients communicating via the *Mercury* remote procedure call package [2]. Each call to the server is implemented as a remote procedure and returns a *status* value to indicate success or failure of the operation. The server is multi-threaded with one thread per client-server communication channel. Clients can take advantage of this by establishing several client-server connections at once. Client-server connections are referred to by the use of a *connection handle*.

During a session, clients create logical audio devices (LAUDs) and other audio abstractions on the server. These abstractions are accessed through a *handle* of the appropriate type. For example, instances of a recording device or a sound file are represented by their respective handles inside the client.

Associated with the VOX server is the VOX library. It contains a set of supporting calls for general usage. In particular, it exports functionality that is used to establish connection(s) to the server.

Caveat: This document is based on our experience with the current prototype implementation of VOX. The below interfaces do not address the issues of speech synthesis or speech recognition. We reserve the right to modify below described interfaces at our full discretion. Code fragments and data structures in this document may not always be synchronized with the current VOX release, so check the source code when in doubt.

5.2 Generalities

The *Mercury* stub generator redeclares some basic C-types:

```
typedef int Integer;
typedef char Char;
typedef short Short;
typedef char Boolean;
```

We shall use these basic data types throughout this document.

The server also introduces a few data types that are used in the communication with clients:

```
typedef struct {
   Integer length;
```

```
   Char sequence[32];
} VOX_String32;

typedef struct {
   Integer length;
   Char sequence[64];
} VOX_String64;

typedef struct {
   Integer length;
   Char sequence[128];
} VOX_String128;
```

The above data types are used to send and receive strings of characters that cannot exceed a predefined maximum length.

As with all *Mercury* sequences, it is important for the client to insure that `seq.length <= length(seq.sequence)`.

when calling the server; otherwise the RPC call involving these data types may fail. The VOX library declares some operations that assist in initializing such sequences (see section 5.15). Furthermore, it may be the case that a sequence is not large enough to hold a result parameter. In this case, a `VOX_Sequence-Overflow` error will be returned to the client[1].

VOX uses this data structure for time intervals[2]. The VOX library declares operations to support the `VOX_Duration` data type (see section 5.15).

```
typedef struct {
   Integer sec;
   Integer muSec;
} VOX_Duration;


typedef enum {
   VOX_Success,
   VOX_Failure,
   VOX_IllegalValue,
   VOX_NoSuchResource,
   VOX_EOQ,
   VOX_SequenceOverflow,
   VOX_EmptySequence,
```

[1] There is currently no way to gracefully handle such errors except to rebuild the server using larger sequences.

[2] And we were obviously inspired by BSD UNIX's `timeval`.

```
   VOX_Aborted,
   VOX_BadDevice,
   VOX_SoundExists,
   VOX_NewSound,
   VOX_EmptyBuffer,
   VOX_LastInterval,
   VOX_NotMapped,
   VOX_NestedLAUD,
   VOX_LockConflict,
   VOX_AlreadyOpen,
   VOX_NotOpen,
   VOX_NotOnSameLAUD,
   VOX_IncompatiblePorts,
   VOX_NoConnection,
   VOX_PortsAlreadyConnected,
   VOX_NoSuchEvent,
   VOX_BadEventType,
} VOX_Status;
```

Every server call returns one of the above status values. Note that `VOX_Success` is equivalent with the integer value 0; thus a programmer can write:

```
if (VOX_Call(...)) error(...);
```

The server defines a set of values that describe the reason for which an audio request terminated:

```
typedef enum {
   VOX_Unknown,
   VOX_InitialPause,     /* initial pause exceeded */
   VOX_FinalPause,       /* recording detected final pause */
   VOX_TimeOut,          /* event timed out */
   VOX_AbortedEv,        /* event was explicitly aborted */
   VOX_EOS,              /* end of sound */
   VOX_NonSilence,
   VOX_Receive_DTMF,
   VOX_LoopCurrentDrop,
   VOX_InterDigitDelay,
   VOX_DTMF_Terminator,
   VOX_PatternMatch,
   VOX_BufferCount
} VOX_Termination;
```

More details on the use of the `VOX_Termination` type can be found in sections 5.10, 5.11, and 5.13.

Every abstraction on the server is referred to by a *handle,* the individual sub-types are explained in individual sections of this document:

```
typedef Integer VOX_Handle;

typedef VOX_Handle EV_Event;
typedef VOX_Handle DEV_Device;
typedef VOX_Handle LAUD_LAUD;
typedef VOX_Handle Q_Queue;
typedef VOX_Handle P_Port;
typedef VOX_Handle BUF_Buffer;
typedef VOX_Handle SND_Sound;

/* sub-types of type LAUD_LAUD */
typedef VOX_Handle MON_Monitor;
typedef LAUD_LAUD PL_Player;
typedef LAUD_LAUD PH_Phone;
typedef LAUD_LAUD REC_Recorder;
typedef LAUD_LAUD SW_Switch;

typedef Integer VOX_Server;
```

It is possible for a client to query the server on the type of a handle:

```
typedef enum {
   VOX_Event,
   VOX_Device,
   VOX_Queue,
   VOX_Port,
   VOX_Buffer,
   VOX_Sound,
   VOX_LAUD,
   VOX_Monitor,
   VOX_Player,
   VOX_Phone,
   VOX_Recorder,
   VOX_Switch
} VOX_HandleType;

VOX_Status VOX_TypeOfHandle(vox, h, t)
```

```
VOX_Server vox;
VOX_Handle h;
VOX_HandleType *t;
```

The `VOX_TypeOfHandle()` call takes a handle value and returns the type of the handle in the argument `t`. If the server does not currently maintain an object associated with that handle, the error `VOX_No-Such-Resource` is returned; else the call returns `VOX_Success`.

5.3 The Event Data Type

Much of the information exchange between client and server is performed through the use of *events*, i.e. data that describes the input events or output requests to the client or the server.

Requests which originate on the client and which are eventually processed on the server are considered to be *output*. *Input* occurs asynchronously from the client's view point and transfers information from the server to the client. With these definitions, a play or record request is considered to be *output*, whereas an incoming telephone ring or speech recognition tokens are *input* events. Note that generally we use the term *event* for input and *request* for output, although in some places the term "event" may denote either.

The data structure describing input events and output requests is the `EV_EventUnion` union[3]. The `designator` tag field in that union type is of type `EV_Type` and identifies the origin of the event.

```
typedef enum {
   EV_Phone,
   EV_Monitor,
   EV_Play,
   EV_Record,
   EV_Switch
} EV_Type;

typedef struct {
   EV_Type designator;
   union {
      MON_Event u_EV_Monitor;
      PL_Event u_EV_Play;
      REC_Event u_EV_Record;
      PH_Event u_EV_Phone;
```

[3]The ugliness of that and various other data structures stems from generating these definitions with the *Mercury* stub generator.

```
   } u;
} EV_EventUnion;
```

Output requests are created by the client via the interfaces of section 5.10, 5.11, and 5.13. These calls return a handle of type `EV_Event` which refers to these requests.

Input and output events transit through I/O queues associated with a top-level LAUD. To filter out undesired events in a LAUD's I/O queues, it is possible to build masks of events types:

```
typedef Integer EV_TypeMask;
```

It is the client's responsibility to correctly create such masks. For instance, a mask that affects events originating from a phone or player LAUD:

```
mask = (1 << EV_Phone) | (1 << EV_Play);
```

See section 5.4, page 71 on how such masks are associated with I/O queues.

Clients can abort an output event that has already been queued for execution:

```
VOX_Status EV_Abort(vox, ev)
VOX_Server vox;
EV_Event ev;
```

If `ev` refers to a valid event handle, the event is either aborted or dequeued and will eventually be destroyed by the server. In case of an aborted recording, the partially recorded sound file is *not* deleted. If `ev` is not a valid event handle, the `VOX_No-Such-Event` error is returned. `VOX_Success` indicates successful completion of the operation.

To query the server on the status of an output request, the following data types and operation are of use:

```
#define EV_New 1
#define EV_Queued 2
#define EV_Active 4
#define EV_Done 8
#define EV_Aborted 16

typedef Integer EV_State;
```

```
typedef struct {
   EV_State state;
   VOX_Duration pos; /* only relevant for play/record requests */
} EV_Status;

VOX_Status EV_StatusOf(vox, ev, status)
VOX_Server vox;
EV_Event ev;
EV_Status *status;
```

Given a legal event handle `ev`, the call returns the current state of the event in `status`. The `state` field of the event's status is set to one a constant indicating the state of an event. Note that the `pos` field of the event's status is only relevant in the case of a play or record request where it indicates how far the request has progressed. If `ev` is not a valid event handle, the `VOX_No-Such-Event` error is generated. `VOX_Success` indicates successful completion of the operation.

It is possible to synchronize a client with the state transitions of an event:

```
typedef Integer EV_StateMask;

VOX_Status EV_WaitForState(vox, ev, mask)
VOX_Server vox;
EV_Event ev;
EV_StateMask mask;
```

The calling thread is suspended[4] until the event reaches one of the states specified in `mask`, where `mask` is a logical union of an event's states. `ev` must be a legal event handle or the `VOX_No-Such-Event` error is returned. `VOX_Success` indicates successful completion of the operation.

The client can query the server for the body of an event to which is has a handle.

```
VOX_Status EV_BodyOf(vox, ev, body)
VOX_Server vox;
EV_Event ev;
EV_EventUnion *body;
```

If `ev` is not a legal event handle, the operation returns a `VOX_NoSuchEvent` error and the value of `body` is undefined. `VOX_Success` indicates successful completion of the operation.

[4]This may be for ever!

5.4 The Queue Data Type

As mentioned, I/O events transit through I/O *queues* that are associated with top-level LAUDs and that support the "classical" queueing operations. In all of these operations, queues are referred to by a handle of type `Q_Queue`.

```
VOX_Status Q_Length(vox, queue, length)
VOX_Server vox;
Q_Queue queue;
Integer *length;
```

`Q_Length()` returns the `length` of the queue referred to by the `queue` argument. If `queue` is not a valid queue handle, a `VOX_No-Such-Resource` error is returned. `VOX_Success` indicates successful completion of the operation.

To flush a queue:

```
VOX_Status Q_Flush(vox, queue, dispose)
VOX_Server vox;
Q_Queue queue;
Boolean dispose;
```

If `dispose` is `true` and `queue` is an output queue, requests contained in the queue are deallocated; otherwise clients may re-enqueue the queue's output requests. If `queue` is an input queue, all input events are deallocated independently of the `dispose` argument. If `queue` is not a valid queue handle, a `VOX_No-Such-Resource` error is generated. `VOX_Success` indicates successful completion of the operation.

To selectively remove an output request from a queue[5]:

```
VOX_Status Q_Remove(vox, queue, ev, dispose)
VOX_Server vox;
Q_Queue queue;
EV_Event ev;
Boolean dispose;
```

If `dispose` is set, the event is deallocated on the server. Both `queue` and `ev` must be legal handles of the appropriate type; otherwise a `VOX_No-Such-Resource` or `VOX_No-Such-Event` error

[5]Input events become not known to the client before a `get()` call which returns not just the handle, but the entire event data.

is generated. If the named queue is empty, a `VOX_Failure` error is returned. If the event is not in the queue, a `VOX_No-Such-Event` error is returned. `VOX_Success` indicates successful completion of the operation.

To enqueue an output request use:

```
VOX_Status Q_Put(vox, queue, ev)
VOX_Server vox;
Q_Queue queue;
EV_Event ev;
```

The named event `ev` must have been created previously (see sections 5.10, 5.11, 5.13), otherwise a `VOX_No-Such-Event` error status is returned. If `queue` is not a legal queue handle, the `VOX_No-Such-Resource` error is returned. `VOX_Failure` is returned if `queue` is not an output queue, the queue is full[6], the event is already in a queue or the queue is temporarily disabled (see `LAUD_Pause`). `VOX_EOQ` is returned if the queue has been destroyed and does no longer accept output events. `VOX_Success` indicates successful completion of the operation.

If the queue's mask disallows events with `ev`'s event type (also see pages 68 and 71), a `VOX_BadEventType` error is returned and the event is *not* enqueued.

Input from a queue can be obtained by:

```
VOX_Status Q_Get(vox, queue, ev)
VOX_Server vox;
Q_Queue queue;
EV_EventUnion *ev;

VOX_Status Q_Peek(vox, queue, ev)
VOX_Server vox;
Q_Queue queue;
EV_EventUnion *ev;
```

Note that both calls return an `EV_EventUnion` and *not* simply an event handle. `Q_Peek` does not remove the event from its queue, `Q_Get()` does. Both calls block the calling thread until input becomes available. If `queue` is not a valid queue handle, the `VOX_No-Such-Resource` error is generated. If `queue` is not an input queue, a `VOX_Failure` error is returned. `VOX_Success` indicates successful completion of the operations.

With each I/O queue the server associates a *filtering mask* that controls the type of events that transit through the queue (see page 68).

[6]Currently, VOX queues may contain up to 64 events or requests.

```
VOX_Status Q_SetMask(vox, queue, mask, oldMask)
VOX_Server vox;
Q_Queue queue;
EV_TypeMask mask, *oldMask;

VOX_Status Q_GetMask(vox, queue, mask)
VOX_Server vox;
Q_Queue queue;
EV_TypeMask *mask;
```

The `Q_SetMask()` call uses `mask` as the new mask and returns the previously used event filtering mask in `oldMask`. `Q_GetMask()` queries the server on the queue's current mask. Both calls return `VOX_No-Such-Resource` if `queue` is not a legal queue handle. `VOX_Success` indicates successful completion of the operations.

The use of a queue's filtering mask is as follows:

> *Output*: enqueueing operations by the client fail and a `VOX_Failure` status is reported if the submitted request is of the wrong type.
>
> *Input*: the client never sees input events of the wrong type since the server does not enqueue events internally.

5.5 The Device Data Type

VOX manages a set of audio *devices* that are manipulated through use of the data types and operations described in this section.

Devices are referred to by either their name or their handle of type `DEV_Device`.

```
typedef struct {
   VOX_String64 major;
   Short minor;
} DEV_Name;

#define DEV_SeqSize 64

typedef struct {
   Integer length;
   DEV_Device sequence[DEV_SeqSize];
} DEV_Sequence;
```

`DEV_Name` is used to name a device; note that a device name consists of a major string name (maximum length is 64) and a minor index. The `DEV_Sequence` structure is used to transmit a sequence of device handles between client and server.

Operations on devices are:

```
VOX_Status DEV_DevicesOn(vox, devs)
VOX_Server vox;
DEV_Sequence *devs;
```

The `DEV_DevicesOn()` call returns the set of device handles that are known to the instance of VOX denoted by `vox`. The call returns a `VOX_Sequence-Overflow` error if there are more than 64 devices on the server; else a status of `VOX_Success` is returned.

Given a device handle, the client can query the server about the device's status (see also [3]):

```
typedef struct {
   DEV_Name name;
   Boolean mapped;
   Boolean locked;
   LAUD_LAUD mappedTo;
} DEV_Status;

VOX_Status DEV_StatusOf(vox, device, status)
VOX_Server vox;
DEV_Device device;
DEV_Status *status;
```

The device status is described with the `DEV_Status` structure. `mapped` is set if a LAUD is currently using the device; `locked` is set if the LAUD has exclusive access to the device. The `mappedTo` field indicates which LAUD, if any, is using the device. The `DEV_StatusOf()` call may return a `VOX_No-Such-Resource` status if the device handle is illegal; otherwise `VOX_Success` is returned.

It may be the case that various devices cannot operate simultaneously on the server. To query the server about the set of devices conflicting with one particular device, the client may use the `DEV_Conflicts()` call:

```
VOX_Status DEV_Conflicts(vox, device, conflicts)
VOX_Server vox;
DEV_Device device;
DEV_Sequence *conflicts;
```

The `conflicts` sequence indicates which devices conflict with `device`. The call may return a `VOX_No-Such-Resource` error if `device` does not denote a legal device handle. If too many conflicts exist, a `VOX_Sequence-Overflow` is generated. `VOX_Success` is returned in case of successful completion of the call.

5.6 The Port Data Type

Logical audio devices (LAUDs) may have audio *ports* associated with them (see section 5.7). This section discusses the data types and operations associated with the `Port` data type. Ports are represented through their handles of type `P_Port`.

A port's status is a triple consisting of the port's string `name`, its `level`, and its `direction`. VOX uses the following data types to describe a port's status:

```
typedef enum {
   P_Mic,
   P_Speaker,
   P_Line,
   P_RJ11,
   P_Video,
   P_RF_Video
} P_Level;

typedef enum {
   P_In,
   P_Out,
   P_InOut
} P_Direction;

typedef struct {
   LAUD_LAUD laud;
   VOX_String32 name;
   P_Level level;
   P_Direction dir;
} P_Status;

#define P_SeqSize 16

typedef struct {
   Integer length;
   P_Port sequence[P_SeqSize];
} P_Sequence;
```

Note that the port-LAUD association is indicated in the `laud` field of the `P_Status` structure.

To query a given port on its current status, use the `P_StatusOf()` call:

```
VOX_Status P_StatusOf(vox, port, status)
VOX_Server vox;
P_Port port;
P_Status *status;
```

The `P_StatusOf()` call returns a `VOX_No-Such-Resource` error if `port` does not indicate a legal port; else `VOX_Success` is returned.

Ports on a given LAUD can be interconnected, provided the hardware supports the actual connection. To query the legality of a connection use the `P_Connectivity()` call:

```
VOX_Status P_Connectivity(vox, port, connections)
VOX_Server vox;
P_Port port;
P_Sequence *connections;
```

The set of ports to which `port` can be connected is returned in `connections`. Note that all of the returned ports reside on the same CLAUD as `port`. The call returns a `VOX_No-Such-Resource` error if `port` indicates an illegal port handle. If a sequence overflow occurs, a `VOX_Sequence-Overflow` error is returned. In case of success, `VOX_Success` is returned.

To find out to which ports a given port is currently connected, use the `P_Connections()` call:

```
VOX_Status P_Connections(vox, port, connections)
VOX_Server vox;
P_Port port;
P_Sequence *connections;
```

This call generates the identical errors as the `P_Connectivity()` call.

Ports can have *aliases,* that is, the same physical port can be named by different string names and consequently have different handles.

```
VOX_Status P_Rename(vox, port, laud, name, alias)
VOX_Server vox;
P_Port port;
```

```
LAUD_LAUD laud;
VOX_String32 name;
P_Port *alias;
```

The `port` argument identifies a port on some LAUD and the call attempts to create an `alias` port with the given `name` on the LAUD `laud`. Note that `laud` must be an ancestor (parent) of the LAUD with which `port` is associated. If not, a `VOX_NotOnSameLAUD` error is generated. The `VOX_No-Such-Resource` error is returned if either `port` or `laud` are illegal handles. In case of success, the `VOX_Success` status is returned.

Note that the `P_Connections` and `P_Connectivity()` calls *include* port aliases in their result argument. That is, if port 1 is connected to port 2 which has aliases 2 2 , the connection set includes 2 2 2 .

To create and destroy connections between ports on a same composite LAUD use:

```
VOX_Status P_Solder(vox, port1, port2)
VOX_Server vox;
P_Port port1, port2;

VOX_Status P_UnSolder(vox, port1, port2)
VOX_Server vox;
P_Port port1, port2;
```

`P_Solder` creates a connection between `port1` and `port2`, if such a connection can be created. The call returns `VOX_No-Such-Resource` if either port handle is illegal. If the ports are *not* on the same CLAUD, a `VOX_NotOnSameLAUD` error is returned. The CLAUD on which the connection is created may *not* be mapped, otherwise a `VOX_Failure` error is returned. If the ports are incompatible due to mismatching level or direction values, a `VOX_IncompatiblePorts` error is returned. If one of either ports is already connected to a third port and it is an exclusive connection, a `VOX_PortsAlreadyConnected` error is returned. If no physical connection between the given ports can be created, a `VOX_No-Connection` error is returned. In case of success, `VOX_Success` is returned.

`P_UnSolder` does the reverse of `P_Solder`. Both ports must exist, otherwise a `VOX_No-Such-Resource` error is returned. The LAUD on which both ports reside may *not* be mapped; a `VOX_Failure` error is otherwise generated.

5.7 The LAUD Data Type

The *LAUD* data type is truly the central data type in VOX; it represents the logical abstractions of audio devices that most applications will use. Within the client, LAUDs are referred to by handles

of type LAUD_LAUD.

5.7.1 LAUD Utility Calls

A data definition regarding this data type used for RPC argument passing is:

```
#define LAUD_SeqSize 64

typedef struct {
   Integer length;
   LAUD_LAUD sequence[LAUD_SeqSize];
} LAUD_Sequence;
```

which declares a sequence of LAUD handles.

```
VOX_Status LAUD_LAUDsOnServer(vox, lauds)
VOX_Server vox;
LAUD_Sequence *lauds;
```

The LAUD_LAUDsOnServer() call returns the set of lauds currently known to the server. A VOX_Sequence-Overflow error is generated if there are more LAUD instances than the lauds sequence can contain. Successful completion is indicated by the VOX_Success status.

To query a given LAUD on the set of devices it uses, use the LAUD_DevicesOf() call:

```
VOX_Status LAUD_DevicesOf(vox, laud, devices)
VOX_Server vox;
LAUD_LAUD laud;
DEV_Sequence *devices;
```

It returns the set of device handles in devices that are used by laud when the LAUD becomes mapped. Note that conflicting devices of other LAUDs are *not* included in the devices sequence. If laud is an illegal handle, the call returns VOX_No-Such-Resource. The call returns a VOX_Sequence-Overflow error if the devices sequence is too short. If laud is a component of a composite LAUD (i.e. a CLAUD), a VOX_NestedLAUD status is returned and the call implicitly uses the top-level LAUD associated with laud. Success is indicated by a return of VOX_Success.

Since devices can conflict, LAUDs that use these devices may also conflict. To determine which LAUDs conflict with a given LAUD use:

```
VOX_Status LAUD_Conflicts(vox, laud, conflicts)
VOX_Server vox;
LAUD_LAUD laud;
LAUD_Sequence *conflicts;
```

The set of conflicting LAUDs is returned in `conflicts`. If `laud` is not a legal handle, `VOX_No-Such-Resource` is returned. If the `conflicts` sequence is too short, `VOX_Sequence-Overflow` is returned. If `laud` is not a top-level LAUD, a `VOX_NestedLAUD` error is generated and VOX implicitly uses the top-level LAUD associated with `laud`. `VOX_Success` indicates successful completion of the operation.

To query a LAUD about its set of ports, the client may use the `LAUD_Ports()` call:

```
VOX_Status LAUD_Ports(vox, laud, ports)
VOX_Server vox;
LAUD_LAUD laud;
P_Sequence *ports;
```

`laud` identifies a LAUD instance. The handles to all the ports that reside on `laud` are returned in `ports`. If `laud` refers to an invalid LAUD handle, a `VOX_No-Such-Resource` error is returned. If `ports` is not large enough to hold the entire set of ports, a `VOX_Sequence-Overflow` status is returned. If `laud` refers to a nested LAUD, the operation implicitly applies to the top-level LAUD and a `VOX_NestedLAUD` status is returned. `VOX_Success` indicates successful completion of the operation.

```
VOX_Status LAUD_PortOf(vox, laud, portName, port)
VOX_Server vox;
LAUD_LAUD laud;
VOX_String32 portName;
P_Port *port;
```

To query a LAUD's port by name, the client uses the `LAUD_PortOf()` call. `laud` refers to a LAUD and `portName` is the name of the port whose handle is returned in `port`. `VOX_Success` indicates successful completion of the operation. If `laud` refers to an invalid LAUD handle, a `VOX_No-Such-Resource` error is returned. If `laud` refers to a nested LAUD, the operation implicitly applies to the top-level LAUD and a `VOX_NestedLAUD` status is returned. If no port of name `portName` exists on the LAUD `laud`, a `VOX_No-Such-Resource` status is returned.

5.7.2 Creating and Using LAUDS

Composite LAUDs (CLAUDs) are built by using the following server calls:

```
VOX_Status LAUD_Create(vox, laud)
VOX_Server vox;
LAUD_LAUD *laud;

VOX_Status LAUD_Insert(vox, laud1, laud2)
VOX_Server vox;
LAUD_LAUD laud1, laud2;

VOX_Status LAUD_Assemble(vox, laud)
VOX_Server vox;
LAUD_LAUD laud;
```

To create a new, empty composite LAUD, the client uses the `LAUD_Create()` call which returns a handle to a newly created CLAUD. Unless memory allocation on the server starts to fail, this call always returns `VOX_Success`.

To insert a LAUD into a CLAUD, or one CLAUD hierarchy into another CLAUD hierarchy, the client calls the `LAUD_Insert()` operation to insert `laud2` into `laud1`. The call fails if:

`VOX_No-Such-Resource`: `laud1` or `laud2` do not refer to legal LAUD handles.

`VOX_Failure`:

- `laud2` is already part of a CLAUD.
- `laud1` and `laud2` use conflicting devices[7].
- `laud1` is an already assembled CLAUD.

To finalize the assembling process of a CLAUD, the client *must* call the `LAUD_Assemble()` operation. If `laud` does not refer to a legal LAUD handle, the `VOX_No-Such-Resource` error status is returned. If `laud` is a nested LAUD, the operation is implicitly applied to the top-level LAUD and the `VOX_NestedLAUD` status is returned. `VOX_Success` is returned for successful completion of the operation.

```
VOX_Status LAUD_Destroy(vox, laud)
VOX_Server vox;
LAUD_LAUD laud;
```

This call recursively destroys a LAUD hierarchy, starting at the top-level LAUD associated with `laud`. If `laud` is not a legal LAUD handle, a `VOX_No-Such-Resource` error is returned. If

[7]Since then VOX may not be able to map the resulting LAUD.

`laud` is not a top-level LAUD, the call implicitly uses the top-level LAUD associated with `laud` and returns a `VOX_NestedLAUD` status. Success is indicated by a return value of `VOX_Success`.

Over the life-time of a LAUD, the client can query the status of a given LAUD:

```
#define LAUD_New 1
#define LAUD_Assembled 2
#define LAUD_Suspended 4
#define LAUD_Mapped 8
#define LAUD_Locked 16

typedef Integer LAUD_State;

VOX_Status LAUD_StatusOf(vox, laud, status)
VOX_Server vox;
LAUD_LAUD laud;
LAUD_Status *status;
```

The `LAUD_StatusOf()` call returns the current status mask of the LAUD named by `laud`. If no such LAUD exists on the server, a `VOX_No-Such-Resource` error is returned. If `laud` is a nested LAUD, the operation implicitly applies to the top-level LAUD and the `VOX_NestedLAUD` status is returned. `VOX_Success` indicates successful completion of the operation.

```
VOX_Status LAUD_Map(vox, laud, lock, block)
VOX_Server vox;
LAUD_LAUD laud;
Boolean lock, block;

VOX_Status LAUD_Unmap(vox, laud)
VOX_Server vox;
LAUD_LAUD laud;
```

The `LAUD_Map()` call attempts to map a composite LAUD onto its devices, thereby starting the execution of I/O requests that are stored in the LAUD's output queue. The role of the `lock` and `block` arguments is as follows:

> `lock`: If the LAUD is mapped as *locked*, no other client can use LAUDs that conflict with `laud` until the LAUD has been either unmapped or unlocked. If the LAUD is *not* locked, it will be unmapped whenever another client maps a (different) LAUD to some device also used by `laud`.
>
> `block`: This argument is only used if `lock = true` and indicates whether or not the client is willing to wait for the LAUD's lock. If `block = false` and the components of the CLAUD are already locked, a `VOX_LockConflict` status value is returned.

If `laud` is not a legal LAUD handle, a `VOX_No-Such-Resource` status is returned. If `laud` is a nested LAUD, the operation is implicitly applied to the top-level LAUD and the `VOX_NestedLAUD` status is returned. `VOX_Success` indicates successful completion of the operation.

The inverse of the `LAUD_Map()` operation is the `LAUD_Unmap()` call. It is used to break the mapping of a LAUD to its devices and, if the LAUD was locked, releases the lock. Any executing I/O events at the time of the call are aborted prior to the unmapping. If `laud` is not a legal LAUD handle, a `VOX_No-Such-Resource` status is returned. If `laud` is a nested LAUD, the operation is implicitly applied to the top-level LAUD and the `VOX_NestedLAUD` status is returned. `VOX_Success` indicates successful completion of the operation.

In addition to being able to lock the devices in the map call, it is possible to acquire exclusive use of a LAUD's devices at a later time by using the `LAUD_Lock()` call:

```
VOX_Status LAUD_Lock(vox, laud)
VOX_Server vox;
LAUD_LAUD laud;

VOX_Status LAUD_Unlock(vox, laud)
VOX_Server vox;
LAUD_LAUD laud;
```

Note that the `LAUD_Lock()` call *never* blocks. This, because it only succeeds if the `laud` is currently mapped to its devices in which case the lock can be acquired without delay. If the LAUD is not mapped, the `VOX_Failure` error status is returned. If `laud` is not a legal LAUD handle, a `VOX_No-Such-Resource` status is returned. If `laud` is a nested LAUD, the operation is implicitly applied to the top-level LAUD and the `VOX_NestedLAUD` status is returned. `VOX_Success` indicates successful completion of the operation.

The `LAUD_Unlock()` call releases a LAUD's exclusive device access. It returns the identical error status values as the `LAUD_Lock()` call.

It is possible to temporarily suspend the processing of I/O requests on a named LAUD by calling the `LAUD_Pause()` operation.

```
VOX_Status LAUD_Pause(vox, laud, unmap, keep)
VOX_Server vox;
LAUD_LAUD laud;
Boolean unmap, keep;

VOX_Status LAUD_Resume(vox, laud, lock, block)
VOX_Server vox;
```

```
LAUD_LAUD laud;
Boolean lock, block;
```

If unmap is set, the devices associated with laud are released and the currently executing I/O request is suspended. (In the current implementation of VOX, information contained in the LAUD's device's internal buffer is discarded between a pause/resume action.) The keep argument indicates whether or not the LAUD accepts new I/O requests while it is suspended: if keep = false, I/O requests are *not* buffered in the LAUD's queues while it is suspended (see LAUD_Put(), Q_Put() and LAUD_Pause()). The call returns VOX_No-Such-Resource if laud is not a legal LAUD handle. If laud refers to a nested LAUD, the operation is implicitly applied to the top-level LAUD and a VOX_NestedLAUD status is returned. VOX_Success indicates successful completion of the operation.

To resume the I/O operations of a suspended LAUD, the client calls the LAUD_Resume() operation. Its semantics are similar to the LAUD_Map() call, with the exception that a suspended I/O request is resumed. The call returns the same error status values as LAUD_Map().

5.7.3 LAUD I/O

To find out which I/O request is currently executing on a given LAUD use the LAUD_Current() call:

```
VOX_Status LAUD_Current(vox, laud, event)
VOX_Server vox;
LAUD_LAUD laud;
EV_Event *event;
```

The call returns the handle to the event that is currently executing on the LAUD laud. If laud is not a legal LAUD handle, a VOX_No-Such-Resource status is returned. If laud is a nested LAUD, the operation is implicitly applied to the top-level LAUD and the VOX_NestedLAUD status is returned. VOX_Success indicates successful completion of the operation.

I/O requests are created by operations of sections 5.10, 5.11, or 5.13 and may be queued for execution with the LAUD_Put() operation:

```
VOX_Status LAUD_Put(vox, laud, event)
VOX_Server vox;
LAUD_Laud laud;
EV_Event event;
```

Both `laud` and `event` must be handles of the correct types; otherwise a `VOX_No-Such-Resource` error is returned. The `LAUD_Put()` operation enqueues an *output* request for processing. The `LAUD_Put()` operation returns a `VOX_Failure` status if the LAUD named by `laud` has no associated output queue or the enqueuing operation fails since the queue is full, output is disabled (see `LAUD_Pause()`), or the event is already in an output queue. If the event is not an output request, `VOX_BadEventType` is returned. If `laud` refers to a nested LAUD, the operations is implicitly applied to the top-level LAUD and a `VOX_NestedLAUD` status is returned. `VOX_Success` indicates successful completion of the operation.

```
VOX_Status LAUD_Get(vox, laud, event)
VOX_Server vox;
LAUD_Laud laud;
EV_EventUnion *event;
```

To query a LAUD for input events, the client may use the `LAUD_Get()` call. It returns the value of the event in the `event` of type `EV_EventUnion`[8] (for a description of that data type, see page 68). The call is *blocking* until an input event can be read from the named LAUD. If `laud` refers to a nested LAUD, the call implicitly applies to the top-level LAUD and a `VOX_NestedLAUD` status is returned. If `laud` refers to an illegal LAUD handle, the call returns with a `VOX_No-Such-Resource` status value. If the top-level LAUD has no associated input queue, the call fails with `VOX_Failure`. If the LAUD is destroyed while the client is waiting for input, the call returns with a `VOX_EOQ` status value. `VOX_Success` indicates successful completion of the operation.

The server also provides operations to directly access I/O queues (see section 5.4). To obtain the handle of a LAUD's I/O queue use:

```
typedef enum {
   LAUD_InQueue,
   LAUD_OutQueue
} LAUD_QueueKind;

VOX_Status LAUD_Queue(vox, laud, which, queue)
VOX_Server vox;
LAUD_LAUD laud;
LAUD_QueueKind which;
Q_Queue *queue;
```

`laud` identifies the LAUD instance under consideration. `which` defines whether the input queue handle or the output queue handle is returned in `queue`. If `laud` is an invalid LAUD handle,

[8] Note that this is *not* an event handle!

the VOX_No-Such-Resource error is returned. If laud refers to a nested LAUD, the operation is implicitly applied to the top-level LAUD and a VOX_NestedLAUD status is returned. If the LAUD does not have a queue of the given kind, a VOX_Failure error is returned. VOX_Success indicates successful completion of the operation.

5.7.4 Directories and Files

In addition to the audio properties of a LAUD, it also serves as the storage abstraction for sound files and therefore associates the notion of a *current directory* with each top-level LAUD. (see also section 5.8.)

In the current implementation of VOX, a LAUD's directory is equivalent to a UNIX directory. The default directory of LAUDs is passed via the UNIX environment variable SND_DIR. Note that directory names *must* end with a "/" character.

```
VOX_Status LAUD_ChangeDirectory(vox, dir, oldDir)
VOX_Server vox;
VOX_String128 dir, *oldDir;
```

To change a LAUD's current directory, the client uses the LAUD_ChangeDirectory() call. dir is the *absolute* path name of the new directory; the old value of the LAUD's directory is returned in oldDir. VOX_Success indicates successful completion of the operation. If laud refers to an invalid LAUD handle, a VOX_No-Such-Resource error is returned. If laud refers to a nested LAUD, the operation implicitly applies to the top-level LAUD and a VOX_NestedLAUD status is returned.

```
VOX_Status LAUD_CurrentDirectory(vox, laud, dir)
VOX_Server vox;
LAUD_LAUD laud;
VOX_String128 *dir;
```

The LAUD_CurrentDirectory() call returns the name of the LAUD's current directory in dir[9]. VOX_Success indicates successful completion of the operation. If laud refers to an invalid LAUD handle, a VOX_No-Such-Resource error is returned. If laud refers to a nested LAUD, the operation implicitly applies to the top-level LAUD and a VOX_NestedLAUD status is returned.

```
VOX_Status LAUD_DirectorySize(vox, laud, size)
VOX_Server vox;
LAUD_LAUD laud;
Integer *size;
```

[9]Note that directory names currently can not exceed 128 characters.

To query the LAUD for the number of sound files that are contained in its current directory, use the `LAUD_DirectorySize()` call. It returns the number of sound files in `size`. `VOX_Success` indicates successful completion of the operation. If `laud` refers to an invalid LAUD handle, a `VOX_No-Such-Resource` error is returned. If `laud` refers to a nested LAUD, the operation implicitly applies to the top-level LAUD and a `VOX_NestedLAUD` status is returned.

```
VOX_Status LAUD_NameOf(vox, laud, fileIndex, fileName)
VOX_Server vox;
LAUD_LAUD laud;
Integer fileIndex;
VOX_String64 *fileName;
```

The `LAUD_NameOf()` call allows the client to query for a sound file's string name given it's rank (index) in the LAUD's current directory. `laud` is a handle to a LAUD, `fileIndex` is the rank of the sound file in `laud`'s current directory, and `fileName` is set to that file's string name[10]. `VOX_Success` indicates successful completion of the operation. If `laud` refers to an invalid LAUD handle, a `VOX_No-Such-Resource` error is returned. If `fileIndex` is not a legal value, the `VOX_IllegalValue` status is returned. If `laud` refers to a nested LAUD, the operation implicitly applies to the top-level LAUD and a `VOX_NestedLAUD` status is returned.

5.8 The Sound Data Type

The *sound* abstraction represent sound data stored in sound files.

To obtain access to a sound, the client uses the `SND_Open()` call:

```
VOX_Status SND_Open(vox, laud, fileName, sound)
VOX_Server vox;
LAUD_LAUD laud;
VOX_String64 fileName;
SND_Sound *sound;
```

`laud` identifies the LAUD in whose current directory the sound file is located. If no such sound currently exists, a new, empty sound is created, its handle returned in `sound`, and the `VOX_NewSound` status is returned. If the named sound exists, `sound` refers to that sound, and a `VOX_SoundExists` status is returned. If `fileName` is an empty sequence, a `VOX_EmptySequence` error is returned. If `laud` refers to an invalid LAUD handle, a

[10]Note that since file names can be at most 64 characters long and since directory names are at most 128 characters, full sound file names are at most 192 (printable) characters long.

VOX_No-Such-Resource error is returned. If laud refers to a nested LAUD, the operation implicitly applies to the top-level LAUD and a VOX_NestedLAUD status is returned. If the named sound cannot be opened, a VOX_Failure status is returned. VOX_Success indicates successful completion of the operation.

To break the association between a sound handle and the actual sound file, the client uses the SND_Close() operation:

```
VOX_Status SND_Close(vox, sound)
VOX_Server vox;
SND_Sound sound;
```

After successful completion of the call, sound no longer refers to an (open) sound file. If sound is not a legal sound handle *prior* to the operation, a VOX_No-Such-Resource error is returned. VOX_Success indicates successful completion of the operation.

To copy one sound onto another one, use:

```
VOX_Status SND_Copy(vox, from, laud, name, to)
VOX_Server vox;
SND_Sound from;
LAUD_LAUD laud;
VOX_String64 name;
SND_Sound *to;
```

from identifies the sound to be copied into the current directory of laud with the string name. The resulting sound is returned in the to sound handle. If from or laud identify non-existing resources, the VOX_No-Such-Resource error is returned. If name is an empty sequence, the VOX_EmtpySequence error is returned. If from refers to an empty sound, a VOX_NewSound status is returned. If laud refers to a nested LAUD, the operation is implicitly applied to the top-level LAUD and the VOX_NestedLAUD status is returned. VOX_Success indicates successful completion of the operation.

```
VOX_Status SND_Rename(vox, sound, name)
VOX_Server vox;
SND_Sound sound;
VOX_String64 name;
```

This call renames the sound file within its containing directory. name identifies the sound's new name[11]. name may not be empty, or a VOX_EmptySequence status is returned. If sound does

[11]The old name is *not* returned.

not refer to a sound object, a VOX_No-Such-Resource error is returned. If the rename operation fails, a VOX_Failure error status is returned. VOX_Success indicates successful completion of the operation.

To remove (delete) a file, the VOX server provides the SND_Remove() operation:

```
VOX_Status SND_Remove(vox, sound)
VOX_Server vox;
SND_Sound sound;
```

sound identifies the sound to be removed. If sound is not a valid sound handle, the VOX_No-Such-Resource error is returned. If the remove operation fails for unknown reasons, the VOX_Failure error status is reported. VOX_Success indicates successful completion of the operation.

VOX defines a "standard" format for sound files as described fully in the files vox.h and sound.h.

```
typedef enum {
   SND_PCM,    /* pulse code modulation */
   SND_ADPCM,  /* adaptive pulse code modulation */
   SND_CVSD,   /* continuously variable slope delta modulation */
   SND_LPC     /* linear prediction encoding */
} SND_Encoding;

typedef enum {
   SND_None,
   SND_Suppressed,
   SND_Eliminated
} SND_SilenceHandling;

typedef struct {
   VOX_String64 host;          /* host name on which sound recorded */
   VOX_String64 name;          /* sound name */
   VOX_String128 directory;    /* directory name in which sound is */
   VOX_Duration date;          /* sound creation date */
   VOX_String64 make;          /* make of creating device */
   VOX_String64 model;         /* model of creating device */
   Integer sampleRate;         /* in Hz */
   Integer sampleSize;         /* bits per sample */
   Integer packetSize;         /* in bytes per packet */
   Integer packetHeaderSize;   /* in bytes per packet header */
   Integer nbrOfPackets;       /* size of sound file in # packets */
```

```
   VOX_Duration length;       /* duration of sound */
   SND_Encoding coding;       /* coding algorithm */
   SND_SilenceHandling handling;  /* silence handling (see below) */
   Integer threshold;
} SND_Header;

VOX_Status SND_HeaderOf(vox, sound, header)
VOX_Server vox;
SND_Sound sound;
SND_Header *header;
```

Most of the fields in the `SND_Header` structure are self-describing. Note that in keeping with UNIX conventions, the `date` field indicates the creation time relative to January 1, 1970.

The `SND_Encoding` type describes the encoding algorithm used for a sound file. The precise algorithms used depend of course on the devices used by a VOX server; the intention here is to provide a set of commonly used encoding algorithms.

Note that the possible encodings and handling of silence is dependent on the actual devices used by VOX, e.g. not all devices may deal gracefully with all of the above silence handling algorithms (see also section 5.11).

The `SND_SilenceHandling` type enumerates the various silence handling available on the server:

1. `SND_None`: No silence handling is performed; the sound file contains all recorded silence.
2. `SND_Suppressed`: This is similar to a "run-length" encoding of silence where an interval of successive silence packets is compressed into a single packet with an associated repeat count. Upon playback, the reproducing device can thus regenerate the silence interval, but space is saved in the sound file.
3. `SND_Eliminated`: In this scheme, silence is eliminated from the recording and can not be regenerated upon playback.

```
VOX_Status      SND_HeaderOf(sound, info)
    SND_Sound            sound;
    register SND_Header *info;
```

The `SND_HeaderOf()` operation queries the server on a sound's header information. `sound` must be a legal sound handle and, upon successful return, `header` contains the sound's header information. If `sound` is not a valid sound handle, the `VOX_No-Such-Resource` error is returned. `VOX_Success` indicates successful completion of the operation.

5.9 The Buffer Data Type

The *buffer* data type implements VOX's notion of a sound editing abstractions. Buffers contain sound data and support traditional editing operation such as cut, paste, and copy.

The current implementation assumes packetized sound data, and editing is only supported at the granularity of one packet. Thus, all time values used in the editing operations are rounded down to the closest packet boundary. A further restriction of the present VOX implementation is that buffers involved in the editing operations must be recorded and played back on the same device. That is, currently one cannot paste a sound recorded on device type into another sound recorded on device type .

To create an empty buffer, use:

```
VOX_Status BUF_Create(vox, buffer)
VOX_Server vox;
BUF_Buffer *buffer;
```

`buffer` is a handle to the newly created, empty buffer. This operation always returns a `VOX_Success` error status.

To load a newly created buffer with some pre-recorded data, use the `BUF_Load()` operation:

```
VOX_Status BUF_Load(vox, dir, name, buffer)
VOX_Server vox;
VOX_String128 dir;
VOX_String64 name;
BUF_Buffer *buffer;
```

The `dir` identifies the directory on the server in which the sound file identified by `name` is located. The operation returns `VOX_Failure` if the named sound file cannot be loaded by the server or if the name refers to an empty sound file. `VOX_Success` indicates successful completion of the operation.

The inverse of the `BUF_Load()` operation is the `BUF_Store()` call:

```
VOX_Status BUF_Store(vox, buffer, dir, name)
VOX_Server vox;
BUF_Buffer buffer;
VOX_String128 dir;
VOX_String64 name;
```

The contents of the buffer referred to by `buffer` is stored in a sound file in directory `dir` with the given `name`. If `buffer` does not refer to a valid buffer handle, a `VOX_No-Such-Resource` error is returned. If the buffer is empty, a `VOX_Empty-Buffer` error is generated and no sound file is created. If the given `dir` and `name` combination would overwrite another sound file already in use by VOX's editor, a `VOX_Illegal-Value` error is returned. If the store operation fails for any other reason, a `VOX_Failure` error is returned. `VOX_Success` indicates successful completion of the operation.

To destroy a buffer, the server provides the `BUF_Destroy()` operation:

```
VOX_Status BUF_Destroy(vox, buffer)
VOX_Server vox;
BUF_Buffer buffer;
```

If `buffer` refers to a valid buffer handle, the server frees any storage associated with that buffer; otherwise a `VOX_No-Such-Resource` error is returned. Note that if the buffer has previously been *stored*, the associated sound file is *not* affected. However, the operation does *not* automatically perform a store. `VOX_Success` indicates successful completion of the operation.

To remove material of a sound buffer, VOX supports the `BUF_Cut()` operation:

```
VOX_Status BUF_Cut(vox, oldBuf, start, stop, newBuf)
VOX_Server vox;
BUF_Buffer oldBuf;
VOX_Duration start, stop;
BUF_Buffer *newBuf;
```

This operation removes the sound data located in the interval `start` to `stop` in the buffer referred to by `oldBuf` and fills a newly created buffer `newBuf` with that material. If `oldBuf` is not a valid buffer handle a `VOX_No-Such-Resource` error is returned. If `oldBuf` is an empty buffer, the `VOX_EmptyBuffer` error status is returned. If `start` is greater than the total length of the buffer `oldBuf` a `VOX_IllegalValue` error is generated. If `stop` exceeds the total length of the buffer `oldBuf`, its value is truncated to the end of the buffer. In case of an error, `newBuf` is not set to a valid buffer handle. `VOX_Success` indicates successful completion of the operation.

To duplicate a piece of one buffer into a new buffer:

```
VOX_Status BUF_Dup(vox, oldBuf, start, stop, newBuf)
VOX_Server vox;
BUF_Buffer oldBuf;
VOX_Duration start, stop;
BUF_Buffer *newBuf;
```

This operation's arguments and return values are identical with the above `BUF_Cut()` operation. The difference lies in the fact that `BUF_Dup()` does *not* remove any sound material from `oldBuf`.

To insert sound material into an existing, possibly empty, buffer, a client uses the `BUF_Paste()` call:

```
VOX_Status BUF_Paste(vox, source, dest, where)
VOX_Server vox;
BUF_Buffer source, dest;
VOX_Duration where;
```

The entire contents of the `source` buffer is inserted into the `dest` buffer at the insertion point `where`. If either `source` or `dest` are not legal buffer handles, a `VOX_No-Such-Resource` error is returned. If `where` is beyond the destination's total length, the insertion takes place at the end of the `dest` buffer (i.e. it is an append). `VOX_Success` indicates successful completion of the operation.

The combination of the `BUF_Dup()` and `BUF_Paste()` operation is the `BUF_Copy()` operation:

```
VOX_Status BUF_Copy(vox, source, dest, start, stop, where)
VOX_Server vox;
BUF_Buffer source, dest;
VOX_Duration start, stop, where;
```

The material between `start` to `stop` from buffer `source` is copied into the destination buffer `dest` and inserted at the position specified by `where`. If `source` or `dest` are illegal buffer handles, the `VOX_No-Such-Resource` error is returned. If `start` is beyond `source`'s total length, a `VOX_IllegalValue` error is generated. If `stop` is beyond `source`'s total length, its value is truncated to the length of `source`. Similarly, if `where` is beyond `dest`'s length, its value is truncated to the length of `dest`. In case of an error, the operation is a no-op. `VOX_Success` indicates successful completion of the operation.

To create an identical copy of an entire buffer, use `BUF_Clone()`:

```
VOX_Status BUF_Clone(vox, old, new)
VOX_Server vox;
BUF_Buffer old, *new;
```

The contents of the `old` buffer is copied into the newly created buffer `new`. If `old` is not a legal buffer handle, the `VOX_No-Such-Resource` error is returned. `VOX_Success` indicates successful completion of the operation.

For editing purposes, the client may need to determine periods of sound and silence in a given buffer:

```
typedef struct {
   VOX_Duration start;
   VOX_Duration length;
   Boolean isSilence;
} BUF_Interval;

VOX_Status BUF_FirstInterval(vox, buffer, interval)
VOX_Server vox;
BUF_Buffer buffer;
BUF_Interval *interval;

VOX_Status BUF_NextInterval(vox, buffer, interval);
VOX_Server vox;
BUF_Buffer buffer;
BUF_Interval *interval;
```

The `BUF_Interval` structure describes an interval of a buffer and indicates whether or not that interval is silence or voice. (See the `BUF_SetStatus()` operation for how to alter the threshold used in the silence detection.) A traversal of a buffer must start with the `BUF_FirstInterval()` after which the `BUF_NextInterval()` may be called repeatedly. In both calls, `buffer` must refer to a valid buffer handle, else the `VOX_No-Such-Resource` error is returned. If `buffer` is an empty buffer, the `VOX_EmptyBuffer` error is generated. If `BUF_FirstInterval()` is called twice in a row without intervening calls to `BUF_NextInterval()` a `VOX_Failure` error is returned (unless the buffer contains only one interval). Similarly, `BUF_NextInterval` returns `VOX_Failure` if no `BUF_FirstInterval()` call has not been performed. Both operations return `VOX_LastInterval` if the current interval is the last interval in the buffer. If the operations fail for some other reason, the `VOX_Failure` error is returned. Successful completions are indicated via `VOX_Success`.

Caveat: The current implementation supports only *one* traversal per buffer at any time.

A buffer's status is described by the `BUF_Status` data type:

```
typedef struct {
   Integer threshold;
   VOX_Duration length;
} BUF_Status;
```

The `threshold` field is used in determining periods of silence and sound in the `BUF_First-Interval()` and `BUF_Next-Interval()` calls. Its

exact interpretation depends on the device on which the sound data was recorded. (To possibly find out that information, store the buffer in a sound `BUF_Store` and get the header of that sound (`SND_HeaderOf()`).). The `length` field of the `BUF_Status` structure indicates the overall duration of the sound contained in a buffer.

To set or get a buffer's status, use:

```
VOX_Status BUF_GetStatus(vox, buffer, status)
VOX_Server vox;
BUF_Buffer buffer;
BUF_Status *status;

VOX_Status BUF_SetStatus(vox, buffer, newStatus, oldStatus)
VOX_Server vox;
BUF_Buffer buffer;
BUF_Status newStatus, *oldStatus;
```

The `BUF_GetStatus()` operation queries the buffer on its current status. `BUF_SetStatus()` sets the buffer's status and returns the previous value of its status. Note that for the status setting case, the `length` field of the `newStatus` argument is ignored. Both operations return a `VOX_No-Such-Resource` error if `buffer` is not a legal buffer handle. `VOX_Success` indicates successful completion of the operations.

5.10 The Player Data Type

The *Player* LAUD is the abstraction used to reproduce sound data, be it from a sound file or a sound buffer.

```
typedef struct {
   VOX_Duration duration;
   Integer volume;
} PL_Status;
```

The `PL_Status` structure describes the player's status: `duration` indicates the maximum length of any playback and `volume` is the device specific volume used during playback.

To get access to an instance of a player LAUD, the client uses the `PL_Open()` call:

```
VOX_Status PL_Open(vox, dev, status, player)
VOX_Server vox;
```

```
DEV_Name dev;
PL_Status status;
PL_Player *player;
```

The dev argument specifies which playback device is used when the LAUD becomes mapped. status indicates the LAUD's audio parameters used during playback. The call returns a handle to a player LAUD in player. The call can fail and return the following error status values:

VOX_No-Such-Resource: there is no device as named by dev on this instance of VOX.

VOX_IllegalValue: the status argument contains an illegal value for the named device.

VOX_EmptySequence: the device major name is an empty sequence.

VOX_Failure: the LAUD instance cannot be created by the server.

VOX_Success indicates successful completion of the operation.

Once a player LAUD has been successfully created, a client may submit playing requests:

```
VOX_Status PL_Prepare(vox, player, sound, from, length, enqueue, ev)
VOX_Server vox;
PL_Player player;
VOX_Handle sound;
VOX_Duration from, length;
Boolean enqueue;
EV_Event *ev;
```

The PL_Prepare() call creates a play request on the named player LAUD (thereby implicitly selecting the device on which the sound is played). from and length specify the interval of the given sound that is to be played. If from is beyond the end of sound, an error is returned, but if from length exceed the sound's total length, the length argument is adjusted to include the end of the sound file. Note that sound must be either a buffer or sound handle. The enqueue flag determines whether or not the event is immediately enqueued in the LAUD's output queue, or if the client defers the enqueueing operation until later (see LAUD_Put() and Q_Put()). ev is the handle for the newly created play request returned to the client.

The call may fail with the following error values:

VOX_IllegalValue: the duration specified in length is zero or the from value exceeds the length of the sound.

VOX_NewSound: the buffer or sound to be played is empty.

`VOX_EmptyBuffer`: the buffer or sound to be played is empty.

`VOX_No-Such-Resource`: either `player` or `sound` are invalid resource handles.

`VOX_Failure`: the given sound cannot be played on the given LAUD because of incompatible encoding algorithms.

`VOX_Success` indicates successful completion of the operation.

Caveat: If `sound` indicates a buffer handle, the client should not perform any editing operation between the play's preparation and its execution. Otherwise, unpredictable behavior may result.

Two kinds of events are associated with a player LAUD. The first is the actual `play` output request, the second is the `termination` event that is generated by the player LAUD upon completion of a play request[12].

```
typedef enum {
   PL_Play,
   PL_Termination
} PL_EvType;

typedef struct {
   VOX_Handle sound;     /* sound or buffer handle */
   VOX_Duration from;    /* start of play interval */
   VOX_Duration length;  /* length of play interval */
   LAUD_LAUD player;     /* handle to player LAUD */
} PL_PlayEv;

typedef struct {
   EV_Event request;       /* refers to play request which terminated */
   VOX_Handle sound;         /* sound or buffer handle */
   PL_Player player;       /* primitive LAUD on which sound was played */
   VOX_Termination reason;  /* why did play terminate? */
} PL_TerminationEv;

typedef struct {
   PL_EvType designator;
   union {
      PL_PlayEv u_PL_Play;
      PL_TerminationEv u_PL_Termination;
   } u;
} PL_Event;
```

[12]The termination event is considered to be an `input` event that must be `read` from the player LAUD by the client.

Note that the play events are automatically deallocated by the server once the play is terminated. Thus, the handle returned in the `request` field is no longer valid within the server, but can be used by the client to associate termination events with the original play requests.

A play event can terminate for the following reasons:

`VOX_TimeOut`: the length of the play exceeded the `duration` specified in the player's status description.

`VOX_AbortedEv`: the play event was explicitly aborted or the processing LAUD was unmapped or destroyed.

`VOX_EOS`: the play reached the end of the sound or the end of the interval indicated in the `PL_Prepare()` call.

To query or set a player LAUD's status, VOX provides the following calls:

```
VOX_Status PL_GetStatus(vox, player, status)
VOX_Server vox;
PL_Player player;
PL_Status *status;

VOX_Status PL_SetStatus(vox, player, status, oldStatus)
VOX_Server vox;
PL_Player player;
PL_Status status, *oldStatus;
```

The `PL_Get-Status()` call returns the player's status in the `status` argument. The `PL_Set-Status()` call takes a new `status` argument and returns the previous value of the player's status in `oldStatus`. If `player` is not a legal player handle, the calls return a `VOX_No-Such-Resource` error. The `PL_Set-Status` call returns a `VOX_Failure` error if the player LAUD is mapped at the time of the call. `VOX_Success` indicates successful completion of the operations.

5.11 The Recorder Data Type

A *recorder* is the LAUD that supports the recording of a sound file or a temporary buffer. The `REC_Status` structure describes the various state parameters of a recording LAUD:

```
typedef struct {
```

```
   SND_Encoding encoding;       /* encoding algorithm */
   Boolean pauseDetection;      /* on/off flag */
   VOX_Duration initialPause;   /* cf. below */
   VOX_Duration finalPause;     /* cf. below */
   VOX_Duration duration;       /* max duration of a recording */
   Integer sampleRate;          /* in Hz */
   SND_SilenceHandling silenceHandling;  /* type of silence handling */
   Integer threshold;          /* silence determination, device spec */
} REC_Status;
```

The following fields of the above structure need more explanation:

`initialPause`: this parameter controls the termination of a recording. If, after starting a recording, no sound is detected for a duration equal to or greater than `initialPause` the recording terminates. In this case, the termination event's `reason` field is set to `VOX_InitialPause` (see page 99).

`finalPause`: this field also controls the termination of a recording. If, during the recording, a lapse of silence with duration greater or equal to `finalPause` occurs, the recording is terminated. The termination event's `reason` field is set to `VOX_FinalPause`.

`duration`: another parameter to terminate a recording. If no pause detection is enabled or if no pauses (silence) occur during the recording, and the recording duration exceeds `duration` the recording is terminated. The termination event's `reason` field is set to `VOX_TimeOut` in this case.

`silenceHandling`: this field is used to control the treatment of silence during the recording. For a discussion, see section 5.8.

`threshold`: this field controls the determination of what energy levels are considered silence versus non-silence. Its value is device-dependent. (see also section 5.8)

Detection of initial and final pauses is enabled and disabled by the `pauseDetection` status flag.

Once a client has prepared a recorder's status, a recording LAUD can be created by the `REC_Open()` call:

```
VOX_Status REC_Open(vox, device, status, recorder)
VOX_Server vox;
DEV_Name device;
REC_Status status;
REC_Recorder *recorder;
```

The `device` argument names the server's device onto which the recording LAUD will eventually be mapped. `status` describes various device settings as described on page 96. The call returns the handle to a recording LAUD instance in `recorder`.

The call can fail and return with the following error status values:

`VOX_EmptySequence`: the device major name is an empty sequence.

`VOX_No-Such-Resource`: the VOX server instance has no device with the given device name `device`.

`VOX_IllegalValue`: one of the status field values is invalid for the named device.

`VOX_Failure`: the server could not create the recording LAUD instance.

`VOX_Success` indicates successful completion of the operation.

Similar to the player LAUD, clients also prepare a recording request that is queued by the recording LAUD and will be eventually processed on the server[13]. The `REC_Prepare()` call creates a recording event:

```
VOX_Status REC_Prepare(vox, recorder, sound, enqueue, ev)
VOX_Server vox;
REC_Recorder recorder;
VOX_Handle sound;
Boolean enqueue;
EV_Event *ev;
```

The `recorder` argument indicates the LAUD to be used for the recording (and thereby implicitly the recording device). `sound` is a handle to either a sound or buffer abstraction. If `sound` refers to an existing buffer or sound file, the recording overwrites the previously recorded data. `enqueue` indicates whether or not the event shall be enqueued for execution immediately or if the client will enqueue it at some later point in time (by a `LAUD_Put()` or `Q_Put()` call). The handle to the event created on the server is returned in the `ev` argument.

The call can fail with the following error values:

`VOX_No-Such-Resource`: the supplied `recorder` handle does not refer to a recording LAUD or the `sound` argument is not a handle to a buffer or a sound.

`VOX_Failure`: when recording into a buffer, the server could not prepare the buffer for recording. If the recording event cannot be enqueued (if `enqueue` is set), the call also returns with `VOX_Failure`.

[13]Unless, of course, the event is aborted or removed from its queue prior to execution.

VOX_Success indicates successful completion of the operation.

A recording LAUD deals with two kinds of events: Output requests to record a sound, input events to signal completion of a recording.

```
typedef enum {
   REC_Record,
   REC_Termination
} REC_EvType;

typedef struct {
   VOX_Handle sound;
   LAUD_LAUD recorder;
} REC_RecordEv;

typedef struct {
   EV_Event request;       /* refers to record request which terminated */
   VOX_Handle sound;         /* sound or buffer that was recorded */
  REC_Recorder recorder;  /* primitive LAUD on which sound was recorded */
   VOX_Termination reason;  /* why did recording terminate ? */
} REC_TerminationEv;

typedef struct {
   REC_EvType designator;
   union {
      REC_RecordEv u_REC_Record;
      REC_TerminationEv u_REC_Termination;
   } u;
} REC_Event;
```

The REC_RecordEv structure describes a recording request.

Completion of recording requests are signaled by the REC_TerminationEv structure which describes the reason for the event's completion. The following termination reasons can be returned in the reason field:

VOX_InitialPause: the recording never "started" since no sound was detected during the initialPause duration as specified in the recorder's status.

VOX_FinalPause: the recording terminated due to a final period of silence (or pause) equal or greater to the duration finalPause specified in the recorder's status.

VOX_TimeOut: the length of the recording exceeded the duration specified in the recorder's status description.

`VOX_AbortedEv`: the record event was explicitly aborted or the processing LAUD was unmapped or destroyed.

Note that although the server deallocates recording events after their completion, the client may use the `request` field in the `REC_Termination-Ev` to associate the termination event with the original recording request.

It is, of course, possible to query or set the status of a recording LAUD:

```
VOX_Status REC_GetStatus(vox, recorder, status)
VOX_Server vox;
REC_Recorder recorder;
REC_Status *status;

VOX_Status REC_SetStatus(vox, recorder, status, oldStatus)
VOX_Server vox;
REC_Recorder recorder;
REC_Status status, *oldStatus;
```

The `REC_SetStatus()` call takes a new `status` argument and returns the previous value of the recorder's status in `oldStatus`. If `recorder` is not a legal recorder handle, the calls return a `VOX_No-Such-Resource` error. The `REC_SetStatus()` call returns a `VOX_Failure` error if the recorder LAUD is mapped at the time of the call. `VOX_Success` indicates successful completion of the operations.

5.12 The Monitor Data Type

The *Monitor* is a specialized LAUD type. It can be used to implement a client that manages the instances of LAUDs currently resident on the server. There should only be one VOX client that uses the Monitor LAUD.

The Monitor's client obtains information on the operations performed on various LAUD instances by reading events from the input queue associated with that LAUD sub-type instance (see `LAUD_Get()`). The `MON_EvType` enumerates the types of notifications created by the server and `MON_Event` furthermore indicates which LAUD instance was involved in the operation.

```
typedef enum {
   MON_Creation,
   MON_Destruction,
   MON_Mapping,
```

```
   MON_Unmapping,
   MON_Locking,
   MON_Unlocking
} MON_EvType;

typedef struct {
   MON_EvType designator;
   union {
      LAUD_LAUD u_MON_Creation;
      LAUD_LAUD u_MON_Destruction;
      LAUD_LAUD u_MON_Mapping;
      LAUD_LAUD u_MON_Unmapping;
      LAUD_LAUD u_MON_Locking;
      LAUD_LAUD u_MON_Unlocking;
   } u;
} MON_Event;


VOX_Status MON_Open(vox, monitor)
VOX_Server vox;
MON_Monitor *monitor;
```

A client accesses the Monitor LAUD by opening it with the `MON_Open()` call. Note that only one client should read input events from the monitor, otherwise the interleaving of input events is arbitrary.

5.13 The Phone Data Type

5.14 The Fader Data Type

5.15 The VOX Library

This section describes a few commonly used calls that assist the VOX client programmer. All of these calls are provided by the *VL* module; hence the `VL` prefix in all of these operations[14].

The most important calls are the ones to start and terminate communication with the server:

```
VOX_Server VL_Open(host)
```

[14]Note that some of these calls are actually C macros.

```
char *host;

VL_Close(vox)
VOX_Server vox;
```

The `VL_Open()` call returns a communication handle to the VOX server on the named host. This handle must be used with all the RPC calls described in the previous sections. Failure of the `VL_Open()` call is indicated by a negative return value. To break the connection with a VOX server, use `VL_Close`.

The library header files declare a few useful macros to define data types used in the RPC protocol:

```
#define VL_PortName(str, pn) (VL_Str2Seq(str, &(pn), sizeof(pn.sequence)))
#define VL_SndName(str, sn) (VL_Str2Seq(str, &(sn), sizeof(sn.sequence)))
#define VL_SetDuration(s, ms, dur) {dur.sec = s; dur.muSec = ms;}
#define VL_NullSeq(s) ((s)->length = 0)  /* sets a sequence to empty */
```

The `VL_Str2Seq()` call used in some of the above macros is defined as:

```
VL_Str2Seq(str, seq, sz);
char *str;
VOX_String32 *seq;
int sz;
```

`str` is a standard C null terminated string. `seq` is the address of a `VOX_String`, whose length is given by `sz`. `sz` may only take the values 32, 64, and 128 depending on the value of the actual parameter `seq`.

To initialize a VOX device name data structures use:

```
VL_DevName(major, minor, name)
char *major;
int minor;
DEV_Name *name;
```

VOX introduces and uses the duration data type for which `VL` defines a few helping operations:

```
VL_MSecs2Dur(msecs, dur)
int msecs;
```

```
VOX_Duration *dur;

int VL_Dur2MSecs(dur)
VOX_Duration *dur;
```

These operations implement the conversion between time values expressed in milliseconds and `VOX_Duration`'s. Note that `VOX_Duration()` arguments are passed by reference in both calls.

To perform arithmetic with `VOX_Duration` values:

```
VL_AddDur(t1, t2, t3)   /* t1 = t2 + t3 */
VOX_Duration *t1, *t2, *t3;

VL_SubDur(t1, t2, t3)  /* t1 = t2 - t3 */
VOX_Duration *t1, *t2, *t3;

int VL_CmpDur(t1, t2)  /* t1 ? t2 */
VOX_Duration *t1, *t2;
```

Note that all arguments are passed by reference. In the `VL_AddDur()` and `VL_SubDur()` calls, `t1`, `t2`, or `t3` may point to the same memory location. The `VL_CmpDur()` call returns a value less than zero, equal to zero, or greater than zero if `t1 < t2`, `t1 = t2`, or `t1 > t2`.

```
VL_ErrMsg(str, status)
char *str;
VOX_Status status;
```

The `VL_ErrMsg` can be used in a spirit similar to UNIX's `perror()`: it prints the supplied `str` and the string name of the supplied `status` to `stdout`.

5.16 What Else do I Have Do Know?

This section tells you all the remaining things to start writing VOX applications.

Clients need to include the files `vox.h` and `vl.h`. The client program must then be linked against the VOX library. Currently, there is a library for regular UNIX clients and one for clients using a lightweight concurrency package.

5.16.1 Environment Variables

VOX uses the following UNIX environment variables:

`SND_DIR`: is the default sound directory for all LAUDs created by any client.

5.17 Examples

5.17.1 Recording a Sound

The first example shows how to record a sound file with the VOX server.

The `Record()` routine below assumes the following data declarations:

```
VOX_Server vox;          /* ipc channel to the VOX server */
VOX_Status s;            /* rpc call return status from VOX */

DEV_Name devName;        /* name of recording device */
REC_Recorder rec;        /* LAUD mapped to recording device */
REC_Status recStatus;    /* status of recording LAUD */

VOX_String64 sndName;    /* sound file string name */
SND_Sound snd;           /* sound file handle */
EV_Event ev;             /* recording request handle */
```

The code fragment to record the sound is:

```
Record(fname)
char *fname;
{

  /* set up device name & status */
  VL_DevName("vxr", 0, &devName);
  recStatus.duration.sec = 5; recStatus.duration.muSec = 0;
  recStatus.encoding = SND_ADPCM;
  recStatus.sampleRate = 8000;
  recStatus.pauseDetection = false;
  recStatus.silenceHandling = SND_None;
  recStatus.initialPause.sec = recStatus.initialPause.muSec = 0;
```

```
  recStatus.finalPause.sec =  recStatus.finalPause.muSec = 0;
  recStatus.threshold = 0;
  /* open record LAUD and assemble it */
 if (s = REC_Open(vox, devName, recStatus, &rec)) VL_ErrMsg("REC_Open", s);
  if (s = LAUD_Assemble(vox, rec)) VL_ErrMsg("LAUD_Assemble", s);
  /* open a sound */
  VL_SndName(fname, sndName);
  if (s = SND_Open(vox, rec, sndName, &snd)) VL_ErrMsg("SND_Open", s);
  /* map LAUD to devices */
  if (s = LAUD_Map(vox, rec, false, false)) VL_ErrMsg("LAUD_Map", s);
  /* create and enqueue a recording request */
 if (s = REC_Prepare(vox, rec, snd, true, &ev)) VL_ErrMsg("REC_Prepare", s);
  /* wait for completion of recording */
 if (s = EV_WaitForState(vox, ev, EV_Done)) VL_ErrMsg("EV_WaitForState", s);
  if (s = SND_Close(vox, snd)) VL_ErrMsg("SND_Close", s);
  if (s = LAUD_Destroy(vox, rec)) VL_ErrMsg("LAUD_Destroy", s);
}
```

The initial actions of the program are to create the recorder's state. In the example, the recorded sound is at most 5 seconds long (`duration`), is using ADPCM encoding at a sampling rate of 8000 Hz and no silence handling or pause detection is performed.

After the `REC_Open()` call comes the mandatory `LAUD_Assemble()` call. Note that it is needed even in the case of a single LAUD CLAUD hierarchy.

The routine then opens a sound file with the `SND_Open()` call, after which the recording LAUD is mapped to its device and a recording request is created. The client synchronizes with the termination of the recording using the `EV_WaitForState()` call, closes the sound file, and finally destroys the recording LAUD.

In the example, the recording will terminate after a sound of 5 seconds was recorded. (The client could use `LAUD_Get()` to read the termination event.)

5.17.2 Playing a Sound

The code to play a recorded sound is quite analogous to the recording of the sound. First, we need a few variable declarations:

```
VOX_Server vox;         /* ipc channel to the VOX server */
VOX_Status s;           /* rpc call return status from VOX */

DEV_Name devName;       /* name of recording device */
```

```
PL_Player pl;           /* LAUD mapped to playing device */
PL_Status plStatus;     /* status of playing LAUD */

VOX_String64 sndName;   /* sound file string name */
SND_Sound snd;          /* sound file handle */
EV_Event ev;            /* recording request handle */
```

The C code to play a sound file then is:

```
Play(fn)
char *fn;
{

  /* set up device name & status */
  VL_DevName("vxp", 0, &devName);
  plStatus.duration.sec = 10;  plStatus.duration.muSec = 0;
  plStatus.volume = 13; /* device dependent... */
 if (s = PL_Open(vox, devName, plStatus, &pl)) VL_ErrMsg(vox, "PL_Open", s);
  if (s = LAUD_Assemble(vox, pl)) VL_ErrMsg("LAUD_Assemble", s);
  /* open a sound */
  VL_SndName(fn, sndName);
  if (s = SND_Open(vox, pl, sndName, &snd)) VL_ErrMsg("SND_Open", s);
  /* create play request, enqueue it */
  VL_SetDuration(0, 0, from); VL_SetDuration(15, 0, length);
  if (PL_Prepare(vox, pl, snd, from, length, true, &ev))
    VL_ErrMsg("PL_Prepare", s);
  /* activate LAUD */
  if (s = LAUD_Map(vox, pl, false, false)) VL_ErrMsg("LAUD_Map", s);
  /* wait for completion */
 if (s = EV_WaitForState(vox, ev, EV_Done)) VL_ErrMsg("EV_WaitForState", s);
  if (s = SND_Close(vox, snd)) VL_ErrMsg("SND_Close", s);
  if (s = LAUD_Destroy(vox, pl)) VL_ErrMsg("LAUD_Destroy", s);
}
```

The playing LAUD's status is set to a maximum play length of 10 seconds and the volume is set to 13[15]. The program then creates the play LAUD and assembles it.

Before the play can be started, the client must open a sound file with the `SND_Open()` call, create a play request (`PL_Prepare()`) and map the LAUD to its device (`LAUD_Map()`). Note that the order of the `LAUD_Map()` call and the `PL_Prepare()` call could be inversed.

[15]Currently the volume setting is device specific; future releases of VOX may achieve further device independence by using a mapping between logical VOX volumes to physical device volumes.

The client synchronizes with the termination of the play request through use of the `EV_WaitForState()` call, after which it closes the played sound and destroys the playing LAUD.

5.17.3 Using Queues

This final example illustrates how the client can make usage of queues in a CLAUD hierarchy that incorporates a recording and a playing LAUD.

Again, we first assume the following variable declarations:

```
VOX_Server vox;          /* ipc channel to the VOX server */
VOX_Status s;            /* rpc call return status from VOX */

DEV_Name devName;        /* name of recording device */

PL_Player pl;            /* LAUD mapped to playing device */
PL_Status plStatus;      /* status of playing LAUD */

REC_Recorder rec;        /* LAUD mapped to recording device */
REC_Status recStatus;    /* status of recording LAUD */

LAUD_LAUD l;             /* a composite LAUD */

VOX_String64 sndName;    /* sound file string name */
SND_Sound snd1, snd2     /* sound file handles */
EV_Event ev1, ev2;       /* recording request handle */


PlayRec(prompt)
char *prompt;
{
  /* open a player laud */
  VL_DevName("vxp", 1, &devName);
  plStatus.duration.sec = 100;  plStatus.duration.muSec = 0;
  plStatus.volume = 10;
  if (s = PL_Open(vox, devName, plStatus, &pl)) VL_ErrMsg("PL_Open", s);
  /* open a record laud */
  VL_DevName("vxr", 1, &devName);
  recStatus.duration.sec = 30;  recStatus.duration.muSec = 0;
  recStatus.encoding = SND_ADPCM;
  recStatus.sampleRate = 8000;
  recStatus.pauseDetection = true;
```

```
  recStatus.initialPause.sec = 0; recStatus.initialPause.muSec = 500000;
  recStatus.finalPause.sec = 1; recStatus.finalPause.muSec = 0;
  recStatus.silenceHandling = SND_None;
  recStatus.threshold = 200;
 if (s = REC_Open(vox, devName, recStatus, &rec)) VL_ErrMsg("REC_Open", s);
  /* build a c-laud */
  if (s = LAUD_Create(vox, &l)) VL_ErrMsg("LAUD_Create", s);
  if (s = LAUD_Insert(vox, l, rec)) VL_ErrMsg("LAUD_Insert", s);
  if (s = LAUD_Insert(vox, l, pl)) VL_ErrMsg("LAUD_Insert", s);
  if (s = LAUD_Assemble(vox, l)) VL_ErrMsg("LAUD_Assemble", s);
  /* open two sounds */
  VL_SndName(prompt, sndName);
  if (s = SND_Open(vox, pl, sndName, &snd1)) {
    if (s != VOX_SoundExists) {VL_ErrMsg("SND_Open", s); exit();}
  }
  VL_SndName("sound2", sndName);
  if (s = SND_Open(vox, rec, sndName, &snd2)) {
    if (s != VOX_NewSound && s != VOX_SoundExists)  {
      VL_ErrMsg("SND_Open", s); exit();
    }
  }
  /* create a play & record event */
  VL_SetDuration(0, 0, from);  VL_SetDuration(100, 0, length);
  if (s = PL_Prepare(vox, pl, snd1, from, length, true, &ev1)) {
    VL_ErrMsg("PL_Prepare", s); exit();
  }
  if (s = REC_Prepare(vox, rec, snd2, true, &ev2)) {
    VL_ErrMsg("REC_Prepare", s); exit();
  }
  /* activate LAUD */
  if (s = LAUD_Map(vox, l, false, false)) VL_ErrMsg("LAUD_Map", s);
  /* wait for completion of recording event */
 if (s = EV_WaitForState(vox, ev2, EV_Done)) VL_ErrMsg("EV_WaitForState", s);
  if (s = SND_Close(vox, snd1)) VL_ErrMsg("SND_Close", s);
  if (s = SND_Close(vox, snd2)) VL_ErrMsg("SND_Close", s);
  if (s = LAUD_Destroy(vox, l)) VL_ErrMsg("LAUD_Destroy", s);
}
```

First, the program opens a recording and a playing LAUD and inserts them with the `LAUD_Insert()` calls into the CLAUD `l`. Then it opens a sound for playback and one to be recorded. Note that the recording will overwrite any existing sound file with the name `''sound2''`. The program then creates a play event that plays at most 100 seconds of sound `snd1` and a recording event. Both events are enqueued with the CLAUD `l`, which then is activated with the `LAUD_Map()` call. The client then awaits completion of the recording through the

`EV_WaitForState()` call, after which it cleans up by closing the sound files and destroying the CLAUD `l`.

5.18 Future Improvements and Miscellany

The names of all procedures, variables, etc. should be made consistent, such as by prepending the string "`VOX_`".

Note that some calls are applied to the top CLAUD in a hierarchy, while others apply to the specific LAUD referenced in the procedure call. (These should be explicitly mentioned in this document).

All status calls make use of a bitmask.

Limit of one `Buf_NextInterval()` at a time needs to be fixed.

5.19 Acknowledgments

Chris Schmandt wrote a first prototype of the editor software; the interface of section 5.9 are based on that work.

Chapter 6

Device Level Interface

Revision History:

Author(s)	Date	Rev.	Comments
Binding	Dec 88		Original Manual
Arons	Oct. 17 89	0.9	Reformatted, cleaned up, re-wrote.

6.1 Introduction

The VOX Audio Server is intended to be adaptable to a variety of speech and audio hardware. This document describes the interfaces between VOX and its hardware-independent *DEV* layer. All definitions referred to in this document can be found in the VOX server header file `dev.h`.

Caveat: This document is based on our experience with one (1) audio board and thus should not be considered representative. In particular, we do not address the issues of speech synthesis or speech recognition. We reserve ourselves the right to modify below described interfaces at our full discretion.

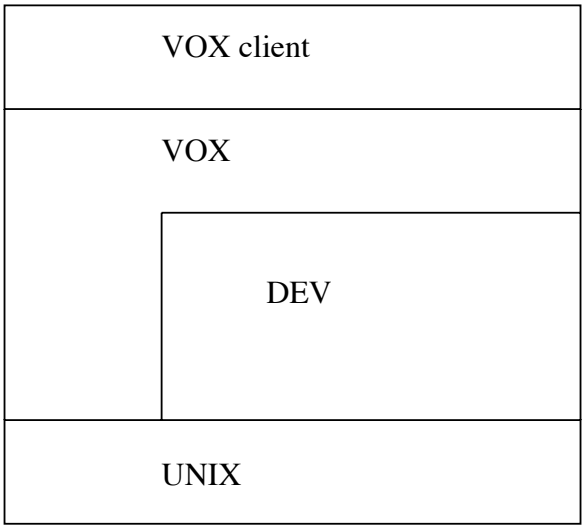


Figure 6.1: VOX audio server internals

The overall software organization of the VOX audio server is shown in Figure 6.1. It can be divided into the following layers:

The *client* or *application* layer: this is any client that uses the programmer interface described in [5].

The *VOX* layer: this is the layer of the software that implements most of the VOX functionality, i.e. LAUD management, sound file management, editing operations, queue handling, etc.

The *DEV* layer: internally, VOX interfaces with this layer to access devices in a hardware-independent fashion. The intention is to make this interface level general enough so that VOX can be adapted to various hardware platforms.

The operating system layer: currently this is a UNIX device driver interface to the actual audio devices. The interface is predefined by the standard UNIX system calls dealing with

devices, i.e. the `open()`,`read()`, `write()`, `ioctl()`, `close()` calls described in volume 2 of the UNIX documentation.

The remainder of this document mainly discusses the *DEV* layer interface.

6.2 Generalities

The organization of the *DEV* layer interface is similar to UNIX's device driver interface. Within VOX there is a table of procedure entries for each device that VOX handle. The name and calling conventions of these procedures are pre-defined. Each VOX device reference contains sufficient information to access these pre-defined device operations.

VOX names devices by their `major` and `minor` names, based on UNIX's naming convention for devices. In contrast to most conventional devices, audio devices may have *audio ports* associated with them. Internal to VOX this information is part of the *config* module (see also [10]) and is used to validate connections requested by the client. Similarly, the *config* module contains information on the exclusivity of use of various devices. For instance, it may not be possible to use recording device `dev1` at the same time as speech recognizer `dev2`, since both are implemented on the same hardware board.

The *DEV* layer interface must support the following routines:

`Map()`: this call opens the audio device and establishes the "correct" audio state.

`UnMap()`: the inverse of the `Map()` call. It resets the audio device and may close the associated UNIX device.

`Start()`: this call starts the actual operation of the device. It is called after the device has been successfully mapped. For instance, this call can start a recording or a playback. For event generating devices such as the telephone or a speech recognizer this call enables event generation by the device.

`Stop()`: this call stops operation of the device.

`Read()`: this call is used to read some data of the device. Thus, this call is only required with devices producing input (i.e. recording, telephone, speech recognition).

`Write()`: this call is used to write data to a device. Thus, this call is only necessary with devices producing output (i.e. playback, telephone, speech synthesis).

`Ioctl()`: this I/O control call is used for *most* everything in handling the devices. It can be mis-used for pure I/O functionality, but should mainly be used for device controlling operations.

All of the above functions take a reference to a device structure as its first argument:

```
typedef struct dev {
  ...
 CF_Device *dev;    /* device descriptor: ref into config device table */
  ...
  unsigned int status; /* device status mask */
  ...
  int fd;              /* UNIX file descriptor */
} DEV_DeviceStruct, *DEV_DeviceRef;
```

The `dev` field points into a jump table containing the device-specific routines described above. The `status` field is used to describe the current status of the device and can take following values:

```
#define DEV_Mapped
#define DEV_Locked
#define DEV_Running
```

and must be set appropriately by the device operations. Note that the `DEV_Locked` status flag is set by VOX and *not* the device operations. Another field of importance is the actual UNIX file descriptor `fd` that is used by the device. Note that several VOX devices may use *the same* UNIX device in the sense that they share the entry in directory `/dev`. It should be possible to have all of these devices mapped simultaneously, though obviously not all of them are operating at once. It is the responsibility of the `Start()` and `Stop()` calls to perform the appropriate context switching of the devices.

6.3 The Map Call

The syntax of the call is:

```
#define DEV_Map(d) (...)
DEV_DeviceRef d;
```

The `Map()` operation should open the UNIX device associated with the VOX device, set the `fd` field in the VOX device descriptor and may pre-establish some device state. If the device is only capable of one function, the `Map()` call should create all the necessary state to perform that function. In case the same UNIX device is acting as multiple VOX devices, it is the `Start()` operation that will establish the actual device mode.

Successful completion of the `Map()` call is indicated by a return value of 0; failure is indicated via a return value of 1.

6.4 The Unmap Call

The syntax of the call is:

```
#define DEV_Map(d) (...)
DEV_DeviceRef d;
```

The `UnMap()` operation may or may not close the actual UNIX device associated with the VOX device given via `d`. It must however set the `fd` field to -1 and return the audio device to a neutral state.

Successful completion of the `UnMap()` call is indicated by a return value of 0; failure is indicated via a return value of -1.

6.5 The Read Call

The `Read()` call is used to transfer data from the device:

```
#define DEV_Read(d, buf, sz) (...)
   DEV_DeviceRef d;
   char *buf;
   int sz;
```

`d` is of type `DEV_DeviceRef`, `buf` points to an array of bytes (chars), and `sz` indicates the size of the buffer.

The call returns the number of bytes actually read of the device. In case of failure, a -1 is returned. A return value of 0 indicates that the device no longer produces input, since it has terminated for some reason (see also page 121).

6.6 The Write Call

The `Write()` call is used to transfer data to the device:

```
#define DEV_Write(d, buf, sz) (...)
   DEV_DeviceRef d;
   char *buf;
   int sz;
```

`d` is the VOX device that receives the data, `buf` is an array of bytes (char) that contains the data, and `sz` indicates the byte count to be transferred.

The call returns the number of bytes actually written to the device. In case of failure, a -1 is returned. A return value of 0 indicates that the device no longer accepts output, since it has terminated for some reason (see also page 121).

6.7 The Ioctl Call

The backdoor in UNIX to control more sophisticated I/O devices is the `ioctl()` system call. Since VOX's device layer is modeled after (and ultimately mapped to) UNIX's device layer, the `Ioctl()` call takes a similar importance. Most of the device control is performed with this call.

The general syntax of the `Ioctl()` call is:

```
#define DEV_Ioctl(d, cmd, arg) (...)
   DEV_DeviceRef d;
   int cmd;
   long arg;
```

`d` is the VOX device reference as in all other device level calls. (From now on, `d` will refer to a such a reference, unless explicitly specified otherwise.) `cmd` specifies what control command is to be executed and `arg` is a 32 bit entity that can be used as an argument by the operation. Since `arg` can be coerced to a pointer, arbitrary amounts of information can be passed with this parameter.

The `Ioctl()` call returns 0 on success; -1 on failure.

We now examine the various possible values of `cmd` (all these values are defined in `dev.h`).

6.7.1 Creating and Breaking Connections

The VOX architecture addresses the routing of audio signals between different audio devices. At the device layer, connections are created through an `Ioctl()` call.

To describe a connection, the *DEV* layer introduces:

```
typedef struct {
    short inPort, outPort;
} DEV_Connection;
```

The `inPort` field is an index into the port table and thus identifies the port that *receives* the audio signal. The *outgoing* audio signal stems from the `outPort` of the connection. Note that a `DEV_Connection` structure fits into a 32 bit entity and thus can be passed directly to the `Ioctl()` call.

To create a connection, VOX first calls the device with the `outPort` using the `DEV_Connect` command argument, then it calls the device with the `inPort` using the `DEV_Accept` command argument.

```
DEV_DeviceRef outDev, inDev;
DEV_Connection connection;
...
DEV_Ioctl(outDev, DEV_Connect, connection);
DEV_Ioctl(inDev, DEV_Accept, connection);
...
```

It is up to the individual device handling software to properly establish the connection. Note that quite often the `DEV_Accept` call may be a no-op.

To break a connection, VOX also calls both devices involved in the connection:

```
DEV_DeviceRef outDev, inDev;
DEV_Connection connection;
...
DEV_Ioctl(outDev, DEV_Break, connection);
DEV_Ioctl(inDev, DEV_Refuse, connection);
...
```

Similarly to establishing the connection, VOX first calls the device "generating" the audio output with the `DEV_Break` command argument, then it calls the device receiving input with the `DEV_Refuse` command argument.

In both calls, the device layer can assume that the `connection` value is legal for the supplied device references, i.e. it has been validated with regards to direction of data flow, signal level, and physical connectivity between ports. This is done by VOX based on information in the *config* module (see also [10]).

6.7.2 Silence Handling and Pause Detection

Recording and playback devices may support various silence handling and pause detection policies. This section addresses these issues at the device level interface.

```
int flag = 0;

DEV_Ioctl(d, DEV_SetPauseDetection, flag);
DEV_Ioctl(d, DEV_GetPauseDetection, &flag);
```

The `DEV_SetPauseDetection` call enables or disables pause detection according to the value of the `flag` argument: 0 to disable, 1 to enable. `DEV_GetPauseDetection` queries the device for the current pause detection mode. If the device does not support pause detection, the `DEV_SetPauseDetection` call fails.

To set and query the values of the initial pause:

```
VOX_Duration duration;

DEV_Ioctl(d, DEV_SetInitialPause, &duration);
DEV_Ioctl(d, DEV_GetInitialPause, &duration);
```

Similarly, to set and query the values of the final pause:

```
VOX_Duration duration;

DEV_Ioctl(d, DEV_SetFinalPause, &duration);
DEV_Ioctl(d, DEV_GetFinalPause, &duration);
```

The `VOX_Duration` type is declared in [5]. If the device does not support pause detection, both calls return a 1.

To set or determine the threshold level used in determining what is silence and what is non-silence (voice):

```
int threshold;

DEV_Ioctl(d, DEV_SetThreshold, threshold);
DEV_Ioctl(d, DEV_GetThreshold, &threshold);
```

Currently the threshold value is interpreted differently by each device and it is up to the VOX application to match threshold values with the actually used hardware. If the device does not support silence handling, the call will return 1.

During recording or playback, audio devices may treat silence in different ways. (see [1, 5])

```
SND_SilenceHandling handling;

DEV_Ioctl(d, DEV_SetSilenceHandling, handling);
DEV_Ioctl(d, DEV_GetSilenceHandling, &handling);
```

The type `SND_SilenceHandling` is defined in [5]. It is up to the device to interpret these values properly. If the device does not support silence handling, the call will return -1.

6.7.3 Miscellaneous Device Control

To set/query the volume of a play device:

```
int volume;

DEV_Ioctl(d, DEV_SetVolume, vol);
DEV_Ioctl(d, DEV_GetVolume, &vol);
```

Note that the volume is hardware specific[1].

To set/query the sample rate of a recording or playback device:

```
int rate;    /* in Hz */

DEV_Ioctl(d, DEV_SetRate, rate);
DEV_Ioctl(d, DEV_GetRate, &rate);
```

Note that sample rates are absolute and given in Hz.

To limit the maximum duration of a recording or a playback:

```
VOX_Duration duration;

DEV_Ioctl(d, DEV_SetDuration, &duration);
DEV_Ioctl(d, DEV_GetDuration, &duration);
```

Different devices support different digital encoding algorithms:

[1] VOX could use a range of volumes and map that to each device, but different devices still may have non-linear mappings etc. The true problem here is that there is no absolute measure for a sound volume unless using a reference acoustic environment.

```
SND_Encoding coding;

DEV_Ioctl(d, DEV_SetEncoding, coding);
DEV_Ioctl(d, DEV_GetEncoding, &coding);
```

The `SND_Encoding` type is defined in [5]. We have not yet defined the exact encodings of the various, commonly used encoding schemes.

A device must be able to identify itself:

```
VOX_String64 make, model;

DEV_Ioctl(d, DEV_GetMake, &make);
DEV_Ioctl(d, DEV_GetModel, &model);
```

The `VOX_String64` data type is declared in [5]. Both calls return a device specific string name for the device's make or model.

All data transfers in the `Write()` and `Read()` calls necessarily use a buffer of a given size. Since the device will buffer data internally, it is optimal to have VOX use the same buffer size used by the device.

```
int packetSize, headerSize;

DEV_Ioctl(d, DEV_GetPacketSize, &packetSize);
DEV_Ioctl(d, DEV_GetHeaderSize, &headerSize);
```

Both sizes are in units of bytes. The header size of a packet may be 0.

To find out the sample size used by the currently active encoding algorithm:

```
int sampleSize;

DEV_Ioctl(d,  DEV_GetSampleSize, sampleSize);
```

The units here are bits per sample. If the concept of a sample size is not meaningful with the currently used encoding algorithm, the call returns -1.

As mentioned in sections 6.5 and 6.6, a device may terminate to produce or accept I/O for a variety of reasons. In the case that VOX detects I/O termination it queries the device with:

```
VOX_Termination termination;

DEV_Ioctl(d, DEV_GetTermination, &termination);
```

The `VOX_Termination` type is defined in [5].

Since VOX uses lightweight concurrency, it queries the device on the availability of I/O.

```
int avail;

DEV_Ioctl(d, DEV_GetIOAvail, &avail);
```

The value of `avail` is interpreted as follows:

> Recording device: the value is the number of available packets to be read off the device without having VOX block.
>
> Playback device: the value is the number of packets that VOX can write to the device without blocking.

A return value of 1 indicates termination of the device's operation; VOX may then use the `DEV_GetTermination` ioctl call.

Note: the `GetIOAvail` call can easily be implemented with the BSD UNIX `select()` or the System V.3 `poll()` call, *provided* the UNIX device drivers are correctly integrated with the UNIX kernel.

Most devices will buffer I/O in the device driver. Queries to a device on its buffer size can be made with:

```
int size;

DEV_Ioctl(d, DEV_GetQSize, &size);
```

The value of `size` is interpreted for recording and playback devices as the value is the maximum number of data packets that can be buffered by the device.

6.7.4 Support for Editing

The VOX interface supports the concept of sound editing. In order to implement editing, the device level interface has to support editing related operations.

First, VOX needs to ascertain whether or not some data represents silence or non-silence (voice).

```
DEV_Packet packet;

DEV_Ioctl(d, DEV_IsSilence, &packet);
```

The argument passed to the device layer is a reference to a `DEV_Packet` structure:

```
typedef struct {
  int sz, cnt;   /* sz: max size of data buffer, cnt: valid cnt of data */
  char *data;    /* data buffer */
  long spare;    /* to be misused however it suits... */
} DEV_Packet;
```

which describes an arbitrary piece of data. The `spare` field is "misused" to pass a silence/voice decision threshold to the device and to return a value of 1 if the packet is interpreted as silence, 0 if the packet contains voice. The data buffer used in the `packet` argument is managed (i.e. allocated and deallocated.) by VOX.

Some encoding algorithms have to maintain a *predictor* and thus it may not always be possible to playback voice starting arbitrarily. The `Smooth` call is meant to reset the predictor's value to an appropriate level to play some sound data.

```
DEV_Packet packet;

DEV_Ioctl(d, DEV_Smooth, &packet);
```

Upon procedure entry, the packet's `spare` field should contain the desired predictor level. Upon return, the packet contains a sequence of bytes that have to be written to the device to reset the algorithms to the desired predictor level. The data buffer in the `packet` argument is managed by the device layer.

If the device is self-resetting, the `cnt` field of the packet will be 0.

6.7.5 Parameter validation

Some of the device settings are hardware device specific and VOX probes the device on the validity of various device setting values.

```
int rate;                       /* in Hz */
int volume;
SND_Encoding coding;            /* see VOX programmer manual */
```

```
SND_SilenceHandling handling;
int threshold;
int sampleSize;                 /* in bits per sample */

DEV_Ioctl(d, DEV_CheckRate, rate);
DEV_Ioctl(d, DEV_CheckVolume, volume);
DEV_Ioctl(d, DEV_CheckEncoding, coding);
DEV_Ioctl(d, DEV_CheckSilenceHandling, handling);
DEV_Ioctl(d, DEV_CheckThreshold, threshold);
DEV_Ioctl(d, DEV_CheckSampleSize, sampleSize);
```

If the checked value is legal, the call returns 0; else a 1 is returned.

6.8 The Start Call

After a device has been mapped, it is ready for operation in its current mode. To start actual operation of the device VOX executes the `DEV_Start()` operation.

```
#define DEV_Start(d) (...)
DEV_DeviceRef d;
```

Successful completion of the operation is indicated via a return value of 0; failure by 1.

The semantics of the `Start()` call should be as follows:

> Playback device: device starts playing from its internals buffers. If buffers are empty, generate silence. Be prepared to accept a `Write()`.
>
> Recording device: device starts recording into internal buffers. If buffers overflow before VOX does a `Read()`, discard overflow.

6.9 The Stop Call

This is the inverse of the `Start()` call. The syntax is:

```
#define DEV_Stop(d, flush) (...)
    DEV_DeviceRef d;
    boolean flush;
```

The `flush` argument indicates whether or not to flush the device internal buffers. Successful completion of the operation is indicated via a return value of 0; failure by 1.

The semantics of the `Start()` call should be as follows:

> Playback device: device stops accepting output from VOX, i.e. a future `Write()` will return 0. If `flush == false`, the device continues playing any output buffered internally to the device. If `flush == true`, the device flushes its internal buffers and stops playing immediately.
>
> Recording device: device stops recording into internal buffers. If `flush == false`, a subsequent `Read()` may return buffered input. If `flush == true`, the internal buffers are purged and a subsequent `Read()` will return 0.

6.10 Miscellaneous

6.10.1 The UNIX Driver Interface

The bulk of this document deals with the VOX internal interface to the DEV layer. The DEV layer in turn is implemented using the raw UNIX system calls such as `read, write, open, close`, and `iotcl`.

Since VOX uses internal lightweight concurrency it assumes that devices can be configured to generate a UNIX signal upon availability of I/O, or use a polling `select()` call (under BSD derivatives of UNIX) or `poll()` call (under System V.3). We prefer the first alternative, i.e. the device generates a SIGIO signal under BSD derivatives or a SIGUSR signal under System V.3.

6.11 Extensions

This section proposes extensions to the current VOX-device interface specifications. In particular we address telephony related issues.

6.11.1 Additional Audio Device Control

On input devices it may be possible to set the `gain`.

```
int gain;
```

```
DEV_Ioctl(d, DEV_SetGain, gain);
DEV_Ioctl(d, DEV_GetGain, &gain);
```

The value of the `gain` argument is device specific.

Some input devices may support automatic gain control calibration.

```
VOX_Duration intvl;

DEV_Ioctl(d, DEV_Calibrate, &intvl);
```

The device calibrates itself for a duration of `intvl`. If `intvl` is set to `(0,0)`, automatic gain control is disabled.

6.11.2 Telephony

Note: this section is incomplete.

We split the telephony functionality into two distinct LAUDs: telephone control and the audio aspects of the telephone device. For the latter, the above I/O functionality should be sufficient. The VOX-client interface introduces a set of I/O termination values that are phone specific.

Overall mode of operation: phone is one LAUD; has two kinds of functionality though. I/O start, stop, read, write only work if phone control has been done a-priori. This must be done by client, i.e. it has to dial first, then start recording or playing over phone. The phone is in fact a sub-type of both play & record Laud (Multiple inheritance...)

To control the access to the telephone network, we need some more I/O control calls.

Dial a number:

```
typedef struct {
    VOX_String32 dialString;     /* the phone number */
    PH_CallParams parameters;    /* calling parameters */
    PH_Termination termination;  /* how did call terminate? */
} DEV_DialArgs;
DEV_DialArgs dialArgs;

DEV_Ioctl(d, DEV_Dial, &dialArgs);
```

Assume `PH_CallParams` and `PH_Termination` declared in VOX-client interface. The call is blocking with the result returned in `termination` field.

Get a DTMF string: can be interspersed with read/writes of phone device. can abort a record/play upon detection of DTMF. Phone status may include a DTMF mask to abort play/record on specific touch tones. (or DTMF sequence)

```
PH_DTMFSeq dtmf;

DEV_Ioctl(d, DEV_GetDTMF, &dtmf);
```

The `PH_DTMFSeq` data type is to be defined in VOX-client interface to allow for exchange of a set of touch tones. Call is non-blocking, may return empty sequence.

Detection of incoming phone call: UNIX devices should signal presence of I/O via SIGIO. Within VOX there is a dedicated thread to handle SIGIO occurrences. This thread calls a LAUD specific I/O routine (up-call) which in turn calls the device for the actual I/O.

Problems:

must distinguish between phone ring and audio I/O. state to be maintained by phone LAUD: waiting for ring or waiting for audio input.

UNIX device must generate SIGIO.

The only information that a ring-in event carries is the number of times the phone rang, which I think we want to ignore. Thus, only need to query:

```
boolean answer;
DEV_Ioctl(d, DEV_PhoneRung, &answer);
```

Non-blocking could be used for polling; destructive, i.e. if phone did ring only one `DEV_PhoneRung` should be done.

Set hook status control:

```
boolean onOff;
DEV_Ioctl(d, DEV_SetHook, onOff);
DEV_Ioctl(d, DEV_GetHook, &onOff);
```

Flashing hook (useful for some local exchanges?):

```
int dummy;
DEV_Ioctl(d, DEV_FlashHook, dummy);
```

Set rings to pick up phone before `DEV_PhoneRung` ioctl call returns `true`:

```
short nbrOfRings;
DEV_Ioctl(d, DEV_SetNbrOfRings, nbrOfRings);
DEV_Ioctl(d, DEV_GetNbrOfRings, nbrOfRings);
```

Chapter 7

MetaMake Facility

Revision History:

Author(s)	Date	Rev.	Comments
Chen	Jun. 23 89		Original
Arons	Dec. 19 89	0.9	Reformat, cleanup

7.1 Introduction

The `MetaMake` facility for VOX is a software suite which consists of `mkall` (1), `mkmf` (1), `imake` (1), `makedepend` (1) and related UNIX processors included in the final `Makefile`. This document explains the conventions of using this facility. For individual processors in question, please refer to their corresponding man pages or supplementary documentation available in the UNIX manual.

7.2 Configuration

`MetaMake` assumes that there is a base directory path for VOX bound to a UNIX environment variable called `VOXDIR`. Under this base directory, the directory `config` includes all the files needed for configuration setup.

Files in the `config` directory and their respective usage are listed below:

file	*usage*
`Imake.template`	template used by `imake` in generating `Makefile`
`Imakefile.local`	`Imakefile` for this `config` directory
`Imakefile.template`	template used by `mkall` in generating `Imakefile`
`Makefile`	the real thing
`env.VOX`	defines VOX specific environment
`macro.SUN`	defines macros for the SUN environment
`rcsinit`	shell script for initializing RCS
`rdistgo`	shell script for rdisting files between machines
`rule.VOX`	VOX specific rules
`system.SUN`	SUN specific system setup

Whenever a configuration file is changed in this directory, be sure to do:

```
make dependAll
```

in the VOX base directory. If the development environment is different from the backup environment (e.g., Mach on `thun` vs. SunOS on `ricerca`), the same system include files may reside in different places. To ensure that file dependencies are correct, the above command must be invoked on one's development environment instead of the backup environment.

7.3 Directory Setup

If a directory has no subdirectories of any significance to `make`, then its local `Imakefile` should have the variable `IHaveSubdirs` undefined, as in

```
#ifdef IHaveSubdirs
#undefine IHaveSubdirs
#endif
```

On the other hand, if a directory is non-terminal,

```
#ifdef IHaveSubdirs
#define IHaveSubdirs 1
#endif
```

is required. Also, the macro `SUBDIRS` must be bound to the list of subdirectories in `Imakefile` so that the same operation can be invoked recursively.

As a convention, an operation `foo` can be invoked by

```
make foo
```

which will have effect on the current directory only. On the other hand,

```
make fooAll
```

will apply `make foo` recursively to all subdirectories and finally to the current directory. The order the subdirectories are listed dictates the order they are made. This order is important if making something in one subdirectory depends on the result of something else being made first in another subdirectory. For a terminal directory, invoking `foo` or `fooAll` yields the same result.

7.4 Revision Control

Our revision control convention is as follows:

Everything should be checked in and then checked out unlocked (therefore read-only) by

```
make ciu
```

on the backup environment (e.g., `ricerca`) as the master copy. Files to be checked in should be bound to `RCSFILES` in the local `Imakefile`.

Files to be reconciled with the master copy must be handled with care. The following steps are recommended:

1. Give an advance notice to group members.
2. Check out and lock the files to be reconciled and conduct the merge manually.
3. Check these files back in and check them our unlocked.
4. Announce to the group when done.

7.5 Remote File Distribution

The master copy can be distributed to one's local environment for development. Files to be distributed should be bound to `RDISTFILES` in the local `Imakefile`. The shell script `rdistgo` in the `config` directory does the real work. To handle more environments than what are currently supported, the shell script might need to be modified.

Chapter 8

The IPC Module

Revision History:

Author(s)	Date	Rev.	Comments
Binding	88		Original
Arons	Oct. 17 89	0.9	Reformat, cleanup

8.1 Introduction

This document describes the IPC module supporting TCP/IP communication channels between a server and client UNIX process which may or may not be distributed accross a local area network. (Notice that much of the terminology is directly borrowed from the UNIX TCP/IP mechanism.)

The fundamental data type is called a *IPC_Channel*. Initially, the server creates a channel on which it *listens* for new clients. A client must *connect* to a (running) server using the *IPC_Connect* call. The server detects the new client and *accepts* the connection, thereby creating a new, fully dedicated channel for server–client communication.

```
typedef char IPC_Channel;  /* 0..31 */
```

An instance of an *IPC_Channel* is similar to a file descriptor in UNIX. Currently we are limited to at most 32 open channels.

Associated with each channel descriptor, the IPC module maintains an input and output buffer. When data is sent over the channel, it is first *put* into the output buffer, which is then *flushed* to transmit the data[1]. Input is *received* from the channel into the input buffer from where clients *get* this data.

8.2 Establishing Connections

When the server is started, it must call the *IPC_StartServer* entry.

```
IPC_StartServer(service, local, remote);
char *service;
IPC_Channel *local, *remote;
```

The *service* argument specifies the name of the service exported by the server. This name must be registered in advance in */etc/services*, due to UNIX naming conventions.

The call returns two *IPC_Channels* on which the server can accept new client connections. The *local* channel is used for connections from the same host as the server; the *remote* channel accepts connections from any other host on the network[2].

The client connects to the server with the *IPC_Connect* call.

[1] Flushing can occur either implicitly or explicitly.

[2] Under System V.3, local channels are not supported and *local* = -1 is returned.

```
IPC_Channel IPC_Connect(host, service)
char *host, *service;
```

The *host* argument identifies the network host on which the server runs. The *service* argument names the service as registered in */etc/services*. The call returns an IPC channel to be used by the client to communicate with the server. In case of a failure, the returned channel id is negative and not valid.

On the server side, a new client is accepted with the *IPC_Accept* call.

```
IPC_Channel IPC_Accept(ch)
IPC_Channel ch;
```

This call creates a new IPC channel that is dedicated to the communication between the server and a client. Recall that the server essentially sits in a loop, listening and accepting client's connection requests. After accepting a new connection, a dedicated connection is created between client and server. In case of a failure, the returned channel id is negative and not valid.

To close a communication channel, the interface exports the *IPC_Close* call.

```
IPC_Close(ch)
IPC_Channel ch;
```

After a channel is closed, it no longer can be used in sending or receiving data. In case of failure, the function returns a negative value; else 0 is returned.

8.3 Data Communication

Client and server communicate by exchanging data through the below function calls.

```
IPC_PutOne(ch, data, sz)
IPC_Channel ch;
char *data, sz;

IPC_PutArr(ch, data, cnt, sz)
IPC_Channel ch;
char *data, sz;
int cnt;
```

These calls send data over the named channel *ch*[3]. *data* may point to a *char*, a *short*, or an *int*, but the caller must indicate the size of a data item through the *sz* argument which is either 1, 2, or 4[4].

The *IPC_PutArr* call can be used to send *cnt* data items over an *IPC_Channel* at once.

Upon successful completion, these calls return the number of data items sent; otherwise a -1 is returned.

The interface also defines a set of macros that make the sending of data simpler.

```
#define IPC_PutB(ch, bp) (IPC_PutOne((ch), (bp), 1))
#define IPC_PutW(ch, wp) (IPC_PutOne((ch), (wp), 2))
#define IPC_PutL(ch, lp) (IPC_PutOne((ch), (lp), 4))

#define IPC_PutArrB(ch, data, cnt) (IPC_PutArr(ch, data, cnt, 1))
#define IPC_PutArrW(ch, data, cnt) (IPC_PutArr(ch, data, cnt, 2))
#define IPC_PutArrL(ch, data, cnt) (IPC_PutArr(ch, data, cnt, 4))
#define IPC_PutStr(ch, str) (IPC_PutArr(ch, str, strlen(str) + 1, 1))
```

These macros are self-explanatory. Only point to remember, *all* arguments are passed by reference!

To receive data from a channel, the interface exports the following calls:

```
IPC_GetOne(ch, data, sz)
IPC_Channel ch;
char *data, sz;

IPC_GetArr(ch, data, cnt, sz)
IPC_Channel ch;
char *data, sz;
int cnt;
```

data indicates the memory address into which data is received. *sz* indicates the data item size, i.e. 1 for *char,* 2 for *short* and 4 for *int*. *cnt* indicates how many items are to be received with the *IPC_GetArr* call. Notice that the call only returns after exactly receiving the specified number of data items.

Successful completion is indicated by the (positive) count of received items; failure is indicated by a return value of -1.

Again, the interface exports a set of macros to make these calls a bit more pleasant:

[3]The actual data transfer over the network occurs when the output buffer is flushed.
[4]These are of course the sizes of the data items in bytes.

```
#define IPC_GetB(ch, bp) (IPC_GetOne(ch, bp, 1))
#define IPC_GetW(ch, wp) (IPC_GetOne(ch, wp, 2))
#define IPC_GetL(ch, lp) (IPC_GetOne(ch, lp, 4))

#define IPC_GetArrB(ch, data, cnt) (IPC_GetArr(ch, data, cnt, 1))
#define IPC_GetArrW(ch, data, cnt) (IPC_GetArr(ch, data, cnt, 2))
#define IPC_GetArrL(ch, data, cnt) (IPC_GetArr(ch, data, cnt, 4))
```

A few additional calls are of interest here:

```
IPC_IOAvail(ch, in)
IPC_Channel ch;
boolean in;
```

The caller can query how much input is available from the channel's input buffer, respectively how much data can be written to the channel's buffer before it is flushed. The call returns the available buffer size in bytes. In case of failure, a negative value is returned.

```
IPC_Flush(ch, flags)
IPC_Channel ch;
int flags;
```

Output on channels is buffered. To force the sending of data, use the *IPC_Flush* call. (Notice, however, that flushes also occur implicitly while sending data over a channel when the output buffer is full.) The *flags* argument is for compatibility reasons with the underlying UNIX system calls and can be safely ignored here (i.e. set to 0). If successful, the call returns 0, else a negative value is returned.

```
IPC_Select(rMsk, wMsk, timeOut)
unsigned int *rMsk, *wMsk;
struct timeval *timeOut; /* cf. <sys/time.h> */
```

When data becomes available on a channel, can be sent on a channel or a new connection is to be accepted, the *IPC_Select* call determines what channels are "ready". The semantics are identical to the Berkeley UNIX 4.2 *select* (2) call[5].

Notice that *IPC_Select* call returns **all** UNIX file descriptors with possible I/O in *rMsk* and *wMsk*, not just file descriptors corresponding to *IPC_Channel*s. In case of success the call returns the number of valid file descriptors in the masks, otherwise the call returns a negative value.

[5]The other arguments in the system call are 0.

Once it is known which channels have (network) input available, use *IPC_RecvAny* to read data off the network:

```
IPC_RecvAny(msk, flags)
unsigned int *msk;
int flags;
```

The call fills the input buffers for all the channels specified in *msk*[6]. (It is from these buffers that routines like *IPC_Get*... read their data.) Success of the call is indicated via a return value of 0; a negative return value returns a negative value.

The mask is modified to indicate which channel descriptors were read.

In general, an *IPC_Select* call should be performed first to decide which channels have input available. Otherwise the *IPC_RecvAny* blocks the calling UNIX process.

The *flags* argument is for compatibility reasons with the underlying UNIX system calls and can be safely ignored here (i.e. set to 0).

8.4 Installation

Under the current version of SUN's UNIX incarnation, local TCP/IP sockets are created in the file system. The IPC module puts these "sockets" into the directory */tmp/.ipc* with names based on the service name the server implements. (This currently limits us to only one instance of every server running at once.)

To use the IPC module, the directory */tmp/.ipc* should thus be created on your machine. The directory should have all privileges associated with it (i.e. *chmod 777 /tmp/.ipc*)

8.5 Examples

It's now time for two examples. First, we present a client–server configuration in a traditional UNIX programming style. Second, we present the same situation using lightweight concurrency.

In both examples, we connect to the "test" service on a given host.

[6]Here, only the file descriptors corresponding to channels are taken into consideration.

8.5.1 Client–Server UNIX Style

First, the code for the client. After creating a connection, it receives some data from the server, and terminates by closing the connection.

```
/* client code for example of ipc module */
/* from UNIX */
#include <stdio.h>
#include <signal.h>

/* from binding */
#include <types/types.h>
#include "ipc.h"

static PrintLgs(lgs, nbr)
int lgs[], nbr;
{ short i;

  for (i = 0; i < nbr; i++) {
    fprintf(stderr, "%d  ", lgs[i]);
    if (i % 5 == 4) fprintf(stderr, "\n");
  }
}

main(argc, argv)
int argc;
char *argv[];
{ IPC_Channel s;
  byte b;
  char msg[64];
  int cnt;
  long lgs[64];
  char host[64];

  if (argc < 2) {
    if (gethostname(host, 64) < 0) {
      perror("gethostname"); exit();
    }
  } else strcpy(host, argv[1]);
  s = IPC_Connect(host, "test");
  cnt = IPC_GetB(s, &b);
  cnt = IPC_GetArrB(s, msg, 16);
  cnt = IPC_GetArrL(s, lgs, 12);
```

```
  PrintLgs(lgs, cnt);
  IPC_Close(s);
}
```

The server is a bit more complex. We must deal with the rather awkward channel creation à la UNIX TCP/IP sockets.

```
/* test server for ipc module */
/* from UNIX */
#include <signal.h>
#include <sys/time.h>
#include <stdio.h>

/* from binding */
#include <types/types.h>
#include "ipc.h"

static InitLgs(lgs, nbr)
long lgs[];
short nbr;
{ short i;

  for (i = 0; i < nbr; i++) lgs[i] = random();
}

static PrintLgs(lgs, nbr)
int lgs[], nbr;
{ short i;

  for (i = 0; i < nbr; i++) {
    fprintf(stderr, "%d  ", lgs[i]);
    if (i % 5 == 4) fprintf(stderr, "\n");
  }
}

main()
{ IPC_Channel local, remote, ch;
  int cnt, msk;
  char *msg = "this is a test\n";
  char c;
  long lgs[12];

  InitLgs(lgs, 12);
```

```
  if (IPC_StartServer("test", &local, &remote) < 0) exit();
  fprintf(stderr, "started server\n");
  while (true) {
    cnt = IPC_Select(&msk, 0); /* blocking */
    if (msk & (1 << local)) ch = IPC_Accept(local);
    if (msk & (1 << remote)) ch = IPC_Accept(remote);
    c = 'a';
    IPC_PutB(ch, &c); IPC_Flush(ch, 0);
    IPC_PutArrB(ch, msg, strlen(msg)+1); /* the null byte ! */
    PrintLgs(lgs, 12);
    IPC_PutArrL(ch, lgs, 12);
    IPC_Flush(ch, 0);
    IPC_Close(ch);
  }
  IPC_Close(local); IPC_Close(remote);
}
```

Notice the infinite loop. As soon as a client requests a new connection, the *IPC_Select* call returns and a new channel is created via the *IPC_Accept* call. The server then sends some data to the client (just a stupid example...), closes the connection, and loops around to accept the next client connection.

8.5.2 Client–Server Using Lightweight Concurrency

The IPC module can also be compiled for use in a lightweight concurrency environment[7].

The main difference to standard UNIX style, is the introduction of a signal handling thread. In fact, when data becomes available on a channel, a SIGIO signal is raised and the handling thread then decides which channels are to be read and reads these. The SIGIO handling thread *SIGIO-Handler* behaves like a high-priority interrupt handling thread receiving data off the network as soon as it becomes available, fills the buffers associated with individual channels, and releases input consuming threads blocked on a *IPC_Get*... call.

Again, first the client code:

```
/* example client code for ipc module */
/* from UNIX */
#include <stdio.h>
#include <signal.h>
#include <sys/time.h>
```

[7] In fact, the server can be using lightweight concurrency without the client being aware of that and vice versa.

```
/* from binding */
#include <types/types.h>
#include "ipc.h"

static struct timeval noTime;  /* null time val */

static SIGIOHandler()
/* waits for SIGIO and receives any possible channel data */
{ int cnt, msk;

  noTime.tv_sec = noTime.tv_usec = 0;
  while(1) {
    AwaitInterrupt(SIGIO);
    cnt = IPC_Select(&msk, &noTime); /* non-blocking */
    if (cnt > 0) IPC_RecvAny(&msk, 0);
  }
}

static PrintLgs(lgs, nbr)
int lgs[], nbr;
{ short i;

  for (i = 0; i < nbr; i++) {
    fprintf(stderr, "%d  ", lgs[i]);
    if (i % 5 == 4) fprintf(stderr, "\n");
  }
}

main(argc, argv)
int argc;
char *argv[];
{ IPC_Channel s;
  byte b;
  char msg[64];
  int cnt;
  long lgs[64];
  char host[64];

  if (argc < 2) {
    if (gethostname(host, 64) < 0) {
      perror("gethostname"); exit();
    }
  } else strcpy(host, argv[1]);
  EnableInterrupt(SIGIO);
```

```
  Fork(SIGIOHandler, 8000, 0);
  s = IPC_Connect(host, "test");
  cnt = IPC_GetB(s, &b);
  cnt = IPC_GetArrB(s, msg, 16);
  cnt = IPC_GetArrL(s, lgs, 12);
  PrintLgs(lgs, cnt);
  IPC_Close(s);
}
```

With the exception of the creation of the *SIGIOHandler* thread, the code is analogous to the UNIX style client.

And now the server code:

```
/* test server for ipc module */
/* from UNIX */
#include <signal.h>
#include <sys/time.h>
#include <stdio.h>

/* from binding */
#include <types/types.h>
#include "ipc.h"

static InitLgs(lgs, nbr)
long lgs[];
short nbr;
{ short i;

  for (i = 0; i < nbr; i++) lgs[i] = random();
}

static PrintLgs(lgs, nbr)
int lgs[], nbr;
{ short i;

  for (i = 0; i < nbr; i++) {
    fprintf(stderr, "%d  ", lgs[i]);
    if (i % 5 == 4) fprintf(stderr, "\n");
  }
}

static struct timeval noTime;  /* null time val */
```

```
static Server(ch)
int ch;
{ long lgs[12];
  char *msg = "this is a test\n";
  char c;

  InitLgs(lgs, 12);
  del.tv_sec = 3; del.tv_usec = 0;
  c = 'a';
  IPC_PutB(ch, &c); IPC_Flush(ch, 0);
  IPC_PutArrB(ch, msg, strlen(msg) + 1); /* the null byte ! */
  PrintLgs(lgs, 12);
  IPC_PutArrL(ch, lgs, 12);
  IPC_Flush(ch, 0);
  IPC_Close(ch);
}

main()
{ IPC_Channel local, remote, ch;
  int cnt, msk;

  noTime.tv_sec = noTime.tv_usec = 0;
  EnableInterrupt(SIGIO);
  if (IPC_StartServer("test", &local, &remote) < 0) exit();
  /* this is the SIGIO handler here ... */
  while (true) {
    AwaitInterrupt(SIGIO);
    msk = 0;  /* reset mask */
    cnt = IPC_Select(&msk, &noTime); /* non-blocking */
    if (cnt == 0) continue;
    if (msk & (1 << local)) {
      ch = IPC_Accept(local);
      Fork(Server,  8000, 1, ch);
    }
    if (msk & (1 << remote)) {
      ch = IPC_Accept(remote);
      Fork(Server, 8000, 1, ch);
    }
    IPC_RecvAny(&msk, 0); /* see if any input on our other channels */
  }
  IPC_Close(local); IPC_Close(remote);
}
```

Notice how easy it has become to create a server thread! Once a new client connection is detected in *main()*; a new *Server* thread is forked and it does the actual work. Thus, multiple client–server connections can be supported at once.

Chapter 9

Mercury Remote Execution Language

Revision History:

Author(s)	Date	Rev.	Comments
Binding	88		Original
Arons	Oct. 17 89	0.9	Reformat, cleanup

9.1 Introduction

This document describes the *Mercury*[1] specification language for procedure and message based inter-process communications layered on top of some communication layer such as TCP/IP. In the remainder of this document we assume familiarity with the general concept of remote procedure call and Berkeley UNIX's interprocess mechanisms [15, 8].

Mercury is based on E. Cooper's implementation of the *Courier* remote procedure call protocol for Berkeley UNIX bsd4.2 [9, 7]. The language introduces a syntax based on Modula-2 and more suitable for programming in C. Thus, *Mercury* does not support multiple return values of a procedure, nor does it allow for error conditions or for non-integer constants.

Other current deficiencies of *Mercury* include[2]:

> Floating point data is not supported. This, because different machines have different floating point representations and for now it was not judged necessary to go the extra mile and support IEEE standard floating point representations.
>
> Strings are not supported. This, because the receiver would have to allocate storage space to receive the string data and it would be difficult to ensure that the allocated data is freed. Garbage collection would obviously be the best fix here.
>
> References (pointers) are not supported. Beside the memory management problem similar to the strings case, we would need more sophisticated generation of marshalling routines for recursive data structures.
>
> Efficiency was not the main goal. Both, *Mercury* and the underlying IPC module [4] strive for robust code rather than ultra-fast execution. Our initial measurements yielded 8 ms for a local null remote procedure call and 9 ms for the remote case (on SUN 3/60 with 8 MBytes of memory).
>
> No Verification is performed. Indeed, there is no support for user authorization, nor is there a check on matching versions between server and client.

As an extension to *Courier*, *Mercury* procedures can declare *variable* arguments and *messages*. The latter are different from procedures in that they do not take any variable arguments and do not return a result. Messages are asynchronous in nature. That is, the client sends a message to the server and *does not* wait for any acknowledgment or result from the server.

Based on the *Mercury* interface specification, the *Mercury* compiler produces:

> A *header* file declaring the data types and constants defined in the *Mercury* interface.

[1] Named after the roman god of messages, Mercury.
[2] Honesty in advertisement: we tell the truth up-front!

A *stub* file containing marshalling routines for all the data types declared in the interface.

A *server* file containing the *SRV_Loop* routine which executes the server routines when (remotely) activated by the client and the stubs necessary to execute these routines.

A *client* file containing the call stubs for procedures or messages declared by the interface.

Mercury does not make any strong assumption on the underlying transmission protocol. All data types are translated into a sequence of bytes, shorts, and longs (8, 16, and 32 bit entities) that are transmitted over a communication channel. It thus becomes straightforward to adapt *Mercury* to different communication protocols implementing such channels. The IPC interface [4] is one possible communication mechanism which is layered on top of UNIX's TCP/IP protocol. Currently it is the only one supported.

The remainder of this document describes *Mercury* declarations and the C-code generated from these. Section 9.3 explains how to declare various data types; section 9.4 and 9.5 describe the remote procedure and message mechanism. The server loop which executes procedures and messages is described in section 9.6 Finally, section 9.7 gives a complete example of a *Mercury* program. Appendix 9.10 lists *Mercury*'s grammar in *yacc* style.

9.2 Constants

Mercury only supports constants of an integer basetype. The *Mercury* constant declaration

```
CONST Count = 10;
```

is translated into the C definition

```
#define Count 10
```

Mercury also supports octal and hexadecimal constants using C's scanning convention.

9.3 Data Types

Mercury has the following built-in data types:

BOOLEAN: a binary value, translated into a C *char* (8 bits).

CHAR: a signed eight-bit entity, translated into a C *char* (8 bits).

SHORT: a signed sixteen-bit entity, translated into a C *short* (16 bits).

INTEGER: a signed thirty-two bit entity, translated into a C *int* (32 bits).

UNSPECIFIED: an eight-bit entity, translated into a C *char* (8 bits).

The above built-in data types can be used to construct more sophisticated data types. For each of these data types, the *Mercury* header file *mercury.h* defines the marshalling and unmarshalling routines.

9.3.1 Arrays

Arrays declare a fixed number of data items. For instance, the *Mercury* array definition of

```
TYPE arr: ARRAY [Count] OF CHAR;
```

produces the C declaration of

```
typedef char arr[Count];
```

Notice the possibility of using a constant declaration to determine the array's size.

9.3.2 Sequence

Sequences are very similar to arrays; they can be seen as partially filled arrays. The sender and receiver of sequences only send or receive the valid portion of the sequence.

```
TYPE seq: SEQUENCE [10] OF CHAR;
```

is translated to

```
typedef struct {
        int length;
        char sequence[10];
} seq;
```

where the *length* field indicates how much of the data array is actually occupied. During sending and receiving sequences only *length* data items are transmitted.

9.3.3 Enumeration

An enumeration type is mapped into a C enumeration:

```
TYPE enum: (blue, red, green, yellow);
```

becomes

```
typedef enum {
        blue = 0,
        red = 1,
        green = 2,
        yellow = 3
} enum;
```

9.3.4 Records

Mercury records are translated into C structures.

```
TYPE rec: RECORD
           a: SHORT;
           b: INTEGER;
          END;
```

maps onto

```
typedef struct {
        short a;
        int b;
} rec;
```

9.3.5 Union

Union are tagged variant records. The tag field *must* be an enumeration type.

```
TYPE uni: UNION enum OF
            blue, yellow: INTEGER;
            red: SHORT;
            green: SHORT;
          END;
```

is translated into the C union:

```
typedef struct {
        enum designator;
        union {
                int u_blue;
#define blue_case u.u_blue
                int u_yellow;
#define yellow_case u.u_yellow
                short u_red;
#define red_case u.u_red
                short u_green;
#define green_case u.u_green
        } u;
} uni;
```

Type *enum* has been declared above. Notice the fashion in which *Mercury* accesses the various fields of the union. (This is used in the marshalling and unmarshalling routines.)

9.4 Procedures

Procedures implement remote procedure calls. They can have value and/or variable arguments and may return a result.

For example,

```
PROCEDURE p (i: enum; s: SHORT; VAR j: INTEGER): INTEGER;
```

This procedure declaration is used to generate a client and a server stub. The server stub is calling the actual routine as follows:

```
SRV_p(_ch_)
        int _ch_;
{
        enum i;
        short s;
        int j;
        int _rslt_;
```

```
        if (Unpackenum(_ch_, &i) < 0) goto bad;
        if (UnpackShort(_ch_, &s) < 0) goto bad;
        if (UnpackInteger(_ch_, &j) < 0) goto bad;
        _rslt_ = p(i, s, &j);
        if (PackInteger(_ch_, &j) < 0) goto bad;
        if (PackInteger(_ch_, &_rslt_) < 0) goto bad;
        if (Flush(_ch_) < 0) goto bad;
        return(1);
   bad: return(-1);
}
```

SRV_p is called from the server's main loop *SRV_Loop* generated by *Mercury* (see section 9.6) and calls the routine *p* exactly as a C programmer would expect.

On the client side, *p* takes **one additional** argument: the communication channel handle must be indicated.

```
int p(_ch_, i, s, j)
        int _ch_;
        enum i;
        short s;
        int *j;
{
        int _rslt_;
        int _opCode_ = 0;

        if (PackInteger(_ch_, &_opCode_) < 0) goto bad;
        if (Packenum(_ch_, &i) < 0) goto bad;
        if (PackShort(_ch_, &s) < 0) goto bad;
        if (PackInteger(_ch_, j) < 0) goto bad;
        if (Flush(_ch_) < 0) goto bad;
        if (UnpackInteger(_ch_, j) < 0) goto bad;
        if (UnpackInteger(_ch_, &_rslt_) < 0) goto bad;
        return(_rslt_);
   bad: abort();
}
```

Before sending the routine's arguments, an *opCode* is sent to the server. It is used in the server's loop to branch to the correct routine. Also, notice, the rather brutal failure handling: if a communication problem with the server arises, the client is aborted[3].

[3]Since procedures can return values of arbitrary types, we can't just return a -1 here!

The *opCode*s are generated by numbering procedures and messages in order of their declaration in the *Mercury* interface.

9.5 Messages

Unlike procedures, messages do not return a value from the server end, nor does the client wait until the server completes execution of the message.

```
MESSAGE m (i: INTEGER; s: SHORT);
```

yields the server function:

```
SRV_m(_ch_)
        int _ch_;
{
        int i;
        short s;

        if (UnpackInteger(_ch_, &i) < 0) goto bad;
        if (UnpackShort(_ch_, &s) < 0) goto bad;
        m(i, s);
        return(1);
   bad: return(-1);
}
```

Clearly, the server loop awaits completion of *m*, but **not** the client as evidenced by the client stub:

```
void m(_ch_, i, s)
        int _ch_;
        int i;
        short s;
{
        int _opCode_ = 1;

        if (PackInteger(_ch_, &_opCode_) < 0) goto bad;
        if (PackInteger(_ch_, &i) < 0) goto bad;
        if (PackShort(_ch_, &s) < 0) goto bad;
        if (Flush(_ch_) < 0) goto bad;
        return;
   bad: abort();
}
```

Again, as in the case of procedures, the client must supply **one additional** argument, namely the communication channel handle.

9.6 The Server Loop

Based on the procedures and messages declared in a *Mercury* interface, the compiler generates a server loop. It receives the operation codes sent by the client and branches to execute the appropriate routine. For our example above, the server loop takes the following form:

```
SRV_Loop(_ch_)
        int _ch_;
{
        int _opCode_;

        while (1) {
                if (UnpackInteger(_ch_, &_opCode_) < 0) goto bad;
                switch (_opCode_) {
                case 1:
                        if (SRV_m(_ch_) < 0) goto bad;
                        break;
                case 0:
                        if (SRV_p(_ch_) < 0) goto bad;
                        break;
                default:
                        fprintf(stderr,
                                "SRV_Loop: No such procedure or message\
                                   (%d)\n", _opCode_);
                        break;
                }
        }
    bad: Close(_ch_); return(-1);
}
```

9.7 Example

This section fully describes a real *Mercury* program. First the interface declaration:

```
INTERFACE Passwd;
```

```
CONST StrSize = 48;

TYPE String : ARRAY [StrSize] OF CHAR;

TYPE Password : RECORD
                  Name, Passwd: String;
                  UID, GID, Quota: INTEGER;
                  Comment, Gecos, Directory, Shell: String;
                END;

PROCEDURE Lookup(user: String): Password;
PROCEDURE LookupUID(uid: INTEGER): Password;

END Passwd.
```

Let's assume that this is declared in file *passwd.m,* using *m* for *Mercury* file name extension.

After successful invocation of *Mercury,* four new files are generated. The names of these files are the *Mercury* interface file-name with the extensions *h, _stb.c, _srv.c,* and *_clnt.c.*

The *header* file*, passwd.h* contains the C equivalents of the *Mercury* interface declaration:

```
/*
 * Declarations for mercury interface Passwd (Nbr 0, Vers 0).
 */
#include <ipc.h>
#include <mercury.h>

#define StrSize 48

typedef Char String[StrSize];

typedef struct {
        String Name;
        String Passwd;
        Integer UID;
        Integer GID;
        Integer Quota;
        String Comment;
        String Gecos;
        String Directory;
        String Shell;
} Password;
```

```
extern Password Lookup();

extern Password LookupUID();
```

Notice that it includes two other header files:

ipc.h: declares the interface to the TCP/IP communication module that is used in transmitting arguments to procedures and messages.

mercury.h: is a header file that declares aliases for the built-in *Mercury* data types and their corresponding marshalling and unmarshalling routines. For instance, it defines *Integer* as an alias of *int*, *Short* of *short*, etc.

The stub file is named *passwd_stb.c* and contains the following C code:

```
/*
 * Marshalling routines for mercury interface Passwd (Nbr 0, Vers 0).
 */
#include "passwd.h"

static Pack0(ch, p)
        Integer ch;
        Char *p;

{
        register Integer i;

        for (i = 0; i < StrSize; i++)
                if (PackChar(ch, &p[i]) < 0) goto bad;
        return(1);
   bad: return(-1);
}

#define PackString Pack0
#define UnpackString Unpack1

static Pack2(ch, p)
        Integer ch;
        Password *p;
{
        if (PackString(ch, p->Name) < 0) goto bad;
        if (PackString(ch, p->Passwd) < 0) goto bad;
```

```
        if (PackInteger(ch, &p->UID) < 0) goto bad;
        if (PackInteger(ch, &p->GID) < 0) goto bad;
        if (PackInteger(ch, &p->Quota) < 0) goto bad;
        if (PackString(ch, p->Comment) < 0) goto bad;
        if (PackString(ch, p->Gecos) < 0) goto bad;
        if (PackString(ch, p->Directory) < 0) goto bad;
        if (PackString(ch, p->Shell) < 0) goto bad;
        return(1);
   bad: return(-1);
}

#define PackPassword Pack2
#define UnpackPassword Unpack3

static Unpack1(ch, p)
        Integer ch;
        Char *p;
{
        register Integer i;

        for (i = 0; i < StrSize; i++)
                if (UnpackChar(ch, &p[i]) < 0) goto bad;
        return(1);
   bad: return(-1);
}

static Unpack3(ch, p)
        Integer ch;
Password *p;
{
        if (UnpackString(ch, p->Name) < 0) goto bad;
        if (UnpackString(ch, p->Passwd) < 0) goto bad;
        if (UnpackInteger(ch, &p->UID) < 0) goto bad;
        if (UnpackInteger(ch, &p->GID) < 0) goto bad;
        if (UnpackInteger(ch, &p->Quota) < 0) goto bad;
        if (UnpackString(ch, p->Comment) < 0) goto bad;
        if (UnpackString(ch, p->Gecos) < 0) goto bad;
        if (UnpackString(ch, p->Directory) < 0) goto bad;
        if (UnpackString(ch, p->Shell) < 0) goto bad;
        return(1);
   bad: return(-1);
}
```

The *stub* file contains the marshalling and unmarshalling routines for the declared data types. In case of failure, they return a -1, otherwise a value of 1 is returned. Their names are defined by a macro as the concatenation of their function (*Pack* or *Unpack*) with the type name.

The client stub file is named *passwd_clnt.c* and its C code is:

```
/*
 * Client stub for mercury interface Passwd (Nbr 0, Vers 0).
 */
#include "passwd_stb.c"

Password Lookup(_ch_, user)
        Integer _ch_;
        String user;
{
        Password _rslt_;
        Integer _opCode_ = 0;

        if (PackInteger(_ch_, &_opCode_) < 0) goto bad;
        if (PackString(_ch_, user) < 0) goto bad;
        if (Flush(_ch_) < 0) goto bad;
        if (UnpackPassword(_ch_, &_rslt_) < 0) goto bad;
        return(_rslt_);
   bad: abort();
}

Password LookupUID(_ch_, uid)
        Integer _ch_;
        Integer uid;
{
        Password _rslt_;
        Integer _opCode_ = 1;

        if (PackInteger(_ch_, &_opCode_) < 0) goto bad;
        if (PackInteger(_ch_, &uid) < 0) goto bad;
        if (Flush(_ch_) < 0) goto bad;
        if (UnpackPassword(_ch_, &_rslt_) < 0) goto bad;
        return(_rslt_);
   bad: abort();
}
```

It declares the routines that the client can call. Notice that the *first* argument must be a channel handle. Otherwise a client calls the routine by supplying value and variable arguments in order of their declaration in the *Mercury* interface.

The last file to be generated by *Mercury* is the server stub which also contains the server loop routine *SRV_Loop*:

```
/*
 * Server stub for mercury interface Passwd (Nbr 0, Vers 0).
 */
#include <stdio.h>
#include "passwd_stb.c"

extern Password Lookup();

SRV_Lookup(_ch_)
        IPC_Channel _ch_;
{
        String user;
        Password _rslt_;

        if (UnpackString(_ch_, user) < 0) goto bad;
        _rslt_ = Lookup(user);
        if (PackPassword(_ch_, &_rslt_) < 0) goto bad;
        if (Flush(_ch_) < 0) goto bad;
        return(1);
   bad: return(-1);
}

extern Password LookupUID();

SRV_LookupUID(_ch_)
        IPC_Channel _ch_;
{
        Integer uid;
        Password _rslt_;

        if (UnpackInteger(_ch_, &uid) < 0) goto bad;
        _rslt_ = LookupUID(uid);
        if (PackPassword(_ch_, &_rslt_) < 0) goto bad;
        if (Flush(_ch_) < 0) goto bad;
        return(1);
   bad: return(-1);
}

SRV_Loop(_ch_)
        Integer _ch_;
{
```

```
        Integer _opCode_;

        while (1) {
                if (UnpackInteger(_ch_, &_opCode_) < 0) goto bad;
                switch (_opCode_) {
                case 1:
                        if (SRV_LookupUID(_ch_) < 0) goto bad;
                        break;
                case 0:
                        if (SRV_Lookup(_ch_) < 0) goto bad;
                        break;
                default:
                        fprintf(stderr,
                      "SRV_Loop: No such procedure or message (%d)\n", _opCode_);
                        break;
                }
        }
    bad: Close(_ch_); return(-1);
}
```

The server file declares the routines to be called from the *SRV_Loop* routine. The latter receives the opcode that is unique for a given procedure or message of the interface and calls the routine after unpacking the arguments. In case of a procedure, variable arguments and a result are then packed and returned to the client.

Notice that both the client and server *include* the stub file *passwd_stb.c,* because of the static declaration of the marshalling routines.

What remains is to write the server and client's main routines. On the server side, we have to initialize the communication layer and wait for incoming client requests. Using the IPC module [4] and a lightweight concurrency package, the server's main routine declared in a file *pw_srv.c* is as follows:

```
/* server for mercury passwd program */
/* CB, ORC, Jul 88 */

#include <stdio.h>
#include <pwd.h>
#include <signal.h>
#include <strings.h>

#include <types/types.h>
#include <kernel/kernel.h>
#include "passwd.h"
```

```
/* these are the server's entries */
Password Lookup(user)
String user;
{ struct passwd *pw;
  Password p;

  pw = getpwnam(user);
  p.UID = pw->pw_uid; p.GID = pw->pw_gid; p.Quota = pw->pw_quota;
  /* strings must be explicitly copied */
  strncpy(p.Name, pw->pw_name, StrSize);
  strncpy(p.Passwd, pw->pw_passwd, StrSize);
  strncpy(p.Comment, pw->pw_comment, StrSize);
  strncpy(p.Gecos, pw->pw_gecos, StrSize);
  strncpy(p.Directory, pw->pw_dir, StrSize);
  strncpy(p.Shell, pw->pw_shell, StrSize);
  return(p);
}

Password LookupUID(uid)
int uid;
{ struct passwd *pw;
  Password p;

  pw = getpwuid(uid);
  p.UID = pw->pw_uid; p.GID = pw->pw_gid; p.Quota = pw->pw_quota;
  strncpy(p.Name, pw->pw_name, StrSize);
  strncpy(p.Passwd, pw->pw_passwd, StrSize);
  strncpy(p.Comment, pw->pw_comment, StrSize);
  strncpy(p.Gecos, pw->pw_gecos, StrSize);
  strncpy(p.Directory, pw->pw_dir, StrSize);
  strncpy(p.Shell, pw->pw_shell, StrSize);
  return(p);
}

extern SRV_Loop();

main()
{ IPC_Channel local, remote, ch;
  unsigned int msk;
  int cnt;

  EnableInterrupt(SIGIO);
  if (IPC_StartServer("test", &local, &remote) < 0) {
```

```
    fprintf(stderr, "pw: couldn't start server\n"); exit();
  }
  fprintf(stderr, "pw: started server\n");
  while(1) {
    AwaitInterrupt(SIGIO);
    fprintf(stderr, "pw: got SIGIO\n");
    msk = 0; /* use only ipc channels !! */
    cnt = IPC_Select(&msk, 0);
    if (cnt < 0)  continue;
    if (msk & (1 << local)) {
      ch = IPC_Accept(local);
      Fork(SRV_Loop, 8000, 1, ch);
      msk &= ~(1 << local);
    }
    if (msk & (1 << remote)) {
      ch = IPC_Accept(remote);
      Fork(SRV_Loop, 8000, 1, ch);
      msk &= ~(1 << remote);
    }
    /* read all other ipc channels */
    if (msk != 0) IPC_RecvAny(msk, 0);
  }
}
```

The client code, file *pw_clnt.c,* is also using the IPC module and lightweight concurrency:

```
/* client code for mercury passwd program */
/* CB, ORC, Jul 88 */

#include <stdio.h>
#include <sys/time.h>
#include <signal.h>

#include <types/types.h>
#include <kernel/kernel.h>
#include "passwd.h"

static SIGIOHandler()
/* waits for SIGIO and receives any possible socket data */
{ int cnt, msk;
  struct timeval noTime;

  noTime.tv_sec = noTime.tv_usec = 0;
```

```
  while(1) {
    fprintf(stderr, "client: awaiting interrupt\n");
    AwaitInterrupt(SIGIO);
    fprintf(stderr, "client: got SIGIO\n");
    cnt = IPC_Select(&msk, &noTime);
    if (cnt > 0) IPC_RecvAny(msk, 0);
  }
}

main(argc, argv)
int argc;
char *argv[];
{ IPC_Channel ch;
  char host[64];
  Password pw;
  String un;

  for (ch = 0; ch < 16; ch++) un[ch] = 0;
  if (argc < 2) {
    if (gethostname(host, 64) < 0) {
      perror("gethostname"); exit();
    }
  } else strcpy(host, argv[1]);
  fprintf(stderr, "server on %s\n", host);
  EnableInterrupt(SIGIO);
  Fork(SIGIOHandler, 8000, 0);
  ch = IPC_Connect(host, "test");
  strncpy(un, "binding", 16);
  pw = Lookup(ch, un);
  IPC_Close(ch);
}
```

9.8 How to Use Mercury

The best thing to build a *Mercury* client and server is to rely on *make*. Here is the excerpt of the makefile to build the password program:

```
# makefile for passwd program
CFLAGS= -g -I.. -I../ipc  -I../mercury
OBJS= ../ipc/ipc.o ../set/set.o ../mem/mem.o ../kernel/kernel.o \
        ../kernel/coroutine.o
```

```
.SUFFIXES: .o .c .h .m

.m.h:
        mercury _*.m
        cc $(CFLAGS) -c $*_srv.c
        cc $(CFLAGS) -c $*_clnt.c

all: pw_srv pw_clnt

pw_srv: passwd.h pw_srv.c
      cc $(CFLAGS) -o pw_srv pw_srv.c passwd_stb.o passwd_srv.o  $(OBJS)

pw_clnt: passwd.h pw_clnt.c
     cc $(CFLAGS) -o pw_clnt pw_clnt.c passwd_stb.o passwd_clnt.o $(OBJS)

passwd.h: passwd.m
```

Notice the dependency of *passwd.h* on *passwd.m* and how *Mercury* not only generates C stubs, but also compiles these.

9.9 Acknowledgement

David Chase got me across my lex/yacc frustrations.

9.10 Mercury Grammar

Mercury has a pretty straightforward grammar, given here in *yacc* format.

```
%token
        identifier        number

%token
        ARRAY             BOOLEAN         CHAR            CONST
        END               INTEGER         INTERFACE
        MESSAGE           OF              PROCEDURE       PROGID
        RECORD            SEQUENCE        SHORT           TYPE
        UNION             UNSPECIFIED     VAR             VERSION

%%
```

```
Interface :
        InterfaceHeader InterfaceBody
        ;

InterfaceHeader :
                INTERFACE identifier PROGID number VERSION number ';'
        |       INTERFACE identifier ';'
        ;

InterfaceBody :
                DeclarationList END '.'
        |       DeclarationList END identifier '.'
        ;

DeclarationList :
                /* empty */
        |       DeclarationList Declaration
        ;

Declaration :
                TYPE identifier ':' Type ';'
        |       CONST identifier '=' NumericValue ';'
        |       PROCEDURE identifier ArgumentList Result ';'
        |       MESSAGE identifier ArgumentList ';'
        ;

Type :
                PredefinedType
        |       ConstructedType
        |       identifier
        ;


PredefinedType :
                BOOLEAN
        |       SHORT
        |       INTEGER
        |       CHAR
        |       UNSPECIFIED
        ;

ConstructedType :
                SEQUENCE '[' NumericValue ']' OF Type
        |        ARRAY '[' NumericValue ']' OF Type
        |       '(' ValueList ')'
        |       RECORD FieldList ';' END
        |       UNION identifier OF CandidateList ';' END
        ;
```

```
NumericValue :
                number
        |       identifier        /* numeric constant only */
        ;

ValueList :
                identifier
        |       ValueList ',' identifier
        ;

CandidateList :
                Candidate
        |       CandidateList ';' Candidate
        ;

Candidate :
                DesignatorList ':' Type
        ;

DesignatorList :
                identifier
        |       DesignatorList ',' identifier
        ;

ArgumentList :
                /* empty */
        |       '(' FieldList ')'
        ;

Result :
                /* empty */
        |       ':' Type
        ;

FieldList :
                Field
        |       FieldList ';' Field
        ;

Field :
                NameList ':' Type
        |       VAR NameList ':' Type
        ;

NameList :
                identifier
        |       NameList ',' identifier
```

```
        ;

%%
```

Lastly some words on scanning:

> Comments follow Modula-2 rules, i.e. they are enclosed between "(*" and "*)". Nesting is not allowed.

Chapter 10

Lightweight Concurrency Module

Revision History:

Author(s)	Date	Rev.	Comments
Binding	88		Original
Arons	Dec. 19 89	0.9	Reformat, cleanup

10.1 Introduction

This document describes a set of lightweight concurrency features for UNIX 4.3 BSD, SunOS 3.2, or System V.3. It describes the client interface to the lightweight kernel, a semaphore package, and the I/O package.

10.2 Concurrency support

The module `Kernel` exports the necessary primitives for lightweight process support, plus support for dealing with UNIX signals.

The kernel supports *non-preemptive* and *priority-based* scheduling. Context switches only occur at specific kernel calls and hence are *not* asynchronous. To achieve satisfactory concurrency, the client programmer should insert explicit context switches in his code. (see also `Tick`)

Interprocess synchronization is through the use of condition variables and their associated `Signal` and `Wait` operations. The kernel allows to bind UNIX software signals to *interrupts*. Each enabled interrupt is mapped onto the corresponding UNIX signal and upon the occurrence of a signal, the interrupt handling process, that was waiting for an interrupt, is resumed.

```
typedef *char Condition; /* condition variables */
typedef long Process;      /* process identification */
typedef unsigned int Priority;  /* 0 is lowest priority */

Process Fork(proc, stackSize, nbrOfArgs, arg1, arg2, arg3, arg4, arg5)
void (*proc) ();
int stackSize, nbrOfArgs, arg1, arg2, arg3, arg4, arg5;
/* fork the procedure 'proc' as a process, using a stack of 'stackSize' bytes */
/* and pass the supplied 'nbrOfArgs' arguments to 'proc'. The process */
/* identification of the child process is returned to the parent process. */

Join(pid)
Process pid;
/* suspend caller until the identified process is terminated */

TerminateProcess()
/* terminate the calling process. Note: This must only be done with the */
/* 'main' routine of the program, all other processes which have been forked */
/* terminate properly upon 'return' */

#define InitCondition(c) ((c) = NULL)
```

```
/* initialize condition variable */

#define Awaited(c) ((c)) != NULL)
/* is there a process waiting on condition 'c' ? */

Signal(c)
Condition *c;
/* signal the first process queued on 'c'. If 'c' is not empty, the signalee */
/* is resumed and the signaler is inserted into the ready queue and will be  */
/* resumed at some later context switch */

Release(c)
Condition *c;
/* release the first process queued on 'c' and put it into the ready queue */
/* but do NOT block the calling process. The released process is resumed at */
/* some later context switch */

Wait(c)
Condition *c;
/* the calling process is queued on condition variable 'c' and the */
/* next ready process is run */

ContextSwitch()
/* execute a context switch to the next ready process and resume it */

extern int _tickCount;
#define _TimeSlice 200
#define Tick ((_tickCount >= _TimeSlice) ? ContextSwitch() : _tickCount++)
/* force a more or less random context switch. Must be inserted by programmer */
/* into long, long, long loops */

Priority GetPriority(pid)
Process pid;
/* return the process' priority. If 'pid' == 0, the priority of the calling */
/* process is returned */

SetPriority(p, pid)
Priority p;
Process pid;
/* set the process' priority to 'p'. If 'pid' == 0 use the calling process. */
/* If the priority of the running process is lowered below the priority */
/* of the next runnable process, perform a context switch. Note: condition */
/* variable queues are not reordered by a SetPriority call */
```

```
Process GetPid()
/* returns the caller's process identification */

EnableInterrupt(sigNr)
int sigNr; /* 1 <= sigNr <= 32 && sigNr != SIG_ALRM */
/* enables the reception of interrupts which are bound in a 1-1 */
/* correspondence to UNIX signals */

AwaitInterrupt(sigNr)
int sigNr; /* 1 <= sigNr <= 32 && sigNr != SIG_ALRM */
/* block the calling process until kernel receives the UNIX signal */
/* that maps onto the specified interrupt. Upon reception of the */
/* signal, the blocked process is inserted into the ready queue and */
/* run at some later context switch */

DiscardInterrupt(sigNr)
int sigNr; /* 1 <= sigNr <= 32 && sigNr != SIG_ALRM */
/* discard possibly buffered UNIX signals which are mapped onto */
/* this interrupt */

DisableInterrupt(sigNr)
int sigNr; /* 1 <= sigNr <= 32 && sigNr != SIG_ALRM */
/* disable reception of UNIX signals that correspond to the specified */
/* interrupt */

boolean Delay(duration)
struct timeval duration; /* see UNIX manual (2): getitimer */
/* suspend the calling process for at least duration time and */
/* resume first ready process. If delay terminates normally, return */
/* true, if a delayed process is precipitated, returns false. */

Precipitate(pid)
Process pid;
/* if the named process has been delayed, force it to resume and abort Delay. */

Abort(msg)
char *msg;
/* print message onto file 'stderr' and abort program, forcing a core dump */
```

10.3 Semaphores

The `Lock` module allows the dynamic creation and use of binary semaphores for synchronization purposes between concurrent processes. Such semaphores can be created reentrant.

```
typedef char *LCK_Lock;

LCK_Lock LCK_New(reentrant)
boolean reentrant;
/* creates and returns a binary semaphore. If 'reentrant' the lock is
reentrant. */

LCK_Destroy(l)
LCK_Lock l;
/* destroys the binary semaphore 'l' */

LCK_Acquire(l)
LCK_Lock l;
/* the 'passeren' (P) operation on 'l' */

LCK_Release(l)
LCK_Lock l;
/* the 'vrijgeven' (V) operation on 'l' */
```

10.4 Input - Output

The `IO` package is intended to handle asynch I/O (this feature is not available on system V.3). The key to it is the kernel's dealing with UNIX *signals* plus the newly released feature of UNIX 4.3 to raise SIGIO when I/O becomes available on a file descriptor.

When a lightweight process performs an I/O request it either succeeds right away or may become blocked until I/O is possible. In the latter case, another lightweight process – if existing – is run. This is brand new and might have glitches and/or performance problems.

```
int IO_Open(path, flags, mode)
char *path;
int flags, mode;
/* UNIX 4.3 open (2) call. enables asynchronous I/O on file descriptor */

#define IO_Close(d) (close(d))
/* UNIX 4.3 close (2) call */
```

```
int IO_Read(d, buf, nbytes)
int d;
char *buf;
int nbytes;
/* UNIX 4.3 read (2) call. blocks process until I/O available */

int IO_Write(d, buf, nbytes)
int d;
char *buf;
int nbytes;
/* UNIX 4.3 write (2) call. blocks process until I/O possible */

IO_Cancel(d)
int d;
/* abort I/O request for an open file descriptor */
```

Chapter 11

Code Style Guidelines

Revision History:

Author(s)	Date	Rev.	Comments
	Jan 31 90	0.9	Created.

11.1 Introduction

In an effort to maintain consistency across multiple files, directories, and authors, we are attempting to standardize on set of coding conventions. Many of these conventions are based on the X Window System, but we have deviated where we felt the X conventions were inadequate. Note that we are heading toward adhering to these guidelines in the sample implementation, but have not yet executed all the necessary changes. See also [12] for a more detailed description of recommended practices, we agree in spirit with [12] and primarily deviate with regard to naming conventions.

11.2 Modules

A program consists of a set of modules. A "module" consists of a definition file ("`.h`" include file) and an implementation file ("`.c`" code file). The definition file and implementation file have the same "base" name, such as in `sample.h` and `sample.c`.

The definition file defines all constants, enums, and macros that are exported. It also declares as externs all variables and procedures that are exported by the interface. The include file is for definitions only, it produces no code in a `.o` file. The standard ordered format of a definition file is as follows:

1. RCS information
2. Header
3. ifdef/def begin (must always be present)
4. Constants, enums, and typedefs that are externed by the implementation file of the module.
5. Macros
6. Procedure declarations that are externed by the implementation file of the module.
7. ifdef/def end (must always be present)

Implementation files have the following format:

1. RCS information
2. Header
3. Includes

4. Constant, enum, and typedef declarations (local, non-exported only)
5. Variable declarations (non-exported and declared static only)
6. Macros (non-exported)
7. Procedure predeclarations (non-exported only and declared static)
8. Procedures

11.3 Naming Conventions

Note the use of names that end in `_ptr`, `_struct`, etc.

procedures `]SampleProcedure()`

1. First letter is capitalized.
2. Each subsequent word is capitalized.
3. Procedures always have a return value (never use the "int" default).

variables `sampleInteger`

1. First letter is lower case.
2. Each subsequent word is capitalized.

constants `SAMPLE_CONSTANT`

1. All upper case.
2. Each subsequent word is separated by an underscore ("_").

macros `SAMPLE_MACRO()` or `SAMPLE_MACRO2(a, b)`

1. All upper case.
2. Each subsequent word is separated by an underscore ("_").
3. Defined as though it were a function, even if it doesn't have any parameters.

structs `typedef struct FooBar_struct *FooBar_ptr;`

```
typedef struct FooBar_struct {
  int           a;
  FooBar_Ptr    fooBarPtr;
} FooBar;
```

1. Structure name starts with a capital letter.
2. Structure tag is the same as structure name, with "`_struct`" appended.
3. If the the struct is self-referential, a typedef for a pointer to the struct should be predeclared and immediately precede the typedef for the structure.
4. All structs are typedef'ed.

enums
```
typedef  enum {
  RED,
  YELLOW,
  BLUE
} MyColor;
```

1. Enum name starts with a capital letter.
2. Elements are all upper case, as they are used like `#define`'d constants.
3. All enums are typedef'ed.

11.4 Statements

All statements should be compound, i.e. an "if" statement *always* has braces:

```
if (a == b) {
  foo;
}
```

is permitted, but:

```
if (a == b)
  foo;
```

is not.

11.5 Comments

When a comment is used to describe a procedure, the comment should proceed the procedure.

11.6 Indenting

This probably doesn't matter too much, because of beautification programs. However, the following is recommended:

1. Tabs should be used for all indentation.
2. Procedures should be declared with the return value indented and the procedure name left justified on the next line.
3. Parameters and local variables be indented.

This allows one to easily find the definition of a function (with "grep" for example, by searching for the beginning of a line followed by the function name. For example:

```
/* Comments about function here. */
  char *
myProcedure(arg1, arg2)
  int    arg1;
  char   arg2;
{
  int    local1;
  char   local2[128];
 .
 .
 .
}
```

11.7 VOX Specific Guidelines

Names of all objects (procedures, typedef's, etc.) that are exported by the VOX–client interface should start with the prefix "`VOX_`". Conversely, any object not exported by the VOX interface (i.e. names of objects defined by a client or names of objects used by VOX but are not exported) should not begin with the `VOX_` prefix. Naming conventions described above prevail for the rest of the names, for example, variables and procedures exported by VOX would have names of the form:

```
VOX_sampleInteger
VOX_SampleProcedure()
```

Similarly if an enum is exported by VOX, elements in the enumeration should also be preceded by `VOX`:

```
typedef enum {
  VOX_RED,
  VOX_YELLOW,
  VOX_BLUE
} VOX_Color;
```

For objects exported by VOX, we use an "object oriented name space" with the most significant to least significant name from left-to-right. The manipulation on the object is the last part of the name. For example, a routine that gets the status of a buffer in VOX might be named:

```
VOX_BufferStatusGet()
```

Chapter 12

Requirements for a General Purpose Audio Board

Revision History:

Author(s)	Date	Rev.	Comments
Arons, Schmandt, Binding	Feb 3 1989	1.1	Minor revisions

12.1 Introduction

This document details the audio input and output specifications for a general purpose workstation-based audio board controlled by the VOX Audio Server. The specifications listed herein are intentionally broad to allow maximum flexibility in designing interactive audio workstations and applications. There are examples describing scenarios where each of the features is needed, these are particularly useful in cases where multiple interactions are occurring simultaneously.

A single audio and telephone channel of an audio board are described, but ideally the board would have two completely symmetrical channels to allow multiple applications to access and use the phone and record channels simultaneously.

12.2 Electrical Characteristics

Input: Line level (about 0.3 to 1.2 volts), high impedance (5-15k)

Output: Line level, low impedance (100-150)

The connectors should be phono (RCA) jacks. Mini jacks or a multi-pin connector may be substituted if necessary due to space constraints.

The source for input signals will typically come from a microphone preamplifier or audio mixer and the sink will be a mixer or amplifier. The audio board will typically be connected to an audio system composed of professional or consumer grade components which have significantly better bandwidth and noise characteristics than the telephone system. *Every effort should be made to increase the signal to noise characteristics of the audio board.* A significant portion of the time the audio board will be used for the local recording of sounds with a microphone or playing over a speaker, not using the telephone network. For telephone use a bandwidth of approximately 3.4kHz is usually required. For local play and record of sounds a minimum bandwidth of 7kHz is desired, but 15kHz is preferable.

Optional microphone or speaker level jacks may be useful for certain stand-alone applications, but are not necessary for our typical research application.

Automatic gain controls are undesirable unless they can be disabled under computer control. AGC tends to amplify background noise during periods of silence and causes non-linearities that may be detrimental to speech recognition and other forms of processing.

12.3 Audio Input

Audio input to the board can be used for either direct recording of sounds or to be transmitted over the phone line.

A microphone can be used to record a prompt.

Playing live music or the output of a text-to-speech synthesizer over the phone line, i.e. "music on hold".

During a teleconference, microphone input to the board can simultaneously be captured and recorded for archiving as well as transmit to the other party in the conference.

12.4 Audio Output

Audio output from the board should either come from the play channel or received off the telephone line. The board should also be able to monitor received audio while recording or playing during a telephone call ("monitor mode").

A previously recorded prompt can be played over a speaker in the workstation.

A simple speaker and microphone in a workstation are used to create a simple half-duplex speakerphone using both silence bits (see section 12.7) to switch modes.

A full-duplex speakerphone can be constructed using external echo canceling hardware.

A *Phone Slave*-style answering machine receives an incoming call. When the caller first speaks her name it is routed to the speaker in my workstation as well as a speech recognizer so that the user can "screen" the call and possibly interrupt the answering machine. Note that "monitor mode" happens simultaneously with recording.

12.5 Telephone Connections and Functionality

Standard functions including on/off hook, call progress detection, DTMF detection and generation.

The board should have a loop through connection for a simple telephone handset. The board should be able to detect when handset is raised or lowered. If the board is inactive, the handset is connected through directly to the line.

Optionally terminate play/record on loop current reversal (on some PBXs, a sign that the other party has hung up, but not reliably on all). Optionally terminate play/record on dial tone detect (i.e., other party has hung up a while ago). Optionally terminate play/record on any DTMF detect.

Note that it is typically the host computer's responsibility (VOX) to stop the transfer of data to or from the audio board's buffers. The host must therefore get notified of such termination events or be able to read an on-board flag before performing each transfer. It is desirable that there be a "play buffer" *and* a "record buffer" on the board which can be used in a sequential manner without interaction from the host. For example, at the termination of playing a prompt from the play buffer, the board automatically starts recording into the other buffer ("record ahead"). This effectively implements a queuing mechanism of depth one for rapid response in interactive applications. The application can always disregard the buffer if the termination condition was not interesting.

12.6 Interconnections

The hardware should be fairly flexible in terms of on-board interconnections to allow maximum flexibility in environments where there is not a lot of external switching capability.

Output from the play channel should be switchable to either the phone transmit or audio output. The switches must be independent because all three possible switching combinations may be needed:

For an answering machine play over the phone only, not into the office.

For playing an "audio reminder" into the office only while on the phone.

For playing an archived sound simultaneously over the phone and into the room during a teleconference.

In a similar fashion, input to the recorder can come from either the phone receive or audio input channels. It is also desirable to have them separately switchable:

Record prompt from user while on the phone, that is put the caller on hold and locally record a message.

Record only the remote caller while a local microphone is active, that is switch off the microphone and record the other party.

Record both local and remote audio during a teleconference. The signals are effectively summed.

In terms of direct interconnections and monitoring, it is necessary to be able to:

Monitor phone receive on the audio out channel for call screening.

Connect audio input to phone transmit for "music on hold".

Connect phone receive to audio output while simultaneously connecting audio input to phone transmit for speakerphone applications. Note that this eliminates the possibility of simply monitoring whatever is on the record channel. Feedback and echo cancellation are not the responsibility of the audio board. It is possible to use a headset mounted microphone and ear piece or a special echo cancellation unit.

Monitor audio input to audio output as a simple system check of microphone and speaker configuration. This is a nice feature, but not essential. A digital loop is sufficient.

It is *not* desirable to connect phone receive to phone transmit.

These configurations lead to the interconnection scheme shown if Figure 12.1. While this capability may be implemented with simple analog switches, it is left up to the board designer to possibly implement these digitally to reduce the switching components needed.

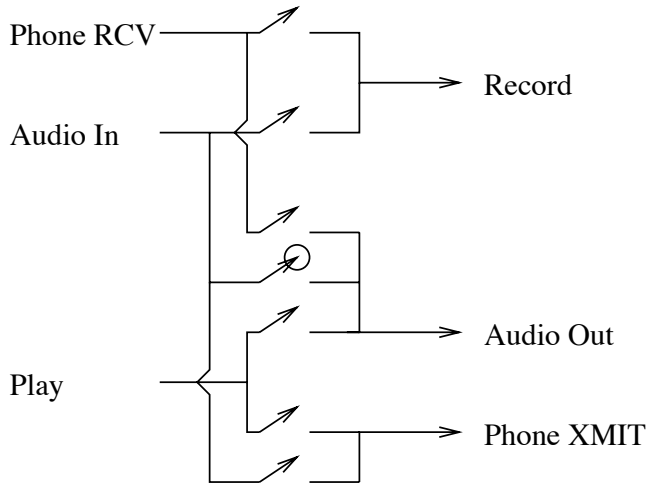


Figure 12.1: Audio I/O Interconnections. Phone RCV is signal from the hybrid, Phone XMIT is the signal to the hybrid. Audio In and Audio Out are to connectors to external devices. Play and Record are to and from the ADPCM recorder. Note that the switch from Audio In to Audio Out (circled) is optional.

12.7 Silence Detection

The hardware should be capable of detecting in real-time whether there is silence or audio/voice present on the telephone line. Silence detection should work regardless of whether the board

is recording or playing. If there is sufficient channel separation with the on-board hybrid, it is desirable to determine if there is audio present on both the transmitted and received telephone signal.

It is desirable to be able to assign a pause length to the silence detection hardware. In a time shared Unix operating system it is best to have all time critical functions implemented at the hardware/firmware level. The relatively simple switching algorithm needed for a half-duplex speakerphone should, for example, be implemented in firmware. If there has been silence/sound on the line for *pause_length* seconds, we can be reasonably sure that that there is or is not speech present. The state of the silence bit should be obtainable by software in real-time. An application program should be able to query the board to get the silence status or be interrupted when the silence bit changes state or goes into a particular state.

The "silence bit" may also be used as a termination condition for recording. For this use, it is desirable that the initial and final pause lengths can be different.

> Imagine two people are engaged in a teleconference with a shared graphical workspace in which only one person has control of the keyboard and mouse at a time. The silence bit is used for "floor control" in the teleconference, i.e. whom ever is talking has control of the keyboard and mouse. If both conferees have an audio board it is necessary to only capture the silence information on the receive end (or possibly the transmit end). However, if there is only a single audio board it must be able to ascertain the silence bit on *both* the transmit and receive signals.

[Authors Note—This section needs a little more discussion and thought as to the number and location of the silence sensors.]

Possible locations for the location of the silence detectors include:

1. Silence-Detect-1 could appear on Phone RCV and Silence-Detect-2 could appear on Phone XMIT, but that then precludes using silence detection to end a recording created with a microphone.
2. Silence-Detect-1 could appear at Record to eliminate the problem above. Silence-Detect-2 should then go on Phone XMIT (or Audio Out?), but there might be conferencing cases where we want to make an archival recording of both Phone RCV and Audio In, so the silence bit on the record channel does not clearly indicate which party is talking in a single board configuration. Using both bits it is possible to deduce if there is audio on the Phone RCV line *or* the Audio In line, but it cannot uniquely determine if there is audio on both lines. This configuration may be sufficient for most applications.
3. Silence-Detect-1 could be on Phone RCV and Silence-Detect-2 could be on Audio in. This would allow for the termination of recording in all cases as well as determining who is talking in a single board conference.

4. Silence-Detect-1 could go on Record and Silence-Detect-2 could go on Audio in (or Phone RCV), but this again does not uniquely determine if there is audio on both the Phone RCV and Audio In lines.

The best configuration is thus as described in case three, but case two or case four would be sufficient. If basing the hardware on existing design with a silence detector already at Record, case four may be the easiest to implement.

In a possible configuration where we want to do silence detection on the Play channel case two is better than three, unless we want to add a third silence detector.

Bibliography

[1] B. Arons, C. Binding, K. Lantz, and C. Schmandt. The VOX Audio Server. Technical report, Olivetti Research Center, Aug 1988.

[2] C. Binding. *The Mercury Remote Execution Language*.

[3] C. Binding. *VOX Audio Server Device Level Interfaces*.

[4] C. Binding. The VOX Audio Server IPC module. Interface Specification of TCP/IP IPC Module.

[5] C. Binding. *VOX Audio Server Programmer Interface*.

[6] C. Binding. *Specification and Implementation of a User Interface Management System*. PhD thesis, University of Washington, August 1987.

[7] Xerox Company. *Courier: The Remote Procedure Call Protocol*. Xerox Systems Institute, 475 Oakmead Parkway, Bldg. 5, Sunnyvale CA 94086, 1981. Order Nr. XNSS 038112.

[8] Computer Science Division, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley. *UNIX Programmer's Manual, Reference Guide*, 4.2 berkeley software distribution edition, 1984.

[9] E. Cooper. *Writing Distributed Programs with Courier*. Computer Science Division, EECS, University of California, Berkeley, 1982.

[10] K. Edwards. *VOX Audio Server Configuration Guide*.

[11] J. Gosling. Sundew - a distributed and extensible window system. In et al. Hopgood, F.R.A., editor, *Methodology of Window Management*, pages 47–58. Springer Verlag, 1986.

[12] et. al. L. Cannon. *Recommended C Style Guide and Coding Standards*. Bell Labs, University of Toronto, University or Washington. This document is an updated version of the *Indian Hill C Style and Coding Standards* paper.

[13] K. Lantz and W. Nowicki. Structured graphics for distributed systems. *ACM Transactions on Graphics*, 3(1):23–51, January 1984.

[14] K. Lantz, P. Tanner, C. Binding, K. Huang, and A. Dwelly. Reference models, window systems, and concurrency. *ACM SIGGRAPH Computer Graphics*, 21(2):87–97, April 1987.

[15] B. Nelson. Remote procedure call. Technical Report CSL-81-9, Xerox PARC, May 1981.

[16] A. Ruiz. Voice and telephony applications for the office workstation. In *Proceedings 1st International Conference on Computer Workstations*, pages 158–163. IEEE Computer Society, Nov 1985.

[17] R. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–106, Apr 1986.

[18] C. Schmandt and B. Arons. A conversational telephone messaging system. *IEEE Trans. on Consumer Electr.*, CE-30(3):xxi–xxiv, Aug 1984.

[19] C. Schmandt, B. Arons, and C. Simmons. Voice interaction in an integrated office and telecommunications environment. In *Proceedings of 1985 Conference*. American Voice I/O Society, 1985.

[20] C. Schmandt and M. McKenna. An audio and telephone server for multi-media workstations. In *Proceedings 2nd IEEE Conference on Computer Workstations*, pages 150–160. IEEE Computer Society, Mar 1988.

[21] D. Swinehart. Telephone management in the Etherphone system. In *Proceedings of GlobeCom 87*. GlobeCom, Nov 1987.

[22] D. Swinehart, L. Stewart, and S. Ornstein. Adding voice to an office computer network. Technical Report CSL-83-8, Xerox Palo Alto Research Center, Feb 1984.

[23] D. Terry and D. Swinehart. Managing stored voice in the Etherphone system. In *11th ACM Symposium on Operating System Principles*, pages 48–61. ACM SIGOPS, Nov 1987.

[24] J. Warnock. *PostScript Language Reference Manual*. Addison-Wesley Publishing Company, 1985.

Index