Seed Draft

Simplifying Networked Gaming Using NetSprocket

For NetSprocket 1.7.1

Preliminary

October 21, 1999
Technical Publications

C H A P T E R 1

Introduction

IMPORTANT

This is a preliminary document. Although it has been reviewed for technical accuracy, it is not final. Apple Computer, Inc. is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. You can check <http://developer.apple.com/techpubs/macos8/SiteInfo/whatsnew.html> for information about updates to this and other developer documents. To receive notification of documentation updates, you can sign up for ADC's free Online Program and receive their weekly Apple Developer Connection News e-mail newsletter. (See <http://developer.apple.com/membership/index.html> for more details about the Online Program.) ▲

NetSprocket is a subset of Apple Game Sprockets that provides networking capabilities specifically geared for game developers. You can use NetSprocket to 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.

This document assumes you are familiar with programming Macintosh computers. It does not discuss the details of networking on Macintosh computers; for that information you should consult *Inside Macintosh: Networking with Open Transport*.

If you are building a game, you may also want to consult other Game Sprocket documentation:

- *Manipulating Displays Using DrawSprocket*

- *Configuring Game Input Devices with InputSprocket*
- SoundSprocket documentation (forthcoming)

This document currently covers NetSprocket in the following chapters:

- Chapter 2, "NetSprocket Reference,"contains a complete programming reference, documenting the functions, data types, and constants available with NetSprocket.
- Appendix A, "Unimplemented or Unused Functions and Data Types," describes NetSprocket functions and types that are currently unused or unimplemented.
- Appendix B, "Document Version History," describes changes made from previous versions of NetSprocket documentation.

For additional information about creating games for the Macintosh, you should check the Apple Developer games Web site:

<http://developer.apple.com/games/>

C H A P T E R

NetSprocket Reference

Contents

Functions 9
- Initializing NetSprocket 10
  - NSpInitialize 10
- Human Interface Functions 12
  - NSpDoModalJoinDialog 12
  - NSpDoModalHostDialog 14
- Hosting and Joining a Game 15
  - NSpGame_Host 16
  - NSpGame_Join 18
  - NSpGame_EnableAdvertising 19
  - NSpGame_Dispose 20
  - NSpGame_GetInfo 21
  - NSpInstallJoinRequestHandler 21
- Sending and Receiving Messages 22
  - NSpMessage_Send 23
  - NSpMessage_SendTo 24
  - NSpMessage_Get 25
  - NSpMessage_Release 26
  - NSpInstallAsyncMessageHandler 27
- Managing Network Protocols 27
  - NSpProtocol_Dispose 28
  - NSpProtocolList_New 29
  - NSpProtocolList_Dispose 29
  - NSpProtocolList_Append 30
  - NSpProtocolList_Remove 31
  - NSpProtocolList_RemoveIndexed 31
  - NSpProtocolList_GetCount 32

NSpProtocolList_GetIndexedRef 32
NSpProtocol_CreateAppleTalk 33
NSpProtocol_CreateIP 34
Managing Player Information 35
NSpPlayer_ChangeType 36
NSpPlayer_Remove 37
NSpPlayer_GetAddress 37
NSpPlayer_GetMyID 38
NSpPlayer_GetInfo 38
NSpPlayer_ReleaseInfo 39
NSpPlayer_GetEnumeration 40
NSpPlayer_ReleaseEnumeration 40
NSpPlayer_GetThruput 41
Managing Groups of Players 42
NSpGroup_New 42
NSpGroup_Dispose 43
NSpGroup_AddPlayer 44
NSpGroup_RemovePlayer 44
NSpGroup_GetInfo 45
NSpGroup_ReleaseInfo 46
NSpGroup_GetEnumeration 46
NSpGroup_ReleaseEnumeration 47
Utility Functions 47
NSpGetVersion 48
NSpSetConnectTimeout 48
NSpClearMessageHeader 49
NSpGetCurrentTimeStamp 49
NSpConvertOTAddrToAddressReference 50
NSpConvertAddressReferenceToOTAddr 51
NSpReleaseAddressReference 51
Application-Defined Functions 52
MyJoinRequestHandler 52
MyMessageHandler 54
Data Types 55
NSpGameID 56
NSpPlayerID 56
NSpGroupID 57
NSpPlayerType 57

NSpFlags 58
NSpPlayerName 58
NSpGameReference 58
NSpProtocolReference 59
NSpListReference 59
NSpAddressReference 59
NSpJoinRequestHandlerProcPtr 60
NSpMessageHandlerProcPtr 60
NSpPlayerInfo 61
NSpPlayerEnumeration 61
NSpGroupInfo 62
NSpGroupEnumeration 63
NSpGameInfo 63
NSpMessageHeader 64
NSpErrorMessage 66
NSpJoinRequestMessage 66
NSpJoinApprovedMessage 67
NSpJoinDeniedMessage 68
NSpPlayerJoinedMessage 69
NSpPlayerLeftMessage 69
NSpGameTerminatedMessage 70
NSpCreateGroupMessage 71
NSpDeleteGroupMessage 71
NSpAddPlayerToGroupMessage 72
NSpRemovePlayerFromGroupMessage 73
NSpPlayerTypeChangedMessage 73
Constants 74
Maximum String Length Constants 75
Network Message Priority Flags 75
Network Message Delivery Flags 76
Options for Hosting, Joining, and Disposing Games 78
Network Message Types 78
Reserved Player IDs for Network Messages 80
Topology Types 81
Summary of NetSprocket 82
NetSprocket Functions 82
Application-Defined Functions 86
Data Types 87

Constants 92
Result Codes 93

NetSprocket Reference

This chapter describes the NetSprocket application programming interface (API) introduced with NetSprocket 1.7. This chapter contains the following sections:

■ "Functions" (page 9)

■ "Data Types" (page 55)

■ "Constants" (page 74)

■ "Result Codes" (page 93)

Some NetSprocket functions and data types that appear in the `NetSprocket.h` header file are currently unimplemented or unused. For descriptions of these APIs, see Appendix A.

Note
This document describes version 1.7 of NetSprocket. For a list of functions changed or added between versions 1.0 and 1.7, see Appendix B. ◆

Functions

This section describes the functions provided by NetSprocket. They are organized according to the following categories:

■ "Initializing NetSprocket" (page 10)

■ "Human Interface Functions" (page 12)

■ "Hosting and Joining a Game" (page 15)

■ "Managing Network Protocols" (page 27)

■ "Managing Player Information" (page 35)

■ "Managing Groups of Players" (page 42)

■ "Utility Functions" (page 47)

■ "Application-Defined Functions" (page 52)

IMPORTANT
With the exception of the function `NSpMessage_Get` (page 25), you should not assume that NetSprocket functions are interrupt-safe. If you must call a NetSprocket function at interrupt time, you should do so through the Deferred Task Manager. ▲

Initializing NetSprocket

You use the function in this section to initialize NetSprocket.

- `NSpInitialize` (page 10) initializes the NetSprocket library.

NSpInitialize

Initializes 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) + 50 KB safety padding). 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; the more frequently you check for messages, the fewer queue elements you need to allocate. 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.
`inGameID`	A unique identifier for your game, typically your application's creator ID. For instance, if you do not specify an NBP (Name Binding Protocol) type to NetSprocket when registering a game on an AppleTalk network, it will use this ID instead.
`inTimeout`	Currently unused. Pass 0 for this parameter.
function result	A result code. See "Result Codes" (page 93).

DISCUSSION

You must initialize NetSprocket before you can call functions 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.

VERSION NOTES

Introduced with NetSprocket 1.0.

Human Interface Functions

NetSprocket provides human interface functions you can use to speed prototyping and to allow players to host and join games.

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` (page 12) presents to the user a default dialog box for finding and joining a game advertised on the network.
- `NSpDoModalHostDialog` (page 14) presents your user with a default modal dialog box for hosting a game on the network.

NSpDoModalJoinDialog

Presents to the user 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. If you pass `NULL` or an empty string, then NetSprocket uses the game ID (as passed to `NSpInitialize` (page 10)) to search for games on the AppleTalk network.

`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 string pointed to by `ioName` 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 the protocol address selected by the user.

DISCUSSION

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. You should then pass this reference to the function `NSpGame_Join` (page 18). Once you have called `NSpGame_Join`, call `NSpReleaseAddressReference` (page 51) to release the reference.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpDoModalHostDialog

Presents your user with a default modal dialog box 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.

DISCUSSION

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.

VERSION NOTES

Introduced with NetSprocket 1.0.

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` (page 16) creates a new game object that other players can then join.
- `NSpGame_Join` (page 18) joins a game specified by an address.
- `NSpGame_EnableAdvertising` (page 19) enables or disables advertising of the game on the network.
- `NSpGame_Dispose` (page 20) removes a player or host from the game.
- `NSpGame_GetInfo` (page 21) obtains information about the game to join.
- `NSpInstallJoinRequestHandler` (page 21) installs the application=defined join request handler

NSpGame_Host

Creates a new game object that other players can then join.

```
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 password that prospective players must match to join the game. Players who do not enter a correct password will not be allowed to join. Pass `NULL` if you do not require a password for players joining your game.
`inPlayerName`	The name of the player hosting the game. If there is no player associated with the computer hosting the game (for example, if the computer is a dedicated game server), you should pass `NULL`.

`inPlayerType`	The player type, which is used only if there is a player associated with 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 currently permissible value of `inFlags` in NetSprocket 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. See "Result Codes" (page 93).

DISCUSSION

You use this function when your application hosts a game.

Once the game is created, the game will automatically be advertised over the protocols in the protocol 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 for joining games; you should use the `NSpGame_Join` (page 18) function instead.

`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.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGame_Join

Joins a game specified by an address.

```
OSStatus NSpGame_Join (
                     NSpGameReference *outGame,
                     NSpAddressReference inAddress,
                     ConstStr31Param inName,
                     ConstStr31Param inPassword,
                     NSpPlayerType inType,
                     Uint32 inCustomDataLen,
                     void *inCustomData,
                     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 return.
`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 in classifying players. It is stored, but not used by NetSprocket.
`inCustomDataLen`	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.
`inCustomData`	A pointer to custom data being sent to the host for use by your custom authentication function. 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 as of NetSprocket version 1.7.
function result	A result code. See "Result Codes" (page 93).

DISCUSSION

This function joins the game specified by the `inAddress` parameter. You can obtain an address reference from `NSpDoModalJoinDialog` (page 12) or `NSpConvertOTAddrToAddressReference` (page 50).

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGame_EnableAdvertising

Enables or disables advertising of the game on the network.

```
OSStatus NSpGame_EnableAdvertising (
                    NSpGameReference inGame,
                    NSpProtocolReference inProtocol,
                    Boolean inEnable);
```

`inGame`	An opaque reference to your game object.
`inProtocol`	An opaque reference to the protocol for which you wish to start or stop advertising. Pass `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. See "Result Codes" (page 93).

DISCUSSION

The function `NSpGame_Host` (page 16) automatically advertises the game, unless you passed `kNSpGameFlag_DontAdvertise` in its `inFlags` field.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGame_Dispose

Removes a player or host from the game.

```
OSStatus NSpGame_Dispose (NSpGameReference inGame,
                          NSpFlags inFlags);
```

`inGame`	An opaque reference to your game object.
`inFlags`	Options for leaving the game. See "Options for Hosting, Joining, and Disposing Games" (page 78) for a list of possible values.
function result	A result code. See "Result Codes" (page 93).

DISCUSSION

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_Dispose` 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.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGame_GetInfo

Obtains information about an available game.

```
OSStatus NSpGame_GetInfo (
                        NSpGameReference inGame,
                        NSpGameInfo *ioInfo);
```

`inGame`	A reference to the game you want to obtain information about.
`ioInfo`	On return, a pointer to game information. See `NSpGameInfo` (page 63) for the format of the returned information.
function result	A result code. See "Result Codes" (page 93).

DESCRIPTION

If you are running a server capable of hosting multiple games, then you could use this function to display information about each available game. Similarly, you could use this function on a player's computer to obtain and display available games to join.

VERSION NOTES

Introduced with NetSprocket 1.7.

NSpInstallJoinRequestHandler

Installs the application-defined join request handler.

```
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.

DISCUSSION

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 whenever a join request occurs. You do not need to develop and install custom join request handlers if the NetSprocket functions already meet your requirements.

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` (page 66) sent by the player who wants to join. You must decide whether or not to allow the player to join, based on whatever criteria you desire. Your function must return a Boolean value to indicate whether the player can join the game.

After your custom join request handler has been installed, any subsequent join requests will be passed to this function for processing.

Also note that 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.

VERSION NOTES

Introduced with NetSprocket 1.0.

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` (page 23) delivers a message to other players in the game.
- `NSpMessage_SendTo` (page 24) creates a message header and sends a message to other players in the game.

- `NSpMessage_Get` (page 25) receives messages that have been delivered to your game.
- `NSpMessage_Release` (page 26) releases a message obtained by calling `NSpMessage_Get`.
- `NSpInstallAsyncMessageHandler` (page 27) installs a message handler for your game object.

NSpMessage_Send

Delivers 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. To impose a reasonable amount of type-safety, you must pass `&myStruct.headerField` 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. See "Network Message Priority Flags" (page 75) and "Network Message Delivery Flags" (page 76) for more information.
function result	A result code. See "Result Codes" (page 93).

DISCUSSION

Before calling this function, you must fill out the message header and message. To send a message and have the message header created for you, call the function `NSpMessage_SendTo` (page 24) instead.

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 that `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.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpMessage_SendTo

Creates a message header and sends a message to other players in the game.

```
OSStatus NSpMessage_SendTo (
                     NSpGameReference inGame,
                     NSpPlayerID inTo,
                     SInt32 inWhat,
                     void * inData,
                     UInt32 inDataLen,
                     NSpFlags inFlags);
```

`inGame`	An opaque reference to your game object.
`inTo`	The ID of the player to whom you want to send the message.
`inWhat`	An integer indicating the type of message to be sent. See "Network Message Types" (page 78) for a listing of possible types.
`inData`	A pointer to the message to send.
`inDataLen`	The length of the message in bytes.

`inFlags`	Flags that specify how the message should be sent. See “Network Message Priority Flags” (page 75) and “Network Message Delivery Flags” (page 76) for more information.
function result	A result code. See “Result Codes” (page 93).

DESCRIPTION

Unlike the `NSpMessage_Send` (page 23) function, `NSpMessage_SendTo` creates a message header based on the information you pass to it. Otherwise it functions identically to `NSpMessage_Send`.

VERSION NOTES

Introduced with NetSprocket 1.7.

NSpMessage_Get

Receives messages that have been delivered to your game.

```
NSpMessageHeader *NSpMessage_Get (
                        NSpGameReference inGame);
```

`inGame`	An opaque reference to your game object.
function result	A pointer to your incoming message data structure.

DISCUSSION

You can use this function to retrieve and process messages whether you are a player in the game or you are hosting a game.

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.

You should call NSpMessage_Get as frequently as you can to get messages that have been sent to your player.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpMessage_Release

Releases a message obtained by calling 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.

DISCUSSION

When you have finished processing a message, you should call NSpMessage_Release to release the memory allocated for it.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpInstallAsyncMessageHandler

Installs a message handler for your game object.

```
OSStatus NSpInstallAsyncMessageHandler (
                        NSpMessageHandlerProcPtr inHandler,
                        void *inContext);
```

`inHandler`	A pointer to your message handling function. See `MyMessageHandler` (page 54) for more information about implementing this function.
`inContext`	The pointer that NetSprocket will pass to your message handling function.
function result	A result code. See "Result Codes" (page 93).

DISCUSSION

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. The message handler is called whenever NetSprocket receives an incoming message.

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.

VERSION NOTES

Introduced with NetSprocket 1.0.

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_Dispose` (page 28) deletes a protocol reference.

- `NSpProtocolList_New` (page 29) creates a new list for storing multiple protocol references.
- `NSpProtocolList_Dispose` (page 29) deletes a protocol list.
- `NSpProtocolList_Append` (page 30) adds a new protocol reference to the list.
- `NSpProtocolList_Remove` (page 31) removes a protocol reference from the list.
- `NSpProtocolList_RemoveIndexed` (page 31) removes the protocol reference at a specific location in the list.
- `NSpProtocolList_GetCount` (page 32) returns the number of protocol references in the list.
- `NSpProtocolList_GetIndexedRef` (page 32) receives the protocol reference at the indicated location in the list.
- `NSpProtocol_CreateAppleTalk` (page 33) Creates an AppleTalk protocol reference using the specified parameters.
- `NSpProtocol_CreateIP` (page 34) creates an IP protocol reference.

NSpProtocol_Dispose

Deletes a protocol reference.

```
void NSpProtocol_Dispose (
                    NSpProtocolReference inProtocolRef);
```

`inProtocolRef` An opaque reference to the protocol being deleted.

DISCUSSION

You should use this function to delete a protocol reference you created (for example, by calling `NSpProtocol_CreateIP` (page 34)).

Note that if 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.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpProtocolList_New

Creates a new list for storing multiple protocol references.

```
OSStatus NSpProtocolList_New (
                    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. See "Result Codes" (page 93).

DISCUSSION

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.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpProtocolList_Dispose

Deletes a protocol list.

```
void NSpProtocolList_Dispose (NSpProtocolListReference inProtocolList);
```

`inProtocolList`
An opaque reference to a list of protocols. When you use `NSpProtocolList_Dispose` to delete a protocol list, all the protocol references in it are deleted.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpProtocolList_Append

Adds 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. See “Result Codes” (page 93).

DISCUSSION

The specified protocol reference is appended to the list of protocol references. Note that after appending, the reference becomes the property of the list; you cannot call `NSpProtocol_Dispose` (page 28) to delete a protocol reference in the list.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpProtocolList_Remove

Removes 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. See "Result Codes" (page 93).

DISCUSSION

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_Dispose` (page 28).

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpProtocolList_RemoveIndexed

Removes 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. The index is zero-based.
function result	A result code. See "Result Codes" (page 93).

DISCUSSION

This function is usually used in conjunction with the `NSpProtocolList_GetCount` (page 32) function for stepping through a protocol list and removing a specific protocol reference.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpProtocolList_GetCount

Returns 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.

DISCUSSION

Use this function when iterating through the protocol list.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpProtocolList_GetIndexedRef

Receives the protocol reference at the indicated location in the list.

```
NSpProtocolReference NSpProtocolList_GetIndexedRef (
                    NSpProtocolListReference inProtocolList,
                    UInt32 inIndex);
```

`inProtocolList`	An opaque reference to a list of protocols.
`inIndex`	A valid index entry. The index is zero-based.
function result	The protocol reference at the specified index.

DISCUSSION

`NSpProtocolList_GetIndexedRef` does not remove the protocol from the list, so you must not delete its reference.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpProtocol_CreateAppleTalk

Creates an AppleTalk protocol reference using the specified parameters.

```
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.

DISCUSSION

Use this function if you wish to preconfigure the AppleTalk protocol before calling `NSpDoModalHostDialog`, or if you want to host the game programmatically.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpProtocol_CreateIP

Creates an IP protocol reference.

```
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 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 to use when joining a game.

`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 same level 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 same level 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.

DISCUSSION

Use this function if you wish to preconfigure the TCP/IP protocol before calling `NSpDoModalHostDialog` (page 14) or if you want to host the game programmatically.

Note that NetSprocket creates both TCP and UDP endpoints. System messages and messages with the Registered flag set are sent using TCP; all others are sent using UDP.

VERSION NOTES

Introduced with NetSprocket 1.0.

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_ChangeType` (page 36) changes the player's type.
- `NSpPlayer_Remove` (page 37) removes a player.
- `NSpPlayer_GetAddress` (page 37) obtains a player's network address.
- `NSpPlayer_GetMyID` (page 38) obtains the ID of the player associated with the game object on the current computer.

- `NSpPlayer_GetInfo` (page 38) obtains information about a player.
- `NSpPlayer_ReleaseInfo` (page 39) releases a player information structure obtained by the `NSpPlayer_GetInfo` function.
- `NSpPlayer_GetEnumeration` (page 40) takes a snapshot that describes each player currently in the game.
- `NSpPlayer_ReleaseEnumeration` (page 40) releases the player enumeration structure.
- `NSpPlayer_GetThruput` (page 41) determines the data throughput between the caller and the specified player.

NSpPlayer_ChangeType

Changes the player's type.

```
OSStatus NSpPlayer_ChangeType (
                    NSpGameReference inGame,
                    NSpPlayerID inPlayerID,
                    NSpPlayerType inNewType);
```

`inGame`	An opaque reference to your game object.
`inPlayerID`	The ID of the player whose player type you want to change.
`inNewType`	The new type to assign. The player type is an arbitrary integer that you can use to help classify players. For example, in a particular game, you may assign a type to indicate players who are wounded or immobilized.
function result	A result code. See "Result Codes" (page 93).

VERSION NOTES

Introduced with NetSprocket 1.7.

NSpPlayer_Remove

Removes a player.

```
OSStatus NSpPlayer_Remove (
                    NSpGameReference inGame,
                    NSpPlayerID inPlayerID);
```

`inGame`	An opaque reference to your game object.
`inPlayerID`	The ID of the player you want to remove.
function result	A result code. See “Result Codes” (page 93).

DESCRIPTION

Unlike the function `NSpGame_Dispose` (page 20), `NSpPlayer_Remove` forcibly removes a player from the game. You can call this function only when the application is hosting the game.

VERSION NOTES

Introduced with NetSprocket 1.7.

NSpPlayer_GetAddress

Obtains a player’s network address.

```
OSStatus NSpPlayer_GetAddress (
                    NSpGameReference inGame,
                    NSpPlayerID inPlayerID,
                    OTAddress ** outAddress);
```

`inGame`	An opaque reference to your game object.
`inPlayerID`	The ID of the player whose network address you want to determine.
`outAddress`	On return, a pointer to the TCP/IP or Appletalk `OTAddress` of the player, as returned by Open Transport.

function result A result code. See “Result Codes” (page 93).

DESCRIPTION

You can call the function `NSpConvertOTAddrToAddressReference` (page 50) to convert the returned `OTAddress` to an address of type `NSpAddressReference`. Note however, that to release the memory associated with the address, you must call `DisposePtr`, not `NSpReleaseAddressReference` (page 51).

VERSION NOTES

Introduced with NetSprocket 1.7.

NSpPlayer_GetMyID

Obtains the ID of the player associated with the game object on the current computer.

```
NSpPlayerID NSpPlayer_GetMyID  (NSpGameReference inGame);
```

`inGame`	An opaque reference to your game object.
function result	A valid player ID. NetSprocket returns 0 if there is no player associated with the game object.

NSpPlayer_GetInfo

Obtains 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. See "Result Codes" (page 93).

DISCUSSION

When you are done with the player's information, you should call `NSpPlayer_ReleaseInfo` (page 39) to release memory associated with the structure.

NetSprocket returns an error if it could not obtain the player's information.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpPlayer_ReleaseInfo

Releases a player information structure obtained by the `NSpPlayer_GetInfo` (page 38) function.

```
void NSpPlayer_ReleaseInfo (
                    NSpGameReference inGame,
                    NSpPlayerInfoPtr inInfo);
```

`inGame`	An opaque reference to your game object.
`inInfo`	The information structure you want to release.

DISCUSSION

You should use the `NSpPlayer_ReleaseInfo` function to release each player information structure obtained by `NSpPlayer_GetInfo` (page 38).

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpPlayer_GetEnumeration

Takes a snapshot that describes each player currently in the 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. See “Result Codes” (page 93).

DISCUSSION

`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.

If there was a problem getting the player information, NetSprocket returns an error; in such cases the value of `outPlayers` is invalid.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpPlayer_ReleaseEnumeration

Releases 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`.

DISCUSSION

For each `NSpPlayer_GetEnumeration` (page 40) call, you should execute a corresponding `NSpPlayer_ReleaseEnumeration` call to release the player enumeration structure when you no longer need it.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpPlayer_GetThruput

Determines the data throughput between the caller and the specified player.

```
UInt32 NSpPlayer_GetThruput (
                NSpGameReference inGame,
                NSpPlayerID inPlayer);
```

`inGame`	An opaque reference to your game object.
`inPlayer`	The ID of the player you are sending the test message to.
function result	The throughput between the caller and the player. Throughput is measured in bytes per second.

DISCUSSION

This function is synchronous. That is, it blocks until it finishes testing throughput unless the timeout 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.

VERSION NOTES

Introduced with NetSprocket 1.0.

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_New` (page 42) creates a new group of players.
- `NSpGroup_Dispose` (page 43) removes a group from the game.
- `NSpGroup_AddPlayer` (page 44) adds a player from a group.
- `NSpGroup_RemovePlayer` (page 44) removes a player from a group.
- `NSpGroup_GetInfo` (page 45) obtains the group's information structure.
- `NSpGroup_ReleaseInfo` (page 46) releases memory held by the group information structure.
- `NSpGroup_GetEnumeration` (page 46) obtains a list of the groups in the game.
- `NSpGroup_ReleaseEnumeration` (page 47) releases memory held by the group enumeration structure.

NSpGroup_New

Creates a new group of players.

```
OSStatus NSpGroup_New (
                        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. See "Result Codes" (page 93).

DISCUSSION

Once a group is created, the value in the `outGroupID` parameter is distributed to each player in the game. This group ID value is independent of the network transport used. Any player in the game can use the `outGroupID` parameter to send messages to the players in the group.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGroup_Dispose

Removes a group from the game.

```
OSStatus NSpGroup_Dispose (
                    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. See “Result Codes” (page 93).

DISCUSSION

`NSpGroup_Dispose` 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.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGroup_AddPlayer

Adds 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. See “Result Codes” (page 93).

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGroup_RemovePlayer

Removes 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. See “Result Codes” (page 93).

DISCUSSION

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 remove the player from the game. It only removes the player from the list of players in the group.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGroup_GetInfo

Obtains 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. See "Result Codes" (page 93).

DISCUSSION

The group information data structure will be allocated by NetSprocket and the structure will be populated with the group's information. When you have finished with the `NSpGroupInfo` data structure, you should release it by calling `NSpGroup_ReleaseInfo` (page 46).

NetSprocket returns an error if NetSprocket could not build the group information data structure or if the group ID was invalid.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGroup_ReleaseInfo

Releases 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.

DISCUSSION

For each `NSpGroup_GetInfo` call, you should execute a corresponding `NSpGroup_ReleaseInfo` call to release the memory held by the group information structure.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGroup_GetEnumeration

Obtains 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. See “Result Codes” (page 93).

DISCUSSION

For efficient memory management, the group enumeration structure should be released by NetSprocket by calling `NSpGroup_ReleaseEnumeration` (page 47).

NetSprocket returns an error if it could not build the group list.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGroup_ReleaseEnumeration

Releases 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.

DISCUSSION

For each `NSpPlayer_GetEnumeration` (page 40) call, you should execute a corresponding `NSpGroup_ReleaseEnumeration` call to release the memory held by the structure.

VERSION NOTES

Introduced with NetSprocket 1.0.

Utility Functions

The following are some useful functions for use with NetSprocket. These include network performance testing, and some functions associated with Open Transport.

- `NSpGetVersion` (page 48) returns the version of NetSprocket.
- `NSpSetConnectTimeout` (page 48) sets the timeout period to create a new network connection.
- `NSpClearMessageHeader` (page 49) initializes the entire header structure.
- `NSpGetCurrentTimeStamp` (page 49) compares time stamps.
- `NSpConvertOTAddrToAddressReference` (page 50) obtains a NetSprocket `NSpAddressReference` from an Open Transport `OTAddress`.
- `NSpConvertAddressReferenceToOTAddr` (page 51) obtains an Open Transport `OTAddress` from a NetSprocket `NSpAddressReference`.
- `NSpReleaseAddressReference` (page 51) releases memory associated with an address reference allocated by NetSprocket.

NSpGetVersion

Returns the version of NetSprocket.

```
NumVersion NSpGetVersion (void);
```

function result The version of NetSprocket.

VERSION NOTES

Introduced with NetSprocket 1.0.3.

NSpSetConnectTimeout

Sets the timeout period to create a new network connection.

```
void NSpSetConnectTimeout (UInt32 inSeconds);
```

`inSeconds`	The timeout period in seconds. If you pass 0, then NetSprocket will use the default TCP timeout of 4 minutes.

DESCRIPTION

If the timeout exceeds the limit set by this function, then NetSprocket will stop trying to create a connection. This timeout period is applies only to the game making the call.

VERSION NOTES

Introduced with NetSprocket 1.7.

NSpClearMessageHeader

Initializes the entire header structure.

```
void NSpClearMessageHeader (NSpMessageHeader *ioMessage);
```

`ioMessage`	A pointer to the message to be initialized.

DISCUSSION

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.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGetCurrentTimeStamp

Compares time stamps.

```
UInt32 NSpGetCurrentTimeStamp (NSpGameReference inGame);
```

`inGame`	An opaque reference to your game object.
function result	The time value in milliseconds.

DISCUSSION

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.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpConvertOTAddrToAddressReference

Obtains a NetSprocket `NSpAddressReference` from an Open Transport `OTAddress`.

```
NSpAddressReference NSpConvertOTAddrToAddressReference (
                         OTAddress *inAddress);
```

`inAddress`	A valid (TCP/IP or AppleTalk) `OTAddress` returned from Open Transport.
function result	A valid `NSpAddressReference`.

DISCUSSION

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.

SPECIAL CONSIDERATIONS

When you no longer need the address reference, do *not* call `NSpReleaseAddressReference` (page 51) to release it. You must dispose of the original `OTAddress` reference in the usual manner (such as by calling `DisposePtr`).

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpConvertAddressReferenceToOTAddr

Obtains an Open Transport `OTAddress` from a NetSprocket `NSpAddressReference`.

```
OTAddress *NSpConvertAddressReferenceToOTAddr (
                         NSpAddressReference inAddress);
```

`inAddress`	A valid `NSpAddressReference` returned from `NSpDoModalJoinDialog`.
function result	A valid `OTAddress`.

DISCUSSION

Use `NSpConvertAddressReferenceToOTAddr` when you want to use the `NSpDoModalJoinDialog` function and you do not plan to use any other functions provided in NetSprocket, such as networking, group, or player functions.

When you no longer need the address reference, you can call `NSpReleaseAddressReference` (page 51) to release it.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpReleaseAddressReference

Releases memory associated with an address reference allocated by NetSprocket.

```
void NSpReleaseAddressReference (
                         NSpAddressReference inAddress);
```

`inAddress`	A valid `NSpAddressReference` returned from `NSpDoModalJoinDialog` or `NSpConvertOTAddrToAddressReference`.

DISCUSSION

For efficient memory management, you should call `NSpReleaseAddressReference` when your game no longer needs an address reference.

SPECIAL CONSIDERATIONS

You should only call this function to release address references that NetSprocket obtains on your behalf, such as when calling the function `NSpDoModalJoinDialog` (page 12). Address references obtained by other means must be disposed by other means. For example, to release an address reference converted from an `OTAddress`, you should release the memory associated with the address by calling `DisposePtr`.

VERSION NOTES

Introduced with NetSprocket 1.0.

Application-Defined Functions

This section describes functions that you can implement for customizing features.

- `MyJoinRequestHandler` (page 52) customizes the criteria for joining a game.
- `MyMessageHandler` (page 54) supplies custom code for handling incoming messages.

MyJoinRequestHandler

Customizes the criteria for joining a game.

```
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.

DISCUSSION

This is a function that you as the game developer must provide if you are going to provide a custom join request handler. 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.

The purpose of the custom function is to allow more flexibility in controlling access to the game. By default, NetSprocket allows players to join the game based on the password and minimum round-trip time of the prospective player. However, you may want to restrict play to a particular network zone, or you may decide that certain levels of games may be played only by players with a previous score history.

Also note that 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.

VERSION NOTES

Introduced with NetSprocket 1.0.

MyMessageHandler

Supplies custom code for handling incoming messages.

```
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.

DISCUSSION

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` (page 25) 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 (by calling `NSpMessage_Get`)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.

VERSION NOTES

Introduced with NetSprocket 1.0.

Data Types

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.

- `NSpGameID` (page 56)
- `NSpPlayerID` (page 56)
- `NSpGroupID` (page 57)
- `NSpPlayerType` (page 57)
- `NSpFlags` (page 58)
- `NSpPlayerName` (page 58)
- `NSpGameReference` (page 58)
- `NSpProtocolReference` (page 59)
- `NSpListReference` (page 59)
- `NSpAddressReference` (page 59)
- `NSpJoinRequestHandlerProcPtr` (page 60)
- `NSpMessageHandlerProcPtr` (page 60)
- `NSpPlayerInfo` (page 61)
- `NSpPlayerEnumeration` (page 61)
- `NSpGroupInfo` (page 62)
- `NSpGroupEnumeration` (page 63)
- `NSpGameInfo` (page 63)
- `NSpMessageHeader` (page 64)
- `NSpErrorMessage` (page 66)
- `NSpJoinRequestMessage` (page 66)

- `NSpJoinApprovedMessage` (page 67)
- `NSpJoinDeniedMessage` (page 68)
- `NSpPlayerJoinedMessage` (page 69)
- `NSpPlayerLeftMessage` (page 69)
- `NSpCreateGroupMessage` (page 71)
- `NSpDeleteGroupMessage` (page 71)
- `NSpAddPlayerToGroupMessage` (page 72)
- `NSpRemovePlayerFromGroupMessage` (page 73)
- `NSpPlayerTypeChangedMessage` (page 73)

NSpGameID

When calling the function `NSpInitialize` (page 10), you must specify a unique ID that NetSprocket will use to keep track of your game on the network. Such an ID has the following type definition:

```
typedef SInt32 NSpGameID;
```

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpPlayerID

Each player in a game has a unique player ID so NetSprocket can keep track of them on the network. Such a player ID has the following type definition:

```
typedef SInt32 NSpPlayerID;
```

NetSprocket automatically assigns a player ID to each player who joins the game.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGroupID

NetSprocket allows you to organize players into arbitrary groups. Each such group is identified by a group ID, which has the following type definition:

```
typedef NSpPlayerID NSpGroupID;
```

NetSprocket automatically assigns an ID to a group when you call the function `NSpGroup_New` (page 42).

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpPlayerType

Each player in a game can have a player type, which is an arbitrary classification determined by the application. The player type has the following type definition:

```
typedef UInt32 NSpPlayerType;
```

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpFlags

A number of NetSprocket functions (such as `NSpGame_Host` (page 16) and `NSpGame_Join` (page 18)) allow you to specify options by passing constants of type `NSpFlags`. Such constants have the following type definition:

```
typedef SInt32 NSpFlags;
```

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpPlayerName

You can handle player names in NetSprocket by passing a string of type `NSpPlayerName`:

```
typedef Str31 NSpPlayerName;
```

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGameReference

You use the game reference to identify your game to the NetSprocket library. You can obtain a game reference by calling the function `NSpGame_Host` (page 16) or `NSpGame_Join` (page 18).

```
typedef struct OpaqueNSpGameReference *NSpGameReference;
```

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpProtocolReference

You use the protocol reference to identify and configure transport protocols without having to know what the protocol actually is.

```
typedef struct OpaqueNSpProtocolReference *NSpProtocolReference;
```

You obtain a protocol reference by calling the function `NSpDoModalHostDialog` (page 14), `NSpProtocol_CreateAppleTalk` (page 33), or `NSpProtocol_CreateIP` (page 34).

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpListReference

You use the protocol list reference to refer to a list of protocol references. You pass this list to the function `NSpGame_Host` (page 16) to tell NetSprocket which protocols the game is to be hosted (advertised) on.

```
typedef struct OpaqueNSpProtocolListReference *NSpProtocolListReference;
```

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpAddressReference

You use the address reference to manipulate protocol references. You obtain an address reference by calling `NSpDoModalJoinDialog` (page 12) or by converting an Open Transport `OTAddress`.

```
typedef struct OpaqueNSpAddressReference *NSpAddressReference;
```

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpJoinRequestHandlerProcPtr

If you want to supply your own custom code for allowing players to join your game, you can specify an application-defined function to do so. Such a function has the following type definition:

```
typedef pascal Boolean (*NSpJoinRequestHandlerProcPtr) (
    NSpGameReference inGame, NSpJoinRequestMessage *inMessage,
    void* inContext, Str255 outReason);
```

See `MyJoinRequestHandler` (page 52) for more information about how to implement this function.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpMessageHandlerProcPtr

If you want to supply your own custom code for handling incoming messages, you can specify an application-defined function to do so. Such a function has the following type definition:

```
typedef pascal Boolean (*NSpMessageHandlerProcPtr) (NSpGameReference
    inGame, NSpMessageHeader *inMessage, void* inContext);
```

See `MyMessageHandler` (page 54) for more information about how to implement this function.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpPlayerInfo

You use the player information structure to obtain 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`	A player type. This parameter is not used by NetSprocket, but you can use it 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.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpPlayerEnumeration

You use the player enumeration structure to obtain 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.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGroupInfo

You use the group information structure to obtain information about each of the players in a group. It includes the number of players, along with an array of the player IDs. 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.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGroupEnumeration

You use the group enumeration structure to obtain 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.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGameInfo

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. See "Topology Types" (page 81) for a list of possible values.
`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 null string means no password is required.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpMessageHeader

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 (as listed in "Network Message Types" (page 78)) 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. You can pass a player ID or a group ID, or the constants `kNSpAllPlayers` or `kNSpServerOnly`.
`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). Otherwise, you can define your own custom message types (for example, keyboard state, voice transmission, or game map). ◆

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpErrorMessage

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. NetSprocket indicates error messages by passing the constant `kNSpError` in the `what` field of the `NSpMessageHeader` (page 64) structure. The error message structure is defined by the `NSpErrorMessage` data type.

```
typedef struct NSpErrorMessage {
    NSpMessageHeader                    header;
    OSStatus                            error;
} NSpErrorMessage;
```

Field descriptions

`header`	An `NSpMessageHeader` structure.
`error`	A constant of `OSStatus` type describing the error encountered.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpJoinRequestMessage

The join request message structure is a standard NetSprocket network message you can use to notify the hosting application that a player wishes to join a game about to start or one that is in progress. NetSprocket indicates join request messages by passing the constant `kNSpJoinRequest` in the `what` field of the `NSpMessageHeader` (page 64) structure. 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` (page 25) function. See the function `NSpInstallJoinRequestHandler` (page 21) for more information. The join request message structure is defined by the `NSpJoinRequestMessage` data type.

```
typedef struct NSpJoinRequestMessage {
    NSpMessageHeader                    header;
    Str31                               name;
    Str31                               password;
    UInt32                              type;
    UInt32                              customDataLen;
    UInt8                               customData[kVariableLengthArray];
} NSpJoinRequestMessage;
```

Field descriptions

`header`	An `NSpMessageHeader` structure.
`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.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpJoinApprovedMessage

When your application is hosting a game, 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. NetSprocket indicates join approved messages by passing the constant `kNSpJoinApproved` in the `what` field of the `NSpMessageHeader` (page 64) structure.

The join approved message structure is defined by the NSpJoinApprovedMessage data type.

```
typedef struct NSpJoinApprovedMessage {
    NSpMessageHeader                    header;
} NSpJoinApprovedMessage;
```

Field descriptions

header	An NSpMessageHeader structure.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpJoinDeniedMessage

When your application is hosting a game, 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. NetSprocket indicates join denied messages by passing the constant kNSpJoinDenied in the what field of the NSpMessageHeader (page 64) structure. 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	An NSpMessageHeader structure.
reason	A string indicating the explanation for refusing entry into the game.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpPlayerJoinedMessage

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. NetSprocket indicates player joined messages by passing the constant kNSpPlayerJoined in the what field of the NSpMessageHeader (page 64) 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	An NSpMessageHeader structure.
playerCount	The number of players in the game.
playerInfo	Player information structure.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpPlayerLeftMessage

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. NetSprocket indicates player left messages by passing the constant kNSpPlayerLeft in the what field of the NSpMessageHeader (page 64) structure. The player left message structure is defined by the NSpPlayerLeftMessage data type.

```
typedef struct NSpPlayerLeftMessage {
    NSpMessageHeader                    header;
    UInt32                              playerCount;
    NSpPlayerID                          playerID;
    NSpPlayerName                        playerName;
} NSpPlayerLeftMessage;
```

Field descriptions

`header`	An `NSpMessageHeader` structure.
`playerCount`	The number of players left in the game.
`playerID`	A valid player ID.
`playerName`	The name of the player who left.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpGameTerminatedMessage

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. NetSprocket indicates game terminated messages by passing the constant `kNSpGameTerminated` in the `what` field of the `NSpMessageHeader` (page 64) structure. The game terminated message structure is defined by the `NSpGameTerminatedMessage` data type.

```
typedef struct NSpGameTerminatedMessage {
    NSpMessageHeader                     header;
} NSpGameTerminatedMessage;
```

Field descriptions

`header`	An `NSpMessageHeader` structure.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpCreateGroupMessage

NetSprocket uses the `NSpCreateGroupMessage` structure to send a message to all players when a group is created. It indicates group created messages by passing the constant `kNSpGroupCreated` in the `what` field of the `NSpMessageHeader` (page 64) structure. Note that NetSprocket handles this message internally unless you had specified a custom message handler, in which case you can interpret the message in your handler and take any desired actions.

```
struct NSpCreateGroupMessage {
    NSpMessageHeader header;
    NSpGroupID groupID;
    NSpPlayerID requestingPlayer;
};
typedef struct NSpCreateGroupMessage NSpCreateGroupMessage;
```

Field descriptions

`header`	An `NSpMessageHeader` structure.
`groupID`	The ID of the group being created.
`requestingPlayer`	The ID of the player requesting the group creation.

VERSION NOTES

Introduced with NetSprocket 1.7.

NSpDeleteGroupMessage

NetSprocket uses the `NSpDeleteGroupMessage` structure to send a message to all players when a group is removed. It indicates group deleted messages by passing the constant `kNSpGroupDeleted` in the `what` field of the `NSpMessageHeader` (page 64) structure. Note that NetSprocket handles this message internally unless you had specified a custom message handler, in which case you can interpret the message in your handler and take any desired actions.

```
struct NSpDeleteGroupMessage {
    NSpMessageHeader header;
    NSpGroupID groupID;
    NSpPlayerID requestingPlayer;
};
typedef struct NSpDeleteGroupMessage NSpDeleteGroupMessage;
```

Field descriptions

`header`	An `NSpMessageHeader` structure.
`groupID`	The ID of the group being deleted.
`requestingPlayer`	The ID of the player requesting the group deletion.

VERSION NOTES

Introduced with NetSprocket 1.7.

NSpAddPlayerToGroupMessage

NetSprocket uses the `NSpAddPlayerToGroupMessage` structure to send a message to all players when a player is added to a group. It indicates player added to group messages by passing the constant `kNSpPlayerAddedToGroup` in the `what` field of the `NSpMessageHeader` (page 64) structure. Note that NetSprocket handles this message internally unless you had specified a custom message handler, in which case you can interpret the message in your handler and take any desired actions.

```
struct NSpAddPlayerToGroupMessage {
    NSpMessageHeader header;
    NSpGroupID groupID;
    NSpPlayerID player;
};
typedef struct NSpAddPlayerToGroupMessage NSpAddPlayerToGroupMessage;
```

Field descriptions

`header`	An `NSpMessageHeader` structure.
`groupID`	The ID of the group.
`player`	The ID of the player being added.

VERSION NOTES

Introduced with NetSprocket 1.7.

NSpRemovePlayerFromGroupMessage

NetSprocket uses the `NSpRemovePlayerFromGroupMessage` structure to send a message to all players when a player is removed from a group. It indicates player removed grom group messages by passing the constant `kNSpPlayerRemovedFromGroup` in the `what` field of the `NSpMessageHeader` (page 64) structure. Note that NetSprocket handles this message internally unless you had specified a custom message handler, in which case you can interpret the message in your handler and take any desired actions.

```
struct NSpRemovePlayerFromGroupMessage {
    NSpMessageHeader header;
    NSpGroupID groupID;
    NSpPlayerID player;
};
typedef struct NSpRemovePlayerFromGroupMessage
                NSpRemovePlayerFromGroupMessage;
```

Field descriptions

`header`	An `NSpMessageHeader` structure.
`groupID`	The group ID
`player`	The ID of the player being removed.

VERSION NOTES

Introduced with NetSprocket 1.7.

NSpPlayerTypeChangedMessage

NetSprocket uses the `NSpPlayerTypeChangedMessage` structure to send a message indicating that a player's type has changed. It indicates player type changed messages by passing the constant `kNSpPlayerTypeChanged` in the `what` field of the `NSpMessageHeader` (page 64) structure.

```
struct NSpPlayerTypeChangedMessage {
    NSpMessageHeader    header;
    NSpPlayerID         player;
    NSpPlayerType       newType;
};
typedef struct NSpPlayerTypeChangedMessage NSpPlayerTypeChangedMessage;
```

Field descriptions

`header`	An `NSpMessageHeader` structure.
`player`	The ID of the player whose type changed.
`newType`	The new player type.

VERSION NOTES

Introduced with NetSprocket 1.7.

Constants

This section describes the constants provided by NetSprocket.

- "Maximum String Length Constants" (page 75)
- "Network Message Priority Flags" (page 75)
- "Network Message Delivery Flags" (page 76)
- "Options for Hosting, Joining, and Disposing Games" (page 78)
- "Network Message Types" (page 78)
- "Reserved Player IDs for Network Messages" (page 80)
- "Topology Types" (page 81)

Maximum String Length Constants

These constants define the maximum lengths allowed for various strings used in NetSprocket. You should use these constants in place of hardcoded values when checking for string lengths.

```
enum {
    kNSpMaxPlayerNameLen        = 31
    kNSpMaxGroupNameLen         = 31,
    kNSpMaxPasswordLen          = 31,
    kNSpMaxGameNameLen          = 31,
    kNSpMaxDefinitionStringLen  = 255
};
```

Constant descriptions

`kNSpMaxPlayerNameLen`	The maximum length for a player’s name.
`kNSpMaxGroupNameLen`	The maximum length for a group.
`kNSpMaxPasswordLen`	The maximum length for a password (used to join a game).
`kNSpMaxGameNameLen`	The maximum length for the name of the game.
`kNSpMaxDefinitionStringLen`	The maximum allowable string length.

VERSION NOTES

Introduced with NetSprocket 1.0.

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 23).

```
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.

VERSION NOTES

Introduced with NetSprocket 1.0.

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. ◆

VERSION NOTES

Introduced with NetSprocket 1.0.

Options for Hosting, Joining, and Disposing Games

These constants are used to control games. You use these constants in the `inFlags` parameter of the `NSpGame_Host` (page 16), `NSpGame_Join` (page 18), and `NSpGame_Dispose` (page 20) 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.

VERSION NOTES

Introduced with NetSprocket 1.0.

Network Message Types

These constants are used to identify standard message types when passed in a message header. 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,
    kNSpGroupCreated            = kNSpSystemMessagePrefix | 0x00000008,
    kNSpGroupDeleted            = kNSpSystemMessagePrefix | 0x00000009,
    kNSpPlayerAddedToGroup      = kNSpSystemMessagePrefix | 0x0000000A,
    kNSpPlayerRemovedFromGroup  = kNSpSystemMessagePrefix | 0x0000000B,
    kNSpPlayerTypeChanged       = kNSpSystemMessagePrefix | 0x0000000C
};
```

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. This message type is unused as NetSprocket does not currently support host renegotiation.
`kNSpGameTerminated`	The game has been permanently stopped.
`kNSpGroupCreated`	Someone has created a group.
`kNSpGroupDeleted`	Someone has deleted a group.
`kNSpPlayerAddedToGroup`	A player was added to a group.
`kNSpPlayerRemovedFromGroup`	A player was removed from a group.

`kNSpPlayerTypeChanged`	A player's type was changed.

Note
All message types with negative values are reserved for use by Apple Computer, Inc. ◆

VERSION NOTES

Introduced with NetSprocket 1.0.

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 also 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. ◆

VERSION NOTES

Introduced with NetSprocket 1.0.

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` (page 16).

```
typedef enum {
    kNSpClientServer                = 0x00000001
} NSpTopology;
```

Constant descriptions

`kNSpClientServer` Client/server topology.

Note

NetSprocket version 1.0 currently supports only client/server topology. ◆

VERSION NOTES

Introduced with NetSprocket 1.0.

Summary of NetSprocket

NetSprocket Functions

Initializing NetSprocket

```
OSStatus NSpInitialize            (UInt32 inStandardMessageSize,
                                   UInt32 inBufferSize,
                                   UInt32 inQElements,
                                   NSpGameID inGameID,
                                   UInt32 inTimeout);
OSStatus NSpInstallCallbackHandler (NSpCallbackProcPtr inHandler,
                                   void *inContext);
```

Human Interface Functions

```
NSpAddressReference NSpDoModalJoinDialog
                                   (ConstStr31Param inGameType,
                                   ConstStr31Param inEntityListLabel,
                                   Str31 ioName,
                                   Str31 ioPassword,
                                   NSpEventProcPtr inEventProcPtr);
Boolean NSpDoModalHostDialog      (NSpProtocolListReference ioProtocolList,
                                   Str31 ioGameName,
                                   Str31 ioPlayerName,
                                   Str31 ioPassword,
                                   NSpEventProcPtr inEventProcPtr);
```

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_Dispose          (NSpGameReference inGame,
                                   NSpFlags inFlags);

OSStatus NSpGame_GetInfo
                                   (NSpGameReference inGame,
                                   NSpGameInfo * ioInfo);

OSStatus NSpInstallJoinRequestHandler
                                   (NSpJoinRequestHandlerProcPtr inHandler,
                                   void *inContext);
```

Sending and Receiving Messages

```
OSStatus NSpMessage_Send (
                                   NSpGameReference inGame,
                                   NSpMessageHeader *inMessage,
                                   NSpFlags inFlags);

OSStatus NSpMessage_SendTo (
                                   NSpGameReference inGame,
                                   NSpPlayerID inTo,
                                   SInt32 inWhat,
                                   void * inData,
                                   UInt32 inDataLen,
                                   NSpFlags inFlags);
```

```
NSpMessageHeader *NSpMessage_Get  (NSpGameReference inGame);

void NSpMessage_Release           (NSpGameReference inGame,
                                   NSpMessageHeader *inMessage);

OSStatus NSpInstallAsyncMessageHandler
                                   (NSpMessageHandlerProcPtr inHandler,
                                   void *inContext);
```

Managing Network Protocols

```
OSStatus NSpProtocol_New          (const char* inDefinitionString,
                                   NSpProtocolReference *outReference);

void NSpProtocol_Dispose          (NSpProtocolReference inProtocolRef);

OSStatus NSpProtocol_ExtractDefinitionString
                                   (NSpProtocolReference inProtocolRef,
                                   char *outDefinitionString);

OSStatus NSpProtocolList_New      (NSpProtocolReference inProtocolRef,
                                   NSpProtocolListReference *outList);

void NSpProtocolList_Dispose      (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

```
OSStatus NSpPlayer_ChangeType (
                                   NSpGameReference inGame,
                                   NSpPlayerID inPlayerID,
                                   NSpPlayerType inNewType);

OSStatus NSpPlayer_Remove (
                                   NSpGameReference inGame,
                                   NSpPlayerID inPlayerID);

OSStatus NSpPlayer_GetAddress (
                                   NSpGameReference inGame,
                                   NSpPlayerID inPlayerID,
                                   OTAddress ** outAddress);

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_GetThruput       (NSpGameReference inGame,
                                   NSpPlayerID inPlayer,
                                   UInt32 inTimeout);
```

Managing Groups of Players

```
OSStatus NSpGroup_New             (NSpGameReference inGame,
                                   NSpGroupID *outGroupID);

OSStatus NSpGroup_Dispose         (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

```
NumVersion NSpGetVersion (void);

void NSpSetConnectTimeout (UInt32 inSeconds);

void NSpClearMessageHeader        (NSpMessageHeader *ioMessage);

UInt32 NSpGetCurrentTimeStamp     (NSpGameReference inGame);

NSpAddressReference NSpConvertOTAddrToAddressReference
                                   (OTAddress *inAddress);

OTAddress *NSpConvertAddressReferenceToOTAddr
                                   (NSpAddressReference inAddress);

void NSpReleaseAddressReference   (NSpAddressReference inAddress);
```

Application-Defined Functions

```
pascal void MyCallbackHandler (NSpGameReference inGame,
                                   void *inContext,
                                   NSpEventCode inCode,
                                   OSStatus inStatus,
                                   void* inCookie);
```

```
pascal Boolean MyJoinRequestHandler (NSpGameReference inGame,
                                    NSpJoinRequestMessage *inMessage,
                                    void* inContext,
                                    Str255 outReason);
pascal Boolean MyMessageHandler (NSpGameReference inGame,
                                    NSpMessageHeader *inMessage,
                                    void *inContext);
```

Data Types

```
typedef         SInt32          NSpEventCode;

typedef         SInt32          NSpGameID;

typedef         SInt32          NSpPlayerID;

typedef         NSpPlayerID     NSpGroupID;

typedef         UInt32          NSpPlayerType;

typedef         SInt32          NSpFlags;

typedef         Str31           NSpPlayerName;
```

Opaque Game Reference Structures

```
typedef         struct          OpaqueNSpGameReference  *NSpGameReference;

typedef         struct          OpaqueNSpProtocolReference  *NSpProtocolReference;

typedef         struct          OpaqueNSpProtocolListReference
                                                        *NSpProtocolListReference;

typedef         struct          OpaqueNSpAddressReference   *NSpAddressReference;
```

Callback Procedure Pointers

```
typedef pascal void (*NSpCallbackProcPtr) (NSpGameReference inGame,
                                    void *inContext, NSpEventCode inCode, OSStatus
                                    inStatus,
                                    void* inCookie);
```

```
typedef pascal Boolean (*NSpJoinRequestHandlerProcPtr) (
                                   NSpGameReference inGame, NSpJoinRequestMessage
                                   *inMessage,
                                   void* inContext, Str255 outReason);

typedef pascal Boolean (*NSpMessageHandlerProcPtr) (NSpGameReference
                                   inGame, NSpMessageHeader *inMessage, void*
                                   inContext);
```

Player Information Structure

```
typedef struct NSpPlayerInfo {
    NSpPlayerID                        id;
    NSpPlayerType                      type;
    Str31                              name;
    UInt32                             groupCount;
    NSpGroupID                         groups[kVariableLengthArray];
} NSpPlayerInfo, *NSpPlayerInfoPtr;
```

Player Enumeration 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;
```

Group Created Message Structure

```
struct NSpCreateGroupMessage {
    NSpMessageHeader header;
    NSpGroupID groupID;
    NSpPlayerID requestingPlayer;
};
typedef struct NSpCreateGroupMessage NSpCreateGroupMessage;
```

Group Deleted Message Structure

```
struct NSpDeleteGroupMessage {
    NSpMessageHeader header;
    NSpGroupID groupID;
    NSpPlayerID requestingPlayer;
};
typedef struct NSpDeleteGroupMessage NSpDeleteGroupMessage;
```

Player Added to Group Message Structure

```
struct NSpAddPlayerToGroupMessage {
    NSpMessageHeader header;
    NSpGroupID groupID;
    NSpPlayerID player;
};
typedef struct NSpAddPlayerToGroupMessage NSpAddPlayerToGroupMessage;
```

Player Removed From Group Message Structure

```
struct NSpRemovePlayerFromGroupMessage {
    NSpMessageHeader header;
    NSpGroupID groupID;
    NSpPlayerID player;
};
typedef struct NSpRemovePlayerFromGroupMessage NSpRemovePlayerFromGroupMessage;
```

Player Type Changed Message Structure

```
struct NSpPlayerTypeChangedMessage {
    NSpMessageHeader    header;
    NSpPlayerID         player;
    NSpPlayerType       newType;
};
typedef struct NSpPlayerTypeChangedMessage NSpPlayerTypeChangedMessage;
```

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,
    kNSpGroupCreated                              = kNSpSystemMessagePrefix | 0x00000008,
    kNSpGroupDeleted                              = kNSpSystemMessagePrefix | 0x00000009,
    kNSpPlayerAddedToGroup                        = kNSpSystemMessagePrefix | 0x0000000A,
    kNSpPlayerRemovedFromGroup                    = kNSpSystemMessagePrefix | 0x0000000B,
    kNSpPlayerTypeChanged                         = kNSpSystemMessagePrefix | 0x0000000C
};
```

Reserved Player IDs for Network Messages

```
enum {
    kNSpAllPlayers                                        = 0x00000000,
    kNSpServerOnly                                        = 0xFFFFFFFF
};
```

Topology Types

```
typedef enum {
    kNSpClientServer                                      = 0x00000001
} NSpTopology;
```

Result Codes

In addition to the following codes, some NetSprocket functions may also return Open Transport 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
kNSpOTVersionTooOldErr	–30371	The version of Open Transport available is too old to use with NetSprocket
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
kNSpMessageTooBigErr	-30397	The message you wanted to send was too long
kNSpCantBlockErr	-30398	The player you sent a message to is not playing the game
kNSpJoinFailedErr	-30399	The attempt to join the game failed

Unimplemented or Unused Functions and Data Types

This appendix describes NetSprocket functions and types that appear in the `NetSprocket.h` header file but are either unimplemented or otherwise unused.

Functions

NSpProtocol_New

Creates a new protocol reference from a definition string.

```
OSStatus NSpProtocol_New (
                    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. See “Result Codes” (page 93).

DISCUSSION

As the definition strings are currently private, in most cases you should use the helper functions `NSpProtocol_CreateAppleTalk` (page 33) or `NSpProtocol_CreateIP` (page 34) to create a protocol reference.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpProtocol_ExtractDefinitionString

Copies 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 whose definition string you want to obtain.
`outDefinitionString`	The buffer into which the string is copied. You must allocate a buffer of size `kNSpMaxDefinitionStringLen` before calling this function.
function result	A result code. See “Result Codes” (page 93).

DISCUSSION

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, it does not support the creation of protocols with definition strings directly.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpPlayer_GetRoundTripTime

This function is currently unimplemented.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpInstallCallbackHandler

Installs 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. See “Result Codes” (page 93).

DISCUSSION

NetSprocket currently does not handle asynchronous callbacks using this handler. Although you can install this callback handler, it will never get called.

VERSION NOTES

Introduced with NetSprocket 1.0.

MyCallbackFunction

Performs application-defined actions for various asynchronous events.

```
pascal void MyCallbackFunction (
                     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.

DISCUSSION

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.

To install this application-defined function, you must call the function `NSpInstallCallbackHandler` (page 99).

Currently NetSprocket does not make any asynchronous callbacks, so you do not need to implement this function.

VERSION NOTES

Introduced with NetSprocket 1.0.

Data Types

NSpEventCode

When calling your application-defined event handling function, NetSprocket passes a value of type `NSpEventCode` to indicate the type of event that occurred.

```
typedef SInt32 NSpEventCode;
```

No event constants are currently defined.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpCallbackProcPtr

If you want to handle asynchronous events, you can specify an application-defined function to do so. Such a function has the following type definition:

```
typedef pascal void (*NSpCallbackProcPtr) (NSpGameReference inGame,
    void *inContext, NSpEventCode inCode, OSStatus inStatus,
    void* inCookie);
```

See `MyCallbackFunction` (page 99) for more information on how to implement this function.

VERSION NOTES

Introduced with NetSprocket 1.0.

NSpHostChangedMessage

NetSprocket uses the host changed message structure to send a message when the host of a game in progress has been changed. NetSprocket indicates host changed messages by passing the constant `kNSpHostChanged` in the `what` field of the `NSpMessageHeader` (page 64) structure. Currently, NetSprocket does not support host renegotiation, so your game will never receive this message.

The host changed message structure is defined by the `NSpHostChangedMessage` data type.

```
typedef struct NSpHostChangedMessage {
    NSpMessageHeader                    header;
    NSpPlayerID                         newHost;
} NSpHostChangedMessage;
```

Field descriptions

`header`	An `NSpMessageHeader` structure.
`newHost`	The player ID of the new host.

VERSION NOTES

Introduced with NetSprocket 1.0.

Document Version History

This document has had the following releases:

Table B-1 NetSprocket documentation revision history

Version	Notes
October 21, 1999	First seed draft release. This document reflects the changes to NetSprocket since version 1.0 documented in Chapter 4 of the Apple Game Sprockets Guide. A summary of changes is as follows: New functions added: `NSpGame_GetInfo` (page 21), `NSpMessage_SendTo` (page 24), `NSpPlayer_ChangeType` (page 36), `NSpPlayer_Remove` (page 37), `NSpPlayer_GetAddress` (page 37), `NSpGetVersion` (page 48), and `NSpSetConnectTimeout` (page 48). The following functions and data types were moved to Appendix A because they are unused or were never implemented: `NSpProtocol_New` (page 97), `NSpProtocol_ExtractDefinitionString` (page 98), `NSpPlayer_GetRoundTripTime` (page 98), `NSpInstallCallbackHandler` (page 99), `MyCallbackFunction` (page 99), `NSpEventCode` (page 101), `NSpCallbackProcPtr` (page 101), and `NSpHostChangedMessage` (page 102). All `NSpxxx_Create` and `NSpxxx_Delete` functions renamed as `NSpxxx_New` and `NSpxxx_Dispose` respectively. For example, the `NSpProtocol_Create` function is now named `NSpProtocol_New` (page 97). Change made to the function `NSpDoModalJoinDialog` (page 12): if you pass `NULL` or an empty string in the `inGameType` parameter , then NetSprocket uses the game ID (as passed to `NSpInitialize` (page 10)) to search for games on the AppleTalk network. Warning added for `NSpReleaseAddressReference` (page 51): You must call this function only to release address references that were allocated by NetSprocket. For example, you should not attempt to use this function to release a reference converted from an Open Transport `OTAddress`.

Table B-1 NetSprocket documentation revision history

Version	Notes
	New data structures and types added: `NSpCreateGroupMessage` (page 71), `NSpDeleteGroupMessage` (page 71), `NSpAddPlayerToGroupMessage` (page 72), `NSpRemovePlayerFromGroupMessage` (page 73), and `NSpPlayerTypeChangedMessage` (page 73).
	New `playerName` field added to the `NSpPlayerLeftMessage` (page 69) structure. Note that this version of the structure (introduced with version 1.1) is not backwards compatible with NetSprocket builds without the extra field.
	The utility function `NSpClearMessageHeader` (page 49) no longer requires a game reference parameter.
	"Maximum String Length Constants" (page 75) added. These were formerly `#define` values in the `NetSprocket.h` header.
	New constants added to "Network Message Types" (page 78).
	The following result codes are new or have changed:
	`kNSpHostFailedErr` (-30371) replaced by `kNSpOTVersionTooOldErr`.
	`kNSpPortTakenErr` (-30397) replaced by `kNSpMessageTooBigErr`.
	`kNSpNotPlayingErr` (-30398) replaced by `kNSpCantBlockErr`.
	`kNSpJoinFailedErr` (-30399) added.

Index

D

document revision history 103

E

error message structure 66

G

game information structure 63
game terminated message structure 70
group 42
group enumeration structure 63
group information structure 62

H

host changed message structure 102
hosting a game 12

I

initializing NetSprocket 10

J

join approved message structure 67
join denied message structure 68
join request message 66

M

message header structure 64
messages
  receiving 22
  sending 22

N

NetSprocket
  constants for 74–81
  data structures for 55–74
  functions in 9–55
  result codes 93
  summary of 82–93
network
  performance testing 47

O

Open Transport 47

P

performance testing 47
player 35
  enumeration structure 61
  information structure 61
  joined message structure 69
  left message structure 69
protocol reference 27

R

result codes 93
revision history, document 103

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.

WRITER
Jun Suzuki

Special thanks to Chris DeSalvo, Quinn (the Eskimo!) and Jasjeet Thind.

Acknowledgements to Dave Bice, Judy Helfland, Tim Monroe, and Larry Wood, who wrote the previous Game Sprockets guide.