Audio Development Guide

Nintendo

NINTENDO 64

Audio Development

Guide

NUS-06-0151-001A
Released: 11/18/97

D.C.N. NUS-06-0151-001 Rev. A

Table of Contents

AUDIO SYSTEM OVERVIEW 7
SGI AUDIO LIBRARY 9
  SOUND PLAYER 9
  SEQUENCE PLAYER 9
  SYNTHESIS DRIVER 10
  AUDIO MICROCODE 10
SOUND PLAYBACK 13
  OVERVIEW OF SOUND PLAYBACK 13
  SYNTHESIZER CONFIGURATION 14
    Initializing the Audio Heap 15
    Setting the Hardware Playback Rate 16
    Creating the Synthesizer 17
    Preparing the DMA Callback Routine 20
  PLAYING A SEQUENCE 21
    Reading the Audio Bank from ROM 21
    Reading MIDI Sequence from ROM 22
    Sequence Player Startup 22
    Sequence Structure Initialization 26
    Configure the Sequence 27
    Audio Bank Initialization 28
    Audio Bank Settings 29
    Sequence Playback 30
    Stopping a Sequence 31
    Sequence Player Deletion 32

NUS-06-0132-001A
Released: 9/9/97

Table of Contents (continued)

PLAYING SOUND EFFECTS 34
Reading the Audio Bank from ROM 34
Sound Player Startup 35
Audio Bank Initialization 36
Allocation of Sound Resources 37
Sound Selection 38
Sound Playback 38
Stopping Sounds 39
Sound Player Deletion 39
AUDIO STACK EXECUTION 40
Buffer Preparation (triple buffer) 40
Synchronizing Retrace Events, SP Events 40
Adjusting the Frame Size 40
Creating the Audio Command List 41
Executing the Audio Task List 41
Audio DAC Settings 44
USE OF AUDIO TOOLS 45
SOUND DEVELOPMENT PROCESS 45
Creating the Wave Table 45
Creating Bank Files 45
TOOLS FOR CREATING WAVE TABLES 46
Tabledesign Tool 46
"vadpcm_enc" Tool 47
The ic (Installment Compiler) 48
Creating the ic Source File 50
Other Tools 59
TOOLS FOR CREATING THE SEQUENCE BANK 60
The midicvt Tool 60
The midicomp Tool 61
The sbc Tool 62
Other Tools 63

Table of Contents (continued)

PROGRAMMING CAUTIONS 65
COMMON VALUES 65
MANAGING RESTRICTIONS AND ALLOCATING RESOURCES ... 65
Determining Hardware Playback Rates 66
Limitations on Voice and Processing Time 66
Sound and Music (BGM) Bank Partitioning 67
ROM Space Limitations 67
CREATING SAMPLE DATA 67
PLAYBACK PARAMETERS AND THE INSTALL (.INST) FILE 69
Sample Parameter Settings for the Install File 69
Use of the Install File 70
Envelopes 70
Keymap 71
Tuning Samples Recorded at the Hardware Playback Rate 71
Tuning Samples Recorded at a Rate Other than the Hardware Playback Rate 72
Sounds 73
Instruments 74
Banks 74
Making a Bank File 74
MIDI FILES 75
Loops in a Sequence 75
Nested Loops 77
Creating a Compact MIDI File that Contains Loops 78
Problems Related to Loops 78
CREATING THE MAKEFILE 78
GENERAL MIDI AND N64 79

AUDIO SYSTEM OVERVIEW

Nearly all of the audio functions for Nintendo 64 (N64) are implemented by software. The N64 does not use a sound-generator chip or dedicated DSP. This provides a greater degree of flexibility in sound development. For example, there are no hardware-imposed restrictions on the number of simultaneous sounds which can be produced. A great variety of effects can be implemented with software.

Because there are limitations on processing speed and memory, it is necessary to examine a means of implementing high-quality audio that does not impose a burden on other types of processing, such as graphics. In addition, not all special effects that are theoretically possible are supported by standard libraries. Thus, the programmer must create waveform synthesis drivers for those effects that are not supported.

Waveform synthesis processing is performed by coordinated operation of the CPU and RSP. The CPU creates an audio command list in the audio command buffer, in RAM. This list is interpreted and executed by the audio microcode run by the RSP, resulting in actual waveform synthesis.

In general, audio data is synthesized by the CPU and RSP for each field of a video signal (1/60 sec for NTSC, 1/50 sec for PAL) and stored in the audio buffer. It is necessary to reserve a buffer just large enough to accommodate the playback rate (the rate that determines the amount of sample data output per second).

The AI (audio interface) reads monaural or stereo waveform data from a memory area specified as the audio buffer at fixed time intervals. It sends this data to the DAC (D/A converter for audio output), resulting in sound output.

A figure that outlines the audio system is shown on the following page.

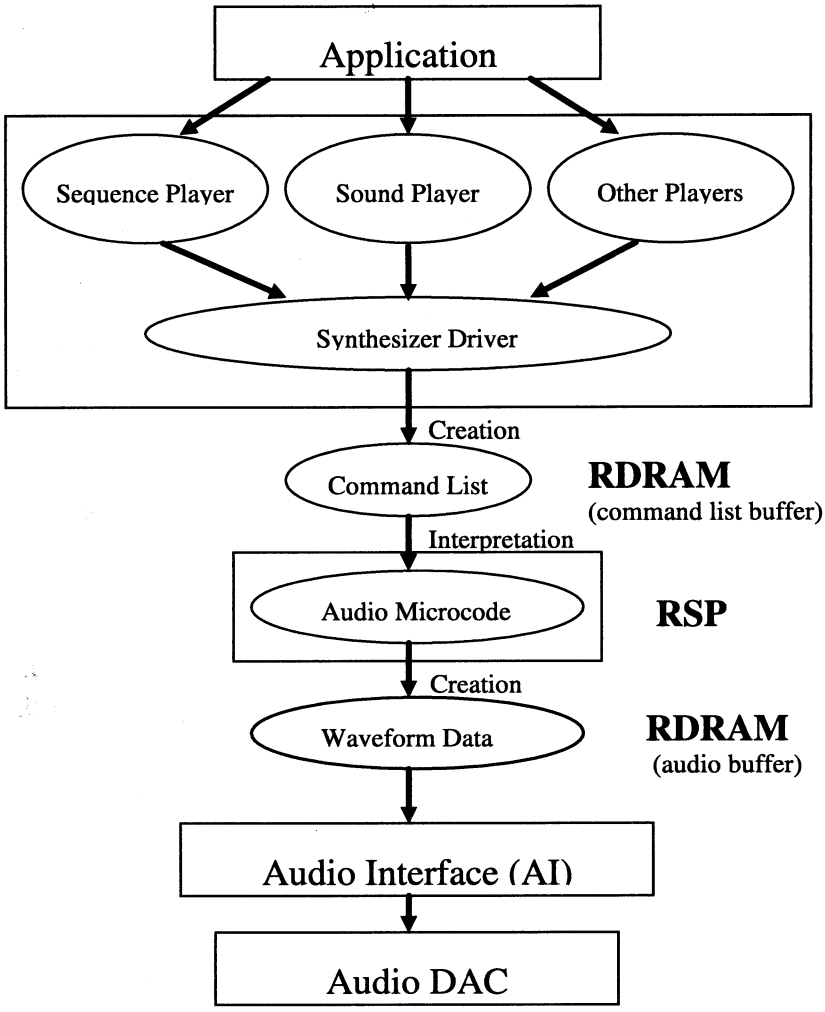


Figure 1: Sound Processing Flow

SGI AUDIO LIBRARY

The descriptions in this manual are based on the assumption that the SGI (Silicon Graphics Inc.) audio library is being used. However, because the synthesis drivers and microcode used are common to those used with other players (Nintendo 64 Sound Tools audio library and players made by the user), it is likely that the information provided here can also be applied to those players.

The N64 audio library made by SGI is a lightweight function library. When used with the N64, it can synthesize and operate audio in an interactive format. This library supports both sound sample playback and wave-table synthesis. Wave-table synthesis is a music synthesis system that involves creating a table containing sample sounds (wave table), changing components of the sample sounds in the table (such as the pitch), and playing music. These functions are implemented using four software objects; a sound player, sequence player, synthesizer driver, and audio microcode.

SOUND PLAYER

The sound player is used mainly when sound effects are reproduced. It can reproduce both ADPCM-compressed sound and 16-bit uncompressed sound, as well as looped or non-looped sound.

SEQUENCE PLAYER

There are two types of sequence players. One is used to play standard MIDI files of format Type 0. The other is used to play compact MIDI files, a special N64 format. Both types perform sequence, instrument bank, and sequence resource allocation; sequence interpretation; and MIDI message scheduling.

Aside from having different sequence formats, the functions of the two sequence players are nearly identical. However, the player of Type 0 standard MIDI files is capable of external loop control, while the compact MIDI player executes loops using loop commands embedded in the sequence data. The compact MIDI player can create a different loop for each track, making nested loops possible.

Both the sequence player and sound player are clients of the synthesis driver. The synthesis driver can have any number of clients, including multiple sound players and sequence players.

SYNTHESIS DRIVER

The synthesis driver creates the audio command list executed by the RSP audio microcode. The application creates tasks based on this audio command list and passes these tasks to the audio microcode.

Use of the synthesis driver enables the player to allocate a wave table to the reproduced voice (synthesizer voice: a voice actually processed by the synthesizer) and thereby control the playback parameters.

The synthesis driver can be used to reproduce ADPCM-compressed 16-bit audio files. The N64 ADPCM compression format compresses waveform data to approximately 1/4 of its original size.

Multiple players can be registered in the synthesis driver. In general, the sequence player and sound player are registered for use, but players created by the user can also be registered.

To conserve memory space, waveform data is generally kept in memory, and data in ROM is retrieved by DMA as needed. The routines for this processing are stored in the configuration structure required to initialize the synthesis driver. These routines must be prepared at the application level.

AUDIO MICROCODE

The audio microcode is used for waveform synthesis. It uses the RSP as the processor for waveform synthesis.

The audio microcode processes tasks received from the application and synthesizes 16-bit L/R stereo sample data.

The RSP can perform eight product-sum calculations in a single cycle, using the vector unit (VU). Using this method, waveform synthesis is performed in units of 8 samples per cycle.

The RSP can directly access only the DMEM within the RSP. Because the capacity of DMEM is limited, the number of samples that can be processed at one time is also limited. Consequently, a single frame's worth of data must be divided and then processed.

The most common synthesis drivers process 160 samples at one time. (A synthesis driver is currently available from Nintendo which processes in units of 180 samples at a time.)

SOUND PLAYBACK

The descriptions provided in this section are generally aimed at programmers who are new to programming sound for the N64.

OVERVIEW OF SOUND PLAYBACK

The process of creating and reproducing sound is summarized in the following steps.

1. Create and initialize the necessary resources (usually audio heap, synthesizer, player).
2. Make repeat calls to `alAudioFrame` to create the audio task lists.
3. Execute these task lists with the RSP.
4. Call `osAiSetNextBuffer` to configure the output DAC, which outputs the audio.

Although the process of creating and initializing the required resources depends to some extent on the demands of the application, it generally proceeds according to the following steps.

1. Call `alHeapInit` to create the audio heap.
2. Call `osAiSetFrequency` to set the output frequency.
3. Call `alInit` to create the synthesizer. (`alInit` requires a callback routine that initializes the audio DMA structure.)
4. Create a message cue to receive signals that coordinate the timing of audio processing.
5. Create a player (e.g. sound player, sequence player) to add to the synthesizer.
6. Initialize the resources allocated to each player.

SYNTHESIZER CONFIGURATION

Before sound can be reproduced, the synthesizer must be configured. The configuration process is listed below.

1. Initialize the audio heap.
2. Set the audio playback rate.
3. Create the synthesizer.
4. Prepare the DMA callback routine.

Each of these steps is described in the following paragraphs.

Initializing the Audio Heap

The memory required for each function in the audio library is dynamically allocated from a memory area called the audio heap. Thus, to use the audio library, the audio heap area must first be initialized. The `alHeapInit()` function is used for this purpose.

alHeapInit()

Syntax

```
#include <libaudio.h>
void alHeapInit(ALHeap *hp, u8 *base, s32 len);
```

Arguments

`hp`	Pointer to the ALHeap structure, which indicates the heap to be initialized
`base`	Pointer to the beginning of the DRAM heap
`len`	Length of the DRAM heap in bytes

With sample programs such as *soundmonkey*, approximately 300 KB of memory is allocated as the audio heap. However, the memory size required for the heap area must be determined by considering factors such as the number of simultaneous sounds and the MIDI sequence data used.

Although there is no formula for deciding the appropriate heap size, the amount of remaining available memory can be determined by checking the ALHeap structure, which designates the heap. This amount can be computed using (heap->cur - heap->base). After all buffers needed by the application are allocated, please check the remaining available memory in the heap and evaluate whether the heap size is appropriate.

Note: Memory allocated to the audio heap cannot be released.

Setting the Hardware Playback Rate

Next, the hardware playback rate must be set. The `osAiSetFrequency()` function is used for this purpose.

osAiSetFrequency()

Syntax

```
#include <ultra64.h>
s32 osAiSetFrequency(u32 frequency)
```

Arguments

`frequency` frequency (Hz)

Return Value

Actual hardware playback rate

Example:

```
c.outputRate = osAiSetFrequency(OUTPUT_RATE);
```

Based on the `frequency` argument, `osAiSetFrequency()` computes an accurate quantity from an internal divisor and returns the actual frequency. Please set the `frequency` argument to 3000-368000 Hz for NTSC systems and 3050-376000 Hz for PAL systems.

Creating the Synthesizer

The `alInit()` function is used to create the synthesizer.

alInit()

Syntax

```
#include <libaudio.h>
void alInit(ALGlobals *globals, ALSynConfig *c);
```

Arguments

`globals`	Pointer to the ALGlobals structure
`c`	Pointer to the synthesis driver configuration structure

Before the synthesis driver is created, the synthesis driver configuration structure must be defined. The following are parameters for the synthesis driver configuration structure.

maxVVoices

This parameter sets the maximum number of virtual voices.

Virtual voices is a hypothetical voice parameter declared in a way that is most useful for the player. It is described by the ALVoice structure. Virtual voices must be allocated to players used in sound playback. Replacement of a virtual voice with a low priority sequence, once it has been allocated to a physical voice, by a virtual voice with a higher priority sequence is called "voice steeling." This occurs when (number of virtual voices) > (number of physical voices). Voice stealing can be prevented by assigning a priority sequence to the virtual voice.

Note: This parameter has not yet been implemented at the current stage of development (development environment 2.0H). Thus, setting this parameter will have no effect.

maxPVoices

This parameter sets the maximum number of physical voices (the number of voices for which wave-form synthesis is actually performed). This corresponds to a sound processing module composed of the ADPCM decompression, pitch shifter, and gain unit.

Allocating even more physical voices than virtual voices results in voices that cannot be used, or wasted voices. Therefore, these parameters should always be set so that the following is true: (number of physical voices) ≤ (number of virtual voices).

maxUpdates

This parameter establishes the maximum number of updates for undetermined parameters. This is concerned with internal parameter updates for the synthesis driver. Within the synthesis driver, parameter settings are processed and the physical voice processor module is divided before synthesis is performed. Thus, memory area is required to temporarily hold the parameter contents until the parameter settings are actually reflected. This parameter can therefore be used to specify the number of ALParams structures to allocate for internal parameter updates. In general, numerous updates are needed when numerous voices are supported.

maxFXBuses

This parameter identifies the maximum number of auxiliary effects buses. Because only one bus is currently supported, this parameter is ignored. Consequently, it can remain unspecified.

dmaproc

This parameter is a pointer to the procedure used during initialization of the DMA callback routine. The synthesis driver is structured so that waveform data does not reside in main memory (to conserve memory space). Consequently, waveform data in ROM must be retrieved by DMA as needed. The callback routine that performs this processing is specified here. This routine must be prepared by the application.

Although waveform data can be read into RDRAM in advance, this is not generally advisable from the perspective of memory efficiency. When developing sound for the 64DD, however, DMA from the disk as needed is not possible. Thus, the need may arise to put waveform data for sound effects in memory, in advance.

heap

This is a pointer to the memory heap used by the audio system. This specifies the heap area created by `alHeapInit()`.

outputRate

This is the number of samples generated by the synthesizer per second. Normally, the return value of `osAiSetFrequency()` can be substituted here.

fxType

This indicates the type of effect used. The following can be specified.

AL_FX_NONE	no effect used
AL_FX_SMALLROOM	reverb (small room)
AL_FX_BIGROOM	reverb (big room)
AL_FX_ECHO	echo
AL_FX_CHORUS	chorus
AL_FX_FLANGE	flange
AL_FX_CUSTOM	specified for a custom effect

params

This is a pointer to a parameter string used with `AL_FX_CUSTOM`. For information on these parameters, please refer to **"The Audio Library"** in the *Nintendo 64 Programming Manual*.

Examples of definitions used for the synthesis driver configuration structure are listed below.

```
#define MAX_VOICES    24
#define MAX_UPDATES   64
#define OUTPUT_RATE   44100

ALSynConfig   c;

c.maxVVoices    = MAX_VOICES;
c.maxPVoices    = MAX_VOICES;
c.maxUpdates    = MAX_UPDATES;
c.dmaproc       = &dmaNew;
c.heap          = &hp;
c.outputRate    = osAiSetFrequency(OUTPUT_RATE);
c.fxType        = AL_FX_SMALLROOM;
```

Preparing the DMA Callback Routine

A DMA callback routine that is called by the synthesis driver must first be created to retrieve waveform data in ROM and move it to RAM as necessary. A pointer must be created in the synthesis driver configuration structure that points to the DMA initialization routine.

This DMA initialization routine is called once for each physical voice. With its first call, it initializes the DMA buffer and, when waveform data is actually required, returns a pointer (`ALDMAproc` pointer) to the called function.

`ALDMAproc` accepts address, length, and status pointers for the required data and returns a pointer to the buffer where this data is stored as its return value.

For specific configurations, please refer to the sample programs.

The procedure described above enables a synthesizer to be created. The next objects created will be the clients of the synthesizer; the sequence player and sound player.

PLAYING A SEQUENCE

Playing a sequence involves the following procedure.

1. Read the audio bank from ROM.
2. Read the MIDI sequence from ROM.
3. Start the sequence player.
4. Initialize the sequence structure.
5. Configure the sequence.
6. Initialize the audio bank.
7. Configure the audio bank.
8. Sequence playback.

The following procedures are then performed, as needed.

9. Sequence termination.
10. Sequence player deletion.

Each of these elements is described below.

Reading the Audio Bank from ROM

To implement sound for audio, the wave table control file must first be moved to RAM using DMA transfer. DMA transfer is not described in detail here. Please refer to sample programs in `/usr/src/PR/`, such as `playseq`.

The following example is a typical DMA transfer of an audio bank.

1. Use `osCreatePiManager` to create a PI manager.
2. Use `osCreateMesgQueue` to create a message cue for confirming that a DMA transfer has been completed.
3. Use `alHeapAlloc` to reserve heap area for the audio bank.
4. Set the stage for DMA by using `osWritebackDCacheAll` to perform a CPU cache writeback.
5. Use `osPiStartDma` to initiate DMA transfer.
6. Use `osRecvMesg` to confirm completion of the DMA transfer.

Reading MIDI Sequence from ROM

To implement sound for a sequence, the MIDI sequence in ROM must first be moved to RAM using DMA transfer. DMA transfer is not described in detail here. Please refer to sample programs in `/usr/src/PR/`, such as `playseq`.

The following is a typical DMA transfer of a MIDI sequence. It is assumed in this process that a PI manager and DMA message cue have already been created.

1. To read in a MIDI sequence, the first part of the .sbk file header (version: 2 bytes + number of sequences: 2 bytes) must be read. For this to occur, the number of sequences must be known (refer to **"Audio File Format"** in the *Nintendo 64 Programming Manual*).

   Because the first part of the .sbk file header is 4 bytes in size, the header area is first reserved by `alHeapAlloc`, and a DMA transfer of 4 bytes is then performed.
2. Next, the entire header is transferred by DMA, including a ALSeqData structure corresponding to the number sequences.
3. `alSeqFileNew` is used to initialize the sequence bank file.
4. Based on information in the ALSeqData structure, the required MIDI sequences are transferred by DMA.

Sequence Player Startup

Either alSeqpNew() or alCSPNew()is used to start the sequence player. alSeqpNew() is the player used for playback of Type0 standard MIDI files and alCSPNew() is used for compact MIDI files.

alSeqpNew()

Syntax

```
#include <libaudio.h>
void alSeqpNew(ALSeqPlayer *seqp, ALSeqConfig *config);
```

Arguments

`seqp`	Pointer to the sequence player structure to initialize
`config`	Pointer to the sequence player configuration structure

`alSeqpNew` implements the configuration settings specified by `config` and initializes the Type 0 MIDI sequence player specified by `seqp` so that it can be registered with the synthesis driver as a client.

Please use caution when allocating memory from the audio heap specified by the `config` structure.

Before `alSeqpNew()` is called, the sequence player configuration structure must be defined. This structure consists of the following parameters.

maxVoices

This parameter sets the maximum number of virtual voices supported.

maxEvents

This parameter sets the maximum number of internal events supported.

In the sequence player and sound player, sound changes occurring in chronological order are managed by a structure called an event. `ALEventListItem` structures are used for event storage and the number of allocated `ALEventListItem` structures can be specified here. To play complex sequences, more events are required.

maxChannels

This parameter sets the number of MIDI channels supported. Typically, the value 16 is specified.

debugFlags

This parameter sets flags for reporting audio errors. If this is set to 0, errors are not reported. Any of the following flags can be specified, or several can be combined using the OR symbol (|).

NO_SOUND_ERR_MASK	Suppress error messages when there is no sound that can be used as that required by the specified pitch.
NOTE_OFF_ERR_MASK	Suppress error messages when a Note-Off occurs but the note of the specified channel does not include the voice currently being played.
NO_VOICE_ERR_MASK	Suppress error messages when voice allocation is requested but no usable voices are available.

***initOsc**

This specifies a function pointer for the initialization handle of the oscillator which processes vibration and tremolo effects. When such effects are not used, 0 must be specified.

***updateOsc**

This specifies a function pointer for the update handle of the oscillator which processes vibration and tremolo effects. When such effects are not used, 0 must be specified.

***stopOsc**

This specifies a function pointer for the interrupt handle of the oscillator which processes vibration and tremolo effects. When such effects are not used, 0 must be specified.

heap

This is a pointer to the initialized audio heap. The audio heap created by `alInitHeap()` is specified.

The following are examples of definition strings for the sequence player configuration structure and use of `alSeqpNew()`.

```
#define MAX_VOICES    24
#define MAX_EVENTS    32

ALSeqPlayer   seqp;
ALSeqConfig   seqc;

seqc.maxVoices          = MAX_VOICES;
seqc.maxEvents          = MAX_EVENTS;
seqc.maxChannels        = 16;
seqc.heap               = &hp;
seqc.initOsc            = 0;
seqc.updateOsc          = 0;
seqc.stopOsc            = 0;
seqc.debugFlags         = NO_VOICE_ERR_MASK |
NOTE_OFF_ERR_MASK | NO_SOUND_ERR_MASK;
alSeqpNew(&seqp, &seqc);
```

The following functions are used with the compact MIDI player.

alCSPNew()

Syntax

```
#include <libaudio.h>
void alCSPNew(ALCPlayer *seqp, ALSeqpConfig *config);
```

Arguments

`seqp`	Pointer for initializing the compact MIDI sequence player structure.
`config`	Pointer to the sequence player configuration structure.

`AlCSPNew` performs the configurations specified by `config` and initializes the compact MIDI sequence player and `seqp` so that the player can be registered in the synthesis driver as a client. The contents of `config` and all other features of this function are the same as for `alSeqpNew`.

Sequence Structure Initialization

To enable sequence playback with the MIDI sequence player, the MIDI sequence structure must first be initialized. To reproduce a MIDI sequence, the sequence structure stores information related to sequence data . This storage is performed using `alSeqNew()` or `alCSeqNew()`. `alSeqNew()` is used for Type 0 MIDI file sequence players, and `alCSeqNew()` is used for compact MIDI file sequence players.

alSeqNew()

Syntax

```
#include <libaudio.h>
void alSeqNew(ALSeq *seq, u8 *ptr, s32 len);
```

Arguments

`seq`	Pointer to the `ALSeq` structure to be initialized.
`ptr`	Pointer to the MIDI data.
`len`	Length of the MIDI data in bytes.

This function initializes the `ALSeq` runtime structure with the MIDI sequence data indicated by `ptr` and the data length indicated by `len`.

The following functions are used with the compact MIDI player.

alCSeqNew()

Syntax

```
#include <libaudio.h>
void alCSeqNew(ALCSeq *seq, u8 *ptr);
```

Arguments

`seq`	Pointer to the `ALCSeq` structure to be initialized.
`ptr`	Pointer to the compact MIDI data.

Configure the Sequence

Once initialization of the sequence structure is completed, the sequence to be reproduced is set in the player. This is accomplished using `alSeqpSetSeq()` or `alCSPSetSeq()`. `alSeqpSetSeq()` is used for Type 0 MIDI players and `alCSPSetSeq()` for compact MIDI players.

alSeqpSetSeq()

Syntax

```
#include <libaudio.h>
void alSeqpSetSeq(ALSeqPlayer *seqp, ALSeq *seq);
```

Arguments

`seqp`	Pointer to the sequence player.
`seq`	Pointer to the sequence structure in which the target sequence is registered.

In the sequence player, `alSeqpSetSeq()` sets the sequence structure (`seq`) in which the sequence to be reproduced is stored.

`alSeqpSetSeq()` always sets the sequence tempo to 120 BPM (Beats Per Minutes, the number of quarter notes counted per minute).

When a bank is linked to the sequence player, `alSeqpSetSeq()` resets the channel parameters of the sequence player. The initial instrument in the bank is used as the default program. The pan, volume, priority, and pitch-bend range of that instrument are used as default channel parameters. When a percussion instrument is in the bank, this instrument is used for MIDI channel 10. The setting for effects is `AL_DEFAULT_FXMIX`.

The following functions are used for compact MIDI players.

alCSPSetSeq()

Syntax

```
#include <libaudio.h>
void alCSPSetSeq(ALCSPlayer *seqp, ALCSeq *seq);
```

Arguments

`seqp`	Pointer to the compact MIDI sequence player.
`seq`	Pointer to the compact MIDI sequence structure in which the target sequence is registered.

Audio Bank Initialization

The audio bank used for sequence playback must be initialized. `alBnkfNew()` is used for this purpose.

alBnkfNew()

Syntax

```
#include <libaudio.h>
void alBnkfNew(ALBankFile *ctl, u8 *tbl);
```

Arguments

`ctl`	Pointer to the control data (.ctl).
`tbl`	Pointer to the wave table data (.tbl).

Two of the files created by the IC (instrument compiler) are used in sound playback. One of these is the control file (`.ctl`), which is responsible for handling instrument performance information. The other is the wave-table file (`.tbl`), responsible for handling wave-table data (waveform data).

The control file includes the offset address, which references the wave-table data. To improve runtime performance, the offset is converted to a virtual address by `alBnkfNew`.

In general, control data is transferred to DRAM at even an earlier stage than wave-table data. This is because wave-table data always turns out to be sizable and, to prevent wasteful use of DRAM, it is transferred to DRAM using DMA as needed. When the size of the wave-table file is small, this data is loaded into DRAM in advance to minimize DMA requests during sequence playback.

Audio Bank Settings

Once the audio bank is initialized, the instrument bank used by the sequence player is configured. `alSeqpSetBank()` or `alCSPSetBank()` are used for this purpose. `alSeqpSetBank()` is used for Type 0 MIDI sequence players, and `alCSPSetBank()` is used for compact MIDI sequence players.

alSeqpSetBank()

Syntax

```
#include <libaudio.h>
void alSeqpSetBank(ALSeqPlayer *seqp, ALBank *b);
```

Arguments

`seqp`	Pointer to the sequence player.
`b`	Pointer to the instrument bank used.

alCSPSetBank()

Syntax

```
#include <libaudio.h>
void alCSPSetBank(ALCSPlayer *seqp, ALBank *b);
```

Arguments

`seqp`	Pointer to the compact MIDI sequence player.
`b`	Pointer to the instrument bank used.

The following is an example of audio bank initialization and settings.

```
alBankfNew((ALBankFile *)midiBankPtr,
_miditableSegmentRomStart);
midiBank = ((ALBankFile *)midiBankPtr)->bankArray[0];
alSeqpSetBank(seqp, midiBank);
```

In this example, it is assumed that the control data are in the RAM area indicated by `midiBankPtr` and the wave-table are in the ROM area indicated by `_miditableSegmentRomStart`.

In the second line, the initial bank containing control data (`bankArray[0]`) is specified as the instrument bank (`midiBank`). For information on the format of the the `ALBankFile` structure, please refer to **"Audio File Format"** or the section on **"`libaudio.h`"** in the *Nintendo 64 Programming Manual*.

Sequence Playback

`alSeqpPlay()` or `alCSPPlay()` are used to start sequence playback. `alSeqpPlay()` is used for Type 0 MIDI sequence players, and `alCSPPlay()` is used for compact MIDI sequence players.

alSeqpPlay()

Syntax

```
#include <libaudio.h>
void alSeqpPlay(ALSeqPlayer *seqp);
```

Arguments

`seqp`	Pointer to the sequence player.

alCSPPlay()

Syntax

```
#include <libaudio.h>
void alCSPPlay(ALCSeqPlayer *seqp);
```

Arguments

`seqp`	Pointer to the compact MIDI sequence player.

Stopping a Sequence

`alSeqpStop()` or `alCSPStop()` is used to stop sequence playback. `alSeqpStop()` is used for Type 0 MIDI sequence players, and `alCSPStop()` is used for compact MIDI sequence players.

alSeqpStop()

Syntax

```
#include <libaudio.h>
void alSeqpStop(ALSeqPlayer *seqp);
```

Arguments

`seqp`	Pointer to the sequence player.

alCSPStop()

Syntax

```
#include <libaudio.h>
void alCSPStop(ALCSeqPlayer *seqp);
```

Arguments

seqp	Pointer to the compact MIDI sequence player.

Sequence Player Deletion

Once the sound player is no longer needed, the client registration of the synthesis driver can be deleted using `alSeqpDelete()` or `alCSPDelete()`.

alSeqpDelete()

Syntax

```
#include <libaudio.h>
void alSeqpDelete(ALSeqPlayer *seqp);
```

Arguments

`seqp`	Pointer to the MIDI sequence player structure to be deleted.

Before executing `alSeqpDelete()`, be sure to call `alSeqpStop` and confirm that no voices are being reproduced. `alSeqpDelete()` does not free up memory.

The following functions are used for compact MIDI players.

alCSPDelete()

Syntax

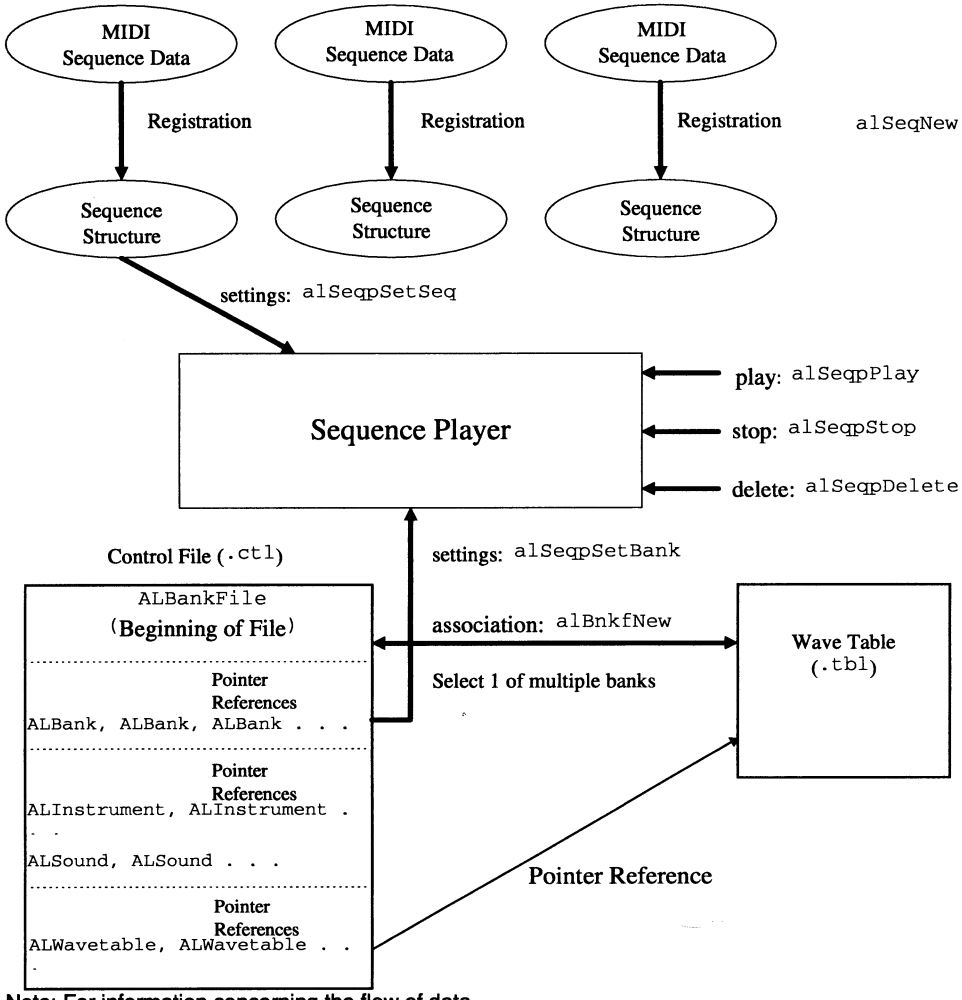
```
#include <libaudio.h>
void alCSPDelete(ALCSeqPlayer *seqp);
```

Arguments

`seqp`	Pointer to the compact MIDI sequence player structure to be deleted.

Before executing `alCSPDelete()`, be sure to call `alSeqpStop` and confirm that no voices are being reproduced. `alCSPDelete()` does not free up memory.

The following figure represents a summary of the flow of data during MIDI data playback and of the role of the functions. Although the sequence player used in this figure is a Type 0 player, the summary for compact MIDI players is the same.



Note: For information concerning the flow of data referencing for the bank file, refer to **"Audio File Format"** in the *Nintendo 64 Programming Manual.*

Figure 2. Data Flow During MIDI Sequence Reproduction

PLAYING SOUND EFFECTS

Sound effects are played using the sound player. The sound player and the sequence player are both 'clients' of the synthesizer.

To apply sound effects, the following procedure must be performed.

1. Read the audio bank from ROM.
2. Sound player startup.
3. Audio bank initialization.
4. Allocation of resources to sound.
5. Selection of sounds.
6. Sound playback.

In addition, the following steps are performed, as needed.

7. Stop sounds.
8. Delete sound player.

These elements are explained in the following paragraphs.

Reading the Audio Bank from ROM

To implement sound for audio, the wave table control file must first be moved to RAM using DMA transfer. DMA transfer is not described in detail here. Please refer to sample programs in `/usr/src/PR/`, such as `playseq`.

The following example is a typical DMA transfer of an audio bank.

1. Use `osCreatePiManager` to create a PI manager.
2. Use `osCreateMesgQueue` to create a message cue for confirming that a DMA transfer has been completed.
3. Use `alHeapAlloc` to reserve heap area for the audio bank.
4. Set the stage for DMA using `osWritebackDCacheAll` to perform a CPU cache writeback.
5. Use `osPiStartDma` to initiate DMA transfer.
6. Use `osRecvMesg` to confirm completion of the DMA transfer.

Sound Player Startup

For the sound player to be used, it must be started by `alSndpNew()`.

alSndpNew()

Syntax

```
#include <libaudio.h>
void alSndpNew(ALSndPlayer *sndp, ALSndpConfig *config);
```

Arguments

`sndp`	Pointer to the sound player structure to be initialized.
`config`	Pointer to the sound player configuration structure.

`alSndpNew()` configures according to the settings specified by `config` and initializes the sound player `sndp` so that it can be registered as a client.

Please exercise caution when using the audio heap specified in `config`.

Before `alSndpNew()` is called, the sound player structure must configured. The parameters of this structure are as follows.

maxSounds

This parameter defines the maximum number of sounds that can be allocated to the player.

maxEvents

This parameter defines the maximum number of internal events that can be supported.

heap

This is a pointer to the initialized audio heap. This allocates the heap area created using `alHeapInit()`.

An example of the sound player configuration structure is listed below.

```
#define MAX_VOICES    24
#define MAX_EVENTS    32

ALSndpConfig  SPConfig;

SPConfig.maxSounds    = MAX_VOICES;
SPConfig.maxEvents    = MAX_EVENTS;
SPConfig.heap         = &hp;
```

Audio Bank Initialization

The audio bank used for sound effect playback must be initialized. `alBnkfNew()` is used for this purpose.

alBnkfNew()

Syntax

```
#include <libaudio.h>
void alBnkfNew(ALBankFile *ctl, u8 *tbl);
```

Arguments

`ctl`	Pointer to the control data (.ctl).
`tbl`	Pointer to the wave table data (.tbl).

Two of the files created by the IC (instrument compiler) are used in sound playback. One of these is the control file (`.ctl`), which is responsible for handling instrument performance information. The other is the wave-table file (`.tbl`), responsible for handling wave-table data (waveform data).

The control file includes the offset address, which references the wave-table data. To improve runtime performance, the offset is converted to a virtual address by `alBnkfNew`.

In general, control data is transferred to DRAM at even an earlier stage than wave-table data. This is because wave-table data always turns out to be sizable and, to prevent wasteful use of DRAM, it is transferred to DRAM using DMA as needed. When the size of the wave-table file is small, this data is loaded into DRAM in advance to minimize DMA requests during sequence playback.

Allocation of Sound Resources

Sounds are allocated to the sound player using `alSndpAllocate()`.

alSndpAllocate()

Syntax

```
#include <libaudio.h>
ALSndId alSndpAllocate(ALSndPlayer *sndp, ALSound
*sound);
```

Arguments

`sndp`	Pointer to the sound player structure.
`sound`	Pointer to the sound structure.

Return Value

A sound ID of the same type as `ALSndId`.

Example

```
ALSndId      *sndid;
ALSndPlayer  sndp;
ALSound      *snd

for (i = 0, numAllocated = 0; i < inst->soundCound; i++){
     snd = inst->soundArray[i];
     if (sndid[i] = alSndpAllocate(&sndp, snd) != -1)
          numAllocated++;
}
```

In the previous example, `inst` is the `ALInstrument` structure included in the audio bank control data (`.ctl`). The `ALSound` structure referenced by `soundArray[i]` in that structure is allocated to the sound player as sounds. The number of sounds that are actually allocated is then substituted for `numAllocated`. `Sndid` is used before this and when sounds are played back or stopped.

`AlSndpAllocate` allocates the sound specified by `sound` to the sound player `sndp`.

If allocation is successful, `AlSndpAllocate` returns an ID of 0 or greater, of the same type as `ALSndId`. If the sound player cannot allocate a sound, -1 is returned.

Sound Selection

Before sounds are played and before settings such as pan and volume are configured, the sounds must be selected. `alSndpSetSound()` is used for this purpose.

alSndpSetSound()

Syntax

```
#include <libaudio.h>
void alSndqSetSound(ALSndPlayer *sndp, ALSndId id);
```

Arguments

`sndp`	Pointer to the sound player.
`id`	A unique value used to identify sounds in the sound player. `id` is returned when sounds are allocated to the sound player.

`id` uses the return value of the previously described `alSndpAllocate()`.

Sound Playback

`alSndpPlay()` is used to play sounds.

alSndpPlay()

Syntax

```
#include <libaudio.h>
void alSndpPlay(ALSndPlayer *sndp);
```

Arguments

`sndp`	Pointer to the sound player.

If no sounds are being played, calling this function has no effect.

Stopping Sounds

`alSndpStop()` is used to stop sounds.

alSndpStop()

Syntax

```
#include <libaudio.h>
void alSndpStop(ALSndPlayer *sndp);
```

Arguments

`sndp`	Pointer to the sound player.

If no sounds are being played, calling this function has no effect.

Sound Player Deletion

The synthesis driver client registration for sound players which are not needed can be deleted using `alSndpDelete()`.

alSndpDelete()

Syntax

```
#include <libaudio.h>
void alSndpDelete(ALSndPlayer *sndp);
```

Arguments

`seqp`	Pointer to the sound player to be deleted.

All sounds must be stopped before this function is called.

AUDIO STACK EXECUTION

This section describes the main points of audio stack execution.

It is assumed in this description that the scheduler is not used. When the scheduler is used, there are differences in some operations, such as SP execution.

> ***Note*** *When a game program becomes complex, using the scheduler is advised. For a information on how to use the scheduler, please refer to* ***"The Audio and Graphics Scheduler"*** *in the "Nintendo 64 Programming Manual", or the sample program* `simple`.

Buffer Preparation (triple buffer)

In general, three buffers (triple buffer) are prepared for sound processing. In addition, two command list buffers are prepared. These are alternately used for sample processing. This is done to ensure that sound is not interrupted due to processing speed.

For details, please refer to the sample programs.

Synchronizing Retrace Events and SP Events

In general, two message cues are prepared for audio task execution. One indicates retraces (V blank), while the other indicates the completion of SP tasks.

The first is generally set using `osViSetEvent`. The second can be implemented by first using `osSetEventMesg` and setting the type of event for `OS_EVENT_SP`.

Retrace events are essential to reliable frame-by-frame sample processing. For details, please refer to the sample programs.

Adjusting the Frame Size

In audio processing, it is not always appropriate to process the same sample quantity. Sample quantity must be regulated according to the circumstances for each instance.

This is generally accomplished using `IO_READ(AI_LEN_REG)` (or `osAiGetLength`) to detect the number of unused samples remaining in the audio buffer and adjusting the number of samples for processing accordingly.

In sample programs such as `playseq`, be sure that the constant `EXTRA_SAMPLES` is included in this processing. If this is absent, clicking sounds may be mixed in with music. The number of `EXTRA_SAMPLES` required varies depending on the program and sound. A suitable value must be determined through actual experience.

For more information concerning the regulation of frame size, please refer to the sample programs.

Creating the Audio Command List

`alAudioFrame()` is used to create the command list needed for sound synthesis by the RSP.

alAudioFrame()

Syntax

```
#include <libaudio.h>
Acmd *alAudioFrame(Acmd *cmdList, s32 *cmdLen, s16
*outBuf, s32 outLen);
```

Arguments

`cmdList`	Pointer indicating the start of the area where the audio command list is to be written.
`cmdLen`	Pointer indicating the length of the command list to be generated by `alAudioFrame`.
`outBuf`	Pointer indicating the location in DRAM where the waveform synthesized by the audio microcode is to be written when the audio command list generated by `alAudioFrame` is executed.
`outLen`	Number of stereo samples to be generated by the audio microcode.

Executing the Audio Task List

The RSP executes the audio task list and creates sound in the audio buffer. `osSpTaskStart()` is used for this purpose.

osSpTaskStart()

Syntax

```
#Include <ultra64.h>
s32 osSpTaskStart(OSTask *task);
```

Arguments

`task` Task structure.

The parameters for the task structure are listed below.

t.type

This parameter indicates the type of task. For audio tasks, set this to `M_AUDTASK`.

t.flags

This parameter indicates the type of task status bit. This is not needed for audio tasks.

t.ucode_boot

This is a pointer to the boot microcode. Initialize this to `rspbootTextStart`.

t.ucode_boot_size

This is the size of the boot microcode. Initialize this to `((u32)rspbootTextEnd - (u32)rspbootTextStart)`.

t.ucode

This is a pointer to the task microcode. For audio tasks, set this to `aspMainTextStart` or `n_aspMainTestStart`.

t.ucode_size

This is the size of the microcode. Initialize this to `SP_UCODE_SIZE`.

t.ucode_data

This is a pointer to the microcode data. For audio tasks, set this to `aspMainDataStart` or `n_aspMainDataStart`.

t.ucode_data_size

This is the size of the microcode data. Initialize this to `SP_UCODE_DATA_SIZE`.

t.dram_stack

This is a pointer to the DRAM matrix stack. For audio tasks, initialize this to 0.

t.dram_stack_size

This is the size of the DRAM matrix stack (bytes). For audio tasks, initialize this to 0.

t.output_buff

This is a pointer to the output buffer. For audio tasks, this can be ignored.

t.output_buff_size

This is a pointer for output buffer size. For audio tasks, this can be ignored.

t.data_ptr

This is a pointer for the SP command list. With an audio stack, this command list is generated by `alAudioFrame`.

t.data_size

This is the length of the SP command list expressed in bytes.

t.yield_data_ptr

This is a pointer to the buffer used for storing the state of yielding tasks. For audio tasks, this is always set to 0.

t.yield_data_size

This specifies the size of the yield buffer in bytes. For audio tasks, this is always set to 0.

Examples of task structure settings and `osSpTaskStart` are shown below.

```
OSTask *tlistp;

tlistp->t.type = M_AUDTASK;
tlistp->t.flags = 0;
tlistp->t.ucode_boot = (u64 *)rspbootTextStart;
tlistp->t.ucode_boot_size = ((u32)rspbootTestEnd -
(u32)rspbootTextStart);
tlistp->t.ucode = (u64 *)aspMainTextStart;
tlistp->t.ucode_size = SP_UCODE_SIZE;
tlistp->t.ucode_data = aspMainDataStart;
tlistp->t.ucode_data_size = SP_UCODE_DATA_SIZE;
tlistp->t.data_ptr = (u64 *)cmdList[curBuf];
tlistp->t.data_size = (cmdlp - cmdList[curBuf]) *
sizeof(Acmd);

osSpTaskStart(tlistp);
```

Audio DAC Settings

`osAiSetNextBuffer()` is used to set the next DMA transfer from RDRAM to the audio interface buffer for the AI buffer.

osAiSetNextBuffer()

Syntax

```
#include <ultra64.h>
s32 osAiSetNextBuffer(void *vaddr, u32 nbytes);
```

Arguments

`vaddr`	Buffer in RDRAM.
`nbytes`	Number of transfer bytes.

The buffer address `vaddr` must have a 64-bit boundary, and `nbytes` must be a multiple of 8 bytes. A maximum transfer size of 262144 bytes is supported. If the interface is busy (`AL_STATUS_FIFO_FULL` is set), `osAiSetNextBuffer` returns -1, and DMA fails.

USE OF AUDIO TOOLS

This section describes the audio tools included in the N64 OS and describes the overall flow of the sound development process.

SOUND DEVELOPMENT PROCESS

The tasks performed by musicians in developing sound are as follows.

Create the Wave Table

1. Sample sound generator data and store it in AIFF format. One loop can be included.
2. Create a code book for ADPCM encoding using the `tabledesign` tool.
3. Compress the sample in ADPCM AIFC format using the `vadpcm_enc` tool.
4. Create an `.inst` file (IC source file) using the `ic` tool.
5. Create bank files (`.tbl, .ctl, .symic`) using the `ic` tool.

Create Bank Files

6. Create a MIDI sequence.
7. If the MIDI file is Type 1, convert it to Type 0 using `midicvt`.
8. If necessary, convert the file to a compressed MIDI sequence (compact MIDI file) using `midicomp`.
9. Create the sequence bank (`.sbk`) using the `sbc` tool.
10. Pass the `.tbl, .ctl, and .sbk` files to the program.

The following paragraphs describe these tools as related to the sound development process.

TOOLS FOR CREATING WAVE TABLES

Tabledesign Tool

This tool creates the code book (table of prediction coefficients) needed for ADPCM compression from AIFF-C(AIFC) or AIFF formatted sample data. The code book is necessary for the ADPCM encoder (`vadcpm_enc`) to optimize sound quality.

The tabledesign tool creates the code book based on a corresponding-type cluster algorithm.

`tabledesign` is called using the following command line specifications.

```
tabledesign [-s book_size] [-f frame_size] [-i
refine_iter] aifcfile
```

`aifcfile` is the AIFC or AIFF file from which the code book is to be created.

The options which can be used in the command line are described below.

-s <value>

The number of code book entries is specified in the `value` position as a log base 2 number. Up to 8 entries are currently supported. Thus, the value of -s can be 0-3 $\log_2 8 = 3$. The default value is 2, or 4 entries. This number of entries is sufficient for nearly all sounds.

-f <value>

The frame size (number of samples) used for the prediction estimate is specifed in the `value` position. The ADPCM encoder uses a frame consisting of 16 samples. Thus, this number must be a multiple of 16. The default value is 16. By increasing the frame size, the creation time for the table can be shortened.

-i <value>

The number of iterations for small-step execution of the clustering algorithm is specified in the `value` position. The default value is 2. Although increasing the value lengthens creation time, it can result in better sound quality. Two iterations are considered sufficient for nearly all sounds.

Re-direction is used to write values to the file in which the code book file is to be written.

Example: `tabledesign sample.aiff > sample.table`

"vadpcm_enc" Tool

Code books created using `tabledesign` are then compressed in ADPCM format. The `vadpcm_enc` tool is used for this purpose.

`vadpcm_enc` encodes AIFF-C(AIFC) or AIFF formatted files, creating a fully compressed binary file. This encoding employs a conversion ADPCM algorithm to define a prediction coefficient table for the code book.

During coding, `vadpcm_encod` selects coefficients from the table to produce the best sound quality. The N64 compressed format currently supports only one loop point. This loop point must be defined in the instrument chunk of the input file (AIFC, AIFF).

When creating an AIFF file, it is probably best to keep in mind that only one sustain loop can be used.

`vacpcm_enc` is called using the following command line input.

```
vadpcm_enc -c codebook [-t] [-l minLoopLength] aifcFile
codedFile
```

`codebook` is the code book file created using `tabledesign`. `AifcFile` is the AIFC or AIFF file to be compressed. `codedFile` specifies the ADPCM-compressed output file.

The options that can be used in the command line are explained below.

-t

Specifying the `-t` option causes file encoding to end after loop termination. The portion after sound loop termination is not used in audio playback.

-l <value>

This option sets the minimum loop length for the encoded file. The default value is 800 samples.

Compression efficiency for the wave table varies with loop length. The longer the loop, the greater the efficiency. As indicated above, the length of the synthesized loop can be set in the ADPCM encoder (`vadpcm_enc`) using the `-l` option. The default minimum loop length is currently defined as 800 samples. This value can be changed. However, because the absolute minimum value is 16 (sample data), the setting must be a multiple of 16. If a length less than the minimum length is specified, the loop is iterated until the total length exceeds the minimum. If possible, please specify a length that is 1 frame longer than the audio frame. The length of one audio frame is equal to (sample rate)/(frame rate).

Example: `vadpcm_enc -c sample.table sample.aiff sample.aifc`

The `ic` (Instrument Compiler)

Once an ADPCM-compressed AIFC file has been created, a wave table can be prepared.

The N64 audio library synthesizes audio files from MIDI files, based on information written to `.ctl` and `.tbl` files. Together with the .sym file, these files are collectively called the "bank files." The `ic` tool is used to create bank files.

The content of each file is as follows.

`.tbl`

This is the file into which wave table data, including ADPCM-compressed sample data, is written.

.ctl

This is the file into which information concerning the method of wave table synthesis is written. This information includes the wave table's envelope, pan, pitch, mapping to the MIDI note number, and velocity. Also included are the offset addresses of actual ADPCM-compressed waveforms contained in the `.tbl` file. For details, please refer to **"The Audio File Format"** in the *Nintendo 64 Programming Manual.*

.sym

This designates the file into which bank file symbolic information is written. This file is used mainly for development and debugging. The `.sym` file is used only by the audio bank tool. It is not used by the audio library.

> ***Note:*** *`ic` can also condense SE to a single bank structure in ROM. In this case, several bank format functions (e.g. keymap, instrument parameters) cannot be used.*

`ic` is called using the following command line input.

```
ic [-v][-c cmpfile][-s midifile][-p][-n] -o <output file
prefix> <source file>
```

The options that can be used in the command line are explained below.

-v

This option turns on verbose mode. As a result, the compiler compiles a large amount of non-essential information. This option is used to obtain information concerning creation of the bank file.

-o <output file prefix>

This option specifies prefixes (the name to be added to the beginning of each extension) for the `.ctl`, `.tbl`, and `.sym` files.

<source file>

This is the name of the file containing the instrument bank source code. The following is a description of this file.

-c <compact sequence bank file>

This analyzes the compact sequence bank file (sequence bank file made from the compact MIDI file) and determines which sounds and objects are used. Unused objects are removed from the output bank. The compact sequence bank file is created using `midicomp` and `sbc`.

-s <midi sequence bank file>

This analyzes the sequence bank file and determines which sounds and objects are used. Unused objects are removed from the output bank. The sequence bank file (if a standard MIDI file is not Type 0) is created using `midicvt` and `sbc`.

-p

This outputs tables related to all programs and displays whether keys are used. A value of 0 indicates no key is used. A value of 1 indicates that a key is used. This option is valid only when used with `-c` or `-s`.

-n

This option does not produce sound according to the keymap. The `-n` option is usually used when the sound-effect bank is compiled. This is because the game program must reference sound according to the contents of the `.inst` file. Conversely, with a MIDI bank, sound must be produced using the keymap. In that case, mapping is performed by matching the MIDI note number with corresponding sound.

Creating the `ic` Source File

The source file for the instrument compiler individually defines each object that makes up the bank, using a language that resembles C. The source file combines these elements into a single object. Composing the bank requires objects that represent the bank, instrument, sound, keymap, and envelope. These objects are described in the following paragraphs.

Bank Object

The bank object begins with the keyword `bank`. In addition to instrument array and sample rate specifications, a default percussion instrument can optionally be specified for this object. The following example shows a bank object named `GenMidiBank`, which has one instrument named `GrandPiano` at instrument location 0. This bank is played at 44.1 KHz.

```
bank GenMidiBank
{
        sampleRate      = 44100;
        instrument [0] = GrandPiano;
}
```

Note: In the General MIDI 1.0 specifications, MIDI channel 10 is specified as the default channel for drums or other percussion instruments. Consequently, a program change message for channel 10 is usually not included in the General MIDI sequence. However, the following definition allows the default instrument (program) to be specified for channel 10.

```
bank GenMidiBank
{
        sampleRate              = 44100;
        percussionDefault       = Standard_kit;
        instrument [0]          = GrandPiano;
}
```

The sequence player specifies `Standard_Kit` as the default instrument for channel 10.

Instrument Object

The instrument object specifies the instrument's total volume and pan, and the sound list that makes up the instrument. It is referenced by the object bank.

The following example defines the instrument object `GrandPiano` which contains eight sounds (`GrandPiano00`, `GrandPiano01`, `GrandPiano02`, `GrandPiano03`, `GrandPiano04`, `GrandPiano05`, `GrandPiano06`, and `GrandPiano07`).

The total volume instrument is 127, and its central pan location is specified as 64.

```
instrument GrandPiano
{
        volume = 127;
        pan = 64;

        sound = GrandPiano00;
        sound = GrandPiano01;
        sound = GrandPiano02;
        sound = GrandPiano03;
        sound = GrandPiano04;
        sound = GrandPiano05;
        sound = GrandPiano06;
        sound = GrandPiano07;
}
```

Sound Object

The sound object is used to specify the sound volume and pan, keyboard mapping, and the envelope. In addition, an ADPCM AIFC sound file containing an ADPCM-compressed wave table can be specified in this object. For information on the ADPCM AIFC file (created using the ADPCM encoding tool) format that can be used with the `ic`, please refer to **"The Audio File Format"** in the *Nintendo 64 Programming Manual.*

> *Note: The sequence player computes the total volume by multiplying the instrument and sound volumes. It computes the total pan by adding the instrument and sequence values.*

The next example defines the sound object `GrandPiano00`. This sound receives wave table data from the file `../sounds/GMPiano_C2.18k.aifc` and uses a pan setting at the center (64) of full volume (127). The keyboard is arranged according to map specified by `piano00key`. The object uses the envelope specified by `GrandPianoEnv`.

```
sound GrandPiano00
{
        use ("../sounds/GMPiano_C2.18k.aifc");
        pan             = 64;
        volume          = 127;
        keymap          = piano00key;
        envelope        = GrandPianoEnv;
}
```

Refer to the following section for a detailed description of the keymap and envelope.

> *Note: When sound effects are arranged using multiple banks, a `keymap` entry is not needed.*

Keymap Object

The keymap object specifies the MIDI velocity and the number of keys to use for its sounds. It is referenced by the sound object. Based on this object, the sequence player determines the sounds to map to designated MIDI note numbers and the pitch of these sounds.

The following example defines the keymap object `piano00key`. This keymap object specifies MIDI Note On, a velocity range of 0-127, and note numbers of 0-43.

Because 41 is specified as `KeyBase` in this example, the sounds referenced by this keymap are generated at original pitch when the MIDI Note On message for this key is issued. If the MIDI Note On message for key 42 is issued, the pitch is shifted upward by a half tone and the same sound is generated.

> *Note: Values outside the range defined by `KeyMin` and `KeyMax` can also be specified for `KeyBase`. When it is desirable to conserve as much ROM space as possible, adjust the amount used by re-sampling the wave table and specify a value outside the above range for `KeyBase`. For example, re-sampling a 44.1 KHz wave table at 22.05 KHz and raising `KeyBase` one octave can compensate for sample rate differences. It must be noted, however, that sound quality will deteriorate if the rate of the pitch-shift is too high.*

In the following definition, `detune` is the parameter that specifies the number of cents added to the default tuning. A half-note corresponds to 100 cents.

```
keymap piano00key
{
        velocityMin     = 0;
        velocityMax     = 127;
        keyMin          = 0;
        keyMax          = 43;
        keyBase         = 41;
        detune          = 0;
}
```

Envelope Object

The envelope object specifies the ADSR envelope contour of the sound. Specify a volume of 0-127 and a time in microseconds.

The following example of an entire bank definition includes an envelope object with the following time and volume settings.

Attack:	0 microseconds and a volume of 0-127
Decay:	400 milliseconds and a volume to 0
Release:	200 milliseconds and volume to 0

The envelope also includes a MIDI Note Off message (sustain). The decay portion of this envelope sets decay to 0. This envelope provides the most realistic sounds for many acoustic instruments, particularly percussion instruments.

> *Note: Use of the sound player envelope has changed in a number of ways. For details, please refer to* ***"The Audio Library"*** *in the Nintendo 64 Programming Manual.*

Example: Definition of an Entire Bank

The following example defines an entire bank. This is the `General MIDI` bank provided with the N64 OS/Library. It defines one instrument, the `GrandPiano`.

```
envelope GrandPianoEnv
{
        attackTime      = 0;
        attackVolume    = 127;
        decayTime       = 4000000;
        decayVolume     = 0;
        releaseTime     = 200000;
        releaseVolume   = 0;
}

keymap piano00key
{
        velocityMin     = 0;
        velocityMax     = 127;
        keyMin          = 0;
        keyMax          = 41;
        keyBase         = 51;
        detune          = 0;
}

sound GrandPiano00
{
        use ("./sounds/GMPiano_C2.18k.aifc");
        pan             = 64;
        volume          = 127;
        keymap          = piano00key;
        envelope        = GrandPianoEnv;
}

keymap piano01key
{
        velocityMin     = 0;
        velocityMax     = 127;
        keyMin          = 42;
        keyMax          = 49;
        keyBase         = 63;
        detune          = 0;
}
```

```
sound GrandPiano01
{
        use ("./sounds/GMPiano_Bb2.16k.aifc");
        pan             = 64;
        volume          = 127;
        keymap          = piano01key;
        envelope        = GrandPianoEnv;
}

keymap piano02key
{
        velocityMin     = 0;
        velocityMax     = 127;
        keyMin          = 50;
        keyMax          = 57;
        keyBase         = 67;
        detune          = 0;
}

sound GrandPiano02
{
        use ("./sounds/GMPiano_F3.19k.aifc");
        pan             = 64;
        volume          = 127;
        keymap          = piano02key;
        envelope        = GrandPianoEnv;
}

keymap piano03key
{
        velocityMin     = 0;
        velocityMax     = 127;
        keyMin          = 58;
        keyMax          = 63;
        keyBase         = 72;
        detune          = 0;
}

sound GrandPiano03
{
        use ("./sounds/GMPiano_C4.22k.aifc");
        pan             = 64;
        volume          = 127;
        keymap          = piano03key;
        envelope        = GrandPianoEnv;
}
```

```
keymap piano04key
{
        velocityMin     = 0;
        velocityMax     = 127;
        keyMin          = 64;
        keyMax          = 69;
        keyBase         = 79;
        detune          = 0;
}

sound GrandPiano04
{
        use ("./sounds/GMPiano_G4.22k.aifc");
        pan             = 64;
        volume          = 127;
        keymap          = piano04key;
        envelope        = GrandPianoEnv;
}

keymap piano05key
{
        velocityMin     = 0;
        velocityMax     = 127;
        keyMin          = 70;
        keyMax          = 75;
        keyBase         = 84;
        detune          = 0;
}

sound GrandPiano05
{
        use ("./sounds/GMPiano_C5.22k.aifc");
        pan             = 64;
        volume          = 127;
        keymap          = piano05key;
        envelope        = GrandPianoEnv;
}

keymap piano06key
{
        velocityMin     = 0;
        velocityMax     = 127;
        keyMin          = 76;
        keyMax          = 81;
        keyBase         = 91;
        detune          = 0;
}
```

```
sound GrandPiano06
{
        use ("./sounds/GMPiano_G5.22k.aifc");
        pan             = 64;
        volume          = 127;
        keymap          = piano06key;
        envelope        = GrandPianoEnv;
}

keymap piano07key
{
        velocityMin     = 0;
        velocityMax     = 127;
        keyMin          = 82;
        keyMax          = 111;
        keyBase         = 99;
        detune          = 0;
}

sound GrandPiano07
{
        use("./sounds/GMPiano_C6.18k.aifc");
        pan             = 64;
        volume          = 127;
        keymap          = piano07key;
        envelope        = GrandPianoEnv;
}

instrument GrandPiano
{
        volume = 127;
        pan    = 64;

        sound = GrandPiano00;
        sound = GrandPiano01;
        sound = GrandPiano02;
        sound = GrandPiano03;
        sound = GrandPiano04;
        sound = GrandPiano05;
        sound = GrandPiano06;
        sound = GrandPiano07;
}

bank GenMidiBank
{
        sampleRate      = 44100;
        instrument[0]   = GrandPiano;
}
```

Other Tools

The following are other types of tools used to create wave tables.

vadpcm_dec

`vadpcm_dec` decodes sound files encoded in N64 ADPCM format by `vadpcm_enc` and rewrites these in standard output format as raw monaural 16-bit samples.

To call `vadpcm_dec`, specify the following code in the command line.

```
vadpcm_dec [-l] codedFile
```

The following options can be used in the command line.

-l

When sound is looped, the loop continues until a key is pressed using standard input.

To actually apply sound, use re-direction to create a sample file containing the sample data, then run `playraw` (tool for playing raw sample data), as shown in the following example.

```
vadpcm_dec sample.aifc > sample.raw
playraw < sample.raw
```

Or use the pipe specification, as shown below.

```
vadpcm_dec sample.aifc | playraw
```

playraw

`playraw` receives files from standard output and plays these on SGI workstations as raw 16-bit monaural or stereo audio sample data. A stereo input file must alternately contain L/R sample data. By piping the output from the previously described `vadpcm_dec` to this tool, ADPCM compact files can be opened.

To call `playraw`, input the following on the command line.

```
playraw [-h] [-s] [-f rate] < infile
```

The following options can be used in the command line.

-h

This option displays the help functions.

-s

This option treats the input as L/R stereo sample data.

-f <value>

This option specifies the sample rate during playback. The default value is 44,100 Hz.

infile

This option provides the raw data from the ADPCM AIFC-formated data, decoded by the `vadpcm_dec` tool.

TOOLS FOR CREATING THE SEQUENCE BANK

The `midicvt` Tool

If the sequence data (standard MIDI file) created are saved as Type 1, the `midicvt` tool must first be used to convert them to Type 0. This is because the audio library can play only Type 0 standard MIDI files.

To call `midicvt`, enter the following in the command line.

```
midicvt [-v] [-s] [-o] <input file> <output file>
```

The following options can be used in the command line.

-v

This option turns on verbose mode.

-s

This option deletes all messages not used by the audio library.

-o

This option organizes MIDI events, in order.

input file

This option specifies the name of a Type 0 or Type 1 standard MIDI file.

output file

The name of the Type 0 file to be output is entered using this option.

Execution Example: `midicvt sample.mid sample.seq`

The `midicomp` Tool

To play a sequence using the compact sequence player, the sequence must be converted to compact MIDI format. `midicomp` is used for this purpose.

The `midicomp` tool converts Type 0 or Type 1 standard MIDI files to compact MIDI format.

To call `midicomp`, specify the following in the command line.

```
midicomp [-o] <input file> <output file>
```

The following options can be used in the command line.

-o

This option organizes MIDI events logically. This is particularly important when processing loops.

<input file>

This option specifies the name of a Type 0 or Type 1 standard MIDI file. This file is compressed.

<output file>

The name of the Type 0 file to be output is entered using this option.

The data compression ratio for compressed files varies with file content. Except for very small files, all files are compressed to some degree. The `midicomp` tool compresses by recognizing patterns. Therefore, the greater the number of iterations in a file, the higher the data compression ratio. If numerous sections in a sequence are created using the copy and paste functions, the data compression ratio becomes very high.

Execution Example: `midicomp -o sample.seq sample.cmp`

The `sbc` Tool

Sequence data is arranged in the MIDI sequence bank. `sbc` is used for this task. `sbc` gathers any number of MIDI sequences into a single MIDI sequence bank (`.sbk` bank). The `sbc` tool attaches a header to the beginning of each sequence and lines each one up in 8-byte units, then writes them to the `.sbk` file in succession. The header at the beginning of each sequence serves as an index when a sequence search is performed.

To call `sbc`, input the following at the command line.

```
sbc [-o <output file>] file0 [file1 file2 file3 ...]
```

The following options can be used in the command line.

-o <output file>

This option specifies the name of the sequence bank file to be output. The default output is `tst.sbk`.

file0 file1 file2 ...

This option specifies the names of the sequence files contained in the sequence bank.

Execution example: `sbc -o sample.sbk sample0.seq sample1.seq sample2.seq`

Other Tools

The following are other tools used to create the sequence bank.

midiprint

The `midiprint` tool prints, as a text list, the time-based events written in Type 0 or Type 1 standard MIDI files.

To call `midiprint`, use the following command line input.

```
midiprint [-v] [-o <output file>] <input file>
```

The following options can be used in the command line.

-v

This option turns on verbose mode.

-o <output file>

This option specifies the output file using the option for MIDI event text.

<input file>

This option specifies a Type 0 or Type 1 standard MIDI file to be passed to the list.

Execution example: `midiprint sample.seq > out.txt`

midistat

The `midistat` tool displays the key numbers and MIDI channels contained in a MIDI sequence file, along with the number of notes for each MIDI program number contained in the file.

To call `midistat`, use the following command line input.

```
midistat [-v] <MIDI file>
```

The following options can be used in the command line.

-v

This option turns on verbose mode.

<MIDI file>

This option specifies the name of a Type 0 or Type 1 standard MIDI file.

Execution example: `midistat sample.seq > out.txt`

PROGRAMMING CAUTIONS

This section discusses items that the user must keep in mind when developing audio for the N64.

COMMON VALUES

Throughout this section, the following three types of values are considered fixed for files such as .inst and MIDI files.

1. C4 indicates middle C (MIDI note 60).
2. The range of pan values is 0-127. 0 corresponds to the left extreme, 64 to the center, and 127 to the right extreme.
3. The range of volume values is 0-127. 0 represents no sound and 127 represents maximum volume.

MANAGING RESTRICTIONS AND ALLOCATING RESOURCES

When the N64 is used, several choices must be made. Nearly all of these choices are made to minimize the resources used while maintaining acceptable quality. If the necessary resources can be used without restriction, the N64 can create fantastic audio.

Nearly all software restrictions can be easily overcome. However, the amount of time required for a development project can be greatly reduced if settings and values are decided upon in advance through discussions between the programmer, game designer, and musician.

Resource restrictions can be classified into the following four categories.

- Determining hardware playback rates
- Limitations on voice number and processing time
- Sound and music bank partitioning
- ROM space limitations

These categories are addressed in the following paragraphs.

Determining Hardware Playback Rates

The most important of the software-related decisions is what settings should be used for the hardware playback rate. Standard choices range from 22.05 KHz to 44.1 KHz. Selecting a high rate increases the number of samples created by the software. As a result, RSP processing time increases.

There are no rigid restrictions on hardware playback rates. Although the rate for music CDs is 44.1 KHz, large losses in sound quality do not occur even at 32 KHz or 22 KHz. The N64 analog filter is designed based upon a rate of 32 KHz. Consequently, at rates above 32 KHz, there is an extreme decrease in treble resulting from filtering. In other words, when rates higher than this are used, there is little noticeable change. However, at rates less than 22.05 KHz, the sampling size becomes pronounced and sound quality suddenly begins to decline.

In addition, if output and sample rates are brought as close together as possible, the sample sound quality is good. When game sounds are sampled at 22.05 KHz, a playback rate of 22.05 KHz results in the best output quality. If playback rates cannot be decided upon in the planning stage, it is probably better to first perform sampling tasks at a higher rate and convert to a lower rate, rather than the alternative.

Limitations on Voice and Processing Time

The number of voices that can be used in playback is limited by the processing time available for audio. The greater the number of voices and the higher the audio playback rate, the longer the processing time required. Although it is not possible to calculate exactly the time required, the approximate time required per voice with a playback rate of 44.1 KHz is estimated to be 1% of RSP time. Therefore, assuming that 20% of RSP processing time is provided for audio, 15-20 voices can be used. Assuming that 40% of RSP processing time is given to audio increases this to 30-40 voices. It must also be kept in mind that processing time shortens as the output playback rate is decreased, increasing the number of voices that can be used for playback.

Sound and Music (BGM) Bank Partitioning

There are no formal rules regulating sound and BGM composition. However, in nearly all cases, except for BGM sample data, it is advisable to construct one or more banks.

Sequences can be stored in a game using two methods. Individual sequences can be stored as independent sequences. Or multiple sequences can be compiled and stored as a `.sbk` file. Please ensure that BGM sample data and MIDI files construction is such that each MIDI file sequence or bank (only at time of use) corresponds to a bank of BGM sample data. Store all sample data shared by multiple MIDI files in the same bank. If these files are not stored in the same bank, copies of sample data will be created in several bank files.

ROM Space Limitations

This problem should be considered by the game developer.

CREATING SAMPLE DATA

The method for creating N64 sample data is essentially the same as that for creating samples used for sound players in general. However, attention should be given to several points in this process.

Samples that the ADPCM tool can recognize are monaural samples of AIFF files or uncompressed AIFC files.

If the sampling rate is the same as the output rate, sample data quality is high. Because all sample data is compressed using ADPCM, playing sample data at a rate which is appreciably different from the sampling rate increases the possibility that undesirable noise will become pronounced.

For example, if data which is sampled at 22.05 KHz and played at an output rate of 44.1 KHz using the original pitch, the sample converter must create two elements of data from each input element. Moreover, if these samples are played at a pitch even one octave lower than the original pitch, the sample converter must create four samples from each input sample. Because of noise and distortion caused by ADPCM, the quality of samples deteriorates to a point where output quality does not compare with input quality when sampled at 44.1 KHz or output at 22.05 KHz. In such cases, it is recommended that all sampling be performed at the output sample rate before ADPCM conversion.

Loops can be used at any point in sample data. In many ADPCM systems, sample data must be looped at specific boundaries. When using the Super NES for example, the loop point must be a multiple of 16. The N64, however, has no restrictions on loop placement. If a sound is looped, it will continue to loop as long as playback continues. Even if the envelope enters the release stage, the sound will continue to loop as before.

Even after completion of a loop, all loop samples are played until the next multiple of 16 is reached. This is because the ADPCM encoder considers 16 to be one block when storing sample data. Consequently, it is recommended that at least 16 items of sample data be left after the end of the loop for all loop samples. The ADPCM tool includes an option to shorten sample data to the minimum variable length.

In other words, it is sufficient only to decide on loop location placement when creating looping sample data. It is not necessary to consider sample data release points. Reducing the volume of sample data to store on a hard disk by shortening the sample data poses no problems. However, please remember to leave at least 16 items of sample data after the end of a loop. When the sample is later encoded, specifying the `-t` option will cause the system to shorten the sample automatically.

PLAYBACK PARAMETERS AND THE INSTALL (.INST) FILE

The following paragraphs describe creation of the `.inst` file.

Sample Parameter Settings for the Install File

To accurately play sample data, the N64 audio system must provide control element information, such as pitch and volume. These parameters are set by creating and editing the `.inst` file. The playback parameters are described in detail below. First, however, an example of an `.inst` file will be examined.

The `.inst` file is a collection of objects defined by text that uses C language syntax. These objects are of the following types.

- Envelopes
- Keymaps
- Sounds
- Instruments
- Banks

These objects are related in the following ways. The basic unit of sample audio is a sound. Corresponding to the sound are keymaps that specify a sample's velocity ranges, key ranges, and tuning . Similarly, envelope ADSR parameters are also associated with the sound for control of sample volume.

Several sounds can be collected to create an instrument, and several instruments can be collected to create a bank. Because program control change is restricted to values of 1-128, the MIDI sequence can use only the first 128 instruments in a bank. Values of 129 or greater can be selected if the audio API is called from a game application.

Use of the Install File

The sound player and sequence player both use bank files created from `.inst` files. However, the ways in which they use the `.inst` file are completely different. The sequence player uses the bank to discriminate between instruments. It uses the keymap to identify which sound (waveform) to play in correspondence with a MIDI note. In contrast, the sound player does not use the bank structure, instrument structure, or keymap parameters. To enable the sound player to be compiled, however, each `.inst` file requires a bank and instrument. In addition, each sound must indicate a keymap. This keymap can be shared by all sounds in an `.inst` file. Therefore, only one keymap is required.

For these reasons, an `.inst` SE file, for example, consists of one bank and one instrument that describes the sounds in succession. Because the sound player ignores keymaps, overlapping them is not necessary. However, the `.inst` file does contain one default keymap for file compilation. When setting SE pitch, the setting is made from the application rather than by the `.inst` file, except for the rate at the time the sample was recorded.

Envelopes

For volume control, the N64 audio system supports the ADSR envelope function. Envelope time is expressed in microseconds. Because a microsecond is much shorter than the control unit of typical synthesizers and samplers, musicians must adjust their concept of time so that they can handle values even larger than those commonly used by samplers.

Please remember, for example, that an Attack Time setting of 100,000 corresponds to 1/10 of 1 second.) The maximum volume value is 127. To prevent popping and clicking noises from occurring at the end of a sound, always end the envelope with a release volume setting of 0. This becomes an important point when looping a sample.

When the sound player is used to apply sound to sound effects, the envelope will not enter the release stage if the delay time is set to 0 (it will loop infinitely). The game must call `alSndpStop()` to stop the sound.

Keymap

Note: Only the sequence player uses keymaps. The sound player ignores them.

In addition to an envelope, all samples have keymaps. The keymap defines to which keys and velocities the sample responds. Through various keymap changes, instruments can be created that play a variety of samples using various keys and velocities.

The keymap object sets the upper and lower limits of the range for sample response velocities and the upper and lower limits for response keys. Please be careful to ensure that keymap ranges are not overlapped. When the sequence player maps a note to be played, it searches for a usable keymap and stops searching at the first usable keymap found.

Note: The N64 sets the value of the upper limit of `keyMax` one octave above `keyBase`.

Tuning Samples Recorded at the Hardware Playback Rate

In addition to velocity and key range information, parameters included in the keymap structure are `keyBase` and `detune`. `keyBase` sets the sample pitch in half-note (semitone) units. `detune` is used to fine tune samples in cent units (1/100 of a half-note). If the sample rate for a sound and the hardware playback rate are the same, `keyBase` is the same as the MIDI note number of the sample's original pitch. If the two rates are different, `keyBase` must be changed to compensate for the difference.

For example, if an F4 note is recorded and played at 44.1 kHz, `keyBase` is 65 (= MIDI note F4), and `detune` is 0.

Tuning Samples Recorded at a Rate Other than the Hardware Playback Rate

Handling the `.inst` file becomes somewhat complicated when tuning samples recorded at a rate different from the hardware output rate. (The hardware output rate is determined by software and can be changed.) Although the sample rate is automatically extracted from AIFF files, the user must set the `keyBase` and `detune` parameters manually, if samples are to be played at the correct pitch.

The formulas for computing the `keyBase` and `detune` parameters from a given sample rate are as follows.

```
N       =       semitones to add to keyBase
N       =       12 log2(hardware rate/sample rate)
```

To simplify the tuning problem, please use Table 1.

1. Select an appropriate rate from the column for the hardware rate used.
2. Record a sample at that rate (or resample a sample at that rate).
3. Add the semitone number in the leftmost column to the note number of the sample pitch. If this is done, `detune` always becomes 0.

For example, consider a case in which the hardware playback rate is 44.1 KHz and a sample of a trumpet playing a B4 note is re-sampled at approximately 32 KHz. Re-sampling will be performed at 33,037.67 Hz, the closest rate to 32,000 Hz. When this file is played at a rate of 44.1 KHz, the pitch that can actually be heard is the `keyBase` written in the `.inst` file. A look at the leftmost column of the table shows that 5 semitones should be added. The MIDI note number for a B4 note is 70. Adding 5 to this shows that a value of 75 should be assigned to `keyBase`.

Table 1 Hardware Playback Rates

Add to MIDI Value	Hardware Playback Rate (Hz) 44,100	Hardware Playback Rate (Hz) 32,000	Hardware Playback Rate (Hz) 22,050
0 semitones	44100	32000	22050
1 semitone	41624.857	30203.978	20812.429
2 semitones	39288.633	28508.759	19644.317
3 semitones	37083.532	26908.685	18541.766
4 semitones	35002.193	25398.417	17501.097
5 semitones	33037.671	23972.913	16518.836
6 semitones	31183.409	22627.417	15591.705
7 semitones	29433.219	21357.438	14716.609
8 semitones	27781.259	20158.737	13890.626
9 semitones	26222.017	19027.314	13111.008
10 semitones	24750.288	17959.393	12375.144
11 semitones	23361.161	16951.410	11680.581
12 semitones	22020	16000	11025

If the user wishes to expand Table 1 or calculate the value for hardware rates other than those given, please use the following formula.

$$S = \frac{H}{2^{N/12}}$$

Where:

S = Sample rate

H = Hardware rate

N = Added semitones

Sounds

The sound structure references samples, keymaps, envelopes, pan values, and volume values. The range of pan values is 0-127, with 0 corresponding to the left extreme, 64 the center, and 127 the right extreme. The range of volume values can also be specified as 0-127.

Instruments

Written in the instrument structure is a list of sounds grouped by instrument. If the instrument is an instrument that uses the sequence player, the number of sounds is limited to 128, the maximum number of MIDI notes. With an instrument that uses the sound player, however, several sounds or as many as desired, can be written. In addition to the sound list, all volume values and pan values can be written to the instrument structure. (The sound player ignores volume and pan values written in the instrument structure. Instead, it uses the pan and volume values specified in the sound object.)

The instrument structure can also be used to create a drum kit. In this case, an instrument that uses multiple sounds can be created, and keymaps can be applied to each sound. (Please refer to the General MIDI Bank of the development package.)

Banks

Located at the top level of the `.inst` file is the bank structure. Several banks can be kept in the `.inst` as needed. Currently, there are no means available to switch banks from MIDI, so banks must be selected by the application.

Making a Bank File

The procedure for creating a sample bank file is as follows.

1. Record the sample and save it as an AIFF file.
2. Compress the sample using `tabledesign` and `vadpcm_enc`.
3. Create an `.inst` file.
4. Compile a bank using `ic`.

MIDI FILES

All sequences can be stored in a game as in Type 0 MIDI file format or compact MIDI file format. When using the Type 0 MIDI file format, run the `midicvt` tool to save a file as a Type 0 or Type 1 MIDI file. When using the compact sequence format, run the `midicomp` tool to save a file as a Type 0 or Type 1 MIDI file.

The procedure for creating a MIDI sequence bank file is listed below.

1. Create the sequences and store them as Type 0 or Type 1 MIDI files.
2. Convert the sequences using `midicvt` or `midicomp`.
3. Compile the sequences using `sbc`.

The following MIDI messages are supported.

- Note On
- Note Off
- Polyphonic key pressure
- MIDI controller:
  a) Controller 07: channel volume
  b) Controller 10: channel pan
  c) Controller 64: sustain
  d) Controller 91: FXMix
- Program controller change (0-127)
- Pitch-bend change

In addition to the above messages, MIDI file meta tempo events are also supported.

Loops in a Sequence

The means of creating a loop varies greatly with sequence format. When Type 0 MIDI format is adopted for use in a game, the program must create the loop by executing an audio library call from within the game code. When the compact sequence format is used, the musician inserts loops using MIDI controllers.

With the compact sequence format, loops can be created in tracks. A maximum of 128 sequential or nested loops can be created for a single track. Each loop must be allocated a number, and its beginning and end points must be specified. In addition, a loop count (number of times the loop section is played) can be arbitrarily specified. A value of 1-255 can be specified for the loop count. If 0 (default) is specified, the section will loop endlessly.

Although the compact MIDI file format will not be explained in detail here, it should be noted that, if a file is compressed, attention must be given to the fact that MIDI events are repositioned by the channel base. That is, MIDI events are repositioned so that all channel 1 MIDI events are placed in track 1, all channel 2 events are placed in track 2, and so forth. When creating loops, please be particularly mindful of this point. If a loop is placed in a track, all MIDI events in that channel will loop.

Loops are inserted in compact MIDI sequences using controllers. These controllers serve as loop markers. Controller 102 defines the start of a loop; controller 103 defines the end of the loop.

Each loop start/end pair in a single channel must be assigned a unique number between 0 and 127. Please set these numbers as controller values for the start and end of a loop.

Define loop counts of 0 to 127 by setting a value of 0-127 for controller 104. Define loop counts 128 to 255 by setting a value of 0-127 for controller 105. (When the system encounters loop count controller 105, it adds its value to 128 to compute loop counts of 128 to 255.)

A simple sequence is shown in the following example.

```
loop 0 start          (controller 102, value: 0)
loop count of 6       (controller 104, value: 6)
loop 0 end            (controller 103, value: 0)
```

In the above example, the section between the loop start and end is played 6 times.

An important point to note is that there is no correspondence between loop count and start/end pairs. If the end of a loop is reached, the system will use the most recent loop count setting and continue looping, regardless of whether the end of the preceding loop was passed. As an example of this, consider the following sequence.

```
loop 0 start          (controller 102, value: 0)
loop count of 8       (controller 104, value: 8)
loop 0 end            (controller 103, value: 0)
loop 1 start          (controller 102, value: 1)
loop 1 end            (controller 103, value: 1)
```

In this case, the loop count of the first loop (loop 0) is set to 8. Once set, however, the loop count will retain the same setting until the user explicitly changes it. Thus, if left as it is, the loop count for the second loop (loop 1) will be 8.

If the user does not define the loop count in a sequence, the system will set the count to the default value of 0 and loop that section endlessly.

Note: All loops must have a start and an end. At least 1 valid MIDI event must be included between them.

Nested Loops

Loops can easily be nested in the compact sequence format. The following example shows a sequence with nested loops.

```
loop 0 start          (controller 102, value: 0)
loop 1 start          (controller 102, value: 1)
loop count of 8       (controller 104, value: 8)
loop 1 end            (controller 103, value: 1)
loop 2 start          (controller 102, value: 2)
loop 2 end            (controller 103, value: 2)
loop 3 start          (controller 102, value: 3)
loop count of 4       (controller 104, value: 4)
loop 3 end            (controller 103, value: 3)
loop forever          (controller 104, value: 0)
loop 0 end            (controller 103, value: 0)
```

In this case, loop 1 loops 8 times. Then loop 2 starts and it also loops 8 times. Loop 3 starts and loops 4 times. The entire sequence loops endlessly.

Creating a Compact MIDI File that Contains Loops

When creating a compact MIDI format with loops using the `midicomp` tool, remember that the `-o` option must be used. This option organizes MIDI events in a logical order. Without this, problems arise, such as audio at loop points sounding flat or sound becoming inaudible.

Problems Related to Loops

When constructing a loop using a Type 0 sequence player, note that the Note On which occurs just before the return to the top of the loop remains in effect.

For example, when a sound for which Note On occurs at the top of the 16th bar is changed to Note Off at the top of the 18th bar, and if the sequence is set so that at the top of the 17th bar there is a return to the top of the 2nd bar; this sound will not be able to obtain a Note Off event and will consequently result in an "endless sound" state. If this sequence continues and approaches the 16th bar, a Note On event will again occur, requiring a new voice. The sequence will finally result in an "insufficient number of voices" status.

To avoid this situation, be sure to execute a Note Off prior to looping back to the point before the piece of music. Or, using the same key (height), prepare a Note Off for the same channel before the loop return.

CREATING THE MAKEFILE

Batch processing using `make` is a convenient means of creating sample and sequence banks from wave files (AIFF) and MIDI files.

The protocols and file dependencies for these processes are written in `Makefile` in the N64 OS directory `/usr/src/PR/assets`. Please refer to this file when creating a new `Makefile`.

GENERAL MIDI AND N64

Although the N64 itself is not a General MIDI device, it can be configured like one. An example of the structure of a General MIDI bank is included in `/usr/lib/PR/soundbanks` in the IRIX version of the N64 OS/Library and in `\ultra\Usr\Lib\PR\soundbanks` in the PC version of the N64 OS/Library. All of the sound files used by this bank are provided, enabling license-holding developers to use these in any project.

The current default setting for MIDI channel 10 is program 128. In the General MIDI bank, this is the setting for the standard drum kit. The standard program can be changed to a note other than that of the standard drum kit if the user sends a program change for channel 10.

Nintendo

NINTENDO 64

Audio

Development Guide