QuickTime Music Architecture

for Macintosh and Windows

Apple Technical Publications

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Printed in the United States of America.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF DISTRIBUTION OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS DISTRIBUTED "AS IS," AND YOU ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

About This Book

This book is a programmer's guide to the music architecture in version 3 of QuickTime for Macintosh and Windows. It describes all the features introduced, added, or changed in the QuickTime music architecture since QuickTime version 1.5, and therefore supersedes all existing documentation for versions 1.6.1, 2.0, 2.1, and 2.5.

Book Structure

Chapter 1 begins with an overview of the new features in the QuickTime music architecture (QTMA), introducing you to its basic concepts. Some programming examples are also provided. Chapter 2 offers a QuickTime music architecture reference, listing all the constants, data types, and functions in the QuickTime 3 QTMA. Appendix A is a General MIDI reference with tables listing General MIDI instrument numbers and General MIDI drum kit numbers.

Conventions Used in This Book

This book provides various conventions to present information. Words that require special treatment appear in specific fonts or font styles.

Special Fonts

All code listings, reserved words, and the names of actual data structures, constants, fields, parameters, and functions are shown in Letter Gothic `(this is Letter Gothic)`.

Types of Notes

There are several types of notes used in this book.

Note
A note like this contains information that is interesting but not essential to an understanding of the main text. ◆

IMPORTANT
A note like this contains especially important information that is essential for an understanding of the main text. ▲

▲ **WARNING**
A warning like this indicates potential problems that you should be aware of as you design your software. Failure to heed these warnings could result in system crashes or loss of data. ▲

Development Environment

The functions described in this book are available using C interfaces. How you access them depends on the development environment you are using.

Code listings in this book are shown in ANSI C. They suggest methods of using various functions and illustrate techniques for accomplishing particular tasks. Although most code listings have been compiled and tested, Apple Computer Inc., does not intend for you to use these code samples in your application.

QuickTime Music Architecture

This chapter describes the **QuickTime music architecture** (QTMA), which allows QuickTime movies, applications, and other software to play individual musical notes, sequences of notes, and a broad range of sounds from a variety of instruments and synthesizers. With QTMA, you can also import Standard MIDI files (SMF) and convert them into QuickTime movies for easy playback.

Because the QTMA is component-based and implemented as Component Manager components, your application can take advantage of a number of QTMA components for extensibility. For example, you can use the QuickTime music synthesizer, which is a software-based music synthesizer included with QuickTime, to generate sounds or music out of a computer's built-in audio device. You can also use the General MIDI component for playing music on a General MIDI device attached to a serial port.

Before reading this chapter, you should already be familiar with QuickTime and QuickTime components. In order to create or use any component, your application must use the Component Manager. If you are not familiar with the Component Manager, see Chapter 6 of *Inside Macintosh: More Macintosh Toolbox*.

You need to read this chapter if you are writing an application that creates QuickTime movies and you want to incorporate music tracks as part of the movie, either by importing MIDI files or by programmatically generating musical sequences. If you want to create a music component or add an instrument to the existing library of instruments, you also need to read this chapter. These capabilities are explained in the section "Using the QuickTime Music Architecture" (page 31). If you are creating new instruments, you should be familiar with QT atoms and atom containers, which are described in Chapter 1, "Movie Toolbox" in *QuickTime 3 Reference.*

Chapter 2 in this book contains an extensive reference section, which describes the constants, data types, and functions of the QTMA.

Introduction to QuickTime Music Architecture

The QuickTime music architecture is implemented as Component Manager components, which is the standard mechanism that QuickTime uses to provide extensibility.

QTMA components exist both in QuickTime for Macintosh and QuickTime for Windows. Note that in QuickTime 3 for Windows, MIDI output is not yet supported; only the QuickTime music synthesizer is available.

Different QTMA components are used by a QuickTime movie, depending on if you are playing music or sounds through the computer's built-in audio device, or if you are controlling, for example, a MIDI synthesizer. During playback of a QuickTime movie, the music media handler component isolates your application and the Movie Toolbox from the details of how to actually play a music track. The task of processing the data in a music track is taken care of for you by the media handler through Movie Toolbox calls.

The following sections provide overviews of these components and their capabilities.

Overview of QTMA Components

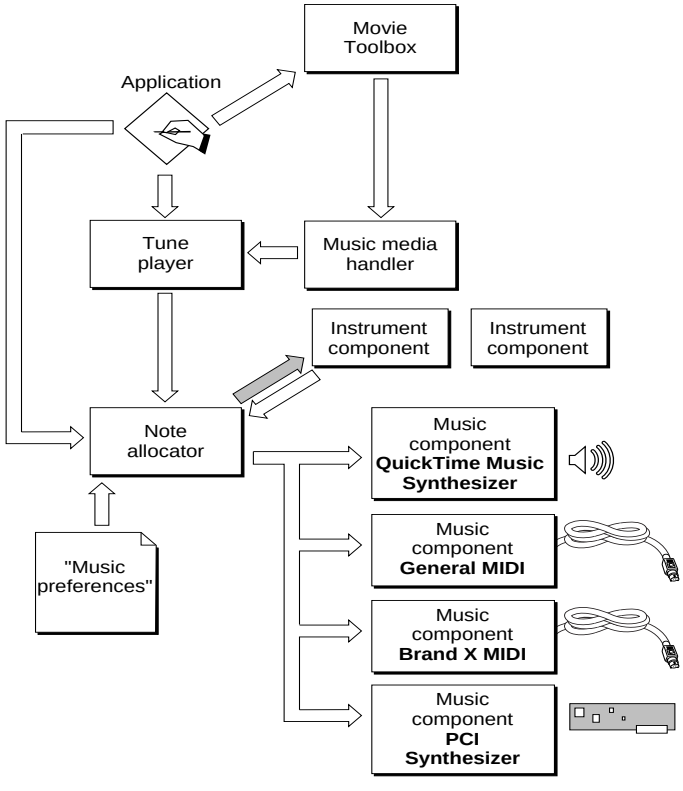
The QuickTime music architecture includes the following components:

- the note allocator, which plays individual musical notes
- the tune player, which plays sequences of musical notes
- the music media handler, which processes data in music tracks of QuickTime movies
- the QuickTime music synthesizer, a software-based music synthesizer included with QuickTime, which plays sounds using the built-in audio of a Macintosh or Mac OS–based computer or the sound card or built-in audio circuitry of other computers
- the General MIDI synthesizer, which plays music on a General MIDI device connected to the computer
- the MIDI synthesizer component, which controls a MIDI synthesizer connected to the computer using a single MIDI channel

- other music components that provide interfaces to specific synthesizers

These components are described in more detail in the following sections. Figure 1-1 illustrates the relationships among the various QTMA components.

Figure 1-1 How QuickTime music architecture components work together



Note Allocator Component

You use the **note allocator component** to play individual notes. Your application can specify which musical instrument sound to use and exactly

which music synthesizer to play the notes on. The note allocator component can also display an Instrument Picker, which allows the user to choose instruments. The note allocator, unlike the tune player, provides no timing-related features to manage a sequence of notes. Its features are similar to a music component, although more generalized. Typically, an application opens a connection to the note allocator, which in turn sends messages to the music component. An application or movie music track can incorporate any number of musical timbres or parts.

To play a single note, your application must open a connection to the note allocator component and call `NANewNoteChannel` with a note request—typically to request a standard instrument within the General MIDI library of instruments. A note channel is similar in some ways to a Sound Manager sound channel in that it needs to be created and disposed of, and can receive various commands. The note allocator provides an application-level interface for requesting note channels with particular attributes. The client specifies the desired polyphony and the desired tone. The note allocator returns a note channel that best satisfies the request.

With an open note channel, an application can call `NAPlayNote` while specifying the note's pitch and velocity. The note is played and continues to play until a second call to `NAPlayNote` is made specifying the same pitch but with a velocity of zero. The velocity of zero causes the note to stop. The note allocator functions let you play individual notes, apply a controller change, apply a knob change, select an instrument based on a required tone, and modify or change the instrument type on an existing note channel.

There are calls for registering and unregistering a music component. As part of registration, the MIDI connections, if applicable, are specified. There is also a call for querying the note allocator for registered music components, so that an application can offer a selection of the existing devices to the user.

Tune Player Component

The **tune player component** can accept entire sequences of musical notes and play them start to finish, asynchronously, with no further need for application intervention. It can also play portions of a sequence. An additional sequence or sequence section may be queued-up while one is currently being played. Queuing sequences provides a seamless way to transition between sections.

The tune player negotiates with the note allocator to determine which music component to use and allocates the necessary note channels. The tune player handles all aspects of timing, as defined by the sequence of **music events.** For

more information about music events and the event sequence that is required to produce music in a QuickTime movie track, see the section "About QuickTime Music Events" (page 19).

The tune player also provides services to set the volume and to stop and restart an active sequence.

Note
If your application simply wants to play background music, it may be easier to use the QuickTime Movie Toolbox, rather than call the tune player directly. ◆

Music Components Included in QuickTime

Individual music components act as device drivers for each type of synthesizer attached to a particular computer. Three music components are included in QuickTime:

- the QuickTime music synthesizer component, for playing music out of a computer's built-in speaker
- the General MIDI synthesizer component, for playing music on a General MIDI device attached to a serial port.
- the MIDI synthesizer component, which allows QuickTime to control a synthesizer that is connected to a single MIDI channel.

Developers can add other music components for specific hardware and software synthesizers. To better understand the role of a music component, see "The QuickTime Music Synthesizer Component" (page 16).

Applications do not usually call music components directly. Instead, the note allocator or tune player handles music component interactions. Music components are mainly of interest to application developers who want to access the low-level functionality of synthesizers and for developers of synthesizers (internal cards, MIDI devices, or software algorithms) who want to make the capabilities of their synthesizers available to QuickTime.

In order for an application to call a music component directly, you must first allocate a note channel and then use `NAGetNoteChannelInfo` and `NAGetRegisteredMusicDevice` to get the specific music component and part number.

You can use music component functions to

- obtain specific information about a synthesizer
- find an instrument that best fits a requested type of sound
- play a note with a specified pitch and volume
- change knob values to alter instrument sounds

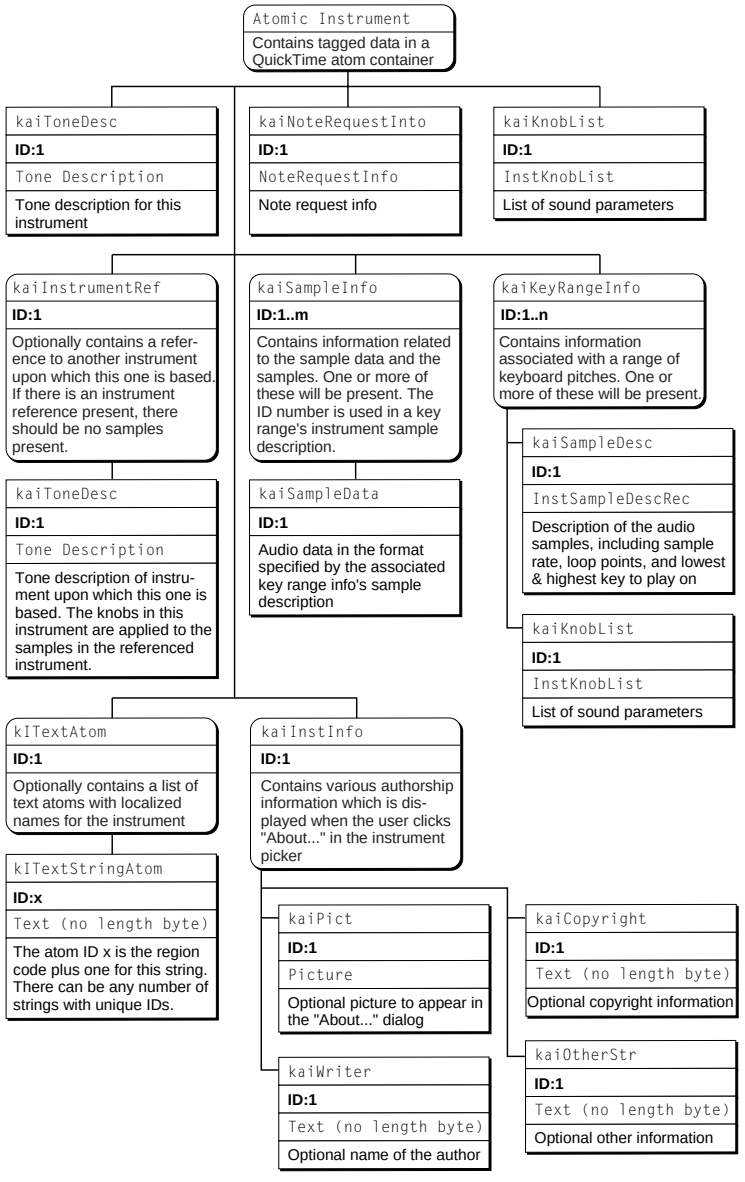
Other functions are for handling instruments and synthesizer parts. You can use these functions to initialize a part to a specified instrument and to get lists of available instrument and drum kit names. You can also get detailed information about each instrument from the synthesizer and get information about and set knobs and controllers.

Instrument Components and Atomic Instruments

When initialized, the note allocator searches for components of type `'inst'`. These components may report a list of atomic instruments. They are called **atomic instruments**, because you create them with QT atoms. (QT atoms are described in Chapter 1, "Movie Toolbox," of *QuickTime 3 Reference*). These sounds can be embedded in a QuickTime movie, passed via a call to QuickTime, or dropped into the System Folder.

QuickTime 3 provides a public format for atomic instruments. Using the QuickTime calls for manipulating atoms, you construct in memory a hierarchical tree of atoms with the data that describes the instrument (see Figure 1-2). The tree of atoms lives inside an atom container. There is one and only one root atom per container. Each atom has a four-character (32-bit) type, and a 32-bit ID. Each atom may be either an internal node or a leaf atom with data.

Figure 1-2 An atomic instrument atom container



The QuickTime Music Synthesizer Component

The QuickTime music synthesizer component is a software-based synthesizer that is included with QuickTime. The sound it generates can be sent to the built-in speaker of a Macintosh or Mac OS–based computer or to the sound card or built-in audio circuitry of other computers.

The QuickTime music synthesizer includes a variety of built-in instruments in the atomic instrument format. You can also create new instruments for the synthesizer. The instruments used by the QuickTime music synthesizer are known as atomic instruments, because they are defined using QuickTime atoms.

The instruments of the QuickTime music synthesizer are described by a set of knobs and one or more waveforms.

IMPORTANT

To play notes, you normally use the note allocator or tune player component. These components invoke the QuickTime music synthesizer or another music component to generate sounds. If you need to use the QuickTime music synthesizer directly, you must open an instance of the note allocator, which is responsible for finding the instrument components that best fit the criteria for instruments, and leave it open while using the synthesizer. If the note allocator is not open, the QuickTime music synthesizer may be forced to repeatedly open and close connections to the note allocator, which can greatly diminish performance. This recommendation may also apply to other music components that use the note allocator's instrument library routines. ▲

Atomic instruments for the QuickTime music synthesizer are defined by some waveform data and a set of knob values. Knobs provide a way to modify the instrument sound—for example, by applying a tremolo. Typically, the instrument has a full list of knobs, and if the instrument contains more than a single sample, each sample contains values for several knobs that are tuned for that particular sample. In this context, a **sample** is defined as a short recording of a musical sound.

Knobs can be specified either by index or by ID. A nonzero value in the high byte of the 24-bit `number` field of an instrument knob record or `knobID` field of a knob description record indicates that it is an ID. The knob index ranges from 1

to the number of knobs; the ID is an arbitrary number. You should generally access knobs by ID, because knob IDs do not change over different versions of the QuickTime software whereas knob index values might.

The General MIDI Synthesizer Component

The General MIDI synthesizer component controls General MIDI devices. These devices support 24 voices of polyphony, and each of their MIDI channels can access any number of voices. A user can choose this synthesizer in the QuickTime Settings control panel. For information about the QuickTime Settings control panel, see “QuickTime Settings Music Panel” (page 32).

The MIDI Synthesizer Component

The MIDI synthesizer component allows QuickTime to control a synthesizer connected to a single MIDI channel. It works with any synthesizer that can be controlled through MIDI.

The MIDI synthesizer component does not get information about the synthesizer instruments. Instead, it simply lists available instruments as “Instrument 1,” “Instrument 2,” and so on—up to “Instrument 128.”

The Base Instrument Component

When you provide additional sounds for the QuickTime music synthesizer, you can simplify the creation of the necessary instrument resources by using the base instrument component. To create an instrument component, you create a component alias whose target is the base instrument component. The component alias’s data resources specify the capabilities of an instrument, while the code resource of the base instrument component handles all of the component requests sent to the instrument component.

For information about component aliases, see Chapter 2, “Component Manager,” in *QuickTime 3 Reference.*

The Generic Music Component

To use a new hardware or software synthesizer with the QuickTime music architecture, you need a music component that serves as a device driver for that synthesizer and that can play notes on the synthesizer. You can simplify the

creation of a music component by using the services of the generic music component. To create a music component, you create several resources, for which you get much of the data by calling functions of the generic music component, and implement functions that the generic music component calls when necessary. When a music component is a client of the generic music component, it handles only a few component calls from applications and more relatively simple calls from the generic music component.

MIDI Components

A MIDI component provides a standard interface between the note allocator component and a particular MIDI transport system, such as the Apple MIDI Manager or the Open Music System™ (OMS) developed by Opcode Systems, Inc. Each MIDI component supports both input and output of MIDI streams.

The QuickTime music architecture includes MIDI components for the following MIDI transport systems:

- The MIDI Manager developed by Apple Computer, Inc.
- The Open Music System (OMS) developed by Opcode Systems, Inc.
- The FreeMIDI system extension for the Mac OS developed by Mark of the Unicorn, Inc.

Hardware and software developers can provide additional MIDI components. For example, the developer of a multiport serial card can provide a MIDI component that supports direct MIDI input and output using the card. Other MIDI components can support MIDI transport systems for operating systems other than the Mac OS.

To use a MIDI component, you use the functions described in “MIDI Component Functions” (page 162). To create a new MIDI component, you create a component that implements these functions.

Note
QuickTime 3 for Windows does not yet support MIDI output; only the QuickTime music synthesizer is available. ◆

About QuickTime Music Events

Music events specify the instruments and notes of a musical composition. A group of music events is called a **sequence.** A sequence of events may define a range of instruments and their characteristics and the notes and rests that, when interpreted, produce the musical composition.

The event sequence required to produce music is usually contained in a QuickTime movie track, which uses a media handler to provide access to the tune player, or an application, which passes them directly to the tune player. QuickTime interprets and plays the music from the sequence data.

The events described in this section initialize and modify sound-producing music devices and define the notes and rests to be played.

Events are constructed as a group of long words. The uppermost 4 bits (nibble) of an event's long word defines its type, as shown in Table 1-1.

Table 1-1 Event types

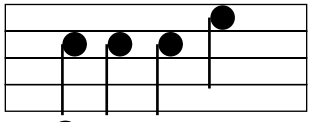
First nibble	Number of long words	Event type
000x	1	Rest
001x	1	Note
010x	1	Controller
011x	1	Marker
1000	2	(reserved)
1001	2	Extended note
1010	2	Extended controller
1011	2	Knob
1100	2	(reserved)

Table 1-1 Event types (continued)

First nibble	Number of long words	Event type
1101	2	(reserved)
1110	2	(reserved)
1111	*n*	General

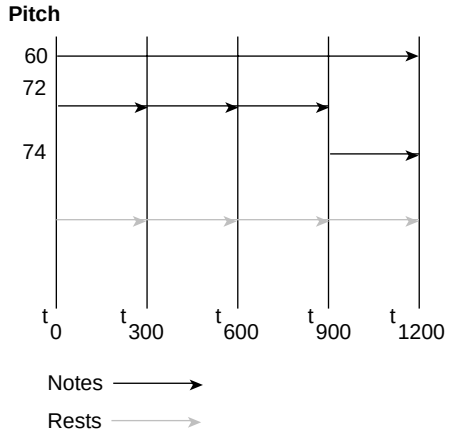
Durations of notes and rests are specified in units of the tune player's time scale (default 1/600 second). For example, consider the musical fragment shown in Figure 1-3.

Figure 1-3 A music fragment



Assuming 120 beats per minute, and a tune player's scale of 600, each quarter note's duration is 300. Figure 1-4 shows a graphical representation of note and rest data.

Figure 1-4 Duration of notes and rests



The general event specifies the types of instruments or sounds used for the subsequent note events. The note event causes a specific instrument, previously defined by a general event, to play a note at a particular pitch and velocity for a specified duration of time.

Additional event types allow sequences to apply controller effects to instruments, define rests, and modify instrument knob values. The entire sequence is closed with a marker event.

In most cases, the standard note and controller events (two long words) are sufficient for an application's requirements. The extended note event provides wider pitch range and fractional pitch values. The extended controller event expands the number of instruments and controller values over that allowed by a controller event.

The following sections describe the event types in detail.

Note Event and Extended Note Event

The standard note event (Figure 1-5) supports most music requirements. The note event allows up to 32 parts, numbered 0 to 31, and support pitches from 2 octaves below middle C to 3 octaves above. The extended note event (Figure 1-6) provides a wider range of pitch values, microtonal values to define

any pitch, and extended note duration. The extended note event requires two long words; the standard note event requires only one.

Figure 1-5 note event

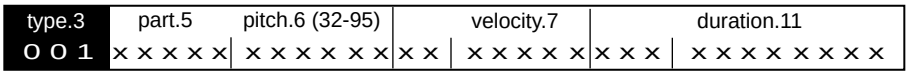


Table 1-2 Contents of a note event

note event type	First nibble value = `001X`
Part number	Unique part identifier
Pitch	Numeric value of 0–63, mapped to 32–95
Velocity	0–127, 0 = no audible response (but used to indicate a NOTE OFF)
Duration	Specifies how long to play the note in units defined by the media time scale or tune player time scale

The part number bit field contains the unique part identifier initially used during the `TuneSetHeader` call.

The pitch bit field allows a range of 0–63, which is mapped to the values 32–95 representing the traditional equal tempered scale. For example, the value 28 (mapped to 60) is middle C.

The velocity bit field allows a range of 0–127. A velocity value of 0 produces silence.

The duration bit field defines the number of units of time during which the part will play the note. The units of time are defined by the media time scale or tune player time scale.

Use this macro call to stuff the note event's long word:

```
qtma_StuffNoteEvent(x, instrument, pitch, volume, duration)
```

Use these macro calls to extract fields from the note event's long word:

```
qtma_Instrument(x)
qtma_NotePitch(x)
qtma_NoteVelocity(x)
qtma_NoteDuration(x)
```

Figure 1-6 Extended note event

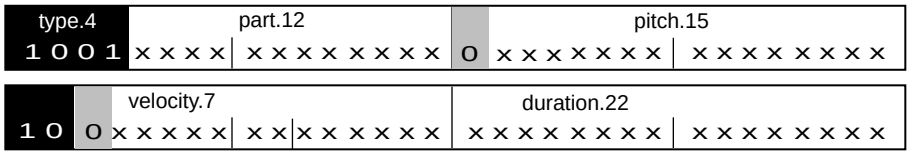


Table 1-3 Contents of an extended note event

Extended note event type	First nibble value = `1001`
Part number	Unique part identifier
Pitch	0–127 standard pitch, 60 = middle C `0x01.00` … `0x7F.00` allowing 256 microtonal divisions between each notes in the traditional equal tempered scale
Velocity	0–127 where 0 = no audible response (but used to indicate a NOTE OFF)
Duration	Specifies how long to play the note in units defined by media time scale or tune player time scale
Event tail	First nibble of last word = `10XX`

The part number bit field contains the unique part identifier initially used during the `TuneSetHeader` call.

If the pitch bit field is less than 128, it is interpreted as an integer pitch where 60 is middle C. If the pitch is 128 or greater, it is treated as a fixed pitch.

Microtonal pitch values are produced when the 15 bits of the pitch field are split. The upper 7 bits define the standard equal tempered note and the lower 8 bits define 256 microtonal divisions between the standard notes.

Use this macro call to stuff the extended note event's long words:

```
qtma_StuffXNoteEvent(w1, w2, instrument, pitch, volume, duration)
```

Use these macro calls to extract fields from the extended note event's long words:

```
qtma_XInstrument(m, l)
qtma_XNotePitch(m, l)
qtma_XNoteVelocity(m, 1)
qtma_XNoteVolume(m, l)
qtma_XNoteDuration(m, l)
```

Rest Event

The rest event (Figure 1-7) specifies the period of time, defined by either the media time scale or the tune player time scale, until the next event in the sequence is played.

Figure 1-7 Rest event

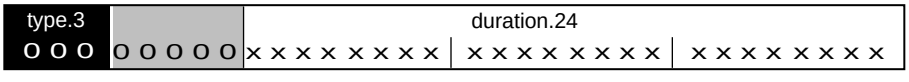


Table 1-4 Contents of a rest event

Rest event type	First nibble value = `000X`
Duration	Specifies the number of units of time until the next note event is played in units defined by media time scale or tune player time scale

Use this macro call to stuff the rest event's long word:

```
qtma_StuffRestEvent(x, duration)
```

Use this macro call to extract the rest event's duration value:

```
qtma_RestDuration(x)
```

Note
Rest events are not used to cause silence in a sequence, but to define the start of subsequent events. ◆

Marker Event

The marker event has three subtypes. The end marker event (Figure 1-8) marks the end of a series of events. The beat marker event marks the beat and the tempo marker event indicates the tempo.

Figure 1-8 Marker event of subtype end

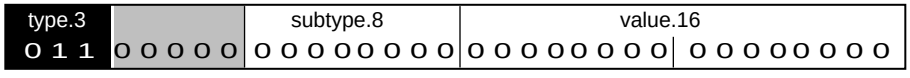


Table 1-5 Contents of a marker event

Marker event type	First nibble value = `011X`
Subtype	8-bit unsigned subtype
Value	16-bit signed value

The marker subtype bit field contains zero for an end marker (`kMarkerEventEnd`), 1 for a beat marker (`kMarkerEventBeat`), or 2 for a tempo marker (`kMarkerEventTempo`).

The value bit field varies according to the subtype:

- For an end marker event, a value of 0 means stop; any other value is reserved.
- For a beat marker event, a value of 0 is a single beat (a quarter note); any other value indicates the number of fractions of a beat in 1/65536 beat.
- For a tempo marker event, the value is the same as a beat marker, but indicates that a tempo event should be computed (based on where the next beat or tempo marker is) and emitted upon export.

Use this macro call to stuff the marker event's long word:

```
qtma_StuffMarkerEvent(x, markerType, markerValue)
```

Use these macro calls to extract fields from the marker events long word:

```
qtma_MarkerSubtype(x)
qtma_MarkerValue(x)
```

Controller Event and Extended Controller Event

The controller event (Figure 1-9) changes the value of a controller on a specified part. The extended controller event (Figure 1-10) allows parts and controllers beyond the range of the standard controller event.

Figure 1-9 Controller event

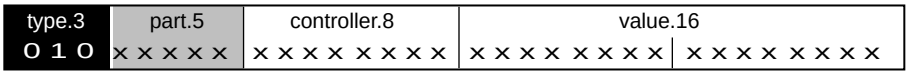


Table 1-6 Contents of a controller event

controller event type	First nibble value = `010X`
Part	Unique part identifier
Controller	Controller to be applied to instrument
Value	8.8 bit fixed-point signed controller specific value

For a list of currently supported controller types see “Controller Numbers” (page 56).

The part field contains the unique part identifier initially used during the `TuneSetHeader` call.

The controller bit field is a value that describes the type of controller used by the part.

The value bit field is specific to the selected controller.

Use this macro call to stuff the controller event’s long word:

```
qtma_StuffControlEvent(x, instrument, control, value)
```

Use these macro calls to extract fields from the controller event's long word:

```
qtma_Instrument(x)
qtma_ControlController(x)
qtma_ControlValue(x)
```

Figure 1-10 Extended controller event

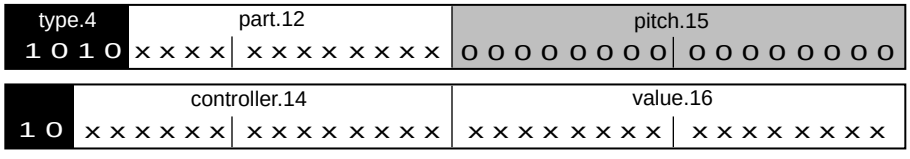


Table 1-7 Contents of an extended controller event

Extended controller type	First nibble value = `1010`
Part	Instrument index for controller
Controller	Controller for instrument
Value	Signed controller specific value
Event tail	First nibble of last word = `10XX`

The part field contains the unique part identifier initially used during the `TuneSetHeader` call.

The controller bit field contains a value that describes the type of controller to be used by the part.

The value bit field is specific to the selected controller.

Use this macro call to stuff the extended controller event's long words:

```
_StuffXControlEvent(w1, w2, instrument, control, value)
```

Use these macro calls to extract fields from the extended controller event's long words:

```
qtma_XInstrument(m, l)
qtma_XControlController(m, l)
qtma_XControlValue(m, l)
```

General Event

For events longer than two words, you use the general event with a subtype. Figure 1-11 illustrates the contents of a general event.

Figure 1-11 A note request general event

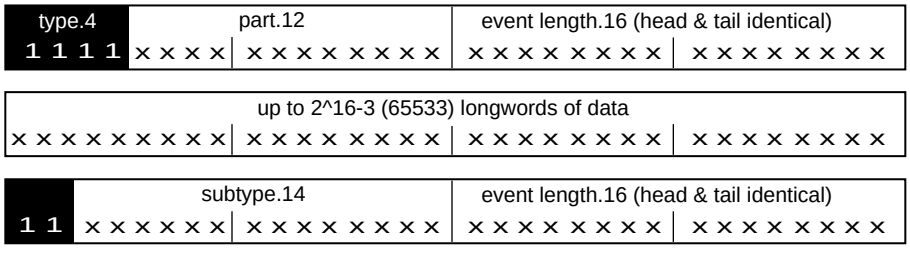


Table 1-8 Contents of a general event

General event type	First nibble value = `1111`
Part number	Unique part identifier
Event length	Head is number of words in event
Data words	Depends on subtype
Subtype	8-bit unsigned subtype
Event length	tail must be identical to head
Event tail	First nibble of last word = `11XX`

The part number bit field contains a unique identifier that is later used to match note, knob, and controller events to a specific part. For example, to play a note the application uses the part number to specify which instrument will play the note. The general event allows part numbers of up to 12 bits. The standard note and controller events allow part numbers of up to 5 bits; the extended note and extended controller events allow 12-bit part numbers.

The event length bit fields contained in the first and last words of the message are identical and are used as a message format check and to move back and forth through the message. The lengths include the head and tail; the smallest length is 2.

The data words field is a variable length field containing information unique to the subtype of the general event. The subtype bit field indicates the subtype of general event. There are nine subtypes:

- A note request general event (`kGeneralEventNoteRequest`) has a subtype of 1. It encapsulates the note request data structure used to define the instrument or part. It is used in the tune header.
- A part key general event (`KGeneralEventPartKey`) has a subtype of 4. It sets a pitch offset for the entire part so that every subsequent note played on that part will be altered in pitch by the specified amount.
- A tune difference general event (`kGeneralEventTuneDifference`) has a subtype of 5. It contains a standard sequence, with end marker, for the tune difference of a sequence piece. Using a tune difference event is similar to using key frames with compressed video sequences. (This subtype halts QuickTime 2.0 music).
- An atomic instrument general event (`kGeneralEventAtomicInstrument`) has a subtype of 6. It encapsulates an atomic instrument. It is used in the tune header. It may be used in place of the `kGeneralEventNoteRequest`.
- A knob general event (`kGeneralEventKnob`) has a subtype of 7. It contains knob ID / knob value pairs. The smallest event is four long words.
- A MIDI channel general event (`kGeneralEventMIDIChannel`) has a subtype of 8. It is used in a tune header. One long word identifies the MIDI channel it originally came from.
- A part change general event (`kGeneralEventPartChange`) has a subtype of 9. It is used in a tune sequence where one long word identifies the tune part that can now take over the part's note channel. (This subtype halts QuickTime 2.0 music.)
- A no-op general event (`kGeneralEventNoOp`) has a subtype of 10. It does nothing in the current version of QuickTime.
- A notes-used general event (`kGeneralEventUsedNotes`) has a subtype of 11. It is four long words specifying which MIDI notes are actually used. It is used in the tune header.

Use these macro calls to stuff the general event's head and tail long words, but not the structures described above:

```
qtma_StuffGeneralEvent(w1, w2, instrument, subType, length)
```

Macros are used to extract field values from the event's head and tail long words.

```
qtma_XInstrument(m, l)
qtma_GeneralSubtype(m, l)
qtma_GeneralLength(m, l)
```

Knob Event

The knob event is used to modify a particular knob or knobs within a specified part.

Figure 1-12 Knob event

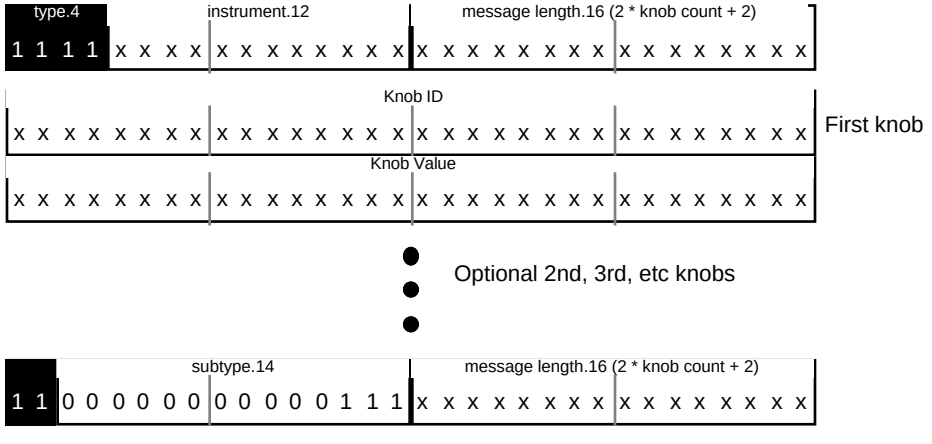


Table 1-9 Contents of a knob event

Knob event type	First nibble value = `1111` (general event), subtype 7
Length	Length of the event will be 2(#knobs+1)
Part	Unique part identifier
Knob ID	Knob ID within specified part
Knob value	Knob value
Event tail	First nibble of last word = `11XX`, subtype 7

The part field contains the unique part identifier initially used during the `TuneSetHeader` call.

The knob number bit field identifies the knob to be changed.

The 32-bit value composed of the lower 16-bit and upper 16-bit field values is used to alter the specified knob.

Using the QuickTime Music Architecture

The QuickTime Music Architecture provides functions that allow applications to control all aspects of playing music tracks and generating musical sounds in QuickTime movies.

This section discusses a few of the more common operations your application can perform with the QTMA, and it has been divided into the following subsections:

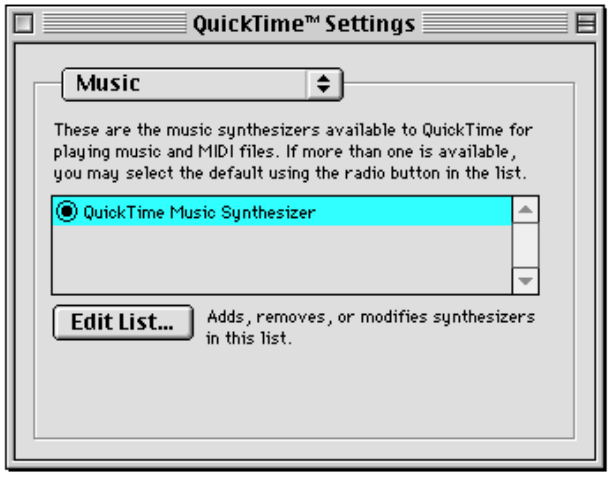
- "QuickTime Settings Music Panel" describes changes to the music panel in the QuickTime Settings control panel in QuickTime 3.
- "Converting MIDI Data to a QuickTime Music Track Using MoviePlayer" describes how you can open a standard MIDI file and convert it into a QuickTime music track.
- "Importing a Standard MIDI File As a Movie Using the Movie Toolbox" shows how you can read a Standard MIDI File (SMF) and convert it into a QuickTime movie.

- "Playing Notes With the Note Allocator" discusses how you can play notes with the note allocator component. A routine is also shown for playing notes in a piano sound with the note allocator component.

QuickTime Settings Music Panel

In QuickTime 3, the Music panel in the QuickTime Settings control panel has been completely revised. It now allows for greater flexibility in setting up QTMA synthesizer configurations, including multiple MIDI ports provided by OMS, FreeMIDI, or the MIDI Manager and multiple synthesizers. Figure 1-13 shows the new panel.

Figure 1-13 The new music panel in the QuickTime Settings control panel

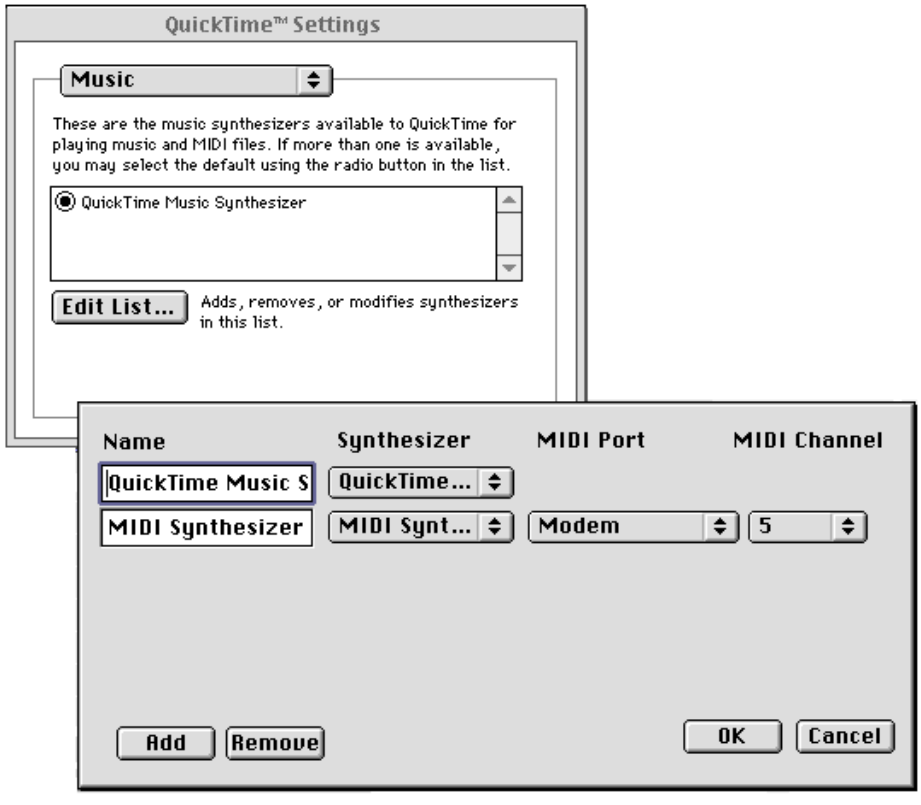


Note that the user can select from a list of available synthesizers for playing music and MIDI files. The user can also configure the synthesizers in the list by clicking the Edit List button.

Figure 1-14 displays the dialog box that appears when the user clicks the Edit List button.

Note
The screen displayed in Figure 1-14 is preliminary and subject to change. The functionality of configuring synthesizers in the list, however, will not change in QuickTime 3. ◆

Figure 1-14 The Edit List popup dialog box for adding, removing, and configuring QTMA synthesizers



If a General MIDI synthesizer is selected in the Synthesizer pop-up menu, the user must also specify which MIDI port the synthesizer is connected to, as shown in Figure 1-14. If there is no MIDI system installed (for example, OMS,

FreeMIDI, or MIDI Manager on the Macintosh), General MIDI does not appear in the Synthesizer pop-up menu.

QuickTime 3 includes one additional synthesizer type: a generic "MIDI Synthesizer," which can be any MIDI device that lives on a single channel. Figure 1-14 shows the control panel set up for a single MIDI Synthesizer on MIDI channel 5.

Converting MIDI Data to a QuickTime Music Track Using MoviePlayer

The MoviePlayer and SimpleText applications allow you to open a standard MIDI file and convert it into a QuickTime music track. After the file is converted, the application prompts you to save the converted file as a QuickTime movie. Once saved, a movie controller is displayed and you can play the music.

Importing a Standard MIDI File As a Movie Using the Movie Toolbox

Most music content exists in Standard MIDI Files (SMF), which have a standard format. All sequencing and composition programs let you save or export files in this format. QuickTime provides facilities for reading an SMF and converting it into a QuickTime movie. During any kind of conversion, the SMF is assumed to be scored for a General MIDI device, and MIDI channel 10 is assumed to be a drum track.

The conversion to a QuickTime movie can happen in one of several ways. Because it is implemented in a QuickTime `'eat '` component, the conversion happens automatically in most cases. Any application that uses the `StandardGetFile` routine to open a movie can also open `'Midi'` files transparently, and can transparently paste Clipboard contents of type `'Midi'` into a movie shown with the standard movie controller.

To explicitly convert a file or handle into a movie, your application can use the Movie Toolbox routines `ConvertFileToMovieFile` and `PasteHandleIntoMovie`, respectively.

When authoring MIDI files to be converted to QuickTime music movies, two MIDI system-exclusive messages can be used for more precise control of the MIDI import process. Note that QuickTime data is divided into media samples.

Within video tracks, each video frame is considered one sample; in music tracks, each sample can contain several seconds worth of musical information.

- `F0 11 00 01 xx yy zz F7` sets the maximum size of each media sample to the 21-bit number `xxyyzz`. (MIDI data bytes have the high bit clear, so they have only seven bits of number.) This message can occur anywhere in an SMF.
- `F0 11 00 02 F7` marks an immediate sample break; it ends the current sample and starts a new one. All messages after a sample break message are placed in a new media sample.

Applications can define their own system-exclusive messages of the form `F0 11 7F ww xx yy zz` ... application-defined data ... `F7`, where `ww xx yy zz` is the application's unique signature with the high bits cleared. This is guaranteed not to interfere with Apple's or any other manufacturer's use of system-exclusive codes.

Playing Notes With the Note Allocator

Playing a few notes with the note allocator component is simple and straightforward. To play notes that have a piano sound, for example, you need to open up the note allocator component, allocate a note channel with a request for piano, and play. When you've finished playing notes, you dispose of the note channel and close the note allocator component. The code to accomplish this is shown in Listing 1-2. Before working through the code, you need to look at some important related data structures.

Note-Related Data Structures

A note channel is analogous to a sound channel in that you allocate it, issue commands to it to produce sound, and close it when you're done. To specify details about the note channel, you use a data structure called a `NoteRequest` (see Listing 1-1).

Listing 1-1 Note-related data structures

```
struct NoteRequest {
    NoteRequestInfo   info;      // in post-QuickTime 2.0 only
    ToneDescription   tone;
};
```

```
struct NoteRequestInfo {
    UInt8       flags;
    UInt8       reserved;
    short       polyphony;
    Fixed       typicalPolyphony;
};

struct ToneDescription {
    OSType      synthesizerType;
    Str31       synthesizerName;
    Str31       instrumentName;
    long        instrumentNumber;
    long        gmNumber;
};
```

The next two fields specify the probable polyphony that the note channel will be used for. **Polyphony** means, literally, *many sounds*. A polyphony of 5 means that five notes can be playing simultaneously. The `polyphony` field enables QTMA to make sure that the allocated note channel can play all the notes you need. The typical `polyphony` field is a fixed-point number that should be set to the average number of voices the note channel will play; it may be whole or fractional. Some music components use this field to adjust the mixing level for a good volume. If in doubt, set the typical `polyphony` field to 0X00010000.

The `ToneDescription` structure is used throughout QTMA to specify a musical instrument sound in a device-independent fashion. This structure's `synthesizerType` and `synthesizerName` fields can request a particular synthesizer to play notes on. Usually, they're set to 0, meaning "choose the best General MIDI synthesizer." The `gmNumber` field indicates the General MIDI (GM) instrument or drum kit sound, which may be any of 135 such sounds supported by many synthesizer manufacturers. (All these sounds are available on a General MIDI Sound Module.) The GM instruments are numbered 1 through 128, and the seven drum kits are numbered 16385 and higher. For synthesizers that accept sounds outside the GM library, you can use the `instrumentName` and `instrumentNumber` fields to specify some other sound.

Playing Piano Sounds With the Note Allocator

The routine in Listing 1-2 plays notes in a piano sound with the note allocator component.

Listing 1-2 Playing notes with the note allocator component

```
void PlaySomeNotes(void)
{
    NoteAllocator      na;
    NoteChannel         nc;
    NoteRequest         nr;
    ComponentResult   thisError;
    long                t, i;

    na = 0;
    nc = 0;

    // Open up the note allocator.
    na = OpenDefaultComponent(kNoteAllocatorType, 0);
    if (!na)
        goto goHome;

    // Fill out a NoteRequest using NAStuffToneDescription to help, and
    // allocate a NoteChannel.
    nr.info.flags = 0;
    nr.info.reserved = 0;
    nr.info.polyphony = 2;      // simultaneous tones
    nr.info.typicalPolyphony = 0x00010000; // usually just one note
    thisError = NAStuffToneDescription(na, 1, &nr.tone);  // 1 is piano
    thisError = NANewNoteChannel(na, &nr, &nc);
    if (thisError || !nc)
        goto goHome;

    // If we've gotten this far, OK to play some musical notes. Lovely.
    NAPlayNote(na, nc, 60, 80);     // middle C at velocity 80
    Delay(40, &t);                  // delay 2/3 of a second
    NAPlayNote(na, nc, 60, 0);      // middle C at velocity 0: end note
    Delay(40, &t);                  // delay 2/3 of a second

    // Obligatory do-loop of rising tones
    for (i = 60; i <= 84; i++) {
        NAPlayNote(na, nc, i, 80);       // pitch i at velocity 80
        NAPlayNote(na, nc, i+7, 80);     // pitch i+7 (musical fifth) at
                                         // velocity 80
        Delay(10, &t);                   // delay 1/6 of a second
```

```
        NAPlayNote(na, nc, i, 0); // pitch i at velocity 0: end note
        NAPlayNote(na, nc, i+7, 0);   // pitch i+7 at velocity 0:
                                        // end note
    }

goHome:
    if (nc)
        NADisposeNoteChannel(na, nc);
    if (na)
        CloseComponent(na);
}
```

You start by calling `OpenDefaultComponent` to open a connection to the note allocator. If this routine returns 0, the component wasn't opened, most likely because QTMA wasn't present. Next, you fill in the `NoteRequestInfo` and `ToneDescription` structures, calling the note allocator's `NAStuffToneDescription` routine and passing it the GM instrument number for piano. This routine fills in the `gmNumber` field and also fills in the other `ToneDescription` fields with sensible values, such as the instrument's name in text form in the `instrumentName` field. (The routine can be useful for converting a GM instrument number to its text equivalent.)

After allocating the note channel with `NANewNoteChannel`, you call `NAPlayNote` to play each note. Notice the last two parameters to `NAPlayNote`:

```
ComponentResult NAPlayNote(NoteAllocator na, NoteChannel nc,
    long pitch, long velocity);
```

The value of the `pitch` parameter is an integer from 1 to 127, where 60 is middle C, 61 is C sharp, and 59 is C flat, or B. Similarly, 69 is concert A and is played at a nominal audio frequency of 440 Hz.

The `velocity` parameter's value is also an integer from 1 to 127, or 0. A velocity of 1 corresponds to just barely touching the musical keyboard, and 127 indicates that the key was struck as hard as possible. Different velocities produce tones of different volumes from the synthesizer. A velocity of 0 means the key was released; the note stops or fades out, as appropriate to the kind of sound being played.

You stop the notes at this point after delaying an appropriate amount of time with a call to the `Delay` routine. Finally, you dispose of the note channel and close the note allocator component.

Music Architecture Reference

This chapter describes the constants, data structures, functions, and result codes provided by QuickTime music architecture.

Constants

This section describes the constants provided by QuickTime music architecture.

Atom Types for Atomic Instruments

These constants specify the types of atoms used to build atomic instruments. Atomic instruments are described in “Instrument Components and Atomic Instruments” (page 14).

```
enum {
    kaiToneDescType                 = 'tone',
    kaiNoteRequestInfoType          = 'ntrq',
    kaiKnobListType                 = 'knbl',
    kaiKeyRangeInfoType             = 'sinf',
    kaiSampleDescType               = 'sdsc',
    kaiSampleDataType               = 'sdat',
    kaiInstRefType                  = 'iref',
    kaiInstInfoType                 = 'iinf',
    kaiPictType                     = 'pict',
    kaiWriterType                   = '©wrt',
    kaiCopyrightType                = '©cpy',
    kaiOtherStrType                 = 'str '
};
```

Constant descriptions

`kaiToneDescType` A tone atom, which describes the tone. It contains a tone description structure (page 75).

`kaiNoteRequestInfoType` A note request information atom, which contains a note request information structure (page 85). The note request information structure includes information about a tone that is not in the tone description. Use a note request information atom when embedding an instrument in a sample description of a QuickTime movie. If this atom is absent, QuickTime assumes “reasonable” values for polyphony.

`kaiKnobListType` A knob list atom, which specifies values for one or more knobs. It contains an instrument knob list structure (page 72). Use it with a custom instrument, a modified built-in instrument, or as part of a sample.

`kaiKeyRangeInfoType` A key range information atom contains several other atoms. It also refers, via an ID, to one or more sibling sample info (`kaiSampleInfoType`) atoms. Use a key range information atom to include a sampled sound in an atomic instrument.

`kaiSampleDescType` A sample description atom, which contains an atomic instrument sample description structure (page 72).

`kaiSampleDataType` A sample data atom, which contains the actual audio data.

`kaiInstRefType` An instrument reference atom, which contains a tone description to be modified by a knob list atom.

`kaiInstInfoType` An instrument information atom, which contains four optional atoms with information for the instrument About box.

`kaiPictType` A picture atom that includes the graphic used in the instrument About box.

`kaiWriterType` A text atom that has the author information used in the instrument About box.

`kaiCopyrightType` A text atom that has the copyright information used in the instrument About box.

`kaiOtherStrType` A text atom that has additional information for the instrument About box.

`kaiSampleInfoType` A text atom that contains a sample data (`kiaSampleDataType`) atom.

Instrument Knob Flags

These flags are used in the `knobFlags` field of an instrument knob list structure (page 72) to indicate what to do if a requested knob is not in the list.

```
enum {
    kInstKnobMissingUnknown         = 0,
    kInstKnobMissingDefault         = 1 << 0
};
```

Constant descriptions

`kInstKnobMissingUnknown`
If the requested knob is not in the list, do not set its value.

`kInstKnobMissingDefault`
If the requested knob is not in the list, use its default value.

Loop Type Constants

You can use these constants in the `loopType` field of an atomic instrument sample description structure (page 72) to indicate the type of loop you want.

```
enum {
    kMusicLoopTypeNormal            = 0,
    kMusicLoopTypePalindrome        = 1
};
```

Constant descriptions

`kMusicLoopTypeNormal`
Use a regular loop.

`kMusicLoopTypePalindrome`
Use a back-and-forth loop.

Music Component Type

Use this constant to specify a QuickTime music component.

```
enum {
    kMusicComponentType     = 'musi'
};
```

Constant description

`kMusicComponentType`
The type of any QTML music component.

Synthesizer Type Constants

You can use these constants in a tone description structure (page 75) to specify the type of synthesizer you want to produce the tone.

```
enum {
    kSoftSynthComponentSubType      = 'ss  ',
    kGMSynthComponentSubType        = 'gm  '
};
```

Constant descriptions

`kSoftSynthComponentSubType`
Use the QuickTime music synthesizer. This is the built-in synthesizer.

`kGMSynthComponentSubType`
Use the General MIDI synthesizer.

Synthesizer Description Flags

These flags describe various characteristics of a synthesizer. They are used in the `flags` field of the synthesizer description structure (page 73).

```
enum {
    kSynthesizerDynamicVoice        = 1,
    kSynthesizerUsesMIDIPort        = 2,
    kSynthesizerMicrotone           = 4,
```

```
    kSynthesizerHasSamples          = 8,
    kSynthesizerMixedDrums          = 6,
    kSynthesizerSoftware            = 32,
    kSynthesizerHardware            = 64,
    kSynthesizerDynamicChannel      = 128,
    kSynthesizerHogsSystemChannel   = 256,
    kSynthesizerSlowSetPart         = 1024,
    kSynthesizerOffline             = 4096,
    kSynthesizerGM                  = 16384
};
```

Constant descriptions

`kSynthesizerDynamicVoice`
: Voices can be assigned to parts on the fly with this synthesizer (otherwise, polyphony is very important).

`kSynthesizerUsesMIDIPort`
: This synthesizer must be patched through a MIDI system, such as the MIDI Manager or OMS.

`kSynthesizerMicrotone`
: This synthesizer can play microtonal scales.

`kSynthesizerHasSamples`
: This synthesizer has some use for sampled audio data.

`kSynthesizerMixedDrums`
: Any part of this synthesizer can play drum parts.

`kSynthesizerSoftware`
: This synthesizer is implemented in main CPU software and uses CPU cycles.

`kSynthesizerHardware`
: This synthesizer is a hardware device, not a software synthesizer or MIDI device.

`kSynthesizerDynamicChannel`
: This synthesizer can move any part to any channel or disable each part. For devices only.

`kSynthesizerHogsSystemChannel`
: Even if the `kSynthesizerDynamicChannel` bit is set, this synthesizer always responds on its system channel. For MIDI devices only.

`kSynthesizerSlowSetPart`
: This synthesizer does not respond rapidly to the various set part and set part instrument calls.

`kSynthesizerOffline`
: This synthesizer can enter an offline synthesis mode.

`kSynthesizerGM`
: This synthesizer is a General MIDI device.

Synthesizer Knob ID Constants

These constants specify knob IDs for the QuickTime music synthesizer. These constants are all of the form `kQTMSKnob`*knobname*`ID`. For example, `kQTMSKnobVolumeLFODelayID` is the ID constant for the `VolumeLFODelay` knob.

```
enum {
    kQTMSKnobEnv1AttackTimeID           = 0x02000027,
    kQTMSKnobEnv1DecayTimeID            = 0x02000028,
    kQTMSKnobEnv1ExpOptionsID           = 0x0200002D,
    kQTMSKnobEnv1ReleaseTimeID          = 0x0200002C,
    kQTMSKnobEnv1SustainInfiniteID      = 0x0200002B,
    kQTMSKnobEnv1SustainLevelID         = 0x02000029,
    kQTMSKnobEnv1SustainTimeID          = 0x0200002A,
    kQTMSKnobEnv2AttackTimeID           = 0x0200002E,
    kQTMSKnobEnv2DecayTimeID            = 0x0200002F,
    kQTMSKnobEnv2ExpOptionsID           = 0x02000034,
    kQTMSKnobEnv2ReleaseTimeID          = 0x02000033,
    kQTMSKnobEnv2SustainInfiniteID      = 0x02000032,
    kQTMSKnobEnv2SustainLevelID         = 0x02000030,
    kQTMSKnobEnv2SustainTimeID          = 0x02000031,
    kQTMSKnobExclusionGroupID           = 0x0200001C,
    kQTMSKnobFilterFrequencyEnvelopeDepthID
                                        = 0x0200003B,
    kQTMSKnobFilterFrequencyEnvelopeID  = 0x0200003A,
    kQTMSKnobFilterKeyFollowID          = 0x02000037,
    kQTMSKnobFilterQEnvelopeDepthID     = 0x0200003D,
                                        /* reverb threshhold */
    kQTMSKnobFilterQEnvelopeID          = 0x0200003C,
    kQTMSKnobFilterQID                  = 0x02000039,
    kQTMSKnobFilterTransposeID          = 0x02000038,
    kQTMSKnobLastIDPlus1                = 0x0200003F
```

```
    kQTMSKnobPitchEnvelopeDepthID       = 0x02000036, /* filter */
    kQTMSKnobPitchEnvelopeID            = 0x02000035,
    kQTMSKnobPitchLFODelayID            = 0x02000013,
    kQTMSKnobPitchLFODepthFromWheelID   = 0x02000025,
                                        /* volume nnv again */
    kQTMSKnobPitchLFODepthID            = 0x02000017,
    kQTMSKnobPitchLFOOffsetID           = 0x0200001B,
    kQTMSKnobPitchLFOPeriodID           = 0x02000015,
    kQTMSKnobPitchLFOQuantizeID         = 0x02000018,
                                        /* stereo related knobs */
    kQTMSKnobPitchLFORampTimeID         = 0x02000014,
    kQTMSKnobPitchLFOShapeID            = 0x02000016,
    kQTMSKnobPitchSensitivityID         = 0x02000023,
    kQTMSKnobPitchTransposeID           = 0x02000012,
                                        /* sample can override */
    kQTMSKnobReverbThresholdID          = 0x0200003E,
    kQTMSKnobStartID                    = 0x02000000,
    kQTMSKnobStereoDefaultPanID         = 0x02000019,
    kQTMSKnobStereoPositionKeyScalingID = 0x0200001A,
    kQTMSKnobSustainInfiniteID          = 0x0200001E,
    kQTMSKnobSustainTimeID              = 0x0200001D,
    kQTMSKnobVelocityHighID             = 0x02000021,
    kQTMSKnobVelocityLowID              = 0x02000020,
    kQTMSKnobVelocitySensitivityID      = 0x02000022,
    kQTMSKnobVolumeAttackTimeID         = 0x02000001,
                                        /* sample can override */
    kQTMSKnobVolumeDecayTimeID          = 0x02000002,
                                        /* sample can override */
    kQTMSKnobVolumeExpOptionsID         = 0x02000026, /* env1 */
    kQTMSKnobVolumeLFODelayID           = 0x02000007,
    kQTMSKnobVolumeLFODepthFromWheelID  = 0x02000024,
    kQTMSKnobVolumeLFODepthID           = 0x0200000B,
    kQTMSKnobVolumeLFOPeriodID          = 0x02000009,
    kQTMSKnobVolumeLFORampTimeID        = 0x02000008,
    kQTMSKnobVolumeLFOShapeID           = 0x0200000A,
    kQTMSKnobVolumeLFOStereoID          = 0x0200001F,
    kQTMSKnobVolumeOverallID            = 0x0200000C,
    kQTMSKnobVolumeReleaseKeyScalingID  = 0x02000005,
    kQTMSKnobVolumeReleaseTimeID        = 0x02000006,
                                        /* sample can override */
    kQTMSKnobVolumeSustainLevelID       = 0x02000003,
```

```
                                        /* sample can override */
    kQTMSKnobVolumeVelocity127ID        = 0x0200000D,
    kQTMSKnobVolumeVelocity16ID         = 0x02000011,
                                        /* pitch related knobs */
    kQTMSKnobVolumeVelocity32ID         = 0x02000010,
    kQTMSKnobVolumeVelocity64ID         = 0x0200000F,
    kQTMSKnobVolumeVelocity96ID         = 0x0200000E
};
```

Constant descriptions

`kQTMSKnobEnv1AttackTimeID`

Specifies the attack time of the first general-purpose envelope. This is the number of milliseconds between the start of a note and the maximum value of the attack.

`kQTMSKnobEnv1DecayTimeID`

Specifies the decay time of the first general-purpose envelope. This is the number of milliseconds between the time the attack is completed and the time the envelope level is reduced to the sustain level.

`kQTMSKnobEnv1ExpOptionsID`

Specifies whether segments of the envelope are treated as exponential curves. Bits 0, 1, 2, and 3 of the knob value specify the interpretation of the attack, decay, sustain, and release segments of the envelope, respectively. If any of these bits is 0, the level of the corresponding segment changes linearly from its initial to final value during the time interval specified by the corresponding envelope time knob. If any of these bits is nonzero, the level of the corresponding segment changes exponentially during the time interval specified by the corresponding envelope time knob. During an exponential decrease, the level changes from maximum amplitude (no attenuation) to approximately 1/65536th of maximum amplitude (96 dB of attenuation) during the time interval specified by the corresponding envelope time knob, and afterward the level immediately becomes 0.

`kQTMSKnobEnv1ReleaseTimeID`

Specifies the release time of the first general-purpose envelope.

`kQTMSKnobEnv1SustainInfiniteID`

Specifies infinite sustain for the first general-purpose envelope. If the value of this knob is `true`, the knob overrides the `kQTMSKnobEnv1SustainTimeID` knob and causes the sustain to last, at undiminished level. Instruments like an organ have infinite sustain.

`kQTMSKnobEnv1SustainLevelID`

Specifies the sustain level of the first general-purpose envelope. This is the percentage of full volume that the sample is initially played at after the decay time has elapsed.

`kQTMSKnobEnv1SustainTimeID`

Specifies the sustain time of the first general-purpose envelope. This is the number of milliseconds it takes for the sample to soften to 90% of its sustain level. This softening occurs in an exponential fashion, so it never actually reaches complete silence. This is used for instruments like a piano, which gradually soften over time even while the key is held down.

`kQTMSKnobEnv2AttackTimeID`

Specifies the attack time of the second general-purpose envelope. This is the number of milliseconds between the start of a note and the maximum value of the attack. Percussive sounds usually have zero attack time; gentler sounds may have short attack times. Long attack times are usually used for special effects.

`kQTMSKnobEnv2DecayTimeID`

Specifies the decay time of the second general-purpose envelope. This is the number of milliseconds between the time the attack is completed and the time the sample is reduced in volume to the sustain level.

`kQTMSKnobEnv2ExpOptionsID`

Specifies whether segments of the envelope are treated as exponential curves. Bits 0, 1, 2, and 3 of the knob value specify the interpretation of the attack, decay, sustain, and release segments of the envelope, respectively. If any of these bits is 0, the level of the corresponding segment changes linearly from its initial to final value during the time interval specified by the corresponding envelope time knob. If any of these bits is nonzero, the level of the

corresponding segment changes exponentially during the time interval specified by the corresponding envelope time knob. During an exponential decrease the level changes from maximum amplitude (no attenuation) to approximately 1/65536th of maximum amplitude (96 dB of attenuation) during the time interval specified by the corresponding envelope time knob, and afterward the level immediately becomes 0.

`kQTMSKnobEnv2ReleaseTimeID`

Specifies the release time of the second general-purpose envelope. This is the number of milliseconds it takes for the sound to soften down to silence after the key is released.

`kQTMSKnobEnv2SustainInfiniteID`

Specifies infinite sustain for the second general-purpose envelope. If the value of this knob is `true`, the knob overrides the `kQTMSKnobEnv2SustainTimeID` knob and causes the sustain to last, at undiminished volume, until the end of the sample. Instruments like an organ have infinite sustain.

`kQTMSKnobEnv2SustainLevelID`

Specifies the sustain level of the first general-purpose envelope. This is the percentage of full volume that the sample is initially played at after the decay time has elapsed.

`kQTMSKnobEnv2SustainTimeID`

Specifies the sustain time of the second general-purpose envelope. This is the number of milliseconds it takes for the sample to soften to 90% of its sustain level. This softening occurs in an exponential fashion, so it never actually reaches complete silence. This is used for instruments like a piano, which gradually soften over time even while the key is held down.

`kQTMSKnobExclusionGroupID`

Specifies an exclusion group. Within an instrument, no two notes with the same exclusion group number, excepting exclusion group, will ever sound simultaneously.This knob is generally used only as an override knob within a key range. (Note that the key range is not an entire instrument.) It is useful for simulating certain mechanical instruments in which the same mechanism produces different sounds. For example, in a drum kit, the open high hat and the closed

high hat are played on the same piece of metal. If you assign both sounds to the same exclusion group, playing a closed high hat sound immediately silences any currently playing open high hat sounds.

`kQTMSKnobFilterFrequencyEnvelopeDepthID`

Controls the depth of the envelope for the filter frequency. This is an 8.8 signed fixed-point value that specifies the number of semitones the frequency is altered when its envelope (specified by the `kQTMSKnobFilterFrequencyEnvelopeID` knob) is at maximum amplitude. If the value of the `kQTMSKnobFilterFrequencyEnvelopeID` knob is 0, which specifies not to use an envelope to affect filter frequency, the `kQTMSKnobFilterFrequencyEnvelopeDepthID` knob is ignored.

`kQTMSKnobFilterFrequencyEnvelopeID`

Specifies which of the two general-purpose envelopes to use to affect the filter frequency, or not to use an envelope to affect filter frequency. If the value of this knob is 0, no envelope is used. If the value of this knob is 1 or 2, the corresponding general-purpose envelope is used.

`kQTMSKnobFilterKeyFollowID`

Specifies how closely the frequency of the filter follows the note being played. The emphasis note is determined by the following formula, expressed in MIDI notes:

EmphasisNote = (*PlayedNote* – 60) * (`kQTMSKnobFilterKeyFollowID` / 100) – 60 – `kQTMSKnobFilterTransposeID`

`kQTMSKnobFilterQEnvelopeDepthID`

Controls the depth of the envelope for the emphasis (“Q”) of the filter. This is an 8.8 signed fixed-point value that specifies the emphasis is altered when its envelope (specified by the `kQTMSKnobFilterQEnvelopeID` knob) is at maximum amplitude. If the value of the `kQTMSKnobFilterQEnvelopeID` knob is 0, which specifies not to use an envelope to affect filter frequency, the `kQTMSKnobFilterQEnvelopeDepthID` knob is ignored.

`kQTMSKnobFilterQEnvelopeID`

Specifies which of the two general-purpose envelopes to

use to affect the emphasis (“Q”) of the filter, or not to use an envelope to affect the emphasis. If the value of this knob is 0, no envelope is used. If the value of this knob is 1 or 2, the corresponding general-purpose envelope is used.

`kQTMSKnobFilterQID`

Specifies the emphasis (“Q”) of the filter. The value must be in the range 0 to 65536, inclusive, where 0 specifies no emphasis and disables the filter, and 65536 specifies relatively steep emphasis, but not so steep that it approaches feedback.

`kQTMSKnobFilterTransposeID`

Specifies a transposition, in semitones, of the frequency of the filter. The emphasis note is determined by the following formula:

EmphasisNote = (*PlayedNote* – 60) * (`kQTMSKnobFilterKeyFollowID` / 100) – 60 – `kQTMSKnobFilterTransposeID`

`kQTMSKnobPitchEnvelopeDepthID`

Specifies the depth of the pitch envelope. This is an 8.8 signed fixed-point value that specifies the number of semitones the pitch is altered when the envelope for the pitch (specified by the `kQTMSKnobPitchEnvelopeID` knob) is at maximum amplitude. If the value of the `kQTMSKnobPitchEnvelopeID` knob is 0, which specifies not to use an envelope to affect pitch, the `kQTMSKnobPitchEnvelopeDepthID` knob is ignored.

`kQTMSKnobPitchEnvelopeID`

Specifies which of the two general-purpose envelopes to use to affect pitch, or not to use an envelope to affect pitch. If the value of this knob is 0, no envelope is used. If the value of this knob is 1 or 2, the corresponding general-purpose envelope is used to affect pitch.

`kQTMSKnobPitchLFODelayID`

Specifies the delay for the pitch LFO. This is the number of milliseconds before the LFO takes effect.

`kQTMSKnobPitchLFODepthFromWheelID`

Specifies the extent to which a synthesizer’s modulation wheel (or the MIDI messages it generates) controls the depth of the pitch LFO. The value of this knob is multiplied

by the modulation wheel value (a value between 0 to 1), and the result is added to the volume LFO depth specified by the `kQTMSKnobPitchLFODepthID` knob. Modulation wheel controllers and the MIDI messages they generate are most often used to create vibrato and tremolo effects.

`kQTMSKnobPitchLFODepthID`

Specifies the depth of the pitch LFO. This is the number of semitones by which the pitch is altered by the LFO. A value of 0 does not change the pitch. A value of 12 changes the pitch from an octave lower to an octave higher, with one exception: if the square up waveform is used for the LFO, the normal pitch is the minimum pitch.

`kQTMSKnobPitchLFOOffsetID`

Specifies the LFO offset. This is a constant value; the units are 8.8 semitones. It is added to the pitch, and is affected by the LFO delay and LFO ramp-up times. It is similar to transposition but subject to the LFO delay and LFO ramp-up times.

`kQTMSKnobPitchLFOPeriodID`

Specifies the period for the pitch LFO. This is the wavelength of the LFO in milliseconds. (The LFO rate in Hz is 1000 / `kQTMSKnobPitchLFOPeriodID`).

`kQTMSKnobPitchLFOQuantizeID`

To be provided

`kQTMSKnobPitchLFORampTimeID`

Specifies the LFO ramp-up time. This is the number of milliseconds after the LFO delay that it takes for the LFO to reach full effect.

`kQTMSKnobPitchLFOShapeID`

Specifies the waveform used for the LFO. The available waveforms are sine, triangle, sawtooth up, sawtooth down, square up, square up-and-down, and random. The sine and triangle shapes both produce a smooth rise and fall of the pitch. The sawtooth up produces a gradual increase in pitch followed by a sudden fall. The sawtooth down shape produces a sudden increase in pitch, followed by a gradual reduction. The square up and square up-and-down shapes apply a sudden pulsing to the pitch; the square up only makes the pitch higher, while the up-and-down variant

makes the sound higher and lower. The random shape applies random changes to the pitch, once per LFO period.

`kQTMSKnobPitchSensitivityID`

Specifies the pitch key scaling. This determines how much the pitch of the struck note affects the pitch of the played note. Typically, this is 100%, meaning that a change in 1 semitone of the struck note produces a change in 1 semitone of the played note. Setting this knob to zero causes every note to play at the same pitch. Setting it to 50% allows for all notes within the quarter-tone scale (24 notes per octave) to be played.

`kQTMSKnobPitchTransposeID`

Specifies a transposition for pitches. The value is the number of semitones to transpose; a positive value raises the pitch anda negative value lowers it. The value can be a real number; the fractional part of the value alters the pitch by an additional fraction of a semitone. For example, to raise the pitch of every note played on the instrument by an octave, set the transpose knob to 12.0.

`kQTMSKnobReverbThresholdID`

To be provided

`kQTMSKnobStartID`

To be provided

`kQTMSKnobStereoDefaultPanID`

Specifies the default pan position for stereo sound. If no pan controller is applied, this determines where in the stereo field notes for this instrument are played.

`kQTMSKnobStereoPositionKeyScalingID`

Specifies the key scaling for stereo sound. Amount to modify the stereo placement of notes based upon pitch. At the highest setting, high pitched notes are placed completely in the right speaker, while low pitched notes are placed entirely in the left speaker.

`kQTMSKnobSustainInfiniteID`

Specifies infinite sustain for the volume envelope. If the value of this knob is `true`, the knob overrides the `kQTMSKnobSustainTimeID` knob and causes the sustain to last, at undiminished volume, until the end of the sample. Instruments like an organ have infinite sustain.

`kQTMSKnobSustainTimeID`

Specifies the sustain time of the volume envelope. This is the number of milliseconds it takes for the note to soften to 90% of its sustain level. This softening occurs in an exponential fashion, so it never actually reaches complete silence. This is used for instruments like a piano, which gradually soften over time even while the key is held down.

`kQTMSKnobVelocityHighID`

Specifies the maximum velocity value that produces sound for a particular note. If the velocity value is greater, the note does not sound. This can be used to assign different samples to be played for selected velocity ranges.

`kQTMSKnobVelocityLowID`

Specifies the minimum velocity value that produces sound for a particular note. If the velocity value is less, the note does not sound. This can be used to assign different samples to be played for selected velocity ranges.

`kQTMSKnobVelocitySensitivityID`

Specifies velocity sensitivity, which determines how much the key velocity affects the volume of the note. This value is a percentage. At 100%, a velocity of 1 is nearly silent, and a velocity of 127 is full volume. At 50%, the volume range is from one fourth to three fourths. At 0%, any velocity of key strike produces a half volume note. If the value of this knob is negative, then the note plays more softly as the key is struck harder.

`kQTMSKnobVolumeAttackTimeID`

Specifies the attack time for the volume envelope. This is the number of milliseconds between the start of a note and maximum volume. Percussive sounds usually have zero attack time; gentler sounds may have short attack times. Long attack times are usually used for special effects.

`kQTMSKnobVolumeDecayTimeID`

Specifies the decay time for the volume envelope. This is the number of milliseconds between the time the attack is completed and the time the volume is reduced to the sustain level.

`kQTMSKnobVolumeExpOptionsID`

Specifies whether segments of the volume envelope are treated as exponential curves. Bits 0, 1, 2, and 3 of the knob value specify the interpretation of the attack, decay, sustain, and release segments of the volume envelope, respectively. If any of these bits is 0, the volume level of the corresponding segment changes linearly from its initial to final value during the time interval specified by the corresponding envelope time knob. If any of these bits is nonzero, the volume level of the corresponding segment changes exponentially during the time interval specified by the corresponding envelope time knob. During an exponential decrease the volume level changes from full volume (no attenuation) to approximately 1/65536th of full volume (96 dB of attenuation) during the time interval specified the corresponding envelope time knob, and afterward the volume level immediately becomes 0.

`kQTMSKnobVolumeLFODelayID`

Specifies the delay for the volume LFO. This is the number of milliseconds before the LFO takes effect.

`kQTMSKnobVolumeLFODepthFromWheelID`

Specifies the extent to which a synthesizer's modulation wheel (or the MIDI messages it generates) controls the depth of the volume LFO. The value of this knob is multiplied by the modulation wheel value (a value between 0 to 1), and the result is added to the volume LFO depth specified by the `kQTMSKnobVolumeLFODepthID` knob. Modulation wheel controllers and the MIDI messages they generate are most often used to create vibrato and tremolo effects.

`kQTMSKnobVolumeLFODepthID`

Specifies the depth of the volume LFO. This is the amount, expressed as a percentage, by which the volume is altered by the LFO. A value of 0 does not change the volume. A value of 100 changes the volume from complete silence to twice the volume specified by the envelope, with one exception: if the square up waveform is used for the LFO, the normal envelope volume is the minimum volume.

`kQTMSKnobVolumeLFOPeriodID`

Specifies the period for the volume LFO. This is the

wavelength of the LFO in milliseconds. (The LFO rate in Hz is 1000 / `kQTMSKnobPitchLFOPeriodID`).

`kQTMSKnobVolumeLFORampTimeID`

Specifies the ramp-up time for the volume LFO. This is the number of milliseconds after the LFO delay has elapsed that it takes for the LFO to reach full effect.

`kQTMSKnobVolumeLFOShapeID`

Specifies the waveform used for the LFO. The available waveforms are sine, triangle, sawtooth up, sawtooth down, square up, square up-and-down, and random. The sine and triangle shapes both produce a smooth rise and fall of the volume. The sawtooth up produces a gradual increase in volume followed by a sudden fall. The sawtooth down shape produces a sudden increase in volume, followed by a gradual reduction (often heard as a “ting” sound). The square up and square up-and-down shapes apply a sudden pulsing to the volume; the square up only makes the sound louder, while the up-and-down variant makes the sound louder and softer. The random shape applies random changes to the volume, once per LFO period.

`kQTMSKnobVolumeLFOStereoID`

If the synthesizer is producing stereo output and the value of this knob is 1, the LFO is applied in phase to one of the stereo channels and 180° out of phase to the other. This often causes a “vibration” effect within the stereo field.

`kQTMSKnobVolumeOverallID`

Specifies the overall volume of the instrument, in decibels. Increasing the value by 6 doubles the maximum amplitude of the signal, increasing the value by 12 quadruples it, and so on.

`kQTMSKnobVolumeReleaseKeyScalingID`

Specifies the release-time key scaling. Modifies the release time based on the key pitch.

`kQTMSKnobVolumeReleaseTimeID`

Specifies the release time of the volume envelope. This is the number of milliseconds it takes for the sound to soften down to silence after the key is released.

`kQTMSKnobVolumeSustainLevelID`

Specifies the sustain level of the volume envelope. This is

the percentage of full volume that a note is initially played at after the decay time has elapsed.

`kQTMSKnobVolumeVelocity127ID`
To be provided

`kQTMSKnobVolumeVelocity16ID`
To be provided

`kQTMSKnobVolumeVelocity32ID`
To be provided

`kQTMSKnobVolumeVelocity64ID`
To be provided

`kQTMSKnobVolumeVelocity96ID`
To be provided

Controller Numbers

The controller numbers used by QuickTime are mostly identical to the standard MIDI controller numbers. These are signed 8.8 values. The full range, therefore, is -128.00 to 127+127/128 (or 0x8000 to 0x7FFF).

All controls default to zero except for volume and pan.

Pitch bend is specified in fractional semitones, which eliminates the restrictions of a pitch bend range. You can bend as far as you want, any time you want.

The last 16 controllers (113–128) are global controllers. Global controllers respond when the part number is given as 0, indicating the entire synthesizer.

```
enum {
    kControllerModulationWheel          = 1,
    kControllerBreath                   = 2,
    kControllerFoot                     = 4,
    kControllerPortamentoTime           = 5,
    kControllerVolume                   = 7,
    kControllerBalance                  = 8,
    kControllerPan                      = 10,
    kControllerExpression               = 11,
    kControllerLever1                   = 16,
    kControllerLever2                   = 17,
    kControllerLever3                   = 18,
    kControllerLever4                   = 19,
```

```
    kControllerLever5                   = 80,
    kControllerLever6                   = 81,
    kControllerLever7                   = 82,
    kControllerLever8                   = 83,
    kControllerPitchBend                = 32,
    kControllerAfterTouch               = 33,
    kControllerSustain                  = 64,
    kControllerSostenuto                = 66,
    kControllerSoftPedal                = 67,
    kControllerReverb                   = 91,
    kControllerTremolo                  = 92,
    kControllerChorus                   = 93,
    kControllerCeleste                  = 94,
    kControllerPhaser                   = 95,
    kControllerEditPart                 = 113,
    kControllerMasterTune               = 114
};
```

Constant descriptions

`kControllerModulationWheel`
: This controller controls the modulation wheel. A modulation wheel adds a periodic change to the volume or pitch of a sounding tone, depending on the modulation depth knobs.

`kControllerBreath` This controller controls breath.

`kControllerFoot` This controller controls the foot pedal.

`kControllerPortamentoTime`
: This controller adjusts the slur between notes. Set the time to 0 to turn off portamento; there is no separate control to turn portamento on and off.

`kControllerVolume` This controller controls volume.

`kControllerBalance` This controller controls balance between channels.

`kControllerPan` This controller controls balance on the QuickTime music synthesizer and some others. Values are 256–512, corresponding to left to right.

`kControllerExpression`
: This controller provides a second volume control.

`kControllerLever1` through `kControllerLever8`
: These are all general-purpose controllers.

`kControllerPitchBend`
This controller bends the pitch. Pitch bend is specified in positive and negative semitones, with 7 bits per fraction.

`kControllerAfterTouch`
This controller controls channel pressure.

`kControllerSustain` This controller controls the sustain effect. The value is a Boolean—positive for on, 0 or negative for off.

`kControllerSostenuto`
This controller controls sostenuto.

`kControllerSoftPedal`
This controller controls the soft pedal.

`kControllerReverb` This controller controls reverb.

`kControllerTremolo` This controller controls tremolo.

`kControllerChorus` This controller controls the amount of signal to feed to the chorus special effect unit.

`kControllerCeleste` This controller controls the amount of signal to feed to the celeste special effect unit.

`kControllerPhaser` This controller controls the amount of signal to feed to the phaser special effect unit.

`kControllerEditPart`
This controller sets the part number for which editing is occurring. For synthesizers that can edit only one part.

`kControllerMasterTune`
This controller offsets the entire synthesizer in pitch.

Controller Range

These constants specify the maximum and minimum values for controllers.

```
enum {
    kControllerMaximum      = 0x7FFF,
    kControllerMinimum      = 0x8000
};
```

Constant descriptions

`kControllerMaximum`
The maximum value a controller can be set to.

`kControllerMinimum`
: The minimum value a controller can be set to.

Drum Kit Numbers

These constants specify the first and last drum kit numbers available to General MIDI drum kits.

```
enum {
    kFirstDrumkit   = 16384,
    kLastDrumkit    = (kFirstDrumkit + 128)
};
```

Constant description

`kFirstDrumkit`
: The first number in the range of drum kit numbers, which corresponds to “no drum kit.” The standard drum kit is `kFirstDrumKit+1=16385`.

`kLastDrumkit`
: The last number in the range of drum kit numbers.

Tone Fit Flags

These flags are returned by the `MusicFindTone` function (page 133) to indicate how well an instrument matches the tone description.

```
enum {
    kInstrumentMatchSynthesizerType     = 1,
    kInstrumentMatchSynthesizerName     = 2,
    kInstrumentMatchName                = 4,
    kInstrumentMatchNumber              = 8,
    kInstrumentMatchGMNumber            = 16
};
```

Constant descriptions

`kInstrumentMatchSynthesizerType`
: The requested synthesizer type was found.

`kInstrumentMatchSynthesizerName`
: The particular instance of the synthesizer requested was found.

`kInstrumentMatchName`
: The instrument name in the tone description matched an appropriate instrument on the synthesizer.

`kInstrumentMatchNumber`
: The instrument number in the tone description matched an appropriate instrument on the synthesizer.

`kInstrumentMatchGMNumber`
: The General MIDI equivalent was used to find an appropriate instrument on the synthesizer.

Knob Flags

Knob flags specify characteristics of a knob. They are used in the `flags` field of a knob description structure. Some flags describe the type of values a knob takes and others describe the user interface. Knob flags are mutually exclusive, so only one should be set (all knob flag constants begin "`kKnobType`").

```
enum {
    kKnobReadOnly               = 16,
    kKnobInterruptUnsafe        = 32,
    kKnobKeyrangeOverride       = 64,
    kKnobGroupStart             = 128,
    kKnobFixedPoint8            = 1024,
    kKnobFixedPoint16           = 2048,
    kKnobTypeNumber             = 0 << 12,
    kKnobTypeGroupName          = 1 << 12,
    kKnobTypeBoolean            = 2 << 12,
    kKnobTypeNote               = 3 << 12,
    kKnobTypePan                = 4 << 12,
    kKnobTypeInstrument         = 5 << 12,
    kKnobTypeSetting            = 6 << 12,
    kKnobTypeMilliseconds       = 7 << 12,
    kKnobTypePercentage         = 8 << 12,
    kKnobTypeHertz              = 9 << 12,
    kKnobTypeButton             = 10 << 12
};
```

Constant descriptions

`kKnobReadOnly` The knob value cannot be changed by the user or with a set knob call.

`kKnobInterruptUnsafe` Alter this knob only from foreground task time.

`kKnobKeyrangeOverride` The knob can be overridden within a single key range (software synthesizer only).

`kKnobGroupStart` The knob is first in some logical group of knobs.

`kKnobFixedPoint8` Interpret knob numbers as fixed-point 8-bit.

`kKnobFixedPoint16` Interpret knob numbers as fixed-point 16-bit.

`kKnobTypeNumber` The knob value is a numerical value.

`kKnobTypeGroupName` The name of the knob is really a group name for display purposes.

`kKnobTypeBoolean` The knob is an on/off knob. If the range of the knob (as specified by the low value and high value in the knob description structure) is greater than one, the knob is a multi-checkbox field.

`kKnobTypeNote` The knob value range is equivalent to MIDI keys.

`kKnobTypePan` The knob value is the pan setting and is within a range (as specified by the low value and high value in the knob description structure) that goes from left to right.

`kKnobTypeInstrument` The knob value is a reference to another instrument number.

`kKnobTypeSetting` The knob value is one of *n* different discrete settings—for example, items on a pop-up menu.

`kKnobTypeMilliseconds` The knob value is in milliseconds.

`kKnobTypePercentage` The knob value is a percentage of the range.

`kKnobTypeHertz` The knob value represents frequency.

`kKnobTypeButton` The knob is a momentary trigger push button.

Knob Value Constants

These constants specify unknown or default knob values and are used in various get knob and set knob calls.

```
enum {
    kUnknownKnobValue       = 0x7FFFFFFF,
    kDefaultKnobValue       = 0x7FFFFFFE
};
```

Constant descriptions

`kUnknownKnobValue` Couldn't find the specified knob value.

`kDefaultKnobValue` Set this knob to its default value.

Music Packet Status

These constants are used in the `reserved` field of the MIDI packet structure (page 79).

```
enum {
    kMusicPacketPortLost        = 1,
    kMusicPacketPortFound       = 2,
    kMusicPacketTimeGap         = 3
};
```

Constant descriptions

`kMusicPacketPortLost`
The application has lost the default input port.

`kMusicPacketPortFound`
The application has retrieved the input port from the previous owner.

`kMusicPacketTimeGap`
The last byte of the packet specifies how long (in milliseconds) to keep the MIDI line silent after sending the packet.

Atomic Instrument Information Flags

These constants specify what pieces of information about an atomic instrument the caller is interested in and are passed to the `MusicGetPartAtomicInstrument` function.

```
enum {
    kGetAtomicInstNoExpandedSamples = 1 << 0,
    kGetAtomicInstNoOriginalSamples = 1 << 1,
    kGetAtomicInstNoSamples         = kGetAtomicInstNoExpandedSamples |
        kGetAtomicInstNoOriginalSamples,
    kGetAtomicInstNoKnobList        = 1 << 2,
    kGetAtomicInstNoInstrumentInfo  = 1 << 3,
    kGetAtomicInstOriginalKnobList  = 1 << 4,
    kGetAtomicInstAllKnobs          = 1 << 5
};
```

Constant descriptions

`kGetAtomicInstNoExpandedSamples`
Eliminate the expanded samples.

`kGetAtomicInstNoOriginalSamples`
Eliminate the original samples.

`kGetAtomicInstNoSamples`
Eliminate both the original and expanded samples.

`kGetAtomicInstNoKnobList`
Eliminate the knob list.

`kGetAtomicInstNoInstrumentInfo`
Eliminate the About box information.

`kGetAtomicInstOriginalKnobList`
Include the original knob list.

`kGetAtomicInstAllKnobs`
Include the current knob list.

Flags for Setting Atomic Instruments

These flags specify details of initializing a part with an atomic instrument and are passed to the `MusicSetPartAtomicInstrument` function (page 146).

```
enum {
    kSetAtomicInstKeepOriginalInstrument    = 1 << 0,
    kSetAtomicInstShareAcrossParts          = 1 << 1,
    kSetAtomicInstCallerTosses              = 1 << 2,
    kSetAtomicInstDontPreprocess            = 1 << 7
};
```

Constant descriptions

`kSetAtomicInstKeepOriginalInstrument`
Keep original sample after expansion.

`kSetAtomicInstShareAcrossParts`
Remove the instrument when the application quits.

`kSetAtomicInstCallerTosses`
The caller isn't keeping a copy of the atomic instrument for later calls to `NASetAtomicInstrument`.

`kSetAtomicInstDontPreprocess`
Don't expand the sample. You would only set this bit if you know the instrument is digitally clean or you got it from a `MusicGetPartAtomicInstrument` call (page 146).

Instrument Info Flags

Use these flags in the `MusicGetInstrumentInfo` function (page 148) and `InstrumentGetInfo` function (page 158) to indicate which instruments and instrument names you are interested in.

```
enum {
    kGetInstrumentInfoNoBuiltIn         = 1 << 0,
    kGetInstrumentInfoMidiUserInst      = 1 << 1,
    kGetInstrumentInfoNoIText           = 1 << 2
};
```

Constant descriptions

`kGetInstrumentInfoNoBuiltIn`
Don't return built-in instruments.

`kGetInstrumentInfoMidiUserInst`
Do return user instruments for a MIDI device.

`kGetInstrumentInfoNoIText`
: Don't return international text strings.

Synthesizer Connection Type Flags

These flags provide information about a MIDI device's connection and are used in the synthesizer connections structure (page 84).

```
enum {
    kSynthesizerConnectionMono      = 1,
    kSynthesizerConnectionMMgr      = 2,
    kSynthesizerConnectionOMS       = 4,
    kSynthesizerConnectionQT        = 8,
    kSynthesizerConnectionFMS       = 16
};
```

Constant descriptions

`kSynthesizerConnectionMono`
: If set, and the synthesizer can be both monophonic and polyphonic, the synthesizer is instructed to take up its channels sequentially from the system channel in monophonic mode.

`kSynthesizerConnectionMMgr`
: This connection is imported from the MIDI Manager.

`kSynthesizerConnectionOMS`
: This connection is imported from the Open Music System (OMS).

`kSynthesizerConnectionQT`
: This connection is a QuickTime-only port.

`kSynthesizerConnectionFMS`
: This connection is imported from the FreeMIDI system.

Instrument Match Flags

These flags are returned in the `instMatch` field of the General MIDI instrument information structure (page 81) to specify how QuickTime music architecture matched an instrument request to an instrument.

```
enum {
    kInstrumentExactMatch               = 0x00020000,
    kInstrumentRecommendedSubstitute    = 0x00010000,
    kInstrumentQualityField             = 0xFF000000,
    kRoland8BitQuality          = 0x05000000
};
typedef InstrumentAboutInfo *InstrumentAboutInfoPtr;
typedef InstrumentAboutInfoPtr *InstrumentAboutInfoHandle;
```

Constant descriptions

`kInstrumentExactMatch`
The instrument exactly matches the request.

`kInstrumentRecommendedSubstitute`
The instrument is the approved substitute.

`kInstrumentQualityField`
The high-order 8 bits of this field specify the quality of the selected instrument. Higher values specify higher quality.

`kRoland8BitQuality`
For built-in instruments, the value of the high-order 8 bits is always `kInstrumentRoland8BitQuality`, which corresponds to the quality of an 8-bit Roland instrument.

Note Request Constants

These flags specify what to do if the exact instrument requested is not found. They are used in the `flags` field of the note request information structure (page 85).

```
enum {
    kNoteRequestNoGM            = 1,
    kNoteRequestNoSynthType     = 2
};
```

Constant descriptions

`kNoteRequestNoGM` Don’t use a General MIDI synthesizer.

`kNoteRequestNoSynthType`
Don’t use another synthesizer of the same type but with a different name.

Pick Instrument Flags

The pick instrument flags provide information to the `NAPickInstrument` (page 120) and `NAPickEditInstrument` (page 122) functions on which instruments to present for the user to choose from.

```
enum {
    kPickDontMix                = 1,
    kPickSameSynth              = 2,
    kPickUserInsts              = 4,
    kPickEditAllowPick          = 16
};
```

Constant descriptions

`kPickDontMix` Show either all drum kits or all instruments depending on the current instrument. For example, if it's a drum kit, show only drum kits.

`kPickSameSynth` Show only instruments from the current synthesizer.

`kPickUserInsts` Show modifiable instruments in addition to ROM instruments.

`kPickEditAllowPick`
Present the instrument picker dialog box. Used only with the `NAPickEditInstrument` function.

Note Allocator Type

Use these constants to specify the QuickTime note allocator component.

```
enum {
    kNoteAllocatorType              = 'nota'
    kNoteAllocatorComponentType     = 'not2'
};
```

Constant description

`kNoteAllocatorType`
The QTMA note allocator type.

`kNoteAllocatorComponentType`
The QTMA note allocator component type.

Tune Queue Depth

This constant represents the maximum number of segments that can be queued with the `TuneQueue` function (page 91).

```
enum {
    kTuneQueueDepth     = 8
};
```

Constant description

`kTuneQueueDepth` Deepest you can queue tune segments.

Tune Player Type

Use this constant to specify the QuickTime tune player component.

```
enum {
    kTunePlayerType         = 'tune'
};
```

Constant descriptions

`kTunePlayerType` The QuickTime music architecture tune player component type.

Tune Queue Flags

Use these flags in the `TuneQueue` function (page 91) to give details about how to handle the queued tune.

```
enum {
    kTuneStartNow               = 1,
    kTuneDontClipNotes          = 2,
    kTuneExcludeEdgeNotes       = 4,
    kTuneQuickStart             = 8,
    kTuneLoopUntil              = 16,
    kTuneStartNewMaster         = 16384
};
```

Constant descriptions

`kTuneStartNow` Play even if another tune is playing.

`kTuneDontClipNotes`
Allow notes to finish their durations outside sample.

`kTuneExcludeEdgeNotes`
Don't play notes that start at end of tune.

`kTuneQuickStart` Leave all the controllers where they are and ignore start time.

`kTuneLoopUntil` Loop a queued tune if there is nothing else in the queue.

`kTuneStartNewMaster`
Start a new master reference timer.

MIDI Component Constants

Use these constants to specify MIDI components.

```
enum {
    kQTMIDIComponentType= FOUR_CHAR_CODE('midi')
};

enum {
    kOMSComponentSubType= FOUR_CHAR_CODE('OMS '),
    kFMSComponentSubType= FOUR_CHAR_CODE('FMS '),
    kMIDIManagerComponentSubType = FOUR_CHAR_CODE('mmgr')
};
```

Constant descriptions

`kQTMIDIComponentType`
The component type for MIDI components.

`kOMSComponentSubType`
The component subtype for a Open Music System MIDI component.

`kFMSComponentSubType`
The component subtype for a FreeMIDI component.

`kMIDIManagerComponentSubType`
The component subtype for a MIDI Manager component.

MIDI System Exclusive Constants

System exclusive constants can be used to control where sample breaks occur when importing a MIDI file. For more information, see the section "Importing a Standard MIDI File As a Movie Using the Movie Toolbox" (page 34).

```
enum {
    kAppleSysexID   = 0x11,
    kAppleSysexCmdSampleSize= 0x0001,
    kAppleSysexCmdSampleBreak= 0x0002,
    kAppleSysexCmdAtomicInstrument = 0x0010,
    kAppleSysexCmdDeveloper= 0x7F00
};
```

MIDI File Import Flags

These flags control the importation of MIDI files.

```
enum {
    kMIDIImportSilenceBefore = 1 << 0,
    kMIDIImportSilenceAfter = 1 << 1,
    kMIDIImport20Playable = 1 << 2,
    kMIDIImportWantLyrics = 1 << 3
};
```

Constant descriptions

`kMIDIImportSilenceBefore`
: Specifies to add one second of silence before the first note.

`kMIDIImportSilenceAfter`
: Specifies to add one second of silence after the last note.

`kMIDIImport20Playable`
: Specifies to import only MIDI data that can be used with QuickTime 2.0. The imported data does not include program changes and has at most 32 parts.

`kMIDIImportWantLyrics`
: Specifies to import karaoke lyrics as a text track.

Part Mixing Flags

Part mixing flags control how a part is mixed with other parts.

```
enum {
    kTuneMixMute= 1,
    kTuneMixSolo= 2
};
```

Constant descriptions

`kTuneMixMute`	Disables the part so that it is not heard.
`kTuneMixSolo`	Specifies to include only soloed parts in the mix if any parts are soloed.

Data Structures

This section describes the data structures provided by QuickTime music architecture.

Instrument Knob Structure

An instrument knob structure contains information about an instrument knob. It is defined by the `InstKnobRec` data type.

```
struct InstKnobRec {
    long                        number;
    long                        value;
};
typedef struct InstKnobRec InstKnobRec;
```

Field descriptions

`number`	A knob ID or index. A nonzero value in the high byte indicates that it is an ID. The knob index ranges from 1 to the number of knobs; the ID is an arbitrary number.
`value`	The value the knob is set to.

Instrument Knob List

An instrument knob list contains a list of sound parameters. It is defined by the `InstKnobList` data type.

```
struct InstKnobList {
    long                        knobCount;
    long                        knobFlags;
    InstKnobRec                 knob[1];
};
typedef struct InstKnobList InstKnobList;
```

Field descriptions

`knobCount`	The number of instrument knob structures in the list.
`knobFlags`	Instructions on what to do if a requested knob is not in the list. See “Instrument Knob Flags” (page 41).
`knob[1]`	An array of instrument knob structures.

Atomic Instrument Sample Description Structure

A sample description structure contains a description of an audio sample, including sample rate, loop points, and lowest and highest key to play on. It is defined by the `InstSampleDescRec` data type.

```
struct InstSampleDescRec {
    OSType                      dataFormat;
    short                       numChannels;
    short                       sampleSize;
    UnsignedFixed               sampleRate;
    short                       sampleDataID;
    long                        offset;
    long                        numSamples;
    long                        loopType;
    long                        loopStart;
    long                        loopEnd;
    long                        pitchNormal;
    long                        pitchLow;
    long                        pitchHigh;
};
typedef struct InstSampleDescRec InstSampleDescRec;
```

Field descriptions

`dataFormat`	The data format type. This is either `'twos'` for signed data or `'raw '` for unsigned data.
`numChannels`	The number of channels of data present in the sample.
`sampleSize`	The size of the sample— 8-bit or 16-bit.
`sampleRate`	The rate at which to play the sample in unsigned fixed-point 16.16.
`sampleDataID`	The ID number of a sample data atom that contains the sample audio data.
`offset`	Set to 0.
`numSamples`	The number of data samples in the sound.
`loopType`	The type of loop. See “Loop Type Constants” (page 41).
`loopStart`	Indicates the beginning of the portion of the sample that is looped if the sound is sustained. The position is given in the number of data samples from the start of the sound.
`loopEnd`	Indicates the end of the portion of the sample that is looped if the sound is sustained. The position is given in the number of data samples from the start of the sound.
`pitchNormal`	The number of the MIDI note produced if the sample is played at the rate specified in `sampleRate`.
`pitchLow`	The lowest pitch at which to play the sample. Use for instruments, such as pianos, that have different samples to use for different pitch ranges.
`pitchHigh`	The highest pitch at which to play the sample. Use for instruments, such as pianos, that have different samples to use for different pitch ranges.

Synthesizer Description Structure

A synthesizer description structure contains information about a synthesizer. It is defined by the `SynthesizerDescription` data type.

```
struct SynthesizerDescription {
    OSType                      synthesizerType;
    Str31                       name;
    unsigned long               flags;
    unsigned long               voiceCount;
```

```
    unsigned long               partCount;
    unsigned long               instrumentCount;
    unsigned long               modifiableInstrumentCount;
    unsigned long               channelMask;
    unsigned long               drumPartCount;
    unsigned long               drumCount;
    unsigned long               modifiableDrumCount;
    unsigned long               drumChannelMask;
    unsigned long               outputCount;
    unsigned long               latency;
    unsigned long               controllers[4];
    unsigned long               gmInstruments[4];
    unsigned long               gmDrums[4];
};
typedef struct SynthesizerDescription SynthesizerDescription;
```

Field descriptions

`synthesizerType` The synthesizer type. This is the same as the music component subtype.

`name` Text name of the synthesizer type.

`flags` Various information about how the synthesizer works. See “Synthesizer Description Flags” (page 42).

`voiceCount` Maximum polyphony.

`partCount` Maximum multi-timbrality (and MIDI channels).

`instrumentCount` The number of built-in ROM instruments. This does not include General MIDI instruments.

`modifiableInstrumentCount` The number of slots available for saving user-modified instruments.

`channelMask` Which channels a MIDI device always uses for instruments. Set to `FFFF` for all channels.

`drumPartCount` The maximum multi-timbrality of drum parts. For synthesizers where drum kits are separated from instruments.

`drumCount` The number of built-in ROM drum kits. This does not include General MIDI drum kits. For synthesizers where drum kits are separated from instruments

modifiableDrumCount
: The number of slots available for saving user-modified drum kits. For MIDI synthesizers where drum kits are separated from instruments

`drumChannelMask`
: Which channels a MIDI device always uses for drum kits. Set to `FFFF` for all channels

`outputCount`
: The number of audio outputs. This is usually two.

`latency`
: Response time in microseconds.

`controllers[4]`
: An array of 128 bits identifying the available controllers. See “Controller Numbers” (page 56). Bits are numbered from 1 to 128, starting with the most significant bit of the long word, and continuing to the least significant of the last bit.

`gmInstruments[4]`
: An array of 128 bits giving the available General MIDI instruments.

`gmDrums[4]`
: An array of 128 bits giving the available General MIDI drum kits.

Tone Description Structure

A tone description structure provides the information needed to produce a specific musical sound. The tune header has a tone description for each instrument used. Tone descriptions are also used in the tone description atoms of atomic instruments. The tone description structure is defined by the `ToneDescription` data type.

```
struct ToneDescription {
    BigEndianOSType             synthesizerType;
    Str31                       synthesizerName;
    Str31                       instrumentName;
    BigEndianLong               instrumentNumber;
    BigEndianLong               gmNumber;
};
typedef struct ToneDescription ToneDescription;
```

Field descriptions

`synthesizerType` The synthesizer type. See “Synthesizer Type Constants” (page 42) for possible types. A value of `0` specifies that any type of synthesizer is acceptable.

`synthesizerName` The name of the synthesizer component instance. A value of `0` specifies that the name can be ignored.

`instrumentName` The name of the instrument to use.

`instrumentNumber` The instrument number of the instrument to use. This value, which must be in the range 1–262143, can specify General MIDI and GS instruments as well as other instruments (see Table 2-2). The instrument specified by this field is used if it is available; if not, the instrument specified by the `gmNumber` field is used. If neither of the instruments specified by the `instrumentNumber` or `gmNumber` fields is available, the instrument specified by the `instrumentName` field is used. Finally, if none of these fields specifies an instrument that is available, no tone is played.

`gmNumber` The instrument number of a General MIDI or GS instrument to use if the instrument specified by the `instrumentNumber` field is not available. This value, which must be in the range 1–16383, can specify only General MIDI and GS instruments (see Table 2-2). The instrument specified by the `instrumentNumber` field is used if it is available; if not, the instrument specified by the `gmNumber` field is used. If neither of the instruments specified by the `instrumentNumber` or `gmNumber` fields is available, the instrument specified by the `instrumentName` field is used. Finally, if none of these fields specifies an instrument that is available, no tone is played.

GS instruments conform to extensions defined by Roland Corporation to the General MIDI specifications. For information about these extensions, see

`http://www.rolandcorp.com/vsc/gs1.html`

on the World Wide Web.

Table 2-1

Name	Low	High	Low (Hex)	High (Hex)
GM Instrument	1	128	0x00000001	0x00000080
GM Drumkit	16385	16512	0x00004001	0x00004080
GS Instrument	128	16383	0x00000081	0x00003FFF
ROM Instrument	32768	65535	0x00008000	0x0000FFFF
User Instrument	65536	131071	0x00010000	0x0001FFFF
Internal Index	131072	262143	0x00020000	0x0003FFFF
All Other Numbers Illegal And Reserved				

Table 2-2 IRange descriptions

GM instrument	An instrument number in this range specifies a standard General MIDI instrument that should sound the same on all synthesizers that support General MIDI.
GM drum kit	An instrument number in this range specifies a standard General MIDI drum kit instrument that should sound the same on all synthesizers that support General MIDI.
GS instrument	An instrument number in this range specifies a standard GS instrument that should sound the same on all synthesizers that support the Roland GS extensions to General MIDI.
ROM instrument	An instrument number in this range specifies an instrument of a synthesizer that not a standard General MIDI or GS instrument.
User instrument	Instruments number in this range are transient and are assigned when necessary for additional instruments, such as instruments in a newly installed GS library or custom instruments for a game. Applications should refer to these additional instruments by name rather by number.
Internal index	An instrument index value returned by the `MusicFindTone` function that can be passed immediately in a call to `MusicSetPartInstrumentNumber`. Values in this range are not

persistent and should never be stored or used in any other way.

Knob Description Structure

A knob description structure contains sound parameter values for a single knob. It is defined by the `KnobDescription` data type.

```
struct KnobDescription {
    Str63                       name;
    long                        lowValue;
    long                        highValue;
    long                        defaultValue;
    long                        flags;
    long                        knobID;
};
typedef struct KnobDescription KnobDescription;
```

Field descriptions

`name`	The name of the knob.
`lowValue`	The lowest number you can set the knob to.
`highValue`	The highest number you can set the knob to.
`defaultValue`	A value to use for the default.
`flags`	Various information about the knob. See “Knob Flags” (page 60).
`knobID`	A knob ID or index. A nonzero value in the high byte indicates that it is an ID. The knob index ranges from 1 to the number of knobs; the ID is an arbitrary number. Use the knob ID to refer to the knob in preference to the knob index, which may change.

Instrument About Information

The instrument About information structure contains the information that appears in the instrument’s About box and is returned by the `MusicGetInstrumentAboutInfo` function (page 148). It is defined by the `InstrumentAboutInfo` data type.

```
struct InstrumentAboutInfo {
    PicHandle                   p;
    Str255                      author;
    Str255                      copyright;
    Str255                      other;
};
typedef struct InstrumentAboutInfo InstrumentAboutInfo;
```

Field descriptions

`p`	A handle to a graphic for the About box.
`author`	The author's name.
`copyright`	The copyright information.
`other`	Any other textual information.

MIDI Packet

The MIDI packet structure describes the data passed by note allocation calls. It is defined by the `MusicMIDIPacket` data type.

```
struct MusicMIDIPacket {
    unsigned short          length;
    unsigned long           reserved;
    UInt8                   data[249];
};
typedef struct MusicMIDIPacket MusicMIDIPacket;
```

Field descriptions

`length`	The length of the data in the packet.
`reserved`	This field contains zero or one of the music packet status constants. See "Music Packet Status" (page 62).
`data[249]`	The MIDI data.

Note
This is the count of data bytes only, unlike MIDI Manager or OMS packets.

Instrument Information Structure

The instrument information structure provides identifiers for instruments and is part of the instrument information list. It is defined by the `InstrumentInfoRecord` data type.

```
struct InstrumentInfoRecord {
    long                        instrumentNumber;
    long                        flags;
    long                        toneNameIndex;
    long                        itxtNameAtomID;
};
typedef struct InstrumentInfoRecord InstrumentInfoRecord;
```

Field descriptions

`instrumentNumber` The instrument number. If the number is 0, the name is an instrument category. See Table 2-2 (page 77) for the ranges of instrument numbers. If the value of the instrument number is greater than 65536, its value is transient, and the instrument should be identified by name rather than by number except when the value is immediately passed to the `MusicSetPartInstrumentNumber` function.

`flags` Unused. Must be 0

`toneNameIndex` The instrument's position in the `toneNames` index stored in the instrument information list this structure is a part of. The index is a one-based index.

`itxtNameAtomID` The instrument's position in the `itxtNames` index stored in the instrument information list this structure is a part of.

Instrument Information List

An instrument information list contains the list of instruments available on a synthesizer. It is defined by the `InstrumentInfoList` data type.

```
struct InstrumentInfoList {
    long                        recordCount;
    Handle                      toneNames;
    QTAtomContainer             itxtNames;
    InstrumentInfoRecord        info[1];
```

```
};
typedef struct InstrumentInfoList InstrumentInfoList;
typedef InstrumentInfoList *InstrumentInfoListPtr;
typedef InstrumentInfoListPtr *InstrumentInfoListHandle;
```

Field descriptions

`recordCount`	The number of structures in the list.
`toneNames`	A string list of the instrument names as specified in their tone descriptions.
`itxtNames`	A list of international text names, taken from the name atoms.
`info[1]`	An array of instrument information structures.

General MIDI Instrument Information Structure

The General MIDI instrument information structure provides information about a General MIDI instrument within an instrument component. It is defined by the `GMInstrumentInfo` data type.

```
struct GMInstrumentInfo {
    long                        cmpInstID;
    long                        gmInstNum;
    long                        instMatch;
};
typedef struct GMInstrumentInfo GMInstrumentInfo;
typedef GMInstrumentInfo *GMInstrumentInfoPtr;
typedef GMInstrumentInfoPtr *GMInstrumentInfoHandle;
```

Field descriptions

`cmpInstID`	The number of the instrument within the instrument component.
`gmInstNum`	The General MIDI, or standard, instrument number.
`instMatch`	A flag indicating how the instrument matches the requested instrument. See “Instrument Match Flags” (page 65).

Non-General MIDI Instrument Information Structure

The non–General MIDI information structure provides information about non-General MIDI instruments within an instrument component. It is defined by the `nonGMInstrumentInfoRecord` data type.

```
struct nonGMInstrumentInfoRecord {
    long                        cmpInstID;
    long                        flags;
    long                        toneNameIndex;
    long                        itxtNameAtomID;
};
typedef struct nonGMInstrumentInfoRecord nonGMInstrumentInfoRecord;
```

Field descriptions

`cmpInstID` The number of the instrument within the instrument component. If the ID is 0, the name is a category name.

`flags` Not used.

`toneNameIndex` The instrument's position in the `toneNames` index stored in the instrument information list this structure is a part of. The index is a one-based index.

`itxtNameAtomID` The instrument's position in the `itxtNames` index stored in the instrument information list this structure is a part of.

Non–General MIDI Instrument Information List

A non–General MIDI instrument information list contains the list of non–General MIDI instruments supported by an instrument component. It is defined by the `nonGMInstrumentInfo` data type.

```
struct nonGMInstrumentInfo {
    long                        recordCount;
    Handle                      toneNames;
    QTAtomContainer             itxtNames;
    nonGMInstrumentInfoRecord   instInfo[1];
};
typedef struct nonGMInstrumentInfo nonGMInstrumentInfo;
typedef nonGMInstrumentInfo *nonGMInstrumentInfoPtr;
typedef nonGMInstrumentInfoPtr *nonGMInstrumentInfoHandle;
```

Field descriptions

`recordCount`	Number of structures in the list.
`toneNames`	A short string list of the instrument names as specified in their tone descriptions.
`itxtNames`	A list of international text names, taken from the name atoms.
`instInfo[1]`	An array of non–General MIDI instrument information structures.

Complete Instrument Information List

The complete instrument information list contains a list of all atomic instruments supported by an instrument component. It is defined by the `InstCompInfo` data type.

```
struct InstCompInfo {
    long                        infoSize;
    long                        GMinstrumentCount;
    GMInstrumentInfoHandle      GMinstrumentInfo;
    long                        GMdrumCount;
    GMInstrumentInfoHandle      GMdrumInfo;
    long                        nonGMinstrumentCount;
    nonGMInstrumentInfoHandle   nonGMinstrumentInfo;
    long                        nonGMdrumCount;
    nonGMInstrumentInfoHandle   nonGMdrumInfo;
};
typedef struct InstCompInfo InstCompInfo;
typedef InstCompInfo *InstCompInfoPtr;
typedef InstCompInfoPtr *InstCompInfoHandle;
```

Field descriptions

`infoSize`	The size of this structure in bytes.
`GMinstrumentCount`	The number of General MIDI instruments.
`GMinstrumentInfo`	A handle to a list of General MIDI instrument information structures.
`GMdrumCount`	The number of General MIDI drum kits.
`GMdrumInfo`	A handle to a list of General MIDI instrument information structures.

`nonGMinstrumentCount`
The number of non–General MIDI instruments.

`nonGMinstrumentInfo`
A handle to the list of non–General MIDI instruments.

`nonGMdrumCount` The number of non–General MIDI drum kits.

`nonGMdrumInfo` A handle to the list of non–General MIDI drum kits.

Synthesizer Connections for MIDI Devices

The synthesizer connection structure describes how a MIDI device is connected to the computer. It is defined by the `SynthesizerConnections` data type.

```
struct SynthesizerConnections {
    OSType                      clientID;
    OSType                      inputPortID;
    OSType                      outputPortID;
    long                        midiChannel;
    long                        flags;
    long                        unique;
    long                        reserved1;
    long                        reserved2;
};
typedef struct SynthesizerConnections SynthesizerConnections;
```

Field descriptions

`clientID` The client ID provided by the MIDI Manager or `'OMS '` for an OMS port.

`inputPortID` The ID provided by the MIDI Manager or OMS for the port used to send to the MIDI synthesizer.

`outputPortID` The ID provided by the MIDI Manager or OMS for the port that receives from a keyboard or other control device.

`midiChannel` The system MIDI channel or, for a hardware device, the slot number.

`flags` Information about the type of connection. See “Synthesizer Connection Type Flags” (page 65).

`unique` A unique ID you can use instead of an index to identify the synthesizer to the note allocator.

`reserved1`	Reserved. Set to 0.
`reserved2`	Reserved. Set to 0.

QuickTime MIDI Port

This structure provides information about a MIDI port.

```
struct QTMIDIPort {
    SynthesizerConnections      portConnections;
    Str63                       portName;
};
typedef struct QTMIDIPort QTMIDIPort;
```

Field descriptions

`portConnections`	A synthesizer connections structure (page 84).
`portName`	The name of the output port.

QuickTime MIDI Port List

This structure contains a list of QuickTime MIDI port structures.

```
struct QTMIDIPortList {
    short               portCount;
    QTMIDIPort          port[1];
};
typedef struct QTMIDIPortList QTMIDIPortList;
```

Field descriptions

`portCount`	The number of MIDI ports in the list.
`port`	An array of QuickTime MIDI port structures.

Note Request Information Structure

The note request information structure contains information for allocating a note channel that's in addition to that included in a tone description structure. It is defined by the `NoteRequestInfo` data type.

```
struct NoteRequestInfo {
    UInt8                       flags;
    UInt8                       reserved;
    short                       polyphony;
    Fixed                       typicalPolyphony;
};
typedef struct NoteRequestInfo NoteRequestInfo;
```

Field descriptions

`flags`	Specifies what to do if the exact instrument requested in a tone description structure is not found. See "Note Request Constants" (page 66).
`reserved`	Reserved. Set to 0.
`polyphony`	Maximum number of voices.
`typicalPolyphony`	Hint for level mixing.

Note Request Structure

A note request structure combines a tone description structure and a note request information structure to provide all the information available for allocating a note channel. It is defined by the `NoteRequest` data type.

```
struct NoteRequest {
    NoteRequestInfo             info;
    ToneDescription             tone;
};
typedef struct NoteRequest NoteRequest;
```

Field descriptions

`info`	A note request information structure (page 85).
`tone`	A tone description structure (page 75).

Tune Status

The tune status structure provides information on the currently playing tune.

```
struct TuneStatus {
    unsigned long           tune;
    unsigned long           tunePtr;
    TimeValue               time;
    short                   queueCount;
    short                   queueSpots;
    TimeValue               queueTime;
    long                    reserved[3];
};
typedef struct TuneStatus TuneStatus;
```

Field descriptions

`tune`	The currently playing tune.
`tunePtr`	Current position within the playing tune.
`time`	Current tune time.
`queueCount`	Number of tunes queued up.
`queueSpots`	Number of tunes that can be added to the queue.
`queueTime`	Total amount of playing time represented by tunes in the queue. This value can be very inaccurate.
`reserved[3]`	Reserved. Set to 0.

Functions

The functions provided by the note allocator component, the tune player component, music components, and instrument components are described in the following sections.

Tune Player Functions

This section describes the functions the tune player provides for setting, queueing, and manipulating music sequences. It also describes tune player utility functions.

TuneSetHeader

The `TuneSetHeader` function prepares the tune player to accept subsequent music event sequences by defining one or more parts to be used by sequence Note events.

```
pascal ComponentResult TuneSetHeader(
                    TunePlayer tp,
                    unsigned long *header);
```

`tp` A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`header` A pointer to a list of instruments that will be used in subsequent calls to the `TuneQueue` function. The list can include note request General events with subtypes of `kGeneralEventNoteRequest`, `kGeneralEventPartKey`, `kGeneralEventAtomicInstrument`, `kGeneralEventMIDIChannel`, and `kGeneralEventUsedNotes`. It can also include atomic instruments. The list is terminated by a marker event of subtype end.

function result A result code.

DISCUSSION

The `TuneSetHeader` function is the first QuickTime music architecture call to play a music sequence. The `header` parameter points to one or more initialized General events and atomic instruments. The event list pointed to by the `header` parameter must conclude with a marker event of subtype end.

Only one call to `TuneSetHeader` is required. Each `TuneSetHeader` call resets the tune player.

SEE ALSO

The `TuneSetHeaderWithSize` function (page 89) and the `TuneSetNoteChannels` function (page 89).

TuneSetHeaderWithSize

The `TuneSetHeaderWithSize` function is like the `TuneSetHeader` function in that it prepares the tune player to accept subsequent music event sequences by defining one or more parts to be used by sequence Note events. But unlike the `TuneSetHeader` function, `TuneSetHeaderWithSize` allows you to specify the header length in bytes. This prevents the call from parsing off the end if the music event sequence is missing an end marker.

```
extern pascal ComponentResult TuneSetHeaderWithSize(
                    TunePlayer tp,
                    unsigned long *header,
                    unsigned long size);
```

`tp` A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`header` A pointer to a list of instruments that will be used in subsequent calls to the `TuneQueue` function. The list can include General events with subtypes of `kGeneralEventNoteRequest`, `kGeneralEventPartKey`, `kGeneralEventAtomicInstrument`, `kGeneralEventMIDIChannel`, and `kGeneralEventUsedNotes`. It can also include atomic instruments. The list is terminated by a marker event of subtype end.

`size` The size of the header in bytes.

SEE ALSO

The `TuneSetHeader` function (page 88) and the `TuneSetNoteChannels` function (page 89).

TuneSetNoteChannels

You use the `TuneSetNoteChannels` function to assign note channels to a tune player.

```
extern pascal ComponentResult TuneSetNoteChannels(
                    TunePlayer tp,
                    unsigned long count,
                    NoteChannel *noteChannelList,
                    TunePlayCallBackUPP playCallBackProc,
                    long refCon);
```

`tp` Specifies the instance of a tune player component for this operation. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`count` The number of note channels to assign.

`noteChannelList`
A pointer to the list of note channels to assign.

`playCallBackProc`
A pointer to a function in your software that is called for each event whose part number is greater than the value of the `count` parameter.

`refCon` A reference constant that is passed to the function specified by the `playCallBackProc` parameter whenever it is called.

DISCUSSION

When you call `TuneSetNoteChannels`, any note channels that were previously assigned to the tune player are no longer used and are disposed of.

The parts for the note channels you assign are numbered from 1 to the value of the `count` parameter.

The `playCallBackProc` and `refCon` parameters let you to use the tune player as a general purpose timer / sequencer. The function in your software pointed to by the `playCallBackProc` parameter is called for each event whose part number is greater than the value of the `count` parameter. Events whose part numbers are

less than or equal to the value of the `count` parameter are passed to the note channel rather than the callback procedure.

The `playCallBackProc` parameter must point to a function with the following prototype:

```
typedef pascal void (*TunePlayCallBackProcPtr)(
    unsigned long *event,
    long seed,
    long refCon);
```

The `event` parameter is a pointer to a QuickTime music event structure in the sequence data. The `seed` parameter is a 32-bit value that is guaranteed to be different for each call to the callback routine (unless `2^32` calls are made, after which the values repeat), with one exception: the value passed at the beginning of a note is also passed at the end of the note's duration, together with a note structure or an extended note in which the velocity bits are set to `0`. The `refCon` parameter is the reference constant that is passed to the `TuneSetNoteChannels` function.

TuneQueue

The `TuneQueue` function places a sequence of music events into a queue to be played.

```
pascal ComponentResult TuneQueue(
                    TunePlayer tp,
                    unsigned long *tune,
                    Fixed tuneRate,
                    unsigned long tuneStartPosition,
                    unsigned long tuneStopPosition,
                    unsigned long queueFlags,
                    TuneCallBackUPP callBackProc,
                    long refCon);
```

`tp` A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`tune`	Pointer to an array of events, terminated by a marker event of subtype end.
`tuneRate`	Fixed-point speed at which to play the sequence. "Normal" speed is `0x00010000`.
`tuneStartPosition`	Sequence starting time.
`tuneStopPosition`	Sequence ending time.
`queueFlags`	Flags with details about how to play the queued tunes. For valid values see "Tune Queue Flags" (page 68).
`callBackProc`	Points to your callback function. Your callback function must have the following form: `pascal void MyCallBackProc (QTCallBack cb, long refcon);`
`refcon`	Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.
function result	A result code. In addition to QuickTime music architecture result codes, this function may return `TimeBase` result codes.

DISCUSSION

The `tuneStartPosition` and `tuneStopPosition` parameters specify, in time units numbered from zero for the beginning of the sequence, which part of the queued sequence to play. To play all of it, pass 0 and 0xFFFFFFFF, respectively.

If there is a sequence currently playing, the newly queued sequence begins as soon as the active sequence ends unless the `queueFlags` parameter is `kTuneStartNow`, in which case the currently playing sequence is immediately terminated and the new one started.

TuneStop

The `TuneStop` function stops a currently playing sequence.

```
pascal ComponentResult TuneStop(
                    TunePlayer tp,
                    long stopFlags);
```

`tp` A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`stopFlags` Must be zero.

function result A result code.

TuneGetVolume

The `TuneGetVolume` function returns the volume associated with the entire sequence.

```
pascal ComponentResult TuneGetVolume(
                    TunePlayer tp);
```

`tp` A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

function result The volume as a value from 0.0 to 1.0 or a negative result code.

TuneSetVolume

The `TuneSetVolume` function sets the volume for the entire sequence.

```
pascal ComponentResult TuneSetVolume(
                    TunePlayer tp,
                    Fixed volume);
```

`tp`	A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
`volume`	The volume to use for the sequence. The value is a fixed 16.16 number.
function result	A result code.

DISCUSSION

The `TuneSetVolume` function sets the volume level of the active sequence to the value of the `volume` parameter ranging from 0.0 to 1.0.

Note
Individual instruments within the sequence can maintain independent volume levels. ◆

TuneSetSoundLocalization

The `TuneSetSoundLocalization` function passes sound localization data to a tune player.

```
extern pascal ComponentResult TuneSetSoundLocalization(
                    TunePlayer tp,
                    Handle data);
```

`tp`	A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
`data`	The sound localization data to be passed.
function result	A result code.

TuneGetTimeBase

The `TuneGetTimeBase` function returns the time base of the tune player.

```
pascal ComponentResult TuneGetTimeBase(
                    TunePlayer tp,
                    TimeBase *tb);
```

`tp` A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`tb` An initialized `TimeBase` object.

function result A result code.

DISCUSSION

The `TuneGetTimeBase` function returns, in the `tb` parameter, the time base used to control the sequence timing. The sequence can be controlled in several ways through its time base. The rate of playback can be changed, or the `TimeBase` object can be slaved to a clock or time base different than real time.

TuneGetTimeScale

The `TuneGetTimeScale` function returns the current time scale, in units-per-second, for the specified tune player instance.

```
pascal ComponentResult TuneGetTimeScale(
                    TunePlayer tp,
                    TimeScale *scale);
```

`tp` A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`scale` An initialized `TimeScale` object.

function result A result code.

TuneSetTimeScale

The `TuneSetTimeScale` function sets the time scale used by the specified tune player instance.

```
pascal ComponentResult TuneSetTimeScale(
                    TunePlayer tp,
                    TimeScale scale);
```

`tp` A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`scale` The time scale value to be used, in units per second.

function result A result code.

DISCUSSION

The `TuneSetTimeScale` function sets the time scale data used by the tune player's sequence data when interpreting time-based events.

TuneGetPartMix

You use the `TuneGetPartMix` function to get volume, balance, and mixing settings for a specified part of a tune.

```
pascal ComponentResult TuneGetPartMix (
                    TunePlayer tp,
                    unsigned long partNumber,
                    long *volumeOut,
                    long *balanceOut,
                    long *mixFlagsOut);
```

`tp` Specifies the instance of a tune player component for this request. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function.

`partNumber` Specifies the part number for this request.

`volumeOut`	Returns the volume for the part.
`balanceOut`	Returns the balance for the part.
`mixFlagsOut`	Returns flags that control part mixing. These flags are described in “Part Mixing Flags” (page 71).

TuneSetPartMix

You use the `TuneSetPartMix` function to set volume, balance, and mixing settings for a specified part of a tune.

```
pascal ComponentResult TuneSetPartMix (
                    TunePlayer tp,
                    unsigned long partNumber,
                    long volume,
                    long balance,
                    long mixFlags);
```

`tp`	Specifies the instance of a tune player component for this request. Your software obtains this reference when calling the Component Manager’s `OpenComponent` or `OpenDefaultComponent` function.
`partNumber`	Specifies the part number for this request.
`volume`	Specifies the volume for the part.
`balance`	Specifies the balance for the part.
`mixFlags`	Flags that control part mixing. These flags are described in “Part Mixing Flags” (page 71).

TuneInstant

You can use the `TuneInstant` function to play the particular sequence events active at a specified position.

```
pascal ComponentResult TuneInstant(
                    TunePlayer tp,
                    unsigned long *tune,
                    unsigned long tunePosition);
```

`tp`	A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
`tune`	Pointer to tune sequence data.
`tunePosition`	Position within tune sequence data in time units.
function result	A result code.

DISCUSSION

The `TuneInstant` function plays the notes that are "on" at the point specified by the `tunePosition` parameter. The notes are started and then left playing on return. The notes can be silenced by calling the `TuneStop` function. This call is useful for enabling user "scrubbing" on a sequence.

TunePreroll

The `TunePreroll` function prepares for playing tune player sequence data by attempting to reserve note channels for each part in the sequence.

```
pascal ComponentResult TunePreroll (TunePlayer tp);
```

`tp`	A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" for details.
function result	A result code.

TuneUnroll

The `TuneUnroll` function releases any note channel resources that may have been locked down by previous calls to `TunePreroll` for this tune player.

```
pascal ComponentResult TuneUnroll (TunePlayer tp);
```

`tp` A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

function result A result code.

TuneGetIndexedNoteChannel

You can use the `TuneGetIndexedNoteChannel` function to determine how many parts the tune is playing and which instrument is assigned to those parts.

```
pascal ComponentResult TuneGetIndexedNoteChannel(
                    TunePlayer tp,
                    long i,
                    NoteChannel *nc);
```

`tp` A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" for details.

`i` Note channel index or 0 to get the number of parts.

`nc` Allocated initialized note channel.

function result A positive value is the number of note channels used by the tune player; a negative value is a result code.

DISCUSSION

The tune player allocates note channels that best satisfy the requested instrument in the tune header. The application can use this call to determine which instrument was actually used for each note channel. The `TuneGetIndexedNoteChannel` function takes the tune player in the `tp` parameter

and returns the number of parts (1...n) allocated to the tune player. You can then pass the function a part index and it returns, in the `nc` parameter, the note channel allocated for that part.

TuneGetStatus

The `TuneGetStatus` function returns an initialized structure describing the state of the tune player instance.

```
pascal ComponentResult TuneGetStatus(
                    TunePlayer tp,
                    TuneStatus *status);
```

`tp` A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`status` A pointer to an initialized tune status structure (page 86).

function result A result code.

TuneSetPartTranspose

The `TuneSetPartTranspose` function modifies the pitch and volume of every note of a tune.

```
extern pascal ComponentResult TuneSetPartTranspose(
                    TunePlayer tp,
                    unsigned long part,
                    long transpose,
                    long velocityShift);
```

`tp` A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" for details.

`part` The part for which you want to change pitch and volume.

`transpose` A value by which to modify the pitch of the note. The value is a small integer for semitones or an 8.8 fixed-point number for microtones.

`velocityShift` A value to add to the `velocity` parameter passed to the `NAPlayNote` function.

function result A result code.

TuneGetNoteAllocator

The `TuneGetNoteAllocator` function returns the instance of the note allocator that the tune player is using.

```
extern pascal NoteAllocator TuneGetNoteAllocator (TunePlayer tp);
```

`tp` A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

function result A note allocator or a result code.

TuneSetSofter

The `TuneSetSofter` function adjusts the volume a tune is played at to the softer volume produced by QuickTime 2.1. Files imported with QuickTime 2.1 automatically played softer. Files imported with QuickTime 2.5 or later play at the new, louder volume.

```
extern pascal ComponentResult TuneSetSofter(
                    TunePlayer tp,
                    long softer);
```

`tp` A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" for details.

`softer`	A value of 1 means play at the QuickTime 2.1 volume; a value of 0 means don't make the volume softer.
function result	A result code.

TuneSetBalance

Use the `TuneSetBalance` function to modify the pan controller setting for a tune player.

```
extern pascal ComponentResult TuneSetBalance(
                    TunePlayer tp,
                    long balance);
```

`tp`	A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
`balance`	Modifies the pan controller setting. Valid values are from –128 to 128 for left to right balance.
function result	A result code.

TuneTask

Call the `TuneTask` function periodically to allow a tune player to perform tasks it must perform at foreground task time.

```
extern pascal ComponentResult TuneTask (TunePlayer tp);
```

`tp`	A tune player identifier, obtained from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
function result	A result code.

DISCUSSION

Certain operations can be performed only at foreground application task time. Specifically, the QuickTime music synthesizer cannot load instruments from disk at interrupt time. As a result, embedded program changes are not performed until `TuneTask` is called.

Note Allocator Functions: Note Channel Allocation and Use

The functions described in this section create, manipulate, and get information about note channels.

NANewNoteChannel

The `NANewNoteChannel` function requests a new note channel with the qualities described in the `noteRequest` structure.

```
pascal ComponentResult NANewNoteChannel(
                    NoteAllocator na,
                    NoteRequest *noteRequest,
                    NoteChannel *outChannel);
```

`na`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
`noteRequest`	A pointer to a note request structure.
`outChannel`	On exit, a pointer to an identifier for a new note channel or `nil` if the function fails to create a note channel.
function result	A result code.

DISCUSSION

The caller can request an instrument that is not currently allocated to a part. In that case, the `NANewNoteChannel` function may return a value in `outChannel`, even though the request cannot initially be satisfied. The note channel may become valid at a later time, as other note channels are released or other music components are registered.

The NANewNoteChannel function searches all available music components for the instrument that best matches the specifications in the ToneDescripion structure that is contained within the noteRequest parameter.

If an error occurs, the note noteChannel is initialized to nil.

NANewNoteChannelFromAtomicInstrument

You can use the NANewNoteChannelFromAtomicInstrument function to request a new note channel for an atomic instrument.

```
extern pascal ComponentResult NANewNoteChannelFromAtomicInstrument(
                    NoteAllocator na,
                    AtomicInstrumentPtr instrument,
                    long flags,
                    NoteChannel *outChannel);
```

na	You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
instrument	A pointer to the atomic instrument. This may be a dereferenced locked QT atom container.
flags	These flags specify details of initializing a part with an atomic instrument. See "Flags for Setting Atomic Instruments" (page 63).
outChannel	On exit, a pointer to an identifier for a new note channel or nil if the function fails to create a note channel.
function result	A result code.

DISCUSSION

The NANewNoteChannelFromAtomicInstrument function takes a note allocator identifier in the na parameter and a pointer to the atomic instrument you are requesting a new channel for in the instrument parameter. Among other things, you can specify how to handle the expanded sample with the flags parameter.

The function returns the note channel allocated for the instrument in the `outChannel` parameter or `nil` if an error occurs.

NADisposeNoteChannel

The `NADisposeNoteChannel` function deletes the specified note channel.

```
pascal ComponentResult NADisposeNoteChannel(
                    NoteAllocator na,
                    NoteChannel noteChannel);
```

`na` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`noteChannel` Note channel to be disposed. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

function result A result code.

NAGetNoteChannelInfo

The `NAGetNoteChannelInfo` function returns the index of the music component for the allocated channel and its part number on that music component.

```
pascal ComponentResult NAGetNoteChannelInfo(
                    NoteAllocator na,
                    NoteChannel noteChannel,
                    long *index,
                    long *part);
```

`na` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

noteChannel	Note channel to get information about. You obtain the note channel identifier from the NANewNoteChannel or the NANewNoteChannelFromAtomicInstrument function.
index	Music component index.
part	Music component part pointer.
function result	A result code.

DISCUSSION

The NAGetNoteChannelInfo function allows direct access to the music component allocated to the note channel by the note allocator. The index returned becomes invalid if music components are subsequently registered or unregistered.

NAGetIndNoteChannel

The NAGetIndNoteChannel function returns the number of note channels handled by the specified note allocator instance. It can also return a requested note channel.

```
extern pascal ComponentResult NAGetIndNoteChannel(
                    NoteAllocator na,
                    long index,
                    NoteChannel *nc,
                    long *seed);
```

na	You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
index	The index of the note channel. If zero, the result is still the number of note channels, but *nc is not filled out.
nc	The note channel requested.
seed	A number that changes on successive calls if anything significant changes about a note channel—for example, if the note channel has been reallocated or released.

function result Positive results are the index count; negative results are error codes.

DISCUSSION

To get a count of the note channels, pass the `NAGetIndNoteChannel` function 0 in the `index` parameter. To get a specific note channel, pass the index value returned by a previous call to `NAGetIndNoteChannel`.

NAUseDefaultMIDIInput

The `NAUseDefaultMIDIInput` function defines an entry point to service external MIDI device events. This routine, in turn, calls the QuickTime MIDI components to query them. `NAGetMIDIPorts` is the correct call for you to make. You should *not* call `QTMIDI`.

```
pascal ComponentResult NAUseDefaultMIDIInput (
                    NoteAllocator na,
                    MusicMIDIReadHookUPP readHook,
                    long refCon,
                    unsigned long flags);
```

`na` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`readHook` Process pointer for MIDI service.

`refcon` Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.

`flags` Must contain zero.

function result A result code.

DISCUSSION

The `NAUseDefaultMIDIInput` function specifies an application's procedure to service external MIDI events. The specified application's procedure call, defined

by readHook, is called when the external default MIDI device has incoming MIDI data for the application.

NALoseDefaultMIDIInput

The NALoseDefaultMIDIInput function removes the external default MIDI service procedure call, if previously defined by NAUseDefaultMIDIInput. This routine, in turn, calls the QuickTime MIDI components to query them. NAGetMIDIPorts is the correct call for users to make. Users should *not* call QTMIDI.

```
pascal ComponentResult NALoseDefaultMIDIInput (NoteAllocator na);
```

na	You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
function result	A result code or -1 if a default MIDI device was not in use.

NAPrerollNoteChannel

The NAPrerollNoteChannel function attempts to reallocate the note channel if it was invalid previously.

```
pascal ComponentResult NAPrerollNoteChannel(
                    NoteAllocator na,
                    NoteChannel noteChannel);
```

na	You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
noteChannel	Note channel to be re-allocated. You obtain the note channel identifier from the NANewNoteChannel or the NANewNoteChannelFromAtomicInstrument function.
function result	A result code.

DISCUSSION

The `NAPrerollNoteChannel` function attempts to reallocate the note channel, if it was invalid previously. It could have been invalid if there were no available voices on any registered music components when the note channel was created.

NAUnrollNoteChannel

The `NAUnrollNoteChannel` function marks a note channel as available to be stolen.

```
pascal ComponentResult NAUnrollNoteChannel(
                    NoteAllocator na,
                    NoteChannel noteChannel);
```

`na` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`noteChannel` Note channel to be unrolled. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

function result A result code.

NAResetNoteChannel

The `NAResetNoteChannel` function turns off all currently "on" notes on the note channel and resets all controllers to their default values.

```
pascal ComponentResult NAResetNoteChannel(
                    NoteAllocator na,
                    NoteChannel noteChannel);
```

`na` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`noteChannel` The note channel to reset. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

function result A result code.

DISCUSSION

The `NAResetNoteChannel` function resets the specified note channel by turning "off" any note currently playing. All controllers are reset to their default state. The effects of the `NAResetNoteChannel` call are propagated down to the allocated part within the appropriate music component.

NASetNoteChannelVolume

The `NASetNoteChannelVolume` function sets the volume on the specified note channel.

```
pascal ComponentResult NASetNoteChannelVolume(
                    NoteAllocator na,
                    NoteChannel noteChannel,
                    Fixed volume);
```

`na` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`noteChannel` The note channel to reset. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

`volume` The volume to set the channel to. The value is a fixed 16.16 number.

DISCUSSION

The `NASetNoteChannelVolume` function sets the volume for the note channel, which is different from a controller 7 (volume controller) setting.

Both volume settings allow fractional values of 0.0 to 1.0. Each value modifies the other. For example, a volume controller value of 0.5 and a `NASetNoteChannelVolume` value of 0.5 result in a 0.25 volume level.

NASetNoteChannelBalance

The `NASetNoteChannelBalance` function modifies the pan controller setting for a note channel.

```
extern pascal ComponentResult NASetNoteChannelBalance(
                    NoteAllocator na,
                    NoteChannel noteChannel,
                    long balance);
```

`na`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
`noteChannel`	The note channel to be balanced. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.
`balance`	Specifies how to modify the pan controller setting. Valid values are from –128 to 128 for left to right balance.
function result	A result code.

NASetNoteChannelSoundLocalization

The `NASetNoteChannelSoundLocalization` function passes sound localization data to a note channel.

```
extern pascal ComponentResult NASetNoteChannelSoundLocalization(
                    NoteAllocator na,
                    NoteChannel noteChannel,
                    Handle data);
```

na	You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
noteChannel	The note channel to pass the data to. You obtain the note channel identifier from the NANewNoteChannel or the NANewNoteChannelFromAtomicInstrument function.
data	Sound localization data.
function result	A result code.

NAPlayNote

The NAPlayNote function plays a note with a specified pitch and velocity on the specified note channel.

```
pascal ComponentResult NAPlayNote(
                    NoteAllocator na,
                    NoteChannel noteChannel,
                    long pitch,
                    long velocity);
```

na	You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
noteChannel	The note channel to play the note. You obtain the note channel identifier from the NANewNoteChannel or the NANewNoteChannelFromAtomicInstrument function.
pitch	The pitch at which to play the note. You can specify values as integer pitch values (0–127 where 60 is middle C) or fractional pitch values (256 (0x1.00) through 32767 (0x7F.FF)).
velocity	The velocity with which the key is struck. A value of 0 is silence; a value of 127 is maximum force.
function result	A result code.

DISCUSSION

The `NAPlayNote` function plays a specific note. If the pitch is a number from 0 to 127, then it is the MIDI pitch, where 60 is middle C. If the pitch is a positive number above 65535, then the value is a fixed-point pitch value. Thus, microtonal values can be specified. The range 256 (`0x01.00`) through 32767 (`0x7F.FF`), and all negative values, are not defined, and should not be used.

The velocity refers to how hard the key was struck (if performed on a keyboard instrument). Typically, this translates directly to volume, but on many synthesizers this also subtly alters the timbre of the tone.

NAGetController

You use the `NAGetController` function to get the controller settings for a note channel.

```
pascal ComponentResult NAGetController (
                    NoteAllocator na,
                    NoteChannel noteChannel,
                    long controllerNumber,
                    long *controllerValue);
```

`na` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`noteChannel` Note channel for which to get controller settings. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

`controllerNumber`
The controller for which to get settings. For valid values, see "Controller Numbers" (page 56).

`controllerValue`
On return, the value for the controller setting, typically 0 (`0x00.00`) to 32767 (`0x7F.FF`).

NASetController

The `NASetController` function changes the controller setting on a note channel to a specified value.

```
pascal ComponentResult NASetController
                    (NoteAllocator na,
                    NoteChannel noteChannel,
                    long controllerNumber,
                    long controllerValue);
```

`na` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`noteChannel` Note channel on which to change controller. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

`controllerNumber`
The controller to set. For valid values, see "Controller Numbers" (page 56).

`controllerValue`
Value for controller setting, typically 0 (`0x00.00`) to 32767 (`0x7F.FF`).

NAGetKnob

Use the `NAGetKnob` function to get the value of a knob for a given note channel.

```
extern pascal ComponentResult NAGetKnob(
                    NoteAllocator na,
                    NoteChannel noteChannel,
                    long knobNumber,
                    long *knobValue);
```

`na` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

noteChannel	The note channel whose knob value you want to get. You obtain the note channel identifier from the NANewNoteChannel or the NANewNoteChannelFromAtomicInstrument function.
knobNumber	The index or ID of the knob whose value you want to get.
knobValue	On exit, the value of the knob.
function result	A result code.

DISCUSSION

The NAGetKnob function takes a note allocator component identifier in the na parameter, a note channel identifier in the noteChannel parameter, and the knob index or ID in the knobNumber parameter. It returns, in the knobValue parameter, a pointer to the current value of the knob.

NASetKnob

The NASetKnob function sets a note channel knob to a particular value.

```
pascal ComponentResult NASetKnob(
                    NoteAllocator na,
                    NoteChannel noteChannel,
                    long knobNumber,
                    long knobValue);
```

na	You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
noteChannel	Note channel on which to set the knob value. You obtain the note channel identifier from the NANewNoteChannel or the NANewNoteChannelFromAtomicInstrument function.
knobNumber	Index or ID of the knob to be set.
knobValue	Value to set knob to.
function result	A result code.

DISCUSSION

The `NASetKnob` function takes a note allocator component identifier in the `na` parameter, a note channel identifier in the `noteChannel` parameter, the knob ID or index in the `knobNumber` parameter, and a knob value in the `knobValue` parameter. It sets the specified knob to the given value.

NAFindNoteChannelTone

The `NAFindNoteChannelTone` function locates the instrument that best fits a requested tone description for a specific channel.

```
pascal ComponentResult NAFindNoteChannelTone(
                    NoteAllocator na,
                    NoteChannel noteChannel,
                    ToneDescription *td,
                    long *instrumentNumber);
```

`na` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`noteChannel` The note channel for which you want an instrument. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

`td` Description for instrument fit.

`instrumentNumber`
On exit, the number of the instrument that best fits the tone description.

function result A result code.

NASetInstrumentNumber

The `NASetInstrumentNumber` function initializes a synthesizer part with the specified instrument.

```
pascal ComponentResult NASetInstrumentNumber(
                    NoteAllocator na,
                    NoteChannel noteChannel,
                    long instrumentNumber);
```

`na` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`noteChannel` Note channel to initialize with the instrument. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

`instrumentNumber`
Number of the instrument to initialize the part with. This number is unique to each synthesizer. General MIDI synthesizers all share the range 1–128 and 16365 to `kLastDrumKit`.

function result A result code.

NASetInstrumentNumberInterruptSafe

You can use the `NASetInstrumentNumberInterruptSafe` function to initialize a synthesizer part with the specified instrument during interrupt time.

```
extern pascal ComponentResult NASetInstrumentNumberInterruptSafe(
                    NoteAllocator na,
                    NoteChannel noteChannel,
                    long instrumentNumber);
```

`na` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`noteChannel` Note channel to initialize with the instrument. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

`instrumentNumber`
Number of the instrument to initialize the part with.

function result A result code.

DISCUSSION

If the instrument is not already loaded when you call the `NASetInstrumentNumberInterruptSafe` function, you have to wait for the next call to the `NATask` function for the instrument to become available.

NASetAtomicInstrument

The `NASetAtomicInstrument` function initializes a synthesizer part with an atomic instrument.

```
extern pascal ComponentResult NASetAtomicInstrument(
                    NoteAllocator na,
                    NoteChannel noteChannel,
                    AtomicInstrumentPtr instrument,
                    long flags);
```

`na` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`noteChannel` The note channel to apply the atomic instrument to. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

`instrument` A pointer to the atomic instrument. This can be a locked, dereferenced atomic instrument.

`flags` Details about how to initialize the part. For a description of the flags, see "Flags for Setting Atomic Instruments" (page 63).

function result A result code.

NASendMIDI

Use the NASendMIDI function to send a MIDI music packet to a synthesizer that contains a specific note channel. This routine, in turn, calls the QuickTime MIDI components to query them. NAGetMIDIPorts is the correct call for users to make. Users should *not* call QTMIDI.

```
extern pascal ComponentResult NASendMIDI(
                    NoteAllocator na,
                    NoteChannel noteChannel,
                    MusicMIDIPacket *mp);
```

na	You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
noteChannel	The function sends the packet to the synthesizer that contains this note channel. You obtain the note channel identifier from the NANewNoteChannel or the NANewNoteChannelFromAtomicInstrument function.
mp	The music packet to be sent.
function result	A result code.

DISCUSSION

The NASendMIDI function sends the MIDI music packet pointed to by the mp parameter to the synthesizer that contains the note channel identified by the noteChannel parameter. The na parameter specifies the note allocator instance to use.

NAGetNoteRequest

The NAGetNoteRequest function gets the note request passed to a note channel.

```
extern pascal ComponentResult NAGetNoteRequest(
                    NoteAllocator na,
                    NoteChannel noteChannel,
                    NoteRequest *nrOut);
```

na	You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
noteChannel	The note channel whose note request you want to get. You obtain the note channel identifier from the NANewNoteChannel or the NANewNoteChannelFromAtomicInstrument function.
nrOut	On exit, a note request structure (page 86).
function result	A result code.

DISCUSSION

The NAGetNoteRequest function takes a note allocator instance in the na parameter and a note channel identifier in the noteChannel parameter. It returns, in the *nrOut parameter, the note request that was used to allocate the specified note channel.

Note Allocator Functions: Miscellaneous Interface Tools

The functions in this section provide a user interface for instrument selection and presenting copyright information.

NAPickInstrument

The NAPickInstrument function presents a user interface for picking an instrument.

```
pascal ComponentResult NAPickInstrument(
                    NoteAllocator na,
                    ModalFilterUPP filterProc,
                    StringPtr prompt,
                    ToneDescription *sd,
                    unsigned long flags,
                    long refCon,
                    long reserved1,
                    long reserved2);
```

`na`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
`filterProc`	Standard modal filter universal procedure pointer.
`prompt`	Dialog box prompt "New Instrument".
`sd`	On entry, the tone description of the instrument that appears in the picker dialog box. On exit, a tone description of the instrument the user selected.
`flags`	Determines whether to display the picker dialog box and what instruments appear for selection. See "Pick Instrument Flags" (page 67).
`refcon`	Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.
`reserved1`	Must contain zero.
`reserved2`	Must contain zero.
function result	A result code or `-1` if there is a problem opening the dialog box.

DISCUSSION

The `flags` values limit which instruments appear within the dialog box. If the `kPickDontMix` flag is set, the dialog box does not display a mix of synthesizer part types. For example, if the current instrument is a drum, only available drums appear in the dialog box. The `kPickSameSynth` flag allows selections only within the current synthesizer. The `kPickUserInsts` flag allows user modifiable instruments to appear.

SEE ALSO

`NAPickEditInstrument` function

NAPickEditInstrument

The `NAPickEditInstrument` function presents a user interface for changing the instrument in a live note channel or modifying an atomic instrument.

```
extern pascal ComponentResult NAPickEditInstrument(
                    NoteAllocator na,
                    ModalFilterUPP filterProc,
                    StringPtr prompt,
                    long refCon,
                    NoteChannel nc,
                    AtomicInstrument ai,
                    long flags);
```

`na` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`filterProc` Standard modal filter universal procedure pointer.

`prompt` Dialog box prompt "New Instrument".

`refCon` Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.

`nc` The live note channel that appears in the dialog box. If you specify a note channel, set the `ai` parameter to 0. You obtain the note channel identifier from the `NANewNoteChannel` or the `NANewNoteChannelFromAtomicInstrument` function.

`ai` The atomic instrument that appears in the dialog box. If you specify an atomic instrument, set the `nc` parameter to 0. You obtain the atomic instrument from the `InstrumentGetInst` function.

`flags` Flags limiting the instruments presented. See "Pick Instrument Flags" (page 67)

function result A result code or –1 if there is a problem opening the dialog box.

DISCUSSION

The `flags` value limits which instruments appear within the dialog box. If the `kPickDontMix` flag is set, the dialog box does not display a mix of synthesizer part types. For example, if the current instrument is a drum, only available drums appear in the dialog box. The `kPickSameSynth` flag allows selections only within the current synthesizer. The `kPickUserInsts` flag allows user modifiable instruments to appear. If the `kPickEditAllowPick` flag is not set, no dialog box appears.

SEE ALSO

`NAPickInstrument` function

NAStuffToneDescription

The `NAStuffToneDescription` function initializes a tone description structure with the details of a General MIDI note channel.

```
pascal ComponentResult NAStuffToneDescription(
                    NoteAllocator na,
                    long gmNumber,
                    ToneDescription *td);
```

`na`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
`gmNumber`	A General MIDI instrument number.
`td`	On exit, an initialized tone description. The instrument name field will be filled in with the string name for the instrument.
function result	A result code.

NAPickArrangement

The `NAPickArrangement` function displays a dialog box to allow instrument selection.

```
pascal ComponentResult NAPickArrangement(
                    NoteAllocator na,
                    ModalFilterUPP filterProc,
                    StringPtr prompt,
                    long zero1,
                    long zero2,
                    Track t,
                    StringPtr songName);
```

`na`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
`filterProc`	Standard modal filter universal procedure pointer.
`prompt`	Dialog box prompt.
`zero1`	Must be 0**.**
`zero2`	Must be 0.
`t`	Arrangement movie track number.
`songName`	Name of song to display in dialog box.
function result	A result code or `-1` if there is a problem opening the dialog box.

NACopyrightDialog

The `NACopyrightDialog` function displays a copyright dialog box with information specific to a music device.

```
pascal ComponentResult NACopyrightDialog(
                    NoteAllocator na,
                    PicHandle p,
                    StringPtr author,
                    StringPtr copyright,
```

```
                    StringPtr other,
                    StringPtr title,
                    ModalFilterUPP filterProc,
                    long refCon);
```

`na`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
`p`	Picture image resource handle for dialog box.
`author`	Author information.
`copyright`	Copyright information.
`other`	Any additional information.
`title`	Title information.
`filterProc`	Standard modal filter universal procedure pointer.
`refcon`	Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.
function result	A result code or –1 if there is a problem opening the dialog box.

Note Allocator Functions: System Configuration and Utility

Use the functions in this section to create and maintain a database of music components, to save configuration information in the QuickTime Preferences file, to establish connections to external MIDI devices, and to allow the note allocator to perform necessary tasks at task foreground time.

NARegisterMusicDevice

The `NARegisterMusicDevice` function registers a music component with the note allocator.

```
pascal ComponentResult NARegisterMusicDevice(
                    NoteAllocator na,
                    OSType synthType,
                    Str31 name,
                    SynthesizerConnections *connections);
```

`na`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
`synthType`	Subtype of the music component.
`name`	The synthesizer name.
`connections`	A synthesizer connection structure (page 84) that describes how a MIDI device is connected.
function result	A result code.

DISCUSSION

The value of the `synthType` parameter is the music component's subtype. The `name` parameter provides a means of distinguishing multiple instances of the same type of device and is a string that can be displayed to the user. If no value is passed in the `name` parameter, the name defaults to the name of the music component type. The name appears in the instrument picker dialog box.

The `connections` parameter specifies the hardware connections to the device.

RESULT CODES

`SynthesizerErr`	If too many synthesizers registered.
`midiManagerAbsentErr`	If MIDI not available.

NAUnregisterMusicDevice

The `NAUnregisterMusicDevice` function removes a previously registered music component from the note allocator.

```
pascal ComponentResult NAUnregisterMusicDevice(
                    NoteAllocator na,
                    long index);
```

`na`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
`index`	Synthesizer to unregister. The value is 1 through the registered music component count returned by the `NAGetRegisteredMusicDevice` function (page 127).
function result	A result code. In addition to QTMA result codes, this function may return a result code from the `CloseComponent` function.

NAGetRegisteredMusicDevice

The `NAGetRegisteredMusicDevice` function returns specifics about music components registered to the specified note allocator instance.

```
pascal ComponentResult NAGetRegisteredMusicDevice(
                    NoteAllocator na,
                    long index,
                    OSType *synthType,
                    Str31 name,
                    SynthesizerConnections *connections,
                    MusicComponent *mc);
```

na	You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
index	The index of the music component to get information about or 0 to get the total number of music components registered with the note allocator.
synthType	Synthesizer type.
name	Synthesizer name as a text string.
connections	A synthesizer connections for MIDI devices structure (page 84).
mc	Music component instance identifier.
function result	Positive values are the number of music components registered with the note allocator; negative values are result codes.

DISCUSSION

To get a count of the registered music components, pass the NAGetRegisteredMusicDevice function 0 in the index parameter. The return value is the count of components. To get information about one of the music components registered with the note allocator, pass the music component index in the index parameter. The index value can be 1 through the number of registered components returned by a previous call to NAGetRegisteredMusicDevice.

If you request information about a specific registered music component, the NAGetRegisteredMusicDevice function returns the type of synthesizer the component supports in the synthType parameter, the name of the synthesizer in the name parameter, and the music component identifier in the mc parameter. For MIDI devices, it returns a pointer to a MIDI devices structure with information about the synthesizer connections.

NAGetDefaultMIDIInput

The NAGetDefaultMIDIInput function is used to obtain external MIDI connection information. This routine, in turn, calls the QuickTime MIDI components to

query them. NAGetMIDIPorts is the correct call for you to make. You should *not* call QTMIDI.

```
pascal ComponentResult NAGetDefaultMIDIInput(
                    NoteAllocator na,
                    SynthesizerConnections *sc);
```

na	You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
sc	On exit, a synthesizer connection structure (page 84) that describes how a MIDI device is connected.

DISCUSSION

The NASGetDefaultMIDIInput function returns an initialized SynthesizerConnections structure containing information about the external MIDI device attached to the system that has been selected as the default MIDI input device. The external MIDI device provides note input directly to the note allocator.

NASetDefaultMIDIInput

The NASetDefaultMIDIInput function initializes an external MIDI device used to receive external note input.This routine, in turn, calls the QuickTime MIDI components to query them. NAGetMIDIPorts is the correct call for users to make. Users should *not* call QTMIDI.

```
pascal ComponentResult NASetDefaultMIDIInput(
                    NoteAllocator na,
                    SynthesizerConnections *sc);
```

na	You obtain the note allocator identifier from the Component Manager's OpenComponent function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
sc	A synthesizer connection structure (page 84) that describes how a MIDI device is connected.

DISCUSSION

The `SynthesizerConnections` structure fields `clientID`, `inputPortID`, and `outputPortID` are MIDI Manager identifiers. The `midiChannel` field is the MIDI system channel value.

function result A result code.

NAGetMIDIPorts

The `NAGetMIDIPorts` function gets the MIDI input and output ports available to a note allocator.This routine, in turn, calls the QuickTime MIDI components to query them. `NAGetMIDIPorts` is the correct call for you to make. You should *not* call `QTMIDI`.

```
extern pascal ComponentResult NAGetMIDIPorts(
                    NoteAllocator na,
                    QTMIDIPortListHandle *inputPorts,
                    QTMIDIPortListHandle *outputPorts);
```

`na` You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`inputPorts` On exit, a handle giving the number of input ports (the first two bytes) followed by a list of QuickTime MIDI port structures (page 85).

`outputPorts` On exit, a handle giving the number of output ports (the first two bytes) followed by a list of QuickTime MIDI port structures (page 85).

function result A result code.

NASaveMusicConfiguration

The `NASaveMusicConfiguration` saves the current list of registered devices to a file.

```
pascal ComponentResult NASaveMusicConfiguration (NoteAllocator na);
```

`na`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
function result	A result code or –1 if there is a problem opening or creating the QuickTime Preferences file.

DISCUSSION

The `NASaveMusicConfiguration` function saves the current list of registered devices to a file. This file is read whenever a note allocator connection is opened, restoring the previously configured list of devices. The list is saved in the QuickTime Preferences file.

NATask

Call the `NATask` function periodically to allow the note allocator to perform tasks in foreground task time.

```
extern pascal ComponentResult NATask (NoteAllocator na);
```

`na`	You obtain the note allocator identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
function result	A result code.

DISCUSSION

The `NATask` function calls each registered music component's `MusicTask` function.

Music Component Functions: Synthesizer

The functions in this section obtain specific information about a synthesizer and obtain a best instrument fit for a requested tone from the available instruments within the synthesizer; play a note with a specified pitch, volume, and duration; get and set a particular synthesizer knob; obtain synthesizer knob information; and get and set external MIDI procedure name entry points.

MusicGetDescription

The `MusicGetDescription` function returns a structure describing the synthesizer controlled by the music component device.

```
pascal ComponentResult MusicGetDescription(
                    MusicComponent mc,
                    SynthesizerDescription *sd);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`sd` Pointer to synthesizer description structure (page 73).

function result A result code.

DISCUSSION

The `MusicGetDescription` function returns a structure describing the specified music component device. The `SynthesizerDescription` structure is filled out by the particular music component.

MusicFindTone

The `MusicFindTone` function returns an instrument number based on a tone description.

```
pascal ComponentResult MusicFindTone(
                    MusicComponent mc,
                    ToneDescription *td,
                    long *libraryIndexOut,
                    unsigned long *fit);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`td` Pointer to a tone description structure (page 75).

`libraryIndexOut` On exit, contains the number of the best-matching instrument. Only General MIDI numbers are guaranteed to be the same for later instantiations of the component.

`fit` On exit, indicates how well an instrument matches the tone description. For valid values, see "Tone Fit Flags" (page 59).

function result A result code.

DISCUSSION

The `MusicFindTone` function returns the number of the best-matching instrument provided by the specified music component. The closeness of the match is specified by the `fit` parameter.

The music component searches for an instrument as follows:

1. If the `synthesizerType` field of the `td` parameter matches the type of the specified music component, it first tries to find an instrument that matches the value of the `instrumentNumber` field of the `td` parameter. If this value is in the range 129–16512, which specifies a GS instrument, and the GS instrument is not available, it tries to find the General MIDI instrument that corresponds to it, which has the number ((`GSinstrumentnumber` – 1) & 0x7F) + 1)). If the value is greater than 16512, which specifies a transient ROM instrument or internal instrument index value, it tries to find an instrument that matches the `synthesizerName` field of the `td` parameter. If that fails, it tries to find an

instrument that matches the value of the value of the `gmNumber` field of the `td` parameter.

2. If the `synthesizerType` field of the `td` parameter does not match the type of the specified music component, it tries to find an instrument that matches the value of the `gmNumber` field of the `td` parameter.

If none of these rules apply, or the fields are "blank" (zero for the type or numeric fields, or zero-length for the strings), then the call returns instrument 1 and a fit value of zero. The `synthesizerName` field may be ignored by the component; it is used by the note allocator when deciding which music device to use.

MusicPlayNote

The `MusicPlayNote` function plays a note on a specified part at a specified pitch and velocity.

```
pascal ComponentResult MusicPlayNote(
                    MusicComponent mc,
                    long part,
                    long pitch,
                    long velocity);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	The part to play the note on.
`pitch`	The pitch at which to play the note. Values are 0–127 for MIDI pitch or greater than 65535 for microtonal values.
`velocity`	How hard to strike the key. Values are 0–127 where 0 is silence.
function result	A result code.

DISCUSSION

The `MusicPlayNote` function is used to play notes by their pitch. If the pitch is specified by a number from 0 to 127, it is a MIDI pitch, where 60 is middle C. If

the pitch is a positive number above 65535, the value is a fixed-point pitch value. Thus, microtonal values may be specified.

Velocity refers to how hard the key is struck (if performed on a keyboard-instrument); typically, this translates directly to volume, but on many synthesizers this also subtly alters the timbre of the tone.

The current note continues to play until a `MusicPlayNote` function with the same pitch and velocity of 0 turns the note off.

MusicGetKnob

The `MusicGetKnob` function returns the value of the specified global synthesizer knob. A global knob controls an aspect of the entire synthesizer. It is not specific to a part within the synthesizer.

```
pascal ComponentResult MusicGetKnob(
                    MusicComponent mc,
                    long knobID);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`knobID` Knob index or ID.

function result A result code.

MusicSetKnob

The `MusicSetKnob` function modifies the value of the specified global synthesizer knob. A global knob controls an aspect of the entire synthesizer. It is not limited to a part within the synthesizer.

```
pascal ComponentResult MusicSetKnob(
                    MusicComponent mc,
                    long knobID,
                    long knobValue);
```

mc	Music component instance identifier returned by NAGetRegisteredMusicDevice.
knobID	Knob index or ID.
knobValue	Value for specified knob.
function result	A result code.

MusicGetKnobDescription

The MusicGetKnobDescription function returns a pointer to an initialized knob description structure describing a global synthesizer knob. A global knob controls an aspect of the entire synthesizer; it is not limited to a part within the synthesizer.

```
pascal ComponentResult MusicGetKnobDescription(
                    MusicComponent mc,
                    long knobIndex,
                    KnobDescription *mkd);
```

mc	Music component instance identifier returned by NAGetRegisteredMusicDevice.
knobIndex	Knob index or ID.
mkd	Pointer to a knob description structure (page 78).
function result	A result code.

DISCUSSION

The initialized KnobDescription structure provides the application default values associated with the particular knob. You can use the information returned by a call to the MusicGetKnobDescription function to reset a knob to some known, usable value.

MusicGetInstrumentKnobDescription

The `MusicGetInstrumentKnobDescription` function gets the description of an instrument knob.

```
extern pascal ComponentResult MusicGetInstrumentKnobDescription(
                    MusicComponent mc,
                    long knobIndex,
                    KnobDescription *mkd);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`knobIndex`	A knob index or knob ID.
`mkd`	On exit, a knob description structure (page 78).
function result	A result code.

DISCUSSION

The `MusicGetInstrumentKnobDescription` function takes a music component instance identifier in the `mc` parameter and a knob index or knob ID in the `knobIndex` parameter. It returns a knob description structure in the `mkd` parameter.

MusicGetDrumKnobDescription

The `MusicGetDrumKnobDescription` function returns a description of a drum kit knob.

```
extern pascal ComponentResult MusicGetDrumKnobDescription(
                    MusicComponent mc,
                    long knobIndex,
                    KnobDescription *mkd);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`knobIndex`	A knob index or knob ID.

`mkd`	A pointer to a knob description structure (page 78).
function result	A result code.

DISCUSSION

The `MusicGetDrumKnobDescription` function takes a music component in the `mc` parameter and a knob index or knob ID in the `knobIndex` parameter. It returns a knob description structure in the `*mkd` parameter.

MusicGetKnobSettingStrings

The `MusicGetKnobSettingStrings` function returns a list of knob setting names known by the specified music component.

```
extern pascal ComponentResult MusicGetKnobSettingStrings(
                    MusicComponent mc,
                    long knobIndex,
                    long isGlobal,
                    Handle *settingsNames,
                    Handle *settingsCategoryLasts,
                    Handle *settingsCategoryNames);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`knobIndex`	The knob index or knob ID.
`isGlobal`	If a knob index is used, indicates whether the specified knob is a global knob.
`settingsNames`	The requested list of knob setting strings formatted as a short followed by packed strings.
`settingsCategoryLasts`	A group of short integers, the first of which contains the number of shorts to follow.

`settingsCategoryNames`
Knob setting category names formatted as a short followed by a list of names.

function result A result code.

Note
All handles must be disposed of by the caller.

MusicSetMIDIProc

The `MusicSetMIDIProc` function tells the music component what procedure to call when it needs to send MIDI data. This call is implemented only by a music component for a MIDI synthesizer.

```
pascal ComponentResult MusicSetMIDIProc(
                    MusicComponent mc,
                    MusicMIDISendUPP midiSendProc,
                    long refCon);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`midiSendProc` A pointer to the procedure to use when sending MIDI data.

`refcon` Contains a reference constant value. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.

function result A result code.

MusicGetMIDIProc

The `MusicGetMIDIProc` function returns a pointer to the procedure a music component is using to process external MIDI notes.

```
pascal ComponentResult MusicGetMIDIProc(
                    MusicComponent mc,
                    MusicMIDISendUPP *midiSendProc,
                    long *refCon);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`midiSendProc` Pointer to a MIDI serial port call.

`refcon` Contains a reference constant. The Movie Toolbox passes this reference constant to your error-notification function each time it calls your function.

function result A result code.

DISCUSSION

The `MusicGetMIDIProc` function returns, in the `midiSendProc` parameter, a pointer to the function that processes external MIDI notes. This function was set by a previous call to the `MusicSetMIDIProc` function. If no function has been set with the `MusicSetMIDIProc` function, `MusicGetMIDIProc` returns zero in the `midiSendProc` parameter.

MusicGetMIDIPorts

The `MusicGetMIDIPorts` function returns the number of input and output ports a MIDI device has.

```
extern pascal ComponentResult MusicGetMIDIPorts(
                    MusicComponent mc,
                    long *inputPortCount,
                    long *outputPortCount);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`inputPortCount`
On exit, the number of input MIDI ports available to the music component.

`outputPortCount`
On exit, the number of output MIDI ports available to the music component.

function result A result code.

DISCUSSION

The function takes a music component identifier in the `mc` parameter and returns, in the `inputPortCount` and `outputPortCount` parameters, the number of MIDI input and output ports available to the music component.

This call is implemented only for a hardware synthesizer, such as a NuBus or PCI card device.

MusicSendMIDI

Use the `MusicSendMIDI` function to send a MIDI packet to a specified port.

```
extern pascal ComponentResult MusicSendMIDI(
                    MusicComponent mc,
                    long portIndex,
                    MusicMIDIPacket *mp);
```

`mc` Music component instance returned by `NAGetRegisteredMusicDevice`.

`portIndex` The index of the port to send the MIDI packet to. The index value is 1 through the port count returned by the `MusicGetMIDIPorts` function.

`mp` The music MIDI packet to be sent.

function result A result code.

DISCUSSION

The `MusicSendMIDI` function takes a music component in the `mc` parameter and a port index in the `portIndex` parameter. It sends the MIDI music packet specified by the `mp` parameter to the specified port.

This call is implemented only for a hardware synthesizer, such as a NuBus or PCI card device.

MusicGetDeviceConnection

You can use the `MusicGetDeviceConnection` function to find out how many hardware synthesizers are available to a music component and to get the IDs for those devices.

```
extern pascal ComponentResult MusicGetDeviceConnection(
                    MusicComponent mc,
                    long index,
                    long *id1,
                    long *id2);
```

`mc`	Music component returned by `NAGetRegisteredMusicDevice`.
`index`	Index of the device for which you want to find out the IDs. Set to 0 if you are calling to get the number of hardware devices.
`id1`	On exit, a hardware synthesizer ID.
`id2`	On exit, another hardware synthesizer ID.
function result	A result code.

DISCUSSION

To get the number of hardware synthesizers available to the music component specified in the `mc` parameter and an index you can use to request ID numbers for a specific device, call the `MusicGetDeviceConnection` function with a value of 0 for the `index` parameter. You can then pass an index value in the `index` parameter, and the function returns hardware synthesizer IDs in the `id1` and `id2` parameters.

This call is implemented only for a hardware synthesizer, such as a NuBus or PCI card device.

MusicUseDeviceConnection

The `MusicUseDeviceConnection` function tells a music component which hardware synthesizer to talk to.

```
extern pascal ComponentResult MusicUseDeviceConnection(
                    MusicComponent mc,
                    long id1,
                    long id2);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`id1` The ID of the device returned in the `*id1` parameter of the `MusicGetDeviceConnection` function.

`id2` The ID of the device returned in the `*id2` parameter of the `MusicGetDeviceConnection` function.

function result A result code.

DISCUSSION

This call is implemented only for a hardware synthesizer, such as a NuBus or PCI card device.

Music Component Functions: Instruments and Parts

The functions described in this section initialize a part with an instrument, store instruments, list available instruments, manipulate parts, and get information about parts.

MusicGetPartInstrumentNumber

The `MusicGetPartInstrumentNumber` function returns the instrument number currently assigned to that part.

```
pascal ComponentResult MusicGetPartInstrumentNumber(
                    MusicComponent mc,
                    long part);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`part` Part number containing instrument.

function result A positive return value is the instrument number; a negative value is a result code.

MusicSetPartInstrumentNumber

The `MusicSetPartInstrumentNumber` function initializes a part with a particular instrument.

```
pascal ComponentResult MusicSetPartInstrumentNumber(
                    MusicComponent mc,
                    long part,
                    long instrumentNumber);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`part` Part to be initialized.

`instrumentNumber`
Number of instrument to initialize part with.

function result A result code.

DISCUSSION

You can use the `MusicFindTone` function (page 133) to find out an instrument number.

This function is superseded by `MusicSetPartInstrumentNumberInterruptSafe`, which can be called at interrupt time. You cannot call `MusicSetPartInstrumentNumber` at interrupt time.

MusicSetPartInstrumentNumberInterruptSafe

The `MusicSetPartInstrumentNumberInterruptSafe` function initializes a part with a particular instrument.

```
pascal ComponentResult MusicSetPartInstrumentNumber(
                    MusicComponent mc,
                    long part,
                    long instrumentNumber);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`part` Part to be initialized.

`instrumentNumber` Number of instrument to initialize part with.

function result A result code.

DISCUSSION

You can use the `MusicFindTone` function (page 133) to find out an instrument number.

You can call the `MusicSetPartInstrumentNumberInterruptSafe` function at interrupt time.

MusicGetPartAtomicInstrument

The `MusicGetPartAtomicInstrument` function returns the atomic instrument currently in a part.

```
extern pascal ComponentResult MusicGetPartAtomicInstrument(
                    MusicComponent mc,
                    long part,
                    AtomicInstrument *ai,
                    long flags);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	The part with the atomic instrument.
`ai`	On exit, an atomic instrument.
`flags`	Specify what pieces of information about an atomic instrument the caller is interested in. See “Atomic Instrument Information Flags” (page 63).
function result	A result code.

MusicSetPartAtomicInstrument

The `MusicSetPartAtomicInstrument` function initializes a part with an atomic instrument.

```
extern pascal ComponentResult MusicSetPartAtomicInstrument(
                    MusicComponent mc,
                    long part,
                    AtomicInstrumentPtr aiP,
                    long flags);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	The part to initialize with the atomic instrument to.
`aiP`	The atomic instrument.

`flags`	These flags specify details of initializing a part with an atomic instrument. See “Flags for Setting Atomic Instruments” on page 63.
function result	A result code.

MusicStorePartInstrument

The `MusicStorePartInstrument` function puts whatever instrument is on the specified part into the synthesizer’s instrument store. This enables you to store modified instruments.

```
pascal ComponentResult MusicStorePartInstrument(
                    MusicComponent mc,
                    long part,
                    long instrumentNumber);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	Part containing the instrument to be stored.
`instrumentNumber`	Instrument number at which to store the part.
function result	A result code.

DISCUSSION

The value of the `InstrumentNumber` parameter must be between 1 and the synthesizer’s modifiable instrument count, as defined by the `modifiableInstrumentCount` field of the synthesizer’s description structure.

MusicGetInstrumentAboutInfo

The `MusicGetInstrumentAboutInfo` function gets the information about an instrument that appears in its About box.

```
pascal ComponentResult MusicGetInstrumentAboutInfo(
                    MusicComponent mc,
                    long part,
                    InstrumentAboutInfo *iai);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`part` Number of the part containing the instrument for which you want information.

`iai` On exit, a pointer to an instrument About information structure (page 78) for the instrument currently on the specified synthesizer part.

MusicGetInstrumentInfo

The `MusicGetInstrumentInfo` function gets a list of instruments supported by a synthesizer. It also gets the names of the instruments.

```
extern pascal ComponentResult MusicGetInstrumentInfo(
                    MusicComponent mc,
                    long getInstrumentInfoFlags,
                    InstrumentInfoListHandle *infoListH);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`getInstrumentInfoFlags`
Use these flags to specify whether you want a list of fixed instruments, modifiable instruments, or all instruments. See “Instrument Info Flags” (page 64).

`infoListH` On exit, the list of instruments (page 80).

function result A result code.

Note
This handle must be disposed of by the caller.

DISCUSSION

The functions takes a music component in the `mc` parameter and instructions regarding which types of instruments to get information for in the `getInstrumentNamesFlags` parameter. It returns a handle to an instrument information list in the `infoListH` parameter.

MusicGetPart

The `MusicGetPart` function returns the MIDI channel and maximum polyphony for a particular part in the `MIDIChannel` and `polyphony` parameters.

```
pascal ComponentResult MusicGetPart(
                    MusicComponent mc,
                    long part,
                    long *MIDIChannel,
                    long *polyphony);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	The music component part requested.
`MIDIChannel`	On exit, a pointer to a MIDI channel.
`polyphony`	On exit, a pointer to the maximum polyphony.
function result	A result code.

DISCUSSION

For non-MIDI devices, the MIDI channel pointed to by the `MIDIChannel` parameter is 0.

MusicSetPart

The `MusicSetPart` function sets the MIDI channel and maximum polyphony for the specified part to the values in the `MIDIChannel` and `polyphony` parameters.

```
pascal ComponentResult MusicSetPart(
                    MusicComponent mc,
                    long part,
                    long MIDIChannel,
                    long polyphony);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	Part whose MIDI channel and polyphony are to be set.
`MIDIChannel`	The MIDI channel to set the part to.
`polyphony`	The maximum voices or polyphony for the part.
function result	A result code.

DISCUSSION

For non-MIDI devices, set the MIDI channel pointed to by the `MIDIChannel` parameter to 0.

MusicGetPartName

The `MusicGetPartName` function returns the string name of a part.

```
pascal ComponentResult MusicGetPartName(
                    MusicComponent mc,
                    long part,
                    StringPtr name);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	Part to get name of.

`name`	On exit, the string containing the part name.
function result	A result code.

DISCUSSION

The name string is used by selection dialog boxes or configuration information.

MusicSetPartName

You can use the `MusicSetPartName` function to change the name of an instrument in a specified part. For example, you might want to change the name of a modified instrument before saving it.

```
pascal ComponentResult MusicSetPartName(
                    MusicComponent mc,
                    long part,
                    StringPtr name);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	Part to apply name to.
`name`	Name to apply to part.
function result	A result code.

DISCUSSION

The instrument name string is used by selection dialog boxes or in configuration information.

MusicGetPartKnob

The `MusicGetPartKnob` function gets the current value of a knob for a part.

```
pascal ComponentResult MusicGetPartKnob(
                    MusicComponent mc,
                    long part,
                    long knobID);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	The part number.
`knobID`	The knob index or ID.
function result	Positive or negative integers are knob values. Result codes are returned as `0x8000xxxx`, where `xxxx` is the result code.

MusicSetPartKnob

The `MusicSetPartKnob` function sets a knob for a specified part.

```
pascal ComponentResult MusicSetPartKnob(
                    MusicComponent mc,
                    long part,
                    long knobID,
                    long knobValue);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`part`	The part number.
`knobID`	The index or ID of the knob to be set.
`knobValue`	The value to set the knob to.
function result	A result code.

MusicResetPart

The `MusicResetPart` function silences all sounds on the specified part, and resets all controllers on that part to their default values. .

```
pascal ComponentResult MusicResetPart(
                    MusicComponent mc,
                    long Part);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`part` The number of the part.

function result A result code.

DISCUSSION

The default value is 0 for all controllers except volume. Volume is set to its maximum 32767 or, in hexadecimal, 7FFF.

MusicGetPartController

The `MusicGetPartController` function returns the value of the specified controller on the specified part.

```
pascal ComponentResult MusicGetPartController(
                    MusicComponent mc,
                    long part,
                    MusicController controllerNumber);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`part` Part whose controller value you want to get.

`controllerNumber`
On exit, the controller number. For a list of controller numbers, see “Controller Numbers” (page 56).

function result A result code.

MusicSetPartController

The `MusicSetPartController` function initializes the value of the specified controller on the specified part.

```
pascal ComponentResult MusicSetPartController(
                    MusicComponent mc,
                    long part,
                    MusicController controllerNumber,
                    long controllerValue);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`part` Part whose controller value you want to set.

`controllerNumber`
Controller number. For valid values see “Controller Numbers” (page 56).

`controllerValue`
Value for controller.

function result A result code.

MusicSetPartSoundLocalization

The `MusicSetPartSoundLocalization` function passes sound localization data to a specified synthesizer part.

```
extern pascal ComponentResult MusicSetPartSoundLocalization(
                    MusicComponent mc,
                    long part,
                    Handle data);
```

`mc` Music component instance identifier.

`part` The part to pass the data to.

`data` The sound localization data.

function result A result code.

Music Component Functions: Miscellaneous

Use the functions described in this section to get and modify the master tuning of the synthesizer, to play off line, and to allow the music component to perform tasks it must perform at foreground task time.

MusicGetMasterTune

The `MusicGetMasterTune` function returns a fixed-point value in semitones, which is the synthesizer's master tuning.

```
pascal ComponentResult MusicGetMasterTune (MusicComponent mc);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

function result The function returns a positive value representing the synthesizer's master tuning or a negative result code.

MusicSetMasterTune

The `MusicSetMasterTune` function alters the synthesizer's master tuning.

```
pascal ComponentResult MusicSetMasterTune(
                    MusicComponent mc,
                    long masterTune);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`masterTune` The amount by which to transpose the entire synthesizer in pitch. The value is a fixed 16.16 number that allows shifts by fractional values.

function result A result code.

MusicStartOffline

The `MusicStartOffline` function informs the QuickTime music synthesizer that the music will not be played through the speakers. Instead, audio data will be sent to a function that will create a sound file to be played back later.

```
extern pascal ComponentResult MusicStartOffline(
                    MusicComponent mc,
                    unsigned long *numChannels,
                    UnsignedFixed *sampleRate,
                    unsigned short *sampleSize,
                    MusicOfflineDataUPP dataProc,
                    long dataProcRefCon);
```

`mc` Music component instance identifier returned by `NAGetRegisteredMusicDevice`.

`numChannels` Number of channels in the music sample. 1 indicates monaural; 2 indicates stereo.

`sampleRate` The number of samples per second.

`sampleSize` The size of the music sample: 8-bit or 16-bit.

`dataProc` A function to handle the audio data.

`dataProcRefCon` A reference constant to pass to the `dataProc` function.

function result A result code.

DISCUSSION

You pass the `MusicStartOffline` function the requested values for the `numChannels`, `sampleRate`, and `sampleSize` parameters. When the function returns, those parameters contain the actual values used.

MusicSetOfflineTimeTo

The `MusicSetOfflineTimeTo` function advances the synthesizer clock when the synthesizer is not running in real time (due to a call to `MusicStartOffline`).

```
extern pascal ComponentResult MusicSetOfflineTimeTo(
                    MusicComponent mc,
                    long newTimeStamp);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
`newTimeStamp`	The number of samples to synthesize.
function result	A result code.

DISCUSSION

Setting the time generates audio output from the synthesizer.

MusicTask

Call the `MusicTask` function periodically to allow a music component to perform tasks it must perform at foreground task time.

```
extern pascal ComponentResult MusicTask (MusicComponent mc);
```

`mc`	Music component instance identifier returned by `NAGetRegisteredMusicDevice`.
function result	A result code.

DISCUSSION

In the case of the QuickTime music synthesizer, instruments cannot be loaded from disk at interrupt time, so if the `NASetInstrumentNumberInterruptSafe` function is called, the instrument is loaded during the next `MusicTask` call.

Instrument Component Functions

This section describes functions that are implemented by instrument components.

InstrumentGetInfo

The `InstrumentGetInfo` function returns information about all the atomic instruments supported by an instrument component.

```
extern pascal ComponentResult InstrumentGetInfo(
                    ComponentInstance ci,
                    long getInstrumentInfoFlags,
                    InstCompInfoHandle *instInfo);
```

`ci` The instrument component instance. You obtain the identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`getInstrumentInfoFlags`
Use these flags to specify whether you want a list of fixed instruments, modifiable instruments, or all instruments. See "Instrument Info Flags" (page 64).

`instInfo` On exit, an instrument information list (page 83).

function result A result code.

InstrumentGetInst

The `InstrumentGetInst` function returns an atomic instrument.

```
extern pascal ComponentResult InstrumentGetInst(
                    ComponentInstance ci,
                    long instID,
                    AtomicInstrument *atomicInst,
                    long flags);
```

`ci`	The instrument component instance. You obtain the identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
`instID`	The instrument component instrument ID from the information list structure returned by the `InstrumentGetInfo` function.
`atomicInst`	On exit, the atomic instrument.
`flags`	Specifies what pieces of information about an atomic instrument the caller is interested in. See "Atomic Instrument Information Flags" (page 63).
function result	A result code.

InstrumentInitialize

Used by developers of instrument components, this is a call the instrument component makes to the base class instrument component to tell it how to interpret the instrument component resources.

```
extern pascal ComponentResult InstrumentInitialize(
                    ComponentInstance ci,
                    long initFormat,
                    void *initParams);
```

`ci`	An instrument component instance. You obtain the identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.
`initFormat`	Set to zero.
`initParams`	Set to `nil`.
function result	A result code.

InstrumentOpenComponentResFile

The `InstrumentOpenComponentResFile` function opens the resource file containing the instruments in the instrument component and makes it the current resource file.

```
extern pascal ComponentResult InstrumentOpenComponentResFile(
                    ComponentInstance ci,
                    short *resFile);
```

`ci` The instrument component instance. You obtain the identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`resFile` On exit, a resource reference.

function result A result code.

InstrumentCloseComponentResFile

The `InstrumentCloseComponentResFile` function closes a resource file.

```
extern pascal ComponentResult InstrumentCloseComponentResFile(
                    ComponentInstance ci,
                    short resFile);
```

`ci` The instrument component instance. You obtain the identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`resFile` A reference to the resource file that was returned previously by the `InstrumentOpenComponentResFile` function.

function result A result code.

InstrumentGetComponentRefCon

The `InstrumentGetComponentRefCon` function gets the reference constant for an instrument component.

```
extern pascal ComponentResult InstrumentGetComponentRefCon(
                    ComponentInstance ci,
                    void **refCon);
```

`ci` The instrument component instance. You obtain the identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`refCon` A reference constant.

function result A result code.

InstrumentSetComponentRefCon

Use the `InstrumentSetComponentRefCon` function to override the Component Manager `SetComponentRefCon` function and set the instrument component's reference constant to a specified value.

```
extern pascal ComponentResult InstrumentSetComponentRefCon(
                    ComponentInstance ci,
                    void *refCon);
```

`ci` The instrument component instance. You obtain the identifier from the Component Manager's `OpenComponent` function. See the chapter "Component Manager" in *QuickTime 3 Reference* for details.

`refCon` A reference constant.

function result A result code.

MIDI Component Functions

This section describes the functions that are implemented by MIDI components.

These functions implemented by MIDI components are MIDI device drivers, and are called by the note allocator MIDI routines.

Note
`NAGetMIDIPorts` is the correct call for you to make. You should *not* call `QTMIDI`. ◆

QTMIDIGetMIDIPorts

You use the `QTMIDIGetMIDIPorts` function to get two lists of MIDI ports supported by the specified MIDI component: a list of ports that can receive MIDI input and a list of ports that can send MIDI output.

```
pascal ComponentResult QTMIDIGetMIDIPorts (
      QTMIDIComponent ci,
      QTMIDIPortListHandle *inputPorts,
      QTMIDIPortListHandle *outputPorts);
```

`ci` Specifies the instance of a MIDI component. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function. See the "Component Manager" chapter in *QuickTime 3 Reference.*

`inputPorts` A list of the MIDI ports supported by the component that can receive MIDI input.

`outputPorts` A list of the MIDI ports supported by the component that can send MIDI output.

DISCUSSION

The caller of this function must dispose of the `inputPorts` and `outputPorts` handles.

QTMIDISendMIDI

You use the `QTMIDISendMIDI` function to send MIDI data to a MIDI port.

```
pascal ComponentResult QTMIDISendMIDI (
      QTMIDIComponent ci,
      long portIndex,
      MusicMIDIPacket *mp);
```

`ci` Specifies the instance of a MIDI component. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function. See the "Component Manager" chapter in *QuickTime 3 Reference.*

`portIndex` The index of the MIDI port to use for this operation.

`mp` A pointer to the MIDI data packet to send.

DISCUSSION

The `QTMIDISendMIDI` function can be called at interrupt time. However, the same interrupt level is used whenever MIDI data is sent by the specified MIDI component.

QTMIDIUseReceivePort

You use the `QTMIDIUseReceivePort` function to allocate a MIDI port for input or to release the port.

```
pascal ComponentResult QTMIDIUseReceivePort (
      QTMIDIComponent ci,
      long portIndex,
      MusicMIDIReadHookUPP readHook,
      long refCon);
```

`ci` Specifies the instance of a MIDI component. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function. See the "Component Manager" chapter in *QuickTime 3 Reference.*

`portIndex`	The index of the MIDI port to use for this operation.
`readHook`	A pointer to a function in your software that receives incoming MIDI data packets, or `nil` to release the port.
`refCon`	A reference constant passed to the function specified by the`readHook` parameter.

DISCUSSION

The MIDI component delivers only MIDI data packets that contain only a single status byte.

QTMIDIUseSendPort

You use the `QTMIDIUseSendPort` function to allocate a MIDI port for output or to release the port.

```
pascal ComponentResult QTMIDIUseSendPort (
      QTMIDIComponent ci,
      long portIndex,
      long inUse);
```

`ci`	Specifies the instance of a MIDI component. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function. See the "Component Manager" chapter in *QuickTime 3 Reference.*
`portIndex`	The index of the MIDI port for this operation.
`inUse`	Specifies whether to allocate the MIDI port for output (if the value is `1`) or to release the port (if the value is `0`).

Functions for Importing MIDI Files

This section describes functions you use to control the importation of MIDI files.

MIDIImportGetSettings

You use the `MIDIImportGetSettings` function to get settings that control the importation of MIDI files.

```
pascal ComponentResult MIDIImportGetSettings (
      TextExportComponent ci,
      long *setting);
```

`ci` Specifies the instance of the text export component used to import a MIDI file. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function. See the "Component Manager" chapter in *QuickTime 3 Reference.*

`setting` Flags that control the importation of MIDI files. These flags are described in "MIDI File Import Flags" (page 70).

DISCUSSION

The flags correspond to the checkboxes in the MIDI Import Options dialog box.

MIDIImportSetSettings

You use the `MIDIImportSetSettings` function to set settings that control the importation of MIDI files.

```
pascal ComponentResult MIDIImportSetSettings (
      TextExportComponent ci,
      long setting);
```

`ci` Specifies the instance of the text export component used to import a MIDI file. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function. See the "Component Manager" chapter in *QuickTime 3 Reference.*

`setting` Flags that control the importation of MIDI files. These flags are described in "MIDI File Import Flags" (page 70).

DISCUSSION

The flags correspond to the checkboxes in the MIDI Import Options dialog box.

Function Provided by the Generic Music Component

The generic music component implements the following function that a client music component can call.

MusicGenericConfigure

You use the `MusicGenericConfigure` function to tell the generic music component what services your music component requires and to point to any resources that are necessary.

```
pascal ComponentResult MusicGenericConfigure (
      MusicComponent mc,
      long mode,
      long flags,
      long baseResID);
```

`mc` Specifies the instance of the generic music component. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function. See the "Component Manager" chapter in *QuickTime 3 Reference.*

`mode` Must be `0`.

`flags` Flags that control the importation of MIDI files.

`baseResID` The resource ID of the lowest-numbered resource used by your music component.

These are the possible flags for the `flags` parameter:

`kGenericMusicDoMIDI`
Implement normal MIDI messages for note, controllers, and program changes 0–127.

`kGenericMusicBank0...kGenericMusicBank32`
If `kGenericMusicBank0` is set, then bank changes for instruments numbered above 127 will be sent on controller

zero; if `kGenericMusicBank32`, then on controller 32. If both flags are set, then the bank is sent on controller zero, and then a zero value is sent to controller 32

`kGenericMusicErsatzMIDI`
: Some musical devices, such as NuBus cards, may internally be driven by a MIDI stream but should not appear to the user to be an external MIDI device. The `kGenericMusicErsatzMIDI` flag instructs the generic music component to allocate channels appropriately and construct MIDI packets. The MIDI packets are always sent to the routine `MusicDerivedMIDISend`, and never to an external MIDI port.

`kGenericMusicCallKnobs`
: Specifies that your music component should receive calls to its routine `MusicDerivedSetKnob` for changes to global or part knobs. This flag should be set if your component implements any knobs.

`kGenericMusicCallParts`
: Specifies that your music component should receive calls to its routine `MusicDerivedSetPart`, in order to alter a specific part's polyphony or, in the case of a MIDI device, MIDI channel number.

`kGenericMusicCallInstrument`
: Specifies that your music component should receive calls to its routine `MusicDerivedSetInstrument`, in order to set a part to a new instrument. This is for devices that support complete user-instruments with knob lists. If this flag is not set, then the generic music component calls your music component many times to set the value of each knob in the instrument.

`kGenericMusicCallNumber`
: Directs the generic music component to call your music component's `MusicDerivedSetInstrumentNumber` function, rather than sending standard MIDI program-change and bank-change messages.

`kGenericMusicCallROMInstrument`
: Allows instruments that appear to the user as instruments built into the synthesizer to be stored in the derived component's resource file, as `'ROMi'` resources. The derived

component gets a call to `MusicDerivedSetInstrument` when one of these instruments is requested.

DISCUSSION

The `baseResID` parameter is the lowest resource ID used by your component for the standard resources described above. Since the resource numbers are relative to this, you can include several music components in a single system extension.

Functions Implemented by e Generic Music Component Clients

The following functions are implemented by client music components of the generic music component. They are called by the generic music component, which make calls that are necessary for responding to function calls made directly by applications.

MusicDerivedSetKnob

The generic music component calls your music component's `MusicDerivedSetKnob` function when any of the synthesizer's knobs are altered.

```
pascal ComponentResult MusicDerivedSetKnob(
                    MusicComponent mc,
                    long knobType,
                    long knobNumber,
                    long knobValue,
                    long partNumber,
                    GCPart *p,
                    GenericKnobDescription *gkd);

ComponentCallNow (kMusicDerivedSetKnobSelect,24);
```

`mc` Specifies the instance of the generic music component. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function. See the "Component Manager" chapter in *QuickTime 3 Reference.*

`knobType` Specifies the type of knob that has been altered.

knobNumber	Specifies the number of the knob that has been altered.
knobValue	Specifies the new value of the altered knob.
partNumber	Specifies the number of the part whose knob has been altered.
p	A pointer to the part whose knob has been altered.
gkd	A generic knob description structure for the knob.

DISCUSSION

This function is called when any knob on the synthesizer is altered. It should look at the Part structure and the GenericKnobDescription structure and address the synthesizer hardware appropriately to set the new knob value. For a MIDI device, this means to construct a system-exclusive MIDI packet and send it to the MIDI routine received by the MusicDerivedSetMIDI call.

These are the possible values for the knobType parameter:

```
#define kGenericMusicKnob 1
#define kGenericMusicInstrumentKnob 2
#define kGenericMusicDrumKnob 3
```

MusicDerivedSetPart

The generic music component calls your music component's MusicDerivedSetPart function to use the polyphony for the part specified in the Part structure.

```
pascal ComponentResult MusicDerivedSetPart (
                    MusicComponent mc,
                    long partNumber,
                    GCPart *p);

ComponentCallNow (kMusicDerivedSetPartSelect, 8);
```

mc	Specifies the instance of the generic music component. Your software obtains this reference when calling the Component Manager's OpenComponent or OpenDefaultComponent function.

partNumber	Specifies the number of the part for this operation.
p	A pointer to the part for this operation.

MusicDerivedSetInstrument

The generic music component calls your music component's MusicDerivedSetInstrument function to get the complete instrument defined by the Part structure to the synthesizer. This is either by hardware addressing in the case of a NuBus card, or by constructing a MIDI packet for an external synthesizer.

```
pascal ComponentResult MusicDerivedSetInstrument (
                    MusicComponent mc,
                    long partNumber,
                    GCPart *p);

ComponentCallNow (kMusicDerivedSetInstrumentSelect,8);
```

mc	Specifies the instance of the generic music component. Your software obtains this reference when calling the Component Manager's OpenComponent or OpenDefaultComponent function. See the "Component Manager" chapter in *QuickTime 3 Reference.*
partNumber	Specifies the number of the part for this operation.
p	A pointer to the part for this operation.

MusicDerivedSetInstrumentNumber

The generic music component calls your music component's MusicDerivedSetInstrumentNumber function to set the specified part to the instrument number in the Part structure.

```
pascal ComponentResult MusicDerivedSetInstrumentNumber (
                    MusicComponent mc,
                    long partNumber,
                    GCPart *p);
```

```
ComponentCallNow (kMusicDerivedSetInstrumentNumberSelect,8);
```

`mc`	Specifies the instance of the generic music component. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function. See the "Component Manager" chapter in *QuickTime 3 Reference*.
`partNumber`	Specifies the number of the part for this operation.
`p`	A pointer to the part for this operation.

DISCUSSION

For a MIDI device that either only supports instruments from 0 to 127 or that supports one of the standard bank-switching controller messages, this call should not be needed. You would set the `kGenericMusicBank0` or `kGenericMusicBank32` (or both) flags, instead.

MusicDerivedSetMIDI

The generic music component calls your music component's `MusicDerivedSetMIDI` function to set the MIDI channel and other MIDI settings for MIDI output only. It sends MIDI out to the synthesizer.

```
pascal ComponentResult MusicDerivedSetMIDI(
                    MusicComponent mc,
                    MusicMIDISendUPP midiProc,
                    long refcon,
                    long midiChannel);

ComponentCallNow (kMusicDerivedSetMIDISelect,12);
```

`mc`	Specifies the instance of the generic music component. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function. See the "Component Manager" chapter in *QuickTime 3 Reference*.
`midiProc`	A pointer to the function in your music component for performing MIDI output.

`refcon`	A reference constant sent to the function specified by the `midiProc` parameter.
`midiChannel`	Specifies the MIDI channel to use for the operation.

DISCUSSION

A derived component for a MIDI synthesizer receives this call soon after it is opened. It should store the `midiProc`, `refCon`, and `midiChannel` in its global variables. When the derived component needs to communicate with the synthesizer, it calls the `midiProc` with this reference constant. The `midiChannel` variable specifies the "system channel" of the device.

MusicDerivedStoreInstrument

The generic music component calls your music component's `MusicDerivedStoreInstrument` function to store the specified instrument in a user instrument location.

```
pascal ComponentResult MusicDerivedStoreInstrument (
                    MusicComponent mc,
                    long partNumber,
                    GCPart *p,
                    long instrumentNumber )

ComponentCallNow (kMusicDerivedStoreInstrumentSelect,8);
```

`mc`	Specifies the instance of the generic music component. Your software obtains this reference when calling the Component Manager's `OpenComponent` or `OpenDefaultComponent` function. See the "Component Manager" chapter in *QuickTime 3 Reference.*
`partNumber`	Specifies the number of the part for this operation.
`p`	A pointer to the part for this operation.
`instrumentNumber`	Specifies the number of the instrument to store.

Result Codes

This section lists all the result codes returned by QuickTime music architecture functions.

NOTIMPLEMENTEDMUSICOSERR	-2071	Call to a routine that is not supported by a particular music component.
CANTSENDTOSYNTHESIZEROSERR	-2072	Attempt to use a synthesizer before it has been initialized, given a MIDI port to use, or told which slot card to use. For example, the `MusicSetMIDIProc` function has not been called.
ILLEGALVOICEALLOCATIONOSERR	-2074	Attempt to allocate more voices than a synthesizer supports.
ILLEGALPARTOSERR	-2075	Usually indicates use of a part number parameter outside the range 1…`partcount`.
ILLEGALCHANNELOSERR	-2076	Attempt to use a MIDI channel outside the range 1…16.
ILLEGALKNOBOSERR	-2077	Attempt to use a knob index or knob ID that is not valid.
ILLEGALKNOBVALUEOSERR	-2078	Attempt to set a knob outside its allowable range, as specified in its knob description structure.
ILLEGALINSTRUMENTOSERR	-2079	Attempt to use an instrument or sound that is not available or there is some other problem with the instrument, such as a bad instrument number.
ILLEGALCONTROLLEROSERR	-2080	Attempt to get or set a controller that is outside the allowable controller number range or is not recognized by this particular music component.
MIDIMANAGERABSENTOSERR	-2081	Attempt to use MIDI Manager for a synthesizer when the MIDI Manager is not installed.
SYNTHESIZERNOTRESPONDINGOSERR	-2082	Various hardware problems with a synthesizer.
SYNTHESIZEROSERR	-2083	Software problem with a synthesizer.
ILLEGALNOTECHANNELOSERR	-2084	Attempt to use a note channel that is not initialized or is otherwise errant.

NOTECHANNELNOTALLOCATEDOSERR	-2085	It was not possible to allocate a note channel.
TUNEPLAYERFULLOSERR	-2086	Attempt to queue up more tune segments (with `TuneQueue`) than allowed.
TUNEPARSEOSERR	-2087	`TuneSetHeader` or `TuneQueue` encountered illegal tune sequence data.

APPENDIX A

General MIDI Reference

General MIDI Instrument Numbers

Table A-1 General MIDI instrument numbers

Number	Instrument	Number	Instrument
1	Acoustic Grand Piano	65	Soprano Sax
2	Bright Acoustic Piano	66	Alto Sax
3	Electric Grand Piano	67	Tenor Sax
4	Honky-tonk Piano	68	Baritone Sax
5	Rhodes Piano	69	Oboe
6	Chorused Piano	70	English Horn
7	Harpsichord	71	Bassoon
8	Clavinet	72	Clarinet
9	Celesta	73	Piccolo
10	Glockenspiel	74	Flute
11	Music Box	75	Recorder
12	Vibraphone	76	Pan Flute
13	Marimba	77	Bottle Blow
14	Xylophone	78	Shakuhachi
15	Tubular bells	79	Whistle
16	Dulcimer	80	Ocarina
17	Draw Organ	81	Square Lead

Table A-1 General MIDI instrument numbers

Number	Instrument	Number	Instrument
18	Percussive Organ	82	Saw Lead
19	Rock Organ	83	Calliope
20	Church Organ	84	Chiffer
21	Reed Organ	85	Synth Lead 5
22	Accordion	86	Synth Lead 6
23	Harmonica	87	Synth Lead 7
24	Tango Accordion	88	Synth Lead 8
25	Acoustic Nylon Guitar	89	Synth Pad 1
26	Acoustic Steel Guitar	90	Synth Pad 2
27	Electric Jazz Guitar	91	Synth Pad 3
28	Electric Clean Guitar	92	Synth Pad 4
29	Electric Guitar Muted	93	Synth Pad 5
30	Overdriven Guitar	94	Synth Pad 6
31	Distortion Guitar	95	Synth Pad 7
32	Guitar Harmonics	96	Synth Pad 8
33	Wood Bass	97	Ice Rain
34	Electric Bass Fingered	98	Soundtracks
35	Electric Bass Picked	99	Crystal
36	Fretless Bass	100	Atmosphere
37	Slap Bass 1	101	Bright
38	Slap Bass 2	102	Goblin
39	Synth Bass 1	103	Echoes
40	Synth Bass 2	104	Space
41	Violin	105	Sitar
42	Viola	106	Banjo

Table A-1 General MIDI instrument numbers

Number	Instrument	Number	Instrument
43	Cello	107	Shamisen
44	Contrabass	108	Koto
45	Tremolo Strings	109	Kalimba
46	Pizzicato Strings	110	Bagpipe
47	Orchestral Harp	111	Fiddle
48	Timpani	112	Shanai
49	Acoustic String Ensemble 1	113	Tinkle Bell
50	Acoustic String Ensemble 2	114	Agogo
51	Synth Strings 1	115	Steel Drums
52	Synth Strings 2	116	Woodblock
53	Aah Choir	117	Taiko Drum
54	Ooh Choir	118	Melodic Tom
55	Synvox	119	Synth Tom
56	Orchestra Hit	120	Reverse Cymbal
57	Trumpet	121	Guitar Fret Noise
58	Trombone	122	Breath Noise
59	Tuba	123	Seashore
60	Muted Trumpet	124	Bird Tweet
61	French Horn	125	Telephone Ring
62	Brass Section	126	Helicopter
63	Synth Brass 1	127	Applause
64	Synth Brass 2	128	Gunshot

General MIDI Drum Kit Numbers

Table A-2 General MIDI drum kit numbers

35	Acoustic Bass Drum	51	Ride Cymbal 1
36	Bass Drum 1	52	Chinese Cymbal
37	Side Stick	53	Ride Bell
38	Acoustic Snare	54	Tambourine
39	Hand Clap	55	Splash Cymbal
40	Electric Snare	56	Cowbell
41	Lo Floor Tom	57	Crash Cymbal 2
42	Closed Hi Hat	58	Vibraslap
43	Hi Floor Tom	59	Ride Cymbal 2
44	Pedal Hi Hat	60	Hi Bongo
45	Lo Tom Tom	61	Low Bongo
46	Open Hi Hat	62	Mute Hi Conga
47	Low Mid Tom Tom	63	Open Hi Conga
48	Hi Mid Tom Tom	64	Low Conga
49	Crash Cymbal 1	65	Hi Timbale
50	Hi Tom Tom	66	Lo Timbale

General MIDI Kit Names

Table A-3 General MIDI kit names

1	Dry Set
9	Room Set
19	Power Set
25	Electronic Set
33	Jazz Set
41	Brush Set
65-112	User Area
128	Default

APPENDIX

General MIDI Reference