The Sound Manager: A Software Architecture for Device Independent Sound

Mark Lentczner
John Worthington

Apple Computer, Inc.
20525 Via Tela
Cupertino, CA 95014 USA

Abstract

The Sound Manager is a collection of system routines supporting sound and music on Apple Computer's Macintosh™ line of personal computers. The system was created and implemented by the authors. The goals of the project, details of current implementations, and future directions are discussed.

Motivations

The initial motivation for the Sound Manager came from a desire to achieve for sound what had been done for graphics at Apple: build an architecture. By this we wanted to create a system that would allow the developers of applications to code to a conceptual model of sound that would not change from machine to machine. As it was, the Macintosh and Apple LiC™ provided software tools for the developer, but they were, for the most part, very machine specific.

In 1986 the then newly formed Sound Group began to look at architectures for sound. Furthermore, the Apple Sound Chip (ASC) was being designed for the Macintosh II. As the ASC was a far departure from the sound hardware in the original Macintosh, it was decided that the new sound architecture would have its first implementation on the Macintosh II.

We started with the goals: hardware independence, complete control, accessibility, and extensibility.

The first two goals deal with the conflicting desires to have an architecture that would work on machines with entirely different hardware, and to allow developers to get at all the features we put in a machine. Reconciling these two goals is perhaps the major achievement of the Sound Manager.

The other two goals are true for any piece of software written at Apple. In our experience, many different kinds of software writers will want to use the work we create. Therefore, they must be both easy for the enthusiast and comprehensive for the experienced developer. In the world of personal computers you simply can't 'recompile the kernel' to load new features. In addition we had no expectations that what we wrote would be perfect. Therefore we insisted that the Sound Manager be extensible.

Technical Overview

In the simplest case an application makes a sound by passing the appropriate commands through a
channel to a synthesizer (Figure 1). The synthesizer interprets the commands and takes any necessary action, such as making a sound.

One of the benefits of this approach is that it isolates the application from changes to the sound hardware. New features or additional sound hardware frequently can be supported with no change to applications.

**Channels, Synthesizers, and Modifiers**

Synthesizers are the only part of the Sound Manager architecture that is dependent on the specific sound hardware available. Sound Manager commands passed down the channel are interpreted by the synthesizer and the necessary actions performed. Programmers needing to access some hardware directly can write a synthesizer, localizing hardware dependencies and minimizing the amount of code having to be rewritten later.

Sound commands sent by the application are placed in the channel's queue sequentially. The Sound Manager handles removing commands from the queue and sending them down the channel at the correct time. Commands without a duration are assumed to take no time, although the timer code is cognizant of the actual amount of time taken to process the command. When commands reach the end of the channel they are passed back to the Sound Manager. This provides a mechanism for synthesizers to communicate.
with the Sound Manager as well as letting the Sound Manager know when commands have been completed.

In the more general case, there may be one or more modulators installed in the channel ahead of the synthesizers (Figure 2). Modulators are similar to synthesizers in that they have the same programmatic interface and respond to Sound Manager commands. They differ in that their primary function is modifying commands or parameters rather than communicating with the host. Transposition, amplitude scaling and the like can be implemented as modulators. Modulators can also send more than one command for each command received, allowing features such as chording and arpeggiation to be easily implemented as well.

Because of the well defined interface, modulators can be written without any knowledge of the application or synthesizer. This allows a music programmer to create a tool box full of modulators that can be added for effect.

Commands

A set of Sound Manager commands is defined by Apple, with room for additional commands to be added for specific synthesizers. A command always has the same generic eight byte format, as shown in Figure 3. All of the commands defined by Apple fall into one of the following categories: performance, timing, and housekeeping. There are additional sets of commands specific to the synthesizers provided on the Macintosh system disks.

The performance commands of either fixed or unspecified duration can be specified. Over the course of these sounds, the frequency and/or amplitude can be altered. The pitch of a sound can be specified either by an absolute frequency or by semitone. An example of how the parameters are packed for the Note command is shown in Figure 3.

The timing commands provide for synchronizing the command stream in different channels as well as requesting periodic or one time "ticks" after a specified amount of time. All time events in the Sound Manager are specified in one-half millisecond units.

Housekeeping commands allow a variety of tasks such as starting and stopping the command queue and querying the synthesizers.

Lavishly, there are a commands that are created for each particular synthesizer. These allow access to the features of any particular synthesizer. Because these commands only set up the features, and not sound the notes, even if a command stream is sent to a synthesizer that doesn't have the features, the note will still play. Commands that are not understood are passed through the channel with

Currently all our synthesizers use equal temperament. However, there is no problem in the future with having a command to change the tuning and thus interpreting the semitones accordingly.

ICNC Proceedings 1988 374
Programmatic Interface

The Sound Manager interface as seen by program-
mers consists of seven calls. These calls allow the
creation and disposal of channels, installation of
synthesizers and modifiers, and transmission of
commands both via the queue and directly to
the modifiers and synthesizers.

Like most Macintosh software, the Sound Manager
makes extensive use of resources. There are two
resource types available: 'nuth' and 'nuth' (Figure 4).

Synthesizers and modifiers are usually stored as
resources of type 'nuth'. These resources contain
executable code. The code is a single function with
the entry point at the beginning of the resource.
Synthesizers and modifiers are usually specified to
the Sound Manager using the ID of the 'nuth'
resource that contains it.

The 'nuth' resource may contain one or all of the
following:
- a list of modifiers and synthesizers ('nuth'
  resources) to load
- semi or all of the commands needed to
  produce a sound
- data tables (e.g., wave tables, sampled
  sound buffers, or patch information) to be
  used in generating sounds.

Thus a 'nuth' resource can contain a melody, with
no information about instrumentation in the
channel setup, a single digitized sound with the
necessary commands to play the sound, an
instrument, or a complete piece specifying the
instrument, score, and channel setup.

In the latter case a special Sound Manager routine
can be used. MPlay will "play" a 'nuth' resource
by setting up a channel with the specified
synthesizer and modifiers, and sending in order
the commands specified in the resource. This
method is especially useful for non-music
programmers since they can take 'nuth' resources
created by someone else and easily incorporate
them into an application.

Implementation Notes

The Sound Manager and synthesizers supplied by
Apple is over 4000 lines of code written primarily
in MWC C, with the interrupt and timing routines
hand coded in assembly language. Efforts have
been made throughout the code to minimize the

375

ICMC Proceedings 1988
Event Timing

While the programmatic interface has a timing accuracy of one half millisecond, any particular implementation of the Sound Manager may not. Internally the time is always kept to full accuracy in the data structures. The code itself is written with a unique two-level timer design. The main bulk of the Sound Manager also deals with fully accurate times. It works with a virtual timer whose resolution is half a millisecond but whose accuracy is unknown. The main code scans all queues in the Sound Manager to determine when the next event is supposed to happen and programs the virtual timer to interrupt at that time. When the main code finally gets interrupted it immediately interrupts the virtual timer for the actual time it had been opposed to the one the code asked to wake up at and accounts for this during the execution of events. Furthermore, when the main code is done processing all current events it again interrupts the virtual timer for the actual time and then takes into account the time spent processing the events.

The virtual timer is a set of routines built on top of whatever real timer is available in the machine. In the case of the Macintosh II it is a VIA 6552Z timer that counts down at 16MHz. This timer is programmed for the interval of one half millisecond and it interrupts when it hits zero. Like the main code, the virtual timer checks the real timer when it receives the interrupt to determine just how much time has really passed. It takes this into account when both interrupting the main code and when reprogramming the real timer for the next time period. Note that the virtual timer need not use the physical timer to interrupt every half-millisecond. On a slower machine it can request to be woken up at a slower rate. As the main code always interrogates the virtual timer before any real time, the virtual time can always indicate how much time has really passed to the main code and all timing can stay on track.

This two level design helps reduce the dependents on either interrupt latency or machine timing significantly. Furthermore it allows graceful degradation when things with in the system become irregular since it is totally self-correcting.

Supporting Different Hardware Capabilities

The Sound Manager is currently implemented on three machines with very different hardware capabilities for sound and varying amounts of processor power. The different capabilities of machines is further accentuated by the availability of plug-in cards with DSP chips, 16 bit data and high sample rates. Rather than deciding to support only the lowest common denominator, the Sound Manager and synthesizers try to render the sound as accurately as possible with the available hardware.

Because the hardware dependencies are localized in the synthesizers, there are several applications originally written for the Mac II that run without modification a Mac Plus using the Sound Manager and the Mac Plus implementation of the synthesizers.

We are currently looking at ways to generalize the Sound Manager architecture beyond the Macintosh environment. As the Sound Manager is ported to machines that have radically different hardware or very different software architectures numerous additional problems are likely to arise. Hopefully our approach of rendering sounds as closely as possible within the limits of the available hardware will allow applications to make sound with little or no modifications.

Future Enhancements & Unaddressed Issues

There are several areas the Sound Manager model does not cover. We are aware of these and are currently looking at ways to extend the Sound Manager to be more complete. We discuss some
of the more interesting of these are here:

Multitasking
Now that we have MultiFinder™, issues of con-
currency and sharing limited sound resources
across multiple applications are both important
and unresolved. Unlike other computer sys-
tems sharing problems, the solution for this is
very unclear. For example, even if you have
enough voices to handle two different requests
for sound, you probably don't want to hear
the 'printer-is-out-of-paper' sound while your
four-part Bach invention is being recorded on
your computer. Which sounds are given priority
and the disposition of sounds interrupted
should be based on the properties of the sound
and user's current activities. We are currently
looking at ways to classify and reorder what a
program is trying to achieve by mixing a sound.
After that we will need to find good heuristics
for how to schedule the sounds.

Audio Input
Currently the Sound Manager model has the
computer as the sole controller with little support
for external control or audio input. There is no
support in the current Sound Manager model for
input of audio data. Clearly a device independent
model of audio input is desirable since it would
extend our approach to applications that want to
record audio data.

MIDI
Originally we set out to encompass MIDI into
the Sound Manager. To a degree this works. One
channel is a MIDI synthesizer and gets
notes out in a synthesizer. However, from
the computer's point of view, MIDI is best thought
of as a communications protocol, not the piece
of sound hardware. Consequently, it does not fit
into the Sound Manager model as we would
have liked. Furthermore, there were a large
number of MIDI applications for the Macintosh
when the Sound Manager was started. When
connected, we found that these developers had
a large host of non-compatible needs for MIDI
control. For example, the timing requirements
were all different and the filtering of MIDI
commands was varied.

One approach that we are currently investigating
is to create a new tool for MIDI. This tool would
meet the needs of these developers using MIDI, in
d user independent manner. In addition we
could create links between the two tools, MIDI and
Sound Manager, so that control could be handled
from either side. This type of system would allow
developers wishing to take advantage of the
programmability of the Sound Manager access to
MIDI while giving MIDI developers a dedicated
tool that better meets their needs.

Summary
The Sound Manager is Apple's second generation
sound tool for programmers. It was created to
support both the new hardware and new software
that is beginning to appear on Apple's machines.

In supporting these aims we have created an
architecture for sound and music in a personal com-
puter environment. This architecture allows us to
be hardware independent, supporting old, current
and future hardware for sound on our computers.
Furthermore, in doing the first implementation of
the architecture for the Macintosh Plus, SE, and II
computers, we have learned some interesting tech-
niques for handling audio and traditional machine
specific aspects such as timing.

The Sound Manager was originally conceived and
implemented by the authors and their group: the
Advanced Video/Graph Sound Group and Product
Development Sound Group. Through a combina-
tion of developer input and innovative design we
felt that we have created a sophisticated tool
for programmers to use. By providing these
tools, it is our hope that we are helping people
create new music and sound applications.