Apple Game Sprockets Guide

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

Simultaneously published in the United States and Canada.

Contents

Figures and Listings

About This Book

Apple Game Sprockets is a set of libraries that enhance your ability to create games for the Macintosh or any computer running the Mac OS. This book introduces you to the features of Apple Game Sprockets and helps you get started programming games in a new and easier way. This book contains four chapters, each corresponding to one of the four libraries that define the Apple Game Sprockets.

The four libraries in Apple Game Sprockets are SoundSprocket, DrawSprocket, InputSprocket, and NetSprocket. These libraries offer games developers several advantages over existing approaches to game development.

- SoundSprocket offers new technologies for presenting sounds localized in 3D space.
- DrawSprocket provides safe ways to manage the display and manipulate graphics, replacing the undocumented methods currently in use. It provides a consistent user experience for configuring game play. It also makes page flipping—long a capability of Apple hardware—accessible to games.
- InputSprocket provides a consistent user experience for configuring how input devices work with games.
- NetSprocket offers a common user interface for hosting and joining network games. Messages are sent and received across a variety of protocols with a simple programming interface.

Apple Game Sprockets alleviates the burden of tedious common details and allows you to concentrate on developing the game play.

Developers of input devices also benefit from Apple Game Sprockets. Apple Game Sprockets lets you concentrate solely on providing access to the device while leaving plenty of room for differentiation and adding value.

You can use the four libraries independently or together, as your game requires. These libraries are delivered to the end users as system extensions, similar to QuickTime or OpenDoc. For example, if your game requires the SoundSprocket library, you can include it as a system extension with your game. Apple Game Sprockets may be used without a license fee and the SDK and runtime libraries are available on Apple's Web site.

These libraries are useful in non-game applications as well. You could use SoundSprocket in a sound-effects processing program and DrawSprocket in applications, such as presentation applications, that need to take over the screen—particularly if animation is involved. Applications with simple networking needs could benefit from NetSprocket. InputSprocket could be used for handicapped-access devices.

SoundSprocket Overview

SoundSprocket is defined in the interface file SoundSprocket.h. Its system extension is named SoundSprocketLib. SoundSprocket provides real-time, 3-D spatial filtering of sounds. You can use SoundSprocket to make a sound appear to emanate from its sound source on the screen. For example, a sound can appear to move from left to right or move closer to the user as its source moves.

SoundSprocket can simulate the Doppler effect, distance attenuation, environmental reverberation, and position in space. Your game does not need to know whether the sounds you want to play are being filtered with hardware- or software-based methods.

SoundSprocket uses a virtual audio environment to help you create the spatial sound effects you want. A user's position and possible movement can be defined in a listener so you can filter the sound appropriately. One or more sound sources can also be defined so you can manipulate each source's location and possible movement in detail. SoundSprocket provides a localization component to handle the 3D filtering as the user is playing your game.

SoundSprocket provides both high-level and low-level interfaces for managing 3D sound filtering. The low-level interface provides a message interface to 3D localization sound components. The high-level interface provides a collection of functions that help you fill in the messages that you send to the 3D localization sound components. The high-level interfaces are integrated with, but don't depend on, QuickDraw 3D.

DrawSprocket Overview

DrawSprocket is defined in the interface file DrawSprocket.h. Its system extension is named DrawSprocketLib. DrawSprocket allows you to take over the screen, customize it to best fit your game's content, and display graphics at cinematic frame rates.

With DrawSprocket your game can easily configure the display, taking advantage of any special display features of the system. You don't need to worry about the system meeting the display needs of your game—any features not directly supported in hardware are emulated by software.

A blanking window cuts through any problems you may have had in hiding the desktop during game play. After you configure the display and are ready to begin play, DrawSprocket completely hides the desktop, menu bar, and other system resources and changes the monitor to the best resolution and pixel depth, giving your game a clean slate. When the game's over, DrawSprocket automatically restores everything to its former state.

DrawSprocket provides double buffering or triple buffering and gamma fading to make your game animations smooth and transitions seamless. Double-buffering is a way to emulate page flipping. If page flipping is available on the hardware, DrawSprocket can provide transparent support for it. Gamma fading achieves a non-linear fade so that bright colors don't remain visible after dark colors disappear. It also allows you to fade out to other colors besides black.

Other DrawSprocket features allow you to provide overlays and underlays, use pixel scaling, set a maximum frame refresh rate so your game runs at a constant speed, and easily manipulate colors in a color lookup table. During development, you can use the special debugger feature to keeps system resources visible, even behind the blanking window.

InputSprocket Overview

InputSprocket is defined in the interface files InputSprocket.h. Its system extension is named InputSprocketLib. InputSprocket makes it easy for your game to synchronize itself with existing input devices, configure input devices, and track data from input devices.

With InputSprocket, you no longer have to emulate a joystick on the keyboard or mouse. The device drivers will give standard information about device controls. This makes it easy for games to configure control options. InputSprocket's input device architecture greatly simplifies interactions between games and input devices.

InputSprocket bases the communication between your game and an input device's controls on elements. Elements are the information that describe an input device's controls, such as its axes, thrust levers, buttons, and direction

pads. During game play, the game automatically determines the state and transitions of the elements through polling or a simple event queue without using `GetNextEvent`.

NetSprocket Overview

NetSprocket is defined in the interface file NetSprocket.h. Its system extension is named NetSprocketLib. NetSprocket provides high-performance data transmission between players that is simple to implement, allowing you to concentrate on the information you need to communicate between players rather than on the low-level details of networking.

The high-level networking functionality makes message routing transparent to the application and lets you specify best-effort or guaranteed message delivery depending on your needs. The protocol-independent model supports multiple transport protocols, such as AppleTalk, TCP/IP, and serial interfaces, based on Open Transport. The TCP/IP protocol allows game play across the Internet.

NetSprocket also handles fault-tolerance—a critical component of any network game architecture. NetSprocket automatically adjusts to the disappearance of a player due to game application crashes or network failure.

You can use NetSprocket's human interface for hosting and joining games or you can develop your own, using NetSprocket as a model. Also, NetSprocket automatically keeps track as players join or leave a game, updating its routing information and informing the other players of changes to the list of game participants.

Format of a Typical Chapter

Each chapter in this book follows a standard structure. Each begins with an overview, then gives programming examples, then lists all the functions in a reference section, and ends with a code summary. For example, the chapter "SoundSprocket" contains these sections:

- "About SoundSprocket." This section gives you a general overview of the SoundSprocket library and the features it provides.
- "Using SoundSprocket." This section describes the tasks you can accomplish using SoundSprocket and gives programming examples.

- "SoundSprocket Reference." This section provides a complete reference to the SoundSprocket library by describing its constants, data structures, and functions. Each function description also follows a standard format, which gives the function declaration and description of every parameter. Some function descriptions also give additional descriptive information, such as result codes.
- "Summary of SoundSprocket." This section provides SoundSprocket's C interface for its constants, data structures, functions, and result codes.

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. Certain types of information, such as parameter blocks, use special fonts so that you can scan them quickly.

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`).

Words that appear in **boldface** are key terms or concepts that are defined in the glossary.

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 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 game. 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.

If you use SoundSprocket, you will need the Sound Manager, version 3.2.1. SoundSprocket will not work with earlier versions of the Sound Manager. If you use NetSprocket, you will need Open Transport 1.1, which is in System 7.5.3. To use DrawSprocket, you need the Display Manager, version 2.0.

For More Information

The *Apple Developer Catalog* (ADC) is Apple Computer's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple computer platforms. Customers receive the *Apple Developer Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. ADC offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *Apple Developer Catalog*, contact

Apple Developer Catalog
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	ORDER.ADC
Internet	order.adc@applelink.apple.com

SoundSprocket

Contents

This chapter describes SoundSprocket, the part of Apple Game Sprockets that you can use to provide spatial filtering for sounds. For example, you can use SoundSprocket functions to make a sound appear to move from the listener's right to left as an image moves from right to left on the screen. Or, you can use SoundSprocket functions to make a sound appear to approach the listener at a specific speed from a specific direction.

Before reading this chapter, you should be generally familiar with Apple Game Sprockets, as described in the preface to this book. To use this chapter, you should already be familiar with the Sound Manager, as described in *Inside Macintosh: Sound*. You need to know how to create and manage sound channels, through which the spatially filtered sounds are played. You also need to use the Sound Manager to load and play sounds (which can be stored in resources or data files, or generated dynamically at run time).

To use this chapter, you might also want to be familiar with QuickDraw 3D. SoundSprocket uses some QuickDraw 3D data types (such as `TQ3Point3D`) to describe the location of sounds in space and to describe rotations or other manipulations on those locations. See *3D Graphics Programming With QuickDraw 3D* for a description of the data types borrowed by SoundSprocket.

This chapter begins by describing the basic capabilities of SoundSprocket. Then it shows how to use some of those capabilities to make sounds appear to move in three-dimensional space. The section "Summary of SoundSprocket" (page 1-74), provides a complete reference to the constants, data structures, and functions provided by SoundSprocket. This section also describes two Sound Manager functions, `SndGetInfo` and `SndSetInfo`, that were added to Sound Manager version 3.1 and which are used by SoundSprocket; see "Getting and Setting Sound Channel Information" (page 1-72) for details.

Note
This chapter describes the application programming interfaces supported by versions 1.0 and later of SoundSprocket. ◆

About SoundSprocket

SoundSprocket is the part of Apple Game Sprockets that provides spatial (that is, three-dimensional) filtering for sounds. SoundSprocket extends the

capabilities of the Sound Manager so that you can make some sounds appear to be emanating from a specific location and distance from the user, possibly also moving in space.

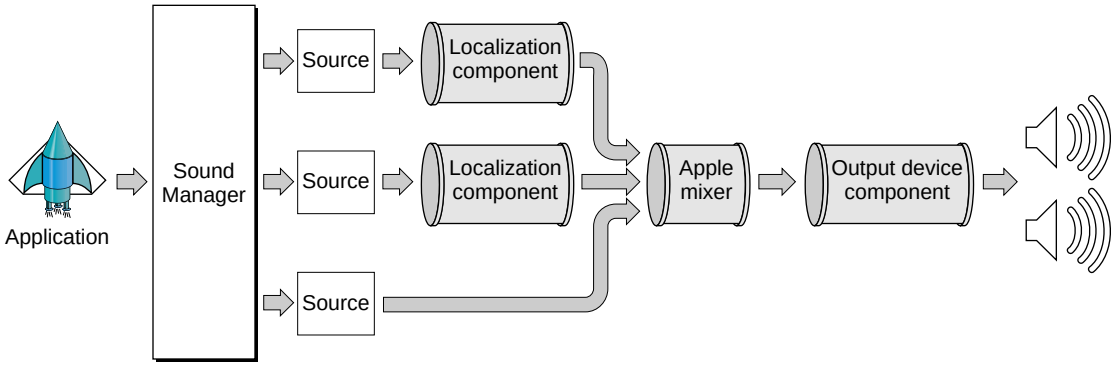
To perceive the spatial filtering provided by SoundSprocket, the listener should have some sort of stereophonic sound output hardware. The hardware can be a set of external speakers (such as the AppleDesign Powered Speakers), the speakers built into a monitor (such as the Apple AudioVision 14 Display), or a set of headphones. When only one speaker is available, directional localization cannot be provided, but some other spatial cues (such as Doppler effect and both distance and angular attenuation) are still provided.

When stand-alone stereo speakers are used, for optimal enjoyment of the sound, they should be placed so they are equally spaced on opposite sides of the monitor, roughly at head level. Optimal for each speaker is approximately 30 degrees left and right from the listener when directly facing the monitor.

Performance is a key factor in most games. SoundSprocket is designed to work with both hardware- and software-based methods of 3D sound filtering. Your application (probably a game) does not need to know how the sounds you play are being filtered. When software-based 3D sound filtering is active, SoundSprocket provides a way to scale the fidelity of the output sound by increasing or reducing the CPU load caused by 3D sound filtering. The filtering algorithms offer discrete performance steps pairing CPU time with fidelity.

SoundSprocket provides both high-level and low-level interfaces for managing 3D sound filtering. The low-level interfaces allow you to install a localization component and send messages to it (using the Sound Manager) to control the sound filtering directly. The **localization component** is a sound component that provides 3D filtering for the sounds passed to it, as illustrated in Figure 1-1. To use the low-level interfaces, you must provide the sound coordinates in listener-relative, polar coordinates, and they must be provided in meters.

Figure 1-1 The position of the localization component



The high-level SoundSprocket interfaces consist principally of functions that you conveniently use to create and manipulate listeners and sound sources (which are described more fully below). The high-level interfaces simply fill in the data structures you pass to the Sound Manager's `SndGetInfo` and `SndSetInfo` functions to control the sound filtering. Coordinates for the high level functions are specified in Cartesian space (with three-dimensional *x*, *y* and *z* axes), either by position and orientation vectors, by transformation matrix, or by QuickDraw 3D camera positions. Unlike the low-level interface, any unit of measure can be used, and velocities may be specified directly or may be computed automatically.

IMPORTANT

Sound Manager version 3.2.1 or later is required for SoundSprocket. In addition, the SoundSprocket filter component must be present in the Extensions Folder at start-up time. (See "Getting and Setting Sound Channel Information" (page 1-72) for a description of the `SndGetInfo` and `SndSetInfo` functions.) ▲

The following sections describe in greater detail the capabilities provided by SoundSprocket.

The Virtual Audio Environment

SoundSprocket provides and manages a virtual audio environment for your application. The virtual audio environment consists of a single listener and one or more 3D sound sources in three-dimensional space. You should limit the number of sources to achieve a clear presentation of sound to your listener. Four sources are typical; let your ear be the final judge, not the arbitrary limit of four. The sources emit sounds that are heard by the listener. Both the listener and all its associated sound sources have a position and an orientation in three-dimensional space. In addition, both the listener and the sound sources have a number of properties that affect the way in which sounds are filtered. For example, a listener can have a velocity that contributes to audio effects such as Doppler shift when it moves relative to a sound source.

Note
An object's velocity is independent of its orientation. It's possible, for example, for an object to be oriented straight ahead while moving to the right. ◆

Each listener and each sound source has a **transformation matrix** (or **transform matrix**) associated with it that is applied to the position, orientation, and velocity of the object to determine its actual position, orientation, and velocity in the virtual audio environment. To move a listener or sound source, you can either change its position or its transformation matrix, or both.

The velocity vector associated with a listener or a sound source specifies the current direction and speed of the object. You can set the velocity vector directly, in which case the actual velocity is the specified velocity transformed by the transformation matrix. Alternatively, you can leave the velocity vector unchanged and allow SoundSprocket to compute the object's velocity by calculating the difference between the object's transformed position in successive updates and dividing that difference by the time elapsed between the updates.

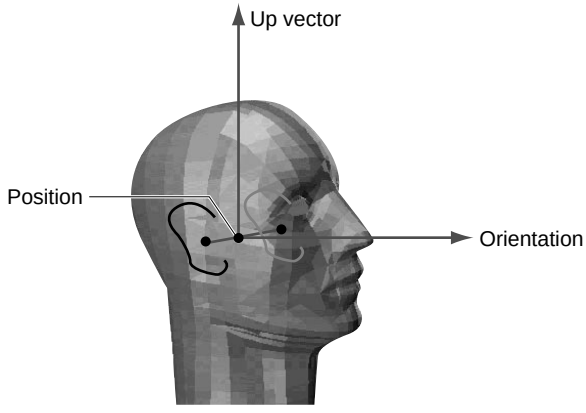
Each source also has additional characteristics. The source sound has a reference distance—when the sound source is this far from the listener, its distance filtering is unchanged. This enables the source to be very loud but very distant (or very quiet and very near). The source may be a volume (such as a cube) that is emitting sound uniformly within, for example a water fountain that makes noise from a broad space. In contrast, a source may emit sound more loudly in one direction than in others such as a space ship that emits noise primarily from the rear where the rocket is firing.

In addition to the audio effects due to the relative motions of a listener and its sound sources, SoundSprocket provides effects due to the medium through which sound is traveling, the relative humidity of the air, reverberation from the walls of the room the listener is in, the distances separating a listener from its sound sources, and the relative orientations of the listener and its sources.

Listeners

A **listener** is the part of the virtual audio environment that perceives sounds. A listener defines the position, orientation, and velocity of the user's "head" in the virtual audio environment. The **position** of a listener is the midpoint of the line connecting the two ears. The **orientation** of the listener is the unit vector that points forward (through the nose) from the listener's position. The **up vector** is the unit vector that points straight up through the top of the listener's head. Figure 1-2 illustrates these features.

Figure 1-2 The position, orientation, and up vector of a listener



A listener has a number of properties that define some global environmental characteristics, including the following:

- The **sound medium** through which sound is traveling. The sound medium determines the speed at which sounds travel and the distance attenuation due to the distance from the listener to a sound source. The default sound medium is dry air at standard temperature and pressure (namely, 20° C at

760 mm of mercury), but you can also specify water as the sound medium. Most games do not need to adjust the sound medium. When the medium is air, you can also specify the **relative humidity** of the air. Humidity is a floating-point value between 0.0 (indicating dry air) and 100.0 (indicating dense fog).

- The **reverberation** caused by sound bouncing off the walls of the room the listener is in. You can adjust the room size, the reflectivity of the reverberant walls, and the amount of reverberation to be mixed into the final output sound.

IMPORTANT

In SoundSprocket version 1.0, the localization component ignores any values you specify for a listener's sound medium and relative humidity. SoundSprocket assumes dry (zero humidity) air. ▲

When you use SoundSprocket's low-level interfaces, you specify all distances in meters. However, when you use SoundSprocket's high-level interfaces, you can choose the unit (called the **listener unit**) in which you want to specify distances**.** You can set the listener unit by calling the `SSpListener_SetMetersPerUnit` function to set the number of meters in the listener unit. For example, to specify all distances in feet, you would set the number of meters per listener unit to 0.3048. Similarly, to specify all distances in miles, you would set the number of meters per listener unit to 1609.3.

Sound Sources

A **3D sound source** (or, more briefly, a **sound source** or **source**) is a part of the virtual audio environment that emits sounds heard by the listener. A source has a position and orientation in three-dimensional space, as well as a velocity and other characteristics that affect the sounds produced by the source.

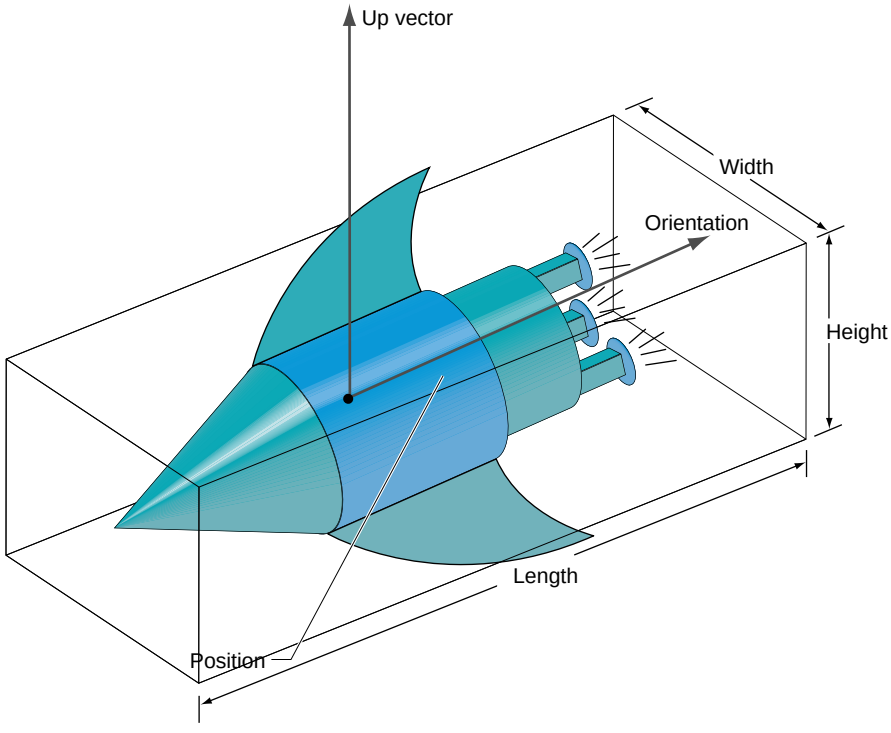
IMPORTANT

Do not confuse SoundSprocket's 3D sound sources with the Sound Manager's sound sources. (A Sound Manager sound source is the origin of a specific channel of sound.) ▲

A sound source has a bounding box that defines its size and position, as shown in Figure 1-3. The bounding box is available only via the high-level interface. The **position** of a sound source is the center of the bounding box. The length of

the source is measured along the orientation vector, and the height is measured along the up vector. The width of the source is measured along the remaining axis (which is perpendicular to both the orientation and up vectors). By default, a sound source is a point source (0.0 for all distances).

Figure 1-3 The position, orientation, and up vector of a sound source



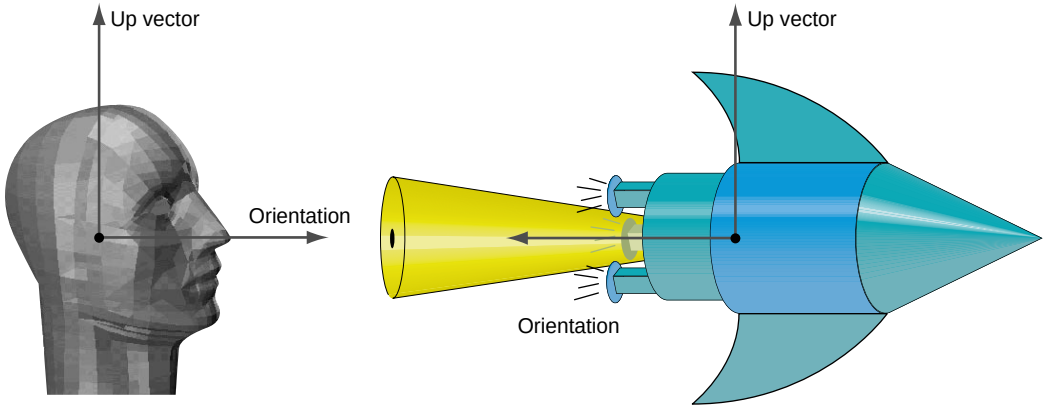
A sound source has a number of properties that define specific characteristics of the sound it emits, including the following:

- A **reference distance** that is the distance, in meters or listener units, from a listener to the point at which a sound was recorded. When the source is located exactly at its reference distance from the listener, there is no **distance attenuation** (that is, loss of volume due to distance) applied to the sound. When the source is farther away than the reference distance, the sound is

attenuated. When the sound is closer than the reference distance, the sound might be amplified.

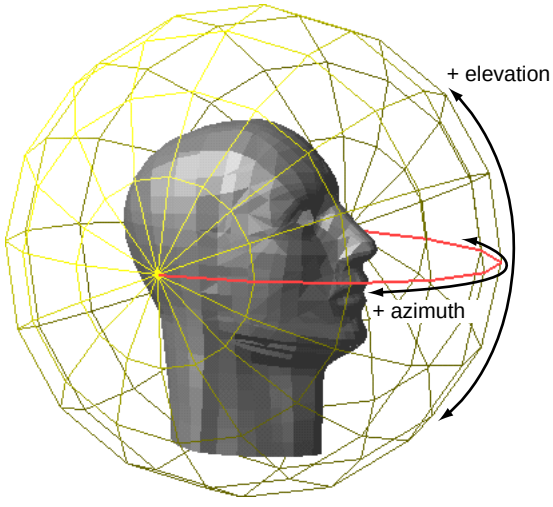
- A **source mode** that specifies the type of filtering that is to be applied to a source sound. By default, a source sound is **localized** in space (that is, made to appear at a specific location). Alternatively, you can specify that a sound be ambient (that is, made to appear to come from all directions). See "Source Modes" (page 1-28) for a complete description of the available source modes.
- An **angular attenuation cone** that determines the direction of maximum sound intensity. A sound source can emit sound louder in the direction of its orientation vector than in other directions. As the angle between the orientation vector and the source-to-listener vector increases toward a predefined limit (the cone angle), the sound amplitude is increasingly attenuated, up to the maximum angular attenuation. The default setting is for uniform attenuation in all directions, which you achieve by setting the cone attenuation to 0.0. Figure 1-4 illustrates the angular attenuation cone.
- In addition to reverberation for the listener, the source may have distinct localized reflections. These reflections are represented by virtual sources with a distinct position. This feature is not supported in version 1.0.

Figure 1-4 The angular attenuation cone



When you use the low-level interfaces, you specify the location of a sound using listener-relative polar coordinates, with all distances specified in meters. Figure 1-5 illustrates the low-level specification of the location of a 3D sound source.

Figure 1-5 Specifying the location of a sound source



As you can see, the coordinate system is like a globe with the poles at the listener's ears and the origin at the listener's nose.

When you use the high-level interfaces, however, you can specify the location of a sound using Cartesian coordinates or other equivalent means (such as QuickDraw 3D camera positions).

Optimizing Sound and Performance

Sounds that contain only high-pitched frequencies don't localize as well as full-spectrum sounds. Also, sounds that are recorded with reverb will have their reverb localized along with the sound itself. This is typically not the desired effect. The reverb effect that SoundSprocket provides is not intended for high-quality music processing.

Sounds played in localized or ambient mode should consist of a single monophonic channel. If a stereo sound is processed in these modes, it is first mixed down to a single channel, resulting in slightly lower performance. Sounds played in binaural mode should be stereo. It is a common practice in game programming to allocate fewer sound channels than there are sonic entities in the game. When a sound needs to be played, an sound channel that isn't busy is used. SoundSprocket can be used in this scenario as well. One thing that you need to do is make sure that any lingering localization parameters are overridden for the new sound. That is, you must send an `siSSpLocalization` message if there is any chance that the last one that was sent to the sound channel had parameters different from the desired ones.

When using the high-level API you may either keep a source per sound entity or a source per sound channel. If you choose the latter, and are letting SoundSprocket compute velocities for an entity, incorrect velocities are generated when the position jumps when the source is used for an entity in a different location. This problem is easily circumvented by setting the velocity of the source to (0,0,0) when it is given the discontinuous position

Experiencing Game Sound Via Speakers and Headphones

The sounds generated by your game are presented to the user through speakers or headphones. When speakers are used, they must be stereo speakers placed on either side of the monitor. A crosstalk cancellation filter allows the left signal to be heard predominantly by the left ear, and the right signal by the right ear. There is a sweet spot several inches across midway between the two speakers, allowing the user to comfortably move their head without losing the effect. If a single speaker is used, directional localization cues are unavailable.

If the user is using headphones during game play, delivering the left and right signals to the appropriate ear is easy, so the crosstalk cancellation filter is not used if headphones are specified. However, the localization effect may be diminished because the virtual audio environment moves with the user, somewhat muting the illusion of a stable environment.

The best of both worlds may be obtained by adding head tracking to the mix. This is a natural thing to do when the game is played on a head-mounted display. The listener position in the virtual audio environment can then be changed as the user’s position changes in the real world. In this instance, the illusion of a stable virtual audio environment is maintained, thus keeping sound quality extremely high.

Using SoundSprocket

To use SoundSprocket in a game, you need to create a listener and one or more sound sources. Localized sounds are played through normal sound channels, into which the localization component has been installed, as illustrated in Figure 1-1 (page 1-7). Then, while the game is under way, you should play the appropriate sounds, change the positions, orientations, and velocities of the sources and the listener as appropriate, and then call `SndSetInfo` to update the 3D sound filtering characteristics. To achieve a sense of continuous motion, you should update the filter characteristics at least several times per second. This section shows how to accomplish these tasks.

Note
The code examples shown in this section provide only rudimentary error handling. ◆

You need to explicitly create listeners and sound sources only when you use the high-level interfaces provided by SoundSprocket. You can, if you prefer, maintain the data for a sound source yourself and then use the low-level interfaces to send messages to the localization component. See "Using the Low-Level Interfaces" (page 1-23) for further information.

Configuring Sound Output Devices

Your game should provide a 3D sound configuration dialog box that allows the user to select from stereo or monophonic speakers, or headphones. If the user is using stereo speakers, you should provide a method to set the angle between the speakers. The game may provide this functionality directly, using the `siSSpSetup` constant, or you can use the `SSpConfigureSetup` function to display a modal dialog that allows the user to control the features. Using `SSpConfigureSetup` is recommended, because this allows Apple to change the contents of the dialog as the setup information evolves.

Note
These controls may be incorporated into a control panel in a future system software release. ◆

`SSpConfigureSetup` displays the modal dialog shown below that allows the user to view and alter the speaker configuration settings. It creates a sound channel, installs the filter, and alters the setup for all filters when the user adjusts the controls.

A sound effect is played while the user is adjusting the speaker angle slider. The slider in the 3D sound configuration dialog box should be adjusted so that the alternating sound effects appear to be as far to left and right of the user as possible. The game should not be playing any sounds when this function is called because it will interfere with the user's ability to adjust the slider for optimal sound enjoyment.

Figure 1-6 Controlling sound output devices

Installing the Localization Component

A localization component is a sound output component that provides 3D filtering for the sounds passed to it. You need to install an instance of the localization component into each sound channel for which you want SoundSprocket to produce 3D filtering. Listing 1-1 illustrates how to create a sound channel and install an instance of the localization component in it.

Listing 1-1 Creating a localized sound channel

```
SndChannelPtr MyCreateLocalizedChannel (void)
{
    SndChannelPtr       mySndChannel = nil;
    OSStatus            myErr;
    SoundComponentLink  myLink;

    /* create a new sound channel */
    myErr = SndNewChannel(mySndChannel, sampledSynth, initMono, nil);
    if (myErr != noErr) {
        return(nil);
    }

    /* define the type of the localization component */
    myLink.description.componentType = kSoundEffectsType;
    myLink.description.componentSubType = kSSpLocalizationSubType;
    myLink.description.componentManufacturer = 0;
    myLink.description.componentFlags = 0;
    myLink.description.componentFlagsMask = 0;
    myLink.mixerID = nil;
    myLink.linkID = nil;

    /* install the localization component BEFORE the Apple Mixer */
    myErr = SndSetInfo(mySndChannel, siPreMixerSoundComponent, &myLink);
    if (myErr != noErr) {
        return(nil);
    } else {
        return(mySndChannel);
    }
}
```

As you can see, the `MyCreateLocalizedChannel` function defined in Listing 1-1 calls `SndNewChannel` to create a new sound channel. If successful, it fills in the fields of a sound component link structure to specify the type of component to be installed. The localization component is of type `kSoundEffectsType` and subtype `kSSpLocalizationSubType`. Finally, `MyCreateLocalizedChannel` calls the `SndSetInfo` function to install the specified component before the Apple Mixer in the sound channel's component chain. If `SndSetInfo` completes successfully, `MyCreateLocalizedChannel` returns the sound channel pointer to the caller.

Controlling Filters

SoundSprocket filters are controlled by the `SndSetInfo` function. The selectors you use to control them are `siSSpFilterVersion`, `siSSpCPULoadLimit`, `siSSpSetup` and `siSSpLocalization`.

Accessing Version Information

To retrieve the version of the sound localization filter that is currently installed, call `SndGetInfo`, as in this example:

```
SSpFilterVersionData filterVersion;
err = SndGetInfo(sndChannel, siSSpFilterVersion, &filterVersion);
if (err != noErr) …
```

You can use the `filterVersion` parameter to insure that the installed version of the sound filter is compatible with your game.

Changing Localization Parameters

To change the characteristics of the 3D filtering on a given sound channel, call `SndSetInfo` with the `siSSpLocalization` selector and the address of an `SSpLocalizationData` data structure. The changes will take effect almost immediately.

```
SSpLocalizationData localization;
err = SndGetInfo(sndChannel, siSSpLocalization, &localization);
if (err != noErr) …

localization.currentLocation.distance = 5;
err = SndSetInfo(sndChannel, siSSpLocalization, &localization);
if (err != noErr) …
```

Note
In future versions of SoundSprocket, the values of some fields may affect all localized sound channels. These fields may include `medium`, `humidity`, `roomSize`, `roomReflectivity` and `reverbAttenuation`. This would occur if the implementation used a post-mix filter to implement these features. In this case, the most recent values sent to any sound channel will affect all sound channels. As a precaution, the application should not vary these values between sound channels. ◆

Calling `SndGetInfo` for `siSSpLocalization` returns the filter parameters that are currently in effect. For most fields, this is what was most recently set for this channel. For fields that are shared between sound channels, the most recent value that was set for any channel is returned.

Determining CPU Load Steps

To determine the number of CPU load steps that the localization algorithms support, you can call `SndGetInfo` this way:

```
UInt32 cpuLoadLimit;
err = SndGetInfo(sndChannel, siSSpCPULoadLimit, &cpuLoadLimit);
if (err != noErr) …
```

This is the maximum value that is defined for the `cpuLoad` field of `SSpLocalizationData`.

Changing Speaker Configurations

To change the speaker configuration, call `SndSetInfo` with the selector `siSSpSetup` and the `SSpSetupData` data structure on a sound channel that has the 3D sound filters installed.

```
SSpSetupData setup;
setup.speakerKind = kSSpSpeakerKind_Headphones;
setup.speakerAngle = 0;
setup.reserved0 = 0;
setup.reserved1 = 0;
err = SndSetInfo(sndChannel, siSSpSetup, &setup);
if (err != noErr) …
```

Calling `SndGetInfo` for `siSSpSetup` returns the current speaker configuration. The speaker configuration of all sound channels is affected by sending this message to any channel that has the component installed. The information is retained even when the system is restarted.

Creating Listeners and Sound Sources

The virtual audio environment managed by SoundSprocket includes a single listener and one or more sound sources. You create a listener by calling the `SSpListener_New` function. Listing 1-2 shows how to create a listener and set the listener unit of measurement to feet.

Listing 1-2 Creating a listener

```
static SSpListenerReference     gListener = nil;

void My3DSoundInit (void)
{
    OSStatus        myStatus;

    myStatus = SSpListener_New(&gListener);
    if (myStatus == noErr) {
        SSpListener_SetMetersPerUnit(gListener, 0.3048);
    }
}
```

To create a sound source, you can call the `SSpSource_New` function, as shown in Listing 1-3.

Listing 1-3 Creating a sound source

```
SSpSourceReference MyCreateSource (void)
{
    SSpSourceReferencemySource = nil;
    OSStatus        myStatus;
```

```
    myStatus = SSpSource_New(&mySource);
    return(mySource);
}
```

It's your responsibility to keep track of which sound sources are associated with which sound channels. A game that allows two sound sources might keep track of them in a pair of arrays accessed using global variables, as illustrated in Listing 1-4.

Listing 1-4 Setting up a two-source audio environment

```
SSpSourceReference      gSources[2];
SndChannelPtr           gChannels[2];

void MyInitSources (void)
{
    gSources[0] = MyCreateSource();                 /* see Listing 1-3 */
    gSources[1] = MyCreateSource();

    gChannels[0] = MyCreateLocalizedChannel();      /* see Listing 1-1 */
    gChannels[1] = MyCreateLocalizedChannel();
}
```

Updating the Virtual Audio Environment

Once you've created a listener and one or more localized sound sources, you're ready to play localized sounds. To do this, you need to specify the characteristics of each sound source (position, orientation, velocity, and so forth), call `SndSetInfo` to pass that information to the localization component, and start a sound playing in the associated sound channel. Thereafter, you can change the 3D characteristics as desired and call `SndSetInfo` to achieve a sense of motion. Listing 1-5 illustrates a simple way to move a sound source from right to left.

Listing 1-5 Panning a sound source from right to left

```
OSStatus MyPanSoundFromRightToLeft (Handle mySndResource)
{
    OSStatus        myStatus;
    TQ3Point3D      myPoint;
    SSpLocalizationData     myLocalization;
    float           myZPos;             /* z coord. of sound source */
    long            myTicks;            /* for Delay */

    /* start playing a sound */
    myStatus = SndPlay(gChannels[0], mySndResource, true);
    if (myStatus != noErr)
        return(myStatus);

    /* slowly move source from listener's right to left */
    for (myZPos = 1.0; myZPos >= -1.0; myZPos -= 0.1) {

        /* set position of source */
        Q3Point3D_Set(myPoint, 1, 0, myZPos);
        SSpSource_SetPosition(gSources[0], myPoint);

        / *retrieve updated info - send it to localization component */
        SSpSource_CalcLocalization(gSources[0], gListener,
            &myLocalization);
        SndSetInfo(gChannels[0], siSSpLocalization, &mymyLocalization);
        Delay(2, &myTicks);
    }
}
```

This sample code produces a sound that moves from left to right, on a line perpendicular to the front of the user. This has the effect of modulating the volume as well as a Doppler effect, increasing the volume as the sound approaches the center and decreasing the volume as the sound moves away to the right.

The `MyPanSoundFromRightToLeft` function defined in Listing 1-5 takes as a parameter a handle to a sound resource. It starts playing the resource using the `SndPlay` function. Then `MyPanSoundFromRightToLeft` sets the initial position of the sound source and updates the virtual audio environment by calling `SSpSource_CalcLocalization` and `SndSetInfo`. By default, the listener is at the

origin looking straight down the positive *x* axis. Accordingly, placing the sound source at the point (1, 0, 1) makes it appear ahead and to the right of the listener. Finally, `MyPanSoundFromRightToLeft` gradually moves the sound source along the line *x* = 1 until it reaches the point (1, 0, –1), updating the audio environment after each position change.

Using the Low-Level Interfaces

As you've seen in the preceding two sections, you use the high-level functions provided by SoundSprocket to create listeners and sound sources and to manipulate the data associated with those objects. However, you can (if you prefer) maintain the listener and source data yourself and send commands directly to the localization component using the Sound Manager's `SndSetInfo` function. To specify the data, you need to use the `SSpLocalizationData` data structure (page 1-33).

Note
The distinction between using SoundSprocket's high-level interfaces and its low-level interfaces is similar to the distinction between using QuickDraw 3D's retained rendering mode and its immediate rendering mode. ◆

Listing 1-6 shows how to pan a sound from right to left using the low-level SoundSprocket interfaces.

Listing 1-6 Panning a sound source from right to left

```
OSStatus MyLowLevelPanSoundFromRightToLeft (Handle mySndResource)
{
    OSStatus        myStatus;
    SSpLocalizationData
                    myLocalization;
    float           myAngle;
    long            myTicks;            /* for Delay */

    /* start playing a sound */
    myStatus = SndPlay(gChannels[0], mySndResource, true);
    if (myStatus != noErr)
        return(myStatus);
```

```
    myLocalization.medium = kSSpMedium_Air;
    myLocalization.humidity = 0;
    myLocalization.roomSize = 0;
    myLocalization.roomReflectivity = -1;
    myLocalization.reverbAttenuation = 0;
    myLocalization.sourceMode = kSSpSourceMode_Localized;
    myLocalization.referenceDistance = 1;
    myLocalization.coneAngleCos = 0;
    myLocalization.coneAttenuation = 0;
    myLocalization.currentLocation.elevation = 0;
    myLocalization.currentLocation.azimuth = 0;
    myLocalization.currentLocation.distance = 1;
    myLocalization.currentLocation.projectionAngle = 1;
    myLocalization.currentLocation.sourceVelocity = 0;
    myLocalization.currentLocation.listenerVelocity = 0;
    myLocalization.reserved0 = 0;
    myLocalization.reserved1 = 0;
    myLocalization.reserved2 = 0;
    myLocalization.reserved3 = 0;
    myLocalization.virtualSourceCount = 0;

    /* slowly move source from listener's right to left */
    for (myAngle = 3.14; myAngle >= -3.14; myAngle -= 0.1) {
        myLocalization.currentLocation.azimuth = myAngle;
        SndSetInfo(gChannels[0], siSSpLocalization, &myLocalization);
        Delay(2, &myTicks);
    }
}
```

Unlike the high-level code sample in Listing 1-5, this code sample causes the sound to move from right to left in a semi-circle about the listener. Because the distance is constant, there is no amplitude change or Doppler effect in this sample.

SoundSprocket Reference

This section describes the constants, data structures, and functions provided by SoundSprocket.

Constants

This section describes the constants provided by SoundSprocket.

Component Types and Subtypes

The `componentType` field of a component description record (of type `ComponentDescription`) specifies the type of a component. For a localization component, you should use this value:

```
enum {
    kSoundEffectsType                           = 'snfx'
};
```

Constant descriptions

`kSoundEffectsType` The component provides special sound effects.

The `componentSubType` field of a component description record specifies the subtype of a component. For a localization component, you should use one of these values:

```
enum {
    kReverbSubType                              = 'revb',
    kSSpLocalizationSubType                     = 'snd3'
};
```

Constant descriptions

`kReverbSubType` The component provides reverberation effects.

`kSSpLocalizationSubType` The component provides 3D filtering effects.

Sound Channel Information Selectors

You can manipulate SoundSprocket's spatial filters using the Sound Manager's `SndGetInfo` and `SndSetInfo` functions. The second parameter of those functions is a **sound channel information selector** which determines the type of information you want to get or set. SoundSprocket defines these constants for sound channel information selectors:

```
enum {
    siPreMixerSoundComponent                    = 'prmx',
    siSSpCPULoadLimit                           = '3dll',
    siSSpSetup                                  = '3dst',
    siSSpLocalization                           = '3dif',
    siSSpFilterVersion                          = '3dfv'
};
```

Constant descriptions

`siPreMixerSoundComponent`
: For `SndSetInfo`, install the sound component specified by the third parameter into the sound channel specified by the first parameter, before the Apple Mixer. The third parameter is a pointer to a sound component link structure; see (page 1-30) for details. For `SndGetInfo`, this message has no effect.

`siSSpCPULoadLimit`
: For `SndGetInfo`, return (in the 4-byte area of memory pointed to by the third parameter of `SndGetInfo`) the number of CPU load levels supported by the 3D sound component and its associated filters. The value returned is the maximum value that you can specify for the `cpuLoad` field of the `SSpLocalizationData` data structure; see (page 1-33) for complete details. For `SndSetInfo`, this message has no effect.

`siSSpSetup`
: For `SndSetInfo`, change the current speaker configuration to that specified by the `SSpSetupData` data structure pointed to by the third parameter of `SndGetInfo`. (See (page 1-31) for a description of the `SSpSetupData` data structure.) The 3D sound component and filters must already have been installed on the specified sound channel. For `SndGetInfo`, return the current speaker configuration in the `SSpSetupData` data structure pointed to by the third parameter. Changing the speaker configuration of a channel that has the 3D sound component installed also changes the speaker configuration of all installed 3D sound components. In addition, the speaker configuration is stored across subsequent reboots of the computer.

`siSSpLocalization`
: For `SndSetInfo`, change the characteristics of the 3D filtering for a specific sound channel to those specified by the `SSpLocalizationData` data structure pointed to by the

third parameter of `SndSetInfo`. (See (page 1-33) for a description of the `SSpLocalizationData` data structure.) The 3D sound component and filters must already have been installed on the specified sound channel. Some characteristics might affect all sound channels, depending on the implementation of the filters. In particular, the values specified in the `medium`, `humidity`, `roomSize`, `roomReflectivity`, and `reverbAttenuation` fields might be applied to all sound channels, not only the one specified in the call to `SndSetInfo`. To be safe, you should not assign different values of these fields to different sound channels. For `SndGetInfo`, return the current 3D filter characteristics in the `SSpLocalizationData` data structure pointed to by the third parameter. For most fields of that structure, the values returned are simply those most recently set for the specified sound channel. For fields that are shared between channels, however, the values returned are those most recently set for any sound channel.

`siSSpFilterVersion` To identify the sound localization filter currently installed, call `SndGetInfo` with `siSSpFilterVersion` as the message, along with the address of a `SSpFilterVersionData` record. You should use `SndGetInfo` after you install a sound filter to determine if it was installed successfully. You can determine this by testing if the `manufacturer` field (or other field) was changed. It may also be appropriate to test version numbers, if available.

Speaker Types

You specify the type of speakers being used by passing one of the following constants in the `speakerKind` field of a 3D sound setup structure:

```
enum {
    kSSpSpeakerKind_Stereo                          = 0,
    kSSpSpeakerKind_Mono                            = 1,
    kSSpSpeakerKind_Headphones                      = 2
};
```

Constant descriptions

`kSSpSpeakerKind_Stereo`
The sound output device is a set of stereo speakers.

`kSSpSpeakerKind_Mono`
The sound output device is a monophonic speaker.

`kSSpSpeakerKind_Headphones`
The sound output device is a set of headphones.

Sound Media

A sound medium is a substance through which sound can travel. You can use these constants to specify a sound medium:

```
enum {
    kSSpMedium_Air                              = 0,
    kSSpMedium_Water                            = 1
};
```

Constant descriptions

`kSSpMedium_Air` The sound is traveling through air at standard temperature and pressure (STP).

`kSSpMedium_Water` The sound is traveling through water.

Source Modes

A listener's source mode determines the type of filtering that is applied to a source sound. You can use these constants to specify a source mode:

```
enum {
    kSSpSourceMode_Unfiltered                   = 0,
    kSSpSourceMode_Localized                    = 1,
    kSSpSourceMode_Ambient                      = 2,
    kSSpSourceMode_Binaural                     = 3
};
```

Constant descriptions

`kSSpSourceMode_Unfiltered`
: Unfiltered sounds are not processed by the sound localization filter in any way.

`kSSpSourceMode_Localized`
: The sound is localized (that is, it appears to be emanating from a specific location in space).

`kSSpSourceMode_Ambient`
: The sound is ambient (that is, it appears to come from all directions).

`kSSpSourceMode_Binaural`
: The sound is binaural (that is, it was recorded with a localized effect). It is played exactly as recorded, except that a crosstalk canceller is used when the `speakerKind` field of a 3D sound setup structure has the value `kSSpSpeakerKind_Stereo`.

Data Structures

This section describes the data structures provided by SoundSprocket. You use these structures to define the configuration of the listener's speakers, to specify the location and rotation of sound sources, and to get and set characteristics of the 3D filtering for a specific sound channel.

Filter Version Structure

The `SSpFilterVersionData` data structure describes the version of the currently installed sound filter. It is used with the `siSSpFilterVersion` selector with the `SndGetInfo` and `SndSetInfo` functions.

```
typedef struct SSpFilterVersionData {
    OSType                              manufacturer;
    NumVersion                          version;
    UInt32                              reserved;
} SSpFilterVersionData;
```

Field descriptions

`manufacturer`	The manufacturer's code for the sound filter component. You may have specified this in `link.description.componentManufacturer` at installation time.
`version`	The version number of the sound filter component. Version number sequences are determined by the manufacturer of the component. The `NumVersion` type is described in *Inside Macintosh: Sound*.
`reserved`	Reserved for use by Apple Computer, Inc. You should set this field to 0.

Sound Component Link Structure

The third parameter passed to `SndSetInfo` for the `siPreMixerSoundComponent` information selector is a pointer to a **sound component link structure.** A sound component link structure is defined by the `SoundComponentLink` data type.

```
struct SoundComponentLink {
    ComponentDescription                description;
    SoundSource                         mixerID;
    SoundSource *                       linkID;
};

typedef struct SoundComponentLink SoundComponentLink;
typedef SoundComponentLink *SoundComponentLinkPtr;
```

Field descriptions

`description`	A component description record.
`mixerID`	A mixer ID.
`linkID`	A link ID. Set this value to `NULL`.

Note
The `ComponentDescription` structure is described in *Inside Macintosh: More Macintosh Toolbox*. ◆

3D Sound Setup Structure

You specify the configuration of the listener's speakers using a 3D sound setup structure. A 3D sound setup structure is defined by the `SSpSetupData` data type.

```
typedef struct SSpSetupData {
    UInt32                              speakerKind;
    float                               speakerAngle;
    UInt32                              reserved0;
    UInt32                              reserved1;
} SSpSetupData;
```

Field descriptions

`speakerKind` The type of speakers being used. See "Speaker Types" (page 1-27) for a description of the values that can occur in this field.

`speakerAngle` The angle, in radians, formed by the speakers and the listener. Values for this field can range from 0.0 to π. This field is used only if the value of the `speakerKind` field is `kSSpSpeakerKind_Stereo`.

`reserved0` Reserved for use by Apple Computer, Inc. You should set this field to 0.

`reserved1` Reserved for use by Apple Computer, Inc. You should set this field to 0.

Source Location Structure

You specify the position of a sound source relative to the listener using a source location structure. (For example, the `currentLocation` field of a 3D sound information structure is a source location structure.) A source location structure is defined by the `SSpLocationData` data type.

Note
You use a source location structure to specify the position of both real and virtual sound sources. See Figure 1-5 (page 1-13) for an illustration showing how to interpret the `elevation` and `azimuth` fields. ◆

```
typedef struct SSpLocationData {
    float                               elevation;
    float                               azimuth;
    float                               distance;
    float                               projectionAngle;
    float                               sourceVelocity;
    float                               listenerVelocity
} SSpLocationData;
```

Field descriptions

`elevation` The angle, in radians, from the orientation vector of the listener to the meridian of longitude on which the sound source lies. The value of this field must lie between –π and π, inclusive, where 0 indicates the meridian of longitude that intersects the listener's orientation vector. Positive values are up, and negative values are down.

`azimuth` The angle, in radians, from the orientation vector of the listener to the parallel of latitude on which the sound source lies. The value of this field must lie between –π/2 and π/2, inclusive, where 0 indicates the parallel of latitude that intersects the listener's orientation vector. Positive values are right, and negative values are left.

`distance` The distance, in meters, between the listener and the sound source. The value in this field should be greater than 0.0; if the value reaches 0.0, the source mode (specified by the `sourceMode` field of the `SSpLocalizationData` structure) should be set to `kSSpSourceMode_Ambient`.

`projectionAngle` The cosine of the angle between the sound source's attenuation cone axis and the vector from the source to the listener. When the attenuation cone points directly at the listener, the value in this field should be 1.0 (because *cos*(0) = 1.0).

`sourceVelocity` The velocity, in meters per second, of the source along the vector from the listener to the source. The value in this field is positive when the source is moving toward the listener.

`listenerVelocity` The velocity, in meters per second, of the listener along the vector from the listener to the source. The value in this

field is positive when the listener is moving toward the source.

Virtual Source Structure

You use a virtual source structure to describe a virtual sound source (that is, a reflection of a real sound source). A virtual source structure is defined by the `SSpVirtualSourceData` data type.

```
typedef struct SSpVirtualSourceData {
    float                               attenuation;
    SSpLocationData                     location;
} SSpVirtualSourceData;
```

Field descriptions

`attenuation` The amount of attenuation, in decibels (dB), that occurs each time this reflection bounces off a reverberant wall. The value of this field should be less than or equal to 0.0.

`location` The current location of the virtual source relative to the listener. See (page 1-31) for a description of the `SSpLocationData` data structure.

3D Sound Information Structure

You use a 3D sound information structure to get and set the characteristics of the filtering to be applied to a specific sound channel. A 3D sound information structure is defined by the `SSpLocalizationData` data type.

```
typedef struct SSpLocalizationData {
    UInt32                              cpuLoad;
    UInt32                              medium;
    float                               humidity;
    float                               roomSize;
    float                               roomReflectivity;
    float                               reverbAttenuation;
    UInt32                              sourceMode;
    float                               referenceDistance;
    float                               coneAngleCos;
    float                               coneAttenuation;
    SSpLocationData                     currentLocation;
```

```
    UInt32                          reserved0;
    UInt32                          reserved1;
    UInt32                          reserved2;
    UInt32                          reserved3;
    UInt32                          virtualSourceCount;
    SSpVirtualSourceData            virtualSource[4];
} SSpLocalizationData;
```

Field descriptions

`cpuLoad` The current CPU load and sound quality setting for the sound localization filters. The value 0 indicates the greatest CPU load (and hence the best sound quality). The value returned by `SndGetInfo` for the `siSSpCPULoadLimit` selector indicates the least CPU load (and hence the worst sound quality).

`medium` The medium through which sound is traveling. See "Sound Media" (page 1-28) for a description of constants that specify the available sound media. In version 1.0 of SoundSprocket, this field is ignored and the medium is always assumed to be dry (zero humidity) air.

`humidity` The relative humidity of the atmosphere. The value of this field should be a floating-point value between 0.0 (indicating dry air) and 100.0 (indicating dense fog). This field is used only if the value of the `medium` field is `kSSpMedium_Air`. In version 1.0 of SoundSprocket, this field is ignored.

`roomSize` The distance, in meters, between the reverberant walls. For no reverberation, set this field to 0.0.

`roomReflectivity` The amount of attenuation, in decibels (dB), that occurs each time a sound bounces off a reverberant wall. The value of this field should be less than 0.0. This field is ignored if the value in the `roomSize` field is 0.

`reverbAttenuation` The amount of reverberant signal, in decibels (dB), to mix into the output sound. This field is ignored if the value in the `roomSize` field is 0.

`sourceMode` The type of filtering to be applied to the source sound. See "Source Modes" (page 1-28) for a description of constants that specify the available source filtering modes.

`referenceDistance`	The distance, in meters, from the listener to the point at which the sound was recorded. This field is used when the `sourceMode` field has the value `kSSpSourceMode_Localized` or `kSSpSourceMode_Ambient`. The value of this field must be greater than 0.0.
`coneAngleCos`	The cosine of half of the angle at the apex of the angular attenuation cone. This field is used when the `sourceMode` field has the value `kSSpSourceMode_Localized` or `kSSpSourceMode_Ambient`.
`coneAttenuation`	The amount of attenuation, in decibels (dB), that occurs outside the angular attenuation cone. This field is used when the `sourceMode` field has the value `kSSpSourceMode_Localized` or `kSSpSourceMode_Ambient`.
`currentLocation`	The current location of the sound source relative to the listener. See (page 1-31) for a description of the `SSpLocationData` data structure.
`reserved0`	Reserved for use by Apple Computer, Inc. You should set this field to 0.
`reserved1`	Reserved for use by Apple Computer, Inc. You should set this field to 0.
`reserved2`	Reserved for use by Apple Computer, Inc. You should set this field to 0.
`reserved3`	Reserved for use by Apple Computer, Inc. You should set this field to 0.
`virtualSourceCount`	The number of virtual sound sources contained in the array specified by the `SSpVirtualSourceData` parameter. This field is used when the `sourceMode` field has the value `kSSpSourceMode_Localized` or `kSSpSourceMode_Ambient`. In SoundSprocket version 1.0, this field is ignored.
`virtualSource`	An array containing up to four virtual source structures describing the virtual sound sources (that is, the reflections) associated with the sound channel. See (page 1-33) for a description of the virtual source structure. This field is used only if the `sourceMode` field has the value `kSSpSourceMode_Localized`. In SoundSprocket version 1.0, this field is ignored.

SoundSprocket Functions

This section describes the functions provided by SoundSprocket. You can use these functions to create and manipulate listeners and sound sources.

IMPORTANT

You need to use the functions described in this section only if you are using the high-level sound filtering interfaces provided by SoundSprocket. For low-level access to listeners and sound sources, use the Sound Manager's `SndGetInfo` and `SndSetInfo` functions with the sound channel commands described in "Sound Channel Information Selectors" (page 1-25). ▲

Controlling Sound Output Devices

SoundSprocket provides functions that you can use to control sound output devices such as speakers and headphones.

SSpConfigureSetup

You can use the `SSpConfigureSetup` function to display a modal dialog box that allows the user to select which sound output device to use and to set the angle of stereo speakers, if they are selected.

```
OSStatus SSpConfigureSetup  (SSpEventProcPtr inEventProcPtr);
```

`inEventProcPtr`
A pointer to your event handling procedure

function result A result code.

DESCRIPTION

When you call `SSpConfigureSetup` to display the modal dialog box, you should use `inEventProcPtr` to point to your own function to handle update or other events as necessary. Your function should be declared as `SSpEventProcPtr` to

return the event as `false` if you want `SSpConfigureSetup` to handle the event and `true` if you have handled the event with your own function.

```
typedef Boolean (*SSpEventProcPtr)          (EventRecord* InEvent);
```

Creating and Managing Listeners

SoundSprocket provides functions that you can use to create and manage listeners. You refer to a listener using the `SSpListenerReference` data type, defined as follows:

```
typedef struct SSpListenerPrivate           *SSpListenerReference;
```

The `SSpListenerReference` data type is opaque: all manipulations on listeners must be accomplished using the functions described in this section.

SSpListener_New

You can use the `SSpListener_New` function to create a new listener.

```
OSStatus SSpListener_New (SSpListenerReference *outListenerReference);
```

`outListenerReference`
On exit, a reference to a new listener.

function result A result code.

DESCRIPTION

The `SSpListener_New` function returns, in the `outListenerReference` parameter, a reference to a new listener.

SSpListener_Dispose

You can use the `SSpListener_Dispose` function to dispose of a listener.

```
OSStatus SSpListener_Dispose (SSpListenerReference inListenerReference);
```

`inListenerReference`
A listener reference.

function result A result code.

DESCRIPTION

The `SSpListener_Dispose` function disposes of the listener specified by the `inListenerReference` parameter.

SSpListener_GetTransform

You can use the `SSpListener_GetTransform` function to get the transform matrix of a listener.

```
OSStatus SSpListener_GetTransform (
                    SSpListenerReference inListenerReference,
                    TQ3Matrix4x4 *outTransform);
```

`inListenerReference`
A listener reference.

`outTransform` On exit, a 4-by-4 matrix that specifies the current transformation to be applied to the listener.

function result A result code.

DESCRIPTION

The `SSpListener_GetTransform` function returns, in the `outTransform` parameter, a 4-by-4 matrix that specifies the current transformation to be applied to the position, velocity, orientation, and up vector of the listener specified by the `inListenerReference` parameter.

SEE ALSO

See *3D Graphics Programming With QuickDraw 3D* for a description of the `TQ3Matrix4x4` data type.

SSpListener_SetTransform

You can use the `SSpListener_SetTransform` function to set the transform matrix of a listener.

```
OSStatus SSpListener_SetTransform (
                    SSpListenerReference inListenerReference,
                    const TQ3Matrix4x4 *inTransform);
```

`inListenerReference`
A listener reference.

`inTransform` A 4-by-4 matrix that specifies the transformation to be applied to the listener.

function result A result code.

DESCRIPTION

The `SSpListener_SetTransform` function sets the transform matrix of the listener specified by the `inListenerReference` parameter to the matrix specified by the `inTransform` parameter. The default transform matrix for a listener is the identity matrix.

SSpListener_GetPosition

You can use the `SSpListener_GetPosition` function to get the position of a listener.

```
OSStatus SSpListener_GetPosition (
                    SSpListenerReference inListenerReference,
                    TQ3Point3D *outPosition);
```

`inListenerReference`
A listener reference.

`outPosition` On exit, the most recent untransformed position of the listener.

function result A result code.

DESCRIPTION

The `SSpListener_GetPosition` function returns, in the `outPosition` parameter, the most recent untransformed position of the listener specified by the `inListenerReference` parameter.

SSpListener_SetPosition

You can use the `SSpListener_SetPosition` function to set the position of a listener.

```
OSStatus SSpListener_SetPosition (
                    SSpListenerReference inListenerReference,
                    const TQ3Point3D *inPosition);
```

`inListenerReference`
A listener reference.

`inPosition` The desired untransformed position of the listener.

function result A result code.

DESCRIPTION

The `SSpListener_SetPosition` function sets the position of the listener specified by the `inListenerReference` parameter to the point specified by the `inPosition` parameter. The `inPosition` parameter specifies the untransformed position of the listener (that is, the position of the listener before the listener's transform is applied). The actual position of the listener is calculated at the time you call the `SSpSource_CalcLocalization` function. The default position of a listener is the origin—the point (0, 0, 0).

SSpListener_GetOrientation

You can use the `SSpListener_GetOrientation` function to get the orientation of a listener.

```
OSStatus SSpListener_GetOrientation (
                    SSpListenerReference inListenerReference,
                    TQ3Vector3D *outOrientation);
```

`inListenerReference`
A listener reference.

`outOrientation`
On exit, the untransformed orientation of the listener.

function result A result code.

DESCRIPTION

The `SSpListener_GetOrientation` function returns, in the `outOrientation` parameter, the untransformed orientation of the listener specified by the `inListenerReference` parameter.

SSpListener_SetOrientation

You can use the `SSpListener_SetOrientation` function to set the orientation of a listener.

```
OSStatus SSpListener_SetOrientation (
                    SSpListenerReference inListenerReference,
                    const TQ3Vector3D *inOrientation);
```

`inListenerReference`
A listener reference.

`inOrientation` The desired untransformed orientation of the listener. This orientation should be a unit vector.

function result A result code.

DESCRIPTION

The `SSpListener_SetOrientation` function sets the orientation of the listener specified by the `inListenerReference` parameter to the orientation specified by the `inOrientation` parameter. The `inOrientation` parameter specifies the untransformed orientation of the listener (that is, the orientation of the listener before the listener's transform is applied). The actual orientation of the listener is calculated at the time you call the `SSpSource_CalcLocalization` function. The default orientation of a listener is the unit *x* vector (1, 0, 0).

SPECIAL CONSIDERATIONS

When you call `SSpSource_CalcLocalization`, the listener's orientation and up vectors should not be parallel.

SSpListener_GetUpVector

You can use the `SSpListener_GetUpVector` function to get the up vector of a listener.

```
OSStatus SSpListener_GetUpVector (
                    SSpListenerReference inListenerReference,
                    TQ3Vector3D *outUpVector);
```

`inListenerReference`
A listener reference.

`outUpVector` On exit, the untransformed up vector of the listener.

function result A result code.

DESCRIPTION

The `SSpListener_GetUpVector` function returns, in the `outUpVector` parameter, the untransformed up vector of the listener specified by the `inListenerReference` parameter.

SSpListener_SetUpVector

You can use the `SSpListener_SetUpVector` function to set the up vector of a listener.

```
OSStatus SSpListener_SetUpVector (
                    SSpListenerReference inListenerReference,
                    const TQ3Vector3D *inUpVector);
```

`inListenerReference`
A listener reference.

`inUpVector` The desired untransformed up vector of the listener. This up vector should be a unit vector.

function result A result code.

DESCRIPTION

The `SSpListener_SetUpVector` function sets the up vector of the listener specified by the `inListenerReference` parameter to the vector specified by the `inUpVector` parameter. The `inUpVector` parameter specifies the untransformed up vector of the listener (that is, the up vector of the listener before the listener's transform is applied). The actual up vector of the listener is calculated at the time you call the `SSpSource_CalcLocalization` function. The default up vector of a listener is the unit *y* vector (0, 1, 0).

Note
When you call `SSpSource_CalcLocalization`, the listener's orientation and up vectors should not be parallel. ◆

SSpListener_GetCameraPlacement

You can use the `SSpListener_GetCameraPlacement` function to get the position, orientation, and up vector of a listener.

```
OSStatus SSpListener_GetCameraPlacement (
                    SSpListenerReference inListenerReference,
                    TQ3CameraPlacement *outCameraPlacement);
```

`inListenerReference`
A listener reference.

`outCameraPlacement`
On entry, a pointer to a camera placement structure. On exit, that structure is updated to specify the position, orientation, and up vector of the listener.

function result A result code.

DESCRIPTION

The `SSpListener_GetCameraPlacement` function updates the camera placement structure pointed to by the `outCameraPlacement` parameter to indicate the position, orientation, and up vector of the listener specified by the `inListenerReference` parameter. (The point of interest of the camera placement is the listener's position plus its orientation vector.) This function is provided as a convenient method for getting that information in a single function call.

Note
The `TQ3CameraPlacement` data type is defined by QuickDraw 3D. See *3D Graphics Programming With QuickDraw 3D* for details. ◆

SEE ALSO

Instead of calling `SSpListener_GetCameraPlacement`, you could call `SSpListener_GetPosition`, `SSpListener_GetOrientation`, and `SSpListener_GetUpVector`.

SSpListener_SetCameraPlacement

You can use the `SSpListener_SetCameraPlacement` function to set the position, orientation, and up vector of a listener.

```
OSStatus SSpListener_SetCameraPlacement (
                    SSpListenerReference inListenerReference,
                    const TQ3CameraPlacement *inCameraPlacement);
```

`inListenerReference`
A listener reference.

`inCameraPlacement`
A pointer to a camera placement structure that specifies the desired position, orientation, and up vector of the listener.

function result A result code.

DESCRIPTION

The `SSpListener_SetCameraPlacement` function sets the position, orientation, and up vector of the listener specified by the `inListenerReference` parameter to the values contained in the camera placement structure pointed to by the `inCameraPlacement` parameter. This function is provided as a convenient method for setting that information in a single function call.

Note
The `TQ3CameraPlacement` data type is defined by QuickDraw 3D. See *3D Graphics Programming With QuickDraw 3D* for details. ◆

SEE ALSO

Instead of calling `SSpListener_SetCameraPlacement`, you could call `SSpListener_SetPosition`, `SSpListener_SetOrientation`, and `SSpListener_SetUpVector`.

SSpListener_GetVelocity

You can use the `SSpListener_GetVelocity` function to get the velocity of a listener.

```
OSStatus SSpListener_GetVelocity (
                    SSpListenerReference inListenerReference,
                    TQ3Vector3D *outVelocity);
```

`inListenerReference`
A listener reference.

`outVelocity` On exit, the most recent untransformed velocity vector of the listener.

function result A result code.

DESCRIPTION

The `SSpListener_GetVelocity` function returns, in the `outVelocity` parameter, the most recent untransformed velocity vector of the listener specified by the `inListenerReference` parameter. The velocity is the velocity set by the most recent call to `SSpListener_SetVelocity`.

SSpListener_SetVelocity

You can use the `SSpListener_SetVelocity` function to set the velocity of a listener.

```
OSStatus SSpListener_SetVelocity (
                    SSpListenerReference inListenerReference,
                    const TQ3Vector3D *inVelocity);
```

`inListenerReference`
A listener reference.

`inVelocity` A vector indicating the desired velocity of the listener.

function result A result code.

DESCRIPTION

The `SSpListener_SetVelocity` function sets the untransformed velocity vector of the listener specified by the `inListenerReference` parameter to the vector specified by the `inVelocity` parameter. The `inVelocity` parameter specifies the untransformed velocity of the listener (that is, the velocity of the listener before the listener's transform is applied). The actual velocity of the listener is calculated at the time you call the `SSpSource_CalcLocalization` function. If you call `SSpSource_CalcLocalization` and then change the location of the listener (either by calling `SSpListener_SetPosition` or `SSpListener_SetTransform`) without calling `SSpListener_SetVelocity`, SoundSprocket will compute a velocity vector for you automatically.

A velocity is specified in **listener units** per second, where a listener unit is either a meter or the unit specified in a call to the `SSpListener_SetMetersPerUnit` function.

SSpListener_GetActualVelocity

You can use the `SSpListener_GetActualVelocity` function to get the world-space velocity of a listener.

```
OSStatus SSpListener_GetActualVelocity (
                    SSpListenerReference inListenerReference,
                    TQ3Vector3D *outVelocity);
```

`inListenerReference`
A listener reference.

`outVelocity` On exit, the world-space velocity vector of the listener.

function result A result code.

DESCRIPTION

The `SSpListener_GetActualVelocity` function returns, in the `outVelocity` parameter, the world-space (that is, the transformed) velocity used in the most recent call to `SSpSource_CalcLocalization` for the listener specified by the `inListenerReference` parameter.

SSpListener_GetMedium

You can use the `SSpListener_GetMedium` function to get the sound medium and relative humidity of a listener.

```
OSStatus SSpListener_GetMedium (
                    SSpListenerReference inListenerReference,
                    UInt32 *outMedium,
                    float *outHumidity);
```

`inListenerReference`
A listener reference.

`outMedium` On exit, the current sound medium of the listener. See "Sound Media" (page 1-28) for a description of the available sound media.

`outHumidity` On exit, the current relative humidity of the listener. Humidity is a floating-point value between 0.0 (indicating dry air) and 100.0 (indicating dense fog).

function result A result code.

DESCRIPTION

The `SSpListener_GetMedium` function returns, in the `outMedium` and `outHumidity` parameters, the current sound medium and relative humidity of the listener specified by the `inListenerReference` parameter.

SSpListener_SetMedium

You can use the `SSpListener_SetMedium` function to set the sound medium and relative humidity of a listener.

```
OSStatus SSpListener_SetMedium (
                    SSpListenerReference inListenerReference,
                    UInt32 inMedium,
                    float inHumidity);
```

`inListenerReference`
A listener reference.

`inMedium` The desired sound medium of the listener. See "Sound Media" (page 1-28) for a description of the available sound media.

`inHumidity` The desired relative humidity of the listener. Humidity is a floating-point value between 0.0 (indicating dry air) and 100.0 (indicating dense fog).

function result A result code.

DESCRIPTION

The `SSpListener_SetMedium` function sets the sound medium and relative humidity of the listener specified by the `inListenerReference` parameter to the values specified in the `inMedium` and `inHumidity` parameters. The `inHumidity` parameter is used only if the value in the `inMedium` parameter is `kSSpMedium_Air`; otherwise, you should set `inHumidity` to 0.0.

SSpListener_GetReverb

You can use the `SSpListener_GetReverb` function to get the current reverberation model of a listener.

```
OSStatus SSpListener_GetReverb (
                    SSpListenerReference inListenerReference,
                    float *outRoomSize,
                    float *outRoomReflectivity,
                    float *outReverbAttenuation);
```

`inListenerReference`
A listener reference.

`outRoomSize` On exit, the current room size of the listener. Room size is the distance, in listener units, between two reverberant walls spaced equidistant from the listener. A room size of 0.0 indicates no reverberation.

`outRoomReflectivity`
On exit, the current room reflectivity attenuation.

`outReverbAttenuation`
On exit, the current reverberation attenuation.

function result A result code.

DESCRIPTION

The `SSpListener_GetReverb` function returns, in the `outRoomSize`, `outRoomReflectivity`, and `outReverbAttenuation` parameters, the current room size, room reflectivity, and reverberation attenuation of the listener specified by the `inListenerReference` parameter. By default, there is no reverberation.

SSpListener_SetReverb

You can use the `SSpListener_SetReverb` function to set the reverberation model of a listener.

```
OSStatus SSpListener_SetReverb (
                    SSpListenerReference inListenerReference,
                    float inRoomSize,
                    float inRoomReflectivity,
                    float inReverbAttenuation);
```

`inListenerReference`
A listener reference.

`inRoomSize` The desired room size of the listener. Room size is the distance, in listener units, between two reverberant walls spaced equidistant from the listener. A room size of 0.0 indicates no reverberation.

`inRoomReflectivity`
The desired room reflectivity attenuation.

`inReverbAttenuation`
The desired reverberation attenuation.

function result A result code.

DESCRIPTION

The `SSpListener_SetReverb` function sets the room size, room reflectivity, and reverberation attenuation of the listener specified by the `inListenerReference` parameter to the values specified in the `inRoomSize`, `inRoomReflectivity`, and `inReverbAttenuation` parameters. The `inRoomReflectivity` and `inReverbAttenuation` parameters are used only when `inRoomSize` is nonzero. If `inRoomSize` is 0, you should set `inRoomReflectivity` and `inReverbAttenuation` to 0 as well.

SSpListener_GetMetersPerUnit

You can use the `SSpListener_GetMetersPerUnit` function to get the number of meters in one listener unit.

```
OSStatus SSpListener_GetMetersPerUnit (
                    SSpListenerReference inListenerReference,
                    float *outMetersPerUnit);
```

`inListenerReference`
A listener reference.

`outMetersPerUnit`
On exit, the number of meters in the listener unit of the specified listener.

function result A result code.

DESCRIPTION

The `SSpListener_GetMetersPerUnit` function returns, in the `outMetersPerUnit` parameter, the number of meters in the listener unit associated with the listener specified by the `inListenerReference` parameter. The default listener unit is one meter—that is, all units passed to SoundSprocket functions are interpreted as meters.

SSpListener_SetMetersPerUnit

You can use the `SSpListener_SetMetersPerUnit` function to set the number of meters in one listener unit.

```
OSStatus SSpListener_SetMetersPerUnit (
                    SSpListenerReference inListenerReference,
                    float inMetersPerUnit);
```

`inListenerReference`
A listener reference.

`inMetersPerUnit`
The number of meters in the listener unit of the specified listener.

function result A result code.

DESCRIPTION

The `SSpListener_SetMetersPerUnit` function sets the number of meters in the listener unit of the listener specified by the `inListenerReference` parameter to the value specified by the `inMetersPerUnit` parameter. For example, to have SoundSprocket interpret all values you pass its high-level functions as feet, you should set the `inMetersPerUnit` parameter to 0.3048.

Creating and Managing 3D Sound Sources

SoundSprocket provides functions that you can use to create and manage sound sources. You refer to a sound source using the `SSpSourceReference` data type, defined as follows:

```
typedef struct SSpSourcePrivate                    *SSpSourceReference;
```

SSpSource_New

You can use the `SSpSource_New` function to create a new sound source.

```
OSStatus SSpSource_New (SSpSourceReference *outSourceReference);
```

`outSourceReference`
On exit, a reference to a new sound source.

function result A result code.

DESCRIPTION

The `SSpSource_New` function returns, in the `outSourceReference` parameter, a reference to a new sound source.

SSpSource_Dispose

You can use the `SSpSource_Dispose` function to dispose of a sound source.

```
OSStatus SSpSource_Dispose (SSpSourceReference inSourceReference);
```

`inSourceReference`
A source reference.

function result A result code.

DESCRIPTION

The `SSpSource_Dispose` function disposes of the sound source specified by the `inSourceReference` parameter.

SSpSource_CalcLocalization

You can use the `SSpSource_CalcLocalization` function to determine the relative positions and velocities of a sound source and its listener.

```
OSStatus SSpSource_CalcLocalization (
                    SSpSourceReference inSourceReference,
                    SSpListenerReference inListenerReference,
                    SSpLocalizationData *outLocalization);
```

`inSourceReference`
A source reference.

`inListenerReference`
A listener reference.

`outLocalization`
On entry, a pointer to a 3D sound information structure. On exit, the structure is filled in with current information about the relative positions and velocities of the sound source and the listener.

function result A result code.

DESCRIPTION

The `SSpSource_CalcLocalization` function returns, in the 3D sound information structure specified by the `outLocalization` parameter, the current relative positions and velocities of the sound source specified by the `inSourceReference` parameter and the listener specified by the `inListenerReference` parameter.

`SSpSource_CalcLocalization` computes the actual velocity vectors of the source and listener independently, based on the most recent calls to `SSpSource_SetVelocity` and `SSpListener_SetVelocity`, and then combines those vectors to obtain a relative velocity. If you called `SSpSource_SetVelocity` for the specified source after the most recent call to `SSpSource_CalcLocalization`, then the new velocity is transformed (using the source transform matrix) to obtain the actual velocity of the source. If you called `SSpSource_SetPosition` or `SSpSource_SetTransform` but not `SSpSource_SetVelocity` after the most recent call to `SSpSource_CalcLocalization`, then the actual velocity is set to the difference in transformed positions divided by the interval since the most recent call to `SSpSource_CalcLocalization`. In you haven't called any of these functions since the most recent call to `SSpSource_CalcLocalization`, then the actual source velocity is left unchanged. The listener's actual velocity is calculated in a similar manner.

If `SSpSource_CalcLocalization` completes successfully, you should then execute the following code to change the sound being played:

```
SndSetInfo(myChannel, siSSpLocalization, &mySSpLocalizationData);
```

SSpSource_GetTransform

You can use the `SSpSource_GetTransform` function to get the transform matrix of a sound source.

```
OSStatus SSpSource_GetTransform (
                    SSpSourceReference inSourceReference,
                    TQ3Matrix4x4 *outTransform);
```

`inSourceReference`
A source reference.

`outTransform` On exit, a 4-by-4 matrix that specifies the current transformation to be applied to the sound source.

function result A result code.

DESCRIPTION

The `SSpSource_GetTransform` function returns, in the `outTransform` parameter, a 4-by-4 matrix that specifies the current transformation to be applied to the position, velocity, orientation, and up vector of the sound source specified by the `inSourceReference` parameter.

SEE ALSO

See *3D Graphics Programming With QuickDraw 3D* for a description of the `TQ3Matrix4x4` data type.

SSpSource_SetTransform

You can use the `SSpSource_SetTransform` function to set the transform matrix of a sound source.

```
OSStatus SSpSource_SetTransform (
                    SSpSourceReference inSourceReference,
                    const TQ3Matrix4x4 *inTransform);
```

`inSourceReference`
A source reference.

`inTransform` A 4-by-4 matrix that specifies the transformation to be applied to the sound source.

function result A result code.

DESCRIPTION

The `SSpSource_SetTransform` function sets the transform matrix of the sound source specified by the `inSourceReference` parameter to the matrix specified by the `inTransform` parameter. The default transform matrix for a sound source is the identity matrix.

SSpSource_GetPosition

You can use the `SSpSource_GetPosition` function to get the position of a sound source.

```
OSStatus SSpSource_GetPosition (
                    SSpSourceReference inSourceReference,
                    TQ3Point3D *outPosition);
```

`inSourceReference`
A source reference.

`outPosition` On exit, the most recent untransformed position of the sound source.

function result A result code.

DESCRIPTION

The `SSpSource_GetPosition` function returns, in the `outPosition` parameter, the most recent untransformed position of the sound source specified by the `inSourceReference` parameter.

SSpSource_SetPosition

You can use the `SSpSource_SetPosition` function to set the position of a sound source.

```
OSStatus SSpSource_SetPosition (
                    SSpSourceReference inSourceReference,
                    const TQ3Point3D *inPosition);
```

`inSourceReference`
A source reference.

`inPosition` The desired untransformed position of the sound source.

function result A result code.

DESCRIPTION

The `SSpSource_SetPosition` function sets the position of the sound source specified by the `inSourceReference` parameter to the point specified by the `inPosition` parameter. The `inPosition` parameter specifies the untransformed position of the sound source (that is, the position of the sound source before the sound source's transform is applied). The actual position of the sound source is calculated at the time you call the `SSpSource_CalcLocalization` function. The default position of a sound source is the origin, the point (0, 0, 0).

SSpSource_GetOrientation

You can use the `SSpSource_GetOrientation` function to get the orientation of a sound source.

```
OSStatus SSpSource_GetOrientation (
                    SSpSourceReference inSourceReference,
                    TQ3Vector3D *outOrientation);
```

`inSourceReference`
A source reference.

`outOrientation`
On exit, the untransformed orientation of the sound source.

function result A result code.

DESCRIPTION

The `SSpSource_GetOrientation` function returns, in the `outOrientation` parameter, the untransformed orientation of the sound source specified by the `inSourceReference` parameter.

SSpSource_SetOrientation

You can use the `SSpSource_SetOrientation` function to set the orientation of a sound source.

```
OSStatus SSpSource_SetOrientation (
                    SSpSourceReference inSourceReference,
                    const TQ3Vector3D *inOrientation);
```

`inSourceReference`
A source reference.

`inOrientation` The desired untransformed orientation of the sound source. This orientation should be a unit vector.

function result A result code.

DESCRIPTION

The `SSpSource_SetOrientation` function sets the orientation of the sound source specified by the `inSourceReference` parameter to the orientation specified by the `inOrientation` parameter. The `inOrientation` parameter specifies the untransformed orientation of the sound source (that is, the orientation of the sound source before the sound source's transform is applied). The actual orientation of the sound source is calculated at the time you call the `SSpSource_CalcLocalization` function. The default orientation of a sound source is the unit *x* vector (1, 0, 0).

SPECIAL CONSIDERATIONS

When you call `SSpSource_CalcLocalization`, the sound source's orientation and up vectors should not be parallel.

SSpSource_GetUpVector

You can use the `SSpSource_GetUpVector` function to get the up vector of a sound source.

```
OSStatus SSpSource_GetUpVector (
                    SSpSourceReference inSourceReference,
                    TQ3Vector3D *outUpVector);
```

`inSourceReference`
A source reference.

`outUpVector` On exit, the untransformed up vector of the sound source.

function result A result code.

DESCRIPTION

The `SSpSource_GetUpVector` function returns, in the `outUpVector` parameter, the untransformed up vector of the sound source specified by the `inSourceReference` parameter.

SSpSource_SetUpVector

You can use the `SSpSource_SetUpVector` function to set the up vector of a sound source.

```
OSStatus SSpSource_SetUpVector (
                    SSpSourceReference inSourceReference,
                    const TQ3Vector3D *inUpVector);
```

`inSourceReference`
A source reference.

`inUpVector` The desired untransformed up vector of the sound source. This up vector should be a unit vector.

function result A result code.

DESCRIPTION

The `SSpSource_SetUpVector` function sets the up vector of the sound source specified by the `inSourceReference` parameter to the vector specified by the `inUpVector` parameter. The `inUpVector` parameter specifies the untransformed up vector of the sound source (that is, the up vector of the sound source before the sound source's transform is applied). The actual up vector of the sound source is calculated at the time you call the `SSpSource_CalcLocalization` function. The default up vector of a sound source is the unit *y* vector (0, 1, 0).

SPECIAL CONSIDERATIONS

When you call `SSpSource_CalcLocalization`, the sound source's orientation and up vectors should not be parallel.

SSpSource_GetCameraPlacement

You can use the `SSpSource_GetCameraPlacement` function to get the position, orientation, and up vector of a sound source.

```
OSStatus SSpSource_GetCameraPlacement (
                    SSpSourceReference inSourceReference,
                    TQ3CameraPlacement *outCameraPlacement);
```

`inSourceReference`
A source reference.

`outCameraPlacement`
On entry, a pointer to a camera placement structure. On exit, that structure is updated to specify the position, orientation, and up vector of the sound source.

function result A result code.

DESCRIPTION

The `SSpSource_GetCameraPlacement` function updates the camera placement structure pointed to by the `outCameraPlacement` parameter to indicate the position, orientation, and up vector of the sound source specified by the `inSourceReference` parameter. (The point of interest of the camera placement is

the sound source's position plus its orientation vector.) This function is provided as a convenient method for getting that information in a single function call.

Note
The `TQ3CameraPlacement` data type is defined by QuickDraw 3D. See *3D Graphics Programming With QuickDraw 3D* for details. ◆

SEE ALSO

Instead of calling `SSpSource_GetCameraPlacement`, you could call `SSpSource_GetPosition`, `SSpSource_GetOrientation`, and `SSpSource_GetUpVector`.

SSpSource_SetCameraPlacement

You can use the `SSpSource_SetCameraPlacement` function to set the position, orientation, and up vector of a sound source.

```
OSStatus SSpSource_SetCameraPlacement (
                    SSpSourceReference inSourceReference,
                    const TQ3CameraPlacement *inCameraPlacement);
```

`inSourceReference`
A source reference.

`inCameraPlacement`
A pointer to a camera placement structure that specifies the desired position, orientation, and up vector of the sound source.

function result A result code.

DESCRIPTION

The `SSpSource_SetCameraPlacement` function sets the position, orientation, and up vector of the sound source specified by the `inSourceReference` parameter to the values contained in the camera placement structure pointed to by the `inCameraPlacement` parameter. This function is provided as a convenient method for setting that information in a single function call.

Note
The `TQ3CameraPlacement` data type is defined by QuickDraw 3D. See *3D Graphics Programming With QuickDraw 3D* for details. ◆

SEE ALSO

Instead of calling `SSpSource_SetCameraPlacement`, you could call `SSpSource_SetPosition`, `SSpSource_SetOrientation`, and `SSpSource_SetUpVector`.

SSpSource_GetVelocity

You can use the `SSpSource_GetVelocity` function to get the velocity of a sound source.

```
OSStatus SSpSource_GetVelocity (
                    SSpSourceReference inSourceReference,
                    TQ3Vector3D *outVelocity);
```

`inSourceReference`
A source reference.

`outVelocity` On exit, the most recent untransformed velocity vector of the sound source.

function result A result code.

DESCRIPTION

The `SSpSource_GetVelocity` function returns, in the `outVelocity` parameter, the most recent untransformed velocity vector of the sound source specified by the `inSourceReference` parameter. The velocity is the velocity set by the most recent call to `SSpSource_SetVelocity`.

SSpSource_SetVelocity

You can use the `SSpSource_SetVelocity` function to set the velocity of a sound source.

```
OSStatus SSpSource_SetVelocity (
                    SSpSourceReference inSourceReference,
                    const TQ3Vector3D *inVelocity);
```

`inSourceReference`
A source reference.

`inVelocity` A vector indicating the desired velocity of the sound source.

function result A result code.

DESCRIPTION

The `SSpSource_SetVelocity` function sets the untransformed velocity vector of the sound source specified by the `inSourceReference` parameter to the vector specified by the `inVelocity` parameter. The `inVelocity` parameter specifies the untransformed velocity of the sound source (that is, the velocity of the sound source before the sound source's transform is applied). The actual velocity of the sound source is calculated at the time you call the `SSpSource_CalcLocalization` function. If you call `SSpSource_CalcLocalization` and then change the location of the source (either by calling `SSpSource_SetPosition` or `SSpSource_SetTransform`) without calling `SSpSource_SetVelocity`, SoundSprocket will compute a velocity vector for you automatically.

A velocity is specified in listener units per second.

SSpSource_GetActualVelocity

You can use the `SSpSource_GetActualVelocity` function to get the world-space velocity of a sound source.

```
OSStatus SSpSource_GetActualVelocity (
                    SSpSourceReference inSourceReference,
                    TQ3Vector3D *outVelocity);
```

`inSourceReference`
A source reference.

`outVelocity` On exit, the world-space velocity vector of the sound source.

function result A result code.

DESCRIPTION

The `SSpSource_GetActualVelocity` function returns, in the `outVelocity` parameter, the world-space (that is, the transformed) velocity used in the most recent call to `SSpSource_CalcLocalization` for the sound source specified by the `inSourceReference` parameter.

SSpGetCPULoadLimit

You can use the `SSpGetCPULoadLimit` function to determine the maximum defined value for the CPU load.

```
OSStatus SSpGetCPULoad (UInt32 *outCPULoadLimit);
```

`inListenerReference`
A listener reference.

`outCPULoadLimit`
On exit, the maximum defined CPU load value.

function result A result code.

DESCRIPTION

The `SSpGetCPULoadLimit` function returns, in the `outCPULoadLimit` parameter, the maximum defined value for the CPU load. The value 0 represents the greatest CPU loading (and hence the best sound quality). This function determines the CPU load by creating a sound channel, calling `SndGetInfo` with the `siSSpCPULoadLimit` information selector and returning the result. If you have a sound channel handy, you may do this directly. The value returned by the `SSpGetCPULoadLimit` function represents the smallest CPU load factor (and hence the worst available sound quality.) The default CPU load factor is 0.

SSpSource_GetCPULoad

You can use the `SSpSource_GetCPULoad` function to get the current CPU load factor.

```
OSStatus SSpSource_GetCPULoad (SSpSourceReference inSourceReference,
                    UInt32 *outCPULoad);
```

`inSourceReference`
A source reference.

`outCPULoad` On exit, the maximum defined CPU load factor.

function result A result code.

DESCRIPTION

The `SSpSource_GetCPULoad` function returns, in the `outCPULoad` parameter, the current CPU load factor.

SSpSource_SetCPULoad

You can use the `SSpSource_SetCPULoad` function to set the CPU load factor of a source.

```
OSStatus SSpSource_SetCPULoad (
                    SSpSourceReference inSourceReference,
                    UInt32 inCPULoad);
```

`inSourceReference`
A source reference.

`inCPULoad` The desired CPU load factor of the source.

function result A result code.

DESCRIPTION

The `SSpSource_SetCPULoad` function sets the CPU load factor of the source specified by the `inSourceReference` parameter to the value specified by the

inCPULoad parameter. The value 0 represents the greatest CPU loading (and hence the best sound quality). The value returned by the SSpSource_SetCPULoad function represents the smallest CPU load factor (and hence the worst available sound quality.) The default CPU load factor is 0.

SSpSource_GetMode

You can use the SSpSource_GetMode function to get the current source mode for a sound source.

```
OSStatus SSpSource_GetMode (
                    SSpSourceReference inSourceReference,
                    UInt32 *outMode);
```

inSourceReference
A source reference.

outMode — On exit, the current source mode of the sound source. See "Source Modes" (page 1-28) for a description of the available source modes.

function result — A result code.

DESCRIPTION

The SSpSource_GetMode function returns, in the outMode parameter, the current source mode of the sound source specified by the inSourceReference parameter.

SSpSource_SetMode

You can use the SSpSource_SetMode function to set the source mode for a sound source.

```
OSStatus SSpSource_SetMode (
                    SSpSourceReference inSourceReference,
                    UInt32 inMode);
```

`inSourceReference`
A source reference.

`inMode` The desired source mode of the sound source. See "Source Modes" (page 1-28) for a description of the available source modes.

function result A result code.

DESCRIPTION

The `SSpSource_SetMode` function sets the source mode of the sound source specified by the `inSourceReference` parameter to the mode specified by the `inMode` parameter. The default source mode is `kSSpSourceMode_Localized`.

SSpSource_GetReferenceDistance

You can use the `SSpSource_GetReferenceDistance` function to get the reference distance of a sound source.

```
OSStatus SSpSource_GetReferenceDistance (
                    SSpSourceReference inSourceReference,
                    float *outReferenceDistance);
```

`inSourceReference`
A source reference.

`outReferenceDistance`
On exit, the distance, in listener units, from a listener to the point at which a sound source was recorded.

function result A result code.

DESCRIPTION

The `SSpSource_GetReferenceDistance` function returns, in the `outReferenceDistance` parameter, the distance from a listener to the point at which the sound source specified by the `inSourceReference` parameter was recorded.

SSpSource_SetReferenceDistance

You can use the `SSpSource_SetReferenceDistance` function to set the reference distance of a sound source.

```
OSStatus SSpSource_SetReferenceDistance (
                    SSpSourceReference inSourceReference,
                    float inReferenceDistance);
```

`inSourceReference`
A source reference.

`inReferenceDistance`
The distance, in listener units, from a listener to the point at which a sound source was recorded.

function result A result code.

DESCRIPTION

The `SSpSource_SetReferenceDistance` function sets the reference distance of the sound source specified by the `inSourceReference` parameter to the value specified by the `inReferenceDistance` parameter. The default reference distance is one meter.

SPECIAL CONSIDERATIONS

You should use the `SSpSource_SetReferenceDistance` function instead of the Sound Manager sound commands `ampCmd` or `volumeCmd` to change a sound's volume.

SSpSource_GetSize

You can use the `SSpSource_GetSize` function to get the size of a sound source.

```
OSStatus SSpSource_GetSize (
                    SSpSourceReference inSourceReference,
                    float *outLength,
                    float *outWidth,
                    float *outHeight);
```

`inSourceReference`
A source reference.

`outLength` On exit, the length of the sound source.

`outWidth` On exit, the width of the sound source.

`outHeight` On exit, the height of the sound source.

function result A result code.

DESCRIPTION

The `SSpSource_GetSize` function returns, in the `outLength`, `outWidth`, and `outHeight` parameters, the length, width, and height of the sound source specified by the `inSourceReference` parameter.

SSpSource_SetSize

You can use the `SSpSource_SetSize` function to set the size of a sound source.

```
OSStatus SSpSource_SetSize (
                    SSpSourceReference inSourceReference,
                    float inLength,
                    float inWidth,
                    float inHeight);
```

`inSourceReference`
A source reference.

`inLength` The desired length of the sound source.

`inWidth`	The desired width of the sound source.
`inHeight`	The desired height of the sound source.
function result	A result code.

DESCRIPTION

The `SSpSource_SetSize` function sets the length, width, and height of the sound source specified by the `inSourceReference` parameter to the values specified by the `inLength`, `inWidth`, and `inHeight` parameters. The length side is determined by the orientation vector, the height side is determined by the up vector, and the width is the remaining side of the box.

The default sides of the sound source box are 0.0 listener units.

SSpSource_GetAngularAttenuation

You can use the `SSpSource_GetAngularAttenuation` function to get the angular attenuation of a sound source.

```
OSStatus SSpSource_GetAngularAttenuation (
                    SSpSourceReference inSourceReference,
                    float *outConeAngle,
                    float *outConeAttenuation);
```

`inSourceReference`
A source reference.

`outConeAngle` On exit, the angle, in radians, at the apex of the angular attenuation cone. This angle is between 0 and 2π.

`outConeAttenuation`
On exit, the amount of attenuation, in decibels (dB), that occurs outside the angular attenuation cone.

function result A result code.

DESCRIPTION

The `SSpSource_GetAngularAttenuation` function returns, in the `outConeAngle` and `outConeAttenuation` parameters, the angle at the apex of the angular attenuation cone and the amount of attenuation that occurs outside that cone.

SSpSource_SetAngularAttenuation

You can use the `SSpSource_SetAngularAttenuation` function to set the angular attenuation of a sound source.

```
OSStatus SSpSource_SetAngularAttenuation (
                    SSpSourceReference inSourceReference,
                    float inConeAngle,
                    float inConeAttenuation);
```

`inSourceReference`
A source reference.

`inConeAngle` The angle, in radians, at the apex of the angular attenuation cone. This angle should be between 0 and 2π.

`inConeAttenuation`
The amount of attenuation, in decibels (dB), that occurs outside the angular attenuation cone.

function result A result code.

DESCRIPTION

The `SSpSource_SetAngularAttenuation` function sets the angle at the apex of the angular attenuation cone and the amount of attenuation for the sound source specified by the `inSourceReference` parameter to the values specified in the `inConeAngle` and `inConeAttenuation` parameters. The default settings provide no attenuation in any direction (that is, `inConeAngle` is 2π and `inConeAttenuation` is 0.0).

Sound Manager Functions

This section describes two functions introduced in Sound Manager version 3.1 that are not documented in *Inside Macintosh: Sound*. You need to use these functions when using the low-level SoundSprocket interfaces.

Getting and Setting Sound Channel Information

The Sound Manager provides functions that you can use to get and set information about a sound channel. For instance, you can use the `SndGetInfo` function to determine the sample rate of the current sound output device. Similarly, you can use the `SndSetInfo` function to change the characteristics of 3D filtering on a sound channel that uses the SoundSprocket localization component.

IMPORTANT
You should use `SndGetInfo` and `SndSetInfo` instead of trying to communicate directly with a sound component. ▲

SndGetInfo

You can use the `SndGetInfo` function to get information about a sound channel.

```
pascal OSStatus SndGetInfo (
                    SndChannelPtr chan,
                    OSType selector,
                    void *infoPtr);
```

`chan` A pointer to a sound channel.

`selector` A sound channel information selector specifying the kind of information you want to get about the sound channel. See "Sound Channel Information Selectors" (page 1-25) for a description of the selectors defined by SoundSprocket.

`infoPtr` A pointer to a buffer in which the information is to be returned. This buffer must be large enough for the type of information specified by the `selector` parameter.

function result A result code.

DESCRIPTION

The `SndGetInfo` function returns, in the `infoPtr` parameter, a pointer to a buffer holding the information of the type specified by the `selector` parameter about the sound channel specified by the `chan` parameter.

SndSetInfo

You can use the `SndSetInfo` function to set information for a sound channel.

```
pascal OSStatus SndSetInfo (
                    SndChannelPtr chan,
                    OSType selector,
                    void *infoPtr);
```

`chan` A pointer to a sound channel.

`selector` A sound channel information selector specifying the kind of information you want to set for the sound channel. See "Sound Channel Information Selectors" (page 1-25) for a description of the selectors defined by SoundSprocket.

`infoPtr` A pointer to a buffer containing the information you want to set. This buffer must be large enough for the type of information specified by the `selector` parameter.

function result A result code.

DESCRIPTION

The `SndSetInfo` function sets information for the sound channel specified by the `chan` parameter. The type of information you want to set is specified by the `selector` parameter. If additional data is required, it is passed in the buffer pointed to by the `infoPtr` parameter.

Summary of SoundSprocket

Constants

Component Types and Subtypes

```
enum {
    kSoundEffectsType                       = 'snfx'
};

enum {
    kReverbSubType                          = 'revb',
    kSSpLocalizationSubType                 = 'snd3'
};
```

Sound Channel Information Selectors

```
enum {
    siPreMixerSoundComponent                = 'prmx',
    siSSpCPULoadLimit                       = '3dll',
    siSSpSetup                              = '3dst',
    siSSpLocalization                       = '3dif',
    siSSpFilterVersion                      = '3dfv'
};
```

Speaker Types

```
enum {
    kSSpSpeakerKind_Stereo                  = 0,
    kSSpSpeakerKind_Mono                    = 1,
    kSSpSpeakerKind_Headphones              = 2
};
```

Sound Media

```
enum {
    kSSpMedium_Air                              = 0,
    kSSpMedium_Water                            = 1
};
```

Source Modes

```
enum {
    kSSpSourceMode_Unfiltered                   = 0,
    kSSpSourceMode_Localized                    = 1,
    kSSpSourceMode_Ambient                      = 2,
    kSSpSourceMode_Binaural                     = 3
};
```

Data Types

```
typedef struct SSpSourcePrivate                 *SSpSourceReference;

typedef struct SSpListenerPrivate               *SSpListenerReference;
```

Filter Version Structure

```
typedef struct SSpFilterVersionData {

    OSType                          manufacturer;
    NumVersion                      version;
    UInt32                          reserved;
} SSpFilterVersionData;
```

Sound Component Link Structure

```
struct SoundComponentLink {
    ComponentDescription            description;
    SoundSource                     mixerID;
    SoundSource *                   linkID;
};
```

```
typedef struct SoundComponentLink SoundComponentLink;
typedef SoundComponentLink *SoundComponentLinkPtr;
```

3D Sound Setup Structure

```
typedef struct SSpSetupData {
    UInt32                              speakerKind;
    float                               speakerAngle;
    UInt32                              reserved0;
    UInt32                              reserved1;
} SSpSetupData;
```

Source Location Structure

```
typedef struct SSpLocationData {
    float                               elevation;
    float                               azimuth;
    float                               distance;
    float                               projectionAngle;
    float                               sourceVelocity;
    float                               listenerVelocity
} SSpLocationData;
```

Virtual Source Structure

```
typedef struct SSpVirtualSourceData {
    float                               attenuation;
    SSpLocationData                     location;
} SSpVirtualSourceData;
```

3D Sound Information Structure

```
typedef struct SSpLocalizationData {
    UInt32                              cpuLoad;
    UInt32                              medium;
    float                               humidity;
    float                               roomSize;
    float                               roomReflectivity;
    float                               reverbAttenuation;
    UInt32                              sourceMode;
    float                               referenceDistance;
```

```
    float                               coneAngleCos;
    float                               coneAttenuation;
    SSpLocationData                     currentLocation;
    UInt32                              reserved0;
    UInt32                              reserved1;
    UInt32                              reserved2;
    UInt32                              reserved3;
    UInt32                              virtualSourceCount;
    SSpVirtualSourceData                virtualSource[4];
} SSpLocalizationData;

typedef Boolean (*SSpEventProcPtr) (EventRecord* InEvent);
```

SoundSprocket Functions

Controlling Sound Output Devices

```
OSStatus SSpConfigureSetup        (SSpEventProcPtr inEventProcPtr);
```

Creating and Managing Listeners

```
OSStatus SSpListener_New          (SSpListenerReference *outListenerReference);

OSStatus SSpListener_Dispose      (SSpListenerReference inListenerReference);

OSStatus SSpListener_GetTransform (SSpListenerReference inListenerReference,
                                   TQ3Matrix4x4 *outTransform);

OSStatus SSpListener_SetTransform (SSpListenerReference inListenerReference,
                                   const TQ3Matrix4x4 *inTransform);

OSStatus SSpListener_GetPosition  (SSpListenerReference inListenerReference,
                                   TQ3Point3D *outPosition);

OSStatus SSpListener_SetPosition  (SSpListenerReference inListenerReference,
                                   const TQ3Point3D *inPosition);

OSStatus SSpListener_GetOrientation (
                                   SSpListenerReference inListenerReference,
                                   TQ3Vector3D *outOrientation);
```

```
OSStatus SSpListener_SetOrientation (
                                    SSpListenerReference inListenerReference,
                                    const TQ3Vector3D *inOrientation);

OSStatus SSpListener_GetUpVector  (SSpListenerReference inListenerReference,
                                    TQ3Vector3D *outUpVector);

OSStatus SSpListener_SetUpVector  (SSpListenerReference inListenerReference,
                                    const TQ3Vector3D *inUpVector);

OSStatus SSpListener_GetCameraPlacement (
                                    SSpListenerReference inListenerReference,
                                    TQ3CameraPlacement *outCameraPlacement);

OSStatus SSpListener_SetCameraPlacement (
                                    SSpListenerReference inListenerReference,
                                    const TQ3CameraPlacement *inCameraPlacement);

OSStatus SSpListener_GetVelocity  (SSpListenerReference inListenerReference,
                                    TQ3Vector3D *outVelocity);

OSStatus SSpListener_SetVelocity  (SSpListenerReference inListenerReference,
                                    const TQ3Vector3D *inVelocity);

OSStatus SSpListener_GetActualVelocity (
                                    SSpListenerReference inListenerReference,
                                    TQ3Vector3D *outVelocity);

OSStatus SSpListener_GetMedium    (SSpListenerReference inListenerReference,
                                    UInt32 *outMedium,
                                    float *outHumidity);

OSStatus SSpListener_SetMedium    (SSpListenerReference inListenerReference,
                                    UInt32 inMedium,
                                    float inHumidity);

OSStatus SSpListener_GetReverb    (SSpListenerReference inListenerReference,
                                    float *outRoomSize,
                                    float *outRoomReflectivity,
                                    float *outReverbAttenuation);

OSStatus SSpListener_SetReverb    (SSpListenerReference inListenerReference,
                                    float inRoomSize,
                                    float inRoomReflectivity,
                                    float inReverbAttenuation);
```

```
OSStatus SSpListener_GetMetersPerUnit (
                                SSpListenerReference inListenerReference,
                                float *outMetersPerUnit);

OSStatus SSpListener_SetMetersPerUnit (
                                SSpListenerReference inListenerReference,
                                float inMetersPerUnit);
```

Creating and Managing 3D Sound Sources

```
OSStatus SSpSource_New            (SSpSourceReference *outSourceReference);

OSStatus SSpSource_Dispose        (SSpSourceReference inSourceReference);

OSStatus SSpSource_CalcLocalization (
                                SSpSourceReference inSourceReference,
                                SSpListenerReference inListenerReference,
                                SSpLocalizationData *outLocalization);

OSStatus SSpSource_GetTransform   (SSpSourceReference inSourceReference,
                                TQ3Matrix4x4 *outTransform);

OSStatus SSpSource_SetTransform   (SSpSourceReference inSourceReference,
                                const TQ3Matrix4x4 *inTransform);

OSStatus SSpSource_GetPosition    (SSpSourceReference inSourceReference,
                                TQ3Point3D *outPosition);

OSStatus SSpSource_SetPosition    (SSpSourceReference inSourceReference,
                                const TQ3Point3D *inPosition);

OSStatus SSpSource_GetOrientation (SSpSourceReference inSourceReference,
                                TQ3Vector3D *outOrientation);

OSStatus SSpSource_SetOrientation (SSpSourceReference inSourceReference,
                                const TQ3Vector3D *inOrientation);

OSStatus SSpSource_GetUpVector    (SSpSourceReference inSourceReference,
                                TQ3Vector3D *outUpVector);

OSStatus SSpSource_SetUpVector    (SSpSourceReference inSourceReference,
                                const TQ3Vector3D *inUpVector);

OSStatus SSpSource_GetCameraPlacement (
                                SSpSourceReference inSourceReference,
                                TQ3CameraPlacement *outCameraPlacement);
```

```
OSStatus SSpSource_SetCameraPlacement (
                                    SSpSourceReference inSourceReference,
                                    const TQ3CameraPlacement *inCameraPlacement);

OSStatus SSpSource_GetVelocity    (SSpSourceReference inSourceReference,
                                    TQ3Vector3D *outVelocity);

OSStatus SSpSource_SetVelocity    (SSpSourceReference inSourceReference,
                                    const TQ3Vector3D *inVelocity);

OSStatus SSpSource_GetActualVelocity (
                                    SSpSourceReference inSourceReference,
                                    TQ3Vector3D *outVelocity);

OSStatus SSpGetCPULoadLimit       (UInt32 *outCPULoadLimit);

OSStatus SSpSource_GetCPULoad     (SSpSourceReference inSourceReference,
                                    UInt32 *outCPULoad);

OSStatus SSpSource_SetCPULoad     (SSpSourceReference inSourceReference,
                                    UInt32 inCPULoad);

OSStatus SSpSource_GetMode        (SSpSourceReference inSourceReference,
                                    UInt32 *outMode);

OSStatus SSpSource_SetMode        (SSpSourceReference inSourceReference,
                                    UInt32 inMode);

OSStatus SSpSource_GetReferenceDistance (
                                    SSpSourceReference inSourceReference,
                                    float *outReferenceDistance);

OSStatus SSpSource_SetReferenceDistance (
                                    SSpSourceReference inSourceReference,
                                    float inReferenceDistance);

OSStatus SSpSource_GetSize        (SSpSourceReference inSourceReference,
                                    float *outLength,
                                    float *outWidth,
                                    float *outHeight);

OSStatus SSpSource_SetSize        (SSpSourceReference inSourceReference,
                                    float inLength,
                                    float inWidth,
                                    float inHeight);
```

```
OSStatus SSpSource_GetAngularAttenuation (
                                    SSpSourceReference inSourceReference,
                                    float *outConeAngle,
                                    float *outConeAttenuation);

OSStatus SSpSource_SetAngularAttenuation (
                                    SSpSourceReference inSourceReference,
                                    float inConeAngle,
                                    float inConeAttenuation);
```

Sound Manager Functions

Getting and Setting Sound Channel Information

```
pascal OSStatus SndGetInfo        (SndChannelPtr chan, OSType selector, void *infoPtr);

pascal OSStatus SndSetInfo        (SndChannelPtr chan, OSType selector, void *infoPtr);
```

Result Codes

`paramErr`	–50	Parameter type error or out of range error
`memFullErr`	–108	Unable to allocate memory required for task
`kSSpInternalErr`	–30340	Corrupted SoundSprocket or other error
`kSSpVersionErr`	–30341	Sound Manager 3.2.1 or later not installed
`kSSpCantInstallErr`	–30342	Filter installation failure
`kSSpParallelUpVectorErr`	–30343	Orientation and up vector are parallel
`kSSpScaleToZeroErr`	–30344	Transformation matrix has a vector of 0

DrawSprocket

Contents

DrawSprocket is the part of Apple Game Sprockets that gives your Macintosh video game control over special display and device-access features. For some features, DrawSprocket interacts directly with the Mac OS; for others, it can make use of specialized video subsystems and third party video cards. Those features that are not directly supported in hardware are emulated in software in a highly optimized manner, allowing you to rely on their presence when creating your game.

Before reading this chapter, you should be generally familiar with Apple Game Sprockets, as described in the preface to this book. This chapter does not discuss Macintosh graphics systems or drawing functions; for that information, please consult the appropriate reference, such as *Inside Macintosh: Imaging With QuickDraw.* This chapter does not discuss Macintosh video hardware; for that information, please consult *Macintosh Family Hardware Reference* and *Designing PCI Cards and Drivers for the Macintosh Family.*

DrawSprocket relies on the Display Manager for some of its tasks, and you can use Display Manager features in conjunction with DrawSprocket. See the Display Manager documentation for more information.

This chapter starts by introducing the features of DrawSprocket, and then shows how your game can use DrawSprocket to choose a resolution and pixel depth for drawing, allocate a display, perform gamma fading for smooth transitions, and use double-buffering or page-flipping to draw to the display. It also describes other features such as overlays, underlays, and pixel scaling. The section "DrawSprocket Reference" (page 2-22) provides a complete reference to the constants, data structures, and functions provided by DrawSprocket. For a list of constants, data structures, and functions see "Summary of DrawSprocket" (page 2-77).

About DrawSprocket

DrawSprocket provides an application interface that lets your game easily access and manipulate system display resources. The features supported by DrawSprocket include the following:

- Display configuration: convenient, optimized selection of display devices and resolution and pixel depth

- Blanking window: a convenient way to hide and show the desktop, menu bar, and other system tools such as the control strip
- Double- or triple-buffered drawing: page flipping when supported by the video hardware or software buffering when page flipping is not available
- Display fading: sophisticated gamma fading capability
- Overlays and underlays: creation of foreground and background images
- Back buffer scaling: support for pixel scaling with optional bilinear interpolation
- Color-table management: easy retrieval and modification of entries in color lookup tables
- Frame skipping: support for evening the frame display rate across your game
- Debugging support: access to the screen and system resources at all times, even when the screen is blanked and faded

When your game uses DrawSprocket, it becomes the "owner" of the system display resources. All video devices in the system are hidden, and you can freely change pixel depths, color palettes, and screen resolutions at will.

Display Configuration

An individual Macintosh computer can simultaneously support more than one monitor, and many monitors can support several different combinations of **resolution** (horizontal and vertical timing of a display mode) and **pixel depth** (number of bits per pixel). In DrawSprocket, each combination of resolution and pixel depth is known as a **context**. At any one time, a given system may allow one or more graphics devices, each with its associated contexts, as well as other characteristics such as color support, hardware acceleration, and display buffering.

Your game, on the other hand, has its own display requirements and preferences, and also may want to take advantage of special display features when they are available. An important aspect of DrawSprocket is that it coordinates your game's needs with the currently supported display characteristics and features of the user's system.

The key to this selection system is the **context attributes structure**, a structure that both your game and DrawSprocket use to pass information to each other. You use certain fields of the context attributes structure to request a context of a

given size (width and height, in pixels), pixel depth, color capabilities, and so on. DrawSprocket then finds the closest match to your requests using the `DSPFindBestContext` function. Alternatively, you can use other routines to let the user or the game determine which context to use. Once a context is identified, you refer to it with the `DSpContextReference` type.

See "Context Attributes Structure" (page 2-27) for descriptions of the structure's fields. See "Choosing a Context" (page 2-12) for examples of its use.

Contexts and the Play State

After you find a context that best suits your needs, you must reserve it before using it. Reserving a context for a display device tells that device to use a particular display depth and resolution. Once a context is reserved, all other contexts for the device become unavailable. When you are finished using a context you release it, making all contexts associated with the display available again.

When you reserve a context, its **play state** is initially inactive. In that play state, the context has no effect on the screen display. However, as soon as you make the play state active, the context's resolution, pixel depth, and other characteristics take effect. At that point, DrawSprocket also covers the entire display with a **blanking window,** which completely hides the desktop, menu bar, and other system adornments to provide a uniform background color for your game to draw over.

When your game is finished using a display, you can make the context's play state inactive. In that state, DrawSprocket removes the blanking window and restores the user's original display characteristics. See "Play State" (page 2-25) for descriptions of a context's play states.

A context reference is not consistent across executions of your game. This means that every time your game executes you must relocate the best context for your game. DrawSprocket does provide calls to enable you to save a particular context description to disk and then attempt to restore it later. Restoration may fail if the user reconfigured the displays in the system. See the `DSpContext_Flatten` function (page 2-39) for more information.

Page Flipping and Software Buffering

During execution your game does all its drawing to a **back buffer,** an offscreen buffer DrawSprocket draws into while another image buffer is being displayed.

DrawSprocket transfers the completed offscreen images to the display in response to a `DSpContext_SwapBuffers` call. DrawSprocket supports **page flipping,** in which the back buffer DrawSprocket provides is a video page located in VRAM that can be displayed by the video controller in rotation with other VRAM pages. Page flipping is a very fast way change an entire graphics display without tearing artifacts, because the game only needs to tell the video controller to begin displaying the new page—the actual page data (which can be very large) does not need to be copied.

If page flipping is not available, or you have told DrawSprocket that you prefer not to use page flipping, the back buffer is located in DRAM. When the game tells DrawSprocket to make the buffer visible to the user, DrawSprocket copies it using a highly optimized blitter that is customized to your game's needs.

For both page flipping and software buffering, you can tell DrawSprocket to use double buffering or triple buffering during the execution of your game. With double buffering you have two buffers—one that is displayed and one that is hidden (used for rendering). With triple buffering you have three buffers—one that is displayed, and two that are hidden. Requests for page flipping and double or triple buffering are made in the context attributes structure passed in when you reserve a context for play.

How Triple Buffering Works

When the game is using double buffering and tells DrawSprocket to make the back buffer visible, you may want the game to begin drawing the next frame right away. If the game is fast enough, it could even be rendering while DrawSprocket is moving the back buffer to the display. But when DrawSprocket is drawing to the display, the back buffer is busy, so the game has to wait, wasting precious CPU cycles. You can request that, if DrawSprocket detects this occurring, it will throw another buffer into the available buffer queue, so that the game may begin rendering into it while the first back buffer is in use. This is called **triple buffering**.

The flip side of triple buffering is that the game may retrieve a back buffer (buffer #1), tell DrawSprocket to show it, retrieve a new back buffer (buffer #2), and then tell DrawSprocket to show buffer #2 before buffer #1 has been displayed. In this case, the game is rendering too fast for its own good and can introduce video lag. DrawSprocket can detect such a situation and will queue the buffer swap and drop the game back to double buffering. The next time the game retrieves the back buffer, all the buffers may be busy and the call may

block (avoid blocking by using `DSpContext_IsBusy` before retrieving the back buffer).

A possible scenario is a game riding the edge between the need for double and triple buffering, which causes DrawSprocket to alternate between the two buffering methods. There is no performance penalty for this condition, and the transitions will not affect game performance. The net effect is that the game will always be triple buffered, because it will always be using the next available back buffer.

A side effect of running your game faster than the achievable display frame rate (and hence, faster than can be triple buffered) is that your game will always run two frames behind what the user sees. Consider what happens when you have two buffers queued for a display swap: a frame is currently visible on the display, the first queued buffer will be displayed at the next (first) VBL, and the second queued buffer will be displayed at the second VBL.

Gamma Fading

When a monitor switches resolution modes, there is typically a visible flash or flicker that can be annoying to the user. When your game starts it usually switches resolution modes, and when it quits it switches back. Depending on its features, your game might also switch modes during execution.

To hide the flicker, you can fade the display to black, make the switch, and then bring the display back to full intensity. A typical procedure for fading out is called *indexed fading*, in which you repeatedly draw the screen with all colors at proportionally lesser and lesser RGB intensities, reaching zero intensity over the course of a second or so. You then wait about half a second for the mode switch to take effect on the monitor, and fade the intensity back in.

DrawSprocket provides a facility to achieve a superior, more convenient, and more flexible fading process. Using the DrawSprocket functions, you can fade out one or all displays in the system and achieve a variety of interesting effects at the same time. The three functions—`DSpContext_FadeGamma`, `DSpContext_FadeGammaIn`, and `DSpContext_FadeGammaOut`—perform a gamma fade. A **gamma fade** is a fade based on a gamma table, which accounts for nonlinearities in the display's color response. With a gamma fade, bright colors won't remain visible after the darker colors have already disappeared, as can happen with indexed fading.

Because of its flexibility, you might want to use DrawSprocket's gamma fading for other transitions besides switches in resolution mode. The DrawSprocket gamma fade functions provides these advantages:

- You can fade direct devices as well as indexed devices.
- You can fade all the graphics devices in your system at once, or one at a time.
- When fading out, you don't have to fade to black. You can define any RGB color as the "zero intensity" color.
- You can overdrive the gamma table by specifying an intensity greater than 100 percent. This feature is good for those blinding flashes that occur when a rocket detonates in the player's face.
- You can automatically fade smoothly and gradually to zero intensity with a single call to `DSpContext_FadeGammaOut`, and fade smoothly back in with a second call to `DSpContext_FadeGammaIn`.
- For more direct control over the fading process, you can make repeated calls to `DSpContext_FadeGamma` and manually control the amount of fade for each step.
- By combining manual control with manipulation of the zero-intensity color, you can achieve special effects. For example, you could fade halfway to red and then the rest of the way to black.

See the description of the `DSpContext_FadeGamma` function (page 2-45) for more information.

Underlays, Overlays, and Transparency Masks

An **underlay** is an image that is used as a background for the back buffer and is restored automatically whenever the back buffer is returned from `DSpContext_GetBackBuffer`. Typically, you use an underlay where you have a static background (or a background that may stay static for at least a few frames), such as in a sprite-based game, or in a scroller game where the background only moves when the player crosses a threshold near the edge of the screen.

An **overlay** is an image that is superimposed on the back buffer before it is displayed. This occurs during the call to `DSpContext_SwapBuffers`. You can use an overlay as a control panel type of image, such as that of an airplane cockpit or a radar screen, or decorative scenery such as a border around the active playing area. An overlay must have transparent areas, known as a

transparency mask, in order for the back buffer to show through. For more information on how underlays and overlays work see “Creating Underlays and Overlays” (page 2-16).

Pixel Scaling

Your game may scale the back buffer. With scaling, the resolution of the underlay, overlay, and display all remain the same, but the area of the back buffer that is used is reduced. For example, with a 2x scaling factor, DrawSprocket uses only the upper-left quadrant of the back buffer when it composites the image and displays it. However, that quadrant is doubled horizontally and vertically to fill the entire display.

You have the option of using bilinear interpolation with pixel scaling. **Bilinear interpolation** averages the pixels and determines a pixel value that falls between the two original pixels. This results in smoother images, but as scaling is increased images begin to look out of focus (but at least they won’t look chunky). For each scaling factor that DrawSprocket provides, there is an identical factor with bilinear interpolation enabled. For more information see “Pixel Scaling” (page 2-25) and the `DSpContext_SetScale` function on (page 2-58).

Other Features

DrawSprocket allows you set a maximum frame-refresh rate for your game, so that it will not appear to run at uneven speeds, slowing down in portions that require complex rendering and speeding up during simpler drawing. By using the `DSpContext_SetMaxFrameRate` function (page 2-55), you can give yourself enough time to render each frame before the screen is refreshed.

You can easily manipulate the colors in a color lookup table with a DrawSprocket convenience function. See the `DSpContext_SetCLUTEntries` function (page 2-70) for more information.

Finally, DrawSprocket provides two utility functions. The special function `DSpSetDebugMode` (page 2-73) helps you when debugging. When you use this function, the display screen and system resources remain visible at all times, even behind the blanking window and even if your code has performed a fade to zero intensity. This feature exists to give you access to the debugger screen at all times. The `DSpContext_SetVBLProc` function (page 2-74) allows you to piggyback your own VBL tasks to a particular context.

Video Driver Support

DrawSprocket never communicates directly with the Macintosh video hardware. Instead, it relies on standard Macintosh video drivers to provide access to any special hardware features. Video drivers that do not currently support page flipping must be revised so that video games that use DrawSprocket can have access to this capability when it is present in the hardware.

The Apple Computer video drivers are being revised, and third-party card and driver developers are strongly encouraged to follow suit. Documentation on how driver developers can best support DrawSprocket is in preparation.

If you are developing a driver to work with DrawSprocket, make sure that, if your video hardware does not support a feature, your driver does not emulate that feature in software. DrawSprocket itself provides highly optimized emulations of hardware capabilities. For example, if page-flipping is not supported by the hardware, your video driver should not attempt to simulate it. If page-flipping is not present, DrawSprocket manages the transition of the background page to the display in as efficient a manner as possible, and the game need not be aware of how the transition occurs.

Using DrawSprocket

To use DrawSprocket in a game, you need to find the context that best matches the display configuration requested by the game and then reserve and activate the context. When activating the context, you should use the gamma fade capabilities to avoid flicker. During game play you can designate alternate buffers as underlays or overlays to avoid redrawing repeated graphic elements. Finally, you will probably want to take advantage of DrawSprocket's hardware or software page flipping. This section gives code examples of these tasks, as well as an example of cleaning up before quitting and using DrawSprocket's debug mode.

Choosing a Context

One of your game's first tasks is to find the best available context. You can use the `DSpFindBestContext` function to have DrawSprocket find a context that matches the attributes the game requests.

When you call `DSpFindBestContext`, passing in a `DSpContextAttributes` structure, DrawSprocket interprets the attributes specified in the attributes structure as requirements. This insures that the game can search for only those contexts that meet a specific criteria, and know that any matches found will meet those specifications. This is in contrast to what happens when you call `DSpContext_Reserve`, where the specified attributes are interpreted as requests. This makes it easy for the game to request a feature at reservation time, knowing that the context will be reserved even if that feature is not available.

For example, if the `kDSpContextOption_PageFlip` bit is set in the attributes structure's `contextOptions` field when you call `DSpFindBestContext`, only contexts with hardware page flipping will be considered in the search. However, when you call `DSpContext_Reserve` to reserve the context and the `contextOptions` field has the `kDSpContextOption_PageFlip` bit set, if page flipping is not available for the context being reserved, DrawSprocket will reserve it anyway and use software buffering.

Initializing the Context Attributes Structure

Before beginning, you should initialize the `DSpContextAttributes` structure you will pass to `DSpFindBestContext` so that the fields contain no garbage that could cause DrawSprocket to return an error. This is especially important for fields that DrawSprocket can't check, such as the `colorTable` field. If there is a bad value in the `colorTable` field when a call to reserve a context is made, DrawSprocket will try to use that color table (causing unpleasant visits to the debugger). The context attributes structure is described on (page 2-27).

Listing 2-1 Initializing a context attributes structure

```
void MyInitAttributes (DSpContextAttributes *inAttributes)
{
    if (NULL == inAttributes)
        DebugStr("\pStimpy! You Idiot!");

    inAttributes->frequency = 0;
    inAttributes->displayWidth = 0;
    inAttributes->displayHeight = 0;
    inAttributes->reserved1 = 0;
    inAttributes->reserved2 = 0;
    inAttributes->colorNeeds = 0;
```

```
    inAttributes->colorTable = NULL;
    inAttributes->contextOptions = 0;
    inAttributes->backBufferDepthMask = 0;
    inAttributes->displayDepthMask = 0;
    inAttributes->backBufferBestDepth = 0;
    inAttributes->displayBestDepth = 0;
    inAttributes->pageCount = 0;
    inAttributes->gameMustConfirmSwitch = false;
    inAttributes->reserved3[0] = 0;
    inAttributes->reserved3[1] = 0;
    inAttributes->reserved3[2] = 0;
    inAttributes->reserved3[3] = 0;
}
```

Finding a Context

This section shows how to find a context that matches attributes required by the game. Listing 2-2 shows how to use `DSpFindBestContext` (page 2-31) to tell DrawSprocket to find a 320x200 color display with a pixel depth of 8.

Listing 2-2 Finding the best context

```
DSpContextAttributes theDesiredAttributes;
DSpContextReference theContext;
OSStatus theError;

MyInitAttributes(&theDesiredAttributes);
theDesiredAttributes.displayWidth = 320;
theDesiredAttributes.displayHeight = 200;
theDesiredAttributes.colorNeeds = kDSpColorNeeds_Require;
theDesiredAttributes.backBufferDepthMask = kDSpDepthMask_8;
theDesiredAttributes.displayDepthMask = kDSpDepthMask_8;
theDesiredAttributes.backBufferBestDepth = 8;
theDesiredAttributes.displayBestDepth = 8;
theError = DSpFindBestContext(&theDesiredAttributes,
    &theContext);
```

Reserving and Activating a Context

Once you have found the best context, you reserve it and then activate it. Follow these steps:

1. Call the `DSpContext_Reserve` function (page 2-41). Reserving a context does not result in any change to the user's display because the play state of the context is initially inactive. At this point, the user is still interacting with the system in a normal fashion.
2. Set the play state of your context to active with the `DSpContext_SetState` function (page 2-43). This causes the display to change to the resolution you have chosen, and causes a blanking window to cover the entire screen, giving your game complete control over the display.

   (Before activating the context, however, you may want to fade the screen display; see "Fading the Display" (page 2-17) for an example.)

Prior to reserving the context be sure to request any features you would like to use that were not requirements passed to the `DSpFindBestContext` call. For example, if you want page flipping, request

```
theDesiredAttributes.contextOptions |= kDSpContextOption_PageFlip;
```

DrawSprocket will use software buffering if hardware page flipping is not available.

You can also request that DrawSprocket use double or triple buffering by setting the `kDSpContextOption_TripleBuffer` flag. If you want only double buffering, perhaps because of memory constraints, you should turn off the `kDSpContextOption_TripleBuffer` option bit.

If both the triple-buffering option bit and the page-flipping option bit are set, and hardware page flipping is available but there are only two video pages — not three—DrawSprocket considers page flipping to be a more important option than triple buffering and drops you down to two VRAM pages, maintaining page flipping.

Listing 2-3 Reserving and activating a context

```
theError = DSpContext_Reserve(theContext, &theDesiredAttributes);

theError = DSpContext_SetState(theContext, kDSpContextState_Active);
```

Creating Underlays and Overlays

You can allocate alternate buffers to use for underlays and overlays. An underlay is useful in games that need to restore from a static background. An overlay is useful for things such as a cockpit view that is superimposed on the back buffer before it is displayed.

How Underlays Work

To create an underlay you first create an alternate buffer and draw into it. Then you designate the alternate buffer as the current underlay. Listing 2-4 shows how to use the `DSpAltBuffer_New` function (page 2-60) to create an alternate buffer and the `DSpContext_SetUnderlayAltBuffer` (page 2-63) to designate the new buffer as the underlay for a context.

Listing 2-4 Creating an underlay

```
theError = DSpAltBuffer_New(theContext, false, &theUnderlay);

theError = DSpContext_SetUnderlayAltBuffer(theContext, theUnderlay);
```

Once you establish an underlay, every time you call `DSpContext_GetBackBuffer`, the invalid rectangles associated with the returned back buffer are used to copy the image data from the underlay into the back buffer (the first time you get the back buffer, the entire buffer is invalid). In a game with sprites overlaid on the background (the underlay), the sprites are erased when the buffer's invalid rectangles are used to copy the underlay data into the back buffer.

You can make changes in the underlay image, but you must invalidate the underlay rectangles that have been modified in order for the data to be updated to each back buffer (otherwise the back buffer will only have its own invalid rectangles restored). To do this call the `DSpAltBuffer_InvalRect` function (page 2-65).

You can also have several underlays and switch between them with additional calls to `DSpContext_SetUnderlayAltBuffer`. However, each underlay change forces a full update in the back buffer unless you immediately invalidate the unique portions of the new underlay with a call to `DSpAltBuffer_InvalRect`. For example, in a scroller game with a constant horizon area (such as a mountain range), after you switched underlays you would invalidate the areas in the new

underlay that differed from the previous one to avoid the full back buffer update. See "Improving Performance With Dirty Rectangles" (page 2-20).

How Overlays Work

As with underlays, you first create a buffer and draw into it. Listing 2-5 shows how to allocate an alternate buffer to use as an overlay using `DSpContext_SetOverlayAltBuffer` (page 2-62).

Listing 2-5 Creating an overlay

```
theError = pAltBuffer_New(theContext, false, &theOverlay);

theError = DSpContext_SetOverlayAltBuffer(theContext, theOverlay);
```

To create the transparency mask that allows the back buffer to show through, DrawSprocket provides the `DSpAltBuffer_RebuildTransparencyMask` function (page 2-66). Building the transparency mask is a CPU intensive operation and shouldn't be done during game play. Build or rebuild your masks at game startup or at a point in the game where a pause is acceptable (such as between levels).

As with underlays, you can change an overlay or designate a different buffer to serve as the overlay during the game's execution. For example, if you have a cockpit image that changes during game play, as when the user looks out the side of a plane, you can set up multiple overlays and switch them between frames.

You should call `DSpAltBuffer_InvalRect` on the new overlay if only portions of the overlay have changed (such as a bullet hole appearing in a window); otherwise, the entire buffer is updated. See "Improving Performance With Dirty Rectangles" (page 2-20).

Fading the Display

When you first make your context's play state active, and then subsequently deactivate it when quitting, that is likely to cause a resolution change and an accompanying annoying flicker or flash. Therefore, you will probably want to fade your display out before the change occurs, and then fade it back in afterward.

The gamma fading supported by DrawSprocket is sophisticated and versatile; you may want to use it at other times as well as during resolution changes. See "Gamma Fading" (page 2-9) for more information.

Listing 2-6 shows how to use `DSpContext_FadeGammaOut` to automatically fade out all displays to black, put a context in the active state with `DSpContext_SetState`, and then fade back in using `DSpContext_FadeGammaIn`. You must have at least one reserved context to automatically fade in or out or you will get an error.

The automatic gamma fade functions allow you to specify a zero-intensity color other than black and to fade only specified displays. For details see the `DSpContext_FadeGammaOut` function (page 2-47) and the `DSpContext_FadeGammaIn` function (page 2-48). DrawSprocket also provides another function for manual fading—you can fade partway in or out at a speed you choose. See the `DSpContext_FadeGamma` function (page 2-45).

Listing 2-6 Automatically fading the display

```
/* fade out all displays */
theError = DSpContext_FadeGammaOut(NULL, NULL);

/* put the context into the active state */
theError = DSpContext_SetState(theContext, kDSpContextState_Active);

/* fade back in */
theError = DSpContext_FadeGammaIn(NULL, NULL);
```

Double-Buffered Drawing

If you want to use double-buffering (or hardware-supported page flipping) for your drawing, DrawSprocket facilitates easy access to your offscreen buffer. Here are the steps required for double-buffered drawing:

1. For each frame you draw, call the `DSpContext_GetBackBuffer` function to obtain a pointer to the graphics port you render to.
2. Draw the portions of the frame that need updating.

3. Call the `DSpContext_InvalBackBufferRect` function for each rectangular area that you have invalidated. This step can greatly enhance performance (see "Improving Performance With Dirty Rectangles" (page 2-20)).
4. Call the `DSpContext_SwapBuffers` function. At the next vertical retrace DrawSprocket copies the invalidated portions of the back buffer to the device's display.

Listing 2-7 shows how to get the back buffer and set it up as the current port, ready to draw into.

Listing 2-7 Getting the back buffer

```
/* get the back buffer */
theError = DSpContext_GetBackBuffer(theContext, kDSpBufferKind_Normal,
    &theBackBuffer);

/* set the back buffer to be the current port */
SetPort((GrafPtr)theBackBuffer);
```

The code in Listing 2-8 draws a rectangle into the back buffer. If you are using an underlay, there is no need to erase the back buffer before drawing; the buffer is automatically filled with the underlay image.

Listing 2-8 Drawing into the buffer

```
if (!inUseUnderlay)
    EraseRect(&theBackBuffer->portRect);

/* fill the display with a rectangle */
RGBForeColor(&theColor);
PaintRect(&theRectangleRect);
```

After invalidating the changed parts of the buffer, you are ready to draw the back buffer to the display by calling `DSpContext_SwapBuffers`, as shown in Listing 2-9.

Listing 2-9 Swapping buffers

```
theError = DSpContext_SwapBuffers(theContext, NULL, 0);
```

Improving Performance With Dirty Rectangles

Using dirty rectangles is crucial to achieving optimum performance for your game. A **dirty rectangle** is an area of the back buffer that has been invalidated so that DrawSprocket knows it has been changed. When you draw into a back buffer, always call `DSpContext_InvalBackBufferRect` to let DrawSprocket know the bounding rectangle of the area that has been altered. If you do not invalidate the rectangles you use, the entire buffer must be transferred at the next swap.

Invalidating rectangles is most important with software buffering, but it also affects performance with page flipping if you are using underlays or overlays in your game. In any event, you won't pay a performance penalty for telling DrawSprocket about the invalid rectangles—you will only pay a penalty for not doing so.

DrawSprocket provides a separate call for invalidating rectangles in the buffer designated as the current underlay or overlay. If you change pixel data for a current underlay or overlay, you must call `DSpAltBuffer_InvalRect` in order for the change to be brought into the back buffer. Furthermore, if you change an overlay in such a way that the transparency mask changes, after invalidating the rectangle where the change occurred, you must call `DSpAltBuffer_RebuildTransparencyMask` in order to re-create the mask. This is a slow process and it is not advisable to do it at runtime.

A "dirty grid" overlays the back buffer and alternate buffers, dividing them into tiles. When you invalidate areas of a buffer, each grid tile that overlaps the dirty rectangle is marked as dirty, even if only a small part of the rectangle intersects with the grid tile. In other words, each grid tile is either all valid or all dirty. The size of the grid is determined in part by the hardware your game is running on and in part by the game itself. DrawSprocket provides functions for adjusting the size of the dirty grid: the `DSpContext_SetDirtyRectGridSize` function (page 2-53), the `DSpContext_GetDirtyRectGridSize` function (page 2-54), and `theDSpContext_GetDrityRectGridUnits` function (page 2-55).

Cleaning Up Before Quitting DrawSprocket

Listing 2-10 illustrates the typical tasks your game must perform before quitting DrawSprocket:

- Fade to black to avoid flicker when you switch contexts.
- Dispose of all alternate buffers.
- Inactivate the context and release it.
- Fade back in.

Listing 2-10 Cleaning up

```
/* fade to black */
theError = DSpContext_FadeGammaOut(NULL, NULL);

/* remove the underlay and release it */
if (inUseUnderlay)
{
    DSpContext_SetUnderlayAltBuffer(theContext, NULL);
    DSpAltBuffer_Dispose(theUnderlay);
    theUnderlay = NULL;
}

/* remove the overlay and release it */
if (inUseOverlay)
{
    DSpContext_SetOverlayAltBuffer(theContext, NULL);
    DSpAltBuffer_Dispose(theOverlay);
    theOverlay = NULL;
}

/* put the context into the inactive state */
theError = DSpContext_SetState(theContext, kDSpContextState_Inactive);

/* fade back in */
theError = DSpContext_FadeGammaIn(NULL, NULL);

/* release the context */
theError = DSpContext_Release(theContext);
```

Debugging

If you are in a debug cycle, you may want to enable debugging mode in DrawSprocket so that a fade out, followed by a break in the debugger, won't leave you with nothing to see. Listing 2-11 shows how to enable debug mode. You must call `DSpSetDebugMode` before activating the context in order for the call to take effect. For more information see the `DSpSetDebugMode` function (page 2-73).

Listing 2-11 Enabling debug mode

```
DSpSetDebugMode(true);
```

You can also cause DrawSprocket to enter debug mode by creating a folder in the same folder as your game and naming it "DSpSetDebugMode". This method is handy if you don't want to rebuild your game with the call just to debug it.

DrawSprocket Reference

This section describes the constants, data types, functions, and callback-function support provided by DrawSprocket.

Constants

This section describes the constants provided by DrawSprocket.

Depth Masks

You provide a depth mask in the `backBufferDepthMask` and `displayDepthMask` fields of the context attributes structure to specify the pixel depths that are acceptable to your program. You can construct the mask from the constants defined by the following enumeration:

```
enum DSpDepthMask {
    kDSpDepthMask_1                 = 1U<<0,
    kDSpDepthMask_2                 = 1U<<1,
    kDSpDepthMask_4                 = 1U<<2,
    kDSpDepthMask_8                 = 1U<<3,
    kDSpDepthMask_16                = 1U<<4,
    kDSpDepthMask_32                = 1U<<5,
    kDSpDepthMask_All               = ~0U
};

typedef enum DSpDepthMask DSpDepthMask;
```

Constant descriptions

`kDSpDepthMask_1` 1-bit pixel depth is acceptable.

`kDSpDepthMask_2` 2-bit pixel depth is acceptable.

`kDSpDepthMask_4` 4-bit pixel depth is acceptable.

`kDSpDepthMask_8` 8-bit pixel depth is acceptable.

`kDSpDepthMask_16` 16-bit pixel depth is acceptable.

`kDSpDepthMask_32` 32-bit pixel depth is acceptable.

`kDSpDepthMask_All` Any pixel depth is acceptable.

Color Needs

You can use the following constants in the `colorNeeds` field of the context attributes structure to specify your program's preferences or requirements for color display:

```
enum DSpColorNeeds {
    kDSpColorNeeds_DontCare         = 0L,
    kDSpColorNeeds_Request          = 1L,
    kDSpColorNeeds_Require          = 2L
};

typedef enum DSpColorNeeds DSpColorNeeds;
```

Constant descriptions

`kDSpColorNeeds_DontCare`
Display can be either color or grayscale.

`kDSpColorNeeds_Request`
: Color display is preferred, but not required.

`kDSpColorNeeds_Require`
: Color display is required.

Special Display Features

You can use the following constants in the `contextOptions` field of the context attributes structure to request special display features or to determine which features a specified display supports.

```
enum DSpContextOption {
    kDSpContextOption_QD3DAccel     = 1U<<0,
    kDSpContextOption_PageFlip      = 1U<<1,
    kDSpContextOption_TripleBuffer  = 1U<<2
};

typedef enum DSpContextOption DSpContextOption;
```

Constant descriptions

`kDSpContextOption_QD3DAccel`
: Use a graphics device that supports QuickDraw 3D acceleration.

`kDSpContextOption_PageFlip`
: Use page flipping (a hardware feature).

`kDSpContextOption_TripleBuffer`
: Use triple buffering.

Buffer Kind

Currently, DrawSprocket supports only one kind of buffer. You pass this constant to the `DSpContext_GetBackBuffer`, `DSpAltBuffer_InvalRect`, and `DSpAltBuffer_GetCGrafPtr` functions.

```
enum DSpBufferKind {
    kDSpBufferKind_Normal           = 0U
};

typedef enum DSpBufferKind DSpBufferKind;
```

Pixel Scaling

Use these constants with the `DSpContext_SetScale` function to indicate how to perform pixel scaling.

```
enum DSpBufferScale {
    kDSpBufferScale_1               = 0x00000001U,
    kDSpBufferScale_2               = 0x00000002U,
    kDSpBufferScale_2Interpolate    = 0x80000002U,
    kDSpBufferScale_3               = 0x00000003U,
    kDSpBufferScale_3Interpolate    = 0x80000003U,
    kDSpBufferScale_4               = 0x00000004U,
    kDSpBufferScale_4Interpolate    = 0x80000004U
};

typedef enum DSpBufferScale DSpBufferScale;
```

Constant descriptions

`kDSpBufferScale_1` No pixel scaling.

`kDSpBufferScale_2` Scale the back buffer by a factor of 2.

`kDSpBufferScale_2Interpolate`
Scale by a factor of 2 and interpolate when the image is drawn to the display.

`kDSpBufferScale_3` Scale the back buffer by a factor of 3.

`kDSpBufferScale_3Interpolate`
Scale by a factor of 3 and interpolate when the image is drawn to the display.

`kDSpBufferScale_4` Scale by a factor of 4.

`kDSpBufferScale_4Interpolate`
Scale by a factor of 4 and interpolate when the image is drawn to the display.

Play State

You set the play state of a context by calling the `DSpContext_SetState` function (page 2-43) and passing it one of the following values:

```
enum DSpContextState {
    kDSpContextState_Active         = 0L,
    kDSpContextState_Paused         = 1L,
    kDSpContextState_Inactive       = 2L
};

typedef enum DSpContextState DSpContextState;
```

Constant descriptions

`kDSpContextState_Active`
: The display is completely controlled by your program. The display is configured as specified in its context attributes structure. All system adornments, such as the menu bar, floating windows, and the desktop, are hidden (removed or covered by the blanking window). In this state you cannot make system calls to managers, such as the Window Manager and Dialog Manger, that expect the display to be in a normal state.

`kDSpContextState_Paused`
: The menu bar and other system adornments are restored, although the resolution mode is still that specified in the context attributes structure for the display. The desktop is still not visible; the blanking window covers it. In this state you can make normal system calls. Page flipping and double buffering are inactive in this state; the display page is set to page 0 if page flipping has been enabled.

  In this state it is safe for your program to call Macintosh Toolbox and system software functions. The paused state gives the user access to the process menu; if your game is suspended because the user switches to another application, you must call the `DSpProcessEvent` function (page 2-72).

`kDSpContextState_Inactive`
: The display is in exactly the state the user has specified from the Monitors control panel. The blanking window is hidden and the resolution mode specified in the context attributes structure for this display is not in effect.

  The user's configuration is restored only if there are no other currently active or paused contexts. As long as there is at least one active or paused context, all displays are

covered by the blanking window, and the resolution mode for each is that of the context, which may not be what the user has selected in the Monitors control panel.

Data Structures

This section describes the context attributes structure.

Context Attributes Structure

The context attributes structure describes a set of characteristics that apply to a given context.

You use the context attributes structure to request specific characteristics when creating a context, and you also use it to retrieve the actual characteristics of a given context. There are field descriptions for input values and output values, but the structure never contains both input and output information at the same time. The context attributes structure is defined by the `DSpContextAttributes` data type.

```
struct DSpContextAttributes {
    Fixed               frequency;
    UInt32              displayWidth;
    UInt32              displayHeight;
    UInt32              reserved1;
    UInt32              reserved2;
    UInt32              colorNeeds;
    CTabHandle          colorTable;
    OptionBits          contextOptions;
    OptionBits          backBufferDepthMask;
    OptionBits          displayDepthMask;
    UInt32              backBufferBestDepth;
    UInt32              displayBestDepth;
    UInt32              pageCount;
    Boolean             gameMustConfirmSwitch;
    UInt32              reserved3[4];
};

typedef struct DSpContextAttributes DSpContextAttributes;
typedef struct DSpContextAttributes *DSpContextAttributesPtr
```

Field descriptions

`frequency` *Input*: Ignored.
Output: The frame-refresh frequency (in Hz) specified by the current resolution mode. (This value is 0 if the actual frequency is not available.)

`displayWidth` *Input:* The requested display width (in pixels).
Output: The display width for the specified context.

`displayHeight` *Input:* The requested display height (in pixels).
Output: The display height for the specified context.

`reserved1` Reserved. Always set this field to 0.

`reserved2` Reserved. Always set this field to 0

`colorNeeds` *Input*: A value that specifies whether the display needs to be in color. Valid constants for this field are described in "Color Needs" (page 2-23).
Output: The color support provided by the current resolution mode.

`colorTable` *Input:* A handle to the color table to use with the context to which this attributes structure applies. (This field applies only to indexed devices; direct devices do not use a color table.)
Output: Ignored.

`contextOptions` *Input:* A set of bit flags that define requested special display features for which either hardware or software implementation is acceptable. Valid constants for this field are described in "Special Display Features" (page 2-24).
Output: The special display features supported in software by the current resolution mode.

`backBufferDepthMask`
Input: A bit array that defines the acceptable pixel depths for the back buffer. Valid constants for this field are described in "Depth Masks" (page 2-22).
Output: The bit depth DrawSprocket recommends for the context.

`displayDepthMask`
Input: A bit array that defines the acceptable pixel depths for the front buffer. Valid constants for this field are described in "Depth Masks" (page 2-22).
Output: A bit array that specifies the pixel depth of the specified context.

`backBufferBestDepth`
Input: The preferred pixel depth, or video mode, for the back buffer.
Output: The bit depth DrawSprocket recommends for the context.

`displayBestDepth` *Input:* The preferred pixel depth for the display.
Output: The pixel depth of the specified context.

`pageCount` *Input*: Ignored.
Output: Gives the number of hardware video pages available. A value of 1 indicates that hardware page flipping is not supported.

`gameMustConfirmSwitch`
Input: Ignored.
Output: A value of `true` indicates that the context may have problems being displayed on the user's system, and the game should confirm that the context is visible after being set to the active state by asking the user if the display can be seen (via a dialog box or some other mechanism). Additionally, a warning code will be returned from `DSpContext_SetState` indicating that the game should confirm that the context is visible.

`reserved3[4]` Reserved. Always set this field to 0.

DrawSprocket Functions

This section describes the DrawSprocket functions that your game can call.

Using DrawSprocket

You use the functions in this section before using DrawSprocket and when you are finished.

DSpStartup

You call the `DSpStartup` function before using any other DrawSprocket functions.

```
OSStatus DSpStartup (void);
```

function result A result code.

DESCRIPTION

The game needs to call the `DSpStartup` function before using any DrawSprocket functions.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpShutdown

You use the `DSpShutdown` function before quitting the game.

```
OSStatus DSpShutdown  (void);
```

function result A result code.

DESCRIPTION

The game must call the `DSpShutdown` function when it is through using DrawSprocket functions.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

Choosing a Context and Saving Preferences

The functions described in this section allow you to determine the display characteristics and features of a given system, help you choose the configuration that best fits your game's needs, and allow you to save a context.

DSpFindBestContext

You can use the `DSpFindBestContext` function to find the context that best matches a context you describe.

```
OSStatus DSpFindBestContext (
                    const DSpContextAttributesPtr inDesiredAttributes,
                    DSpContextReference *outContext);
```

`inDesiredAttributes`
A pointer to a context attributes structure. You should specify as best as possible the details of a context you would like to use.

`outContext` On exit, a reference to the context that best meets or exceeds the specified attribute requirements.

function result A result code.

DESCRIPTION

The function `DSpFindBestContext` takes, in the `inDesiredAttributes` parameter, a pointer to a context attributes structure describing display characteristics, such as display height and width, preferred pixel depth, and color capability, that your program needs. It returns, in the `outContext` parameter, a reference to the context that best matches the description. All display contexts are considered that meet or exceed the requirements in the attributes. If no such contexts exist, then an error is returned.

The game should check the attributes of the chosen context by calling `DSpContext_GetAttributes`. It is possible that the game may want to use attributes of the context that exceed those asked for. For example, the game may request a mode such as 320x200x8 but the best match is a 640x480x8 display; the game can adapt to a full screen mode once it is aware of the situation.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpGetFirstContext

You can use the `DSpGetFirstContext` function to get the first context in the list of contexts available for a specified display. This call is provided so that you may iterate over the list and choose one that best suits your needs, should you decide not to let DrawSprocket find one for you with `DSpFindBestContext` or let the user select one by calling `DSpUserSelectContext`.

```
OSStatus DSpGetFirstContext (
                    DisplayIDType displayID,
                    DSpContextReference *outContext);
```

`displayID` The ID of the display for which you want to get the context. You can obtain the ID by using the Display Manager.

`outContext` On exit, a reference to the first context in the list of available contexts for the specified display.

function result A result code.

DESCRIPTION

The `DSpGetFirstContext` function takes, in the `displayID` parameter, the ID of a display and returns, in the `outContext` parameter, a reference to its first context. You cannot use this context with any function other than `DSpContext_GetAttributes`, `DSpContext_GetFlattendSize`, `DSpContext_Flatten`, and `DSpContext_GetDisplayID` until you reserve it with `DSpContext_Reserve`.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpGetNextContext

You can use the `DSpGetNextContext` function to get the next context in a list of available contexts for a display. Use this function in conjunction with the `DSpGetFirstContext` function, which gets the first context in the list for a specified display.

```
OSStatus DSpGetNextContext (
                    DSpContextReference inCurrentContext,
                    DSpContextReference *outContext);
```

`inCurrentContext` The context reference that was just returned by `DSpGetFirstContext` or `DSpGetNextContext`.

`outContext` On exit, a reference to the next context in the list of available contexts.

function result A result code.

DESCRIPTION

The `DSpGetNextContext` function takes, in the `inCurrentContext` parameter, a reference to a context in a list of contexts that are available for a display and returns, in the `outContext` parameter, a reference to the next context in the list. If `inCurrentContext` contains the last context in the list, `DSpGetNextContext` returns an error.

A sample use would be:

```
DSpContextReference theContext;

theError = DSpGetFirstContext(theDisplayID, &theContext);
/* process the error */
while (theContext)
{
    /* process the context */

    /* get the next context */
    theError = DSpGetNextContext(theContext, &theContext);
    /* process the error */
}
```

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_GetAttributes

You can use the `DSpContext_GetAttributes` function to get the attributes of a context as if it were in the active state.

```
OSStatus DSpContext_GetAttributes (
                    DSpContextReference inContext,
                    DSpContextAttributesPtr outAttributes);
```

`inContext` The context whose attributes you want to get.

`outAttributes`
On exit, a pointer to an attributes structure describing the context.

function result A result code.

DESCRIPTION

The `DSpContext_GetAttributes` function takes, in the `inContext` parameter, a reference to a context and returns, in the `outAttributes` parameter, a pointer to an attributes structure describing that context. The context is described as if it were in the active play state. The monitor frequency may not be known until a context is actually in the active play state, so it may be returned as zero.

You can use this function to confirm that the context returned from `DSpFindBestContext` has the characteristics you need. You may even adjust your drawing plans based on the results. For example, you might have requested a resolution mode such as 320x200x8 when calling `DSpFindBestContext`, but then learned from calling `DSpContext_GetAttributes` that the context is a 640x480x8 display. In such a case, you might still use the 640x480x8 display, but display a larger game image.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpCanUserSelectContext

You can use the `DSpCanUserSelectContext` function to determine whether there is a meaningful choice of contexts to present to the user with the `DSpUserSelectContext` function.

```
OSStatus DSpCanUserSelectContext  (
                    DSpContextAttributesPtr inDesiredAttributes
                    Boolean *outUserCanSelectContext );
```

`DSpContextAttributesPtr`
A pointer to a context attributes structure that specifies the required attributes.

`outUserCanSelectContext`
On exit, the value is `true` if there are multiple contexts that meet the specified attribute requirements; `false` if there are not.

function result A result code.

DESCRIPTION

The `DSpCanUserSelectContext` function takes a pointer to a context attributes structure in the `inDesiredAttributes` parameter and returns `true` in the `outUserCanSelectContext` parameter if there is more than one context that has the specified attributes. This function allows you to check before calling the `DSpUserSelectContext` function so as to avoid presenting the user with a selection dialog box when there is no choice of displays.

DSpUserSelectContext

You can use the `DSpUserSelectContext` function to present a dialog box that allows the user to select a display.

```
OSStatus DSpUserSelectContext (
                    DSpContextAttributesPtr inDesiredAttributes,
                    DisplayIDType inDialogDisplayLocation,
                    DSpEventProcPtr inEventProc,
                    DSpContextReference *outContext );
```

`inDesiredAttributes`
A pointer to an attributes structure that specifies a minimum set of required display characteristics.

`inDialogDisplayLocation`
The display ID of the display on which to present the selection dialog box. If this parameter is 0, DrawSprocket positions the dialog box automatically on the main screen.

`inEventProc` A pointer to an application-defined event processing routine that allows you to handle events received by the dialog box that DrawSprocket cannot process, such as update events, in your game context area.

`outContext` On exit, a reference to a context.

function result A result code.

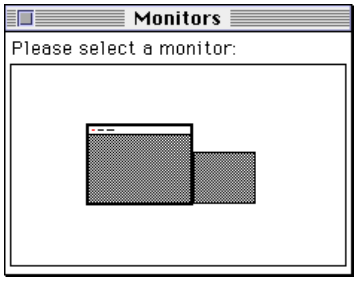
DESCRIPTION

The `DSpUserSelectContext` function presents a dialog box the user can select a context from. In the selection dialog box (Figure 2-1), all graphics devices appear, although the user can select only those contexts that meet or exceed the minimum characteristics given in the `inDesiredAttributes` parameter. The function returns, in the `outContext` parameter, a reference to the context the user selects.

You can specify which display to present the dialog box on in the `inDialogDisplayLocation` parameter; if this parameter is 0, DrawSprocket positions the dialog box automatically on the main screen.

The game must provide an event processing function to handle events that are generated when the user moves the dialog box around on the screen. The `inEventProc` parameter contains a pointer to this function. See the `MyEventHandler` function (page 2-76).

Figure 2-1 A context-selection dialog box



CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_Restore

You can use the `DSpContext_Restore` function to restore a context that was saved previously, most likely to preserve a user's preferences.

```
OSStatus DSpContext_Restore (
                    void *inFlatContext,
                    DSpContextReference *outRestoredContext);
```

`inFlatContext`
A pointer to the flattened context.

`DSpContextReference`
On exit, a reference to the restored context.

function result A result code.

DESCRIPTION

The `DSpContext_Restore` function takes, in the `inFlatContext` parameter, a pointer to a flattened context. Typically, the flat context would have been saved out to disk and reloaded on a later execution of the game before calling this function. When you call this function, DrawSprocket attempts to find a context

that matches the flattened context in the current system and returns, in the `DSpContextReference` parameter, a reference to the context it finds.

If it can't find a match, the user probably has reconfigured the displays since the last time your game was run, and `DSpContext_Restore` returns an error. This function has a high probability of failing, so your game should not rely on being able to restore the context. However, the game should attempt to do so as part of the normal saving of the user preferences.

If you save a context, flatten it before you first make the context's play state active; otherwise, the saved data will not contain the proper information with which to locate the display.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_GetFlattenedSize

Use the `DSpContext_GetFlattenedSize` function to find out how much memory is required to store a flattened version of a context.

```
OSStatus DSpContext_GetFlattenedSize (
                    DSpContextReference inContext,
                    UInt32 *outFlatContextSize);
```

`inContext` A reference to the context you intend to flatten.

`outFlatContextSize`
On exit, the size of the buffer required to store a flattened version of the context.

function result A result code.

DESCRIPTION

You pass the `DSpContext_GetFlattenedSize` function a reference to a context in the `inContext` parameter and it returns, in the `outFlatContextSize` parameter, the size of the buffer required to store a flattened version of the context. You can then allocate a buffer of this size and pass it to `DSpContext_Flatten`.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_Flatten

You can use the `DSpContext_Flatten` function to convert a context into a format suitable for saving to disk—for example, to save user preferences.

```
OSStatus DSpContext_Flatten (
                    DSpContextReference inContext,
                    void *outFlatContext);
```

`inContext` A reference to the context to be flattened.

`outFlatContext`
On exit, the flattened context.

function result A result code.

DESCRIPTION

The `DSpContext_Flatten` function takes, in the `inContext` parameter, a reference to a context to be converted into a format that can be saved to disk. You also pass, in the `outFlatContext` parameter, a pointer to a buffer to hold the flattened context. The buffer must be large enough to hold the flattened context. You can find out the correct size by calling `DSpContext_GetFlattenedSize`. The function returns, in the `outFlatContext` parameter, the flattened context.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_GetDisplayID

Use the `DSpContext_GetDisplayID` function to get the ID of the display a context is associated with.

```
OSStatus DSpContext_GetDisplayID (
                    DSpContextReference inContext,
                    DisplayIDType *outDisplayID);
```

`inContext` A reference to a context.

`outDisplayID` On exit, a display monitor ID for the device the context is associated with.

function result A result code.

DESCRIPTION

You pass the `DSpContext_GetDisplayID` function a reference to a context in the `inContext` parameter and it returns the ID of the display the context is associated with in the `outDisplayID` parameter.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

Manipulating a Context

The functions described in this section allow you to reserve a context, to set the play state for a context, and to set the color of the blanking window.

DSpContext_Reserve

You can use the `DSpContext_Reserve` function to reserve a context so that you can begin using it in your game.

```
OSStatus DSpContext_Reserve (
                    DSpContextReference inContext,
                    const DSpContextAttributesPtr inDesiredAttributes);
```

`inContext` A reference to the context to reserve.

`DSpContextAttributesPtr`
A pointer to an attributes structure that specifies the configuration you would like for the display when it is in the active or paused state.

function result A result code.

DESCRIPTION

You pass the function a reference to the context to reserve in the `inContext` parameter. When the context is reserved, it is in the inactive state. There will be no visible indication that the context has been reserved at this point. To enable your context, call `DSpContext_SetState` (page 2-43). The context will show up on the display once the context has been placed in the active state.

The attributes structure you can specify in the `DSpContextAttributesPtr` parameter provides the configuration information for the display when it is in the active or paused states. If you would like to override the attributes of the context, you may do so in the attributes structure. For example, if you ask for a 320x240x16 display but the closest match is a context that is 640x480x32, passing in your requested attributes when you reserve the context will cause the `DSpContext_GetBackBuffer` function to return a graphics pointer that refers to a 320x240x16 drawing environment.

Along the same lines, you should turn off features that you are not interested in when you reserve the context. For example, if the context supports page flipping (and you know this because you requested the actual capabilities of the context using `DSpContext_GetAttributes`), you can turn off the page-flipping bit in your desired attributes so that you will be assured of using software buffering.

You should only specify a back buffer bit depth different from the display bit depth when you absolutely must, as it is the worst case scenario for DrawSprocket and will result in a synchronous call to `CopyBits` to bring your back buffer to the display.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_Release

You can use the `DSpContext_Release` function to release a context you are finished using.

```
OSStatus DSpContext_Release  (DSpContextReference inContext);
```

`inContext` A reference to the context to be released.

function result A result code.

DESCRIPTION

When you are finished using a context, you must release it with the `DSpContext_Release` function. You pass the function a reference to the context to be released in the `inContext` parameter. Releasing the context does not necessarily remove the blanking window from its graphics device's display. All displays remain covered by the blanking window until all contexts have been released or put in an inactive play state.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_SetState

You can use the `DSpContext_SetState` function to set the play state of a context.

```
OSStatus DSpContext_SetState (
                    DSpContextReference inContext,
                    DSpContextState inState);
```

`inContext` A reference to the context whose play state you want to set.

`inState` The state to set the context to. Valid input values for this parameter are `kDSpContextState_Active`, `kDSpContextState_Paused`, and `kDSpContextState_Inactive`.

function result A result code.

DESCRIPTION

The `DSpContext_SetState` function sets the play state of the context specified in the `inContext` parameter to the state specified in the `inState` parameter. See "Play State" (page 2-25) for more information on valid values for the `inPlayState` parameter. In summary, you can make these choices:

- A context's initial play state is inactive. When all contexts for a display are set to `kDSpContextState_Inactive`, the display looks exactly as it does when the user is using their Macintosh normally: the monitor resolutions are set to the default, the menu bar is available, and so on.
- Set the play state to `kDSpContextState_Active` to use the display. In this state, the attributes of the context are used to change the display resolution, remove the menu bar, and so on. When at least one context is active, all the display devices in the system are covered by a blanking window. When a context is in the active state, the display is completely owned by the game.
- Set the play state to `kDSpContextState_Paused` to temporarily restore system adornments, while maintaining the attributes used by the context. This gives the user the opportunity to use the menus and switch to other applications. While the context is in the paused state, it is very important to call `DSpProcessEvent` to allow DrawSprocket to correctly handle events such as suspend or resume (see the `DSpProcessEvent` function on (page 2-72). Page flipping and double buffering are inactive in this state, and the context will be placed back at page 0 if page flipping was being used.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_GetState

Use the `DSpContext_GetState` function to find out the current play state of a context.

```
OSStatus DSpContext_GetState (
                    DSpContextReference inContext,
                    DSpContextState *outState);
```

`inContext` A reference to the context whose play state you want to get.

`outState` On exit, the play state of the context. Valid return values are `kDSpContextState_Active`, `kDSpContextState_Paused`, and `kDSpContextState_Inactive`.

function result A result code.

DESCRIPTION

The function takes, in the `inContext` parameter, a reference to a context and returns, in the `outState` parameter, the current play state of the context. See "Play State" (page 2-25) for information on interpreting the value returned in the `outState` parameter.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpSetBlankingColor

You can use the `SetBlankingColor` function to assign a background color to the blanking window for all displays.

```
OSStatus DSpSetBlankingColor  (const RGBColor *inRGBColor);
```

`inRGBColor` A pointer to the background color to use for the blanking window.

DESCRIPTION

The `DSpSetBlankingColor` function sets the blanking color to the color specified in the `inRGBColor` parameter. The blanking color replaces the desktop and system adornments, such as the menu bar, for all display devices as long as any context is active.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

Drawing and Double Buffering

The functions described in this section allow you to draw to the display, control various aspects of display visibility and frame speed, and to scale the back buffer.

DSpContext_FadeGamma

You can use the `DSpContext_FadeGamma` function to manually fade the specified context either in or out to a color of your choice.

```
OSStatus DSpContext_FadeGamma (
                    DSpContextReference inContext,
                    SInt32 inPercentOfOriginalIntensity,
                    RGBColor *inZeroIntensityColor);
```

`inContext` A reference to the context whose display is to be faded. If you pass `NULL` for this parameter, the fade operation applies simultaneously to all displays.

`inPercentOfOriginalIntensity`
The amount of fading to apply, expressed as the resultant percentage (0–100) of the display's full intensity that you want

to achieve with this call. Values above 100 percent begin to converge on white. If you have specified an intensity color, values less than zero begin to converge on black.

`inZeroIntensityColor`
A pointer to the color that is to correspond to zero intensity (represented by a value of 0 in the `inPercentOfOriginalIntensity` parameter). If you pass `NULL` for this parameter, the zero-intensity color is black.

function result A result code.

DESCRIPTION

The `DSpContext_FadeGamma` function fades the context specified in the `inContext` parameter by the percentage specified in the `inPercentOfOriginalIntensity` parameter. Fading the context is an aesthetically pleasing way to transition into and out of your game and between different sections of it. When performing a resolution-mode switch (as when activating and deactivating your context's play state), it is important to fade the display to hide the flash that occurs.

`DSpContext_FadeGamma` performs a gamma fade, which gives better results than a simple indexed fade. See "Gamma Fading" (page 2-9) for more information on this and other advantages of DrawSprocket's support for fading.

Fading is normally an incremental process. Over a period of time, you make repeated, timed calls to `DSpContext_FadeGamma`, each time passing it an incrementally different value for the `inPercentOfOriginalIntensity` parameter, until the final desired intensity is achieved. The intensity value you pass is usually an integer between 0 and 100. It can be greater than 100, if you want to use fading to create a high-intensity burst of light, or less than 100 if you have specified a zero-intensity color and want to fade the color toward black.

The zero-intensity value that you fade out to is by default black, but it can be any color that you specify in the `inZeroIntensityColor` parameter. You can achieve special effects by fading partially toward one zero-intensity color and then completing the fade to a different one. At the point when you actually switch resolution modes, the zero-intensity color must be black and your display must be completely faded if there is to be no visible flash.

To automatically accomplish a smooth fade all the way from full intensity to zero intensity, or vice versa, in a single operation use the `DSpContext_FadeGammaIn` function (page 2-48) and the `DSpContext_FadeGammaOut` (page 2-47) function.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_FadeGammaOut

You can use the `DSpContext_FadeGammaOut` function to completely fade out a display to a color of your choice.

```
OSStatus DSpContext_FadeGammaOut (
                    DSpContextReference inContext,
                    RGBColor *inZeroIntensityColor);
```

`inContext` A reference to the context whose display is to be faded. If you pass `NULL` for this parameter, the fade operation applies simultaneously to all displays.

`inZeroIntensityColor`
A pointer to the color that is to correspond to zero intensity. If you pass `NULL` for this parameter, the zero-intensity color is black.

function result A result code.

DESCRIPTION

The `DSpContext_FadeGammaOut` function automatically performs a gamma fade on the display of the context specified in the `inContext` parameter. It fades the display from 100 percent to 0 percent intensity over a period of one second. You specify the zero-intensity color in the `inZeroIntensityColor` parameter. A key press or a mouse button click will jump the fade to its end point immediately.

You can perform a manual fade with the `DSpContext_FadeGamma` function (page 2-45). For a complete discussion of gamma fades see "Gamma Fading" (page 2-9).

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_FadeGammaIn

You can use the `DSpContext_FadeGammaIn` function to completely fade in a display to a color of your choice.

```
OSStatus DSpContext_FadeGammaIn (
                    DSpContextReference inContext,
                    RGBColor *inZeroIntensityColor);
```

`inContext` A reference to the context whose display is to be faded. If you pass `NULL` for this parameter, the fade operation applies simultaneously to all displays.

`inZeroIntensityColor`
The color that is to correspond to zero intensity. If you pass `NULL` for this parameter, the zero-intensity color is black.

function result A result code.

DESCRIPTION

The `DSpContext_FadeGammaIn` function automatically performs a gamma fade on the display of the context specified in the `inContext` parameter. It fades the display from 0 percent to 100 percent intensity over a period of one second. You specify the zero-intensity color in the `inZeroIntensityColor` parameter. A key press or a mouse-button click will jump the fade to its end point immediately.

You can perform a manual fade with the `DSpContext_FadeGamma` function (page 2-45). For a complete discussion of gamma fades see "Gamma Fading" (page 2-9).

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_GetBackBuffer

The `DSpContext_GetBackBuffer` function returns the back buffer for the context, which is where the game should image to. The back buffer is the next buffer that will be displayed on a call to `DSpContext_SwapBuffers`.

```
OSStatus DSpContext_GetBackBuffer (
                    DSpContextReference inContext,
                    DSpBufferKind inBufferKind,
                    CGrafPtr *outBackBuffer);
```

`inContext` A reference to the context whose back buffer is to be returned.

`inBufferKind` The kind of buffer. Currently the only supported buffer kind is `kDSpBufferKind_Normal`.

`outBackBuffer`
On exit, a pointer to the back buffer.

function result A result code.

DESCRIPTION

The `DSpContext_GetBackBuffer` function returns, in the `outBackBuffer` parameter, a graphics pointer designating a graphics port containing the back buffer of the context specified in the `inContext` parameter. Your game should draw to that back buffer. You specify the kind of buffer in the `inBufferKind` parameter, although currently the only supported buffer kind is `kDSpBufferKind_Normal`.

The pointer to the back buffer may change after a call to `DSpContext_SwapBuffers`, so you must call this function before rendering every frame.

If you have specified an underlay for the context, the back buffer will have the underlay image restored before this call returns.

If there are no available back buffers (they are all queued up for display), this function will block until one is available. To avoid blocking, call `DSpContext_IsBusy` (page 2-52) until it returns `false`.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_InvalBackBufferRect

You can use the `DSpContext_InvalBackBufferRect` function to invalidate a specific area of a context's back buffer, so that only a portion of the screen needs to be redrawn when the buffers are next swapped.

```
OSStatus DSpContext_InvalBackBufferRect (
                    DSpContextReference inContext,
                    const Rect *inRect);
```

`inContext` A reference to the context whose back buffer is to be invalidated.

`inRect` A pointer to a rectangle specifying the area (in back-buffer coordinates) to invalidate.

function result A result code.

DESCRIPTION

The `DSpContext_InvalBackBufferRect` function invalidates the area of the back buffer specified in the `inRect` parameter for the context specified in the `inContext` parameter. If you do not call this function between buffer swaps, the entire back buffer is considered invalid when a swap occurs. The invalid rectangles must be set prior to each call to `DSpContext_SwapBuffers`; the dirty rectangle list is emptied before `DSpContext_GetBackBuffer` returns the back buffer for re-use.

You can make multiple calls to this function between swaps to accumulate invalid rectangular areas. For information on the importance of using dirty rectangles, see "Improving Performance With Dirty Rectangles" (page 2-20).

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_SwapBuffers

You can use the `DSpContext_SwapBuffers` function to draw a context's back buffer to the screen.

```
OSStatus DSpContext_SwapBuffers(DSpContextReference inContext,
                    DSpCallbackProcPtr inBusyProc,
                    void *inUserRefCon);
```

`inContext`	A reference to the context whose buffers are to be swapped.
`inBusyProc`	A pointer to an application-supplied callback function that performs any required pre-swap tasks.
`inUserRefCon`	A reference constant to be handed back by DrawSprocket when it calls the callback specified by the `inBusyProc` parameter.
function result	A result code.

DESCRIPTION

Calling the `DSpContext_SwapBuffers` function causes the invalid parts of the back buffer of the context specified in the `inContext` parameter (or the entire back buffer, if its invalid-rectangle list is empty) to be drawn to the screen.

This function returns immediately, even if the buffer swap has not yet occurred. To determine when the next call to `DSpContext_GetBackBuffer` will not block, you can repeatedly call the `DSpContext_IsBusy` function (page 2-52) until it returns a value of `false`.

Before performing the buffer swap, DrawSprocket repeatedly calls an application-supplied callback function, pointed to by the `inBusyProc` parameter, to make sure that any constraints you impose are satisfied before the swap occurs. For example, your callback might check to ensure that any QuickDraw 3D acceleration hardware has completed its rendering. When DrawSprocket calls the callback routine, it passes the callback the reference constant you passed to `DspContext_SwapBuffers` in the `refCon` parameter.

The buffer swap does not occur until the callback function returns `false`. The callback pointer must be of this type:

```
typedef Boolean (*DSpCallbackProcPtr)(DSpContextReference inContext,
                    void *inRefCon);
```

See the `MyCallbackFunction` function (page 2-75) for a description of the callback function.

In a worst case scenario where the back buffer and the display have different bit depths, `DSpContext_SwapBuffers` immediately calls `CopyBits` to transfer the data. To avoid this, and to use the optimized DrawSprocket blitters, always insure that your back buffer and display bit depths are identical.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_IsBusy

You can use the `DSpContext_IsBusy` function to find out whether a back buffer is available.

```
OSStatus DSpContext_IsBusy (
                    DSpContextReference inContext,
                    Boolean *outBusyFlag);
```

`inContext`	A reference to the context for which you want to determine if a back buffer is available.
`outBusyFlag`	On exit, contains `true` if no back buffer is available, `false` if a back buffer is available.
function result	A result code.

DESCRIPTION

The `DSpContext_IsBusy` function returns `true` in the `outBusyFlag` flag if there are no back buffers available for the context specified in the `inContext` parameter. This is useful if you would like to know if a call to `DSpContext_GetBackBuffer` will block.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_SetDirtyRectGridSize

You can use the `DSpContext_SetDirtyRectGridSize` function to suggest a grid cell size for the context's dirty rectangles.

```
OSStatus DSpContext_SetDirtyRectGridSize (
                    DSpContextReference inContext,
                    UInt32 inCellPixelWidth,
                    UInt32 inCellPixelHeight);
```

`inContext` A reference to a context for which you want to set the size of the dirty rectangle grid units.

`inCellPixelWidth`
The width of the grid unit in pixels.

`inCellPixelHeight`
The height of the grid unit in pixels.

function result A result code.

DESCRIPTION

The `DSpContext_SetDirtyRectGridSize` function takes a reference to a context in the `inContext` parameter and sets the dirty rectangle grid cell size for that context as closely as possible to the dimensions passed in the `inCellPixelWidth` and `inCellPixelHeight` parameters. The size used depends on factors such as the L1 cache size and the CPU bus width, so your suggested values may not be the actual values used, but DrawSprocket will attempt to match your suggested size as closely as possible.

To find out what size grid cells DrawSprocket is actually using, call `DSpContext_GetDirtyRectGridSize` (page 2-54). To find out the base size that all grid units must be a multiple of, use the `DSpContext_GetDirtyRectGridUnits` function ((page 2-55)).

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_GetDirtyRectGridSize

You can use the `DSpContext_GetDirtyRectGridSize` function to find out the current grid cell size for a context's dirty rectangles.

```
OSStatus DSpContext_GetDirtyRectGridSize (
                    DSpContextReference inContext,
                    UInt32 *outCellPixelWidth,
                    UInt32 *outCellPixelHeight);
```

`inContext` A reference to a context for which you want to know the current grid cell size of the dirty rectangles.

`outCellPixelWidth`
On exit, the width of the grid cell in pixels.

`outCellPixelHeight`
On exit, the height of the grid cell in pixels.

function result A result code.

DESCRIPTION

The `DSpContext_GetDirtyRectGridSize` function takes, in the `inContext` parameter, a reference to a context and returns, in the `outCellPixelWidth` and `outCellPixelHeight` parameters, the current width and height of a dirty rectangle grid cell in pixels. The height and width values may be different from the values specified in `DSpContext_SetDirtyRectGridSize` because the grid cells must be multiples of the base grid cell. To find out the dimensions of the base grid cell, you can use the `DSpContext_GetDirtyRectGridUnits` function.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_GetDirtyRectGridUnits

Use the `DSpContext_GetDirtyRectGridUnits` function to find out the size of the base dirty rectangle grid cell for a context.

```
OSStatus DSpContext_GetDirtyRectGridUnits (
                    DSpContextReference inContext,
                    UInt32 *outCellPixelWidth,
                    UInt32 *outCellPixelHeight);
```

`inContext` A reference to a context for which you want to know the base size of the dirty rectangle grid cells.

`outCellPixelWidth`
On exit, the width of the grid cell in pixels.

`outCellPixelHeight`
On exit, the height of the grid cell in pixels.

function result A result code.

DESCRIPTION

The `DSpContext_GetDirtyRectGridUnits` function takes, in the `inContext` parameter, a reference to a context and returns, in the `outCellPixelWidth` and `outCellPixelHeight` parameters, the width and height of a base grid cell in pixels. The grid cell size is based on a number of machine characteristics such as the bus width and L1 cache size. When you specify a grid cell size with the `DSpContext_SetDirtyRectGridSize` function, DrawSprocket rounds the requested size to a multiple of the base grid cell size.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_SetMaxFrameRate

Use the `DSpContext_SetMaxFrameRate` function to set a maximum frame rate for your game. This call does not guarantee that your game will achieve the

maximum rate, but if it attempts to exceed the rate, DrawSprocket will slow down the buffer swapping.

```
OSStatus DSpContext_SetMaxFrameRate (
                    DSpContextReference inContext,
                    UInt32 inMaxFPS);
```

`inContext` A reference to the context whose maximum frame rate you want to set.

`inMaxFPS` The maximum frame rate in frames per second.

function result A result code.

DESCRIPTION

The `DSpContext_SetMaxFrameRate` function takes, in the `inContext` parameter, a reference to a context and sets its maximum frame rate to the value you pass in the `inMaxFPS` parameter. The actual frame rate that is set is not necessarily the frame rate you specified, because DrawSprocket internally converts the specified maximum frame rate into a value that can be used to skip a number of frames for each frame that is drawn.

For example, if the monitor refresh rate is 66.7 Hz, and you request a frame rate of 30 fps DrawSprocket internally skips every other frame, and your resulting frame rate is about 33.3 Hz.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_GetMaxFrameRate

Use the `DSpContext_GetMaxFrameRate` function to find out what a context's maximum frame rate is set to.

```
OSStatus DSpContext_GetMaxFrameRate (
                    DSpContextReference inContext,
                    UInt32 *outMaxFPS);
```

`inContext`	A reference to the context whose maximum frame rate you want to get.
`outMaxFPS`	On exit, the maximum frame rate in frames per second.
function result	A result code.

DESCRIPTION

The `DSpContext_GetMaxFrameRate` function returns, in the `outMaxFPS` parameter, the maximum frame rate for the context specified in the `inContext` parameter. The frame rate returned is not necessarily the same as the maximum frame rate passed by the most recent call to the `DSpContext_SetMaxFrameRate` function. If 0 is returned as the maximum frame rate, there are no frame rate restrictions in place.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_GetMonitorFrequency

You can use the `DSpContext_GetMonitorFrequency` function to get the frequency for the display associated with a context.

```
OSStatus DSpContext_GetMonitorFrequency (
                    DSpContextReference inContext,
                    Fixed *outFrequency);
```

`inContext`	A reference to a context for which you want to get the display frequency.
`outFrequency`	On exit, the display frequency.
function result	A result code.

DESCRIPTION

The `DSpContext_GetMonitorFrequency` function takes a reference to a context in the `inContext` parameter and returns the frequency of the display associated

with that context in the `outFrequency` parameter. The context must have been active for a reasonable amount of time (at least two seconds) in order to receive a correct value, because the value returned is calculated by timing the frame rate of the context in its active state.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_SetScale

Use the `DSpContext_SetScale` function to scale the back buffer for a context.

```
OSStatus DSpContext_SetScale (
                    DSpContextReference inContext,
                    DSpBufferScale inScale);
```

`inContext` A reference to a context whose back buffer you want to scale.

`inScale` The amount by which to scale the back buffer. See "Pixel Scaling" (page 2-25) for a description of the available settings.

function result A result code.

DESCRIPTION

You pass the function a reference to a context in the `inContext` parameter and, in the `inScale` parameter, the amount by which to scale the context's back buffer. When scaling is activated, the resolution of the context remains the same but the area of the back buffer that is used is reduced. That area is then expanded vertically and horizontally to fill the entire display.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_GetScale

Use the `DSpContext_GetScale` function to return the current pixel scaling setting for the context.

```
OSStatus DSpContext_GetScale (
                    DSpContextReference inContext,
                    DSpBufferScale *outScale);
```

`inContext` A reference to the context whose pixel scaling setting you want.

`outScale` On exit, the pixel scaling setting. See "Pixel Scaling" (page 2-25) for a description of the returned value.

function result A result code.

DESCRIPTION

You pass the function a reference to a context in the `inContext` parameter and it returns, in the `outScale` parameter, the pixel scaling setting for that context.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

Using Alternate Buffers

Use the functions in this section to create and draw into an alternate buffer and to designate an alternate buffer to serve as an underlay or overlay.

DSpAltBuffer_New

You can use the `DSpAltBuffer_New` function to create an alternate buffer for an underlay or overlay.

```
OSStatus DSpAltBuffer_New (
                    DSpContextReference inContext,
                    Boolean inVRAMBuffer,
                    DSpAltBufferReference *outAltBuffer);
```

`inContext` A reference to the context to create an alternate buffer for.

`inVRAMBuffer` A value of `true` requests that DrawSprocket create the buffer in VRAM if possible (it may be created in the current heap). A value of `false` means to create the buffer in the current heap.

`outAltBuffer` On exit, an alternate buffer reference.

function result A result code.

DESCRIPTION

This function creates an alternate buffer for the context specified in `inContext` and returns a reference to the buffer in the `outAltBuffer` parameter. The alternate buffer will have the same characteristics as the specified context. To request that DrawSprocket create the buffer in VRAM, set the `inVRAMBuffer` parameter to `true`. The alternate buffer may be created in VRAM or in the current heap. If the `inVRAMBuffer` parameter is `false`, DrawSprocket creates the alternate buffer in the current heap.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpAltBuffer_Dispose

Use the `DSpAltBuffer_Dispose` function to dispose of an alternate buffer.

```
OSStatus DSpAltBuffer_Dispose  (DSpAltBufferReference inAltBuffer);
```

`inAltBuffer` A reference to the buffer to be disposed of.

function result A result code.

DESCRIPTION

The `DSpAltBuffer_Dispose` function takes a reference to an alternate buffer in the `inAltBuffer` parameter and disposes of that buffer.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpAltBuffer_GetCGrafPtr

Use the `DSpAltBuffer_GetCGrafPtr` function to get the drawing area for an alternate buffer.

```
OSStatus DSpAltBuffer_GetCGrafPtr (
                    DSpAltBufferReference inAltBuffer,
                    DSpBufferKind inBufferKind,
                    CGrafPtr *outCGrafPtr);
```

`inAltBuffer` A reference to an alternate buffer.

`inBufferKind` The kind of buffer. Currently the only supported buffer kind is `kDSpBufferKind_Normal`.

`outCGrafPtr` On exit, the graphics pointer associated with an alternate buffer.

function result A result code.

DESCRIPTION

You pass the `DSpAltBuffer_GetCGrafPtr` function a reference to an alternate buffer in the `inAltBuffer` parameter and the kind of buffer in the `inBufferKind` parameter. The function returns a graphics pointer in the `outCGrafPtr` parameter. The pointer may then be used to image into the alternate buffer. After you have imaged into the alternate buffer, remember to invalidate the

rectangles that you have worked in using the `DSpAltBuffer_InvalRect` function (page 2-65).

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_SetOverlayAltBuffer

Use the `DSpContext_SetOverlayAltBuffer` function to designate an alternate buffer to be used as the current overlay buffer for a context.

```
OSStatus DSpContext_SetOverlayAltBuffer (
                    DSpContextReference inContext,
                    DSpAltBufferReference inNewOverlay);
```

`inContext` A reference to the context the overlay is to be used with.

`inNewOverlay` A reference to the alternate buffer that holds the overlay.

function result A result code.

DESCRIPTION

The `DSpContext_SetOverlayAltBuffer` function makes the alternate buffer specified in the `inNewOverlay` parameter serve as the current overlay for the context specified in the `inContext` parameter. You can have multiple overlay buffers—for example, if you have alternate cockpit views—and change between them with this call.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_GetOverlayAltBuffer

You can use the `DSpContext_GetOverlayAltBuffer` function to get the current overlay associated with a context.

```
OSStatus DSpContext_GetOverlayAltBuffer (
                    DSpContextReference inContext,
                    DSpAltBufferReference *outOverlay);
```

`inContext` A reference to the context whose overlay you want to get.

`outOverlay` On exit, a reference to the alternate buffer that holds the overlay.

function result A result code.

DESCRIPTION

The `DSpContext_GetOverlayAltBuffer` function takes, in the `inContext` parameter, a reference to a context and returns, in the `outOverlay` parameter, a reference to the alternate buffer that holds the current overlay for that context.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_SetUnderlayAltBuffer

Use the `DSpContext_SetUnderlayAltBuffer` function to designate an alternate buffer to be used as the current underlay buffer for a context.

```
OSStatus DSpContext_SetUnderlayAltBuffer (
                    DSpContextReference inContext,
                    DSpAltBufferReference inNewUnderlay);
```

`inContext` A reference to the context the underlay is to be used with.

`inNewUnderlay`
A reference to the alternate buffer that holds the underlay.

function result A result code.

DESCRIPTION

The `DSpContext_SetUnderlayAltBuffer` function makes the alternate buffer specified in the `inNewUnderlay` parameter serve as the current underlay for the context specified in the `inContext` parameter. Underlay buffers are used to "clean" a back buffer when `DSpContext_GetBackBuffer` is called. When a back buffer is retrieved and there is an underlay buffer, the invalid areas in the back buffer are restored from the underlay buffer. This is most useful in sprite games, or in games where the background is static (or changes infrequently).

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_GetUnderlayAltBuffer

You can use the `DSpContext_GetUnderlayAltBuffer` function to get the current underlay associated with a context.

```
OSStatus DSpContext_GetUnderlayAltBuffer (
                    DSpContextReference inContext,
                    DSpAltBufferReference *outUnderlay);
```

`inContext` A reference to the context whose underlay you want to get.

`outUnderlay` On exit, a reference to the alternate buffer that holds the underlay.

function result A result code.

DESCRIPTION

The `DSpContext_GetUnderlayAltBuffer` function takes, in the `inContext` parameter, a reference to a context and returns, in the `outUnderlay` parameter, a reference to the alternate buffer that holds the current underlay for that context.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpAltBuffer_InvalRect

You can use the `DSpAltBuffer_InvalRect` function to invalidate a rectangle in an alternate buffer.

```
OSStatus DSpAltBuffer_InvalRect (
                    DSpAltBufferReference inAltBuffer,
                    const Rect *inInvalidRect);
```

`inAltBuffer` A reference to an alternate buffer.

`inInvalidRect` A pointer to the rectangle to be invalidated.

function result A result code.

DESCRIPTION

You would invalidate a rectangle in an alternate buffer in the following situations:

- Before calling `DSpAltBuffer_RebuildTransparencyMask` so as to rebuild only those sections of a transparency mask that have changed.
- To invalidate areas of an overlay or underlay you have changed so that the changes are transferred to the back buffer on the next `DSpContext_SwapBuffers` call.

You pass the `DSpAltBuffer_InvalRect` function a reference to an alternate buffer in the `inAltBuffer` parameter and a pointer to the rectangle to be invalidated in the `inInvalidRect` parameter.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpAltBuffer_RebuildTransparencyMask

Use the `DSpAltBuffer_RebuildTransparencyMask` function to rebuild the transparency mask associated with an alternate buffer being used as an overlay.

```
OSStatus DSpAltBuffer_RebuildTransparencyMask (
                    DSpAltBufferReference inAltBuffer,
                    UInt32 inTransparencyValue);
```

`inAltBuffer` The buffer whose transparency mask is to be rebuilt.

`inTransparencyValue`
The pixel values that are to be transparent.

function result A result code.

DESCRIPTION

You must rebuild the transparency masks associated with an alternate buffer if you are using the alternate buffer as an overlay in the following situations:

- You have just created the alternate buffer and rendered its image for the first time. Any time a context is activated, the transparency mask is built automatically.
- You have modified the overlay image in such a way that you need a different transparency mask.

The `DSpAltBuffer_RebuildTransparencyMask` function takes a reference to the buffer containing the transparency mask to be rebuilt in the `inAltBuffer` parameter and instructions as to which pixels to make transparent in the `inTransparencyValue` parameter.

If you have modified only a subsection of the alternate buffer, you can invalidate the rectangles enclosing the areas you have modified with the `DSpAltBuffer_InvalRect` function (page 2-65) and then rebuild the transparency mask for those areas only.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

Handling a Mouse

Because the coordinate system of a context may not correspond exactly to the global system coordinates, you must use the functions in this section to track the position of the mouse.

DSpFindContextFromPoint

You can use the `DSpFindContextFromPoint` function to find out which context corresponds to a point given in global coordinates.

```
OSStatus DSpFindContextFromPoint (
                    Point inGlobalPoint,
                    DSpContextReference *outContext);
```

`inGlobalPoint` A point in global coordinates.

`outContext` On exit, a reference to a context that contains that point.

function result A result code.

DESCRIPTION

If the user moves the mouse, the game needs to know which context it's in so that the global coordinates can be properly translated into local coordinates for the context. The function takes, in the `inGlobalPoint` parameter, the global coordinates of a point and returns, in the `outContext` parameter, a reference to the context that point is in.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpGetMouse

Use the `DSpGetMouse` function to get the global coordinates of the mouse position.

```
OSStatus DSpGetMouse  (Point *outGlobalPoint);
```

`outGlobalPoint`
On exit, the global coordinates of the mouse position.

function result A result code.

DESCRIPTION

The function returns the global coordinates of the mouse position in the `outGlobalPoint` parameter.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_GlobalToLocal

Use the `DSpContext_GlobalToLocal` function to translate a point in global coordinates into local coordinates for a context.

```
OSStatus DSpContext_GlobalToLocal (
                    DSpContextReference inContext,
                    Point *ioPoint);
```

`inContext` A reference to the context whose local coordinates you want to translate into.

`ioPoint` Takes a point in global coordinates. On exit, contains the point in local coordinates.

function result A result code.

DESCRIPTION

You pass the `DSpContext_GlobalToLocal` function a point in global coordinates in the `ioPoint` parameter and the context in which the point is located in the `inContext` parameter. The function returns, in the `ioPoint` parameter, the point in the context's local coordinates.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_LocalToGlobal

You can use the `DSpContext_LocalToGlobal` function to translate a point's coordinates given in a context's local coordinates into global coordinates.

```
OSStatus DSpContext_LocalToGlobal (
                    DSpContextReference inContext,
                    Point *ioPoint);
```

`inContext`	The context whose local coordinate system describes the point's coordinates.
`ioPoint`	Takes a point's local coordinates. On exit, contains the point's global coordinates.
function result	A result code.

DESCRIPTION

You pass the function the local coordinates of a point in the `ioPoint` parameter and, in the `inContext` parameter, the context to which the local coordinates belong. The function returns, in the `ioPoint` parameter, the global coordinates of the point.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

Manipulating Color Lookup Tables

The functions described in this section give you convenient access to the entries of a color lookup table.

DSpContext_SetCLUTEntries

You can use the `DSpContext_SetCLUTEntries` function to assign one or more color entries to a color lookup table.

```
OSStatus DSpContext_SetCLUTEntries (
                    DSpContextReference inContext,
                    const ColorSpec *inEntries,
                    UInt16 inStartingEntry,
                    UInt16 inEntryCount);
```

`inContext` The context whose color lookup table is to be modified.

`inEntries` A pointer to an array of color specification records.

`inStartingEntry`
The (zero-based) index position in the color lookup table of the first entry to replace.

`inEntryCount` The number of entries to replace.

function result A result code.

DESCRIPTION

The `DSpContext_SetCLUTEntries` function allows you to change a range of entries in a color lookup table, for purposes such as color-table animation. You pass in the context whose color lookup table is to be modified in the `inContext` parameter, the index position of the first entry to replace in the `inStartingEntry` parameter, and the number of entries to replace in the `inEntryCount` parameter. The `inEntries` parameter contains a pointer to the array of new entries.

Because of video hardware limitations, the changes you make to a color table with this function may not take effect until the next vertical retrace. Nevertheless, this function attempts to execute asynchronously and return

immediately, so your program can continue execution without having to wait for the changes to be made.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

DSpContext_GetCLUTEntries

You can use the `DSpContext_GetCLUTEntries` function to retrieve one or more color entries from a color lookup table.

```
OSStatus DSpContext_GetCLUTEntries (
                    DSpContextReference inContext,
                    ColorSpec *outEntries,
                    UInt16 inStartingEntry,
                    UInt16 inEntryCount);
```

`inContext` The context whose color lookup table is to be accessed.

`outEntries` On exit, an array of color specification records that contain the retrieved table entries.

`inStartingEntry`
The (zero-based) index position in the color lookup table of the first entry to retrieve.

`inEntryCount` The number of entries to retrieve.

function result A result code.

DESCRIPTION

The `DSpContext_GetCLUTEntries` function gets a range of entries from the color lookup table for the context specified in the `inContext` parameter. You pass in the number of entries to get in the `inEntryCount` parameter and the index position of the first entry in the `inStartingEntry` parameter. The function returns, in the `outEntries` parameter, an array of records containing the table entries it retrieved.

After you get the entries you can modify them and reassign them to the color table, for purposes such as color-table animation, with the `DSpContext_SetCLUTEntries` function (page 2-70).

CALLING RESTRICTIONS

Do not call this function during an interrupt.

Processing System Events

The function described in this section passes system events through to DrawSprocket.

DSpProcessEvent

Use the `DSpProcessEvent` function to pass system events through to DrawSprocket so that it can correctly handle events it must know about.

```
OSStatus DSpProcessEvent (
                    EventRecord *inEvent,
                    Boolean *outEventWasProcessed);
```

`inEvent` A pointer to the event to be passed to DrawSprocket.

`outEventWasProcessed`
On exit, `true` if DrawSprocket processed the event; `false` if the event was not processed.

function result A result code.

DESCRIPTION

You pass the `DSpProcessEvent` function a pointer to the event to be handled in the `inEvent` parameter. The function returns `true` in the `outEventWasProcessed` parameter if the event was processed. Your game must pass system events through DrawSprocket so that it can correctly handle events. In particular, whenever your game receives a suspend or resume event, it must call the `DSpProcessEvent` function so that DrawSprocket can correctly set the system state for the process switch.

When DrawSprocket is suspended, it returns the display to the resolution mode it was in before your context's play state first became active. When DrawSprocket resumes, it restores the display to the resolution mode used by your context. However, it is your responsibility to update the contents of the display at this time.

CALLING RESTRICTIONS

Do not call this function during an interrupt.

Utility Functions

This section describes two functions: one that aids your debugging efforts by maintaining your access to the debugging screen at all times, and one that facilitates implementation of VBL tasks.

DSpSetDebugMode

During debugging you can use the `DSpSetDebugMode` function to keep the screen and system resources visible at all times.

```
OSStatus DSpSetDebugMode(Boolean inDebugMode);
```

`inDebugMode` `true` if the desktop display is to remain visible, even after fading; `false` otherwise.

DESCRIPTION

During development, if you drop into the debugger when the display has been faded out, you cannot fade the display back in so that you can see the debugger screen. Calling the `DSpSetDebugMode` function with the `inDebugMode` flag set to a value of `true` causes your program to enter a mode in which the blanking window is not drawn and every fade operation (either in or out) causes only a partial dimming and immediate restoration of the screen intensity. Calling this function with the `inDebugMode` flag set to a value of `false` ends the mode and resumes normal operation.

To make use of this function, you must call it before activating your context. Once the blanking window is in place, this function effects only gamma fades.

This function is ignored in nondebugging builds of DrawSprocket.

DSpContext_SetVBLProc

You can use the `DSpContext_SetVBLProc` function to piggyback your own VBL task to a particular context.

```
OSStatus DSpContext_SetVBLProc (
                    DSpContextReference inContext,
                    DSpCallbackProcPtr inProcPtr,
                    void *inRefCon);
```

`inContext`	The context the VBL task is associated with.
`inProcPtr`	A pointer to an application-supplied callback function.
`inRefCon`	An reference constant to be handed back by DrawSprocket when it calls the `inProcPtr` callback.
function result	A result code.

DESCRIPTION

Because DrawSprocket needs to set up VBL tasks of its own, you can piggyback your own VBL task to a particular context easily with this function, instead of digging down through the system to find the correct slot ID and installing your own. Pass the function the context in the `inContext` parameter and a pointer to the VBL task callback function in the `inProcPtr` parameter. The callback pointer must be of this type:

```
typedef Boolean (*DSpCallbackProcPtr)(DSpContextReference inContext,
                 void *inRefCon);
```

The reference constant passed in the `inrefCon` parameter will be handed back by DrawSprocket when it calls the callback procedure. The return value from the `inProcPtr` parameter is ignored by DrawSprocket. See the `MyCallbackFunction` function (page 2-75) for a description of the callback.

Application-Defined Functions

This section describes the interfaces to two callback functions that you can provide: one that's called before buffers are swapped or when using the `DSpContext_SetVBLProc` function, and one that's called to pass events to the game during dialog-box display.

MyCallbackFunction

You can use the `MyCallbackFunction` function to perform any necessary tasks in preparation for swapping display buffers or piggybacking VBL tasks to a context.

```
Boolean MyCallbackFunction
                    (DSpContextReference inContext,
                    void *inRefCon);
```

`inContext` A reference to a context.

`inRefCon` A reference constant to be handed back to the game by the DrawSprocket function that calls `MyCallbackFunction`.

function result Returns `true` if your game has handled the event, `false` if not.

DESCRIPTION

The `MyCallbackFunction` takes, in the `inContext` parameter, a reference to a context and, in the `inRefCon` parameter, a reference constant to be handed back to the game by DrawSprocket. The function should return `false` if your tasks or checks are complete. If it returns `true`, the function is still performing necessary tasks.

MyEventHandler

You can use the `MyEventHandler` function to allow your game to handle events during the display of the device-selection dialog box.

```
pascal Boolean MyEventHandler (
                    EventRecord *event);
```

`event` A pointer to an event record.

function result Returns `true` if your game has handled the event, `false` if not.

DESCRIPTION

The `MyEventHandler` function returns `true` if your game handled the event specified in the `event` parameter; it returns `false` if it did not. Calls to `MyEventHandler` result from a call to the `DSpUserSelectContext` function. If you pass a pointer to this callback function to `DSpUserSelectContext`, your game can handle events, such as update events, that occur while the dialog box is displayed.

Summary of DrawSprocket

Constants

Depth Masks

```
enum DSpDepthMask {
    kDSpDepthMask_1                 = 1U<<0,
    kDSpDepthMask_2                 = 1U<<1,
    kDSpDepthMask_4                 = 1U<<2,
    kDSpDepthMask_8                 = 1U<<3,
    kDSpDepthMask_16                = 1U<<4,
    kDSpDepthMask_32                = 1U<<5,
    kDSpDepthMask_All               = ~0U
};

typedef enum DSpDepthMask DSpDepthMask;
```

Color Needs

```
enum DSpColorNeeds {
    kDSpColorNeeds_DontCare         = 0L,
    kDSpColorNeeds_Request          = 1L,
    kDSpColorNeeds_Require          = 2L
};

typedef enum DSpColorNeeds DSpColorNeeds;
```

Special Display Features

```
enum DSpContextOption {
    kDSpContextOption_QD3DAccel     = 1U<<0,
    kDSpContextOption_PageFlip      = 1U<<1,
    kDSpContextOption_TripleBuffer  = 1U<<2
```

```
};

typedef enum DSpContextOption DSpContextOption;
```

Buffer Kind

```
enum DSpBufferKind {
    kDSpBufferKind_Normal           = 0U
};

typedef enum DSpBufferKind DSpBufferKind;
```

Pixel Scaling

```
enum DSpBufferScale {
    kDSpBufferScale_1               = 0x00000001U,
    kDSpBufferScale_2               = 0x00000002U,
    kDSpBufferScale_2Interpolate    = 0x80000002U,
    kDSpBufferScale_3               = 0x00000003U,
    kDSpBufferScale_3Interpolate    = 0x80000003U,
    kDSpBufferScale_4               = 0x00000004U,
    kDSpBufferScale_4Interpolate    = 0x80000004U
};

typedef enum DSpBufferScale DSpBufferScale;
```

Play State

```
enum DSpContextState {
    kDSpContextState_Active         = 0L,
    kDSpContextState_Paused         = 1L,
    kDSpContextState_Inactive       = 2L
};

typedef enum DSpContextState DSpContextState;
```

Data Types

```
typedef struct DSpContextPrivate        *DSpContextReference;

typedef struct DSpAltBufferPrivate      *DSpAltBufferReference;

typedef Boolean (*DSpEventProcPtr)      (EventRecord *inEvent);

typedef Boolean (*DSpCallbackProcPtr)   (DSpContextReference inContext,
                                         void *inRefCon);
```

Context Attributes Structure

```
struct DSpContextAttributes {
    Fixed               frequency;
    UInt32              displayWidth;
    UInt32              displayHeight;
    UInt32              reserved1;
    UInt32              reserved2;
    UInt32              colorNeeds;
    CTabHandle          colorTable;
    OptionBits          contextOptions;
    OptionBits          backBufferDepthMask;
    OptionBits          displayDepthMask;
    UInt32              backBufferBestDepth;
    UInt32              displayBestDepth;
    UInt32              pageCount;
    Boolean             gameMustConfirmSwitch;
    UInt32              reserved3[4];
}

typedef struct DSpContextAttributes DSpContextAttributes;
typedef struct DSpContextAttributes *DSpContextAttributesPtr
```

DrawSprocket Functions

Using DrawSprocket

```
OSStatus DSpStartup                 (void);
```

```
OSStatus DSpShutdown                (void);
```

Choosing a Context and Saving Preferences

```
OSStatus DSpFindBestContext         (const DSpContextAttributesPtr inDesiredAttributes,
                                     DSpContextReference *outContext);

OSStatus DSpGetFirstContext         (DisplayIDType displayID,
                                     DSpContextReference *outContext);

OSStatus DSpGetNextContext          (DSpContextReference inCurrentContext,
                                     DSpContextReference *outContext);

OSStatus DSpContext_GetAttributes   (DSpContextReference inContext,
                                     DSpContextAttributesPtr outAttributes);

OSStatus DSpCanUserSelectContext    (DSpContextAttributesPtr inDesiredAttributes
                                     Boolean *outUserCanSelectContext );

OSStatus DSpUserSelectContext       (DSpContextAttributesPtr inDesiredAttributes,
                                     DisplayIDType inDialogDisplayLocation,
                                     DSpEventProcPtr inEventProc,
                                     DSpContextReference *outContext );

OSStatus DSpContext_Restore         (void *inFlatContext,
                                     DSpContextReference *outRestoredContext);

OSStatus DSpContext_GetFlattenedSize            (DSpContextReference inContext,
                                                 UInt32 *outFlatContextSize);

OSStatus DSpContext_Flatten                     (DSpContextReference inContext,
                                                 void *outFlatContext);

OSStatus DSpContext_GetDisplayID                (DSpContextReference inContext,
                                                 DisplayIDType *outDisplayID);
```

Manipulating a Context

```
OSStatus DSpContext_Reserve     (DSpContextReference inContext,
                                 const DSpContextAttributesPtr inDesiredAttributes);

OSStatus DSpContext_Release                     (DSpContextReference inContext);

OSStatus DSpContext_SetState                    (DSpContextReference inContext,
                                                 DSpContextState inState);
```

```
OSStatus DSpContext_GetState                (DSpContextReference inContext,
                                             DSpContextState *outState);

OSStatus DSpSetBlankingColor                (const RGBColor *inRGBColor);
```

Drawing and Double Buffering

```
OSStatus DSpContext_FadeGamma               (DSpContextReference inContext,
                                             SInt32 inPercentOfOriginalIntensity,
                                             RGBColor *inZeroIntensityColor);

OSStatus DSpContext_FadeGammaOut            (DSpContextReference inContext,
                                             RGBColor *inZeroIntensityColor);

OSStatus DSpContext_FadeGammaIn             (DSpContextReference inContext,
                                             RGBColor *inZeroIntensityColor);

OSStatus DSpContext_GetBackBuffer           (DSpContextReference inContext,
                                             DSpBufferKind inBufferKind,
                                             CGrafPtr *outBackBuffer);

OSStatus DSpContext_InvalBackBufferRect     (DSpContextReference inContext,
                                             const Rect *inRect);

OSStatus DSpContext_SwapBuffers             (DSpContextReference inContext,
                                             DSpCallbackProcPtr inBusyProc,
                                             void *inUserRefCon);

OSStatus DSpContext_IsBusy                  (DSpContextReference inContext,
                                             Boolean *outBusyFlag);

OSStatus DSpContext_SetDirtyRectGridSize    (DSpContextReference inContext,
                                             UInt32 inCellPixelWidth,
                                             UInt32 inCellPixelHeight);

OSStatus DSpContext_GetDirtyRectGridSize    (DSpContextReference inContext,
                                             UInt32 *outCellPixelWidth,
                                             UInt32 *outCellPixelHeight);

OSStatus DSpContext_GetDirtyRectGridUnits   (DSpContextReference inContext,
                                             UInt32 *outCellPixelWidth,
                                             UInt32 *outCellPixelHeight);

OSStatus DSpContext_SetMaxFrameRate         (DSpContextReference inContext,
                                             UInt32 inMaxFPS);
```

```
OSStatus DSpContext_GetMaxFrameRate          (DSpContextReference inContext,
                                              UInt32 *outMaxFPS);

OSStatus DSpContext_GetMonitorFrequency      (DSpContextReference inContext,
                                              Fixed *outFrequency);

OSStatus DSpContext_SetScale                 (DSpContextReference inContext,
                                              DSpBufferScale inScale);

OSStatus DSpContext_GetScale                 (DSpContextReference inContext,
                                              DSpBufferScale *outScale);
```

Using Alternate Buffers

```
OSStatus DSpAltBuffer_New                    (DSpContextReference inContext,
                                              Boolean inVRAMBuffer,
                                              DSpAltBufferReference *outAltBuffer);

OSStatus DSpAltBuffer_Dispose                (DSpAltBufferReference inAltBuffer);

OSStatus DSpAltBuffer_GetCGrafPtr            (DSpAltBufferReference inAltBuffer,
                                              DSpBufferKind inBufferKind,
                                              CGrafPtr *outCGrafPtr);

OSStatus DSpContext_SetOverlayAltBuffer      (DSpContextReference inContext,
                                              DSpAltBufferReference inNewOverlay);

OSStatus DSpContext_GetOverlayAltBuffer      (DSpContextReference inContext,
                                              DSpAltBufferReference *outOverlay);

OSStatus DSpContext_SetUnderlayAltBuffer     (DSpContextReference inContext,
                                              DSpAltBufferReference inNewUnderlay);

OSStatus DSpContext_GetUnderlayAltBuffer     (DSpContextReference inContext,
                                              DSpAltBufferReference *outUnderlay);

OSStatus DSpAltBuffer_InvalRect              (DSpAltBufferReference inAltBuffer,
                                              const Rect *inInvalidRect);

OSStatus DSpAltBuffer_RebuildTransparencyMask (DSpAltBufferReference inAltBuffer,
                                              UInt32 inTransparencyValue);
```

Handling a Mouse

```
OSStatus DSpFindContextFromPoint            (Point inGlobalPoint,
                                             DSpContextReference *outContext);

OSStatus DSpGetMouse                        (Point *outGlobalPoint);

OSStatus DSpContext_GlobalToLocal           (DSpContextReference inContext,
                                             Point *ioPoint);

OSStatus DSpContext_LocalToGlobal           (DSpContextReference inContext,
                                             Point *ioPoint);
```

Manipulating Color Lookup Tables

```
OSStatus DSpContext_SetCLUTEntries          (DSpContextReference inContext,
                                             const ColorSpec *inEntries,
                                             UInt16 inStartingEntry,
                                             UInt16 inEntryCount);

OSStatus DSpContext_GetCLUTEntries          (DSpContextReference inContext,
                                             ColorSpec *outEntries,
                                             UInt16 inStartingEntry,
                                             UInt16 inEntryCount);
```

Processing System Events

```
OSStatus DSpProcessEvent                    (EventRecord *inEvent,
                                             Boolean *outEventWasProcessed);
```

Utility Functions

```
OSStatus DSpSetDebugMode                    (Boolean inDebugMode);

OSStatus DSpContext_SetVBLProc              (DSpContextReference inContext,
                                             DSpCallbackProcPtr inProcPtr,
                                             void *inRefCon);
```

Application-Defined Functions

```
Boolean MyCallbackFunction                  (DSpContextReference inContext,
                                             void *inRefCon);pascal Boolean

pascal Boolean MyEventHandler               (EventRecord* event);
```

Result Codes

`kDSpNotInitializedErr`	–30440L	`DSpStartup` has not yet been called.
`kDSpSystemSWTooOldErr`	–30441L	System software too old.
`kDSpInvalidContextErr`	–30442L	Invalid context reference.
`kDSpInvalidAttributesErr`	–30443L	Some field in an attributes structure has an invalid value.
`kDSpContextAlreadyReservedErr`	–30444L	The context is already reserved.
`kDSpContextNotReservedErr`	–30445L	The context is not reserved.
`kDSpContextNotFoundErr`	–30446L	DrawSprocket couldn't find the context.
`kDSpFrameRateNotReadyErr`	–30447L	Not enough time has passed for DrawSprocket to calculate a frame rate.
`kDSpConfirmSwitchWarning`	–30448L	The `gameMustConfirmSwitch` flag is set.
`kDSpInternalErr`	–30449L	Corrupted DrawSprocket or other error.

InputSprocket

Contents

InputSprocket is the part of Apple Game Sprockets that your application can use to match up game controls with existing device controls, configure device controls, and track data from device controls.

Before reading about InputSprocket, you should be generally familiar with Apple Game Sprockets, as described in the preface to this book. Both game developers and developers of input devices can benefit from using InputSprocket. This chapter focuses on game applications. Developers of input devices should also consult the driver-specific information found on Apple's Web site.

This chapter begins by describing the basic capabilities of InputSprocket. Then it shows how to use some of those capabilities for such tasks as choosing the best display for your game, creating overlays and underlays, transitioning into the game, double and triple buffered drawing, cleaning up before quitting, and debugging. The section "InputSprocket Reference" (page 3-17), provides a complete reference to the constants, data structures, and functions provided by InputSprocket in the InputSprocket.h file. (The InputSprocketDriver.h file reference is available on Apple's Web site.) For a list of constants, data structures, and functions see the section "Summary of InputSprocket" (page 3-60).

Note
This chapter describes the application programming interfaces supported by versions 1.0 and later of InputSprocket. ◆

About InputSprocket

InputSprocket is the part of Apple Game Sprockets through which games can communicate with joysticks and other game-oriented input devices, as well as the keyboard and mouse. Most games on the MacOS access game input devices through emulation on the keyboard and mouse. This strategy suffers from two problems—complex devices and configuration difficulties.

Todays input devices may have directional pads, levers, point of view hats, wheels, and several buttons and a game may use several input devices. Such complexity becomes impossible to emulate through a mouse and keyboard. Even with simple devices, configuration can be a two-step process. The user

has to go to the device to configure what keys and mouse actions it generates and then go to the game to configure which keys and mouse actions it accepts. The user can become confused if the configurations of device and game don't mesh.

InputSprocket solves these problems by creating an input device architecture that allows games developers to create innovative games that can use a wide variety of input devices. Input device developers can use InputSprocket to build device drivers that provide a description of input device controls that the game can use to automatically configure its control options. The device driver can also provide a user interface that allows the user to change default control options.

During game play, the game determines the state and transitions of input device controls through polling or a simple event queue.

Elements

InputSprocket bases communication between games and input devices on elements. An **element** is a building block used to describe a device. Each control on the device has a corresponding element, so any device, no matter how complex, can be described by a handful of elements. For example, a simple input device with two buttons and a thrust lever consisting of an x-axis and y-axis could be described by four elements. A complex device with many more axes, several buttons, and directional pads might require 12 or more elements.

Three pieces of data describe a given element: a human readable identifier, a kind, and a label. The **human readable identifier** is a string that a game can display to identify that element for the user during configuration—for example, "trigger" or "thumb button".

The **kind** is an four-character sequence that identifies the type of data the element produces. For example, it produces data like a button (two states) or like an axis (continuous data). Currently, InputSprocket defines four kinds of elements to correspond to four kinds of controls— button, directional pad, axis, and general movement. (A fifth element kind identifies virtual elements.)

- A **button element** is a two-state element. It both generates events and can be polled—for example, a mouse button, keyboard key, or joystick trigger. It is by far the most common input element.

- A **directional pad element** is a nine state element. It has an idle position and eight states corresponding to eight directions—for example, a point of view hat on a joystick or a directional pad on a Sega controller. (Some directional pads have only four directions.)
- There are two types of **axis elements**. One is a symmetrical axis that the user sees as being continuous and having a meaningful center position—for example, the axis of a joystick. The other type of axis is one that the user sees as being continuous but without a meaningful center position—for example, a gas pedal or a brake.
- A **movement** element produces data that is given both as x-y axis data and directional pad data, allowing the game to use whichever is suitable.

Apple may define other element kinds later, and device driver developers can create element kinds as well. The kind imposes a structure on the data. For a description of existing element kinds, see "Built-in Element Kinds" (page 3-18).

A **label** is a four-character sequence that gives the suggested use for an element. For example, a button element may be intended as the start button or the firing button. During configuration, a game uses labels to find the elements it requires for play. For a description of existing labels, see "Built-in Element Labels" (page 3-19).

Device Configuration

When InputSprocket is loaded, it loads all the InputSprocket drivers, which it finds by scanning the Extensions folder. Each device driver provides information about its controls, which InputSprocket stores for use by the game during configuration and play.

For each element corresponding to a control, the driver specifies the element kind, label, and human readable identifier and identifies elements that work as a unit—for example, sets of x-axis and y-axis elements on a control that has both a joystick and a throttle. It gives the size of the data produced by the element so that functions that get data can allocate sufficient memory to hold it.

The device driver also fills out a configuration information structure for each control. These structures contain information specific to the element kind, for example, whether a directional pad has four directions or eight. The game may use the information during configuration and to interpret data from the control during play.

For more information, see the documentation provided for input device developers on Apple's Web site.

Game Configuration

Games must match their input requirements with input device controls on the system. InputSprocket makes this easy by

- giving the games a standard way to publish their requirements for input
- providing functions for autoconfiguration and user reconfiguration

When a game is loaded, it fills out a **need structure** to provide information about each type of input it wants to receive. When the game is ready for configuration, it calls the `ISpInit` function. This function organizes the information in the need structures into a list of input requirements that InputSprocket can pass around to the input device drivers. Each device driver inspects the list and checks off requirements it can fulfill. The device driver keeps track of which of its device elements will be providing data to the game. For more information on need structures see "Need Structure" (page 3-29).

After calling `ISpInit` to perform autoconfiguration, the game can call the `ISpConfigure` function to put up a modal window where the user can make changes to the configuration. Each device driver manages its own user configuration process with some help from InputSprocket, which is still aware of the game's input requirements.

In one possible model for user configuration, the device driver lists the requirements the game has for control—for example, move horizontally, fire, jump, duck, and so on. Next to the list of input requirements, it displays a list of controls, using the elements' human readable identifiers to name the controls that can meet those needs. The user could select a requirement from the list and then activate one of the controls to indicate which control the user wants to use to meet that input requirement. For example, after selecting "duck" the user could push button three on the joystick.

Apple recommends that game developers take advantage of the configuration functions provided by InputSprocket. However, it is possible for a game to perform configuration using other functions. Since every element in the system has a label suggesting a use for the control that element represents, the game can search for elements with specific labels. If the game can't find an element with that label, it can use the element kind information to find an element that

generates the type of data it wants. InputSprocket provides functions to help you search for elements by kind and by label.

Devices that provide several elements of the same kind that have the same label or no meaningful label can order the elements. This lets the game know which control to use first. The device driver puts ordering information in the `id` field of the trigger, button, and directional pad configuration structures. After finding the elements that best match its input needs, the game would need to let the user know which physical controls correspond to the game controls.

Obtaining Data During Play

There are two ways for the game to obtain information from the input device during play—polling and events. It takes only a single call to determine the state of any element.

You can get events over all the elements in the system, but a game usually polls elements such as axes and uses an event queue for elements such as buttons. InputSprocket provides several functions for searching the system and building a list of elements so that the game can get the next event on the list. Don't add those elements you don't want to get events for to your element list; instead, get their current state by polling.

InputSprocket uses the **element event structure** to pass event data. See "Element Event Data Structure" (page 3-28).

Using InputSprocket

This section illustrates basic ways of using InputSprocket. In particular, it provides source code samples that show how you can

- Initialize need structures and allocate virtual elements
- Build an element list
- Process input data during game play
- Deactivate keyboard and mouse devices

Note
The code samples shown in this section provide no error handling. ◆

Initializing Need Structures and Allocating Virtual Elements

During initialization the game describes its requirements for input by filling out a `ISpNeed` data structure for each requirement (the structure is described on (page 3-29). The need structure has fields for an element kind and an element label, so the game can describe the data and its intended use. Other fields specify how to use the keyboard and mouse for keyboard emulation. The game also provides a string and an icon to identify the input requirement in a user interface.

After filling out the need structures, the game allocates a virtual element to meet each input requirement. Allocation of virtual elements must happen before you call `ISpInit` because after that call the device driver may begin using the virtual elements.

The `InitNeeds` routine shown in Listing 3-1 creates an array of initialized `ISpNeed` structures. It then uses the `ISpElement_NewVirtualFromNeeds` function (page 3-36) to allocate virtual elements. You can use `ISpElement_NewVirtualFromNeeds` only with built-in element kinds, which is the case in the example. Otherwise, you allocate virtual elements individually using the `ISpElement_NewVirtual` function (page 3-36).

Listing 3-1 Initializing need structures and allocating virtual elements

```
OSStatus InitNeeds(ISpNeed *theNeeds,ISpElementReference *elements)
{
    ISpNeed tempNeeds[kNumNeeds] =
    {
        {
            "\pForward Thrust",
            kIconSuiteID_YThrust,
            kISpElementKind_Axis,
            kISpElementLabel_YAxis,
            0
        },
        {
```

```
            "\pSide Thrust",
            kISpElementKind_Axis,
            kISpElementLabel_XAxis,
            0
        },
        {
            "\pVertical Thrust",
            kIconSuiteID_ZThrust,
            kISpElementKind_Axis,
            kISpElementLabel_ZAxis,
            0
        },
        {
            "\pLook",
            kIconSuiteID_Look,
            kISpElementKind_Movement,
            kISpElementLabel_None,
            0
        },
        {
            "\pLaser",
            kIconSuiteID_Fire,
            kISpElementKind_Button,
            kISpElementLabel_Fire,
            0
        },
        {
            "\pMissle",
            kIconSuiteID_Pause,
            kISpElementKind_Button,
            kISpElementLabel_Fire,
            0
        },
        { "\pShields",
            kIconSuiteID_Shields,
            kISpElementKind_Button,
            kISpElementLabel_None,
            0
        },
        {
            "\pStart/Stop",
```

```
            kIconSuiteID_Start,
            kISpElementKind_Button,
            kISpElementLabel_Start,
            kISpNeedFlag_NoMultiConfig
        }
    };

    int itr;
    for(itr = 0; itr < kNumNeeds; itr++)
    {
        theNeeds[itr] = tempNeeds[itr];
    }

    OSStatus result = ISpElement_NewVirtualFromNeeds(kNumNeeds, theNeeds,
        elements, 0);

    return result;
}
```

Building an Element List

When configuration is finished and virtual elements are allocated, you probably want to build a list of elements the game wants to get events for during play. Listing 3-2 shows the `BuildMyElementList` routine, which puts all the button and directional pad elements on an element list. Games usually poll axis and movement elements, so they are not included on the list.

`BuildMyElementList` also makes the reference constant returned by `ISpElementList_AddElements` for each element it adds equal to the index of the element in the array of need structures passed to the drivers during autoconfiguration. This is also the same as the index of the element in the list of virtual elements allocated during the autoconfiguration process (see Listing 3-1 for an example of creating virtual elements). The game uses the reference constant (which is passed in the `ISpElementEvent` structure) to identify the element when it gets element events using `ISpElementList_GetNextEvent`.

Listing 3-2 Building an element list

```
ISpElementListReference BuildMyElementList(UInt32 count, ISpNeed *needs,
    ISpElementReference *elements)
{
    int itr;
    ISpElementListReference theList = nil;

    ISpElementList_New(0, nil, &theList, 0);
    for(itr = 0; itr < count; itr++)
    {
        if ((needs[itr].theKind == kISpElementKind_Button) ||
            (needs[itr].theKind == kISpElementKind_DPad))
        {
                ISpElementList_AddElements(theList, itr, 1,
                    &(elements[itr]));
        }
    }

    return theList;
}
```

`BuildMyElementList` first defines an element list reference to the list that will be built. It next uses the `ISpElementList_New` function (page 3-51) to create an empty element list. Finally, `BuildMyElementList` uses the `ISpElementList_AddElements` function (page 3-53) to add elements of kind `kISpElementKind_Button` and `kISpElementKind_DPad` to the list. They are added one at a time so that each element can be assigned the reference constant that corresponds to its index in the array of need structures.

Processing Input Data During Play

During game play the game obtains data from all the input devices either by polling or getting events. Listing 3-3 shows the `ProcessInput` routine. This routine runs the main game loop, polling for the state of axis and movement elements and getting events from button elements.

`ProcessInput` takes an element list that has been built so that the index of each element corresponds to its position in the array of need structures passed to the drivers during autoconfiguration (see Listing 3-2 for an example of how to do this).

Notice that there are two routines for getting the state of an element—`ISpElement_GetSimpleState` and `ISpElement_GetComplexState`. The `ISpElement_GetSimpleState` function (page 3-47) is for elements whose data fits in an unsigned 32-bit integer. For other elements, in this case, movement kind elements, use `ISpElement_GetComplexState` (page 3-47) where you can specify the size of the buffer needed to hold the data.

Buttons work in a variety of ways and the code illustrates how to get events for three types—a firing button, where only down events matter (the laser and missile buttons); a toggle (the stop button); and a hold-to-activate button (the shield button).

Listing 3-3 Processing input data

```
enum
{
    kForwardThrust = 0,
    kSideThrust,
    kVerticalThrust,
    kLook,
    kLaser,
    kMissle,
    kShields,
    kStartStop,
    kNumNeeds
};

typedef struct InputData
{
    UInt32      x,y,z;           // x,y and z thrust
    UInt32      xLook, yLook;
    Boolean     fireLasers;
    Boolean     fireMissles;
    Boolean     shields;        // down activates; up deactivates
    Boolean     stopped;        // toggle
} InputData;

void ProcessInput(ISpElementReference *theElements,
    ISpElementListReference myList, InputData *myInput)
{
```

```
    ISpElement_GetSimpleState(theElements[kForwardThrust],
         &(myInput->x));
    ISpElement_GetSimpleState(theElements[kSideThrust], &(myInput->y));
    ISpElement_GetSimpleState(theElements[kVerticalThrust],
        &(myInput->z));

    ISpMovementData tempMovement;
    ISpElement_GetComplexState(theElements[kLook],
        sizeof(ISpMovementData),&tempMovement);
    myInput->xLook = tempMovement.xAxis;
    myInput->yLook = tempMovement.yAxis;

    myInput->fireLasers = false;
    myInput->fireMissles = false;

    Boolean wasEvent;
    ISpElementEvent theEvent;

    while(1)
    {
        ISpElementList_GetNextEvent(myList, sizeof(ISpElementEvent),
            &theEvent, &wasEvent);

        switch(theEvent.refCon)
        {
            case kLaser:
                if (theEvent.data == kISpButtonDown)
                {
                    myInput->fireLasers = true;
                }
            break;

            case kMissle:
                if (theEvent.data == kISpButtonDown)
                {
                    myInput->fireMissles = true;
                }
            break;

            case kShields:
                if (theEvent.data == kISpButtonDown)
```

```
                {
                    myInput->shields = true;
                }
                else if (theEvent.data == kISpButtonUp)
                {
                    myInput->shields = false;
                }
            break;

            case kStartStop:
                if (theEvent.data == kISpButtonDown)
                {
                    myInput->stopped = !myInput->stopped;
                }
            break;
        }

        if (!wasEvent)
        {
            return;
        }
    }
}
```

Turning the Keyboard and Mouse On and Off

By default, keyboard and mouse input devices are inactive. This is fine for games that run in a window where the user probably expects the mouse and keyboard to behave normally. However, most games will want to activate the keyboard and mouse. Later, there may be a point in the game where the user needs to enter text, in which case, you would want to deactivate the keyboard and mouse as game input devices so that you can return to using the standard methods provided by the OS to manage them.

Listing 3-4 shows the `SetKeyboardMouseActivation` function that deactivates and activates the keyboard and mouse devices.

Listing 3-4 Turning off keyboard and mouse devices

```
void SetKeyboardMouseActivation(Boolean active)
{
    enum                { kSimple = 100 };
    ISpDeviceReference  buffer[kSimple];
    UInt32              count;

    ISpDevices_ExtractByClass(kISpDeviceClass_Mouse, kSimple, &count,
        buffer);

    if (active) { ISpDevices_Activate(count,buffer); }
    else { ISpDevices_Deactivate(count, buffer); }

    ISpDevices_ExtractByClass(kISpDeviceClass_Keyboard,kSimple,&count,
        buffer);

    if (active) { ISpDevices_Activate(count,buffer); }
    else { ISpDevices_Deactivate(count, buffer); }
}
```

The `SetKeyboardMouseActivation` function uses `ISpDevices_ExtractByClass` (page 3-39) to find the keyboard and mouse devices. Notice that the buffer allocated to hold the device references is allocated for 100 devices, a number likely to be sufficient. The `ISpDevices_Deactivate` function is described on (page 3-41); `ISpDevices_Activate` is on (page 3-41).

InputSprocket Reference

This section describes the constants, data structures, functions, and resources provided by InputSprocket.

Constants

This section describes the constants provided by InputSprocket.

Built-in Device Categories

These constants identify the general category of devices the input device belongs to. Use the constants in the `theDeviceClass` field of the `ISpDeviceDefinition` data structure.

```
enum {
    kIspDeviceClass_SpeechRecognition = 'talk',
    kISpDeviceClass_Mouse =             'mous',
    kISpDeviceClass_Keyboard =          'keyd',
    kISpDeviceClass_Joystick =          'joys',
    kISpDeviceClass_Wheel =             'whel',
    kISpDeviceClass_Pedals =            'pedl',
    kISpDeviceClass_Levers =            'levr'
};
```

Constant descriptions

`kIspDeviceClass_SpeechRecognition`
: The device is primarily a speech-recognition device.

`kISpDeviceClass_Mouse`
: The device is a one-button mouse.

`kISpDeviceClass_Keyboard`
: The device is a keyboard.

`kISpDeviceClass_Joystick`
: The device is a joystick.

`kISpDeviceClass_Wheel`
: The device is primarily a wheel.

`kISpDeviceClass_Pedals`
: The device is primarily a pedal.

`kISpDeviceClass_Levers`
: The device is primarily a lever—for example, a device built around a thrust lever.

Built-in Element Kinds

Use these constants to specify the kind of data an element produces and to identify virtual elements. Element kind constants are used in the `ISpElementInfo` structure and by the `ISpElementList_ExtractByKind` function.

```
enum {
    kISpElementKind_Button =    'butn',
    kISpElementKind_DPad =      'dpad',
    kISpElementKind_Axis =      'axis',
    kISpElementKind_Movement =  'move'
    kISpElementKind_Virtual =   'virt'
};
```

Constant descriptions

`kISpElementKind_Button`
: Button data.

`kISpElementKind_DPad`
: Directional pad data.

`kISpElementKind_Axis`
: Axis data, either with or without a meaningful center position (as determined by the `ISpAxisConfigurationInfo` data structure).

`kISpElementKind_Movement`
: Movement data that is given both as x-y axis data and directional pad data, allowing the game to use whichever is suitable.

`kISpElementKind_Virtual`
: A virtual element created by the `ISpElement_NewVirtual` function. Those created by the `ISpElement_NewVirtualFromNeeds` function have the element kind specified by the need structure they correspond to.

Built-in Element Labels

InputSprocket provides constants to define labels for controls common to most games. You can use these labels to specify an element's intended use. You can also define your own labels as needed. Element label constants are used in `ISpElementInfo` and structure and by the `ISpElementList_ExtractByLabel` function.

```
enum {
    kISpElementLabel_None =     'none',

    kISpElementLabel_XAxis =    'xaxi',
```

```
    kISpElementLabel_YAxis =    'yaxi',
    kISpElementLabel_ZAxis =    'zaxi',

    kISpElementLabel_Rx =       'rxax',
    kISpElementLabel_Ry =       'ryax',
    kISpElementLabel_Rz =       'rzax',

    kISpElementLabel_Gas =      'gasp',
    kISpElementLabel_Brake =    'brak',
    kISpElementLabel_Clutch =   'cltc',

    kISpElementLabel_Throttle = 'thrt',
    kISpElementLabel_Trim =     'trim',
    kISpElementLabel_POVHat =   'povh',
    kISpElementLabel_PadMove =  'move',

    kISpElementLabel_Fire =     'fire',
    kISpElementLabel_Start =    'strt',
    kISpElementLabel_Select =   'optn'
};
```

Constant descriptions

`kISpElementLabel_None`
: A generic label; the element can be used as anything.

`kISpElementLabel_XAxis`
: Suggest using the element as an x-axis.

`kISpElementLabel_YAxis`
: Suggest using as the element a y-axis.

`kISpElementLabel_ZAxis`
: Suggest using the element as a z-axis.

`kISpElementLabel_Rx`
: Suggest using the element for rotation around the x-axis.

`kISpElementLabel_Ry`
: Suggest using the element for rotation around the y-axis.

`kISpElementLabel_Rz`
: Suggest using the element for rotation around the z-axis.

`kISpElementLabel_Gas`
: Suggest using the element as a gas pedal.

`kISpElementLabel_Brake`
: Suggest using the element as a brake.

`kISpElementLabel_Clutch`
: Suggest using the element as a clutch.

`kISpElementLabel_Throttle`
: Suggest using the element as a throttle.

`kISpElementLabel_Trim`
: Suggest using the element as a trim control.

`kISpElementLabel_POVHat`
: Suggest using the element as a point-of-view hat.

`kISpElementLabel_PadMove`
: Suggest using the element for directional pad movement.

`kISpElementLabel_Fire`
: Suggest using the element as a firing button.

`kISpElementLabel_Start`
: Suggest using the element as a start button.

`kISpElementLabel_Select`
: Suggest using the element as an option or select button.

Button Element Data Information

These constants specify information for data of kind `kButtonData`. Use the `ISpButtonConfigurationInfo` structure for help in interpreting the data.

```
typedef enum {
    kISpButtonUp =              0,
    kISpButtonDown =            1
} ISpButtonData;
```

Constant descriptions

`kISpButtonUp` Button is up.

`kISpButtonDown` Button is down.

Directional Pad Element Data Information

These constants specify information for data of kind `kElementData`. Use the `ISpDPadConfigurationInfo` structure for help in interpreting the data.

```
typedef enum {
    kISpPadIdle =               0,
    kISpPadLeft,
    kISpPadUpLeft,
    kISpPadUp,
    kISpPadUpRight,
    kISpPadRight,
    kISpPadDownRight,
    kISpPadDown,
    kISpPadDownLeft
} ISpDPadData;
```

Constant descriptions

`kISpPadIdle`	Idle position.
`kISpPadLeft`	Left position.
`kISpPadUpLeft`	Upper-left position.
`kISpPadUp`	Upper-center position.
`kISpPadUpRight`	Upper-right position.
`kISpPadRight`	Right position.

Axis Element Data Information

These constants specify information for data of kind `kAxisData`. Use the `ISpAxisConfigurationInfo` structure for help in interpreting the data.

```
#define kISpAxisMinimum  0x00000000U
#define kISpAxisMiddle   0x7FFFFFFFU
#define kISpAxisMaximum  0xFFFFFFFFU
```

Constant descriptions

`kISpAxisMinimum`	Minimum distance along the axis.
`kISpAxisMiddle`	Center of the axis.
`kISpAxisMaximum`	Maximum distance along the axis.

Need Flags

Use these constants in the `flags` bit field of an `ISpNeed` structure. Currently there is only one flag defined, which indicates whether or not to allow multiconfiguration of an input requirement.

```
typedef enum ISpNeedFlagBits {
    kISpNeedFlag_NoMultiConfig =            1
};
```

Constant descriptions

`kISpNeedFlag_NoMultiConfig`
Allow only one device to bind to this requirement during autoconfiguration.

Virtual Element Flag

Use this constant with the `ISpElement_NewVirtual` and `ISpElement_NewVirtualFromNeeds` functions to tell InputSprocket to allocate the virtual elements in temporary memory.

```
{
    kISpVirtualElementFlag_UseTempMem = 1
};
```

Constant descriptions

`kISpVirtualElementFlag_UseTempMem`
Allocate the virtual element in temporary memory.

Element List Flag

Use this constant with the `ISpElementList_New` function to tell InputSprocket to allocate the element list in temporary memory.

```
enum
{
    kISpElementListFlag_UseTempMem = 1
};
```

Constant descriptions

`kISpElementListFlag_UseTempMem`
Allocate the element list in temporary memory.

Resource Types

These constants identify resource types defined by InputSprocket. See “Resources” (page 3-58).

```
enum
{
    kISpSetListResourceType =       'setl',
    kISpSetDataResourceType =       'tset'
};
```

Constant descriptions

`kISpSetListResourceType`
A resource of type `'setl'`.

`kISpSetDataResourceType`
A resource of type `'tset'`.

Data Structures

This section describes the data structures provided by InputSprocket.

Device Definition Structure

A device definition structure provides all the information about the input device available within the system. A device definition structure is defined by the `ISpDeviceDefinition` data type.

```
typedef struct ISpDeviceDefinition {
    Str63                   deviceName;
    ISpDeviceClass          theDeviceClass;
    ISpDeviceIdentifier     theDeviceIdentifier;
    UInt32                  permanentId;
    UInt32                  flags;
    UInt32                  reserved1;
```

```
    UInt32                  reserved2;
    UInt32                  reserved3;
} ISpDeviceDefinition;
```

Field descriptions

`deviceName` A human readable string with the name of the device.

`theDeviceClass` The general category of device—for example, joystick, keyboard, mouse, and so on. There are built-in values for the `theDeviceClass` parameter, and device driver developers may create new ones. For a list of built-in values see "Built-in Device Categories" (page 3-18).

`theDeviceIdentifier` The registered four-character sequence that identifies the type of device.

`permanentId` An ID that identifies the device even after rebooting. If the device driver cannot distinguish this device from other devices with the same value for `theDeviceIdentifier`, this field should be 0.

`flags` Status flags.

`reserved1` Reserved. Always set this field to 0.

`reserved2` Reserved. Always set this field to 0.

`reserved3` Reserved. Always set this field to 0.

Element Information Structure

The element information structure provides basic information about an element that is common to all elements, regardless of kind. An element information structure is defined by the `ISpElementInfo` data type.

```
typedef struct ISpElementInfo {
    ISpElementLabel     theLabel;
    ISpElementKind      theKind;
    Str63               theString;
    UInt32              reserved1;
    UInt32              reserved2;
} ISpElementInfo, *ISpElementInfoPtr;
```

Field descriptions

`theLabel`	The label of the element. See "Built-in Element Labels" (page 3-19) for built-in values.
`theKind`	The kind of data produced by the element. See "Built-in Element Kinds" (page 3-18) for built-in values.
`theString`	The localized human readable identifier for use in the user interface.
`reserved1`	Reserved. Always set this field to 0.
`reserved2`	Reserved. Always set this field to 0.

Button Configuration Information Structure

The button configuration information structure provides information used during configuration and in interpreting button element data. For each element of kind `kISpElementKind_Button`, the device driver fills out a button configuration information structure, which is stored by InputSprocket. A button configuration information structure is defined by the `ISpButtonConfigurationInfo` data type.

```
typedef struct {
    UInt32              id;
} ISpButtonConfigurationInfo;
```

Field descriptions

`id`	Use this ID to indicate which button element to assign first during configuration if the device has more than one button element. A lower value indicates priority. For example, make the value of `id` for an easy to reach button 1 and make it 6 for a button in the back so that the easy-to-reach button gets assigned first. A value of 0 means use the button elements in any order.

Directional Pad Configuration Information Structure

The directional pad configuration information structure provides information used during configuration and in interpreting directional pad element data. For each element of kind `kISpElementKind_DPad`, the device driver fills out a directional pad configuration information structure, which is stored by

InputSprocket. A directional pad configuration information structure is defined by the `ISpDPadConfigurationInfo` data type.

```
typedef struct {
    UInt32              id;
    Boolean             fourWayPad;
} ISpDPadConfigurationInfo;
```

Field descriptions

`id`	Use this ID to indicate which directional pad to assign first during configuration if the device has more than one directional pad element. A lower value indicates priority. A value of 0 means use the directional pads in any order.
`fourWayPad`	This value is `true` if the pad can produce only four directions plus idle.

Axis Configuration Information Structure

The axis configuration information structure provides information used during configuration and in interpreting axis element data. For each element of kind `kISpElementKind_Axis`, the device driver fills out an axis configuration information structure, which is stored by InputSprocket. An axis configuration information structure is defined by the `ISpAxisConfigurationInfo` data type.

```
typedef struct {
    Boolean             SymetricAxis;
} ISpAxisConfigurationInfo;
```

Field descriptions

`SymetricAxis`	This value is `true` if the axis has a meaningful center—for example, the axis of a joystick. In this case, 0 or idle falls at the `kISpAxisMiddle` position. This value is `false` if the axis has no meaningful center—for example, the axis of a brake or gas pedal.

Movement Data Structure

The movement data structure provides data from elements of kind `kISpElementKind_Movement`. This element kind produces data that is given both

as x-y axis data and directional pad data, allowing the game to use whichever is suitable. The movement data structure is defined by the ISpMovementData data type.

```
typedef struct ISpMovementData {
    UInt32              xAxis;
    UInt32              yAxis;
    UInt32              direction;
};
```

Field descriptions

xAxis	Movement data given in terms of the x-axis of an x-y axis pair.
yAxis	Movement data given in terms of the y-axis of an x-y axis pair.
direction	Movement data given as a direction.

Element Event Data Structure

The element event structure is a variable length structure that passes element event data. An element event data structure is defined by the ISpElementEvent data type.

```
typedef struct {
    AbsoluteTime        when;
    ISpElementReference element;
    UInt32              refCon;
    Uint32              data;
} ISpElementEvent, *ISpElementEventPtr;
```

Field descriptions

when	The time the event happened. If the application is running on a PCI machine, the time will be in units of AbsoluteTime (as generated by the UpTime call). Otherwise, the when.hi field will be 0 and the when.lo field will be the time generated by TickCount.
element	A reference to the element that generated the event.
refCon	The reference constant assigned to this element on the element list that contains it (see the ISpElementList_AddElements function on (page 3-53). If you

got this event from the `ISpElement_GetNextEvent` function or from a global list created by the systems, this field is 0.

`data` The data for the event. It is a variable length field that is often, but not always, a 32-bit unsigned integer. Valid values include those given in the description of the "Button Element Data Information" (page 3-21), "Directional Pad Element Data Information" (page 3-21), "Axis Element Data Information" (page 3-22), and the "Movement Data Structure" (page 3-27). You need to examine the element kind to determine what size and type of data is reported.

For example, a new type of input data may refer to RGB color, so it requires 6 bytes of data instead of 4 bytes. It would be typed by a new element kind—for example, `'rgbc'`. If you wanted to get RGB element kind data, you would have to allocate a larger structure when you were getting events via the event mechanism than you would allocate for 4-byte data.

The type of data varies for different elements.

Need Structure

During initialization the game fills out a need structure for each input requirement. The need structure describes the type of data that will satisfy the input requirement and also gives information you can use in a user interface during configuration. The need structure is defined by the `ISpNeed` data type.

```
typedef struct ISpNeed {
    Str63               name;
    short               iconSuiteResourceId;
    ISpElementKind      theKind;
    ISpElementLabel     theLabel;
    UInt32              flags;
    UInt32              reserved1;
    UInt32              reserved2;
    UInt32              reserved3;
} ISpNeed;
```

Field descriptions

`name` A human readable string that can be used during configuration to describe the input requirement to the user.

`iconSuiteResourceId` A resource ID of an icon suite residing in the games resource fork. This is the same resource ID you would pass to the `GetIconSuite` function if you wanted to load the icon yourself. The icon is a picture representing the input requirement.

`theKind` The kind of element that will produce the data the game requires. This can be a standard element kind—for example, `kISpElementKind_Button`—or an extended virtual kind. The only extended kind that exists currently is the movement kind, which consists of an x-y axis pair and a direction, where both are based on the same data and the game uses whichever is appropriate.

`theLabel` A standard element label representing what you are going to use the data for.

`flags` You should OR in `kISpNeedFlag_NoMultiConfig` if you would prefer that only one device bind to the need during autoconfiguration. Otherwise, several devices may agree to meet the same input requirement.

`reserved1` Reserved. Always set this field to 0.

`reserved2` Reserved. Always set this field to 0.

`reserved3` Reserved. Always set this field to 0.

InputSprocket Functions

This section describes the functions provided by InputSprocket.

Configuring the Application

This section describes the functions that InputSprocket uses during autoconfiguration and to help the game provide a user interface for user-directed reconfiguration. This section also describes functions you can use to create virtual elements, to determine the characteristics of the input device

controls in the system, and to help in choosing the optimum configuration for your game.

ISpInit

You can use the `ISpInit` function to initialize the InputSprocket layer and autoconfigure all the devices.

```
OSStatus ISpInit (
                    UInt32 count,
                    ISpNeed *needs,
                    ISpElementReference *inReferences,
                    OSType appCreatorCode,
                    OSType subCreatorCode,
                    UInt32 flags,
                    short setListResourceID
                    UInt32 version);
```

`count` The number of input requirements the game has. Each requirement is described in an `ISpNeed` structure.

`needs` A pointer to an array of `ISpNeed` structures. The order of the need structures in the array is important because the input devices will try to fulfill input requirements beginning with the first need structure in the array. More important requirements should be put first—for example, "jump" before "look at map."

`inReferences`
A pointer to an array of virtual elements identifying the elements that can meet your game's input requirements. The array will contain the number of element references specified in the `count` parameter. You can use all the usual calls to get events or poll these element references.

`appCreatorCode`
The creator code of the application.

`subCreatorCode`
The subcreator code. InputSprocket and device drivers use a union of the creator and subcreator codes to save and restore preferences. For every pair of codes the game's requirements

for input should be identical; otherwise, there is an unknown result. The subcreator code gives you a domain in which to have multiple different preference settings within any given application.

`flags` Leave as 0.

`setListResourceID`
A resource of type `kISpSetListResourceType`. See "The InputSprocket Set List Resource" (page 3-58)

`version` The version of InputSprocket the game depends on.

function result A result code.

DESCRIPTION

The `ISpInit` function takes the number of input requirements in the `count` parameter, a pointer to an array of need structures in the `needs` parameter, and a pointer to an array of virtual element references in the `inReferences` parameter. A virtual element reference does not correspond to any physical control on a physical device. Devices push data into a virtual element reference when they have data that corresponds to the input requirement that reference represents. DrawSprocket provides two functions for creating virtual elements—`ISpElement_NewVirtual` (page 3-36) and `ISpElement_NewVirtualFromNeeds` (page 3-36).

To determine which elements can meet the game's requirements for input, `ISpInit` goes down a list of system devices and asks each device in turn if it can meet any of the requirements listed in the `needs` array. On the list of system devices, which is created the first time InputSprocket is loaded, the keyboard appears last and the mouse next to last; there is no particular order for other devices. The game can use the following InputSprocket routines for extracting devices from the list and deactivating or activating devices to eliminate devices that are unsuited to the game from the list: the `ISpDevices_Extract` function (page 3-38), the `ISpDevices_ExtractByClass` function (page 3-39), the `ISpDevices_ExtractByIdentifier` function (page 3-40), the `ISpDevices_Activate` function (page 3-41), and the `ISpDevices_Deactivate` function (page 3-41).

As each device tries to fulfill the input requirements, it looks at the need structures in the order in which they appear in the array. If a particular requirement has already been fulfilled by a prior device and has the `kInputSprocketNoMultiConfig` flag set in the `ISpNeed` structure `flags` field, the device will ignore it. The device driver keeps track of how its actual device

elements are matched to the virtual element references it creates— in other words, which elements will meet which input requirements.

For each device driver, InputSprocket stores its configuration information, using the codes passed in the `appCreatorCode` and `subCreatorCode` parameters to identify them for future use.

The game passes a resource of type `kISpSetListResourceType` in the `setListResourceID` parameter. The game also passes a version number in the `version` parameter so that if the system is running an incompatible version of InputSprocket, the user can be alerted.

CALLING RESTRICTIONS

Do not call during interrupt time.

ISpConfigure

You can use the `ISpConfigure` function to generate a modal window where the user can match device elements with the game's input requirements.

```
OSStatus ISpConfigure  (ISpEventProcPtr inEventProcPtr);
```

`inEventProcPtr`
A pointer to an application-supplied function for handling update events.

function result A result code.

DESCRIPTION

When you call `ISpConfigure`, InputSprocket generates a modal window where the user can choose device elements to meet game input requirements. This allows the user to modify the autoconfiguration. You pass the function a pointer to an application-supplied procedure for handling events. If an event happens that the game needs to deal with, it can handle the event and return `true`. If it does not handle the event, it returns `false`. When the `ISpConfigure` call returns, the reconfiguring process is completed and any changes are saved to disk.

CALLING RESTRICTIONS

Do not call during interrupt time.

ISpStop

You can use the `ISpStop` function to stop the flow of data into the virtual elements that began with the `ISpInit` call. This is data that is switched from device driver callbacks. The application should call the function before it quits.

```
OSStatus ISpStop  (void);
```

function result A result code.

DESCRIPTION

The `ISpStop` function stops data from device driver callbacks from being pushed into virtual elements.

ISpSuspend

You can use the `ISpSuspend` function to suspend InputSprocket.

```
OSStatus ISpSuspend  (void);
```

function result A result code.

DESCRIPTION

The `ISpSuspend` function suspends InputSprocket. It must be called in response to suspend events. It also can be called any time you want to stop getting InputSprocket data.

ISpResume

You can use the `ISpResume` function to resume running InputSprocket.

```
OSStatus ISpResume  (void);
```

function result A result code.

DESCRIPTION

The `ISpResume` function resumes InputSprocket (after a suspend event). You can call it in response to a resume event or anytime you want to resume running InputSprocket after having suspended it.

ISpGetVersion

You can use the `ISpGetVersion` function to find out what version of InputSprocket is running.

```
NumVersion ISpGetVersion  (void);
```

function result Returns the version number of InputSprocket.

DESCRIPTION

The `ISpGetVersion` function returns the version number of InputSprocket.

ISpElement_NewVirtual

You can use the `ISpElement_NewVirtual` function to create a single virtual element for an element with data of a certain size.

```
OSStatus ISpElement_NewVirtual (
                    UInt32 dataSize,
                    ISpElementReference *outElement;
                    UInt32 flags);
```

`dataSize` The size of the data.

`outElement` On exit, a reference to the virtual element.

`flags` Set the `kISpVirtualElementFlag_UseTempMem` bit to tell InputSprocket to allocate the virtual element in temporary memory. See "Virtual Element Flag" (page 3-23).

function result A result code.

DESCRIPTION

You pass the `ISpElement_NewVirtual` function the size of the data in the `dataSize` parameter and it returns, in the `outElement` parameter, a reference to a virtual element that can be allocated to an element whose data is that size. If you want the virtual element allocated in temporary memory pass `kISpVirtualElementFlag_UseTempMem` in the `flags` parameter.

ISpElement_NewVirtualFromNeeds

You can use the `ISpElement_NewVirtualFromNeeds` function to allocate virtual elements for all items in a need structure array.

```
OSStatus ISpElement_NewVirtualFromNeeds (
                    UInt32 count,
                    ISpNeed *needs,
                    ISpElementReference *outElements
                    UInt32 flags);
```

`count`	The number of need structures for which to allocate virtual elements.
`needs`	A pointer to an array of need structures.
`outElements`	On exit, an array of element references.
`flags`	Set the `kISpVirtualElementFlag_UseTempMem` bit to tell InputSprocket to allocate the virtual element in temporary memory. See "Virtual Element Flag" (page 3-23).
function result	A result code.

DESCRIPTION

You pass the `ISpElement_NewVirtualFromNeeds` function an array of need structures in the `needs` parameter and the number of those structures in the `count` parameter. The function returns, in the `outElements` parameter, the element references for the virtual elements allocated. If you want the virtual elements allocated in temporary memory pass `kISpVirtualElementFlag_UseTempMem` in the `flags` parameter.

This function only works if you have built-in element kinds, which are described in "Built-in Element Kinds" (page 3-18). For other element kinds, use the `ISpElement_NewVirtual` function to create virtual elements.

ISpElement_DisposeVirtual

You can use the `ISpElement_DisposeVirtual` function to dispose of virtual elements. You must call the `ISpStop` function before disposing of any elements that are receiving data.

```
OSStatus ISpElement_DisposeVirtual (
                    UInt32 count,
                    ISpElementReference *inElements);
```

`count`	The number of elements to be disposed of.
`inElements`	A pointer to an array of element references.
function result	A result code.

DESCRIPTION

To dispose of virtual elements, you pass the `ISpElement_DisposeVirtual` function a pointer to an array of element references in the `inElements` parameter and the number of elements to dispose of in the `count` parameter.

ISpDevices_Extract

You can use the `ISpDevices_Extract` function to extract and count devices listed on the systemwide list of devices. This may be useful if you want to find and deactivate input devices—usually the keyboard and mouse—prior to autoconfiguration.

```
OSStatus ISpDevices_Extract (
                    UInt32 inBufferCount,
                    UInt32 *outCount,
                    ISpDeviceReference *buffer);
```

`inBufferCount` The number of device references in the array pointed to by the `buffer` parameter.

`outCount` The number of device references on the systemwide list.

`buffer` A pointer to an array of device references.

function result A result code.

DESCRIPTION

The `ISpDevices_Extract` function takes, in the `buffer` parameter, a pointer to an array of device references and, in the `inBufferCount` parameter, the number of device references in the array. The `ISpDevices_Extract` function copies device references from the systemwide list of devices into that array. If there are more devices in the list than there is space in your array, it copies only as many device references as fit. The function returns the total number of devices in the system wide list of devices in the `outCount` parameter.

ISpDevices_ExtractByClass

You can use the `ISpDevices_ExtractByClass` function to extract and count devices of a certain class listed on the systemwide list of devices. This may be useful if you want to find and deactivate input devices—usually the keyboard and mouse—prior to autoconfiguration.

```
OSStatus ISpDevices_ExtractByClass (
                    ISpDeviceClass theClass,
                    UInt32 inBufferCount,
                    UInt32 *outCount,
                    ISpDeviceReference *buffer);
```

`theClass` The category of device to count and extract. See "Built-in Device Categories" (page 3-18) for built-in values for this parameter.

`inBufferCount` The number of device references in the array pointed to by the `buffer` parameter.

`outCount` The number of devices of the specified category on the systemwide list.

`buffer` A pointer to an array of device references.

function result A result code.

DESCRIPTION

The `ISpDevices_ExtractByClass` function takes, in the `buffer` parameter, a pointer to an array of device references and, in the `inBufferCount` parameter, the number of device references in the array. The `ISpDevices_ExtractByClass function` copies into that array, device references of the class specified by the `theClass` parameter found on the systemwide list of devices. If there are more devices of that class in the list than there is space in your array, it copies only as many device references as fit. The function returns, in the `outCount` parameter, the total number of devices of the specified category in the systemwide list of devices.

ISpDevices_ExtractByIdentifier

You can use the `ISpDevices_ExtractByIdentifier` function to extract and count devices of a certain type that are listed on the systemwide list of devices. This may be useful if you want to find and deactivate input devices—usually the keyboard and mouse—prior to autoconfiguration.

```
OSStatus ISpDevices_ExtractByIdentifier (
                    ISpDeviceIdentifier theIdentifier,
                    UInt32 inBufferCount,
                    UInt32 *outCount,
                    ISpDeviceReference *buffer);
```

`theIdentifier`
The type of device to count and extract.

`inBufferCount`
The number of device references in the array pointed to by the `buffer` parameter.

`outCount` The number of devices of the specified type on the systemwide list.

`buffer` A pointer to an array of device references.

function result A result code.

DESCRIPTION

The `ISpDevices_ExtractByIdentifier` function takes, in the `buffer` parameter, a pointer to an array of device references and, in the `inBufferCount` parameter, the number of device references in the array. The `ISpDevices_ExtractByIdentifier function` copies into that array, device references of the type specified by the `theIdentifier` parameter found on the systemwide list of devices. If there are more devices of that type in the list than there is space in the array, it copies only as many device references as fit. The function returns, in the `outCount` parameter, the total number of devices of the specified type in the systemwide list of devices.

ISpDevices_Activate

You can use the `ISpDevices_Activate` function to activate devices.

```
OSStatus ISpDevices_Activate (
                    UInt32 inDeviceCount,
                    ISpDeviceReference *inDevicesToActivate);
```

`inDeviceCount`
The number of references in the array pointed to by the `inDevicesToActivate` parameter.

`inDevicesToActivate`
A pointer to an array of device references.

function result A result code.

DESCRIPTION

You pass in a pointer to an array of references to the devices to be activated in the `inDevicesToActivate` parameter and pass in the number of references in the array in the `inDeviceCount` parameter. The `ISpDevices_Activate` function activates the devices. When a device is activated, InputSprocket receives events from it.

The following categories of devices are inactive by default: `kIspDeviceClass_SpeechRecognition`, `kISpDeviceClass_Mouse`, and `kISpDeviceClass_Keyboard`.

ISpDevices_Deactivate

You can use the `ISpDevices_Deactivate` function to deactivate devices from which you do not want to receive events. For example, you might want to get input from the keyboard or mouse as text style data. All devices in the system start out activated.

```
OSStatus ISpDevices_Deactivate (
                    UInt32 inDeviceCount,
                    ISpDeviceReference *inDevicesToDeactivate);
```

`inDeviceCount`
The number of references in the array pointed to by the `inDevicesToActivate` parameter.

`inDevicesToDeactivate`
A pointer to an array of device references.

function result A result code.

DESCRIPTION

You pass in a pointer to an array of references to the devices to be deactivated in the `inDevicesToActivate` parameter and pass in the number of references in the array in the `inDeviceCount` parameter. The `ISpDevices_Deactivate` function deactivates the devices. When a device is deactivated, InputSprocket no longer receives events from it.

ISpDevice_IsActive

You can use the `ISpDevice_IsActive` function to find out if a device is active.

```
OSStatus ISpDevice_IsActive (
                    ISpDeviceReference inDevice,
                    Boolean *outIsActive);
```

`inDevice` A reference to a device.

`outIsActive` This value is `true` if the device is active, `false` if it is inactive.

function result A result code.

DESCRIPTION

You pass the `ISpDevice_IsActive` function a reference to a device in the `inDevice` parameter and it returns, in the `outIsActive` parameter, `true` if the device is active or `false` if it is inactive.

ISpDevice_GetDefinition

You can use the `ISpDevice_GetDefinition` function to get a device definition structure for a specified device.

```
OSStatus ISpDevice_GetDefinition (
                    const ISpDeviceReference inDevice,
                    UInt32 buflen,
                    ISpDeviceDefinition *outStruct);
```

`inDevice` The device whose device definition structure you want to get.

`buflen` The length of the device definition buffer.

`outStruct` A pointer to a device definition structure.

function result A result code.

DESCRIPTION

The `ISpDevice_GetDefinition` function returns a pointer to a device definition structure for the device specified in the `inDevice` parameter and stores it in the location passed in the `outStruct` parameter. You also pass in, in the `buflen` parameter, the length of the buffer needed for a device definition structure.

ISpDevice_GetElementList

You can use the `ISpDevice_GetElementList` function to get an element list for a specified device.

```
OSStatus ISpDevice_GetElementList (
                    const ISpDeviceReference inDevice,
                    ISpElementListReference *outElementList);
```

`inDevice` A reference to the device whose element list you want to get.

`outElementList`
On exit, a reference to the element list.

function result A result code.

DESCRIPTION

The `ISpDevice_GetElementList` function returns a reference to the element list for the device specified in the `inDevice` parameter and stores it in the location indicated by the `outElementList` parameter. The element list is read-only and you cannot modify it.

ISpElement_GetDevice

You can use the `ISpElement_GetDevice` function to find out what device an element belongs to.

```
OSStatus ISpElement_GetDevice (
                    const ISpElementReference inElement,
                    ISpDeviceReference *outDevice);
```

`inElement` A reference to the element whose device you want to get.

`outDevice` On exit, a device reference.

function result A result code.

DESCRIPTION

The `ISpElement_GetDevice` function returns, in the `outDevice` parameter, a device reference for the device the element specified in the `inElement` parameter belongs to.

ISpElement_GetGroup

You can use the `ISpElement_GetGroup` function to find out what group an element belongs to.

```
OSStatus ISpElement_GetGroup (
                    const ISpElementReference inElement,
                    UInt32 *outGroup);
```

`inElement` A reference to the element whose group you want to get.

outGroup	On exit, a group ID.
function result	A result code.

DESCRIPTION

The ISpElement_GetGroup function returns in the outGroup parameter the group ID of the element specified in the inElement parameter. If the specified element does not belongs to a group, the function returns 0.

ISpElement_GetInfo

You can use the ISpElement_GetInfo function to get an element information structure for an element. This function returns information common to all elements (kind, label, and human-readable string). To get element kind–specific information use the ISpElement_GetConfigurationInfo function (page 3-46).

```
OSStatus ISpElement_GetInfo (
                    const ISpElementReference inElement,
                    ISpElementInfoPtr outInfo);
```

inElement	A reference to the element whose element information structure you want to get.
outInfo	On exit, a pointer to an element information structure.
function result	A result code.

DESCRIPTION

The ISpElement_GetInfo function takes an element reference in the inElement parameter and returns a pointer to the element information structure for that element in the outInfo parameter.

ISpElement_GetConfigurationInfo

You can use the `ISpElement_GetConfigurationInfo` function to get element kind–specific information for an element—for example, whether a directional pad has eight directions or four or whether to use a certain button first when matching game input requirements with controls. InputSprocket stores this type of information in configuration information structures—for example, the `ISpButtonConfigurationInfo`, `ISpDPadConfigurationInfo`, and `ISpAxisConfigurationInfo` structures, which are built-in.

```
OSStatus ISpElement_GetConfigurationInfo (
                    const ISpElementReference inElement,
                    UInt32 buflen,
                    void *configInfo);
```

`inElement` A reference to the element whose element kind–specific information you want to get.

`buflen` The length of the buffer for holding the information. The size of the configuration structures varies by element kind.

`configInfo` On exit, a pointer to the buffer containing the data.

function result A result code.

DESCRIPTION

The `ISpElement_GetConfigurationInfo` function returns element kind–specific information for the element specified in the `inElement` parameter and stores it in the memory pointed to by the `configInfo` parameter. You pass in the length of the buffer to hold the data in the `buflen` parameter. If the buffer is not long enough to hold the data, the `ISpElement_GetConfigurationInfo` function copies as many bytes of data as fit and returns an error.

Obtaining Data From Elements

You can use the functions in this section during game play to obtain data from the various elements.

ISpElement_GetSimpleState

Use the `ISpElement_GetSimpleState` function to obtain the current state of an element whose data fits in an unsigned 32-bit integer.

```
OSStatus ISpElement_GetSimpleState (
                    const ISpElementReference inElement,
                    Uint32 *state);
```

`inElement` A reference to the element whose state you want to get.

`state` The current state of the specified element. For a description of some values that may be returned, see "Button Element Data Information" (page 3-21), "Directional Pad Element Data Information" (page 3-21), and "Axis Element Data Information" (page 3-22).

function result A result code.

DESCRIPTION

The `ISpElement_GetSimpleState` function gets the current state of the element specified in the `inElement` parameter and stores it in the location indicated by the `state` parameter. This function is a specialized version of `ISpElement_GetComplexState` in that it is only useful with elements whose data fits in an unsigned 32-bit integer.

ISpElement_GetComplexState

Use the `ISpElement_GetComplexState` function to obtain the current state of an element.

```
OSStatus ISpElement_GetComplexState (
                    const ISpElementReference inElement,
                    UInt32 buflen,
                    void *state);
```

`inElement` A reference to the element whose state you want to get.

`buflen`	The length of the buffer allocated to hold the data. This must match the `dataSize` field of the element's element definition structure.
`state`	On entry, a pointer to a buffer to hold the data. On exit, the state of the specified element.
function result	A result code.

DESCRIPTION

The `ISpElement_GetComplexState` function gets the current state of the element specified in the `inElement` parameter and stores it in the location indicated by the `state` parameter. You pass in the length of the buffer to hold the data in the `buflen` parameter. The `ISpElement_GetComplexState function` copies as much data as will fit into the buffer.

ISpElement_GetNextEvent

You can use the `ISpElement_GetNextEvent` function to get event data for a single element.

```
OSStatus ISpElement_GetNextEvent (
                    ElemenReference inElement,
                    UInt32 bufSize,
                    ISpElementEventPtr event,
                    Boolean *wasEvent);
```

`inElement`	A reference to the element whose event data you want to get.
`bufSize`	The size of the buffer allocated to hold the data.
`event`	A pointer to an element event structure, which is a variable length structure.
`wasEvent`	On exit, this value is `true` if there was an event; otherwise, it is `false`.
function result	A result code.

DESCRIPTION

The `ISpElement_GetNextEvent` function takes, in the `inElement` parameter, an element reference and, in the `bufSize` parameter, the buffer size of an element event structure. It gets the event data, if any, for the specified element and stores the data in the location indicated by the `event` parameter. It sets the `wasEvent` parameter to `true` if there was an event; otherwise, it sets it to `false`. It there is not enough space to hold the entire event data structure, the event is removed from the event queue and `ISpElement_GetNextEvent` returns an error.

This function is the same as `ISpElementList_GetNextEvent` except that it operates on a single event.

ISpElementList_GetNextEvent

You can use the `ISpElementList_GetNextEvent` function to get the most recent event from a list of elements.

```
OSStatus ISpElementList_GetNextEvent (
                    ISpElementListReference inElementList,
                    UInt32 bufSize,
                    ISpElementEventPtr event,
                    Boolean *wasEvent);
```

`inElementList`
: A reference to the element list to get the event from.

`bufSize`
: The size of the buffer allocated to hold the event data.

`event`
: A pointer to an element event structure, which is a variable length structure.

`wasEvent`
: On exit, this value is `true` if there was an event; otherwise, it is `false`.

function result
: A result code.

DESCRIPTION

The `ISpElementList_GetNextEvent` function takes, in the `inElementList` parameter, a reference to an element list and, in the `bufSize` parameter, the buffer size of an element event structure. It gets the event data, if any, from the

specified element list and stores the data in the location indicated by the `event` parameter. It sets the `wasEvent` parameter to `true` if there was an event; otherwise, it sets it to `false`. If there is not enough space to hold the entire event data structure, the event is removed from the event queue and `ISpElement_GetNextEvent` returns an error.

This function is the same as `ISpElement_GetNextEvent` except that it operates on an element list.

ISpElement_Flush

You can use the `ISpElement_Flush` function to flush all the events on an element.

```
OSStatus ISpElement_Flush  (ISpElementReference inElement);
```

`inElement` A reference to the element whose events you want to flush.

function result A result code.

DESCRIPTION

The `ISpElement_Flush` function takes, in the `inElement` parameter, an element reference and flushes all the events on that element. It guarantees to flush only those events that made it to the InputSprocket layer before the call. It will not flush any events that made it to the InputSprocket layer after the call returns. The outcome for events that occur during the call is undefined.

The `ISpElement_Flush` function is the same as `ISpElementList_Flush` except that it operates on a single element.

ISpElementList_Flush

You can use the `ISpElementList_Flush` function to flush all the events on an element list.

```
OSStatus ISpElementList_Flush  (ISpElementListReference inElementList);
```

`inElementList`
A reference to the list of elements whose events you want to flush.

function result A result code.

DESCRIPTION

The `ElementList Flush` function takes, in the `inElementList` parameter, an element list reference and flushes all the events on that element list. It guarantees to flush only those events that made it to the InputSprocket layer before the call. It will not flush any events that made it to the InputSprocket layer after the call returns. The outcome for events that occur during the call is undefined.

The `ISpElementList_Flush` function is the same as `ISpElement_Flush` except that it operates on an element list.

Managing Element Lists

Use the functions in this section to create element lists and to add, remove, and extract elements from those lists.

ISpElementList_New

You can use the `ISpElementList_New` function to create a new element list.

```
OSStatus ISpElementList_New (
             UInt32 inCount,
             ISpElementReference *inElements,
             ISpElementListReference *outElementList
             UInt32 flags);
```

`inCount` The number of element references in the list pointed to by the `inElements` parameter.

`inElements` A pointer to a list of elements to put in the list.

`outElementList`
A pointer to a reference to the new element list.

`flags` Set the `kISpElementListFlag_UseTempMem` bit to tell InputSprocket to allocate the element list in temporary memory. See "Element List Flag" (page 3-23).

function result A result code.

DESCRIPTION

The `ISpElementList_New` function creates a new element list. You pass in the element references to put on the list in the `inElements` parameter and the number of those references in the `inCount` parameter. If you pass in 0 in the `inCount` parameter, `ISpElementList_New` creates an empty list. If you want the list allocated in temporary memory, pass in `kISpElementListFlag_UseTempMem` in the `flags` parameter.

The `ISpElementList_New` function returns a reference to the new element list in the `outElementList` parameter. If it fails to create a new list because it was out of memory, it returns 0 in the `outElementList` parameter.

CALLING RESTRICTIONS

Do not call `ISpElementList_New` at interrupt time because it allocates memory.

ISpElementList_Dispose

You can use the `ISpElementList_Dispose` function to dispose of a specified element list.

```
OSStatus ISpElementList_Dispose  (ISpElementListReference inElementList);
```

`inElementList`
A reference to the element list you want to dispose of.

function result A result code.

DESCRIPTION

The `ISpElementList_Dispose` function disposes of the element list specified in the `inElementList` parameter.

CALLING RESTRICTIONS

Do not call `ISpElementList_Dispose` at interrupt time.

ISpGetGlobalElementList

You can use the `ISpGetGlobalElementList` function to get a global element list. A global element list is a list of all the elements in the system.

```
OSStatus ISpGetGlobalElementList (
                    ISpElementListReference *outElementList);
```

`outElementList`
A reference to the global element list.

function result A result code.

DESCRIPTION

The `ISpGetGlobalElementList` function gets the global element list and returns a reference to it in the `outElementList` parameter.

ISpElementList_AddElements

You can use the `ISpElementList_AddElements` function to add more elements to an element list.

```
OSStatus ISpElementList_AddElements (
                    ISpElementListReference inElementList,
                    UInt32 refCon,
                    UInit32 count,
                    ISpElementReference *newElements);
```

`inElementList`
A reference to the element list you want to add elements to.

`refCon` A reference constant used to locate the new elements in the element list.

`count`	The number of elements to be added.
`newElements`	A pointer to a block of element references.
function result	A result code.

DESCRIPTION

The `ISpElementList_AddElements` function adds the number of elements specified in the `count` parameter to the element list referred to by the `inElementList` parameter. It takes the new elements from the block of references pointed to by the `newElements` parameter. The function returns a reference constant in the `refCon` parameter to use in identifying the new elements in the list.

CALLING RESTRICTIONS

Do not call `ISpElementList_AddElements` at interrupt time.

ISpElementList_RemoveElements

You can use the `ISpElementList_RemoveElements` function to remove elements from an element list.

```
OSStatus ISpElementList_RemoveElements (
                    ISpElementListReference inElementList,
                    UInit32 count,
                    ISpElementReference *oldElement);
```

`inElementList`	A reference to the element list you want to remove elements from.
`count`	The number of elements to remove from the list.
`oldElement`	A pointer to a block of element references.
function result	A result code.

DESCRIPTION

The `ISpElementList_RemoveElements` function removes the number of elements specified by the `count` parameter from the element list referred to by the `inElementList` parameter. The `oldElement` parameter points to a block of references to the elements to be removed.

CALLING RESTRICTIONS

Do not call `ISpElementList_RemoveElements` at interrupt time.

ISpElementList_Extract

You can use the `ISpElementList_Extract` function to extract and count elements in an element list.

```
OSStatus ISpElementList_Extract (
                    ISpElementListReference inElementList,
                    UInt32 inBufferCount,
                    UInt32 *outCount,
                    ISpElementReference *buffer);
```

`inElementList`
A reference to the element list to extract elements from.

`inBufferCount`
The number of element references in the array pointed to by the `buffer` parameter.

`outCount` The number of element references on the element list.

`buffer` A pointer to an array of element references.

function result A result code.

DESCRIPTION

The `ISpElementList_Extract` function takes, in the `buffer` parameter, a pointer to an array of element references and, in the `inBufferCount` parameter, the

number of element references in the array. The `ISpElementList_Extract` function copies element references from the list specified in the `inElementList` parameter into that array. If there are more elements in the list than there is space in your array, it copies only as many element references as fit. The function returns the total number of elements in the element list in the `outCount` parameter.

ISpElementList_ExtractByKind

You can use the `ISpElementList_ExtractByKind` function to extract and count elements of a specified kind in an element list.

```
OSStatus ISpElementList_ExtractByKind (
                    ISpElementListReference inElementList,
                    ISpElementKind theKind,
                    UInt32 inBufferCount,
                    UInt32 *outCount,
                    ISpElementReference *buffer);
```

`inElementList`
A reference to the element list to extract elements from.

`theKind` The kind of elements to extract and count.

`inBufferCount`
The number of element references in the array pointed to by the `buffer` parameter.

`outCount` The number of element references of the specified kind on the element list.

`buffer` A pointer to an array of element references.

function result A result code.

DESCRIPTION

The `ISpElementList_ExtractByKind` function takes, in the `buffer` parameter, a pointer to an array of element references and, in the `inBufferCount` parameter, the number of element references in the array. The `ISpElementList_ExtractByKind` function copies element references of the kind

specified by the `theKind` parameter from the list specified in the `inElementList` parameter into that array. If there are more elements of the specified kind in the list than there is space in the array, it copies only as many element references as fit. The function returns, in the `outCount` parameter, the total number of elements of the specified kin in the element list.

ISpElementList_ExtractByLabel

You can use the `ISpElementList_ExtractByLabel` function to extract and count elements with a specific label in an element list.

```
OSStatus ISpElementList_ExtractByLabel (
                    ISpElementListReference inElementList,
                    ISpElementLabel theLabel,
                    UInt32 inBufferCount,
                    UInt32 *outCount,
                    ISpElementReference *buffer);
```

`inElementList`
A reference to the element list to count and extract elements from.

`theLabel` The label an element must have in order to be counted.

`inBufferCount`
The number of element references in the array pointed to by the `buffer` parameter.

`outCount` The number of element references with the specified label in the element list.

`buffer` A pointer to an array of element references.

function result A result code.

DESCRIPTION

The `ISpElementList_ExtractByLabel` function takes, in the `buffer` parameter, a pointer to an array of element references and, in the `inBufferCount` parameter, the number of element references in the array. The function copies element references with the label specified by the `theLabel` parameter from the list

specified in the `inElementList` parameter into that array. If there are more elements of the specified label in the list than there is space in the array, it copies only as many element references as fit. The function returns, in the `outCount` parameter, the total number of elements with the specified label in the element list.

Resources

This section describes two resources: a set resource and a set list resource, a resource of type `'setl'`. You need to understand these resources in order to build a set list to pass to the `ISpInit` function.

The InputSprocket Set Resource

An InputSprocket set resource, a resource of type `'tset'`, is simply a block of data. The interpretation is up to the individual driver. Do not attempt to edit a set resource except through the dialog box provided by the `ISpConfigure` function. The `ISpConfigure` function saves each set resource in the InputSprocket preferences file. You should then copy the information in the preferences file to your game's resource file.

To get the best results from autoconfiguration you should have resources of this type inside you application.

The InputSprocket Set List Resource

You use a set list resource to specify a list of initial saved sets of preferences for the various devices that a user might use to play your game. A set list resource allows you to have one default configuration as well as various other named configurations. A set list resource is of type `'setl'`.

The set list resource contains the following elements:

- Version. This 4-byte field specifies the version of the `'setl'` data type. The current version is `0x00000001`.
- Count. This 4-byte field specifies the number of set list entries that follow.
- Set list entries. Following the count field is an array of set list entry records. Each set list entry record occupies an 80-byte field. A single set list may have

multiple sets in it. These sets may correspond to multiple devices as well as multiple configurations for a single device.

The set list entry record contains the following elements:

- Name. This 64-byte field contains the name of this set. It will be displayed to the user.
- Set length. This 4-byte field is the length of data in the `'tset'` resource that corresponds to this entry.
- Device identifier. This 4-byte field identifies the device that this set of entry records is for.
- Flags. Two fields are defined for the 4-byte flag field. All others are reserved and should be set to zero. For the first field you should OR in `kSetListEntryNotSelected`. The following information is provided for completeness only:

```
kSetListEntrySelectedMask = 0x000000001
kSetListEntryIsSelected =   0x000000001  // set list entry is selected
kSetListEntryNotSelected =  0x000000000  // set list entry is not
                                         // selected
```

For the second field you should OR in `kSetListEntryDefaultEntry` if you want this entry to be the default; otherwise, OR in `kSetListEntryAppEntry`.

```
kSetListEntryWhoMask =      0x000000006
kSetListEntryNormalEntry =  0x000000000     // a normal user-created
                                            // entry
kSetListEntryDefaultEntry = 0x000000002     // the default entry
kSetListEntryAppEntry =     0x000000004     // an application-provided
                                            // entry
kSetListEntryDriverEntry =  0x000000006     // a driver-provided entry
```

- Set resource ID. This 2-byte field contains the resource ID of the `'tset'` resource that corresponds to this entry.
- Reserved. This 2-byte field is reserved and should be set to zero.

Summary of InputSprocket

Constants

Built-in Device Categories

```
enum {
    kIspDeviceClass_SpeechRecognition = 'talk',
    kISpDeviceClass_Mouse =             'mous',
    kISpDeviceClass_Keyboard =          'keyd',
    kISpDeviceClass_Joystick =          'joys',
    kISpDeviceClass_Wheel =             'whel',
    kISpDeviceClass_Pedals =            'pedl',
    kISpDeviceClass_Levers =            'levr'
};
```

Built-in Element Kinds

```
enum {
    kISpElementKind_Button =            'butn',
    kISpElementKind_DPad =              'dpad',
    kISpElementKind_Axis =              'axis',
    kISpElementKind_Movement =          'move'
    kISpElementKind_Virtual =           'virt'
};
```

Built-in Element Labels

```
enum {
    kISpElementLabel_None =         'none',

    kISpElementLabel_XAxis =        'xaxi',
    kISpElementLabel_YAxis =        'yaxi',
    kISpElementLabel_ZAxis =        'zaxi',
```

```
    kISpElementLabel_Rx =           'rxax',
    kISpElementLabel_Ry =           'ryax',
    kISpElementLabel_Rz =           'rzax',

    kISpElementLabel_Gas =          'gasp',
    kISpElementLabel_Brake =        'brak',
    kISpElementLabel_Clutch =       'cltc',

    kISpElementLabel_Throttle =     'thrt',
    kISpElementLabel_Trim =         'trim',
    kISpElementLabel_POVHat =       'povh',
    kISpElementLabel_PadMove =      'move',

    kISpElementLabel_Fire =         'fire',
    kISpElementLabel_Start =        'strt',
    kISpElementLabel_Select =       'optn'
};
```

Button Element Data Information

```
typedef enum {
    kISpButtonUp =                      0,
    kISpButtonDown =                    1
} ISpButtonData;
```

Directional Pad Element Data Information

```
typedef enum {
    kISpPadIdle =                   0,
    kISpPadLeft,
    kISpPadUpLeft,
    kISpPadUp,
    kISpPadUpRight,
    kISpPadRight,
    kISpPadDownRight,
    kISpPadDown,
    kISpPadDownLeft
} ISpDPadData;
```

Axis Element Data Information

```
#define kISpAxisMinimum  0x00000000U
#define kISpAxisMiddle   0x7FFFFFFFU
#define kISpAxisMaximum  0xFFFFFFFFU
```

Need Flags

```
typedef enum ISpNeedFlagBits {
    kISpNeedFlag_NoMultiConfig =        1
};
```

Virtual Element Flag

```
{
    kISpVirtualElementFlag_UseTempMem =  1
};
```

Element List Flag

```
enum
{
    kISpElementListFlag_UseTempMem =    1
};
```

Resource Types

```
enum
{
    kISpSetListResourceType =           'setl',
    kISpSetDataResourceType =           'setd'
};
```

Data Types

```
typedef struct ISpDevicePrivate             *ISpDeviceReference;

typedef struct ISpElementPrivate            *ISpElementReference;
```

```
typedef struct ISpElementListPrivate        *ISpElementListReference;

typedef OSType ISpDeviceClass;

typedef OSType ISpDeviceIdentifier;

typedef OSType ISpElementLabel;

typedef OSType ISpElementKind;
```

Device Definition Structure

```
typedef struct ISpDeviceDefinition {
    Str63                   deviceName;
    ISpDeviceClass          theDeviceClass;
    ISpDeviceIdentifier     theDeviceIdentifier;
    UInt32                  permanentId;
    UInt32                  flags;
    UInt32                  reserved1;
    UInt32                  reserved2;
    UInt32                  reserved3;
} ISpDeviceDefinition;
```

Element Information Structure

```
typedef struct ISpElementInfo {
    ISpElementLabel             theLabel;
    ISpElementKind              theKind;
    Str63                       theString;
    UInt32                      reserved1;
    UInt32                      reserved2;
} ISpElementInfo, *ISpElementInfoPtr;
```

Button Configuration Information Structure

```
typedef struct {
    UInt32                      id;
} ISpButtonConfigurationInfo;
```

Directional Pad Configuration Information Structure

```
typedef struct {
    UInt32                      id;
    Boolean                     fourWayPad;
} ISpDPadConfigurationInfo;
```

Axis Configuration Information Structure

```
typedef struct {
    Boolean                     SymetricAxis;
} ISpAxisConfigurationInfo;
```

Movement Data Structure

```
typedef struct ISpMovementData {
    UInt32                      xAxis;
    UInt32                      yAxis;
    UInt32                      direction;
};
```

Element Event Data Structure

```
typedef struct {
    AbsoluteTime            when;
    ISpElementReference     element;
    UInt32                  refCon;
    Uint32                  data;
} ISpElementEvent, *ISpElementEventPtr;
```

Need Structure

```
typedef struct ISpNeed {
    Str63                   name;
    short                   iconSuiteResourceId;
    ISpElementKind          theKind;
    ISpElementLabel         theLabel;
    UInt32                  flags;
    UInt32                  emulateHow;
```

```
    UInt32                  emulationData1;
    UInt32                  emulationData2;
} ISpNeed;
```

InputSprocket Functions

Configuring the Application

```
OSStatus ISpInit                        (UInt32 count,
                                         ISpNeed *needs,
                                         ISpElementReference *inReferences,
                                         OSType appCreatorCode,
                                         OSType subCreatorCode,
                                         UInt32 flags,
                                         short setListResourceID
                                         UInt32 version);

OSStatus ISpConfigure                   (ISpEventProcPtr inEventProcPtr);

OSStatus ISpStop                        (void);

OSStatus ISpSuspend                     (void);

OSStatus ISpResume                      (void);

NumVersion ISpGetVersion                (void);

OSStatus ISpElement_NewVirtual          (UInt32 dataSize,
                                         ISpElementReference *outElement;
                                         UInt32 flags);

OSStatus ISpElement_NewVirtualFromNeeds (UInt32 count,
                                         ISpNeed *needs,
                                         ISpElementReference *outElements
                                         UInt32 flags);

OSStatus ISpElement_DisposeVirtual      (UInt32 count,
                                         ISpElementReference
                                         *inElements);
```

```
OSStatus ISpDevices_Extract             (UInt32 inBufferCount,
                                         UInt32 *outCount,
                                         ISpDeviceReference *buffer);

OSStatus ISpDevices_ExtractByClass      (ISpDeviceClass theClass,
                                         UInt32 inBufferCount,
                                         UInt32 *outCount,
                                         ISpDeviceReference *buffer);

OSStatus ISpDevices_ExtractByIdentifier (ISpDeviceIdentifier theIdentifier,
                                         UInt32 inBufferCount,
                                         UInt32 *outCount,
                                         ISpDeviceReference *buffer);

OSStatus ISpDevices_Activate            (UInt32 inDeviceCount,
                                         ISpDeviceReference *inDevicesToActivate);

OSStatus ISpDevices_Deactivate          (UInt32 inDeviceCount,
                                         ISpDeviceReference *inDevicesToDeactivate);

OSStatus ISpDevice_IsActive             (ISpDeviceReference inDevice,
                                         Boolean *outIsActive);

OSStatus ISpDevice_GetDefinition        (const ISpDeviceReference inDevice,
                                         UInt32 buflen,
                                         ISpDeviceDefinition *outStruct);

OSStatus ISpDevice_GetElementList       (const ISpDeviceReference inDevice,
                                         ISpElementListReference *outElementList);

OSStatus ISpElement_GetDevice           (const ISpElementReference inElement,
                                         ISpDeviceReference *outDevice);

OSStatus ISpElement_GetGroup            (const ISpElementReference inElement,
                                         UInt32 *outGroup);

OSStatus ISpElement_GetInfo             (const ISpElementReference inElement,
                                         ISpElementInfoPtr outInfo);

OSStatus ISpElement_GetConfigurationInfo(const ISpElementReference inElement,
                                         UInt32 buflen, void *configInfo);
```

Obtaining Data From Elements

```
OSStatus ISpElement_GetSimpleState      (const ISpElementReference inElement,
                                         Uint32 *state);

OSStatus ISpElement_GetComplexState     (const ISpElementReference inElement,
                                         UInt32 buflen, void *state);

OSStatus ISpElement_GetNextEvent        (ElemenReference inElement,
                                         UInt32 bufSize,
                                         ISpElementEventPtr event,
                                         Boolean *wasEvent);

OSStatus ISpElementList_GetNextEvent    (ISpElementListReference inElementList,
                                         UInt32 bufSize,
                                         ISpElementEventPtr event,
                                         Boolean *wasEvent);

OSStatus ISpElement_Flush               (ISpElementReference inElement);

OSStatus ISpElementList_Flush           (ISpElementListReference inElementList);
```

Managing Element Lists

```
OSStatus ISpElementList_New             (UInt32 inCount,
                                         ISpElementReference *inElements,
                                         ISpElementListReference *outElementList
                                         UInt32 flags);

OSStatus ISpElementList_Dispose         (ISpElementListReference inElementList);

OSStatus ISpGetGlobalElementList        (ISpElementListReference *outElementList);

OSStatus ISpElementList_AddElements     (ISpElementListReference inElementList,
                                         UInt32 refCon,
                                         UInt32 count,
                                         ISpElementReference *newElements);

OSStatus ISpElementList_RemoveElements  (ISpElementListReference inElementList,
                                         UInt32 count,
                                         ISpElementReference *oldElement);
```

```
OSStatus ISpElementList_Extract         (ISpElementListReference inElementList,
                                         UInt32 inBufferCount,
                                         UInt32 *outCount,
                                         ISpElementReference *buffer);

OSStatus ISpElementList_ExtractByKind   (ISpElementListReference inElementList,
                                         ISpElementKind theKind,
                                         UInt32 inBufferCount,
                                         UInt32 *outCount,
                                         ISpElementReference *buffer);

OSStatus ISpElementList_ExtractByLabel  (ISpElementListReference inElementList,
                                         ISpElementLabel theLabel,
                                         UInt32 inBufferCount,
                                         UInt32 *outCount,
                                         ISpElementReference *buffer);
```

Result Codes

kISpInternalErr	–30420	Internal error.
kISpSystemListErr	–30421	Operation is not allowed a system-maintained element list.
kISpBufferToSmallErr	–30422	The buffer is too small.
kISpElementInListErr	–30423	The element is already in the element list.
kISpElementNotInListErr	–30424	The element is not in the element list.
kISpSystemInactiveErr	–30425	InputSprocket is currently inactive.
kISpDeviceInactiveErr	–30426	The device is currently inactive.
kISpSystemActiveErr	–30427	Input Sprocket is currently active.
kISpDeviceActiveErr	–30428	The device is currently active.
kISpListBusyErr	–30429	The element list is marked as busy.

NetSprocket

Contents

This chapter describes NetSprocket, the component of Apple Game Sprockets that your game can use to provide easy-to-implement, high performance data transmission between players. By implementing NetSprocket in your game, you can enable a player to play a game of chess via modem or join a game with dozens of others on the Internet.

Before reading this chapter, you should be generally familiar with Apple Game Sprockets, as described in the preface to the book. You should know the basic principals of networking before reading this chapter.

This chapter begins by describing the basic capabilities of NetSprocket. In the section "Using NetSprocket," you'll learn how to use some of these capabilities. The section "NetSprocket Reference" is where you'll find descriptions of the constants and data structures, along with complete explanations and descriptions of each function included in NetSprocket. The reference section is followed by "Summary of NetSprocket," which contains a comprehensive listing of the constants, data structures, and functions provided by NetSprocket.

Note
This chapter describes the application programming interface supported by versions 1.0 and later of NetSprocket. ◆

About NetSprocket

NetSprocket is the part of Apple Game Sprockets that provides networking capabilities in your application and is geared specifically for game developers.

Unlike the standard networking API provided in *Inside Macintosh* for more traditional applications, NetSprocket provides you with the functions you need to network your game without worrying about endpoints, listeners, function callbacks, or other low-level networking details.

Using the NetSprocket functions you can set up and host your game, respond to incoming messages, and send messages to other players. NetSprocket handles delivery and receipt of all game messages without any additional effort on your part. NetSprocket is implemented as a shared library and provides an application programming interface (API) that focuses on players, groups, and messages.

Players, Groups and Messages

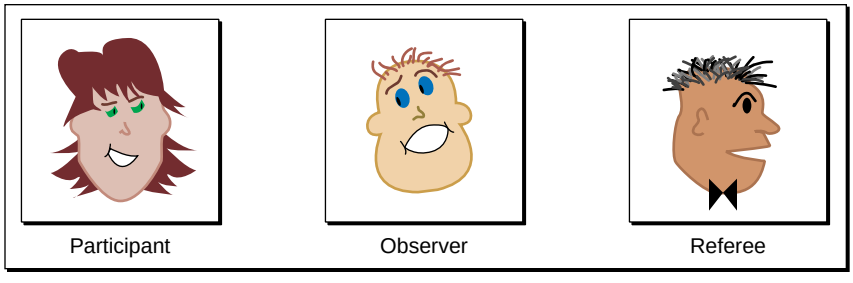
NetSprocket focuses on three distinct needs of game developers, organized just the way the game often is: by player, groups of players, and the messages that are sent and received in the course of a game.

Players

A **player** in NetSprocket is any person using your game. The primary element of information about a player is his numerical identification, or ID. NetSprocket—and your game—uses this ID to track all of the information you acquire about a player during the course of a game, or even over the course of several games if you choose. This information might include real names, aliases, passwords, a game history, a score history, or any other information pertinent to the player.

Anyone joining a game must be a player. However, a player may not actually be playing the game in the literal sense. Players often have different roles, as illustrated in Figure 4-1. For example, a player might be a judge or referee in a sports-style game. A player might also be a host, an observer, or even a spy.

Figure 4-1 Players in a game may have different roles



Groups

A **group** in NetSprocket is one or more players with a common theme. The primary element of information about a group is a numerical identification, or ID, just like a player. You use the group ID to manage the information you acquire about a group of players just as you do an individual player. This

information might include team names, positions, scores, or other information pertinent to the group.

A group is composed of players with a common theme. An obvious example is members of opposing teams, such as 5 players each in a basketball game, as shown in Figure 4-2. The referees might also be grouped together so you can send each of them a message easily. However, groups might not be static. In a search-and-rescue scenario, players may move from the lost group to the found group as your searchers locate survivors.

The primary purpose of groups is to allow you to send messages to a subset of the players in the game by sending one message to a group ID, rather than sending the message repeatedly, using the player ID for each member of the group.

Figure 4-2 Players are often in groups

Group of players

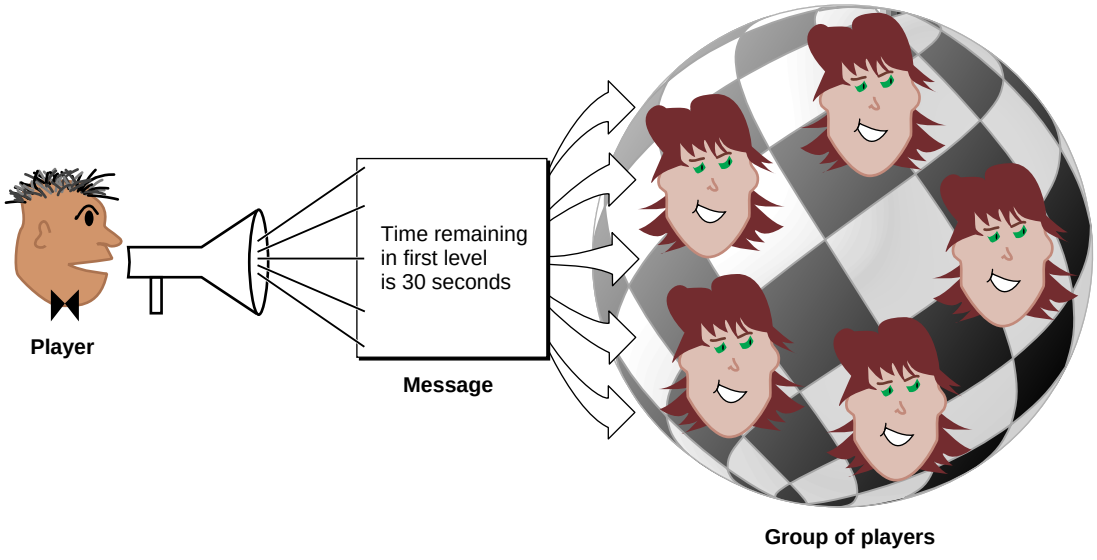
Messages

Information travels among players and groups in the form of **messages**. In fact, messages are the only way players can communicate with each other. Messages

can be sent from player to player (using the player ID) or from a player to a group of players (by using the group ID), as illustrated in Figure 4-3.

Messages can contain any data type or data structure. Messages are often characters and numbers, but they might also be graphics, sound or other complex data structures.

Figure 4-3 Players use messages to communicate with each other



Key Features of NetSprocket

NetSprocket focuses on key features that dramatically reduce the effort required to wire your game. In fact, most of the work required to implement networking is already done for you, and provides features that your players will benefit from each time they play, including fault-tolerance, speed and human interface elements, to name a few.

The following paragraphs describe in greater detail the capabilities provided by NetSprocket.

High Level Network Interface

NetSprocket provides a high level network interface to networking capabilities. This allows you to concentrate on creating the game itself, rather than the low-level details of complex networking issues. For example, a default human interface is provided for hosting a game or joining a game that is about to begin (or already in progress). Message routing is completely transparent to the application, and message delivery can be specified as guaranteed, or best-effort, depending on your needs. NetSprocket provides for dynamic addition and deletion of players and provides fault-tolerance as well.

Multiple Protocol Support

The library supports both AppleTalk and TCP/IP and can support future protocols without requiring any changes to your game. The messaging model allows each player to choose a different protocol and still participate in a game. Using this protocol-independent model, you can host a multiplayer game on the Internet and allow players using AppleTalk or TCP/IP to join the game. Even though a player that has joined the game may be using different protocols, the game application hosting the session will route all messages correctly without burdening the game itself.

Fault-tolerance

Fault-tolerance is a critical component of any network game architecture. If a player's game application crashes, or inexplicably disappears because of a network failure, NetSprocket will automatically alter its messaging tables and topology, recovering to the best of its ability without game intervention. While NetSprocket's ability to recover is dependent on many factors, it will make a best-effort approach to remedy the situation and inform the game host that a fault has occurred. The game host can then make a determination of whether to continue or halt the current game session.

Client/Server Topology

NetSprocket version 1.0 uses the client/server topology for its game experiences, the most versatile and generally the most efficient architecture for this type of application. NetSprocket employs efficient messaging and data management techniques to make sure the message is delivered to its intended recipients as quickly as possible.

Network Efficiency and Speed

Network efficiency and speed is a critical component of any game. The more time you spend sending and receiving messages, the less time you have to entertain your users by rendering scenes or updating the game world. The NetSprocket functions are written to quickly and efficiently process incoming and outgoing message, dispatching them to the network or the game as rapidly as possible.

Human Interface Elements

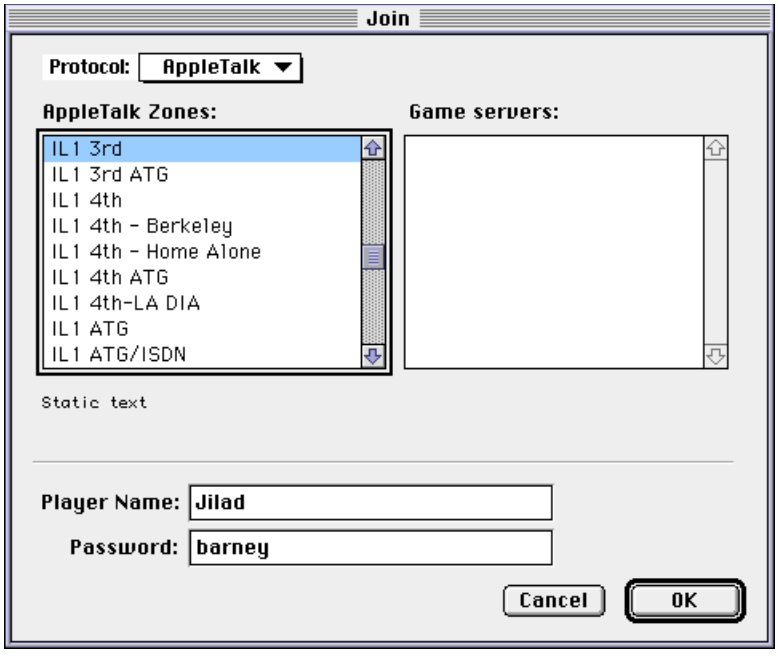
NetSprocket also provides some human interface elements. You can use these default dialog boxes to speed prototyping or include them as part of the final product if they meet your requirements. NetSprocket provides dialog boxes for hosting a game and dialog boxes for joining a game about to begin or already in progress on the network.

The dialog box for hosting a game allows the user to select which network protocols will be allowed in the game, as illustrated in Figure 4-4. Text edit fields are also provided for naming the game that is being hosted, along with the player's name. The user may also set a password required by anyone on the network interested in joining the game.

Figure 4-4 Modal dialog box for hosting a game

The dialog box for browsing an AppleTalk network and joining a game provides players with a browser, as shown in Figure 4-5. Using this dialog box, someone looking for a game server can browse the various zones, and select the game of his choice. Then, he can fill in his name and password and join the game.

Figure 4-5 Dialog box for browsing and joining games on AppleTalk



The dialog box for joining a game on a TCP/IP network or a serial connection provides text edit fields for entering the address, as shown in Figure 4-6. When players join or leave a game in progress, NetSprocket will automatically update its routing information and inform each player of entering and departing players.

Figure 4-6 Dialog box for joining a game on a TCP/IP network or serial connection

NetSprocket's human interface elements will continue to evolve and support more transport protocols over time, without requiring changes to your game. This means that your game may someday be played using a protocol or an online service that didn't even exist when your game was designed and written.

Using NetSprocket

This section illustrates some of the basic ways you can use NetSprocket. It provides source code samples of a sample game that shows how you can

- initialize NetSprocket for use in your game
- host a game (optionally using the default human interface)
- join a game

- send and receive messages
- end the game.

Note
The code examples shown in this section provide only very rudimentary error handling. ◆

Initializing NetSprocket

The NetSprocket shared library must be initialized by your application before you can use any NetSprocket functions. The library should be initialized once at application startup. Listing 4-1 shows you how to initialize NetSprocket.

Listing 4-1 Initializing NetSprocket

```
err = NSpInitialize(550, 20000, 100, 'nspx', 100);
if (err != noErr)  HandleError();
```

Note
Be sure to check for an error from this function. If NetSprocket returns an error, you may not call any other NetSprocket functions. ◆

Hosting a Game

There are several things you must do to host a game, as detailed in the section below.

Creating a Protocol Reference and Advertising Your Game

First, you may optionally create protocol references. NetSprocket uses opaque protocol references and lists in order to insure that your game will function properly with transport protocols that are added to NetSprocket after your game is published. Since AppleTalk and TCP/IP are available on every Macintosh computer, you can specify default information for these protocols. Listing 4-2 illustrates how you can preconfigure AppleTalk with a default game name as it will appear to people browsing the AppleTalk network and create the protocol reference.

You do not have to create protocol references if you are going to use the default NetSprocket human interface. You only need to specify them if you want to set defaults yourself.

Note
This is the default name only. This name can be modified by the user if you display the host dialog box prior to actually hosting the game on the network. ◆

Listing 4-2 Establishing the Game's Name and Creating a Protocol Reference

```
NSpProtocolReference atRef, ipRef;

// Do the dialog
CopyPStr(gameName,   "\pMyGame");
CopyPStr(gameType,   "\pExample10");

// Set up our protocol list
atRef = NSpProtocol_CreateAppleTalk(gameName, gameType, 0, 0);
```

Note
NetSprocket version 1.0 does not support the exclusion of players based on network speed or latency. ◆

You can also create a TCP/IP protocol reference, as shown in Listing 4-3.

Listing 4-3 Creating a TCP/IP Protocol Reference

```
ipRef = NSpProtocol_CreateIP(2002, 0, 0);
```

Note
There are a number of internet standards governing the assignment of ports. In general, ports 1-1023 are considered reserved, and should never be used by game developers. Ports 1024 and above are dynamic. Ports greater than 5000 are generally used for less-known services. Apple encourages you to choose a port greater than 5000 if you are going to assign a default port for your game. The port number is an unsigned 16 bit value from 0 to 65535.

Creating a Protocol List

The reason for creating and maintaining a protocol list is that NetSprocket can support multiple protocols in a single game. Whether or not you created any protocol references, you must create a protocol list. This protocol list will be passed to the `NSpDoModalHostDialog` function, or to the `NSpGame_Host` function. Listing 4-4 shows you how to create a protocol list.

Listing 4-4 Creating a Protocol List

```
NSpProtocolListReferenceprotocolList;

err = NSpProtocolList_New(NULL, &protocolList);
if (err != noErr)  HandleError();
```

Note
When you create a protocol list, you can optionally append a protocol reference by passing it as the first parameter. ◆

Adding Protocols to the Protocol List

When you have created your protocol list, you should add the protocols you support to your protocol list, as show in Listing 4-5.

Listing 4-5 Adding Supporting Protocols to a Protocol List

```
    err = NSpProtocolList_Append(protocolList, atRef);
    if (err != noErr)  HandleError();
```

Note
Once a protocol reference has been added to a protocol list, the list owns the reference. You should not attempt to free the reference in your game.

Presenting the Host Dialog Box

Once you've created your protocol list (and optionally set default information), you can present the default dialog box illustrated in Figure 4-4 for hosting a game by as shown in Listing 4-6.

Listing 4-6 Displaying the Host Dialog Box

```
    ok = NSpDoModalHostDialog(protocolList, gameName, playerName,
password, NULL);
    if (!ok)  return;
```

Using the Host Function

Now that you have obtained and set the necessary configuration information, you are ready to actually host your game. Listing 4-7 shows you how.

Listing 4-7 Hosting your game on the Network

```
    NSpGameReferencegMyGame;

    err = NSpGame_Host(&gMyGame, protocolList, 8, gameName, password,
playerName, 0, kNSpClientServer, 0);
    if (err != noErr)  HandleError();
```

Disposing of the Protocol List

When you have created the game reference, you can dispose of your protocol list. You do not need to dispose of each element you added to the list. You can just dispose of the list itself, as shown in Listing 4-8.

Listing 4-8 Disposing of your Protocol List

```
NSpProtocolList_Dispose(protocolList);
```

Joining A Game

The NSpDoModalJoinDialog function may be used to allow a player to browse an AppleTalk network and join a game. The dialog box provided for you in NetSProcket is shown in Figure 4-5.You may also develop your own join dialog box. Listing 4-9 shows you how to use the `NSpDoModalJoinDialog` function.

Listing 4-9 Calling the NSpDoModalJoinDialog function

```
CopyPStr(gameType,   "\pExample10");
address = NSpDoModalJoinDialog(gameType, "\pAvailable Games:",
playerName, password, NULL);
```

The actual process of joining a game is show in Listing 4-10. The address is the address reference returned by a call to the `NSpDoModalJoinDialog` function. Alternatively, you could get your own `OTAddress` through your own join dialog box, and then convert it into a NetSprocket address reference by calling `NSpConvertOTAddrToAddressReference`.

The player name, type, and password are sent to the host. If the host requires a password, and the password provided here is not correct, this function will succeed, but your first message you receive from NetSprocket will be a join denied message.

Listing 4-10 Joining the game

```
    err = NSpGame_Join(&gMyGame, address, playerName, password, 0, 0,
NULL, 0);
    if (err != noErr)  HandleError();
```

After executing the `NSpGame_Join` function, you should release the address reference, as shown in Listing 4-11

Listing 4-11 Releasing the address reference

```
        NSpReleaseAddressReference(ad.dress);
```

Receiving Messages From Other Players

Now that the game is in play, you will normally drop into your main event loop. In your event loop, you should send any messages you have, and receive any messages that may be pending. Listing 4-12 shows an example of how to receive message from other players.

Listing 4-12 sample game event loop

```
void DoHandleMessages(void)

{
    NSpMessageHeader*message;

    while ((message = NSpMessage_Get(gMyGame)) != NULL)
    {
        switch(message->what)
        {
            case kNSpJoinApproved:
                DoJoinApproved((NSpJoinApprovedMessage *) message);
            break;
            case kNSpPlayerJoined:
                DoPlayerJoined((NSpPlayerJoinedMessage*) message);
            break;
```

```
            case kNSpPlayerLeft:
                DoPlayerLeft((NSpPlayerLeftMessage*) message);
            break;

            case kPlayerLocation:
                DoPlayerLocation((PlayerLocationMessage*) message);
            break;
        }

        NSpMessage_Release(gMyGame, message);
    }
}
```

Sending Messages to Other Players

In the course of play, it is normal to send messages to other players. In Listing 4-13, you see an example of sending a message to everyone but yourself.

Listing 4-13 Sending a message to all players

```
    PlayerLocationMessagemessage;

    NSpClearMessageHeader(&message.header);

    message.header.what = kPlayerLocation;
    message.header.to = kNSpAllPlayers;

    message.header.messageLen = sizeof(PlayerLocationMessage);
    message.where = where;

    NSpMessage_Send(gMyGame, &message.header, kNSpSendFlag_Normal |
kNSpSendFlag_SelfSend);
```

Ending The Game

When it is time to end the game, you simply call the `NSpGame_Dispose` function, as shown in Listing 4-14. If you specify the `kNSpGameFlag_ForceTerminateGame`

flag when you call `NSpGame_Dispose`, the other players will be notified that the game is over, and it will be disposed. If you don't pass this flag, and you're the host, NetSprocket will attempt to negotiate a new host. If a new host cannot be found, the function will return an error and you should call it with the `kNSpGameFlag_ForceTerminateGame` flag if you want to force it to quit. If you're a client, the `kNSpGameFlag_ForceTerminateGame` is ignored

Listing 4-14 Terminating the game

```
        NSpGame_Dispose(gMyGame, kNSpGameFlag_ForceTerminateGame);

}
```

NetSprocket Reference

This section describes the constants, data structures, and functions provided by NetSprocket.

Constants

This section describes the constants provided by NetSprocket.

Network Message Priority Flags

These constants are used to identify various priorities you may assign to network messages using a mail service metaphor. You use these flags in the `NSpFlags` parameter of the `NSpMessage_Send` function (page 4-50).

```
enum {
    kNSpJunk                        = 0x10000000,
    kNSpNormal                      = 0x20000000,
    kNSpRegistered                  = 0x30000000
};
```

Constant descriptions

`kNSpJunk` This message is junk mail. This type of message will be sent only when no other messages of higher priority are pending. This is essentially a "fire and forget" message. Delivery will only be attempted once, and there is no guarantee of receipt.

`kNSpNormal` This message is an ordinary, every-day message. It will be sent immediately, but like `kNSpJunk`, delivery will only be attempted once, and there is no guarantee of receipt.

`kNSpRegistered` Like registered mail, this message is quite important. Delivery is of the highest priority. For example, if `kNSpNormal` or `kNSpJunk` messages are being sent (or if a message is being chunked for delivery in multiple packets), they will be interrupted in favor of a `kNSpRegistered` message. NetSprocket will demand proof of receipt and will continue retrying until the maximum retry limit has been exceeded.

Network Message Delivery Flags

These constants are message delivery flags to assist you in determining and controlling the status of message delivery. You can OR these constants together with the network message priority flags.

Note

A message that is successfully sent does not ensure receipt by the intended players unless `kNSpRegistered` is specified. It simply means that NetSprocket successfully delivered the message to the appropriate network protocol handler and the message has been duly passed on. ◆

```
enum {
    kNSpFailIfPipeFull              = 0x00000001,
    kNSpSelfSend                    = 0x00000002,
    kNSpBlocking                    = 0x00000004
};
```

Constant descriptions

`kNSpFailIfPipeFull` NetSprocket will not accept the network message you are attempting to send if there are too many messages pending in the output buffer. Use this if you want to send data that is extremely time critical and useless if not delivered immediately.

`kNSpSelfSend` This flag is used to instruct NetSprocket to send a copy of this message to yourself as a player in addition to any other players or groups it is addressed to. You will receive a copy of your message in the message queue. If you send a message to all players (`kNSpAllPlayers`) without setting this flag, NetSprocket will not deliver the message to the sender.

`kNSpBlocking` This flag is used to have NetSprocket block the call and not return until the message has been successfully sent. The combination of `kNSpBlocking` and `kNSpRegistered` may cause your application to wait a significant period of time before satisfying these requirements, because it will wait until all the recipients have acknowledged receipt of the message or the retry limit has been reached.

Note

In NetSprocket version 1.0, a message sent from any player who is not the host with this flag set will return when the message has been delivered to the host. The message may or may not have been received by all of the intended recipients. ◆

Options for Hosting, Joining, and Deleting Games

These constants are used to control games. You use these constants in the `inFlags` parameter of the `NSpGame_Host`, `NSpGame_Join`, and `NSpGame_Delete` functions.

```
enum {
    kNSpGameFlag_DontAdvertise      = 0x00000001,
    kNSpGameFlag_ForceTerminateGame = 0x00000002
};
```

`kNSpGameFlag_DontAdvertise`

When this flag is passed with `NSpGame_Host`, the game object is created, but the game is not advertised on any protocols. By default, a call to `NSpGame_Host` advertises the game on the protocols in the protocol list.

`kNSpGameFlag_ForceTerminateGame`

When the host calls `NSpGame_Delete` with this flag set, NetSprocket will end the game without attempting to find a host replacement. All the players will receive a message that the game has been ended, and any further calls from them will return an error. Normally, a call to `NSpGame_Delete` by the host will cause NetSprocket to negotiate a new host.

Network Message Types

These constants are used to identify standard message types. NetSprocket uses these types to clearly identify the network messages so you can process the message with the appropriate data structure.

```
enum {
    kNSpSystemMessagePrefix     = 0x80000000,
    kNSpError                   = kNSpSystemMessagePrefix | 0x7FFFFFFF,
    kNSpJoinRequest             = kNSpSystemMessagePrefix | 0x00000001,
    kNSpJoinApproved            = kNSpSystemMessagePrefix | 0x00000002,
    kNSpJoinDenied              = kNSpSystemMessagePrefix | 0x00000003,
    kNSpPlayerJoined            = kNSpSystemMessagePrefix | 0x00000004,
    kNSpPlayerLeft              = kNSpSystemMessagePrefix | 0x00000005,
    kNSpHostChanged             = kNSpSystemMessagePrefix | 0x00000006,
    kNSpGameTerminated          = kNSpSystemMessagePrefix | 0x00000007
};
```

Constant descriptions

`kNSpSystemMessagePrefix`

This is the prefix of all NetSprocket system messages. You can OR a message's `what` field with this constant to determine if the message is a system message.

`kNSpError`

A local error has occurred. It may have occurred when receiving a message, attempting to send a message, or attempting to allocate memory.

`kNSpJoinRequest`	A player wants to join a game. You do not need to respond to this message. NetSprocket will either use the default password check, or your custom join handler (if installed) to approve or deny the join request.
`kNSpJoinApproved`	Your request to join a game has been approved.
`kNSpJoinDenied`	Your request to join a game has been denied.
`kNSpPlayerJoined`	A player has joined the game.
`kNSpPlayerLeft`	A player has left the game.
`kNSpHostChanged`	The host of the game has changed.
`kNSpGameTerminated`	The game has been permanently stopped.

Note
All message types with negative values are reserved for use by Apple Computer, Inc. ◆

Reserved Player IDs for Network Messages

These constants are used to identify player IDs that are reserved for message delivery. Specify one of these special IDs in the `to` field of a message structure.

```
enum {
    kNSpAllPlayers                  = 0x00000000,
    kNSpServerOnly                  = 0xFFFFFFFF
};
```

Constant descriptions

`kNSpAllPlayers`	Send the message to all players.
`kNSpServerOnly`	Send the message to the player currently hosting the game.

Note
It is possible for the host to change during the course of a game. It is aso possible for a host to not have a player ID, because someone may host a game without participating as a player. Therefore you should not use a player ID to send a message to the host. Instead, you should use `kNSpServerOnly` reserved for a host. ◆

Topology Types

You use these constants to identify the topology you are choosing for your game. You pass this value in the `inTopology` field of `NSpGame_Host`.

```
typedef enum {
    kNSpClientServer                = 0x00000001
} NSpTopology;
```

Constant descriptions

`kNSpClientServer` Client/server topology.

Note
NetSprocket version 1.0 currently supports only client/server topology. ◆

Data Structures

This section describes the data structures provided by NetSprocket for use in your game. These structures define the players, groups, and message header structure, as well as information about the game.

Game Reference

You use the game reference to identify your game to the NetSprocket library.

```
typedef         struct          NSpGamePrivate
*NSpGameReference;
```

Field descriptions

`NSpGameReference` An opaque reference returned to you via a successful call to `NSpGame_Host` or `NSpGame_Join`. This reference is passed to most NetSprocket functions as the first parameter.

Protocol Reference

You use the protocol reference to identify and configure transport protocols without having to know what the protocol actually is.

```
typedef         struct          NSpProtocolPrivate
*NSpProtocolReference;
```

Field descriptions

`NSpProtocolReference`
: An opaque reference returned either from a call to `NspDoModalHostDialog` or by a call to `NSpProtocol_Create`. It is used to identify and configure various protocols, without requiring prior knowledge of the protocol.

Protocol List Reference

You use the protocol list reference as a reference to a list of protocol references.

```
typedef         struct          NSpListPrivate
*NSpProtocolListReference;
```

Field descriptions

`NSpProtocolListReference`
: An opaque reference to a list of protocol references. It is used because a user may request that a game be hosted on multiple protocols, within the `DoModalHostDialog` call. Use this data structure when a user requests that a game be hosted on multiple protocols using the `NSpDoModalDialog` function. The protocol reference list is passed to the `NSpGame_Host` function to tell NetSprocket which protocols the game is to be hosted (advertised) on.

Address Reference

You use the address reference as a reference to a protocol address.

```
typedef         struct          NSPAddressPrivate
*NSpAddressReference;
```

Field descriptions

`NSpAddressReference`
: An opaque reference to a protocol address. It is created

from a call to `NSpDoModalJoinDialog` or by converting an Open Transport address.

Player Information Structure

You use the player information structure to store information about each player in the game. It contains the player's ID, along with pertinent information about the player, including the groups he may belong to. The player information structure is defined by the `NSpPlayerInfo` data type.

```
typedef struct NSpPlayerInfo {
    NSpPlayerID                         id;
    NSpPlayerType                       type;
    Str31                               name;
    UInt32                              groupCount;
    NSpGroupID                          groups[kVariableLengthArray];
} NSpPlayerInfo, *NSpPlayerInfoPtr;
```

Field descriptions

`id`	A unique number for each player within a game. A player who leaves and re-enters a game will receive a new ID.
`type`	Reserved for the use by the game developer. This parameter is not used by NetSprocket, but may be used by you to classify players.
`name`	A user-readable Pascal string (maximum 31 characters).
`groupCount`	The number of groups the player is currently in.
`groups`	An array containing a list of the group IDs the player currently belongs to.

Player Enumeration Structure

You use the player enumeration structure to maintain a list of all the players currently in the game. It contains a count of the players, followed by pointers to each of the `playerInfo` structures. The player enumeration structure is defined by the `NSpPlayerEnumeration` data type.

```
typedef struct NSpPlayerEnumeration {
    UInt32                          count;
    NSpPlayerInfoPtr                playerInfo[kVariableLengthArray];
} NSpPlayerEnumeration, *NSpPlayerEnumerationPtr;
```

Field descriptions

`count`	The number of players (and player information structures) listed in `NSpPlayerEnumeration`.
`playerInfo`	An array of pointers to player information structures for each player in the game.

Group Information Structure

You use the group information structure to maintain information about each of the players in a group. It includes the number of players, along with an array of the player's ID. The group information structure is defined by the `NSpGroupInfo` data type.

```
typedef struct NSpGroupInfo {
    NSpGroupID                      id;
    UInt32                          playerCount;
    NSpPlayerID                     players[kVariableLengthArray];
} NSpGroupInfo, *NSpGroupInfoPtr;
```

Field descriptions

`id`	A unique number identifying the group.
`playerCount`	The number of players in the group.
`players`	An array of pointers to player information structures.

Group Enumeration Structure

You use the group enumeration structure to maintain a list of all the groups currently in the game. In addition to the number of groups currently in the game, it contains an array of pointers to the group information structures. The group enumeration structure is defined by the `NSpGroupEnumeration` data type.

```
typedef struct NSpGroupEnumeration {
    UInt32                          count;
    NSpGroupInfoPtr                 groups[kVariableLengthArray];
} NSpGroupEnumeration, *NSpGroupEnumerationPtr;
```

Field descriptions

`count`	The number of groups in the game.
`groups`	An array of pointers to group information structures.

Game Information Structure

Basic information about the game is organized in the game information structure. You can use this structure to maintain and obtain basic information about key elements in the game, including information about players, groups, and topology. The game information structure is defined by the `NSpGameInfo` data type.

```
typedef struct NSpGameInfo {
    UInt32                          maxPlayers;
    UInt32                          currentPlayers;
    UInt32                          currentGroups;
    NSpTopology                     topology;
    UInt32                          reserved;
    Str31                           name;
    Str31                           password;
} NSpGameInfo;
```

Field descriptions

`maxPlayers`	The maximum number of players allowed in the game, as specified in the `inMaxPlayers` parameter `NSpGame_Host` function. A value of 0 means there is no limit.
`currentPlayers`	The number of players currently participating in the game.
`currentGroups`	The number of groups in the game.
`topology`	A constant describing the topology of the network.
`reserved`	This field is reserved for future use in NetSprocket. Do not modify or rely upon any data in this field.
`name`	The text descriptor identifying the game.

password	The password required to join the game. A value of NULL means no password is required.

Message Header Structure

The most important structure in NetSprocket is the abstract message type. It is comprised of the NSpMessageHeader itself and is followed by custom data. The message header structure contains information about the nature of the message being delivered. The message header structure is defined by the NSpMessageHeader data type.

The fields of the message header are used by NetSprocket to deliver your message to the specified recipients. Before you send a network message, you should fill in the what, to, and messageLen parameters. NetSprocket will set the remaining parameters.

```
typedef struct NSpMessageHeader {
    UInt32                          version;
    SInt32                          what;
    NSpPlayerID                     from;
    NSpPlayerID                     to;
    UInt32                          id;
    UInt32                          when;
    UInt32                          messageLen;
} NSpMessageHeader;
```

Field descriptions

version	Private version information for NetSprocket. Do not modify or rely upon any data in this field.
what	A constant describing the network message type. You should set this field with the constants defined in NetSprocket or a network message type that you have defined in your application.
from	A read-only parameter, which NetSprocket sets to the player ID of the sender.
to	The ID of the intended recipient. Set this value to the ID of the intended player, the group ID of the intended group of recipients, kNSpAllPlayers to send it to every player in the game, or kNSpServerOnly to send it to the server.

`id`	This is a read-only parameter. The NetSprocket library will assign a unique ID to each message emanating from a given player. Thus, the `from` and `id` parameters make up a unique message identifier. This allows you to identify duplicate messages.
`when`	This is a read-only parameter. NetSprocket will place a time stamp in milliseconds here when the message is sent from its originator. When you receive a message, you can compare this field against the value returned by the `NSpGetCurrentTimeStamp` function to find out how long ago the message was sent. This value is only a relative value and is accurate only to about 30 to 60 milliseconds.
`messageLen`	Set this field to the size of your entire message structure (as specified in the `sizeof` parameter), including the header and any data that follows the header.

Note
Apple reserves all messages that have a negative value (anything with the high bit set to 1). You can define your own custom message types (for example, keyboard state, voice transmission, or game map). ◆

Error Message Structure

The error message structure is a standard NetSprocket message you receive when extreme error conditions occur in NetSprocket. This can occur when functions fail or when network failures among players are detected by NetSprocket in the course of the game. The error message structure is defined by the `NSpErrorMessage` data type.

```
typedef struct NSpErrorMessage {
    NSpMessageHeader                    header;
    OSStatus                            error;
} NSpErrorMessage;
```

Field descriptions

`header`	A valid game object.
`error`	A constant of `OSStatus` type describing the error encountered.

Join Request Message Structure

The join request message structure is a standard NetSprocket network message you can use to notify the host that a player wishes to join a game about to start or one that is in progress. This structure will only be passed to your application if you install a custom join request handler. You will not get this structure via the `NSpMessage_Get` function. See the `NSpInstallJoinRequestHandler` function on (page 4-48) for more information. The join request message structure is defined by the `NSpJoinRequestMessage` data type.

This is an example of how you can construct your own message structures. Each structure is made using the message header structure followed by any custom data. Be sure to set the `messageLen` field in the header of the record to the size of your structure. ◆

```
typedef struct NSpJoinRequestMessage {
    NSpMessageHeader                    header;
    Str31                               name;
    Str31                               password;
    UInt32                              type;
    UInt32                              customDataLen;
    UInt8                               customData[kVariableLengthArray];
} NSpJoinRequestMessage;
```

Field descriptions

`header`	A valid game object.
`name`	The name of a prospective player.
`password`	A string being passed by the prospective player to attempt to match the password required by the host.
`type`	The type of the prospective player.
`customDataLen`	The length of the custom data passed by the game attempting to join.
`customData`	Data that was passed to a call to the `NSpGame_Join` function made by the prospective player. This is not used by NetSprocket.

Join Approved Message Structure

As a host, you can use the join approved message structure to send a message to the player who has been granted entry to the game. This is an advisory

message; there are no additional information fields. The join approved message structure is defined by the `NSpJoinApprovedMessage` data type.

```
typedef struct NSpJoinApprovedMessage {
    NSpMessageHeader                    header;
} NSpJoinApprovedMessage;
```

Field descriptions

`header` A valid game object.

Join Denied Message Structure

As a host, you can send the join denied message structure to a prospective player who has been denied entry into the game. If a request to join a game is denied, subsequent calls by the player attempting to join the game will return an error from NetSprocket. The game object should be deleted when a join request is denied. The join denied message structure is defined by the `NSpJoinDeniedMessage` data type.

```
typedef struct NSpJoinDeniedMessage {
    NSpMessageHeader                    header;
    Str255                              reason;
} NSpJoinDeniedMessage;
```

`header` A valid game object.

`reason` A string indicating the explanation for refusing entry into the game.

Player Joined Message Structure

The player joined message structure is used to send a message to all players in the game to notify them that a player has joined a game. It includes an updated count of players and the new player's data structure. The player joined message structure is defined by the `NSpPlayerJoinedMessage` data type.

```
typedef struct NSpPlayerJoinedMessage {
    NSpMessageHeader                    header;
    UInt32                              playerCount;
    NSpPlayerInfo                       playerInfo;
} NSpPlayerJoinedMessage;
```

Field descriptions

`header`	A valid game object.
`playerCount`	The number of players in the game.
`playerInfo`	Player information structure.

Player Left Message Structure

The player left message structure is used to send a message to all players when a player leaves a game. It includes the updated count of players and the ID of the player who has departed. The player left message structure is defined by the `NSpPlayerLeftMessage` data type.

```
typedef struct NSpPlayerLeftMessage {
    NSpMessageHeader                    header;
    UInt32                              playerCount;
    NSpPlayerID                         playerID;
} NSpPlayerLeftMessage;
```

Field descriptions

`header`	A valid game object.
`playerCount`	The number of players left in the game.
`playerID`	A valid player ID.

Host Changed Message Structure

NetSprocket uses the host changed message structure to send a message when the host of a game in progress has been changed. The host changed message structure is defined by the `NSpHostChangedMessage` data type.

```
typedef struct NSpHostChangedMessage {
    NSpMessageHeader                    header;
    NSpPlayerID                         newHost;
} NSpHostChangedMessage;
```

Field descriptions

`header`	A valid game object.
`newHost`	The player ID of the new host.

Game Terminated Message Structure

NetSprocket uses the game terminated message structure to send a message to all players when a game in progress has ended. This is an advisory message that contains no additional information. The game terminated message structure is defined by the `NSpGameTerminatedMessage` data type.

```
typedef struct NSpGameTerminatedMessage {
    NSpMessageHeader                    header;
} NSpGameTerminatedMessage;
```

Field descriptions

`header`	A valid game object.

NetSprocket Functions

This section describes the functions provided by NetSprocket. They are organized according to functionality.

Initializing NetSprocket

You use the functions in this section to initialize NetSprocket and for installing and using callback functions in your application.

NSpInitialize

You can use the `NSpInitialize` function to initialize the NetSprocket library.

```
OSStatus NSpInitialize (
                    UInt32 inStandardMessageSize,
                    UInt32 inBufferSize,
                    UInt32 inQElements,
                    NSpGameID inGameID,
                    UInt32 inTimeout);
```

`inStandardMessageSize`
This value is the maximum size (in bytes) of each message you expect to send regularly. For example, if your game is sending keyboard state most of the time and the keyboard state message is 40 bytes long (including the `NSpMessageHeader`) then you should set this value to 40. NetSprocket uses this value to optimize the message receipt buffers. Typically, games send messages that are either relatively constant in size, or the size of the message is proportional to the number of players. If your game doesn't have a typical message size, or if you want NetSprocket to choose a size for you, set this parameter to 0. Setting this value greater than 586 bytes while using AppleTalk may force NetSprocket to use multiple packets for sending the message and potentially decrease game performance.

`inBufferSize` The number of bytes that NetSprocket will allocate in its interrupt-safe memory pool for networking during initialization. Usually, 200 KB or more is recommended for most games. You can approximate the networking pool with this formula: ((size of standard message * (send frequency or get frequency)) * max players) + fudge factor of 50 KB). NetSprocket cannot allocate memory at interrupt time. If you do not plan to call NetSprocket functions at interrupt time or use the asynchronous functions, or if you want NetSprocket to allocate the default amount (currently 400 KB), set this value to 0. Because NetSprocket is unable to grow its buffer after initialization, it is important to allocate enough memory in NetSprocket to send, receive, and queue the messages your game will be using.

`inQElements` The maximum number of queue elements that NetSprocket will allocate. The queue elements are used to store messages until you receive them from NetSprocket. If you get messages frequently from NetSprocket, you can allocate fewer queue elements than if you do not get them frequently. NetSprocket can automatically expand its message queue, if necessary, but this will degrade performance. Specifying a small number (< 10) will use less memory, but may cause messages to be discarded due to lack of buffer space. Specifying a larger number (> 20) will allow you to call `NSpMessage_Get` less often and more efficiently. If you do not specify the size of a standard message, you should use the equation described for the `inBufferSize` parameter.

`inGameID` A unique identifier for your game. For instance, if you choose not to specify an NBP type to NetSprocket when registering a game on an AppleTalk network, it will use this ID. You should use your application's creator ID.

`inTimeout` The default time-out value you want NetSprocket to use when sending guaranteed messages and performing other network tasks that may require a time-out. Some NetSprocket functions require NetSprocket to keep attempting some action until a response is received. In order to prevent NetSprocket from waiting forever, it uses this value to determine when it should stop trying. If you pass 0 here, NetSprocket will use default timeouts, which vary depending on the operation.

function result A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

NetSprocket must be initialized once in your application before making function calls from the NetSprocket library.

This function may fail under a variety of circumstances, including the failure to allocate enough application memory, insufficient system memory, or failure to initialize networking in the Mac OS.

Your Callback Function

You can define a callback function that NetSprocket will call for various asynchronous events. You do not need to define a callback function unless you plan to use certain advanced features of NetSprocket.

```
typedef pascal void (*NSpCallbackProcPtr) (
                    NSpGameReference inGame,
                    void *inContext,
                    NSpEventCode inCode,
                    OSStatus inStatus,
                    void* inCookie);
```

`inGame`	An opaque reference to the game object making the callback.
`inContext`	A pointer that you passed in to the installation function.
`inCode`	A value describing what kind of event is being passed to your generic callback function when the callback is made.
`inStatus`	A status code containing `noErr`, a NetSprocket error, or an Open Transport error.
`inCookie`	A pointer that may be `NULL` or point to certain extra data for certain kinds of events.

DESCRIPTION

NetSprocket 1.0 does not make any generic asynchronous callbacks.

NSpInstallCallbackHandler

You can use the `NSpInstallCallbackHandler` function to install a generic callback handler for using NetSprocket in asynchronous mode.

```
OSStatus NSpInstallCallbackHandler (
                    NSpCallbackProcPtr inHandler,
                    void *inContext);
```

`inHandler`	A pointer to your callback function.

`inContext`	A context pointer for your use. This is passed back to your callback function.
function result	A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

This is an advanced function, and should not be used unless you plan to handle many of the complex networking details that NetSprocket would normally handle for your application.

Human Interface Functions

NetSprocket provides human interface functions you can use to speed prototyping and to allow players to host and join games. Use the `NSpDoModalHostDialog` function when you want to start a new game and enable others to join it. Use the `NSpDoModalJoinDialog` function when you want to join a game that is either in progress or waiting to begin.

The NetSprocket human interface functions are forward-compatible with new protocols as they become available. This means that you don't have to change your code to accommodate new protocols when joining or hosting a game.

NSpDoModalJoinDialog

You can use the `NSpDoModalJoinDialog` function to present the user with a default dialog box for finding and joining a game advertised on the network.

```
NSpAddressReference NSpDoModalJoinDialog (
                    ConstStr31Param inGameType,
                    ConstStr31Param inEntityListLabel,
                    Str31 ioName,
                    Str31 ioPassword,
                    NSpEventProcPtr inEventProcPtr);
```

`inGameType`	A Pascal (maximum 31 characters) string used to register your game's NBP (Name Binding Protocol) type. This must be the same as the one used to host a game.

`inEntityListLabel`
A Pascal string that will be displayed above the entity list in the AppleTalk panel of the dialog box, as a label for the list of available games.

`ioName` A Pascal (maximum 31 characters) string of the user name (generally from a preferences setting). Pass an empty string (not `NULL`) if you do not want a default name displayed in the dialog box. This field will contain any changes the user made to the name on return.

`ioPassword` A Pascal (maximum 31 characters) string of the password (generally from a preferences setting). Pass an empty string (not `NULL`) if you do not want a default password displayed in the dialog box. This field will contain the any changes the user made to the password on return.

`inEventProcPtr`
A pointer to `DialogProcUPP`, the dialog filter function for handling Mac OS events that may affect other windows you have displayed on the screen concurrently. Pass `NULL` if you do not need to receive Mac OS events while the NetSprocket dialog box is being displayed.

function result An opaque reference to a protocol address selected by the user.

DESCRIPTION

If the user cancels the dialog box, the function will return `NULL`. If the user selects OK, it will return a reference to the protocol address of a game host. This reference should then be passed to `NSpGame_Join`. Once you have called `NSpGame_Join`, call `NSpReleaseAddressReference` to release the reference.

NSpDoModalHostDialog

You can use the `NSpDoModalHostDialog` function to present your user with a default modal dialog box on for hosting a game on the network.

```
Boolean NSpDoModalHostDialog (
                    NSpProtocolListReference ioProtocolList,
                    Str31 ioGameName,
                    Str31 ioPlayerName,
                    Str31 ioPassword,
                    NSpEventProcPtr inEventProcPtr);
```

`ioProtocolList`

An opaque reference to a list of protocols. You can create an empty list that will be filled in with information about the protocols the user selects, but you cannot pass `NULL`. If you wish to preconfigure certain protocols, you can create protocol references for them, then add them to your protocol list before passing it to this function.

`ioGameName` A Pascal string (maximum 31 characters) of the name of the game to be registered in NBP and displayed to users if you are using the `NSpGame_Join` function in their game. Pass an empty string (not `NULL`) if you don't want to display a default game name. The value of `ioGameName` is often obtained from a preferences setting. This field contains changes (if any) the user made to the `ioGameName` field.

`ioPlayerName` A Pascal (maximum 31 characters) string of the user name (generally from a preferences setting). Pass an empty string (not `NULL`) if you do not want a default name displayed in the dialog box. This field contains any changes the user has made to the name.

`ioPassword` A Pascal (maximum 31 characters) string of the password (generally from a preferences setting). Pass an empty string (not `NULL`) if you do not want a default password displayed in the dialog box. This field contains any changes the user made to the password.

`inEventProcPtr`

A pointer to `DialogProcUPP`, the dialog filter function for handling Mac OS events that may affect other windows you

have displayed on the screen concurrently. Pass `NULL` if you do not need to receive Mac OS events while the dialog box is being displayed.

function result A value of `true` if the user selected OK, `false` if the user selected Cancel.

DESCRIPTION

This function fills in the protocol list with the protocol(s) the user has selected and configures the protocol references in the list with the proper information. If the user did not cancel the dialog box, you should then pass the protocol list to the `NSpGame_Host` function.

Hosting and Joining a Game

You can use the NetSprocket functions in this section to create and manage your game. This includes both hosting and joining games on a network using various protocols and instantiating custom network message handlers as required by your game.

NSpGame_Host

You can use the `NSpGame_Host` function to create a new game object. You use this function when you are preparing to host a game.

```
OSStatus NSpGame_Host (
                    NSpGameReference *outGame,
                    NSpProtocolListReference inProtocolList,
                    UInt32 inMaxPlayers,
                    ConstStr31Param inGameName,
                    ConstStr31Param inPassword,
                    ConstStr31Param inPlayerName,
                    NSpPlayerType inPlayerType,
                    NSpTopology inTopology,
                    NSpFlags inFlags);
```

`outGame` The address of a game reference which will be filled in by this function. Upon successful return, it will contain a pointer to the newly created game object. This field is invalid if the function returns anything other than `noErr`.

`inProtocolList` An opaque reference to a list of protocols that has been returned from `DoModalHostDialog`, or created by you in your own application for advertising your game on the network.

`inMaxPlayers` The maximum number of players permitted to join the game. If you want to allow unlimited players, set this value to 0. NetSprocket is more efficient when the maximum number of players is set in the `inMaxPlayers` field. The number of allowed groups does not affect the maximum number of players.

`inGameName` A Pascal string containing the name of the game that will appear in game browsers. You must pass a valid Pascal string in this field.

`inPassword` The value that prospective players must match to join the game. Players who do not enter a correct password will not be allowed to join. Use `NULL` if you do require a password for players joining your game.

`inPlayerName` The name of the player using the application hosting the game. If there is no player using the application hosting the game, this value should be set to `NULL`. If the value is set to `NULL`, then no player will be created on the application hosting the game.

`inPlayerType` The player type, which is used only if there is a player using the application hosting the game. This parameter is stored in NetSprocket's player information table and may be used by the game application. It is not used by NetSprocket.

`inTopology` A constant indicating the topology to use in the game. The only topology implemented in version 1.0 is client/server, indicated by the constant `kNSpClientServer`.

`inFlags` Options for creating the new game object. The only permissible value of `inFlags` in NetSprocket version 1.0 is `kNSpGameFlag_DontAdvertise`, which causes the `NSpGame_Host` function to create a game object, but not actually advertise the game on the network.

function result A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

Once the game is created, the game will automatically be advertised on the protocols in the list.

When you have created a game object by calling `NSpGame_Host`, you will pass the game object to other host functions you call. Do not use this function when you plan to join a game. If you are joining a game, use the `NSpGame_Join` function.

`NSpGame_Host` will return `noErr` upon successful completion, placing the new game object in the `outGame` parameter. If the game could not be created for some reason, the `NSpGameReference` will be invalid (`NULL`). You should check the result code and determine the appropriate course of action.

NSpGame_Join

You can use the `NSpGame_Join` function to join a game.

```
OSStatus NSpGame_Join (
                    NSpGameReference *outGame,
                    NSpAddressReference inAddress,
                    ConstStr31Param inName,
                    ConstStr31Param inPassword,
                    NSpPlayerType inType,
                    Uint32 inUserDataLen,
                    void *inUserData,
                    NSpFlags inFlags);
```

`outGame` A pointer to a game reference structure that is filled in by the function. You must provide a pointer to an `NSpGameReference` in the `outGame` parameter. This pointer will be filled in with a valid `NSpGameReference` on exit.

`inAddress` A valid address reference returned from the `NSpDoModalJoinDialog` function or created by the application.

`inName`	The player's name as it will appear to other players in the game. You must pass a valid Pascal string. `NULL` is not permitted in this field.
`inPassword`	The password entered by the user to join the game. Pass `NULL` or an empty string if no password is required.
`inType`	The player's type. This value is for your own use. It is stored, but not used by NetSprocket.
`inUserDataLen`	The length of custom authentication data being sent to the host as part of the join request. If your game does not use a custom authentication mechanism, you must set the value to 0.
`inUserData`	A pointer to your custom data being sent to the host for use by your custom authentication function you have installed to override the standard password mechanism offered by NetSprocket. This parameter is passed to the host, but not used by NetSprocket. If your game does not use a custom authentication mechanism, you should set this value to `NULL`.
`inFlags`	Options for joining the game. There are no options for this field in NetSprocket version 1.0.
function result	A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

This function joins the game specified by the `inAddress` parameter. You can obtain an address reference from `NSpDoModalJoinDialog` or `NSpConvertOTAddrToAddressReference`.

NSpGame_EnableAdvertising

You can use the `NSpGame_EnableAdvertising` function to enable or disable advertising.

```
OSStatus NSpGame_EnableAdvertising (
                    NSpGameReference inGame,
                    NSpProtocolReference inProtocol,
                    Boolean inEnable);
```

`inGame`	An opaque reference to your game object.
`inProtocol`	An opaque reference to a protocol for which you wish to stop advertising, or `NULL` to stop advertising on all protocols.
`inEnable`	A value of `true` to start advertising or `false` to stop advertising.
function result	A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

You should set the `inEnable` parameter to `true` or `false`, depending on whether you want to start advertising or stop advertising. The `NSpGame_Host` function automatically advertises the game, unless you passed `kNSpGameFlag_DontAdvertise` in its `inFlags` field.

NSpGame_Delete

You can use the `NSpGame_Delete` function when you want to leave the game.

```
OSStatus NSpGame_Delete (NSpGameReference inGame,
                    NSpFlags inFlags);
```

`inGame`	An opaque reference to your game object.
`inFlags`	Options for leaving the game.
function result	A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

If your application is hosting the game and you pass `kNSpGameFlag_ForceTerminateGame` in the `inFlags` parameter, the game will be stopped for all participants and the game object will be deleted. However, if you do not pass `kNSpGameFlag_ForceTerminateGame`, NetSprocket will attempt to negotiate with another player to become the host. If the negotiation is successful, the other players will be notified that the host has changed and you will be dropped from the game. If the negotiation fails, `NSpGame_Delete` returns an error and no further action is taken.

If your application is operating as a player (created by `NSpGame_Join`), the other players are notified that you are leaving the game. The game is not terminated if you make this call as a player.

NSpJoinRequestHandlerProcPtr

This is a function that you as the game developer must provide if you are going to provide a custom join request handler. You must write the function as defined below. Once you have installed your join request handler, it will be called whenever a new player wishes to enter the game. Your function must return true or false, telling NetSprocket whether or not to admit the prospective player.

```
typedef pascal Boolean (*NSpJoinRequestHandlerProcPtr) (
                    NSpGameReference inGame,
                    NSpJoinRequestMessage *inMessage,
                    void* inContext,
                    Str255 outReason);
```

`inGame` An opaque reference to the game object that received the join request.

`inMessage` A pointer to the join request message. This is data passed to your function by NetSprocket. It will contain the name, password, and any custom data that your game specifies.

`inContext` The context pointer you passed to NetSprocket when you first installed the join request handler.

`outReason`	A pointer to a Pascal string that NetSprocket will allocate for you. You can use this string to send textual information to a player. For example, if you are going to deny a join request, you may send your reason for denial into `outReason`.
function result	A value of `true` to inform NetSprocket to allow the prospective player into the game, or `false` to deny entry based on the criteria you have established.

DESCRIPTION

The purpose of the custom function is to provide the game developer more flexibility in controlling access to the game, rather than having NetSprocket allow players to join the game based on the password and minimum round-trip time of the prospective player. For example, you may want to restrict play to a particular network zone. Also, you may decide that certain levels of games may be played only by players with a previous score history.

Note
Before calling your request handler, NetSprocket will always make two checks for a prospective player. First, it will make sure that the prospective player's round-trip time meets your minimum requirements, if you have specified any. Second, it will make sure that allowing this player into the game will not exceed your maximum player count.You should not release the message passed to this function. ◆

NSpInstallJoinRequestHandler

You can use the `NSpInstallJoinRequestHandler` function to install a special function to process join requests for your game object. When your custom function is installed, NetSprocket will call this function when a join request comes in, rather than using the `NSpGame_Join` function. You do not need to

develop and install custom join request handlers if the NetSprocket functions meet your requirements.

```
OSStatus NSpInstallJoinRequestHandler (
                    NSpJoinRequestHandlerProcPtr inHandler,
                    void *inContext);
```

`inHandler` A pointer to your join request function.

`inContext` A pointer that will be passed to your handler when it is called by NetSprocket.

function result A result code of `noErr`, or a NetSprocket result code.

DESCRIPTION

You can install a custom join request handler to override the standard authentication method of NetSprocket. By default, when a NetSprocket host receives a join request, it will first make sure that the maximum number of players has not been exceeded. Then, it will check the prospective player's password (if required) and admit the player if the password matches.

When you override this behavior, your join request function is called and passed the `NSpJoinRequestMessage`. You must decide whether or not to allow the player to join, based on whatever criteria you desire. Your function must return a Boolean telling NetSprocket whether or not to allow the player to join the game.

After your custom join request handler has been installed, any subsequent join requests will be passed to this function for processing.

Note
Since the maximum round-trip time is specified when hosting a game, requests from prospective players who do not meet the maximum criterion will not be passed to your game. ◆

NetSprocket returns an error if there was a problem installing the handler.

Sending and Receiving Messages

You may use these NetSprocket functions to manage the messages being sent and received by your game. You should construct your messages based on the

message header structure and pass them to `NSpMessage_Send` for delivery to their intended recipients.

NSpMessage_Send

You can use the `NSpMessage_Send` function to deliver a message to other players in the game.

```
OSStatus NSpMessage_Send (
                    NSpGameReference inGame,
                    NSpMessageHeader *inMessage,
                    NSpFlags inFlags);
```

`inGame` An opaque reference to your game object.

`inMessage` A pointer to the message you want to deliver. This structure can contain any data your game requires, provided that it begins with a `NSpMessageHeader`. The header must contain valid information about the intended recipient and the size of the message.

Note
To impose a reasonable amount of type-safety, you must pass `&myStruct.header` to ensure the structure contains an `NSpMessageHeader` as its first element. ◆

`inFlags` Flags that specify how the message should be sent, as specified in the message header structure.

function result A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

Although there is no restriction on the size of your message, extremely large messages (about 50 percent of the memory allocated to NetSprocket at initialization) may not be delivered if the receiver lacks the memory to process your message.

NetSprocket will return an error if it was unable to deliver your message.

Note
`NSpMessage_Send` may return `NoErr`, even though the intended recipient did not receive the message. Depending on the options you have chosen and other network conditions beyond the knowledge or control of the application, the message may not have been received by its intended recipients. ◆

NSpMessage_Get

You can use the `NSpMessage_Get` function to receive messages that have been delivered to your game. You can use this function to retrieve and process messages whether you are a player in the game or you are hosting a game.

```
NSpMessageHeader *NSpMessage_Get (
                    NSpGameReference inGame);
```

`inGame` An opaque reference to your game object.

function result A pointer to your incoming message data structure.

DESCRIPTION

Once game play has begun, you will probably want to call this function each time you pass through your game loop to process all network messages as quickly and efficiently as possible.

`NSpMessage_Get` returns `NULL` if there are no messages pending. If a message has been received, NetSprocket will return a pointer to a message structure.

`NSpMessage_Get` returns a pointer to an `NSpMessageHeader`-based structure that is allocated by NetSprocket. You should call `NSpMessage_Release` to release the memory back to NetSprocket when you're done with the message. Failure to release memory in a timely fashion will limit NetSprocket's ability to handle more incoming messages. `NSpMessage_Get` and `NSpMessage_Release` are a more efficient method of message processing than the time-consuming process of copying incoming messages from NetSprocket into your application's message buffer.

Error messages such as `kNSpError` may also be returned by NetSprocket.

You should call `NSpMessage_Get` as frequently as you can to get messages that have been sent to your player.

NSpMessage_Release

You can use the `NSpMessage_Release` function to release a message that has been received via a call to `NSpMessage_Get`.

```
void NSpMessage_Release (
                    NSpGameReference inGame,
                    NSpMessageHeader *inMessage);
```

`inGame` An opaque reference to your game object.

`inMessage` A pointer to the message to be released.

DESCRIPTION

When you have finished processing a message you have received, you should call `NSpMessage_Release` to release the memory being used by NetSprocket.

NSpMessageHandlerProcPtr

You can define a function of this type for NetSprocket to call asynchronously when a message arrives.

```
typedef pascal Boolean (*NSpMessageHandlerProcPtr) (
                    NSpGameReference inGame,
                    NSpMessageHeader *inMessage,
                    void* inContext);
```

`inGame` An opaque reference to the game object that received the message.

`inMessage` A pointer to the message.

`inContext` The context pointer you passed in when you installed the handler.

DESCRIPTION

Your function must handle the message and return as quickly as possible. You should not free the message, as it will be automatically freed when your function returns. If you return `true`, then NetSprocket will put the message back into the incoming message queue. When you call the `NSpMessage_Get` function you will receive the message again. If you return `false`, the message will be deleted when your function returns. As an example, if you receive a message and you want to change part of the message or add to it, you can make a note in the message and then receive it again with the note added to the message. You can also use this as a mechanism for time stamping messages and only act on the latest messages.

You do not need to define a function of this type if you use NetSprocket in the normal event-loop mode.

Your handler must obey all the rules of interrupt-safe functions.

NSpInstallAsyncMessageHandler

You can use the `NSpInstallAsyncMessageHandler` function to install a message handler for your game object and put the messaging mechanism into asynchronous mode. You do not need to install a message handler, unless you want NetSprocket to call your handler function back as soon as a completed message has arrived.

```
OSStatus NSpInstallAsyncMessageHandler (
                    NSpMessageHandlerProcPtr inHandler,
                    void *inContext);
```

`inHandler` A pointer to your message handling function.

`inContext` A pointer that will be passed back to your handler when it is called by NetSprocket.

function result A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

Your message handler should be in place and ready to receive messages before this function returns. NetSprocket returns an error if there was a problem installing the handler.

Managing Network Protocols

In order to be protocol-independent and forward-compatible with new protocols, NetSprocket uses opaque protocol references. You can use the following functions to define and create protocol references for use by the `NSpGame_Host` function in your game, rather than having NetSprocket maintain them for you.

NSpProtocol_Create

You can use the `NSpProtocol_Create` function to create a new protocol reference from a definition string.

```
OSStatus NSpProtocol_Create (
                    const char* inDefinitionString,
                    NSpProtocolReference *outReference);
```

`inDefinitionString`
A string defining which protocol to use and what values to set for various configuration options.

`outReference` An opaque reference to the protocol created. Valid only if the function returns `noErr`.

function result A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

NetSprocket version 1.0 does not allow you to call the `NSpProtocol_Create` function directly. Instead, you should use the helper functions defined in this section.

NSpProtocol_Delete

You can use the `NSpProtocol_Delete` function to delete a protocol reference.

```
void NSpProtocol_Delete  (
                    NSpProtocolReference inProtocolRef);
```

`inProtocolRef` An opaque reference to the protocol being deleted.

DESCRIPTION

You should use this function to delete a protocol reference you have created with `NSpProtocol_Create`. Once you have added a protocol reference to a protocol list, the list owns the memory associated with the protocol reference and will delete it when the list is deleted.

NSpProtocol_ExtractDefinitionString

You can use the `NSpProtocol_ExtractDefinitionString` function to copy the definition string of the given protocol into the provided buffer.

```
OSStatus NSpProtocol_ExtractDefinitionString (
                    NSpProtocolReference inProtocolRef,
                    char *outDefinitionString);
```

`inProtocolRef` An opaque reference to the protocol for which you want the definition string.

`outDefinitionString`
A buffer you allocate into which the string is copied. Use the constant `kNSpMaxDefinitionStringLen` to allocate the memory.

function result A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

You can extract the definition string to clone the protocol reference or modify it for use when you create a new protocol reference. Even though this function is

implemented in NetSprocket version 1.0, it does not support the creation of protocols with definition strings directly.

NSpProtocolList_Create

You can use the `NSpProtocolList_Create` function to create a new list for storing multiple protocol references.

```
OSStatus NSpProtocolList_Create (
                    NSpProtocolReference inProtocolRef,
                    NSpProtocolListReference *outList);
```

`inProtocolRef` An opaque reference to the protocol reference to be added to the list when it is created. Pass `NULL` if you don't want to add any protocol references at this time.

`outList` An opaque reference to the protocol list that was created. This is only valid if the function returns `noErr`.

function result A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

The `NSpGame_Host` function requires a list of protocol references, so that the game can be hosted on multiple protocols. Also, the `NSpDoModalHostDialog` function requires you to pass a protocol list that it fills in. Once a protocol reference has been added to a list, its memory belongs to the list.

NSpProtocolList_Delete

You can use the `NSpProtocolList_Delete` function to delete a protocol list.

```
void NSpProtocolList_Delete  (NSpProtocolListReference inProtocolList);
```

`inProtocolList`
An opaque reference to a list of protocols.

DESCRIPTION

When you use `NSpProtocol_Create` to delete a protocol list, all the protocol references in it are deleted.

NSpProtocolList_Append

You can use the `NSpProtocolList_Append` function to add a new protocol reference to the list.

```
OSStatus NSpProtocolList_Append (
                    NSpProtocolListReference inProtocolList,
                    NSpProtocolReference inProtocolRef);
```

`inProtocolList`
An opaque reference to a protocol list.

`inProtocolRef` An opaque reference to the protocol being appended.

function result A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

The specified protocol reference will be appended to the list of protocol references, and it then becomes the property of the list. A protocol reference added to a list with this function should not be deleted with a call to `NSpProtocol_Delete`.

NSpProtocolList_Remove

You can use the `NSpProtocolList_Remove` function to remove a protocol reference from the list.

```
OSStatus NSpProtocolList_Remove (
                    NSpProtocolListReference inProtocolList,
                    NSpProtocolReference inProtocolRef);
```

`inProtocolList`
An opaque reference to a protocol list.

`inProtocolRef` An opaque reference to the protocol you are removing.

function result A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

When a protocol reference is removed from a protocol list, its memory once again belongs to the application and should be released with a call to `NSpProtocol_Delete`.

NSpProtocolList_RemoveIndexed

You can use the `NSpProtocolList_RemoveIndexed` function to remove the protocol reference at a specific location in the list.

```
OSStatus NSpProtocolList_RemoveIndexed (
                    NSpProtocolListReference inProtocolList,
                    UInt32 inIndex);
```

`inProtocolList`
An opaque reference to a protocol list.

`inIndex` The index entry to be removed.

function result A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

This function is usually used in conjunction with the `NSpProtocolList_GetCount` function for stepping through a protocol list.

NSpProtocolList_GetCount

You can use the `NSpProtocolList_GetCount` function to return the number of protocol references in the list.

```
UInt32 NSpProtocolList_GetCount (
                    NSpProtocolListReference inProtocolList);
```

`inProtocolList`
An opaque reference to a protocol list.

function result The number of protocol references in the list.

DESCRIPTION

Use this function for iterating through the protocol list.

NSpProtocolList_GetIndexedRef

You can use the `NSpProtocolList_GetIndexedRef` function to receive the protocol reference at the indicated location in the list for iteration.

```
NSpProtocolReference NSpProtocolList_GetIndexedRef (
                    NSpProtocolListReference inProtocolList,
                    UInt32 inIndex);
```

`inProtocolList`
An opaque reference to a list of protocols.

`inIndex` A valid index entry.

function result A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

This is a peek function that allows you to look at the contents of a protocol list. `NSpProtocolList_GetIndexedRef` does not remove the protocol from the list, so you must not delete the reference to it.

NSpProtocol_CreateAppleTalk

You can use the `NSpProtocol_CreateAppleTalk` function to have NetSprocket create an AppleTalk protocol reference for you, using the parameters you provide.

```
NSpProtocolReference NSpProtocol_CreateAppleTalk (
                    ConstStr31Param inNBPName,
                    ConstStr31Param inNBPType,
                    UInt32 inMaxRTT,
                    UInt32 inMinThruput);
```

`inNBPName` The Name Binding Protocol name you wish users to see when browsing the AppleTalk network.

`inNBPType` The Name Binding Protocol type to use when advertising the game on an AppleTalk network. This name should be representative of your game, but is never displayed to users. This name must be the same as the one you use in the `ioGameType` field of the `NSpGame_Join` function.

`inMaxRTT` The maximum round-trip time (RTT) allowed for new players. Pass 0 if you do not wish to have round-trip time checked. This does not guarantee that RTT will remain at the level it is when the player joins. RTT is in milliseconds.

`inMinThruput` The minimum throughput required of any prospective entrant into the game. Pass 0 if you do not wish to have throughput checked. This does not guarantee that throughput will remain at the level it is when the player joins. Throughput is measured in bytes per second.

function result A reference to the created protocol, or `NULL` if there was an error in specifying the protocol.

DESCRIPTION

Use this function if you wish to preconfigure the AppleTalk protocol before calling `NSpDoModalHostDialog`, or if you want to host the game programmatically.

NSpProtocol_CreateIP

You can use the `NSpProtocol_CreateIP` function to have NetSprocket create an IP protocol reference for you.

```
NSpProtocolReference NSpProtocol_CreateIP (
                     InetPort inPort,
                     UInt32 inMaxRTT,
                     UInt32 inMinThruput);
```

`inPort` The port on which you wish to listen for new players. Since there is no dynamic name lookup in IP, prospective players cannot know what port a game is being played on unless they receive that information from the hosting player in a manner external to the real-time network. In order to notify you, the person hosting the game might send you electronic mail, call you, or leave a sticky note on your computer telling you what game the port is on and what time to join. When you use the `NSpProtocol_CreateIP` function, you can specify the default port your game is hosted on. You can then specify the same port as the default port t use when joining a game.

`inMaxRTT` The maximum round-trip time allowed for new players. Pass 0 if you do not wish to have round-trip time checked. This does not guarantee that RTT will remain at the level it is when the player joins. RTT is specified in milliseconds.

`inMinThruput` The minimum throughput required of any prospective entrant into the game. Pass 0 if you do not wish to have throughput checked. This does not guarantee that throughput will remain at the level it is when the player joins. Throughput is measured in bytes per second.

function result A reference to the created protocol, or `NULL` if there was an error in specifying the protocol.

DESCRIPTION

Use this function if you wish to preconfigure the TCP/IP (Internet) protocol before calling `NSpDoModalHostDialog` or if you want to host the game programmatically.

Managing Player Information

You can use these NetSprocket functions to get information about the players participating in your game. You can also use these functions to control player information data structures.

NSpPlayer_GetMyID

You can use the `NSpPlayer_GetMyID` function to obtain your player ID.

```
NSpPlayerID NSpPlayer_GetMyID  (NSpGameReference inGame);
```

`inGame` An opaque reference to your game object.

function result A valid player ID.

DESCRIPTION

`NSpPlayer_GetMyID` returns the player ID for the game object you have specified. NetSprocket returns 0 if there is no player associated with the game object.

NSpPlayer_GetInfo

You can use the `NSpPlayer_GetInfo` function to get information about a player.

```
OSStatus NSpPlayer_GetInfo (
                    NSpGameReference inGame,
                    NSpPlayerID inPlayerID,
                    NSpPlayerInfoPtr *outInfo);
```

`inGame` An opaque reference to your game object.

`inPlayerID` The ID of the player you want information about.

`outInfo` A pointer to `NSpPlayerInfoPtr` which contains a pointer to the player's information data structure you have requested.

function result A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

When you are done with the player's information, you should pass the `NSpPlayerInfoPtr` back to NetSprocket via `NSpPlayer_ReleaseInfo`.

NetSprocket returns an error if the player's information could not be obtained.

NSpPlayer_ReleaseInfo

You can use the `NSpPlayer_ReleaseInfo` function to release a player information structure obtained by the `NSpPlayer_GetInfo` function.

```
void NSpPlayer_ReleaseInfo (
                    NSpGameReference inGame,
                    NSpPlayerInfoPtr inInfo);
```

`inGame` An opaque reference to your game object.

`inInfo` The data structure obtained by `NSpPlayer_GetInfo`.

DESCRIPTION

For effective memory management in NetSprocket, you should use the `NSpPlayer_ReleaseInfo` function to release each player information structure obtained by `NSpPlayer_GetInfo`.

NSpPlayer_GetEnumeration

You can use the `NSpPlayer_GetEnumeration` function to take a snapshot of each player in your game.

```
OSStatus NSpPlayer_GetEnumeration (
                    NSpGameReference inGame,
                    NSpPlayerEnumerationPtr *outPlayers);
```

`inGame`	An opaque reference to your game object.
`outPlayers`	A pointer to a player enumeration structure which is allocated and set by NetSprocket.
function result	A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

`NSpPlayer_GetEnumeration` places the information on each player in the player enumeration structure. This structure is made available to your game via `NSpPlayerEnumerationPtr`.

It is important to release the memory held by the player enumeration structure by calling the `NSpPlayer_ReleaseEnumeration` function when you are done.

NetSprocket returns an error if there was a problem getting the player information.

NSpPlayer_ReleaseEnumeration

You can use the `NSpPlayer_ReleaseEnumeration` function to release the player enumeration structure.

```
void NSpPlayer_ReleaseEnumeration (
                    NSpGameReference inGame,
                    NSpPlayerEnumerationPtr inPlayers);
```

`inGame`	An opaque reference to your game object.
`inPlayers`	The player enumeration structure obtained from `NSpPlayer_GetEnumeration`.

DESCRIPTION

For each `NSpPlayer_GetEnumeration` call, you should execute a corresponding `NSpPlayer_ReleaseEnumeration` call to release the memory held by the player enumeration structure.

NSpPlayer_GetRoundTripTime

You can use the `NSpPlayer_GetRoundTripTime` function to determine how many milliseconds elapse between sending a message to a specific player and receiving a reply. Round-trip time between any two players may vary greatly during the course of a game.

```
UInt32 NSpPlayer_GetRoundTripTime (
                    NSpGameReference inGame,
                    NSpPlayerID inPlayer);
```

`inGame`	An opaque reference to your game object.
`inPlayer`	The player ID you are sending the test message to.
function result	The time elapsed between sending and receiving a message.

DESCRIPTION

This function is synchronous. That is, it blocks until a reply is received unless the time-out is reached. If time-out is exceeded, -1 will be returned.

NSpPlayer_GetThruput

You can use the `NSpPlayer_GetThruput` function to determine the data throughput between the caller and the specified player. Throughput is measured in bytes per second.

```
UInt32 NSpPlayer_GetThruput (
                    NSpGameReference inGame,
                    NSpPlayerID inPlayer);
```

`inGame`	An opaque reference to your game object.
`inPlayer`	The player ID you are sending the test message to.
function result	The throughput between the caller and the player.

DESCRIPTION

This function is synchronous. That is, it blocks until it finishes testing throughput unless the time-out is reached. If time-out is exceeded, -1 will be returned. Throughput between any two players may vary greatly during the course of a game.

Managing Groups of Players

You may use these NetSprocket functions to create and manage groups of players participating in your game. Groups are a shared resource of the entire game. When a group is created by one player, it can be used, modified, or deleted by any player in the game.

NSpGroup_Create

You can use the `NSpGroup_Create` function to create a new group of players.

```
OSStatus NSpGroup_Create (
                    NSpGameReference inGame,
                    NSpGroupID *outGroupID);
```

`inGame`	An opaque reference to your game object.
`outGroupID`	A unique number identifying the new group you have created.
function result	A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

Once a group is created, the value in the `outGroupID` parameter is distributed to each player in the game. Any player in the game can use the `outGroupID` parameter to send messages to the players in the group.

NetSprocket returns an error if NetSprocket could not create the group.

NSpGroup_Delete

You can use the `NSpGroup_Delete` function to delete a group from the game.

```
OSStatus NSpGroup_Delete (
                    NSpGameReference inGame,
                    NSpGroupID inGroupID);
```

`inGame`	An opaque reference to your game object.
`inGroupID`	The ID of the group to delete.
function result	A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

`NSpGroup_Delete` does not delete the players in the group. It simply deletes the group ID. A deleted group is no longer usable by any player in the game.

NetSprocket returns an error if it could not delete the group

NSpGroup_AddPlayer

You can use the `NSpGroup_AddPlayer` function to add a player to a group.

```
OSStatus NSpGroup_AddPlayer (
                    NSpGameReference inGame,
                    NSpGroupID inGroupID,
                    NSpPlayerID inPlayerID);
```

`inGame`	An opaque reference to your game object.
`inGroupID`	The group to which you are adding the player.
`inPlayerID`	The player to be added.
function result	A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

NetSprocket returns an error if it could not add the player to the group or the group ID is invalid.

NSpGroup_RemovePlayer

You can use the `NSpGroup_RemovePlayer` function to remove a player from a group.

```
OSStatus NSpGroup_RemovePlayer (
                    NSpGameReference inGame,
                    NSpGroupID inGroupID,
                    NSpPlayerID inPlayerID);
```

`inGame`	An opaque reference to your game object.
`inGroupID`	The group from which the player is to be removed.
`inPlayerID`	The player to be removed.
function result	A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

NetSprocket returns an error if the `NSpGroup_RemovePlayer` function could not remove the player or if the player ID or group ID is invalid. This function does not delete the player from the game. It only removes the player from the list of players in the group.

NSpGroup_GetInfo

You can use the `NSpGroup_GetInfo` function to obtain the group's information structure.

```
OSStatus NSpGroup_GetInfo (
                    NSpGameReference inGame,
                    NSpGroupID inGroupID,
                    NSpGroupInfoPtr *outInfo);
```

`inGame`	An opaque reference to your game object.
`inGroupID`	The group you want information about.
`outInfo`	A pointer to an array of group information structures.
function result	A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

The group information data structure will be allocated by NetSprocket and the structure will be populated with the group's information. For efficient memory management, the `NSpGroupInfo` data structure should be released with a call to `NSpGroup_ReleaseInfo` when you have finished using it.

NetSprocket returns an error if NetSprocket could not build the group information data structure or if the group ID was invalid.

NSpGroup_ReleaseInfo

You can use the `NSpGroup_ReleaseInfo` function to release memory held by the group information structure.

```
void NSpGroup_ReleaseInfo (
                    NSpGameReference inGame,
                    NSpGroupInfoPtr inInfo);
```

`inGame`	An opaque reference to your game object.
`inInfo`	A pointer to an array of group information structures.

DESCRIPTION

For each `NSpGroup_GetInfo` call, you should execute a corresponding `NSpGroup_ReleaseInfo` call to release the memory held by the group information structure.

NSpGroup_GetEnumeration

You can use the `NSpGroup_GetEnumeration` function to obtain a list of the groups in the game.

```
OSStatus NSpGroup_GetEnumeration (
                    NSpGameReference inGame,
                    NSpGroupEnumerationPtr *outGroups);
```

`inGame`	An opaque reference to your game object.
`outGroups`	A pointer to the group enumeration structure that is allocated and populated by NetSprocket.
function result	A result code of `noErr`, or a NetSprocket or Open Transport result code.

DESCRIPTION

For efficient memory management, the group enumeration structure should be released by NetSprocket by calling `NSpGroup_ReleaseEnumeration`.

NetSprocket returns an error if it could not build the group list.

NSpGroup_ReleaseEnumeration

You can use the `NSpGroup_ReleaseEnumeration` function to release memory held by the group enumeration structure.

```
void NSpGroup_ReleaseEnumeration (
                    NSpGameReference inGame,
                    NSpGroupEnumerationPtr inGroups);
```

`inGame`	An opaque reference to your game object.
`inGroups`	A pointer to a group enumeration structure.

DESCRIPTION

For each `NSpGroup_GetEnumeration` call, you should execute a corresponding `NSpGroup_ReleaseEnumeration` call to release the memory held by `inGroups`.

Utility Functions

The following are some useful functions for use with NetSprocket. These include network performance testing, and some functions associated with Open Transport.

NSpClearMessageHeader

You can use the `NSpClearMessageHeader` function to initialize the entire header structure.

```
void NSpClearMessageHeader (
                    NSpGameReference inGame,
                    NSpMessageHeader *ioMessage);
```

`inGame`	An opaque reference to your game object.
`ioMessage`	A pointer to the message to be initialized.

DESCRIPTION

You should call the `NSpClearMessageHeader` function each time before you start filling in your message structures. If you fail to initialize your message structures, you may end up with invalid data.

NSpGetCurrentTimeStamp

You can use the `NSpGetCurrentTimeStamp` function to compare time stamps.

```
UInt32 NSpGetCurrentTimeStamp (
                    NSpGameReference inGame);
```

`inGame` An opaque reference to your game object.

function result The time value in milliseconds.

DESCRIPTION

You can use this function to compare the time stamp of a message with the current time stamp to determine how long ago a message was sent. This value is only as accurate as the round-trip time to the application hosting the game. This is a normalized value established by the server. That is, anyone in the current game who calls this function will get the same value.

NSpConvertOTAddrToAddressReference

You can use the `NSpConvertOTAddrToAddressReference` function to obtain a `NSpAddressReference`.

```
NSpAddressReference NSpConvertOTAddrToAddressReference (
                    OTAddress *inAddress);
```

`inAddress` A valid (TCP/IP or AppleTalk) `OTAddress` returned from Open Transport.

function result A valid `NSpAddressReference`.

DESCRIPTION

Given an `OTAddress`, `NSpConvertOTAddrToAddressReference` returns a new `NSpAddressReference` for the `OTAddress` you have passed to NetSprocket. For efficient memory management, the reference should be released by calling `NSpReleaseAddressReference` when you're done. You can delete `OTAddress` once the call has completed.

You should use this function when you do not wish to use the human interface functions provided by NetSprocket for standard hosting, browsing, and joining.

NSpConvertAddressReferenceToOTAddr

You can use the `NSpConvertAddressReferenceToOTAddr` function to obtain an `OTAddress`.

```
OTAddress *NSpConvertAddressReferenceToOTAddr (
                    NSpAddressReference inAddress);
```

`inAddress` A valid `NSpAddressReference` returned from `NSpDoModalJoinDialog`.

function result A valid `OTAddress`.

DESCRIPTION

Use `NSpConvertAddressReferenceToOTAddr` when you want to use the `NSpDoModalJoinDialog` function and you do not plan to use the any other functions provided in NetSprocket, such as networking, group, or player functions.

NSpReleaseAddressReference

You can use the `NSpReleaseAddressReference` function to release memory associated with an address reference.

```
void NSpReleaseAddressReference (
                    NSpAddressReference inAddress);
```

`inAddress` A valid `NSpAddressReference` returned from `NSpDoModalJoinDialog` or `NSpConvertOTAddrToAddressReference`.

DESCRIPTION

For efficient memory management, you should call `NSpReleaseAddressReference` with `NSpAddressReference` when your game no longer needs the memory allocated.

Summary of NetSprocket

Constants

```
#define     kNSpMaxPlayerNameLen                31
#define     kNSpMaxGroupNameLen                 31
#define     kNSpMaxPasswordLen                  31
#define     kNSpMaxGameNameLen                  31
#define     kNSpMaxDefinitionStringLen          255
```

Network Message Priority Flags

```
enum {
    kNSpJunk                                    = 0x10000000,
    kNSpNormal                                  = 0x20000000,
    kNSpRegistered                              = 0x30000000
};
```

Network Message Delivery Flags

```
enum {
    kNSpFailIfPipeFull                          = 0x00000001,
    kNSpSelfSend                                = 0x00000002,
    kNSpBlocking                                = 0x00000004
};
```

Options for Hosting, Joining, and Ending Games

```
enum {
    kNSpGameFlag_DontAdvertise                  = 0x00000001,
    kNSpGameFlag_ForceTerminateGame             = 0x00000002
};
```

Network Message Types

```
enum {
    kNSpSystemMessagePrefix                 = 0x80000000,
    kNSpError                               = kNSpSystemMessagePrefix | 0x7FFFFFFF,
    kNSpJoinRequest                         = kNSpSystemMessagePrefix | 0x00000001,
    kNSpJoinApproved                        = kNSpSystemMessagePrefix | 0x00000002,
    kNSpJoinDenied                          = kNSpSystemMessagePrefix | 0x00000003,
    kNSpPlayerJoined                        = kNSpSystemMessagePrefix | 0x00000004,
    kNSpPlayerLeft                          = kNSpSystemMessagePrefix | 0x00000005,
    kNSpHostChanged                         = kNSpSystemMessagePrefix | 0x00000006,
    kNSpGameTerminated                      = kNSpSystemMessagePrefix | 0x00000007
};
```

Reserved Player IDs for Network Messages

```
enum {
    kNSpAllPlayers                                  = 0x00000000,
    kNSpServerOnly                                  = 0xFFFFFFFF
};
```

Topology Types

```
typedef enum {
    kNSpClientServer                                = 0x00000001
} NSpTopology;
```

Data Types

```
typedef         SInt32          NSpEventCode;

typedef         SInt32          NSpGameID;

typedef         SInt32          NSpPlayerID;

typedef         NSpPlayerID     NSpGroupID;

typedef         UInt32          NSpPlayerType;

typedef         SInt32          NSpFlags;
```

Opaque Game Reference Structures

```
typedef         struct          NSpGamePrivate          *NSpGameReference;

typedef         struct          NSpProtocolPrivate      *NSpProtocolReference;

typedef         struct          NSpListPrivate          *NSpProtocolListReference;

typedef         struct          NSPAddressPrivate       *NSpAddressReference;
```

Player Information Structure

```
typedef struct NSpPlayerInfo {
    NSpPlayerID                         id;
    NSpPlayerType                       type;
    Str31                               name;
    UInt32                              groupCount;
    NSpGroupID                          groups[kVariableLengthArray];
} NSpPlayerInfo, *NSpPlayerInfoPtr;
```

Player List Structure

```
typedef struct NSpPlayerEnumeration {
    UInt32                              count;
    NSpPlayerInfoPtr                    playerInfo[kVariableLengthArray];
} NSpPlayerEnumeration, *NSpPlayerEnumerationPtr;
```

Group Information Structure

```
typedef struct NSpGroupInfo {
    NSpGroupID                          id;
    UInt32                              playerCount;
    NSpPlayerID                         players[kVariableLengthArray];
} NSpGroupInfo, *NSpGroupInfoPtr;
```

Group Enumeration Structure

```
typedef struct NSpGroupEnumeration {
    UInt32                              count;
    NSpGroupInfoPtr                     groups[kVariableLengthArray];
} NSpGroupEnumeration, *NSpGroupEnumerationPtr;
```

Game Information Structure

```
typedef struct NSpGameInfo {
    UInt32                          maxPlayers;
    UInt32                          currentPlayers;
    UInt32                          currentGroups;
    NSpTopology                     topology;
    UInt32                          reserved;
    Str31                           name;
    Str31                           password;
} NSpGameInfo;
```

Message Header Structure

```
typedef struct NSpMessageHeader {
    UInt32                          version;
    SInt32                          what;
    NSpPlayerID                     from;
    NSpPlayerID                     to;
    UInt32                          id;
    UInt32                          when;
    UInt32                          messageLen;
} NSpMessageHeader;
```

Error Message Structure

```
typedef struct NSpErrorMessage {
    NSpMessageHeader                header;
    OSStatus                        error;
} NSpErrorMessage;
```

Join Request Message Structure

```
typedef struct NSpJoinRequestMessage {
    NSpMessageHeader                header;
    Str31                           name;
    Str31                           password;
    UInt32                          type;
    UInt32                          customDataLen;
    UInt8                           customData[kVariableLengthArray];
} NSpJoinRequestMessage;
```

Join Approved Message Structure

```
typedef struct NSpJoinApprovedMessage {
    NSpMessageHeader                    header;
} NSpJoinApprovedMessage;
```

Join Denied Message Structure

```
typedef struct NSpJoinDeniedMessage {
    NSpMessageHeader                    header;
    Str255    reason;
} NSpJoinDeniedMessage;
```

Player Joined Message Structure

```
typedef struct NSpPlayerJoinedMessage {
    NSpMessageHeader                    header;
    UInt32                              playerCount;
    NSpPlayerInfo                       playerInfo;
} NSpPlayerJoinedMessage;
```

Player Left Message Structure

```
typedef struct NSpPlayerLeftMessage {
    NSpMessageHeader                    header;
    UInt32                              playerCount;
    NSpPlayerID                         playerID;
} NSpPlayerLeftMessage;
```

Host Changed Message Structure

```
typedef struct NSpHostChangedMessage {
    NSpMessageHeader                    header;
    NSpPlayerID                         newHost;
} NSpHostChangedMessage;
```

Game Terminated Message Structure

```
typedef struct NSpGameTerminatedMessage {
    NSpMessageHeader                    header;
} NSpGameTerminatedMessage;
```

NetSprocket Functions

Initializing NetSprocket

```
OSStatus NSpInitialize            (UInt32 inStandardMessageSize,
                                   UInt32 inBufferSize,
                                   UInt32 inQElements,
                                   NSpGameID inGameID,
                                   UInt32 inTimeout);

typedef pascal void (*NSpCallbackProcPtr)
                                   (NSpGameReference inGame,
                                   void *inContext,
                                   NSpEventCode inCode,
                                   OSStatus inStatus,
                                   void* inCookie);

OSStatus NSpInstallCallbackHandler (NSpCallbackProcPtr inHandler,
                                   void *inContext);
```

Human Interface Functions

```
NSpAddressReference NSpDoModalJoinDialog
                                   (ConstStr31Param inGameType,
                                   ConstStr31Param inEntityListLabel,
                                   Str31 ioName,
                                   Str31 ioPassword);

Boolean NSpDoModalHostDialog      (NSpProtocolListReference ioProtocolList,
                                   Str31 ioGameName,
                                   Str31 ioPlayerName,
                                   Str31 ioPassword);
```

Hosting and Joining a Game

```
OSStatus NSpGame_Host             (NSpGameReference *outGame,
                                   NSpProtocolListReference inProtocolList,
                                   UInt32 inMaxPlayers,
                                   ConstStr31Param inGameName,
                                   ConstStr31Param inPassword,
                                   ConstStr31Param inPlayerName,
```

```
                                    NSpPlayerType inPlayerType,
                                    NSpTopology inTopology,
                                    NSpFlags inFlags);

OSStatus NSpGame_Join              (NSpGameReference *outGame,
                                    NSpAddressReference inAddress,
                                    ConstStr31Param inName,
                                    ConstStr31Param inPassword,
                                    NSpPlayerType inType,
                                    Uint32 inUserDataLen,
                                    void *inUserData,
                                    NSpFlags inFlags);

OSStatus NSpGame_EnableAdvertising
                                    (NSpGameReference inGame,
                                    NSpProtocolReference inProtocol,
                                    Boolean inEnable);

OSStatus NSpGame_Delete            (NSpGameReference inGame,
                                    NSpFlags inFlags);

typedef pascal Boolean (*NSpJoinRequestHandlerProcPtr)
                                    (NSpGameReference inGame,
                                    NSpJoinRequestMessage *inMessage,
                                    void* inContext,
                                    Str255 outReason);

OSStatus NSpInstallJoinRequestHandler
                                    (NSpJoinRequestHandlerProcPtr inHandler,
                                    void *inContext);
```

Sending and Receiving Messages

```
OSStatus NSpMessage_Send           (NSpGameReference inGame,
                                    NSpMessageHeader *inMessage,
                                    NSpFlags inFlags);

NSpMessageHeader *NSpMessage_Get   (NSpGameReference inGame);

void NSpMessage_Release            (NSpGameReference inGame,
                                    NSpMessageHeader *inMessage);

typedef pascal Boolean (*NSpMessageHandlerProcPtr)
                                    (NSpGameReference inGame,
                                    NSpMessageHeader *inMessage,
                                    void *inContext);
```

```
OSStatus NSpInstallAsyncMessageHandler (NSpMessageHandlerProcPtr inHandler,
                                        void *inContext);
```

Managing Network Protocols

```
OSStatus NSpProtocol_Create       (const char* inDefinitionString,
                                   NSpProtocolReference *outReference);

void NSpProtocol_Delete           (NSpProtocolReference inProtocolRef);

OSStatus NSpProtocol_ExtractDefinitionString
                                   (NSpProtocolReference inProtocolRef,
                                   char *outDefinitionString);

OSStatus NSpProtocolList_Create   (NSpProtocolReference inProtocolRef,
                                   NSpProtocolListReference *outList);

void NSpProtocolList_Delete       (NSpProtocolListReference inProtocolList);

OSStatus NSpProtocolList_Append   (NSpProtocolListReference inProtocolList,
                                   NSpProtocolReference inProtocolRef);

OSStatus NSpProtocolList_Remove   (NSpProtocolListReference inProtocolList,
                                   NSpProtocolReference inProtocolRef);

OSStatus NSpProtocolList_RemoveIndexed
                                   (NSpProtocolListReference inProtocolList,
                                   UInt32 inIndex);

UInt32 NSpProtocolList_GetCount   (NSpProtocolListReference inProtocolList);

NSpProtocolReference NSpProtocolList_GetIndexedRef
                                   (NSpProtocolListReference inProtocolList,
                                   UInt32 inIndex);

NSpProtocolReference NSpProtocol_CreateAppleTalk (ConstStr31Param inNBPName,
                                   ConstStr31Param inNBPType,
                                   UInt32 inMaxRTT,
                                   UInt32 inMinThruput);

NSpProtocolReference NSpProtocol_CreateIP (InetPort inPort,
                                   UInt32 inMaxRTT,
                                   UInt32 inMinThruput);
```

Managing Player Information

```
NSpPlayerID NSpPlayer_GetMyID     (NSpGameReference inGame);
```

```
OSStatus NSpPlayer_GetInfo        (NSpGameReference inGame,
                                   NSpPlayerID inPlayerID,
                                   NSpPlayerInfoPtr *outInfo);

void NSpPlayer_ReleaseInfo        (NSpGameReference inGame,
                                   NSpPlayerInfoPtr inInfo);

OSStatus NSpPlayer_GetEnumeration (NSpGameReference inGame,
                                   NSpPlayerEnumerationPtr *outPlayers);

void NSpPlayer_ReleaseEnumeration (NSpGameReference inGame,
                                   NSpPlayerEnumerationPtr inPlayers);

UInt32 NSpPlayer_GetRoundTripTime (NSpGameReference inGame,
                                   NSpPlayerID inPlayer,
                                   UInt32 inTimeout);

UInt32 NSpPlayer_GetThruput       (NSpGameReference inGame,
                                   NSpPlayerID inPlayer,
                                   UInt32 inTimeout);
```

Managing Groups of Players

```
OSStatus NSpGroup_Create          (NSpGameReference inGame,
                                   NSpGroupID *outGroupID);

OSStatus NSpGroup_Delete          (NSpGameReference inGame,
                                   NSpGroupID inGroupID);

OSStatus NSpGroup_AddPlayer       (NSpGameReference inGame,
                                   NSpGroupID inGroupID,
                                   NSpPlayerID inPlayerID);

OSStatus NSpGroup_RemovePlayer    (NSpGameReference inGame,
                                   NSpGroupID inGroupID,
                                   NSpPlayerID inPlayerID);

OSStatus NSpGroup_GetInfo         (NSpGameReference inGame,
                                   NSpGroupID inGroupID,
                                   NSpGroupInfoPtr *outInfo);

void NSpGroup_ReleaseInfo         (NSpGameReference inGame,
                                   NSpGroupInfoPtr inInfo);

OSStatus NSpGroup_GetEnumeration  (NSpGameReference inGame,
                                   NSpGroupEnumerationPtr *outGroups);
```

```
void NSpGroup_ReleaseEnumeration  (NSpGameReference inGame,
                                   NSpGroupEnumerationPtr inGroups);
```

Utility Functions

```
void NSpClearMessageHeader        (NSpGameReference inGame,
                                   NSpMessageHeader *ioMessage);

UInt32 NSpGetCurrentTimeStamp     (NSpGameReference inGame);

NSpAddressReference NSpConvertOTAddrToAddressReference
                                   (OTAddress *inAddress);

OTAddress *NSpConvertAddressReferenceToOTAddr
                                   (NSpAddressReference inAddress);

void NSpReleaseAddressReference   (NSpAddressReference inAddress);
```

Result Codes

noErr	0	No error
kNSpInitializationFailedErr	–30360	NetSprocket could not be initialized
kNSpAlreadyInitializedErr	–30361	NetSprocket has already been initialized
kNSpTopologyNotSupportedErr	–30362	The requested topology is unimplemented, or invalid value
kNSpMessageSizeTooBigErr	–30363	Current memory conditions prevent message from being sent
kNSpBufferTooSmallErr	-30364	The buffer allocated is too small to handle the data
kNSpReceiveDataErr	-30365	A problem occurred when attempting to receive data
kNSpProtocolNotAvailableErr	–30366	A protocol reference indicated a protocol that is unavailable
kNSpInvalidGameRefErr	–30367	An invalid game reference was passed
kNSpInvalidNetMessageErr	–30368	An invalid message was passed
kNSpInvalidParameterErr	-30369	A generic parameter error occurred
kNSpOTNotPresentErr	–30370	Open Transport is not installed or not installed correctly
kNSpHostFailedErr	–30371	An attemp to host a game failed
kNSpNotHostAddressErr	–30372	The address specified does not identify a NetSprocket host
kNSpMemAllocationErr	–30373	NetSprocket has run out of memory
kNSpAlreadyAdvertisingErr	–30374	The game is already being advertised on the specified protocol
kNSpNoTypeSpecifiedErr	–30375	An AppleTalk protocol reference that does not specify an NBP type was passed to the host
kNSpNotAdvertisingErr	–30376	The game is not being advertised at this time
kNSpInvalidAddressErr	–30377	An invalid address was passed
kNSpFreeQExhaustedErr	–30378	NetSprocket has exhausted its allocated queue elements and new messages will be dropped
kNSpRemovePlayerFailedErr	-30379	Your attempt to remove a player failed
kNSpAddressInUseErr	-30380	You are attempting to use an address that is already in use
NSpFeatureNotImplementedErr	-30381	You called a NetSprocket function that is not implemented
NSpNameRequiredErr	-30382	You atemped to join a game without specifying a player name
NSpInvalidPlayerIDErr	-30383	You tried to send a message to or get information about a player that is not currently in the game
NSpInvalidGroupIDErr	-30384	You tried to send a message to, or get information about a group that is not currently in the game
NSpNoPlayersErr	-30385	Returned by `NSpPlayer_Enumerate` when there are no players
NSpNoGroupsErr	-30386	Returned by `NSpGroup_Enumerate` when there are no groups
NSpNoHostVolunteersErr	-30387	Returned by `NSpGame_Delete` when called by a host and there are no other players capable of taking over the game
kNSpCreateGroupFailedErr	-30388	The attempt to creat a group failed
NSpAddPlayerFailedErr	-30389	An attempt to add a player to a group failed
NSpInvalidDefinitionErr	-30390	A invalid protocol definition string was passed to `NSpProtocol_Create`
NSpInvalidProtocolRefErr	-30391	An invalid protocol reference was passed
NSpInvalidProtocolListErr	-30392	An invalid protocol list was passed
kNSpTimeoutErr	-30393	A time out error has occurred
kNSpGameTerminatedErr	-30394	An attempt to terminate the game has failed
kNSpConnectFailedErr	-30395	A connection attempt has failed

kNSpSendFailedErr	-30396	An attempt to send a message has failed
kNSpPortTakenErr	-30397	The port you attempted to use is already in use
kNSpNotPlayingErr	-30398	The player you sent a message to is not playing the game

Glossary

3D See **three-dimensional**. See also **spatial.**

3D sound component See **localization component.**

3D sound source A part of the virtual audio environment provided by SoundSprocket that emits sounds. Defined by the `SSpSourceReference` data type. Compare **listener.**

Apple Game Sprockets A collection of system software components that provide capabilities (such as three-dimensional sound filtering or machine-to-machine communication) specifically intended to facilitate the development of games for Macintosh computers.

ambient Of a sound, to appear to be emanating from all directions. Compare **binaural, localized.**

angular attenuation The loss of a sound's volume due to a change in the angle between the sound source orientation and the vector between the source and the listener. Compare **distance attenuation, reverberation attenuation, room reflectivity attenuation.**

angular attenuation cone A cone that determines the direction of maximum sound intensity and the amount of attenuation that occurs as the angle between the orientation vector and the source-to-listener vector increases toward a predefined limit.

attenuation The loss of sound volume caused by some physical action on the sound (such as its traveling over a distance or reverberating off a wall). See also **angular attenuation, distance attenuation, reverberation attenuation, room reflectivity attenuation.**

axis element A continuous control element with or without a meaningful center—for example, a joystick or a gas pedal. See also **button element, directional pad element.**

back buffer The buffer DrawSprocket draws into while another image buffer is being displayed.

bilinear interpolation Bilinear interpolation averages the pixels and determines a pixel value that falls between the two original pixels.

binaural Of a sound, recorded with a localized effect. Compare **ambient, localized.**

blanking window A window with which DrawSprocket completely covers the display, hiding the desktop, menu bar, and other system resources, and providing a uniform background color for the game to draw over.

button element A two-state element. See also **axis element, directional pad element, movement element.**

camera placement structure A data structure that contains information about the placement (that is, the location, orientation, and direction) of a camera. Defined by the `TQ3CameraPlacement` data type.

Cartesian coordinate system A system of assigning planar positions to objects in terms of their distances from two mutually perpendicular lines (the *x* and *y* coordinate axes), or of assigning spatial positions to objects in terms of their distances from three mutually perpendicular lines (the *x*, *y*, and *z* coordinate axes). Compare **polar coordinate system.**

component A piece of code that provides a defined set of services to one or more clients. Applications, system extensions, and other components can use the services of a component. See also **localization component, sound output device component.**

context A combination of resolution and pixel depth associated with a particular display.

context attributes structure A data structure that DrawSprocket uses to pass and receive information about a context.

coordinate system Any system of assigning planar or spatial positions to objects. Compare **Cartesian coordinate system, polar coordinate system.**

dB See **decibel.**

decibel (dB) A unit of loudness, equal to 20 times the common logarithm of the ratio of the pressure produced by a sound wave to a pre-established reference pressure.

direct device A graphics device whose pixels directly specify the color with which they are to be drawn. Compare **indexed device.**

directional pad element A nine (sometimes five) state element. It has an idle position and eight (or four) states corresponding to directions. See also **axis element, button element, movement element.**

dirty rectangle An area of the back buffer that has been invalidated so that DrawSprocket knows it has been changed.

distance attenuation The loss of a sound's volume as it travels over the distance from the sound source to the listener. Compare **angular attenuation, reverberation attenuation, room reflectivity attenuation.**

Doppler effect The perceived change in the frequency of a sound caused by motion of the listener relative to the source of the sound.

Doppler shift The magnitude of the perceived change in the frequency of a sound due to the Doppler effect.

double buffering A way to emulate hardware page flipping in software. The game draws to a back buffer and then transfers the completed offscreen image to the hardware display.

DrawSprocket The component of Apple Game Sprockets that gives video games access to and control over display and device-access features.

echo See **reverberation.**

element A building block that corresponds to a device control and consists of a kind, a label, and a unique human readable identifier.

element event structure A data structure used to pass element event data during game play. Defined by the `ISpElementEvent` data type.

fault-tolerance That quality or capability of network system services that handles and recovers from unexpected network problems and failures transparently to the applications using the services, and that adapts well to the current conditions under which it is operating, shielding the applications as well as possible from these difficulties.

forward-pointing vector See **orientation.**

host The application that is sponsoring or controlling a game. A host application may also support a player in the same application.

gamma fade Alters the pixel intensity mapping to produce a fading effect. This type of fade is based on a gamma table, which accounts for nonlinearities in the display's color response.

group A set or logical collection of players currently participating in a game sharing a common element, such as team members or players with scores over 1000. A player may belong to more than one group.

human readable identifier A string that a game can display to identify an element for the user.

identity matrix Any $n \times n$ square matrix with elements a_{ij} such that $a_{ij} = 1$ if $i = j$ and $a_{ij} = 0$ otherwise. A point or vector that is transformed by the identity matrix remains unchanged.

indexed device A graphics device whose pixel colors are specified indirectly, through a color lookup table. Compare **direct device.**

InputSprocket A part of Apple Game Sprockets through which games can communicate with joysticks and other game-oriented input devices.

invalid-rectangle list A list of the rectangular areas in the back buffer that must be redrawn to the screen the next time the buffers are swapped.

kind A four-character sequence that identifies the type of data an element produces. Compare **label**.

label A four-character sequence that identifies the suggested use for an element. Compare **kind.**

listener (1) The part of the virtual audio environment provided by SoundSprocket that perceives sounds. Defined by the `SSpListenerReference` data type. (2) The user who is listening to the sounds produced using SoundSprocket.

listener unit The unit of measurement associated with a particular listener. By default, the listener unit is the meter, but you can change the listener unit by calling the `SSpListener_SetMetersPerUnit` function.

localization component A sound component that provides 3D filtering for the sounds passed to it.

localized Of a sound, to appear to be emanating from a specific location in space. Compare **ambient, binaural.**

medium See **sound medium.**

message A complete set of information transmitted among players in a game. Messages may be broken into packets by NetSprocket for actual transmission over the network.

monophonic sound Sound consisting of a single channel. Compare **stereo sound.**

multichannel sound See **stereo sound.**

need structure A data structure that describes the requirements the game has for input. Defined by the `ISpNeed` data type.

NetSprocket A part of Apple Game Sprockets that provides streamlined, high-performance networking capabilities for multi-user games.

normal (a.) Perpendicular. (n.) A normalized vector.

normalized vector A vector whose length is 1.

opaque For a data structure, not publicly defined. You must use functions to get and set values in an opaque data structure.

orientation A vector that indicates the direction of a listener or a sound source. For a listener, the orientation is the unit vector beginning at the midpoint of the line connecting the two ears and pointing straight out through the listener's nose. Compare **up vector.**

origin In Cartesian coordinates, the point (0, 0) or (0, 0, 0). The coordinate axes intersect at the origin.

overlay An image that appears over the back buffer and is automatically applied just before the back buffer is swapped.

page flipping Drawing done to an offscreen video page located in VRAM that can be displayed by the video controller in rotation with other VRAM pages.

pixel depth Number of bits per pixel.

player A logical entity associated with each person who is participating in a networked game session.

play state The state of a context; it can be one of active, paused, or inactive.

polar coordinate system A system of assigning planar positions to objects in terms of their distances (r) from a point (the polar origin, or pole) along a ray that forms a given angle (θ) with a coordinate line (the polar axis). The polar origin has the polar coordinates (0, θ), for any angle θ. Compare **Cartesian coordinate system.**

polar origin The point in a plane from which the polar axis radiates. Also called the *pole*.

pole See **polar origin.**

polyphonic sound See **stereo sound.**

position (1) Of a listener, the midpoint of the line connecting the two ears. (2) Of a sound source, the center of the bounding box.

private See **opaque.**

QuickDraw 3D A graphics library developed by Apple Computer, Inc., that you can use to create, configure, render, and interact with models of three-dimensional objects. You can also use QuickDraw 3D to read and write three-dimensional data.

reference distance The distance, in meters or listener units, from a listener to the point at which a sound was recorded.

resolution The horizontal and vertical timing of a display mode.

reverb See **reverberation.**

reverberation The bouncing of a sound off walls or other objects.

reverberation attenuation The amount of reverberant signal, in decibels (dB), to mix into the output sound. Compare **angular attenuation, distance attenuation, room reflectivity attenuation.**

room reflectivity attenuation The loss of a sound's volume as it bounces off walls. Compare **angular attenuation, distance attenuation, reverberation attenuation.**

room size The distance, in listener units, between two reverberant walls spaced equidistant from the listener. A room size of 0.0 indicates that no reverberation is to occur.

rotate To reposition an object by revolving (or turning) each point of the object by the same angle around a point or axis.

sound Anything perceived by the ear.

sound channel A path that sound data traverses from an application to the sound output device. A sound channel is associated with a queue of sound commands and with other information about the audio characteristics of the sound data. See also **sound channel record.**

sound channel information selector A value of type `OSType` that determines the kind of information to get or set for a sound channel.

sound channel record A structure that represents a sound channel. Defined by the `SndChannel` data type.

sound component A component that works with the Sound Manager to manipulate audio data or to communicate with a sound output device. See also **localization component, sound output device component.**

sound component chain A chain of sound components that links a sound source to a sound output device.

sound component link structure A structure that represents a link in the sound component chain of a sound channel. Defined by the `SoundComponentLink` data type.

Sound Manager The part of the Macintosh system software that manages the production and manipulation of sounds on Macintosh computers.

sound medium A substance through which sound can travel.

sound output device Any hardware device (such as a speaker or sound synthesizer) that produces sound.

sound output device component A sound component that communicates with a sound output device.

SoundSprocket A part of the Apple Game Sprockets that provides three-dimensional sound filtering and other sonic capabilities for Macintosh computers.

sound source See **3D sound source.** (Do not confuse with sound sources as defined by the Sound Manager.)

source See **3D sound source.**

source mode A feature of a listener that determines the type of filtering that is applied to a source sound.

spatial Contained completely in three dimensions (for example, a box).

standard temperature and pressure (STP) A temperature of 20° C and a barometric pressure of 760 mm.

stereo sound Sound that simultaneously consists of two or more channels. Also called *polyphonic sound* or *multichannel sound*. Compare **monophonic sound.**

STP See **standard temperature and pressure.**

three-dimensional Appearing to exist in three dimensions.

transformation matrix A matrix that defines the transformation to be applied to a listener or a sound source.

transform matrix See **transformation matrix.**

transparency mask The transparent areas of an overlay that allow the back buffer to show through.

movement element An element that produces data given both as x-y data and as directional data, allowing the game to use whichever data is suitable. See also **axis element, button element, directional pad element.**

triple buffering Adding a extra buffer to the back buffer queue so that the game can continue drawing into the extra buffer while the first back buffer is being drawn to the display.

underlay An image that is used as a background for your back buffer and is automatically applied just before the back buffer is drawn into.

unit cube A box whose three defining edges have a length of 1.

unit vector See **normalized vector.**

up vector A vector that indicates which direction is up. Compare **orientation.**

vector A pair of or three of floating-point numbers that obeys the laws of vector arithmetic. Compare **normal.**

virtual audio environment The combination of a single listener and one or more 3D sound sources.

virtual source A reflection of a real sound source. Defined by the `SSpVirtualSourceData` data type.

Index

E

F

G

H

I

J

K

L

M

N

O

P, Q

R

S

T

U

V, W, X, Y, Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe™ Illustrator and Adobe Photoshop.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITERS
Dave Bice, Judy Helfand, Tim Monroe, Larry Wood

ILLUSTRATORS
Deb Dennis, Sandee Karr

PRODUCTION EDITOR
Gerri Gray

LEAD WRITER
Linda Kyrnitszke

WRITING MANAGER
Trish Eastman