macromedia®

DIRECTOR® 8.5 SHOCKWAVE® STUDIO

Using the Shockwave Multiuser Server and Xtra

macromedia®

Acknowledgments

Writing: Jay Armstrong, Greg Barnett, Stephanie Gowin, Tom Higgins, Marcelle Taylor, and Frank Welsch

Editing: Rosana Francescato, Mlle. Godbois, and Anne Szabla

Production Management: John “Zippy” Lehnus

Multimedia Design and Production: Aaron Begley and Noah Zilberberg

Print and Help Design and Production: Chris Basmajian

Web Editing and Production: Jane Flint DeKoven and Jeff Harmon

Emergency helmsman: Joe Schmitz

Special thanks to David Calaprice, Grace Gellerman, Kraig Mentor, Megy Nascimento, Masayo Noda, Chris Nuuja, Roy Pardi, Jonathon Powers, Meredith Tomlin, and the Director engineering and QA teams.

First Edition: March 2001

Macromedia, Inc.
600 Townsend St.
San Francisco, CA 94103

CONTENTS

CHAPTER 4

CHAPTER 5

CHAPTER 1
Using the Multiuser Xtra

The Macromedia Shockwave Multiuser Server 3 and Xtra allow Director movies running in projectors or in Shockwave and Shockmachine to exchange information with other Director movies over the Internet or smaller networks. The Xtra and a 1000-user version of the server are included with the Director 8.5 Shockwave Studio. The Xtra is also included with Shockwave 8.5.

You can use the Multiuser Server and Xtra to do the following:

- Create a custom chat movie that allows real-time conversation.
- Hold an online meeting with a shared "whiteboard" that each participant can write on.
- Provide a shared presentation, allowing a presenter and an audience to all watch the presentation at the same time.
- Run a multiplayer interactive game.

Movies that use the Multiuser Xtra can exchange information in three basic ways:

- By sending it to the Shockwave Multiuser Server, which then sends it on to the movie or movies it is intended for.
- By establishing peer-to-peer connections directly with other movies.
- By connecting to a text-based server such as a standard mail or Internet Relay Chat server. In order to communicate with a text-based server, you must be familiar with the commands the server understands. You can send these commands to the text-based server in the same way you send other messages.

The Shockwave Multiuser Server and Xtra are two separate components that work together to enable multiuser movies. The server is a separate application that runs on a separate computer. The server can also be run on the same computer as the Director application during development of your movies. The Multiuser Xtra checks messages for errors, prepares them for passage over the network, and then sends them to the server. The server then determines whom the message was intended for and sends it to the appropriate recipient(s). The recipient's Xtra gets the message from the network so that it can be used by the movie.

When using the server, movies can communicate with other instances of the same movie connected to the server (such as a chess movie exchanging player-move information with other instances of the chess movie), or with different movies (such as a chess movie exchanging chat messages with a checkers movie in a virtual game room).

The types of messages your movie sends depends on what you want the movie to do. Movies can share all the types of data that Lingo supports, including strings, integers, floating-point numbers, colors, dates, points, rects, lists, 3D vectors, and 3D transforms. In addition, cast members may be exchanged by using the `media` or `picture of member` cast member properties. This enables chat movies to share pictures of the participants, or collaborators to share diagrams, for example.

As a Lingo Xtra, the Multiuser Xtra extends Lingo by adding new commands and other elements to the Lingo vocabulary. The Xtra is used by writing Lingo scripts that include these commands. The multiuser behaviors included in Director's Behavior Library provide all of the functionality that a basic chat application requires.

In addition to simply passing information from movie to movie, the Shockwave Multiuser Server also provides functions that make it easier to create rich and complex multiuser experiences:

- You can store and retrieve information such as user names or profiles in databases.
- You can create groups in order to organize users in logical ways, such as opposing teams in an adventure game.
- You can assign attributes to those groups, such as a team's score.
- You can send messages directly to the server to get information about the server and the other movies connected to it.
- You can access text files on the server and make use of their contents.
- You can add Lingo scripts to the server that provide server-side logic and multithreading to support your multiuser movies.

New features of the Shockwave Multiuser Server

Version 3 of the Shockwave Multiuser Server introduces important new features for creating rich, flexible multiuser experiences on the Internet.

What's new in version 3

Version 3 of the server includes these enhancements:

- Server-side scripting with Lingo
- Server-side multithreading with Lingo
- Server-side Lingo for reading and writing files on the server computer
- Support for debugging Lingo scripts that run on the server
- Additional support for multiple IP addresses on a client machine
- Support for extending the server's functionality by writing server Xtras

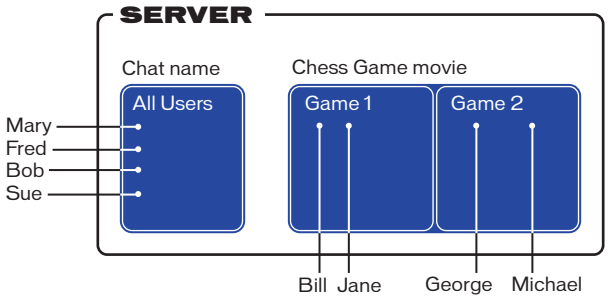
Creating multiuser movies

To create a multiuser experience, you must design a movie that connects to the server or directly to another movie and then sends and receives messages that contain the information needed for the movie to function. For example, you might design a chat movie that connects to the server and then allows each user to send text messages and pictures to all users of the chat movie or to specific users. Or you might design a Ping-Pong game that has two people connect to the server and then sends messages describing the location of each user's paddle and score.

For an example of the Lingo used to connect to the server, see the example Director movies in the Director 8.5/Learning/Lingo_examples/Multiuser_examples folder.

You use Lingo to control how a movie makes connections and handles messages. For those who want to avoid writing their own Lingo, Director includes several behaviors that perform tasks such as making server connections, sending chat messages, and creating a shared whiteboard. For information on using these behaviors, see the Multiuser section of Director's Library palette and its accompanying tooltips.

The following sections describe how to write custom Lingo for managing multiuser connections.



The server manages movies, groups and users. Here the Chat name movie contains one group with four users. The Chess Game movie contains two groups, one for each game, each with two users.

Connecting to the server

To connect to the Shockwave Multiuser Server, you may write Lingo scripts that execute each step of setting up the Multiuser Xtra and establishing the Xtra's communication with the server. For detailed explanations of each Multiuser Xtra Lingo command, see "Multiuser Lingo Dictionary" on page 67. You may also use the multiuser behaviors included in Director's Behavior Library to connect to the server without writing Lingo of your own.

To set up a server connection for sending messages:

1. Make sure you have a working server running on a computer on your network and that you know its network address.
2. Create an instance of the Multiuser Xtra.

   For example, these statements place the Xtra instance into the global variable `gMultiuserInstance`:

   ```
   global gMultiuserInstance
   gMultiuserInstance = new(xtra "Multiuser")
   ```

   A single Xtra instance always corresponds to a single server connection. To make more than one connection at a time, use multiple Xtra instances. When you are ready to disconnect from the server, set the variable containing your Xtra instance to 0.

3 Set up one or more callbacks for handling incoming messages using `setNetMessageHandler()`.

When your movie initially connects to the server, the server responds with a message confirming the connection. In order to handle this and subsequent messages, you must create callbacks. These are Lingo handlers that are run when messages arrive at a movie from the server or from peer-to-peer users. You must create at least one callback before your movie connects to the server. The arrival of a message is treated as a Lingo event in the same way that a mouse click is treated as a `mouseUp` event. Once the callback is declared in Lingo it will be triggered each time a message arrives from the network. A callback handler should be written to perform tasks based on the contents of the incoming message that triggers it.

For example, this statement declares that the handler `defaultMessageHandler` in the script cast member Connection Script is the handler to run when any incoming message is received from the server:

```
errCode = gMultiuserInstance.setNetMessageHandler( \
#defaultMessageHandler, script "Connection Script")
```

The message handler would look like this:

```
on defaultMessageHandler
    global gMultiuserInstance
    newMessage = gMultiuserInstance.getNetMessage()
    member("messageOutput").text = string(newMessage)
    if newMessage.errorCode <> 0 then
       alert "Incoming message contained an error."
    end if
end
```

The first parameter you specify with `setNetMessageHandler()` is a symbol for the handler you are declaring. You make it a symbol by adding a pound sign (#) to the beginning of the handler name. The second parameter is the name of the script object that contains the handler. This could be a script cast member or the name of a variable containing a behavior instance, a parent script, or a simple Lingo value to be passed to global scripts.

You can specify individual handlers to be run based on a particular message subject, a sender, or both by adding optional parameters to `SetNetMessageHandler()`. These individual handlers are run instead of the default message handler when the specified type of message arrives.

You can also add an optional integer parameter to the end of the `setNetMessageHandler()` command to tell Lingo to pass the message contents to the message handler as an argument. This prevents the handler from needing Lingo specifically to get the message out of the incoming message queue, as in the previous example. When you specify a value of `TRUE (1)`, the message contents are passed to the handler as an argument. If you omit this parameter, it defaults to `FALSE (0)` and the message contents are not passed as an argument.

This statement declares a handler that runs when a message containing the subject Chat Text arrives from sender Guest Speaker, and tells Lingo to pass those messages to the handler as arguments:

```
errCode = gMultiuserInstance.setNetMessageHandler( \
#guestMessageHandler, script "Connection Script", "Chat Text",\
"Guest Speaker", True)
```

The simplified message handler would look like this:

```
on guestMessageHandler me, message
    member("messageOutput").text = string(message)
    if message.errorCode <> 0 then
       alert "Incoming message from Guest Speaker contained \
       an error."
    end if
end
```

If you want to include the integer parameter without specifying a subject or a sender, use empty strings for the subject and sender.

```
errCode = gMultiuserInstance.setNetMessageHandler( \
#guestMessageHandler, script "Connection Script", "", "", 1)
```

For information about message subjects, senders, and contents, see "Sending messages" on page 18.

4 Establish the server connection with `connectToNetServer`.

When the message handlers have been declared, you are ready to make your server connection. The Xtra connects to the server with a user name and password, along with other parameters you supply. You must know the server's address and port number as well as the name you want your movie to use to identify itself.

Note: If you expect that your multiuser Shockwave movies will be run primarily on double-byte systems, such as Japanese and Korean, and your movies allow user names and other data to be entered in double-byte text, you should run your server on a double-byte system as well. On roman systems, user names and other properties are not case-sensitive. However, passwords are case sensitive. On Japanese and Korean systems, the server uses double-byte string comparisons. Server passwords can be as long as 250 characters on roman (single-byte) systems and 125 characters on double-byte systems.

The following Lingo connects the movie Tech Chat to a Shockwave Multiuser Server with a user ID of Bob and a password of MySecret. The example server name is chatserver.mycompany.com and the communications port number, 1626, is the default.

```
errCode = gMultiuserInstance.connectToNetServer( \
"chatserver.mycompany.com", 1626, [#userID: "Bob", #password: \
"MySecret", #movieID: "Tech Chat"])
```

The server uses the name you provide for the movie to associate other instances of the same movie with each other. By default, all messages sent by the movie Tech Chat will be sent only to other users of the same movie on the network.

When the server accepts the connection, it will respond with a message whose subject is ConnectToNetServer, which will trigger the `defaultMessageHandler` you declared earlier.

The entire Lingo script for connecting to the server looks like this:

```
on makeAServerConnection
   -- declare a global variable to hold the Xtra instance
   global gMultiuserInstance

   -- create the Xtra instance
   gMultiuserInstance = new(xtra "Multiuser")

   -- declare message handler callback(s) to handle incoming
   -- messages, including the server's initial connection response
   errCode = gMultiuserInstance.setNetMessageHandler( \
   #defaultMessageHandler, script "Connection Script", True)
   if errCode <> 0 then
      alert "Problem with setNetMessageHandler"
   end if

   -- connect to the server
   errCode = gMultiuserInstance.connectToNetServer( \
   "chatserver.mycompany.com", 1626, [#userID: "Bob", #password: \
   "MySecret", #movieID: "Tech Chat"])
   if errCode <> 0 then
      alert "Problem with connectToNetServer"
   end if

end

-- message handler for incoming messages
on defaultMessageHandler newMessage
   member("messageOutput").text = string(newMessage)
   if newMessage.errorCode <> 0 then
      alert "Incoming message contained an error."
   end if
end
```

The `connectToNetServer()` command can also accept its parameters in another format that allows the server to check the Internet location of the movie that is connecting. This provides additional security when distributing movies over the Web.

The second format takes the server name and port, followed by a property list with the user name, password, and movie name, and then optional connection mode and encryption key parameters. The connection mode can be either `#smus` or `#text`. The symbol `#smus` indicates a normal Shockwave Multiuser Server connection, and `#text` indicates a text-mode connection. See "Creating text connections" on page 22.

When `#smus` is specified with the second format, the Multiuser Xtra will include the Internet address of the movie with the user name and password information it sends to the server when logging on. The Internet address that is passed to the server takes the form http://server.company.com/directory/movieName.dcr. This way the server can verify that the movie is connecting from the Internet location that the author intended. For information about configuring the server to make use of the movie location information, see "Using the Multiuser.cfg file" on page 54.

The following Lingo connects the movie Tech Chat to a Shockwave Multiuser Server with a user ID of Bob and a password of MySecret. The example server name is chatserver.mycompany.com and the communications port number, 1626, is the default. The connection mode is Shockwave Multiuser Server. No encryption key is specified.

```
errorCode = myConnection.connectToNetServer( \
"chatserver.mycompany.com", 1626, [#userID:"Bob", \
#password:"MySecret", #movieID:"Tech Chat"], #smus)
```

The following code uses the second format for `connectToNetServer()` and declares two message callback handlers. One will handle generic messages and one will handle messages with a `#subject` property of Chat Text. The `setNetMessageHandler()` calls use the optional `TRUE/FALSE` parameter so the message contents are passed directly to the message handlers.

```
on makeAServerConnection
   -- declare a global variable to hold the Xtra instance
   global gMultiuserInstance

   -- create the Xtra instance
   gMultiuserInstance = new(xtra "Multiuser")

   -- declare message callback handler to handle incoming messages
   -- that don't meet the specific criteria of the second message
   -- callback handler declared below
   errCode = gMultiuserInstance.setNetMessageHandler( \
   #defaultMessageHandler, script"Connection Script", "", "", \
   True)
   if errCode <> 0 then
      alert "Problem with setNetMessageHandler"
   end if

   -- this is the second message callback declaration
   -- declare a specific message callback handler for messages
   -- with #subject Chat Text
```

```
   errCode = gMultiuserInstance.setNetMessageHandler( \
   #chatMessageHandler, script "Connection Script", "Chat Text",\
   "", True)
   if errCode <> 0 then
      alert "Problem with setNetMessageHandler"
   end if

   -- connect to the server
   errCode = gMultiuserInstance.connectToNetServer( \
   "chatserver.mycompany.com", 1626, [#userID: "Bob", #password:\
   "MySecret", #movieID: "Tech Chat"], #smus)
   if errCode <> 0 then
      alert "Problem with connectToNetServer"
   end if
end

-- message handler for generic incoming messages
on defaultMessageHandler me, message
   member("messageOutput").text = string(message)
   if message.errorCode <> 0 then
      alert "Incoming message contained an error."
   end if
end

-- message handler for incoming messages with #subject Chat Text
on chatMessageHandler me, message
   put message.content after member "chatField"
   if message.errorCode <> 0 then
      alert "Incoming message contained an error."
   end if
end
```

Sending messages

When you have successfully connected to the server, you are ready to send messages. Your messages may contain any kind of data that Lingo supports. The messages you receive will take the form of Lingo property lists, and you may choose to send your outgoing messages as property lists as well.

To send any kind of data as a message to another movie, use `sendNetMessage`. This function can take two forms: either a property list or the same parameters, separated by commas.

```
errCode = gMultiuserInstance.sendNetMessage([#recipients: \
whichUsersOrGroups, #subject: "Example Subject", #content: \
whatMessage])
```

or

```
errCode = gMultiuserInstance.sendNetMessage( \
"whichUsersOrGroups", "Example Subject", whatMessage)
```

You can send messages to specific users and to named groups of users in the same movie. For example, user Guest Speaker in the movie Tech Chat could be in New York and send a message to user Bob in the same movie in San Francisco.

```
errCode = sendNetMessage(gMultiuserInstance, "Bob", "Chat Text", \
"Hello. Welcome to our conversation.")
```

To send a message to all the members of a group, use the group name, in this case @MultimediaAuthors, as the recipient parameter. You must begin all group names with the @ sign. For information on creating groups, see “Using groups” on page 24.

```
errCode = gMultiuserInstance.sendNetMessage( \
"@MultimediaAuthors","Chat Text", "How is everyone doing?")
```

You can also send messages to users of other movies connected to the server. A player of a chess game might send a message to a player of a checkers game:

```
errCode = gMultiuserInstance.sendNetMessage("Chris@Checkers", \
"Question", "Who's winning?")
```

To send a message to a user in any other movie on the server, use @AllMovies after the user name. You can use this to page a user to see if he or she is connected to the server at all.

```
errCode = gMultiuserInstance.sendNetMessage("Chris@AllMovies", \
"Page", "Are you there?")
```

Retrieving messages

Your incoming message handlers will receive the message from the Multiuser Xtra as an argument. If you chose not to use the optional parameter that invokes this functionality, use `getNetMessage()` in your message callback handlers.

```
newMessage = gMultiuserInstance.getNetMessage()
```

The contents of `newMessage` are a property list that looks like this:

```
[#errorCode: 0, #recipients: ["Bob"], #senderID: "Guest Speaker",
#subject: "Chat Text", #content: "Hello. Welcome to our
conversation.", #timeStamp: 66114697]
```

You can set up individual message handlers to be triggered by specific values in the `#subject` or `#sender` properties and to perform different actions based on the values of each of the properties in the incoming message.

#errorCode contains an integer that can be translated to a usable error message with `getNetErrorString`.

#recipients contains a list of strings that are the names of the users the message was sent to.

#senderID contains a string that is the user name of the sender.

#subject contains a string chosen by the movie author. You can use this string to indicate what kind of data is in the message and to have the message processed by a specific message callback handler.

#content can contain whatever type of Lingo data your movie requires, including integers, floating-point numbers, strings, lists, rects, points, colors, dates, 3D vectors, 3D transforms, and the `media of member` and `picture of member` properties.

#timeStamp contains an integer that is the number of milliseconds since the server was launched. You can use this number to synchronize events in several instances of your movie or to determine how much time the server is using to process your messages.

Once you have created a simple message callback handler, you can add additional Lingo to use the contents of the message to do specific things in your movie. For instance, you can use the data to set the location of sprites, add new images to the Stage, display chat text, update catalog information, or keep a running score in a game.

The Multiuser Xtra checks the network for incoming messages during idle time. Some complex movies may not allow enough idle time to retrieve all the incoming messages that the movie is receiving. This is often the case in movies that use `go to the frame` Lingo. You can determine how many messages have arrived but have not yet been processed by using the `getNumberWaitingNetMessages()` command, which will give you the number of unprocessed messages. You can then use this number to call `checkNetMessages()`, which will process the number of waiting messages you specify.

Error checking

It is essential to check for errors in your multiuser movies. Latency and stalled connections are common network problems. Servers may be temporarily unavailable because of power failures or other physical malfunctions. Error checking will let you detect and fix problems so that the end-user experience is not affected.

Because most of the commands used with the Shockwave Multiuser Server and Xtra are dependent on one another, it is important to verify that each command used is successful before executing the next one. Many multiuser commands can produce both synchronous and asynchronous results. A synchronous result is one returned immediately by the Xtra, which indicates whether the Xtra was able to correctly execute the command with the parameters you provided. An error code of 0 indicates success.

An asynchronous result is one that may be returned after some operation takes place on the server, such as when your `connectToNetServer` request is accepted or declined. This error code is provided as part of the contents of an incoming message and should be checked in each of your message callback handlers.

The following Lingo script sets the variable `errCode` to the number returned by the Xtra as the immediate result of the `connectToNetServer` command. If one or more of the parameters you provide is unacceptable to the Xtra, it will return a nonzero result and trigger an alert.

```
global gMultiuserInstance
errCode = gMultiuserInstance.connectToNetServer( \
"chatserver.mycompany.com", 1626, [#userID: "Bob", #password: \
"MySecret", #movieID: "Tech Chat"])
if errCode <> 0 then
   alert "Problem with connectToNetServer" & RETURN & \
   gMultiuserInstance.getNetErrorString(errCode)
end if
```

To check the asynchronous results returned from commands as the `#errorCode` property of incoming messages, use a Lingo script such as the following:

```
on defaultMessageHandler newMessage
   global gMultiuserInstance
   member("messageOutput").text = string(newMessage)
   if newMessage.errorCode <> 0 then
      alert "Incoming message contained an error."
   end if
end
```

You should keep checking the error codes in the messages you receive so that you are aware when conditions change in ways that will affect your movie. For instance, other movies on the network may disconnect, or other events may prevent them from communicating efficiently with your movie. When you receive a message containing an error, use `getNetErrorString` to convert the numeric error code into a useful description of the error that occurred.

For example, this handler contains statements that place the string returned by `getNetErrorString()` into a field for the user to see:

```
on defaultMessageHandler newMessage
   global gMultiuserInstance
   errCode = newMessage.errorcode
   member("errorText").text = \
   gMultiuserInstance.getNetErrorString(errCode)
end
```

Setting up peer-to-peer connections

In some situations, you might prefer to design your movies to communicate directly with one another. By using the peer-to-peer functions of the Multiuser Xtra, you can set up one instance of a movie as a virtual server, or peer host, and connect up to 16 other client movies to it. This method lets you set up private chats, create games for limited numbers of users, or make presentations to small groups, without using a server. The disadvantage of using peer connections is that you do not have access to the server's group, database, or administrative functions.

One of the movies in a peer-to-peer connection must be the host. You set up a movie to be a peer host by using `waitForNetConnection`. Once the movie is in peer-host mode, other movies can connect to it in the same way they connect to the server using `connectToNetServer`.

Movies that wish to connect to a peer host must know the Internet address of the computer the host movie is running on. This will be either a number, such as 192.98.168.1, or a name, such as myServer.myCompany.com. You can get the numeric Internet address of your computer with `getNetAddressCookie`. By default, this function returns an encrypted version of your IP address. In addition, `getNetAddressCookie` can return an unencrypted result, but only in projectors and the Director authoring application. One way to make the address available to other users is to meet them on the server and then send them your Internet address cookie before making a peer connection.

You send messages in peer-to-peer mode in the same way as when connected to the server. You can send messages directly to other peer users or to the peer host. Messages sent to other peers are routed to the recipient by the Xtra in the host movie, but their contents are not made available to the host movie. Only messages sent to the host user or to `@AllUsers` are actually received by the host as messages that can be accessed in Lingo. In peer-host mode, the Xtra does not provide any of the server features such as group management or database access.

Peer hosts can control who is connected to their movie by keeping track of the list of connected users and breaking connections if necessary. Using `getPeerConnectionList`, you can get a list of the current peer connections on a peer host. You can sever the connection of a specific user in the list by using `breakConnection`.

When implementing a peer-to-peer movie design, you may choose to make one Director movie that always acts as a host and another Director movie that is used for the clients. You may also choose to author your movie so that it can function as either host or client, allowing anyone who uses the movie to initiate the peer-to-peer session.

Keep in mind that because all peer messages are routed through the peer host, there will be an extra CPU burden and network throughput required on that machine. The degree of this burden will depend on the size and number of messages your movie sends and the number of peers connected to the host. For ways to minimize this load, see "Optimizing multiuser movies" on page 32.

Creating text connections

Another way the Multiuser Xtra can communicate is through text-based servers such as Internet Relay Chat, Internet mail servers, or even proprietary text-based servers. These servers are called text-based because they respond to simple text commands that are used to control them. You make a text-based server connection by adding the integer 1 or the symbol `#text` to the end of the `connectToNetServer()` command, depending on which form of `connectToNetServer()` you are using. This statement tells the Xtra to enter text mode:

```
errCode = gMultiuserInstance.connectToNetServer("Bob", \
"MySecret", "mailserver.mycompany.com", 110, "Tech Chat", 1)
```

or

```
errorCode = gMultiuserInstance.connectToNetServer \
("mailserver.mycompany.com", 110, [#userID: "Bob", #password: \
"MySecret", #movieID: "Tech Chat"], #text)
```

The port number must also change to reflect the port your text-based server is using for communication. In this example the port is 110, which is commonly used with Internet mail servers. Consult the documentation for your text-based server to determine which port it uses.

In Shockwave, if a user tries to make a text-based connection to a text server that resides in a different domain name than the connecting movie, a security dialog box appears on the user's computer.

After the connection is made, you issue commands to your server with `sendNetMessage()`. Use System as the recipient and place your command into the content parameter. The subject parameter will be ignored. The following example sends a command to a mail server to retrieve the first piece of mail in the mailbox.

```
command = "RETR 1" & RETURN
errCode = gMultiuserInstance.sendNetMessage("System", \
"anySubject", command)
```

The server responds with whatever data is appropriate. The Xtra retrieves the data as the `#content` property of a typical incoming message property list.

Using server functions

You can retrieve many kinds of information from the Shockwave Multiuser Server and perform administrative functions by sending special commands to the server in the contents of `sendNetMessage()`. You can get a list of movies connected to the server, control users' and movies' access to the server, and get connected users' IP addresses. Using these functions, you can design a movie that lets you remotely monitor and control activity on the server.

The server commands themselves are sent in the recipient parameter of a `sendNetMessage()`, and any additional parameters required by the command are sent as the content. To determine the version of the server you are using and the platform it is running on, use `getVersion`:

```
errCode = gMultiuserInstance.sendNetMessage \
("system.server.getVersion", "anySubject")
```

When you use server functions, the recipient parameter uses a simple dot syntax that contains a reference to the server (since this is a server command), the object the command will be performed on, and the command itself. In this case the object is the server; in other cases, it could be a user or a movie.

The message returned from the server containing the version and platform information looks like this:

```
[#errorCode: 0, #recipients: ["Bob"], #senderID:
"system.server.getVersion", #subject: "anySubject", #content:
[#vendor: "Macromedia", #version: "3.0", #platform: "Macintosh"]]
```

The subjects of these command messages sent to the server are arbitrary and can be set to any string you want to use. The server's responses will contain the same subject string. You can use the subject string to invoke specific message handlers by specifying the subjects in your `setNetMessageHandler()` commands. See "Connecting to the server" on page 12.

To get a list of all the movies connected to the server, use `getMovies`:

```
errCode = gMultiuserInstance.sendNetMessage\
("system.server.getMovies", "anySubject")
```

To prevent all instances of a particular movie from connecting to the server, use `movie` as the object and `disable` as the command. Here the movie being disabled is Tech Chat:

```
errCode = gMultiuserInstance.sendNetMessage\
("system.movie.disable", "anySubject", "Tech Chat")
```

To disconnect a specific user from the server, the object is `user` and the command is `delete`:

```
errCode = gMultiuserInstance.sendNetMessage("system.user.\
delete", "anySubject", "RudePerson")
```

Server commands, including `getMovies`, `disable`, and `delete`, require the sender to have a specific user level. The required user levels for each command are set in the server's Multiuser.cfg file. For more information, see "Configuring the server" on page 54.

Some server commands can be used to return information about a movie other than the one that sends the message. If you want to get the list of groups in a movie called Chess but you are using an administrative movie called GameAdmin, you can send the command `getGroups` to the movie Chess:

```
errCode = gMultiuserInstance.sendNetMessage\
("system.movie.getGroups@Chess", "anySubject")
```

For a complete list of the server functions that are available, see "Multiuser Lingo Dictionary" on page 67.

Using groups

Another function that the Shockwave Multiuser Server provides is the ability to define groups of connected users. By creating groups, you can organize users in logical ways, such as into opposing teams, collaborating companies, and so on.

Using groups allows you to do the following:

- Send a single message to all the members of a group.
- Prevent users from sending messages to users outside their own group.
- Assign attributes to the groups, such as scores, team colors, or group leaders.

Defining groups

When users connect to the server, they are automatically added to the group @AllUsers for the movie they are using. This lets you easily send messages to everyone connected to the movie without having to create a new group and have everyone join it individually.

You create a new group by having one or more connected users send a message to the server specifying `group` as the object and `join` as the command in the recipient parameter of a `sendNetMessage()` and sending the new group name in the `#content` parameter. If the group does not exist when the first user joins, the server creates the group for you. Once the group has some members, you can ask the server for the number of members in the group, for a list of the members, or for any attributes that have been assigned to the group.

Once a movie has connected a user to the server, the user can join an existing group by sending a message to the server specifying `group` as the object and `join` as the command in the recipient parameter of a `sendNetMessage()` and sending the group name in the `#content` parameter. The following message joins a user to the group called @RedTeam.

```
errCode = gMultiuserInstance.sendNetMessage("system.group.join", ¬
"anySubject", "@RedTeam")
```

It is important that all your group names begin with the @ symbol. The server uses this symbol to distinguish group names from user names when routing standard messages like this one:

```
errCode = gMultiuserInstance.sendNetMessage("@RedTeam", "Status ¬
Update", "Your team is winning.")
```

If the recipient were simply RedTeam, the server would attempt to find the user named RedTeam instead of the group. No error message is generated when a user is not found by the server.

You can leave a group by specifying `system.group.leave` in the recipient parameter of a `sendNetMessage()` and indicating which group you want to leave in the `#content` parameter:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.leave", ¬
"anySubject", "@RedTeam")
```

You can determine how many users are in a group by using `getUserCount`. For example, you can keep team sizes even as more users join a game by adding users to whichever team has a smaller number of players. Or you could create a new group when an existing one exceeds a certain number of users.

```
errCode = gMultiuserInstance.sendNetMessage¬
("system.group.getUserCount", "anySubject", "@RedTeam")
```

If you want to get a list of the users in a group, use `getUsers`:

```
errCode = gMultiuserInstance.sendNetMessage¬
("system.group.getUsers", "anySubject", "@RedTeam")
```

Working with group attributes

You may wish to keep track of the traits of a group or set of groups. For example, for groups of people working for two different companies that are collaborating, you might want to store the name of the leader of each group. If the groups are negotiating the price of a contract, you might store the current amount of each group's bid. In a gaming scenario, each team might choose a mascot, a theme song, or a starting location in a virtual world. Each of these can be stored as an attribute of a group as long as the group exists. Attributes can be set to any of the Lingo values that can be sent using `sendNetMessage`. Group attributes persist as long as the group exists on the server. To store information that must persist indefinitely on the server, use databases instead; see "Using databases" on page 28 for more information.

You use `setAttribute` to add an attribute or update its value and `getAttribute` to determine the current value of the attribute. These commands are specified in the `#recipient` parameter of a `sendNetMessage()`, and the attributes you wish to access are specified in the `#content` parameter of the message.

The Lingo to define the current location and mascot of the group @RedTeam looks like this:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.¬
setAttribute", "anySubject", [#group: "@RedTeam", #attribute: ¬
[#currentLocation: "New York City", #mascot: "dalmatian", ¬
#lastUpdateTime: "1999/07/27 15:52:11.123456"]])
```

When you use `setAttribute` the `#content` parameter of your message contains a `#group` property and an `#attribute` property. The `#group` property indicates which group you are setting attributes for. The `#attribute` property indicates which attributes you are setting. The names of the attributes you set should always be symbols beginning with the # character. In the example above, the `#group` is @RedTeam and the `#attribute` is a list of two attributes, `#currentLocation` and `#mascot`.

The `#lastUpdateTime` property is optional and lets you determine whether some other user has updated the attributes of the group since you last checked them with `getAttribute`. When you use `getAttribute` the server responds with the values of the attributes you requested plus a `#lastUpdateTime` property, which indicates the moment in time when the server read the values of those attributes for the group you requested. The `#lastUpdateTime` property is a string containing the year, month, day, hour, minutes, seconds, and microseconds on the server. By sending this same string with your `setAttribute` command, you allow the server to check whether the attributes for the group have been updated since you last checked them.

If the server determines that any attributes for the group have been updated by someone else since you checked them, it responds with an `#errorCode`, indicating a concurrency error. If no one else has updated the attributes since you checked them, the server responds with a new `#lastUpdateTime` for the group, indicating that you have just updated the attributes.

This statement gets the values of the attributes #currentLocation and #mascot for the group @RedTeam:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.¬
getAttribute", "anySubject", [#group: "@RedTeam", #attribute: ¬
[#currentLocation, #mascot]])
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["Bob"], #senderID: "system.group.¬
getAttribute", #subject: "anySubject", #content: ["@RedTeam": ¬
[#currentLocation: "New York City", #mascot: "dalmatian", ¬
#lastUpdateTime: "1999/07/27 15:53:32.123456"]]]
```

You can get attributes for more than one group at a time by including multiple group names in your request:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.¬
getAttribute", "anySubject", [#group: ["@RedTeam", "@BlueTeam"], ¬
#attribute: [#currentLocation, #mascot]])
```

When you make a request for multiple groups, it is possible that part of the request will succeed while part of it produces errors. In the example above, attributes are requested for the group @BlueTeam. If this group has not yet been created, the server response will include errors in both the #errorCode property of the response and the attributes list for @BlueTeam. If a requested attribute has not been set, no error is generated and the attribute is simply omitted from the response.

The server's response looks something like this:

```
[#errorCode: -2147216173, #recipients: ["Bob"], #senderID: ¬
"system.group.getAttribute", #subject: "anySubject", #content: ¬
["@RedTeam": [#currentLocation: "New York City", #mascot: ¬
"dalmatian", #lastUpdateTime: "1999/07/27 15:53:32.123456"], ¬
"@BlueTeam": [#errorCode: [-2147216194]]]]
```

The first of these error codes indicates that the #content property of the message contains errors that you should check. The second error code indicates that the group @BlueTeam does not exist. (See "getNetErrorString()" on page 98 for a list of error codes and their strings.)

If you want to determine exactly what attributes have been declared for a particular group, use getAttributeNames. It returns a list of the attribute names for the group:

```
errCode = gMultiuserInstance.sendNetMessage ("system.group.¬
getAttributeNames", "anySubject", [#group: "@RedTeam"])
```

To remove an attribute from the attribute list for a group, use deleteAttribute. The following example deletes the #mascot property from the group @RedTeam:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.¬
deleteAttribute", "anySubject", [#group: "@RedTeam", #attribute: ¬
#mascot])
```

Keep in mind that the attributes you define for groups will persist only while the group exists. If you want this kind of information to be stored indefinitely so that it can be recalled from one multiuser session to the next, use database commands.

Using databases

The Multiuser Server provides extensive database functionality that allows your movies to store a wide variety of information. When you use databases, the information you store on the server will be available each time your movies connect to the server. You can use databases to do the following:

- Store information about individual users, such as their e-mail address or the type of computer they are using.
- Keep a record of a user's high score in a particular game.
- Keep track of the status of a game or other multiuser application.
- Store environment information for an application, such as map data for an adventure game.

For an example of the Lingo used to work with databases, see the example Director movies in the Director 8.5/Learning/Lingo_examples/Multiuser_examples folder.

Using data objects

The Shockwave Multiuser Server can store information for you in four different types of data objects. A data object is simply a container that holds data that you put into it. You decide what kinds of objects to use and what data to put into them. Which kind of data object you use depends on the kind of information you want to store and what that information will be used for. You can define as many objects of each type as you need to.

The following are the four types of data objects you can use to store information:

DBUser is the type of object you should use if you are storing information about a user that is specific to the user but not to any particular movie they might use, such as an e-mail address.

DBPlayer is used to store information that is specific to both the user and a particular game they are playing, such as their high score. Since a user may use many different multiuser movies with your Multiuser Server, they might have many `DBPlayer` objects but just one `DBUser` object.

DBApplication objects are used to store information that is specific to a particular movie, such as the highest score ever achieved in a particular game.

DBApplicationData objects are appropriate for information that will be read-only, such as map data for an adventure game. Typically there will be more than one `DBApplicationData` object for each `DBApplication`. For example, in a trivia game you might use `DBApplicationData` objects for configuration information such as lists of trivia questions, buzzer and bell sounds, or suits the host might wear. Each of these would be stored in its own `DBApplicationData` object.

Storing information

Once you have decided what types of objects to use, it is time to store your data. You start by creating a new object, and then you add attributes to the object. The attributes will be whatever items of information you want to store, such as an e-mail address or a high score. As with other server functions, these operations are performed by sending commands to the server in the `#recipient` parameter of a `sendNetMessage()` and indicating which type of object you want to create or edit.

Keep in mind that you can set different combinations of attributes for different objects. You could assign user Bob `#email` and `#favoriteFood` attributes but assign user Mary `#email` and `#favoritePlace` attributes. What attributes you define and the information you put into them is up to you and what you want your movie to do. Object attributes can contain any Lingo value you wish to store.

You might want to start by storing user-specific information in a `DBUser` object. Since the `DBUser` object for the new user does not yet exist, you must tell the server to create it for you. This is one of several administrative functions you can perform on the server by specifying `DBAdmin` as the object of your command. To create a new `DBUser` object for a new user, use `createUser` in conjunction with `DBAdmin`:

```
errCode = gMultiuserInstance.sendNetMessage("system.¬
DBAdmin.createUser", "anySubject", [#userID: "Bob", #password: ¬
"MySecret", #userlevel: 20])
```

This creates a new `DBUser` object with a `#userID` property of Bob.

To add new information to the new user's `DBUser` object, you must name the information, which becomes an attribute of the user. If you are storing e-mail addresses, you could call the attribute `#email`. The attribute name should be a symbol preceded by the # sign. You declare a new attribute by using `declareAttribute` with the `DBAdmin` object. Once an attribute has been declared, its name may be used with any of the database object types.

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin. ¬
declareAttribute", "anySubject", [#attribute: #email])
```

Now the attribute named `#email` has been declared and may be set for any of the objects you create. To set the e-mail address in the database that was created earlier for the user Bob, use `setAttribute` with the `DBUser` object:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBUser.¬
setAttribute", "anySubject", [#userID: "Bob", #attribute: ¬
[#email: "bobsmith@companyname.com"]])
```

The `#content` parameter of this `sendNetMessage()` is a nested property list containing the parameters for the `setAttribute` command. The `#userID` property tells the server which user's information is being edited. The `#attribute` property indicates which attributes are being set and is followed by a list of attributes and values. In this case, only one attribute is being set.

To create a DBApplication object and set its attributes, you start by creating a new object, using createApplication, and then declaring the attributes you want to use for the object. You then place information into those attributes with setAttribute. You add attributes to a DBPlayer object by supplying both #userID and #application properties with setAttribute. See "Multiuser Lingo Dictionary" on page 67 for detailed examples of these.

To construct DBApplicationData objects, you put all the data you want them to hold in the form of lists of attributes and then write them to the server as an administrator-level user with createApplicationData.

This Lingo statement creates a DBApplicationData object containing values that describe a room in an online casino:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin. ¬
createApplicationData", "anySubject", [#application: "Casino", ¬
#attribute: [#roomName: "BlackJack", #dealerName: "Larry", ¬
#wallArt: "Mona Lisa", #minimumBet: 50, #music: "Classical"]])
```

Subsequent Lingo statements might create additional DBApplicationData objects with the attributes for additional rooms in the casino.

Once you have placed the objects on the server, you retrieve them for use in the movie by using getApplicationData. This command returns the list of attributes and values for the object that matches the criteria you specify for the current application. To retrieve the DBApplicationData object you previously created, you would specify the attribute #roomName and the string BlackJack as the search criteria. You supply the string in the #text property.

```
errCode = gMultiuserInstance.sendNetMessage ( "system.¬
DBApplication.getApplicationData", "anySubject", [#application: ¬
"Casino", #attribute: "#roomName", #text: "BlackJack"] )
```

In addition to the attributes you define, DBUser, DBPlayer, and DBApplication objects each have their own default attributes that are assigned by the server. There are no default attributes for DBApplicationData objects.

DBUser objects include these attributes:

#userID must be unique.

#password can be accessed with getAttribute and setAttribute, but only by users with an appropriate user level.

#lastLoginTime is changed by the server each time a user logs in and can be edited by an administrator-level user. See "Configuring the server" on page 54 for more information.

#status can be written only by administrators and can be used for whatever purposes you see fit. You might use this to keep track of whether a customer's balance is due or paid in full for a site membership subscription.

#userlevel determines privileges based on the levels you define in the server's configuration file. See "Configuring the server" on page 54. This attribute may be changed only by administrators.

#lastUpdateTime allows you to check whether other users have changed an attribute since you last checked it. This is identical to the way this is used with group attributes. See "Working with group attributes" on page 26.

`DBPlayer` objects include default attributes of `#creationTime` and `#lastUpdateTime`. The `#creationTime` attribute indicates when the object was created on the server and will usually correspond to when the user began participating in a particular movie.

`DBApplication` objects include default attributes of `#userID`, `#description`, and `#lastUpdateTime`. The `#userID` attribute defaults to the name of the movie that created the object, which you supplied in the `connectToNetServer()` command. The `#description` attribute contains whatever description you choose to supply when you create the `DBApplication` object.

A database scenario

Suppose you want to create an online casino in which users will play various games, such as poker, blackjack, and roulette. Each game could be a separate movie you create and could contain several different rooms to play in. You could use each of the types of database objects to store information about users, players, games, and the casino environment.

You could create `DBUser` objects for each new person who logs in and wants to enter the casino. The attributes you store for each user might include their e-mail address, preferred games, and so on. Your `DBPlayer` objects might include a player's current winnings for a particular game.

You could use `DBApplication` objects to store the name of the user who has won the most money in each game. You might also have an attribute in each of these `DBApplication` objects that is a list of the `DBApplicationData` objects you've created for different rooms in each game. You could then use this list to retrieve the data for a particular room in a particular game. The `DBApplication` object for the blackjack movie might contain an attribute called `#roomList` with room names in a linear list like this one:

```
["Art Deco", "Western Saloon", "Contemporary", "Egyptian"]
```

You could then use `getApplicationData` to retrieve the list of attributes for each room. The `DBApplicationData` object for one room might look like this:

```
[#roomName: "Art Deco", #dealerName: "Larry", #wallArt: "Mona
Lisa", #minimumBet: 50, #music: "Classical"]
```

By creating `DBApplicationData` objects for each room, you can design your movies to display certain graphics and sounds based on which room object is chosen by the user.

Controlling user access to the server

You can control who connects to the server by using `DBUser` objects. You can choose to allow anyone to connect to the server, or you can limit access to users who already have a `DBUser` object. To limit user access you would use an administrative movie to create new users' `DBUser` objects before allowing them to connect on their own. You can also choose to let all users connect, but restrict certain privileges to only those users with `DBUser` objects. You do this by giving users with `DBUser` objects a higher value for their `#userlevel` attribute and letting other users default to a lower value. See "Configuring the server" on page 54 for more information.

Making projectors

Because the Multiuser Xtra is a Lingo Xtra and has no cast member types associated with it, it is not added automatically to a movie's Xtra list. Before you make a projector from a multiuser movie, you must add the Multiuser Xtra to the Xtra list in the Movie Xtras dialog box.

To add the Multiuser Xtra to a movie's Xtra list:

1. Open the movie.
2. Choose Modify > Movie > Xtras.
3. Click Add.
4. Select the Multiuser Xtra from the list that appears.
5. Click OK, and then click OK again.
6. Save the movie.

If the Multiuser Xtra is not added to the movie's Xtra list, the projector produces a script error when it is run.

Optimizing multiuser movies

When designing a multiuser experience, you should ensure that your movie responds to the information coming in from other movies in a prompt manner. You will also want your movie to send its own outgoing messages as soon as they are needed. The following guidelines will help you to create movies that are as fast and responsive as possible.

Minimizing message frequency

Design your movies to send only information that is absolutely necessary. For example, if you are tracking the position of a player's sprite, design your movie to send position information only when the sprite's position changes, rather than sending it at some regular interval regardless of whether the sprite has moved or not.

For some applications, it will make sense to collect a set of very small messages and send them together as one larger message. For example, you might collect several points of a whiteboard brush stroke and send them together in a list. For chat movies, send the entire chat message after the user presses Return or Enter instead of sending every character individually.

Prioritizing receiving over sending

In most cases, you will want your movies to receive incoming messages before sending messages, since the contents of incoming messages may affect the information you want to send.

One way to accomplish this is to increase the frame rate of your movie. The Multiuser Xtra checks for incoming messages during idle periods between frames, so higher frame rates will cause the Xtra to check for messages more often. If you don't want to increase the frame rate, you can set the `idleHandlerPeriod` to 0 in Lingo. This will maximize the frequency of idle events and allow the Xtra to check for messages during frames as well as between them.

You can tell the Xtra to check for messages at any moment you wish by using the `checkNetMessages()` function. Or you can find out how many messages are waiting by using `getNumberWaitingNetMessages()`. This allows you to specify a number of messages to retrieve as a parameter of `checkNetMessages()` so any backlog of messages can be handled all at once.

Minimizing Stage updates

In general, drawing sprites on the Stage is much more time-consuming than sending, receiving, or processing messages. Therefore, you won't necessarily want to update the screen every time you receive a message. For example, if every player in a game sends their status or position, you may want to collect that information and then update the sprites on the Stage all at once.

Sending targeted messages

If only one user needs the information in a particular message, send it to the individual user rather than to a whole group.

Using setNetMessageHandler()

By assigning different subjects to the different types of messages you are sending, you can use `setNetMessageHandler()` to process incoming messages depending on the subject, the sender, or both. The Xtra's message-dispatching routines are faster than similar code written in Lingo.

To change which message callback handler is triggered by a certain type of message, use `setNetMessageHandler()` to remove the reference to the first handler by using a zero in place of the handler symbol:

```
errCode = gMultiuserInstance.setNetMessageHandler(0, script ¬
"Connection Script", "Chat Text", "Guest Speaker", 1)
```

You can then declare a new handler for the same message criteria by calling `setNetMessageHandler()` again and specifying the new handler symbol:

```
errCode = gMultiuserInstance.setNetMessageHandler¬
(#newMessageHandler, script "Connection Script", "Chat Text", ¬
"Guest Speaker", 1)
```

Extending the server with Xtras

You can extend the functionality of the server by writing server Xtras. For information about writing server Xtras, see the Multiuser Support Center on Macromedia's Web site at http://www.macromedia.com/support/director/multiuser/.

CHAPTER 2
Multiuser Server-Side Scripting

Version 3 of the Shockwave Multiuser Server includes the ability to add Lingo scripts to the server. Because they run on the server computer itself, these scripts enable you to write simpler multiuser movies. The client movies only need to include the simple logic for sending and receiving messages and reacting to their content. The Lingo running on the server can handle the tasks of tracking the state of each client movie and broadcasting information to all of the clients. Without server-side Lingo, each client movie must include more complex Lingo that handles both of these responsibilities.

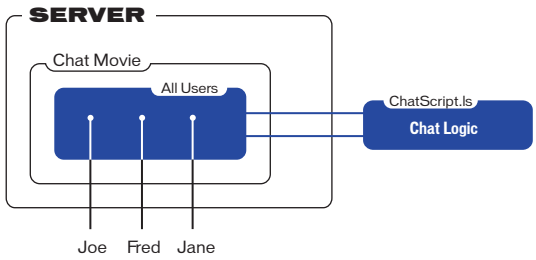
In addition to making it easier to author multiuser Lingo, server-side Lingo helps protect multiuser movies from errors. For example, because the server can run Lingo and keep track of client movie states, it is much easier to prevent client movies from getting out of sync with one another.

While server-side Lingo makes it easier to create multiuser movies, using it requires a strong fundamental knowledge of the Lingo language. See the Writing Scripts with Lingo chapter in *Using Director*.

Server-side Lingo is enabled by the LingoVM (Virtual Machine) Xtra, found in the server's Xtras folder. This Xtra contains the server's Lingo engine. When the server starts up, it loads the LingoVM Xtra, which then reads certain script files from the scripts folder.

The scripts folder is located next to the server application. It contains the two primary script files required by the server. These script files, Dispatcher.ls and Scriptmap.ls, are text files that the server reads and uses to set up the server-side scripting environment.

Each Director movie ID that you want to use server-side scripts must have a script file of its own in the server's scripts folder. In addition to the Dispatcher.ls and Scriptmap.ls files, the server will read each movie's script file. Once these files have been read by the server, it is ready to execute server-side scripts.



The Chat movie uses the Lingo inside the ChatScript.ls file.

For an example of server-side scripting, see the example Director movies in the Director 8.5/Learning/Lingo_examples/Multiuser_examples folder.

The Lingo core

In order for the server to use Lingo, it must be accompanied by the LingoVM Xtra, located in the server's Xtras folder. The LingoVM Xtra contains Lingo's core engine. The core is a subset of Lingo as a whole. It includes all the fundamental commands and functions that make up a programming language, such as terms for working with variables and lists, testing conditions, and repeating instructions. However, the Lingo core does not include commands and properties related to things that are specific to Director movies, such as sprites, cast members, cast libraries, movies, and frames. For example, the server does not know what the `stageColor` property is, because it has no Stage. Lingo that is provided through Director's Xtras is also not included in the Lingo core.

The following is a list of the Lingo elements included in the Lingo core:

Keywords

case	if/then/else	put	return
delete	next repeat	repeat	set
exit			

Commands and functions

abort	exp()	new()	rgb()
abs()	float()	nothing	runMode()
append	floatP()	numToChar()	sin()
atan()	ilk()	objectP()	sqrt()
bitAnd()	inside()	offset()	string()
bitNot()	integer()	param()	stringP()
bitOr()	integerP()	paramCount	symbol()
bitXor()	intersect()	pictureP()	symbolP()
call()	length()	point()	systemDate
callAncestor()	list()	power()	tan()
chars()	listP()	propList()	text
charToNum()	log()	random()	time
color()	map	rect()	union()
cos()	max()	result	value()
date	maxInteger	return	voidP()
do	min()		

Data types and objects

child objects	float	the picture	script objects
color	integer	point	string
compressedMedia	list	property list	symbol
date	the media	rect	void

The server's LingoVM Xtra includes many Lingo elements in addition to the Lingo core. These elements support features that are specific to the server, such as server-side scripting, multithreading, and server file access.

About server-side scripting

The server uses three files to set up its server-side scripting environment: the LingoVM Xtra, found in the server's xtras folder, and the Dispatcher.ls and Scriptmap.ls script files, found in the server's scripts folder. In order for server-side scripting to function, these files must be present in the scripts folder when the server starts up.

During startup, the server loads the LingoVM Xtra, which in turn reads the Dispatcher.ls file and converts its contents to a running server script. This Dispatcher script then loads the Scriptmap.ls file and converts it to a running server-side script as well.

The Dispatcher script contains some handlers that are required for server-side scripting and some that are useful but not mandatory. The required handlers are `on initialize`, `on configCommand`, `on serverEvent`, and `on incomingMessage`. You may choose to edit the Dispatcher.ls file to customize its functionality, but these four handlers must be present.

Loading the Scriptmap.ls file

After creating the Dispatcher script, the server calls the Dispatcher's `on initialize` handler. This handler sets up some variables and then calls the Dispatcher.ls file's `on loadScriptMapFile` handler and passes it the Scriptmap.ls file name. The `on loadScriptMapFile` handler uses file-access Lingo (see "Accessing files on the server" on page 49) to read the Scriptmap.ls file and assigns its contents to a string variable. This string variable is then converted to a script object with the `createScript()` function.

The `on loadScriptMapFile` handler then calls the Scriptmap.ls file's `on scriptMap` handler. The default `on scriptMap` handler looks like this:

```
on scriptMap
   theMap = []

   --theMap.Append( [ #movieID: "BlackJack*", #scriptFileName:\
   "BlackJack.ls" ] )
   --theMap.Append( [ #movieID: "Debug", #groupID:"@DebugGroup",\
   #scriptFileName: "Debug.ls" ] )

   return theMap
end
```

This handler returns a list of property lists, each containing a movie ID and a script file name. Each of these property lists is used by the server to associate a specific movie ID that might log on to the server with a specific script file located in the server's scripts folder. The two lines beginning with `theMap.append` are commented because they refer to example movies you may or may not want to activate on the server.

The Dispatcher script next uses file-access Lingo to read these script files and convert them to script objects with the `createScript()` function. Once each script object is created, it is ready to receive and handle events from its associated movie.

Adding server-side scripts

Adding server-side scripting functionality to a movie requires that you create a script file for your movie and edit the Scriptmap.ls file.

To add a script to the server for your movie:

1. Make a text file of the script, name the file and give it an "ls" extension. This script should contain handlers for each server event you want the script to react to, plus any custom events you define.
2. Place the file in the server's scripts folder.

3 Open the Scriptmap.ls file in a text editor and add a line to it with the following syntax:

```
theMap.append( [movieID: "yourMovieID", scriptFileName: \
"yourScriptFileName"] )
```

This line should be added immediately after the last line that contains the same syntax. The string *yourMovieID* is the movie ID your movie will use when logging into the server with `connectToNetServer()`. The string *yourScriptFileName* is the actual file name of your movie's script file that you placed into the server's scripts folder.

For example, to add the file TestScript.ls to the Scriptmap.ls file and associate it with the movie ID TestMovie, you might start with a Scriptmap.ls file like this:

```
on scriptMap
    theMap = []

    theMap.append( [ #movieID: "BlackJack*", #scriptFileName: \
    "BlackJack.ls" ] )
    theMap.append( [ #movieID: "SimpleChat", #scriptFileName: \
    "SimpleChat.ls" ] )

    return theMap
end
```

Then you would add a line to it like the one in the following:

```
on scriptMap
    theMap = []

    theMap.append( [ #movieID: "BlackJack*", #scriptFileName: \
    "BlackJack.ls" ] )
    theMap.append( [ #movieID: "SimpleChat", #scriptFileName: \
    "SimpleChat.ls" ] )
    -- This is the new line
    theMap.append( [ #movieID: "TestMovie", #scriptFileName: \
    "TestScript.ls" ] )

    return theMap
end
```

Note that the `append` commands at the beginning of each line cause the lists to be combined, resulting in a list of lists that Lingo interprets like this:

```
[ [ movieID: "BlackJack*", scriptFileName: "BlackJack.ls" ], \
[movieID: "SimpleChat", scriptFileName: "SimpleChat.ls" ], \
[movieID: "TestMovie", scriptFileName: "TestScript.ls" ] ]
```

If you want to associate a server-script with a specific group within a movie, add a `#groupID` property to the list. This will cause only messages sent from members of that group to be forwarded to the specified script.

```
theMap.append( [#movieID: "TestMovie", #groupID: "@chatUsers",\
#scriptFileName: "TestScript.ls"] )
```

4 Restart the server.

You can also make the server reload the Scriptmap.ls file without restarting by having a movie send a message to the server with a subject of `System.Script.Admin.Reload`. In order for this to work, the `case` statement in the Dispatcher.ls file's `incomingMessage` handler must be uncommented.

The following excerpt of the Dispatcher script's `on incomingMessage` handler contains the `case` statement:

```
-- These commands may be useful during development; be sure to
    -- disable them for a production server
    --
    --  case subject of
    --    "System.Script.Admin.Reload":
    --      put "LingoVM: reloading all scripts."
    --        tlist = thread().list
    --        repeat with t in tlist
    --          t.forget()
    --        end repeat
    --      the timeoutList = []
    --      me.loadScriptMap()
    --      exit
    --    "System.Script.Admin.Ping":
    --      sender.sendMessage(subject, msg)
    --      exit
    --    "System.Script.Admin.ShowState":
    --      showServerState()
    --      exit
       --  end case
```

Note that the entire `case` statement is commented so that it will not be active by default. Reloading the Scriptmap.ls file will abort any script threads in progress. See "Multithreading" on page 46.

When adding server-side scripts, you can choose to associate several movie IDs with a single script file or several script files with a single movie ID, depending on the requirements of your multiuser application. The asterisk (*) after the `#movieID` BlackJack indicates that any movie ID beginning with the string BlackJack will be associated with the BlackJack.ls file. The Dispatcher script's `on wildCompare` handler resolves movie IDs beginning with the given string and associates them with the correct script file.

Standard server events

Each script file you add should contain a handler for each type of server event (user log-ons, group creation, etc.) you want your script to react to. When standard events such as users entering or leaving groups occur, the server calls the `on serverEvent` handler in the Dispatcher.ls file. This handler determines the type of event and the movie it came from and forwards it to that movie's server-side script.

For example, if the event forwarded by the server is a user log-on, the script's `on userLogOn` handler is called. If the event is a group creation, the script's `on groupCreate` handler is called.

Each of the handlers in a movie's script must include arguments that inform the handler about where the event came from. For example, a `userLogOn` handler might look like this:

```
on userLogOn (me, movie, group, user)
   put "User Log On occurred"
end
```

The arguments are passed by the server to the handler and indicate exactly which movie, group, and user generated the event. The `me` argument is required and refers to the script object containing the handler. In the preceding example, the `put` statement displays text in the server's console window.

Distributing server events

When a movie sends an event to the server, the server forwards the event to that movie's script object using the `on distributeServerEvent` handler in the Dispatcher script. These events include the following:

- `on movieCreate`: the first user of the movie logging on to the server
- `on movieDelete`: the last user of the movie logging off of the server
- `on userLogOn`: a user logging on to the movie
- `on userLogOff`: a user logging off of the movie
- `on groupCreate`: a group being created (when the first user joins the group)
- `on groupDelete`: a group being deleted (when the last user in a group leaves)
- `on groupJoin`: a user joining a group
- `on groupLeave`: a user leaving a group
- `on incomingMessage`: a message sent to the movie's server-side script
- `on serverShutDown`: the server is shutting down
- custom events created by the Lingo author

Script files associated with movies may contain handlers for any of these events. These scripts may also contain custom handlers that can be called by specially formatted messages sent by the movie.

Sending custom events to server-side scripts

To call a handler in a movie's server-script, send a message with `"system.script"` in the `#recipients` parameter and the handler name in the `#subject` parameter. The following statement sends a message to the server that calls the handler `testHandler` in the movie's server-side script:

```
errCode = gMultiuserInstance.sendNetMessage([#recipients: \
"system.script", #subject: "testHandler", #content: \
"This is the text to be displayed"])
```

The movie's server-side script will receive this message, and its `on incomingMessage` handler will be called. The `on incomingMessage` handler should contain a `case` statement that tests the contents of the `#subject` property and calls the handler named in it.

The server-side script would look something like this:

```
on incomingMessage (me, movie, group, user, fullMsg)
   -- if the #subject of the incoming message is "testHandler",
   -- call that handler and pass it the #content as an argument
   case fullMsg.subject of
      "testHandler":
         me.testHandler(fullMsg.content)
   end case
end

on testHandler me, textArg
   -- display the #content of the message in the server's
   -- console window
   put textArg
end
```

Note that the arguments `me`, `movie`, `group`, `user`, and `fullMsg` are provided by the server and must be included with the `incomingMessage` handler. By passing these arguments to your handler, the server lets you know exactly who the message came from.

Specifying `"system.script"` as the recipient tells the server to send an `incomingMessage` event to the server-side script associated with the movie that sent the message. The `#subject` gets passed to the handler as part of the argument `fullMsg`. The `put` statement in the handler causes the server to display the specified text in its console window.

Sending messages from server-side scripts

Your server scripts can send messages to their movies by using the `sendMessage()` command. The following server script sends a message back to the movie that calls the handler `testHandler`:

```
on incomingMessage (me, movie, group, user, fullMsg)
   -- if the #subject of the incoming message is "testHandler"
   -- call that handler and pass it the fullMsg as an argument
   case fullMsg.subject of
      "testHandler":
         me.testHandler(user, fullMsg)
   end case
end

on testHandler me, user, fullMsg
   -- send a message back to the user saying that the incoming
   -- message was received
   user.sendMessage(fullMsg.senderID, "Server response", \
   "Your message was received")
end
```

By specifying `user.sendMessage` you tell the server to send the message to the `user` object passed in with the original incoming message.

About server-side script objects

The script objects created by the server with the `createScript()` function are different from most other script objects created at run time. They are not child objects but are similar to Director's script cast members. When the server reads an LS file containing a script, it gets only the string contained in the file. In order to treat that string as a working script, the server must create a script object from it with the `createScript()` function. Once the string has been converted to a script object, it is available to the LingoVM Xtra and can be thought of as a script cast member, with handlers that can be called by the server, connected movies, or other scripts.

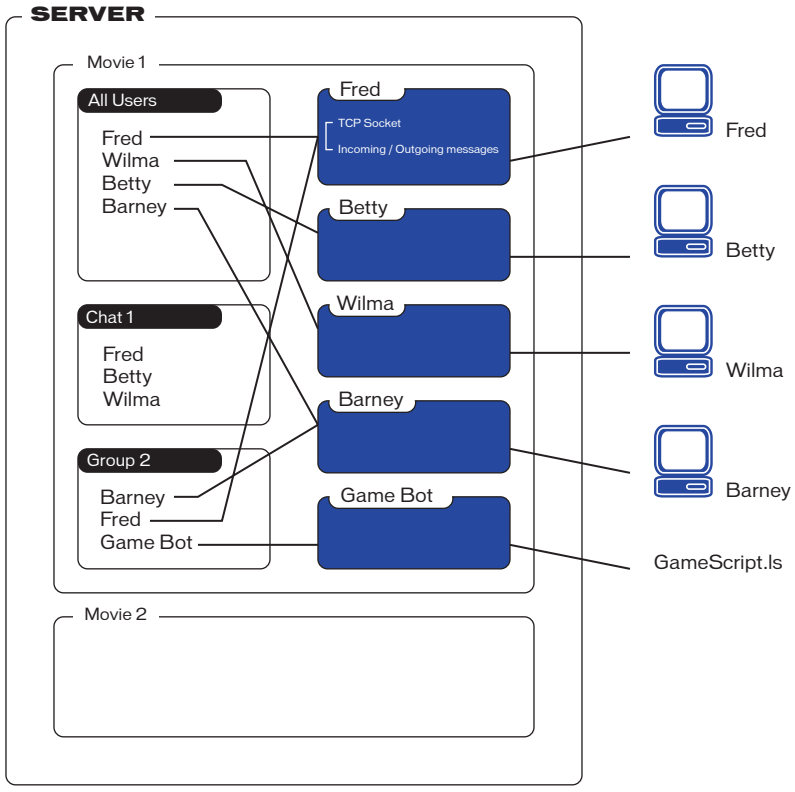
By default, these script objects are created as parent scripts. However, for them to function as true parent scripts, you must choose to write the script for a movie in such a way that it can birth child objects by reading additional script files and creating child objects from those scripts with the `new()` function.

To create a global script with the `createScript()` function, you must include a `#global` argument with the function. See "createScript()" on page 77 in the Multiuser Lingo Dictionary.

These statements create a parent script from the text of the file testScript.ls:

```
scriptText = file("testScript.ls").read()
scriptObject = createScript(scriptText, #global)
```

Whether true parent scripts or global scripts are appropriate for a particular situation depends on the movie and the author's judgment.



The GameScript file contains a script that acts like a client and manages the logic required for the game.

Multithreading

In order to enhance performance in situations where large numbers of movies are executing server-side scripts simultaneously, the server supports cooperative multithreading. Cooperative multithreading allows the scripts of many movies to execute at the same time. This prevents one movie from having to wait for all the movies ahead of it to finish their script processing before being able to process its own scripts on the server.

The type of multithreading supported by the server is known as *cooperative multithreading* because the different threads, or script processes, take turns using the computer's CPU in a cooperative manner. While one thread can issue a command to block another thread from executing for a period of time, the default behavior of the server's threads is to share the CPU equally.

Threads are particularly advantageous when your server-side scripts are executing complex or repetitive tasks or during development, when your code may produce errors. By using threads, you can prevent script errors or repetitive tasks from getting in the way of tasks running in other threads. Note that the Dispatcher script runs in the main thread that starts when the server launches.

To take advantage of multithreading, a script must first create a new thread with the `thread().new()` command. Once the new thread is created, subsequent handlers that are called may be assigned to the thread so that they execute within it.

The following script contains two handlers. The first creates a new thread, and the second is executed in the newly created thread:

```
on makeNewThread me
   theThread = thread().new("testThread")
   -- get a random number by assigning the calculateRandomNumber
   -- handler to the newly created thread
   theNumber = theThread.call(#calculateRandomNumber, me)
end

on calculateRandomNumber me
   a = random(999)
   return a
end
```

In this example, the `calculateRandomNumber` handler is assigned to the thread named `testThread`. It runs in that thread and allows other handlers assigned to other threads to run simultaneously. You can choose to have all the handlers in a movie's server script run in one or more threads.

Sharing data between threads

Once one or more threads exist, they can share information with each other. To have one thread pass a value to another single thread, use the `produceValue()` function. The following handler runs in a thread named `testThread`, assigns a value to the variable `testValue`, and makes the value available to another thread with the `produceValue()` function:

```
on sendValue
   testValue = 123456
   testThread.produceValue(testValue)
end
```

The thread `testThread` will stop running until another thread accesses the value by using `awaitValue()`. The following handler runs in its own thread and accesses the value `testValue` from the thread `testThread`:

```
on retrieveNumber
   testThread.awaitValue()
end
```

The thread containing this handler will also stop running until the value is produced by the thread named `testThread`.

The `produceValue()` and `awaitValue()` commands are useful for a single thread sharing data with a single other thread. To have a thread send a value to more than one other thread, use the `lock()`, `wait()` and `notifyAll()` commands. These commands work only on lists, so you must place the data you want to share into a linear list or a property list.

Use these steps to send data from one thread to several other threads:

1. Use the `lock()` command in the thread that will be editing the list to prevent any other thread from accessing the list while the first thread is editing it.

   ```
   myList = [12, 32, 43, 34, 45]
   lock(myList)
   ```

2. Use the `wait()` command in every other thread that you plan to have receive the new value of the list. These threads will stop executing until the new value of `myList` becomes available.

   ```
   wait(myList)
   ```

3. In the original thread that locked the list, edit the value of the list. The editing of the list must be limited to changing, adding, or deleting individual values from within the list.

   For example, the following Lingo edits only the 4th value inside the list named `myList`:

   ```
   myList[4] = 66
   ```

   The result is a list that looks like this:

   ```
   myList = [12, 32, 43, 66, 45]
   ```

   Do not set the list to a whole new list, as shown in the following Lingo:

   ```
   myList = [1, 2, 3, 4]
   ```

   This will result in the lock that was originally placed on `myList` in step 1 being removed. Other threads could then edit the list themselves while the original thread is editing it, producing unpredictable results.

4. Use the `notifyAll()` command on the list in the original thread to pass the new value of `myList` to all the threads that are waiting on it.

   ```
   notifyAll(myList)
   ```

Threads can also call handlers in other threads, report their status, and start and stop their operations on command. For more information, see “Multiuser Lingo Dictionary” on page 67.

Accessing files on the server

In order to create script objects from the text files in the scripts folder, the server makes use of file-access Lingo that is included in the LingoVM Xtra. These Lingo elements allow you to read from and work with files on the server computer. You can work with text files as well as files of other types that can contain any value that can be stored in a Lingo variable. You can read and write integers, floating-point numbers, lists, images, and more.

You perform these operations by using the file-access Lingo elements. These commands and functions allow you to get information about the size, content, and locked state of files, as well as add new files, edit the content of existing files, and delete files.

To check whether a file you want to read is present on the server computer, use the `exists()` function. The following handler checks whether the file Sunset.jpg exists in the test_files folder on the server computer.

```
on checkForFile
   return file("C:\test_files\Sunset.jpg").exists
end
```

Once you know a file exists, you can read the contents of the file into a variable with the `read()` function. The following statement reads the contents of the file Sunset.jpg and assigns them to the variable `myImage`:

```
myImage = file("C:\test_files\Sunset.jpg").read()
```

If you want to read only a part of the file, you can specify the number of bytes to be read. The following statement reads just the first 400 bytes (characters) from the file LongSpeech.txt and assigns them to the variable `theText`.

```
theText = file("C:\test_files\LongSpeech.txt").read(400)
```

To read from files other than text files, use the `readValue()` function.

To change the contents of a text file or create a new text file, use the `write()` function. To write any type of Lingo value to a file, use the `writeValue()` function.

These statements write the string “Four score and seven years ago...” to the file Gettysburg.txt:

```
theString = "Four score and seven years ago..."
file("C:\test_files\Gettysburg.txt").write(theString)
```

This statement writes the contents of the variable `tempImage`, which contains bitmap image data, to the file Newimage.tmp:

```
file("C:\test_files\NewImage.tmp").writeValue(tempImage)
```

Using `write()` and `writeValue()` will cause the entire file to be overwritten. It is best to read the entire file into a variable, manipulate the variable, and then rewrite the entire file.

You can also copy files with the `copyTo()` command. The following statement copies the contents of the file Gettysburg.txt into the file Longspeech.txt:

```
file("C:\test_files\Gettysburg.txt").copyTo \
("C:\test_files\Longspeech.txt")
```

For more information on other kinds of file manipulation with Lingo, see "Multiuser Lingo Dictionary" on page 67.

Server security

Because the server is able to run Lingo scripts and access files on the server, it is possible that a malevolent user could access and use the server in ways you don't intend unless certain precautions are taken.

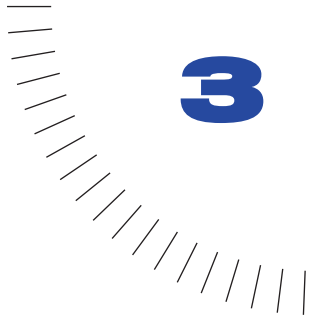
Server-side scripting is an administrator-level activity, so server scripts are always given administrator-level access to the server. Therefore any user who calls these scripts will be able to do anything that is enabled by the scripts on the server. A user who discovers the names of handlers on the server and the movie ID used by your Director movie could write their own movie to call those same handlers.

Keep in mind the following precautions:

- If you have scripts that perform administration functions on the server, consider calling those scripts from a separate Director movie that is different from the movie you will distribute to the public.
- When you are ready to deploy your multiuser movie, it is a good idea to disable or remove the adminstration functionality that appears at the beginning of the Dispatcher script's `on incomingMessage` handler. These functions can be dangerous if misused.
- Use caution when calling database administration functions from within server-side scripts. For example, if you use `createUser` within a server-side handler you are assumed to be an administrator-level user by the server.
- Use care when using file access Lingo inside server-side handlers. These commands could be used to manipulate the server computer's hard drive in destructive ways.
- If you edit the Dispatcher script, keep in mind the security implications of any changes you make.

Advanced topics

Visit the Multiuser section of Macromedia's Director Support Center at http://www.macromedia.com/support/director/multiuser/ for additional articles about creating advanced multiuser applications.



CHAPTER 3
The Server Application

By default, the Shockwave Multiuser Server is included as part of the Macromedia Director 8.5 installation and can be found in the Director folder. The installation creates a folder that contains the following items:

- The server application
- A Multiuser.cfg file, which contains parameters you can set in order to control certain aspects of the server's behavior
- An example Movie.cfg file, which can be used to set server parameters differently for specific movies
- A ReadMe file that directs you to online resources and release notes
- A folder called DBObjectFiles, which the server uses to store object databases that can be created and modified by multiuser movies at run time
- An Xtras folder containing server Xtras, separate files containing software code that extends the server's functionality
- A scripts folder containing script files that enable server-side scripting
- Three DLLs: C4dll.dll, Iml32.dll, and Msvcrt.dll (Windows only)
- Two code libraries, called MacromediaRuntimeLib and ImlLib (Macintosh only)

Running the server

Before you launch the server, you may want to increase the number of simultaneous connections allowed from the default number of 50. To increase this number to up to 1000 connections, edit the Multiuser.cfg file by removing the # character from the beginning of the `ServerSerialNumber` line and pasting your Director serial number into the end of the line. For more information, see "Using the Multiuser.cfg file" on page 54. On the Macintosh, you should increase the amount of RAM allocated to the server in its Get Info window before hosting large numbers of connections or using significant server-side scripts. See "Multiuser Server-Side Scripting" on page 35.

To launch the server:

Double-click the server's application icon.

At startup, the server will read through the Multiuser.cfg file, load the three default server Xtras, and then allocate its connections. When the connections have been allocated, the server is ready to accept connections and process messages from Director movies. Note that if the server does not recognize any of the items in its Multiuser.cfg file, it will shut down. For more information, see "Using the Multiuser.cfg file" on page 54.

Viewing server information

You can view details about the server's activity in two ways. The View menu lets you see messages in the Server window each time certain events happen, such as when users log on to or out of the server. This is useful during testing of movies or anytime you want to see a complete log of activity. The Status menu lets you see information on demand about the total number of users on the server, movies connected, databases being used, groups created, and so on.

To view server information using the View menu:

- Choose View > Server Response Time to see how much time the server is taking to process each incoming message.

  A check mark next to the menu item indicates that it is active. This command will display a message every 10 seconds that shows the number of messages processed and the average time in milliseconds used for each message. You can adjust the time interval from 10 seconds to another number by editing the Multiuser.cfg file. See "Using the Multiuser.cfg file" on page 54.

- Choose View > Users Log On and Off to see a message each time a user logs in or out of the server.

- Choose View > Movie Creation and Deletion to see a message each time the first instance of a new movie connects to the server or the last instance of a movie logs off.

  This command also displays a message when the last instance of a particular movie disconnects from the server. By using the Users Join and Leave Groups and Group Creation and Deletion menu items, you can choose to see messages each time a connected user joins or leaves a group or when new groups are created or deleted.

To view general server status information using the Status menu:

Choose Status > Server. The following information will be displayed:

Server IP address: 123.45.67.89. This is the address you specify in the Multiuser.cfg file. See "Configuring the server" on page 54.

Server port: 1626. This is the communications port number assigned to the Shockwave Multiuser Server.

Number of connections: 47 available, 3 in use. In this example, three users are connected to the server. The number of connections is based on the number permitted by the server license key.

Connections waiting to be recycled: 0. These are connections that have logged out or timed out but have not yet been reset by the server to be able to accept new connections. This number will usually be 0 except on very active servers.

Number of connections awaiting logon: 0. These are connection requests that the server has received but not yet processed.

Number of movies: 3. This is the number of different movies connected to the server. These might be Chess, Pinball, and TechChat, for example, for a total of three movies.

To view specific status information using the Status menu:

- Choose Status > Movies to display a list of movies currently connected to the server as well as the number of users, groups, and databases used by each movie.
- Choose Status > Users to see the total number of users currently connected to each movie on the server. This will also display each user's name, user level, and IP address.
- Choose Status > Groups to see a list of the groups for each movie on the server and the members in each group.
- Choose Status > Databases to see a list of the databases used by each movie.

Configuring the server

You can change many aspects of how the server operates by editing the Multiuser.cfg file. You can specify separate settings for different movies by creating additional CFG files. By editing parameters in any of these CFG files, you can do the following:

- Change the amount of memory available for handling messages.
- Limit the number of users that can connect to the server or to a particular movie.
- Specify which types of users are allowed to connect to the server.
- Specify which movies may connect to the server.
- Set the user levels required to execute each server command.

Using the Multiuser.cfg file

There are many other settings that can also be changed by editing the Multiuser.cfg file. Examples of each of these can be found in the Multiuser.cfg file that came with your server. Comments can be added to any of the configuration files by preceding each comment line with a pound sign (#). You can also add a # character to the beginning of an optional parameter line to prevent it from being read by the server. For parameters that can take more than one value, place a space and a backslash (\) after the first value and place the next value on the next line. If the server does not recognize a particular setting name or a setting is invalid, it displays an error message in its console window and does not allocate any incoming connections. For changes to the file to take effect, the server must be restarted.

When you edit the Multiuser.cfg file, keep a copy of the original file provided with the server installation.

The following server settings can be edited:

Echo allows you to add text that is displayed in the Server window when the server reads the Multiuser.cfg file at startup.

ServerOwnerName is an optional parameter that allows you to enter your name so that it appears in the server window at startup.

ServerSerialNumber allows you to enter your Director serial number. This lets you increase the number of possible server connections from the default value (50) up to 1000. Both Macintosh and Windows serial numbers work on either platform's version of the server.

ServerPort specifies the communications port used by the server. The Shockwave Multiuser Server is assigned port 1626 so that it will not conflict with other server applications running on the same computer. You may change this if you find it necessary to do so.

ServerIPAddress is an optional parameter that allows you to specify an IP address if your server computer uses more than one. Most computers use only one IP address. You can also use this parameter to make the server listen for messages on multiple port numbers on a single IP address.

MaxMessageSize sets the amount of memory to use for the incoming and outgoing message buffers. The total memory used is twice the number you specify. The value must be in bytes and should be a multiple of 1024 (1 kilobyte).

ConnectionLimit is an optional parameter that specifies the maximum number of users that can connect to the server. It defaults to the number indicated by the license you purchased. You can use this parameter to limit the connections to a smaller number.

LogonRejectionDelay is an optional parameter that lets you set the number of seconds to wait before sending a message informing a client whose log-on has not been accepted. The default is 10 seconds.

EncryptionKey is an optional parameter that lets you specify a string to use for encrypting user name and password information when users connect to the server. If you use this parameter, you must include an identical string when using the `connectToNetServer` command in a client movie.

UserLevel lets you specify required user levels for each of the database commands and server commands. You can choose any number between 0 and 100 for each command you want to control access to. By using different numbers for different commands, you can create custom privileges for users of different levels. For the default values for each command, see the original Multiuser.cfg file that came with your server.

MovieCFGPath is an optional parameter that lets you specify the directory where movie-specific configuration files are located.

LogFileName is an optional parameter that lets you specify the name and location of the text file the server generates with its status messages. This file contains all the information displayed in the server's application window and is useful for looking back at server activity and debugging movies.

AllowMovies is an optional parameter that lets you specify the names of movies that you want to allow to connect to the server. If this parameter is not used, any movie may connect.

MoviePathName is an optional parameter that lets you specify the absolute Internet addresses of movies that you want to allow to connect to the server. The server checks a movie's address against the `MoviePathName` settings only if the movie uses the property list format of `connectToNetServer()`. This prevents movies from connecting from any location other than the one the author intended. See "connectToNetServer()" on page 72.

IdleTimeOut is an optional parameter that lets you specify how many seconds to let movies stay connected to the server if they are not sending any messages. Movies that remain idle for longer than the time you specify are disconnected by the server. If omitted, this parameter defaults to 1 hour.

ScanTimeReportInterval lets you set the number of seconds between each Server Response Time message in the server's application window. This takes effect only when the Server Response Time item in the server's View menu is turned on.

You can change the default settings for which types of messages are displayed in the server's application window by editing the following parameters. These parameters change which items in the server's View menu are turned on and off at startup. A value of 1 turns the item on, and a value of 0 turns the item off.

ShowLogonMessages corresponds to the Users Log On and Off item in the View menu.

ShowCreateMovieMessages corresponds to the Movie Creation and Deletion item in the View menu.

ShowScantimeMessages corresponds to the Server Response Time item in the View menu.

ShowCreateGroupMessages corresponds to the Group Creation and Deletion item in the View menu.

ShowJoinGroupMessages corresponds to the Users Join and Leave Groups item in the View menu.

The following parameters specify which server Xtras the server loads at startup and which commands each Xtra provides:

ServerExtensionXtras lets you specify the names of Server Xtras to load at startup. These names are the internal names of the Xtras, and not necessarily their file names. The Xtras need to be located in the Xtras folder at the same directory level as the server.

XtraConfigCommands lets you specify which Xtra is referred to by the `XtraCommands` that follow it. Each series of `XtraCommand` entries should be preceded by an `XtraConfigCommands` entry.

XtraCommand lets you specify each command that a particular Xtra uses. This makes the server aware of what strings may be sent as commands for the Xtra. The default Multiuser.cfg file that comes with the server has a series of commands declared for the ObjectDB Xtra, which controls the use of database objects with the server and determines whether users without a preexisting `DBUser` object can log on to the server. For details on these log-on controls, see the Multiuser.cfg file that was installed with your server.

Using the MultiuserCommon.cfg file

If you have multiple servers running on multiple computers, you can use a MultiuserCommon.cfg file to make it easy to apply the same settings to all your servers. This file is read by each of the servers when they start up, before they read the Multiuser.cfg file. By using both of these files, you can give your servers the same values for some settings and different values for others.

To configure multiple servers using both a MultiuserCommon.cfg and a Multiuser.cfg file:

1. Install the server on each computer.
2. Create a single MultiuserCommon.cfg file with the settings you want to be common to all the servers.
3. Place the MultiuserCommon.cfg file in a location of your choice.
4. Enable file sharing for the file so each copy of the server can have access to it. See your operating system documentation for more information on this.
5. Create an alias or shortcut to the file and place a copy of it next to each copy of the server.
6. Edit the Multiuser.cfg file for each server with the individual settings you choose. If a parameter appears in both the MultiuserCommon.cfg file and a specific Multiuser.cfg file, the setting in the Multiuser.cfg file has precedence.

Using movie configuration files

Each different movie that connects to the server may also have its own configuration file. You can use the same parameters in this configuration file that you use in the server's Multiuser.cfg file to give the movie its own specific settings for those parameters. The server reads the movie's configuration file when the first instance of the movie connects to the server. Give the movie's configuration file the same name that the movie uses to connect to the server, such as Whiteboard.cfg, and place it in the location you specify in the Multiuser.cfg file. If the server does not recognize a particular setting name or a setting is invalid, users will not be able to connect with that movie.

Movie.cfg files cannot use the `serverIPAddress` or `serverPort` tags. In addition to the other parameters used in the server's Multiuser.cfg file, you can use the following parameters to control specific movie actions:

NotifyDisconnect lets you specify a group name for the server to send a message to each time a user of the movie disconnects from the server.

GroupSizeLimits lets you specify a maximum number of users to allow in each group you specify for the movie. You can specify a different limit for each group. When this parameter is used, the groups will exist on the server even when they have no members.

Using the MovieCommon.cfg file

The MovieCommon.cfg file allows you to give certain parameters the same value for all movies that are connecting to the server and to give other parameters different values for each movie. For example, you might want all movies to have a connection limit of 20, so you could include the tag `ConnectionLimit = 20` in the MovieCommon.cfg file next to the server.

About administering the server

In order to run the server successfully in a network environment, it is important that you or your network administrator understand how the server interacts with the rest of the network and what features of your network the server depends on.

If your server is installed behind a firewall and you want to allow users outside the firewall to connect, you must make sure your firewall is set to allow incoming traffic on the port number that the server uses. By default, this is port 1626. You can choose another port number and configure the server's Multiuser.cfg file to use it, but you must be sure no other application on your server computer will try to use the same port number as the server. If your network uses a proxy server, it must also be configured to allow incoming traffic on the port that the Multiuser Server is using.

To ensure that your server will always be available to users in the event of a hardware or software failure, you can implement a redundant server installation by using two servers on two computers and connecting them with a third-party load-balancing solution. These include products such as Central Dispatch™ from Resonate and Local Director™, a hardware tool from Cisco Systems.

You can set up the server to start automatically after a hardware restart by adding a server alias to the Startup Items folder in the Macintosh System Folder or a server shortcut to the Startup folder in Windows. This will cause the server to restart any time there is an unplanned hardware restart, such as those caused by power failures.

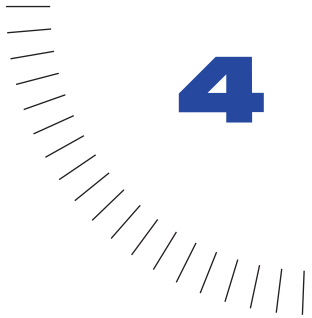
Extensibility

The Multiuser Server uses Xtras, separate files that contain software code for specific server functions. You can choose not to load certain parts of the server's functionality by removing Xtras from the server's Xtras folder. For example, you can remove the Database Xtra if you are not using databases. You can also write your own Xtras to add custom functions to the server if you are a competent C or C++ programmer. A separate Xtra Developer's Kit is available from Macromedia's Web site (http://www.macromedia.com/support) for those who wish to learn more about writing Xtras for the server.

Troubleshooting

The following are three common issues you might encounter when configuring and using the server:

- Connection attempts are blocked. This usually means that the user is behind a firewall that is not configured to allow outgoing connections on the port that Shockwave uses. Contact your network administrator to adjust the firewall settings to allow outgoing TCP/IP connections on port 1626. Client movies running behind firewalls need to be able to make outgoing connections through the firewall in order to communicate with the server. Clients movies acting as peer hosts will need the firewall to allow incoming connections.
- Large messages are not getting through the server. This usually indicates that the client's or server's message buffers have not been set large enough to handle the size of the messages you are sending. You can change this setting by editing the `MaxMessageSize` parameter in the Multiuser.cfg file. You may also need to use `setNetBufferLimits` to adjust the Xtra's buffer sizes in your movie's Lingo scripts.
- Proxy servers can interfere with the Shockwave Multiuser Server or multiuser movies that act as peer hosts, since the host uses the clients' IP addresses for routing messages. Director movies or servers running behind a proxy server may have difficulty making connections and sending messages because the proxy server may mask the real IP addresses of the movies.



CHAPTER 4
Multiuser Lingo by Feature

There are several categories of Lingo scripting elements for multiuser applications:

- Multiuser Lingo functions are provided by the Multiuser Xtra. They control the Director movie itself as it functions in a multiuser application, connecting to remote hosts, sending and receiving messages, checking error codes, monitoring the state of the movie, and so on.
- Server commands are instructions to the Shockwave Multiuser Server. Send them as messages with the `sendNetMessage()` Lingo function.
- Group commands are server commands used to manage groups of users and group attributes.
- Server database commands are server commands for interacting with database files on a remote host.
- Multithreading commands are server-side commands that allow you to create a multithreaded environment on the server so your server-side scripts run faster.
- File-access commands are server-side commands that let you open, read, and manipulate files on the server computer.
- Debugging commands are server-side commands that let you examine and isolate errors in your server-side scripts.

This chapter lists Director's various multiuser features and the corresponding Lingo elements that you can use to implement those features.

Establishing and managing server connections

These commands are used for creating and managing server connections.

checkNetMessages	sendMessage()
getNetErrorString()	sendNetMessage()
getNetMessage()	setNetMessageHandler
getNetOutgoingBytes()	getNetAddressCookie()
getNumberWaitingNetMessages()	setNetBufferLimits
connectToNetServer()	

Peer-to-peer connections

These commands are used to establish and manage peer-to-peer connections.

setNetMessageHandler	getPeerConnectionList()
waitForNetConnection()	breakConnection
connectToNetServer()	getNetAddressCookie()

Server commands

The following commands control the Shockwave Multiuser Server. Send the commands as messages with `sendNetMessage()`, using a recipient containing the syntax `System.object.command`, a subject of your choosing, and any required parameters in the message contents.

getVersion	getUserCount
getTime	delete
getMovieCount	disable
getMovies	enable

Group commands

These server commands control groups and group attributes.

join	getAttribute
leave	getAttributeNames
getGroupCount	delete
getGroups	deleteAttribute
getUserCount	disable
getUsers	enable
setAttribute	createUniqueName

Database commands

The server supports databases that are managed entirely by the server and can be manipulated with Lingo. The following commands control these databases and use the syntax "`server.databaseObjectType.commandName`". They are listed according to the types of objects they can be applied to.

DBAdmin	**DBUser**
createUser	setAttribute
deleteUser	getAttribute
createApplication	getAttributeNames
deleteApplication	deleteAttribute
createApplicationData	**DBPlayer**
deleteApplicationData	setAttribute
declareAttribute	getAttribute
deleteAttribute	getAttributeNames
DBApplication	deleteAttribute
setAttribute	**Group**
getAttribute	getAttribute
deleteAttribute	setAttribute
getAttributeNames	getAttributeNames
getApplicationData	deleteAttribute

Server-side multithreading

Use these commands to run server-side scripts in multiple threads on the server computer.

createScript()	new (thread)
count (thread)	thread()
list	sweep()
call()	sleep()
name (thread)	status
forget (thread)	awaitValue()
produceValue()	lock()
unlock()	wait()
notify()	notifyAll()
resume()	abort()

Server-side file access

Use these commands to access and manipulate files on the server computer.

readValue()	writeValue()
volumeInfo	exists
type (file)	creator
locked	read()
write()	delete() (file)
rename()	exchange()
getTempPath()	copyTo()
getAt()	createFolder()
deleteFolder()	open()
flush()	close()
position	size
folderChar	

Server-side debugging

Use these commands to debug scripts running on the server.

`setBreakPoint()`	`breakPointList`
`name (script)`	`stackSize`
`name (variable)`	`type (variable)`
`stackLevel`	`stepInto()`
`stepOver()`	`frame() (thread)`
`script (thread)`	`handler()`
`line()`	`variableCount`
`variable()`	`value (variable)`

CHAPTER 5
Multiuser Lingo Dictionary

The Multiuser Xtra provides the following commands and functions for controlling multiuser applications.

The commands and properties of the Multiuser Xtra are organized by category in the "Multiuser Lingo by Feature" chapter. In this chapter the same commands and properties are presented in standard Lingo dictionary format, with detailed descriptions, correct syntax, and scripting examples.

abort()

Syntax

```
whichThread.abort()
```

Description

Multiuser Server server-side command; stops the specified thread completely. All handler calls, including nested calls, are cleared from the thread's execution stack.

addUser()

Syntax

whichGroup.addUser(*whichUser*)

Description

Multiuser Server server-side command; adds the given user to the specified group.

Example

This handler adds the user who has just logged on to the server to the group @TestUsers:

```
on userLogOn  (me, movie, group, user)
   voGroup = movie.serverGroup("@TestUsers")
   voGroup.addUser(user)
end
```

See also

removeUser()

appendRecord

This command is obsolete. Use createUser, createApplication, or createApplicationData instead.

awaitValue()

Syntax

whichThread.awaitValue()

Description

Multiuser Server server-side function; returns a value in a variable from the specified thread. The current thread stops executing until the specified thread produces a value with produceValue().

This function should be used for a single thread awaiting a value for a single other thread. To send values to multiple threads, use lock(), wait(), notifyAll(), and unlock().

Example

The following handler gets a value from the thread testThread. The thread that contains the handler is blocked from running further until the value is produced by testThread.

```
on retrieveNumber
   testThread.awaitValue()
end
```

See also

lock(), wait(), notifyAll(), unlock(), produceValue(), sleep()

breakConnection

Syntax

```
gMultiuserInstance.breakConnection(userIDString)
```

Description

Multiuser Server Lingo command; breaks a peer connection accepted with `waitForNetconnection()`.

Example

This statement has the host movie break a connection with the user logged on as Spencer:

```
errCode = gMultiuserInstance.breakConnection("Spencer")
```

See also

`getNetErrorString()`, `waitForNetConnection()`

breakPointList

Syntax

```
whichScript.breakPointList()
```

Description

Multiuser Server server-side debugging function; returns a list of the line numbers in the script that have breakpoints set on them.

call()

Syntax

```
whichThread.call(#whichHandler, targetObject, arg1, arg2 ...)
```

Description

Multiuser Server server-side command; calls a handler in the *targetObject* parameter in *whichThread*. The current thread continues execution while the handler *whichHandler* is called in the script object *targetObject*.

Threads may also pass values between themselves using shared objects and the `lock()` and `unlock()` commands to ensure data integrity within the shared objects.

Arguments required by the handler being called should be passed after the *targetObject* parameter.

When used in threads, the `call()` command should only be used to call the specified handler in a single target object at a time. Using lists of target objects, or using `call()` more than once in the same thread before the first one has completed can produce unpredictable results. It is recommended that separate simultaneous calls to handlers be done in separate threads. This is different from the `call()` command in the Director authoring and playback environments, where it can take a list of target objects.

Example

The following server-side statements assign the integer 55 to the variable `startValue` and then pass that value to the `on MultiplyValue` handler and run the handler in the thread `testThread`. The `me` object indicates that the handler is located in the same script object that contains these statements:

```
startValue = 55
testThread.call(#multiplyValue, me, startValue)
```

See also

`produceValue()`, `awaitValue()`, `lock()`, `unlock()`

checkNetMessages

Syntax

```
gMultiuserInstance.checkNetMessages({numberOfMessages})
```

Description

Multiuser Server Lingo command; forces a check for any incoming messages and calls the callback handlers specified by `setNetMessageHandler()`. It is necessary only when the movie itself does not allow the normal idle time in which the Xtra would check for incoming messages automatically. The optional parameter defaults to 1 and specifies the maximum number of messages to process. Setting this number to a high value can cause performance problems, since this function does not return until the waiting messages have been processed.

This function is useful in movies that sit in one frame with a `go to the frame` loop, for example. It should not be called from a message handler specified by the `setNetMessageHandler()` command, or by any handler called by a message handler (including `beginSprite`, `enterFrame`, and so on) that can be executed as the result of a `go to frame n` statement.

Example

This statement tells the Multiuser Xtra to check for incoming messages, with a maximum of four messages:

```
errCode = gMultiuserInstance.checkNetMessages(4)
```

See also

`setNetMessageHandler`

close()

Syntax

```
file("fileName").close()
```

Description

Multiuser Server server-side function; closes the specified file on the server. The file is closed automatically when the file reference object is deleted.

Example

This statement closes the file Longspeech.txt on the server:

```
file("C:\Longspeech.txt").close()
```

See also

`read()`, `rename()`, `write()`, `writeValue()`

connectToNetServer()

Syntax

```
gMultiuserInstance.connectToNetServer(userNameString, \
passwordString, serverIDString, portNumber, movieIDString \
{, #mode} {, encryptionKey})

gMultiuserInstance.connectToNetServer(serverIDString, portNumber, \
[#userID: userNameString, #password: passwordString, #movieID: \
movieIDString] {, #mode} {, encryptionKey})

gMultiuserInstance.connectToNetServer([#logonInfo: [#userID: \
userNameString, #password: passwordString, #movieID: \
movieIDString], #remoteAddress: serverIDString {, #remoteTCPPort: \
serverPortNumber} {, #localAddress: clientIPAddress} \
{, #encryptionKey: encryptionKeyString}])
```

Description

Multiuser Server Lingo command; starts a connection to the server and returns an error code indicating whether the connection was initiated.

This command has three formats, shown above. The second format requires version 2.1 or later of the Shockwave Multiuser Server and makes use of additional security features. When this format is used, the Multiuser Xtra passes the Internet location of the movie to the server in addition to the standard login information. This location takes the form http://www.company.com/directory/movieName.dcr. If the server's Multiuser.cfg file has been configured to verify the location of the movie with the `moviePathName` tag, this feature can be used to prevent movies from connecting to the server from locations other than where the author intended.

userNameString and **passwordString** provide account information for logging on to the host system. User names must not contain any control characters or the characters @ or #.

serverIDString represents the Internet address of the server, such as gameServer.macromedia.com or 123.45.67.1.

portNumber is an integer specifying the connection port to be used. (By default, the Shockwave Multiuser Server uses port 1626.) This must be the same port specified in the server configuration file.

movieIDString represents the multiuser application name for this connection, such as GameChat or PingPong.

mode is an optional parameter. With the first format of `connectToNetServer()`, this parameter must be an integer; the default value is 0. If set to 1, it opens the connection as a text-based connection for communicating with IRC and SMTP servers. With the second format, this parameter must be a symbol. The default is `#smus`, which indicates a normal Shockwave Multiuser Server connection. If set to `#text`, the connection opens as a text-based connection.

A movie running in a browser or in Shockmachine that attempts make a text server connection invokes a security dialog box unless the text server resides in the same domain name as the Web server the movie is served from. Projectors running with `the safePlayer` set to `TRUE` also display a security dialog box when making text connections, unless the movie being played is a remote DCR file on the same server as the text server. If this command is used to connect to POP, SMTP, or IRC chat servers, none of the Multiuser Server command functionality is available. The command supports only simple text messaging.

encryptionKey is another optional parameter. This string is used to encrypt log-on information sent to the server and must match the encryption key specified in the Multiuser.cfg file for the server.

The third format of `connectToNetServer()` takes a property list and allows you to specify a particular local IP address to be used for messaging on clients that have multiple IP addresses. This third format also compares the client movie's location with the address string in the `moviePathName` tag in the server's Multiuser.cfg file.

The #logonInfo property list contains the user name, password and movie ID.

#remoteAddress indicates the address of the Multiuser Server you are connecting to.

#remoteTCPPort indicates the server port number to use for TCP messages. The default is 1626.

#localAddress indicates the IP address of the client machine. You can use the `getNetAddressCookie()` function to determine the IP address and to select a specific IP address if there is more than one address on the client machine.

Note that the third format of `connectToNetServer()` should be used only for connecting to the Shockwave Multiuser Server and does not contain the `#mode` parameter.

Use this command to connect to multiuser servers. Finding and connecting to the server can take some time. This function first returns an error message immediately from the Xtra. If the function's parameters are valid, the returned error code is 0. The server sends a subsequent message and calls the message handler when the connection is actually made.

Before using the `connectToNetServer` command, you should first set up a handler to receive the message that `connectToNetServer` returns. To set up the connection, create an instance of the Xtra, define the message handler using a `setNetMessageHandler` statement, and then connect with `connectToNetServer()`.

The message returned from the `getNetMessage` function is a property list with the following information:

#errorCode	Resulting error code: 0 if there is no error
#senderID	System
#subject	ConnectToNetServer
#content	Empty string
#timeStamp	Time in milliseconds on the server

You disconnect from the server by setting the variable containing the Multiuser Xtra instance to 0.

Example

This statement is a typical call that initiates a connection that is not encrypted:

```
errCode = gMultiuserInstance.connectToNetServer( "Fred", \
"secret", "serverName.company.com", 1626, "ExcitingGame")
```

See also

`waitForNetConnection()`, `setNetMessageHandler`, `getNetAddressCookie()`

copyTo()

Syntax

`file("fileName").copyTo("destinationFileName")`

Description

Multiuser Server server-side function; copies the specified file to a new file with the name *destinationFileName*. This function returns a nonzero error code if it fails.

Example

This statement copies the file Longspeech.txt to the new file Shortspeech.txt:

```
file("C:\Longspeech.txt").copyTo(C:\Shortspeech.txt")
```

See also

`exchange()`

count (thread)

Syntax

```
thread().count
```

Description

Multiuser Server server-side function; returns the total number of threads running on the server.

Example

These server-side statements count the number of threads currently in memory and then output their names and status to the server console:

```
threadCount = thread().count
repeat with i = 1 to threadCount
   t = thread(i)
   put "Thread " & t.name & " status = " & t.status
end repeat
```

createApplication

Syntax

```
system.DBAdmin.createApplication [#application:
"ApplicationName", #description: "DescriptionString"]
```

Description

Multiuser Server database command; adds a new DBApplication database object to the server. Application names may contain up to 100 characters. The description string may contain up to 255 characters.

Example

This statement creates a new DBApplication object for the movie called Checkers with a description of Two-player Checkers game:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin.
createApplication", "anySubject", [#application: "Checkers",
#description: "Two-player Checkers game"])
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID:
"system.DBAdmin.createApplication", #subject: "anySubject",
#content: [#application: "Checkers", #description: "Two-player
Checkers game"], #timeStamp: 186034087]
```

See also

connectToNetServer(), sendNetMessage()

createApplicationData

Syntax

```
system.DBAdmin.createApplicationData [#application:
"ApplicationName", #attribute: [#Attribute1: value1 {,
#Attribute2: value2, #Attribute3: value3, #Attribute4: value4}]
```

Description

Multiuser Server command; adds a new DBApplicationData object to the server. Once created, DBApplicationData objects contain read-only data associated with a particular multiuser application.

Example

This statement creates a new DBApplicationData object for the movie called Poker with the attributes `#dealerName`, `#tableColor`, and `#wallArt`:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin.
createApplicationData", "anySubject", [#application: "Poker",
#attribute: [#dealerName: "Larry", #tableColor: color(#rgb, 155,
0, 75), #wallArt: member(3).media]])
```

It is important that at least one of the attributes contain a string or an integer so that the object can be identified with `getApplicationData` or `deleteApplicationData`.

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID:
"system.DBAdmin.createApplicationData", #subject: "anySubject",
#content: [#application: "Poker"], #timeStamp: 189123520]
```

See also

`sendNetMessage()`, `getApplicationData`, `deleteApplicationData`

createFolder()

Syntax

```
file("folderName").createFolder()
```

Description

Server-side function; creates a folder on the server volume with the name *folderName*. The function returns a nonzero error code if it fails.

Example

This statement creates a folder called Tempfolder on the server volume:

```
file("C:\Multiuser_Server\Tempfolder").createFolder()
```

createScript()

Syntax

```
createScript(whichString {, #whichType })
```

Description

Multiuser Server server-side function; creates a script object compiled from the specified string. The string must contain Lingo code in order to become a script object.

#WhichType is an optional symbol indicating the type of script to create. Specify #parent to create a parent script. Specify #global to create a global script. Handlers in global scripts are available to all Lingo running on the server. The default type is #parent.

Example

These statements read the file Testscript.ls and create a script object on the server from the string contained in the file:

```
scriptText = file("Testscript.ls").read()
scriptObject = createScript(scriptText)
```

See also

read()

createUniqueName

Syntax

```
system.group.createUniqueName
```

Description

Multiuser Server command; obtains the name of an unused group from the server. This group name is unique and not previously used in the movie.

Example

The following statement has the server respond with a message that has the same subject. An error code of 0 means that the operation was successful. The contents contain a string that can be used as a group name.

```
errCode = gMultiuserInstance.sendNetMessage("system.group.\
createUniqueName", "anySubject")
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.group.createUniqueName", #subject: "anySubject", \
#content: "@RndGroup0", #timeStamp: 34653020]
```

See also

sendNetMessage()

createUser

Syntax

```
system.DBAdmin.createUser [#userID: userName, #password: \
passwordString, #userlevel: integer]
```

Description

Multiuser Server command; adds a new DBUser database object to the server. The user ID and password must be limited to 40 characters each and may not contain # or @ symbols.

Example

This statement creates a new DBUser object for the user Bob with the password MySecret and a user level of 40:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin.\
createUser", "anySubject", [#userID: "Bob", #password: \
"MySecret", #userlevel: 40])
```

The `#userlevel` attribute is optional. If omitted, it defaults to the level specified in the server's Multiuser.cfg file.

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.DBAdmin.createUser", #subject: "anySubject", #content: \
[#userID: "Bob"], #timeStamp: 180885670]
```

creator

Syntax

```
file("fileName").creator
```

Description

Multiuser Server server-side file property; on the Macintosh, gets or sets the creator code of the file. The creator code is a 4-byte (character) string.

Example

This server-side statement gets the creator code of the Director file Testmovie.dir on the server and displays it in the server's console window:

```
put file("Hard Drive:Multiuser_Server:Testmovie.dir").creator
-- "MD01"
```

See also

type (file)

declareAttribute

Syntax

```
system.DBAdmin.declareAttribute [#attribute: #attributeName]
```

Description

Multiuser Server command; declares a new attribute name that can be used by any database object. Attributes may not be set until they have been declared. Attribute names must always be symbols. Group attributes do not need to be declared with `declareAttribute`.

Example

This statement declares a new attribute called `#email`:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin. \
declareAttribute", "anySubject", [#attribute: #email])
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.DBAdmin.declareAttribute", #subject: "anySubject", \
#content: [#attribute: #email], #timeStamp: 184545570]
```

See also

`sendNetMessage()`, `setAttribute`

delete

Syntax

```
system.movie.delete ["movieName"]
system.movie.delete ["movieName1", "movieName2", "movieName3"]
system.group.delete ["groupName"]
system.group.delete ["groupName1", "groupName2", "groupName3"]
system.user.delete ["userID"]
system.user.delete ["userID1", "userID2", "userID3"]
```

Description

Multiuser Server command; when `movie` is specified, immediately deletes all instances of the given movie or movies from the server. New connections to that movie are still possible if the `disable` command has not been called.

If `group` is specified, the given group(s) is deleted from the server. This does not delete the users who occupied the group.

If `user` is specified, the given user(s) is immediately disconnected from the server. To disconnect several users, send a list of users.

The server responds with a message that has the same object and command in the `#sender` parameter and the same `#subject` and `#contents`.

Note: There is also a `delete` command in the general *Lingo Dictionary* that is applied to chunk expressions. It should not be confused with the Multiuser Server `delete` command, which is always used in the context of a `sendNetMessage()` command.

Examples

This statement deletes all instances of the movie TankWars on the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.\
delete", "anySubject" "TankWars")
```

To delete more than one movie, use a list containing the movie names:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.\
delete", "anySubject" ["TankWars", "TicTacToe", "Checkers"])
```

This statement deletes the group @RedTeam from the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.\
delete", "anySubject" "@RedTeam")
```

This statement disconnects the user BillyJoe from the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.user.delete", \
"anySubject", "BillyJoe")
```

See also

`sendNetMessage()`, `disable`

delete() (file)

Syntax

```
file("whichFile").delete()
```

Description

Multiuser Server server-side function; deletes the specified file. The function returns a nonzero error code if it fails.

Example

This server-side statement deletes the file Sunset.jpg from the server computer:

```
file("C:\Images\sunset.jpg").delete()
```

See also

`deleteFolder()`

deleteApplication

Syntax

```
system.DBAdmin.deleteApplication[#application: applicationName]
```

Description

Multiuser Server command; deletes one or more DBApplication objects from the server. Also deletes all attributes of the DBApplication object and all DBApplicationData and DBPlayer objects associated with it.

Example

This statement deletes the DBApplication object for the movie Poker from the server.

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin. \
deleteApplication", "anySubject", [#application: "Poker"])
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.DBAdmin.deleteApplication", #subject: "anySubject", \
#content: [#application: "Poker"], #timeStamp: 186056753]
```

See also

`sendNetMessage()`, `createApplication`

deleteApplicationData

Syntax

```
system.DBAdmin.deleteApplicationData [#application: \
"applicationName", #attribute: #attributeName, #text: "String"]
```

Description

Multiuser Server command; deletes one or more DBApplicationData objects from the server.

Example

This statement deletes the DBApplicationData object for the movie Poker with an attribute called `#dealerName` containing the string Larry:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin. \
deleteApplicationData", "anySubject", [#application: "Poker", \
#attribute: #dealerName, #text: "Larry"])
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.DBAdmin.deleteApplicationData", #subject: "anySubject", \
#content: [#application: "Poker", #attribute: #dealerName, #text: \
"Larry"], #timeStamp: 193099437]
```

See also

`sendNetMessage()`, `createApplicationData`

deleteAttribute

Syntax

```
system.group.deleteAttribute [#group: "@groupName", #attribute: \
#attributeName]
system.group.deleteAttribute [#group: ["@groupName1", \
"@groupName2", "@groupName3"], #attribute: [#attributeName1, \
#attributeName2]]
system.DBUser.deleteAttribute [#userID: "userName", #attribute: \
#attributeName]
system.DBUser.deleteAttribute [#userID: ["userName1", \
"userName2"], #attribute: [#attributeName1, #attributeName2, \
#attributeName3]]
system.DBPlayer.deleteAttribute [#userID: "userName", #application: \
"appName", #attribute: #attributeName]
system.DBApplication.deleteAttribute [#application: "appName", \
#attribute: #attributeName]
```

Description

Multiuser Server command; deletes an attribute with the given name from the given group or database object. Either a `#userID` or an `#application` parameter may be supplied. If both are supplied, the attribute is deleted from the DBPlayer object.

Example

This statement deletes the attribute `#accountBalance` from the DBPlayer object for the user Bob in the movie Poker:

```
errCode = gMultiuserInstance.sendNetMessage ( "system.DBPlayer.\
deleteAttribute", "anySubject", [#userID: "Bob", #application: \
"Poker", #attribute: #accountBalance] )
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.DBPlayer.deleteAttribute", #subject: "anySubject", \
#content: [:], #timeStamp: 7430457]
```

See also

`sendNetMessage()`

deleteFolder()

Syntax

`file("whichFolder").deleteFolder()`

Description

Multiuser Server server-side function; deletes the specified folder. The folder must be empty before being deleted. Use the `delete() (file)` function to remove the contents of a folder. The function returns a nonzero error code if it fails.

Example

This server-side statement deletes the Images folder from the server computer:

```
file("C:\Images").deleteFolder()
```

See also

`delete() (file)`

deleteMovie

This command is obsolete. Use `delete` instead.

deleteRecord

This command is obsolete. Use `deleteUser`, `deleteApplication`, or `deleteApplicationData` instead.

deleteUser

Syntax

```
system.DBAdmin.deleteUser [#userID: userName]
```

Description

Multiuser Server command; deletes a DBUser database object from the server.

Example

This statement deletes the DBUser object for the user Bob from the server:

```
errCode = gMultiuserInstance.sendNetMessage ( "system.DBAdmin.\
deleteUser", "anySubject", [#userID: "Bob"] )
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.DBAdmin.deleteUser", #subject: "anySubject", #content: \
[#userID: "Bob"], #timeStamp: 183543403]
```

See also

sendNetMessage(), createUser

disable

Syntax

```
system.movie.disable ["movieName1" {,"movieName2", "movieName3"}]
system.group.disable ["groupName1" {,"groupName2", "groupName3"}]
```

Description

Multiuser Server command; when `movie` is specified, disables the given movie from making any future connections. Current connections are not broken. When `group` is specified, the given group is disabled from accepting new members. Current members are not removed from the group.

If the server is not set up with the `allowMovie` command in the configuration file, by default it allows connections from all movies. The `disable` command blocks log-ons from specific movies, while the `enable` command reenables a movie disabled by `disable`. Never disable all movies; this prevents even an administrator movie from logging in to reenable other movies.

The server responds with a message that has the same object and command in the `#sender` parameter and the same `#subject` and `#contents`.

Examples

This statement disables all future connections by the movie TankWars on the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.\
disable", "anySubject" "TankWars")
```

This statement disables more than one movie by using a list containing the movie names:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.\
disable", "anySubject" ["TankWars", "TicTacToe", "Checkers"])
```

This statement disables the group @RedTeam from accepting new members:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.\
disable", "anySubject" "@RedTeam")
```

See also

`sendNetMessage()`, `delete`, `enable`

disableMovie

This command is obsolete. Use `disable` instead.

disconnectUser

This command is obsolete. Use `delete` instead.

enable

Syntax

```
system.movie.enable ["movieName1" {, "movieName2", "movieName3"}]
system.group.enable ["groupName1" {, "groupName2", "groupName3"}]
```

Description

Multiuser Server command; when `movie` is specified, enables the given movie or movies on the server so future connections can be made. Current connections are not affected. When `group` is specified, the given group or groups are enabled to accept new members.

The server responds with a message that has the same object and command in the `#sender` parameter and the same `#subject` and `#contents`.

If the server is not set up with the `allowMovie` command in the configuration file, by default it allows connections from all movies. The `disable` command blocks log-ons from specific movies, while the `enable` command reenables a movie disabled by `disable`. Never disable all movies; this would prevent even an administrator movie from logging on to reenable other movies.

Examples

This statement enables the movie TankWars on the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.\
enable", "anySubject", "TankWars")
```

To enable more than one movie, use a list containing the movie names:

```
errCode =
gMultiuserInstance.sendNetMessage("system.movie.enable", \
"anySubject", ["TankWars", "TicTacToe", "Checkers"])
```

This statement enables the group @RedTeam to allow new members to join:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.\
enable", "anySubject", "@RedTeam")
```

See also

`sendNetMessage()`, `disable`

enableMovie

This command is obsolete. Use `enable` instead.

exchange()

Syntax

```
file("whichFile").exchange("whichFile")
```

Description

Multiuser Server server-side function; exchanges the file name and other information (type, creator, date, and so on) of two files. Returns a nonzero error code if it fails.

Example

This server-side statement exchanges the header information between the files Sunset.jpg and Sunrise.jpg on the server:

```
file("C:\Images\Sunset.jpg").exchange("C:\Images\Sunrise.jpg")
```

See also

`copyTo()`

exists

Syntax

```
file("whichFile").exists
```

Description

Multiuser Server server-side function; returns `1 (TRUE)` if the specified file exists or `0 (FALSE)` if the file does not exist on the server computer.

Example

These server-side statements test whether the file Moon.bmp exists in the Images folder on the server computer and displays the result in the server's console window:

```
if file("C:\Images\Moon.bmp").exists then
   put "Moon.bmp is present"
else
   put "Moon.bmp is not present"
end if
```

flush()

Syntax

```
file("whichFile").flush()
```

Description

Multiuser Server server-side function; writes any buffered information to the file. Information becomes buffered when writing small amounts of data with the `write()` or `writeValue()` commands.This occurs automatically when the file is closed. The function returns a nonzero error code if it fails.

Example

This server-side statement writes information from the server's buffer to the file Testfile.txt:

```
file("C:\Multiuser_Server\Testfile.txt).flush()
```

See also

`write()`, `writeValue()`

folderChar

Syntax

```
file().folderChar
```

Description

Multiuser Server server-side function; returns the character used to separate folders and files in path name strings. On the Macintosh it's a colon (:); in Windows it's a backslash (\).

Example

This server-side statement assigns the `folderChar` to the variable `theChar`:

```
theChar = file().folderChar
```

forget (thread)

Syntax

whichThread.forget()

Description

Multiuser Server server-side function; removes the given thread from the list of running threads. The thread may still be held in a variable, but it is no longer run.

Examples

These server-side statements assign the thread named `testThread` to the variable `gThread` and stop it from executing:

```
global gThread
gThread = thread("testThread")
gThread.forget()
```

This server-side handler forgets all the threads in the thread list:

```
on forgetAllThreads me
   threadList = thread().list
   repeat with t in threadList
      t.forget()
   end repeat
end
```

See also

lock(), unlock(), new (thread)

frame() (thread)

Syntax

whichThread.frame(*frameNumber*)

Description

Multiuser Server server-side debugging function; returns a stack frame reference. Each nested handler call (a handler called from within another handler) produces a stack frame. Frames are numbered from 1 to `n`, with 1 being the current stack frame, 2 being the frame corresponding to the handler that called the current handler, and so on.

The `frame()` function is useful for creating references to variables and line numbers.

Example

To refer to the handler two levels above the current handler (the current handler is frame 1) use a *frameNumber* of 3.

```
frameRef = testThread.frame(3)
```

See also

frameCount (thread), line(), variable()

frameCount (thread)

Syntax

```
whichThread.frameCount
```

Description

Multiuser Server server-side debugging function; returns the number of execution stack frames in the given thread. Each nested handler call (a handler called from within another handler) produces a stack frame.

See also

```
frame() (thread)
```

getAddress

Syntax

```
system.user.getAddress ["userID"]
```

Description

Multiuser Server command; when sent to the server, returns a specific user's IP address.

Example

This statement obtains the IP address for the user logged in as Jane:

```
errCode = gMultiuserInstance.sendNetMessage("system.user.\
getAddress", "anySubject", "Jane")
```

The server responds with a message that has the same subject. The content of the message is a property list containing the requested information.

Note: The IP address may not be correct if the user is behind a firewall.

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.user.getAddress", #subject: "anySubject", #content: \
[#userID: "Jane", #ipAddress: "123.45.67.1"], #timeStamp: 763283481]
```

See also

```
getNetAddressCookie()
```

getApplicationData

Syntax

```
system.DBApplication.getApplicationData [#application: "appName", \
#attribute: #attributeName, #text: "searchString"]
system.DBApplication.getApplicationData [#application: "appName", \
#attribute: #attributeName, #number: integer]
system.DBApplication.getApplicationData [#application: "appName", \
#attribute: #attributeName, #lowNum: integer, #highNum: integer]
```

Description

Multiuser Server command; obtains the list of attributes and values from all DBApplicationData objects that correspond to the given application and contain the given attribute with the given value. The given value may be a string, an integer, or a range of integers. The result is a list of lists, each of which is the list of attributes and values for a single DBApplicationData object.

If the `#application` parameter is omitted, it defaults to the movie ID of the current movie used to connect to the server.

Up to 100 DBApplicationData objects may be returned per request.

Example

This statement returns the lists of attributes from the DBApplicationData objects for the movie Poker that contain the attribute `#dealerName` with a value of Larry:

```
errCode = gMultiuserInstance.sendNetMessage ("system.\
DBApplication.getApplicationData", "anySubject", [#application: \
"Poker", #attribute: #dealerName, #text: "Larry"])
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.DBApplication.getApplicationData", #subject: \
"anySubject", #content: [[#dealerName: "Larry", #tableColor: \
color(#rgb, 155, 0, 75), #wallArt: (media 7afa4d0)]], \
#timeStamp: 189027987]
```

See also

`createApplicationData`, `deleteApplicationData`, `sendNetMessage()`

getAt()

Syntax

```
file("whichFolder").getAt(index)
```

Description

Multiuser Server server-side function; returns the name, folder, and visible attributes of the file at the specified index in the specified folder. These attributes are returned as a property list of the format `[#name: "the folder or filename", #folder: TrueOrFalse, #visible: TrueOrFalse]`.

The `#name` property is the name of the file as a string. The `#folder` property is `TRUE` if the file is a folder, `FALSE` if it is a file. The `#visible` property is `TRUE` if the file is visible, `FALSE` if it is invisible.

The function's *index* parameter is the relative position of the file in the folder: the first file is index 1, the second file is index 2, and so on. The function returns `void` if there is no file at the specified index.

Example

This server-side statement returns the property list for the third file in the Images folder on the server computer and displays it in the server's console window:

```
put string( file.("C:\Images").getAt(3) )
-- "[#name: "Sunset.jpg", #folder: 0, #visible: 1]"
```

getAttribute

Syntax

```
system.group.getAttribute [#group: "@groupName", #attribute: \
[#attributeName1, #attributeName2]]
system.DBPlayer.getAttribute [#userID: "userName", #application: \
"appName", #attribute: [#attributeName1, #attributeName2]]
system.DBUser.getAttribute [#userID: "userName", #attribute: \
[#attributeName1, #attributeName2]]
system.DBApplication.getAttribute [#application: "appName", \
#attribute: [#attributeName1, #attributeName2]]
```

Description

Multiuser Server command; obtains from the server the values of the given attributes for the given group or object. Attributes may contain any Lingo value. You must declare an attribute before it can be used. See `declareAttribute`.

Example

This statement gets the values of the attributes `#accountBalance` and `#cardHand` for the user Bob in the movie Poker:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBPlayer.\
getAttribute", "anySubject", [#userID: "Bob", #application: \
"Poker", #attribute: [#accountBalance, #cardHand]])
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.DBPlayer.getAttribute", #subject: "anySubject", #content: \
["Bob": [#accountBalance: 3500, #cardHand: "Royal Flush", \
#lastUpdateTime: "2001/08/26 12:43:33.070364"]], #timeStamp: 7025771]
```

See also

`setAttribute`, `sendNetMessage()`, `declareAttribute`

getAttributeNames

Syntax

```
system.group.getAttributeNames [#group: "@groupName"]
system.DBUser.getAttributeNames [#userID: "userName"]
system.DBApplication.getAttributeNames [#application: "appName"]
system.DBPlayer.getAttributeNames [#userID: "userName", \
#application: "appName"]
```

Description

Multiuser Server command; gets the list of attribute names that have been set for the given group or database object. If the `#userID` parameter is supplied, the attribute list is returned for the user's DBUser object. If the `#application` parameter is supplied, the attribute list is returned for the movie's DBApplication object. If both are supplied, the attribute list is returned for the user's DBPlayer object for the given movie.

Example

This statement gets the list of attributes that have been set for the DBPlayer object of the user Bob in the movie Poker:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBPlayer.\
getAttributeNames", "anySubject", [#userID: "Bob", #application: \
"Poker"])
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.DBPlayer.getAttributeNames", #subject: "anySubject", \
#content: ["Bob": [#accountBalance, #cardHand, #lastUpdateTime]], \
#timeStamp: 7326833]
```

See also

`declareAttribute`, `setAttribute`, `sendNetMessage()`

getFields

This command is obsolete. Use `getAttribute` instead.

getGroupCount

Syntax

```
system.movie.getGroupCount ["movieName"]
system.user.getGroupCount ["userName"]
```

Description

Multiuser Server command; when `movie` is specified, returns the number of groups that exist in the specified movie. If no movie is specified, the result is for the current movie.

When `user` is specified, returns the number of groups the given user is a member of. If no user is specified, the number returned is for the current user.

Examples

This statement gets the number of groups in the current movie:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.\
getGroupCount", "anySubject")
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.movie.getGroupCount", #subject: "anySubject", #content: \
["currentMovieName": 2], #timeStamp: 763283481]
```

This statement gets the number of groups the current user is a member of:

```
errCode = gMultiuserInstance.sendNetMessage("system.user.\
getGroupCount", "anySubject")
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.user.getGroupCount", #subject: "anySubject", #content: \
["userName": 2], #timeStamp: 763283987]
```

This statement gets the number of groups for users Bob and Mary:

```
errCode = gMultiuserInstance.sendNetMessage("system.user.\
getGroupCount", "anySubject", ["Bob", "Mary"])
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.user.getGroupCount", #subject: "anySubject", #content: \
["Bob": 3, "Mary": 4], #timeStamp: 763284273]
```

getGroupList

This command is obsolete. Use `getGroups` instead.

getGroupMembers

This command is obsolete. Use `getUsers` instead.

getGroups

Syntax

```
system.movie.getGroups
system.user.getGroups
```

Description

Multiuser Server command; when `movie` is specified, gets the list of groups for the application in the current connection, including the predefined group @AllUsers. When `user` is specified, returns a list of current groups the user is a member of.

The server responds with a message that has the same object and command in the `#sender` parameter, and the same `#subject`, and `#contents` comprising a list of strings naming the groups.

Examples

This statement retrieves the current list of groups in this movie connection:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.\
getGroups", "anySubject")
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.movie.getGroups", #subject: "anySubject", #content: \
[#movieID: "theMovieName", #groups: ["@AllUsers", "@RedTeam", \
"@BlueTeam"]], #timeStamp: 79349843]
```

This statement returns a list of the current groups that the sender is a member of:

```
errCode = gMultiuserInstance.sendNetMessage("system.user.\
getGroups", "anySubject")
```

The returned list looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.user.getGroups", #subject: "anySubject", #content: \
[#userID: "userName", #groups: ["@AllUsers", "@Photographers", \
"@Designers"]], #timeStamp: 79349843]
```

See also

`sendNetMessage()`

getListOfAllMovies

This command is obsolete. Use `getMovies` instead.

getMovieCount

Syntax

```
system.server.getMovieCount
```

Description

Multiuser Server command; gets the number of different applications on the server. This is the number of different movies on the server, such as Checkers and Chess, not different instances of the same movie.

Example

This statement gets the number of different applications on the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.server.\
getMovieCount", "anySubject")
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.server.getMovieCount", #subject: "anySubject", #content: \
3, #timeStamp: 30214905]
```

getMovies

Syntax

```
system.server.getMovies
```

Description

Multiuser Server command; gets a list of the current movies connected to the server.

Example

This statement returns a list of all the movies currently connected to the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.server.\
getMovies", "anySubject")
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.server.getMovies", #subject: "anySubject", #content: \
["TankWars", "TicTacToe", "TechChat"], #timeStamp: 61726385]
```

See also

```
sendNetMessage()
```

getNetAddressCookie()

Syntax

```
gMultiuserInstance.getNetAddressCookie({encryptFlag}
{, whichIPAddress})
```

Description

Multiuser Server Lingo function; returns a network address cookie for the current machine. By default, the returned string is an encrypted string containing the local computer's IP address.

The `getNetAddressCookie()` function lets Lingo work with the local IP address without knowing the actual address. This is a security precaution to prevent movies being run by an end user behind a firewall from determining the address of the end user's computer.

The cookie's contents are "MacromediaSecretIPAddressCookie <encrypted IP address>". This value can be passed over the network to another computer, which can then use it as the server address for `connectToNetServer()`.

This is a typical scenario when you meet another user or users on a server and want to create a peer-to-peer connection without revealing your IP address. You can send an encrypted address to the other users and then wait for the other users by using `waitForNetConnection`.

If the optional *`encryptFlag`* parameter is set to 0, the returned string looks like a regular IP address, such as 123.45.67.1. This unencrypted address information is not available when movies are playing in Shockwave or if the `safePlayer` movie property is set to `TRUE`.

The second optional parameter lets you specify which IP address you want to return when the client machine is using multiple IP addresses. If you specify an index of 2, the second IP address on the machine will be returned.

Example

This statement sets the variable `myLocalAddress` to the third IP address on the client machine in unencrypted format:

```
myLocalAddress = gMultiuserInstance.getNetAddressCookie(0, 3)
```

See also

`connectToNetServer()`, `waitForNetConnection()`, `getUserIPAddress`

getNetErrorString()

Syntax

```
gMultiuserInstance.getNetErrorString(errorCodeNumber)
```

Description

Multiuser Server Lingo command; returns a string explaining the error code that is provided in place of *errorCodeNumber*. If the error code is invalid, this command returns a string representing an unknown error. Error codes are negative integers; 0 indicates that no error occurred.

Possible error codes and their strings are as follows:

Error code	Translated error string
0	No error
-2147216223	Unknown error
-2147216222	Invalid movie ID
-2147216221	Invalid user ID
-2147216220	Invalid password
-2147216219	Incoming data has been lost
-2147216218	Invalid server name
-2147216217	Server or movie is full; no connections are available
-2147216216	Bad parameter
-2147216215	No socket manager present
-2147216214	No current connection
-2147216213	No waiting message
-2147216212	Bad connection ID
-2147216211	Wrong number of parameters
-2147216210	Unknown internal error
-2147216209	Connection was refused
-2147216208	Message is too large or message buffer is full
-2147216207	Invalid message format
-2147216206	Invalid message length
-2147216205	Message is missing
-2147216204	Server initialization failed

Error code	Translated error string
-2147216203	Server send failed
-2147216202	Server close failed
-2147216201	Connection is a duplicate
-2147216200	Invalid number of message recipients
-2147216199	Invalid message recipient
-2147216198	Invalid message
-2147216197	Server internal error
-2147216196	Error joining group
-2147216195	Error leaving group
-2147216194	Invalid group name
-2147216193	Invalid server command
-2147216192	Not permitted with this user level
-2147216191	Error with database
-2147216190	Invalid server initialization file
-2147216189	Error writing database
-2147216188	Error reading database
-2147216187	User ID not found in database
-2147216186	Error adding new user
-2147216185	Database is locked
-2147216184	Data record is not unique
-2147216183	No current record
-2147216182	Record does not exist
-2147216181	Moved past beginning or end of database
-2147216180	Data not found
-2147216179	No current tag selected
-2147216178	No current database
-2147216177	Can't find configuration file
-2147216176	Current database record is not locked

Error code	Translated error string
-2147216175	Operation not allowed at current security level
-2147216174	Requested data or object was not found
-2147216173	Message content contains error information
-2147216172	Data concurrency error

Example

These statements attempt to connect to a multiuser server and display the error string in an alert if the attempt fails:

```
errCode = gMultiuserInstance.connectToNetServer([#logonInfo: \
[#userID: "Howard", #password: "mySecret", #movieID: \
"chatMovie"], #remoteAddress: "chatServer.myCompany.com", \
#remoteTCPPort: 1626, #localAddress: 123.23.45.678])
if errCode <> 0 then
   alert "Connection attempt failed!" & RETURN & \
   gMultiuserInstance.getNetErrorString(errCode)
end if
```

getNetMessage()

Syntax

```
gMultiuserInstance.getNetMessage()
```

Description

Multiuser Server Lingo function; returns the oldest waiting network message in the Xtra's message queue. The message is then deleted from the Xtra's memory space. These messages queue up internally for each connection, waiting for this function to be called. If no messages are waiting, this function returns an empty message. This function does not check the connection to the server; it only returns messages that are stored in the Xtra's message queue. It should be called from a network message handler.

If the connection has been opened as a text connection, the sender is set to `System` and the subject is `String`. The content is a string containing the data from the server. The Xtra returns all the data available, which might not terminate with the end of a text line. The Xtra includes and does not alter end-of-line characters sent by the server (CR or CRLF). If the server sends a zero byte, the Xtra drops the byte and does not pass it to Lingo.

The function's results are in the form of a property list with the following information. Symbols are used to access the list, retrieving an error code, the sender's ID, the subject, and the message contents.

#errorCode	Resulting error code: 0 if there is no error
#recipients	Users or groups the message was sent to
#senderID	String such as "Fred"
#subject	String subject of the message
#content	List of values, depending on the subject
#timeStamp	Server's time code stamp at the point of handling the message

Example

These statements retrieve a message from the queue:

```
netMsg = gMultiuserInstance.getNetMessage()
errCode = netMsg.errorCode
if (errCode = 0) then
   senderID = netMsg.senderID
   subject = netMsg.subject
   messageList = netMsg.content
   --handle the message
else
   --do error processing
   alert gMultiuserInstance.getNetErrorString(errCode)
end if
```

The contents of the retrieved message look like this:

```
[#errorCode: 0, #recipients: ["@AllUsers"], #senderID: "Fred", \
#subject: "ExampleSubject", #content: "ExampleContent", \
#timeStamp: 36437632]
```

See also

`getNetErrorString()`

getNetOutgoingBytes()

Syntax

```
gMultiuserInstance.getNetOutgoingBytes({userIDString})
```

Description

Multiuser Server Lingo function; returns the number of bytes currently in the outgoing message buffer, which is data waiting to be sent. This command is useful for determining if all data has been sent or if the client's outgoing buffer has room for large data chunks the user may want to send.

The total number of outgoing bytes possible is set by the `maxMessageSize` parameter of the `setNetBufferLimits` command. The default value of `maxMessageSize` is 16K.

The optional *`userIDString`* parameter is used to specify a particular user's buffer when hosting peer connections.

Example

The following code determines how much data is currently in the buffer for the current movie and sends a new message containing a large chunk of data (such as a cast member image) only if there is less than 500K in the buffer. The statements allow for sending pictures of up to 100K:

```
gMultiuserInstance.setNetBufferLimits(16 * 1024, 600 * 1024, 100)
totalWaiting = gMultiuserInstance.getNetOutgoingBytes
if totalWaiting < (500 * 1024) then
   gMultiuserInstance.sendNetMessage("@AllUsers", "New image", \
   member("Latest snapshot").picture)
end if
```

See also

`sendNetMessage()`, `setNetBufferLimits`

getNewGroupName

This command is obsolete. Use `createUniqueName` instead.

getNumberOfMembers

This command is obsolete. Use `getUserCount` instead.

getNumberWaitingNetMessages()

Syntax

```
gMultiuserInstance.getNumberWaitingNetMessages()
```

Description

Multiuser Server Lingo command; returns an integer representing the number of messages that have arrived in the Xtras queue and have not been read yet. This can be useful for determining whether you should call `checkNetMessages`.

Example

This statement gets the number of messages waiting to be processed:

```
numMessages = gMultiuserInstance.getNumberWaitingNetMessages()
```

See also

`checkNetMessages`

getPeerConnectionList()

Syntax

```
gMultiuserInstance.getPeerConnectionList()
```

Description

Multiuser Server Lingo function; returns a list of all the peer users connected to the host movie. An outgoing connection made with the `connectToNetServer()` command is not included in this list; only peer connections accepted with the `waitForNetConnection()` command are included.

The return value is a list containing the user IDs or an empty list, in the case of an error or no connections.

Example

This statement obtains the list of users connected to the host movie:

```
userList = gMultiuserInstance.getPeerConnectionList()
```

See also

`connectToNetServer()`, `waitForNetConnection()`

getReadableFieldList

This command is obsolete. Use `getAttributeNames` instead.

getRecordCount

This command is obsolete. Use `getAttributeNames` instead.

getRecords

This command is obsolete. Use `getAttribute` instead.

getServerTime

This command is obsolete. Use `getTime` instead.

getServerVersion

This command is obsolete. Use `getVersion` instead.

getTempPath()

Syntax

```
file("folderPathName").getTempPath( {[#extension: ".aaa", \
#create: TrueOrFalse]} )
```

Description

Multiuser Server server-side function; returns a file name that is unique in the specified folder. To create a file with the name, pass the optional parameters as a property list. The `#extension` property is the three-character string to be used as the file name extension. Always include a dot (.) with the file extension. Set the `#create` property to `TRUE` to create a file with the name.

The *`folderPathName`* can be an absolute path name or it may be relative to the server application. An error code of 1 is returned if the specified *`folderPathName`* is invalid.

Example

This server-side statement creates a file on the server computer with the name Tempimage.bmp:

```
file("C:\Images\Tempimage").getTempPath([#extension: ".bmp", \
 #create: 1])
```

getTime

Syntax

```
system.server.getTime
```

Description

Multiuser Server command; returns the current time from the server. The server responds with a message containing a string representing the time.

For synchronization between movies, it is better to examine the `#timeStamp` property returned in a message from the server. To get the current server `#timeStamp` value, just send a message to yourself.

Example

This statement shows a request for the server to send back the current time:

```
errCode = gMultiuserInstance.sendNetMessage("system.server.\
getTime", "anySubject")
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.\
server.getTime", #subject: "anySubject", #content: "2001/03/25 \
18:22:27", #timeStamp: 30203034]
```

See also

`sendNetMessage()`

getUserCount

Syntax

```
system.movie.getUserCount [movieName]
system.group.getUserCount [groupName]
system.DBAdmin.getUserCount
```

Description

Multiuser Server function; returns the number of users logged in to the given movie or group, or the number of `DBUser` objects in the server database. When `movie` is specified, the result is the same as calling `getUsers` for the group @AllUsers. If no movie is specified, the result is for the current movie. When `group` is specified, the result is for the given group. The reply message consists of a property list.

Examples

This statement returns the number of users logged into the current movie ID on the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.movie.\
getUserCount", "anySubject")
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.\
movie.getUserCount", #subject: "anySubject", #content: 17,
#timeStamp: 30231031]
```

This statement has the server report the number of members in the group @RedTeam:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.\
getUserCount", "anySubject", "@RedTeam")
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.\
group.getUserCount", #subject: "anySubject", #content: \
[#groupName: "@RedTeam", #numberMembers: 6], #timeStamp: 30234705]
```

To find the number of members in more than one group at a time, put the group names in a list in a statement similar to this:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.\
getUserCount", "anySubject", ["@RedTeam", "@BlueTeam", \
"@GreenTeam"])
```

The server responds with a separate message for each group.

See also

`sendNetMessage()`

getUserGroups

This command is obsolete. Use `getGroups` instead.

getUserIPAddress

This command is obsolete. Use `getAddress` instead.

getUserNames

Syntax

```
System.DBAdmin.GetUserNames {[#lowNum: firstNumber {, #highNum: \
lastNumber} ]}
```

Description

Multiuser server database function; returns a list of the user IDs in the server database.

The two optional parameters `#lowNum` and `#highNum` indicate the first and last `userID` records to get, respectively. This is useful when getting names out of a large database that would be unwieldy to list all at once. If `#highNum` isn't included, the first 100 users after `#lowNum` will be returned.

Example

This statement returns the list of user IDs in the server database between records 57 and 89:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBAdmin.\
getUserNames", "anySubject", [#lowNum: 57, #highNum: 89])
```

getUsers

Syntax

```
system.group.getUsers ["@groupName"]
```

Description

Multiuser Server command; gets the list of members for a particular group or groups.

If there is more than one group, the server responds with a separate message for each of the groups requested. Each response has the same subject as the original request. The reply message's contents are in the form of a property list with two properties, the first for the group name and the second for the members.

For large groups, the return message may be fairly long. Be certain that the server and Xtra message buffers are large enough to contain the list of all group members. See `setNetBufferLimits` and The Server Application.

Example

This statement has the server return a list of members in the @RedTeam group:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.\
getUsers", "anySubject", "@RedTeam")
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.\
group.getUsers", #subject: "anySubject", #content: [#groupName: \
"@RedTeam", #groupMembers: ["Mark", "Jane", "Chris"]], #timeStamp: \
98324982]
```

See also

`sendNetMessage()`

getVersion

Syntax

```
system.server.getVersion
```

Description

Multiuser Server command; this function requires no content. The server returns a message with the same subject and content containing information regarding the server application itself.

Possible values for `#platform` are currently `Macintosh` and `Windows`.

Example

The following statement shows the request for server information:

```
errCode = gMultiuserInstance.sendNetMessage("system.server.\
getVersion", "anySubject")
```

The reply message looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.\
server.getVersion", #subject: "anySubject", #content: [#vendor: \
"Macromedia", #version: "3.0", #platform: "Macintosh"], \
#timeStamp: 30196205]
```

See also

`sendNetMessage()`

getWriteableFieldList

This command is obsolete. Use `getAttributeNames` instead.

goToRecord

This command is obsolete. Use `getAttribute` and `setAttribute` instead.

handler()

Syntax

`whichFrame``.handler()`

Description

Multiuser Server server-side debugging function; returns as a symbol the name of the handler currently being run in the stack frame.

Example

These statements get the name of the handler running in frame 5 of the execution stack:

```
frame = theThread.frame(5)
theHandler = frame.handler()
return theHandler
```

isRecordDeleted

This command is obsolete. See “Database commands” in the “Multiuser Lingo by Feature” chapter instead.

join

Syntax

```
system.group.join ["@groupName"]
```

Description

Multiuser Server command; adds the sender to a group. If the group doesn't exist, it is automatically created. If the user already belongs to the group, no error occurs.

The server responds with a message for each group joined, matching the subject of the request message and containing the name of the group successfully joined.

Example

This statement adds the sender to the group @BeatleLovers:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.join", \
"anySubject", "@BeatleLovers")
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.group.join", #subject: "anySubject", #content: \
"@BeatleLovers", #timeStamp: 21765127]
```

This statement adds the sender to more than one group at the same time by putting the names of the groups in a list:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.join", \
"anySubject", ["@BeatleLovers", "@Photographers", "@Designers"])
```

The server sends a separate response for each group.

See also

`sendNetMessage()`

joinGroup

This command is obsolete. Use `join` instead.

language

Syntax

```
whichServer.language
```

Description

Multiuser Server server-side property; indicates the language version of the server. This property can be tested but not set. The return value is an integer.

This property can have the following values:

Integer	Language
0	English
1	French
2	German
9	Korean
10	Japanese

leave

Syntax

```
system.group.leave ["@groupName"]
```

Description

Multiuser Server command; removes the current user from the given group. If the user is not a member of the group, the server returns an error message.

Calling `leave` for the default group @AllUsers returns an error message.

The server responds with a message for each group that is left, matching the subject of the request message and containing the name of the group successfully left.

Example

This statement removes the sender from the group @RedTeam:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.leave", \
"anySubject", "@RedTeam")
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.\
group.leave", #subject: "anySubject", #content: "@RedTeam", \
#timeStamp: 762551131]
```

See also

```
sendNetMessage()
```

leaveGroup

This command is obsolete. Use `leave` instead.

line()

Syntax

whichFrame.line()

Description

Multiuser Server server-side debugging function; returns the script line number of the Lingo code executing in the given stack frame.

Example

These statements return the line number of the Lingo running in frame 17 of the execution stack:

```
whichFrame = 17
theLine = frame(whichFrame).line()
return theLine
```

list

Syntax

thread().list

Description

Multiuser Server server-side function; creates a list of the objects that exist in the current thread. This list reflects the threads in existence at the moment the list is tested. Thread objects are not automatically added to or deleted from this list as they come into and go out of existence. To update the list, test it again.

Example

These server-side statements put the thread object list into the variable `currentObjects` and return the value to the handler that called them:

```
currentObjects = thread().list
return currentObjects
```

lock()

Syntax

```
lock(whichObject)
```

Description

Multiuser Server server-side command; locks the given data object so that other threads may not make changes to the object until the `unlock()` command has been called. Lockable data objects include lists and property lists.

If another thread attempts to lock the same data object, it is blocked until the thread originally placing the lock explicitly unlocks the object via the `unlock()` command.

This command returns `1` if it places the first lock on the object, `2` if another thread has already locked the object.

Note: A list object will be automatically unlocked if a thread both locks the list and then resets the list to a new list value. The lock stays in effect only when individual values inside the list are edited.

Example

This `lock()` statement occurs in a separate thread and locks the list object `theList`, which has been defined earlier in the default server thread.

```
lock(theList)
```

See also

`unlock()`, `wait()`

locked

Syntax

```
file("whichFile").locked
```

Description

Multiuser Server server-side file property; gets or sets the locked state of a file. Set the `locked` property to `TRUE` to lock the file, and to `FALSE` to unlock the file. If a file is locked, it cannot be deleted with the `delete() (file)` command.

Example

This statement locks the file LongSpeech.txt on the server computer:

```
file(""C:\Text_files\LongSpeech.txt").locked = TRUE
```

See also

`delete() (file)`

lockRecord

This command is obsolete. See “Database commands” in the “Multiuser Lingo by Feature” chapter instead.

name (script)

Syntax

scriptObjectReference.name

Description

Multiuser Server server-side script property; indicates the name of the given script. This property can be tested and set.

Example

These statements call a custom handler added to the Dispatcher script that returns a script object reference and then return the name of the referenced script:

```
aScript = gDispatcher.findScriptByName( scriptName )
theName = aScript.name
return theName
```

name (thread)

Syntax

whichThread.name

Description

Multiuser Server server-side thread property; indicates the name of the given thread. This property can be tested and set.

Example

These statements return the name of the fourth thread in the thread list:

```
whichThread = 4
theThread = thread(whichThread)
return theThread.name
```

name (variable)

Syntax

whichVariable.name

Description

Multiuser Server server-side debugging property; returns the name of the given variable as a symbol.

Example

These statements return the name of the 12th variable in the variable list:

```
varNum = 12
variable = frame.variable(varNum)
variableName = variable.name
return variableName
```

new (thread)

Syntax

```
thread().new(newThreadName {, stackSize})
```

Description

Multiuser Server server-side command; creates a new thread with the name *newThreadName*. To specify a stack size, include the optional *stackSize* parameter. The stack size grows dynamically while Lingo instructions execute, as needed. For simple tasks, use a small stack size. For complex tasks, do not specify a stack size.

Example

This statement creates a new thread called `thread7` with a stack size of 128 bytes:

```
myThread = thread().new("thread7", 128)
```

notify()

Syntax

```
notify(whichObject)
```

Description

Multiuser Server server-side command; unblocks one thread that has been set to wait on the given object with the `wait()` command.

Example

This statement tells the thread that is waiting for the object `theList` to continue executing with the new value of `theList`:

```
notify(theList)
```

See also

`notifyAll()`

notifyAll()

Syntax

```
notifyAll(whichObject)
```

Description

Multiuser Server server-side command; unblocks all threads waiting on the given object.

Example

This statement tells all threads waiting for the object `theList` to continue executing with the new value of `theList`:

```
notifyAll(theList)
```

See also

`notifyAll()`

open()

Syntax

```
file("whichFile").open( {[#read: TrueOrFalse, #write:
TrueOrFalse, #create: TrueOrFalse]} )
```

Description

Multiuser Server server-side command; opens the specified file. The optional parameters are formatted as a property list. Set `#read` to `TRUE` (the default) to enable reading from the file. Set `#write` to `TRUE` to enable writing to the file. The default value for `#write` is `FALSE`. Set `#create` to `TRUE` to create the file if it does not yet exist. The default value for `#create` is `FALSE`.

Example

This server-side statement opens the file LongSpeech.txt with reading and writing enabled:

```
file("C:\Text_files\LongSpeech.txt").open([#read: 1, #write: 1, \
 #create: 0])
```

pack

This command is obsolete. See “Database commands” in the “Multiuser Lingo by Feature” chapter instead.

position

Syntax

```
file("whichFile").position
```

Description

Multiuser Server server-side property; indicates the relative position in the file of the next byte to be read or written. The first byte in a file is at position 0. This property can be tested and set.

Use this property to write data to specific locations within the file.

Example

This server-side statement sets the next write operation to occur at the 445th byte in the file Longspeech.txt:

```
file("C:\Text_files\Longspeech.txt").position = 445
```

produceValue()

Syntax

```
produceValue(whichValue)
```

Description

Multiuser Server server-side function; produces a value in a variable in the current thread for transfer to another thread. This function blocks the current thread until some other thread picks up the value using the `awaitValue()` function.

This function should be used for a single thread producing a value for a single other thread. To send values to multiple threads, use `lock()`, `wait()`, `notifyAll()`, and `unlock()`.

Example

These server-side statements set the variable `theValue` to the integer 96 and the pass the value to another thread with `produceValue()`:

```
theValue = 96
produceValue(theValue)
```

See also

`awaitValue()`, `lock()`, `wait()`, `notifyAll()`, `unlock()`

read()

Syntax

```
file("WhichTextFile").read( {bytesToRead} )
```

Description

Multiuser Server server-side function; reads a string from the specified text file. If no number of bytes is specified, all the bytes in the file are read. If a number of bytes is specified, only that number of bytes is read from the beginning of the file.

Example

This server-side statement reads 255 bytes from the text file Longspeech.txt and assigns the string to the variable `tempText`:

```
tempText = file("HardDrive:TextFiles:Longspeech.txt").read(255)
```

readValue()

Syntax

```
file("whichFile").readValue()
```

Description

Multiuser Server server-side function; reads the value written in the specified file. All core Lingo data types are supported (void, integer, string, symbol, floating-point number, list, propertyList, point, rect, color, date, media, picture, 3D vector, and 3D transform). Files containing other data types return `Void`.

Example

This server-side statement assigns the value in the file Sunset.tmp to the variable `tempImage`:

```
tempImage = file("HardDrive:Images:Sunset.tmp").readValue()
```

recallRecord

This command is obsolete. See “Database commands” in the “Multiuser Lingo by Feature” chapter instead.

reIndex

This command is obsolete. See “Database commands” in the “Multiuser Lingo by Feature” chapter instead.

removeUser()

Syntax

whichGroup.removeUser(*whichUser*)

Description

Multiuser Server server-side command; removes the given user from the specified group.

Example

This handler removes the user who has just logged on to the server from the group @AllUsers:

```
on userLogOn  (me, movie, group, user)
   voGroup = movie.serverGroup("@AllUsers")
   voGroup.removeUser(user)
end
```

See also

addUser()

rename()

Syntax

file("*whichfile*").rename("*newName*")

Description

Multiuser Server server-side function; renames the specified file to *newName*. The function returns a nonzero error code if it fails.

Example

This server-side statement renames the file Sunset.bmp to Horizon.bmp:

```
file("HardDrive:Images:Sunset.bmp").rename("Horizon.bmp"
```

resume()

Syntax

```
whichThread.resume()
```

Description

Multiuser Server server-side command; resumes normal execution of a thread that has been paused with a breakpoint.

Example

This statement resumes execution of the thread `thread7`:

```
thread("thread7").resume()
```

See also

`setBreakPoint()`

script (thread)

Syntax

```
whichFrame.script
```

Description

Multiuser Server server-side debugging function; returns a reference to the script object for the given stack frame.

Example

These statements return the script object reference for the script running in stack frame 4 of the thread `testThread`:

```
theThread = thread(testThread)
frameNum = 4
frame = theThread.frame(frameNum)
script = frame.script
```

See also

`variable()`, `variableCount`, `line()`, `frame() (thread)`, `frameCount (thread)`

seek

This command is obsolete. See “Database commands” in the “Multiuser Lingo by Feature” chapter instead.

selectDatabase

This command is obsolete. See “Database commands” in the “Multiuser Lingo by Feature” chapter instead.

selectTag

This command is obsolete. See “Database commands” in the “Multiuser Lingo by Feature” chapter instead.

sendMessage()

Syntax

```
whichServerMovie.sendMessage( string/listRecipient, \
"system.script.subject", messageContents {, errorCode \
{, protocolFlag {, stringSenderID}}} )

whichServerGroup.sendMessage( string/listRecipient, \
"system.script.subject", messageContents {, errorCode \
{, protocolFlag {, stringSenderID}}} )

whichServerUser.sendMessage( "system.script.subject", \
messageContents {, errorCode {, protocolFlag {, stringSenderID}}})
```

Description

Multiuser Server server-side command; sends a message from within a server-side script to the specified movie, group, or user.

When sending messages to a movie, the *protocolFlag*, *errorCode*, and *stringSenderID* are optional, but when used must be appear together and in the correct order. The *protocolFlag* is intended for future enhancements to the server and should be set to FALSE.

When sending a message to a user with the third syntax shown, the recipient parameter is omitted, since the specified user is the recipient.

The subject must begin with "system.script." to ensure that responses to the message are sent back to the server-side script.

This command is similar to sendNetMessage(), which is used in client movies.

Example

The following statement sends a message from the server-side script to the user Bob in the movie ChessMovie informing him that his opponent's rook has moved three squares forward. The error code is 0, the *protocolFlag* is FALSE and the *senderID* is the opponent whose name is John.

```
errCode = ChessMovie.sendMessage("Bob", "system.script.\
movePiece", ["Rook", 3, 0], 0, FALSE, "John")
```

See also

sendNetMessage()

sendNetMessage()

Syntax

```
gMultiuserInstance.sendNetMessage(string/ListRecipient, \
stringSubject, string/ListMessage)

gMultiuserInstance.sendNetMessage(propertyListMessage)
```

Description

Multiuser Server Lingo command; sends a message to one or more recipients in the current movie or a specified movie on the same server. The recipient may be a specific user, such as Sarah, a group, such as @TicTacToePlayers, or a list of strings containing individual user names or groups. To send a message to all users connected to the same movie, set the recipient to the default group @AllUsers. To send a message to a user in a different movie on the server, add the movie name, preceded by the @ symbol, to the end of the user name: for example, Jane@TechChat.

The message recipient can be a single string or a list of strings, in the case of multiple recipients. The subject must be a valid string. The contents may be a single Lingo value, such as a string, integer, floating-point number, point, rect, property, list, property list, color, date, 3D vector, 3D transform, or data such as the media of member property or the picture of member property.

If you are sending large data, such as a picture or the media of a cast member, make sure the buffer limits are large enough for both the client and the server. See setNetBufferLimits and Configuring the server.

The second possible syntax accepts a property list containing all the information for the message. The property list contains values for the symbols #errorCode, #recipient, #subject, and #content. (The #errorCode and #content values are optional.) This format is similar to the format for messages received by getNetMessage().

If you send a message to a group of which you are a member, you receive the message as well. If the server has problems sending the message to any recipients in a group or list, no error message is returned.

To send requests to the server, you must use a special syntax for the recipient, containing `System`, an object name, and the server command name. If the operation is successful, the server responds with a message that has the same subject, with an error code of 0.

To execute script handlers on the server, address the message to `system.script` and include the name of the handler to be executed as the subject of the message. The server-side script must have an `on incomingMessage` handler that parses the subject and then calls the handler named in the subject of the message.

If the connection is opened as a text connection, the recipient and subject are ignored. The message contents should all be simple Lingo values that will be converted to strings. Any carriage-return-line-feed (CRLF) combinations, such as those required by a POP server, must be explicitly added to the text sent by the Director movie.

Examples

These statements are some possible variations that can be used to send messages:

```
errCode = gMultiuserInstance.sendNetMessage("@AllUsers", \
"ChatMsg", gChatText)
errCode = gMultiuserInstance.sendNetMessage("BlackBeard", \
"MoveShipTo", point(shipX, shipY))
errCode = gMultiuserInstance.sendNetMessage("@groupRedTeam", \
"Help", "I am lost")
errCode = gMultiuserInstance.sendNetMessage(["Bill", "Joe", \
"Sue"] "JoinMe", "")
```

This statement provides its parameters in the form of a property list:

```
errCode = gMultiuserInstance.sendNetMessage([#recipients: "Bob", \
#subject: "Hello", #content: "How are you?"])
```

This statement sends a message to user Jane in a separate movie called TechChat:

```
errCode = gMultiuserInstance.sendNetMessage("Jane@TechChat", \
"ChatMsg", "What are you talking about?")
```

This statement sends the command `getUsers` to the server:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.\
getUsers", "anySubject", "@RedTeam")
```

This statement sends the command `getGroupCount` to the server as a request for information about a separate movie on the server called AdventureGame:

```
errCode = gMultiuserInstance.sendNetMessage ("system.movie.\
getGroupCount@AdventureGame", "anySubject")
```

This statement sends a message to the `on incomingMessage` handler in the movie's server-side script, which will parse the subject and call the `on updatePlayers` handler in the server-side script:

```
errCode = gMultiuserInstance.sendNetMessage ("system.script", \
"updatePlayers")
```

The following statement sends a message to the `on incomingMessage` handler in the server's Dispatcher script and causes it to immediately send the same message back to the client. The `case` section of the Dispatcher's `on incomingMessage` handler must be uncommented for the `ping` command to work.

```
errCode = gMultiuserInstance.sendNetMessage ("system.script", \
"System.Script.Admin.Ping")
```

This statement sends the RETR command to a text-based mail server to retrieve one piece of mail:

```
errCode = gMultiuserInstance.sendNetMessage("system", \
"anySubject", "RETR 1")
```

setAttribute

Syntax

```
system.group.setAttribute [#group: "@groupName", #attribute: \
[#attribute1: value1 {, #attribute2: value2} {, #lastUpdateTime: \
"timeString"}]]
system.DBUser.setAttribute [#userID: "userName", #attribute: \
[#attribute1: value1 {, #attribute2: value2} {, #lastUpdateTime: \
"timeString"}]]
system.DBPlayer.setAttribute [#userID: "userName", #application: \
"appName", #attribute: [#attribute1: value1 {, #attribute2: \
value2} {, #lastUpdateTime: "timeString"}]]
system.DBApplication.setAttribute [#application: "appName", \
#attribute: [#attribute1: value1 {, #attribute2: value2} {, \
#lastUpdateTime: "timeString"}]]
```

Description

Multiuser Server command; sets the value of an attribute for a group or a database object. To set a group attribute, supply the group name. To set a database object attribute, supply the `#userID` attribute, the `#application` attribute, or both. If both are supplied, the attribute is set for the `DBPlayer` object for the given user in the given application.

The `#lastUpdateTime` property is optional and lets you determine whether some other user has updated the attributes of the group since you last checked them with `getAttribute`. When you use `getAttribute`, the server responds with the values of the attributes you requested plus a `#lastUpdateTime` property, which indicates the moment in time when the server read the values of those attributes for the group you requested. The `#lastUpdateTime` property is a string containing the year, month, day, hour, minutes, seconds, and microseconds on the server. By sending this same string with your `setAttribute` command, you allow the server to check whether the attributes for the group have been updated since you last checked them.

If the server determines that the attributes for the group have been updated by someone else since you checked them, it responds with a value in the #errorCode property indicating a concurrency error. If no one else has updated the attributes since you checked them, the server responds with a new value in the #lastUpdateTime property for the group or database object, indicating that you have just updated the attributes.

Examples

This statement sets the attributes #teamLeader and #location for the group @RedTeam:

```
errCode = gMultiuserInstance.sendNetMessage("system.group.\
setAttribute", "anySubject", [#group: "@RedTeam", #attribute: \
[#teamLeader: "Mary", #location: "New York", #lastUpdateTime: \
"2001/07/26 15:26:33:123456"]])
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.group.setAttribute", #subject: "anySubject", #content: \
["@RedTeam": [#lastUpdateTime: "2001/08/25 18:55:33.132456"]], \
#timeStamp: 32189685]
```

This statement sets the attribute #favoriteColor for the DBUser object Bob:

```
errCode = gMultiuserInstance.sendNetMessage("system.DBUser.\
setAttribute", "anySubject", [#userID: "Bob", #attribute: \
[#favoriteColor: "Blue", #lastUpdateTime: "2001/07/26 \
15:26:33:123456"]])
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.DBUser.setAttribute", #subject: "anySubject", #content: \
["Bob": [#lastUpdateTime: "2001/08/25 13:28:22.123456"]], \
#timeStamp: 184472070]
```

The following statement sets the attributes #accountBalance and #cardHand for the DBPlayer object of the user Bob in the movie Poker. The #accountBalance and #cardHand attributes have already been declared with declareAttribute.

```
errCode = gMultiuserInstance.sendNetMessage("system.DBPlayer.\
setAttribute", "anySubject", [#userID: "Bob", #application: \
"Poker", #attribute: [#accountBalance: 3500, #cardHand: "Royal \
Flush", #lastUpdateTime: "2001/07/26 15:26:33:123456"]])
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: "system.\
DBPlayer.setAttribute", #subject: "anySubject", #content: ["Bob": \
[#lastUpdateTime: "2001/08/26 12:43:33.647483"]], #timeStamp: \
6461476]
```

The following statement sets the attribute `#highScore` for the DBApplication object Basketball. The `#highScore` attribute has already been declared with `declareAttribute`.

```
errCode =
gMultiuserInstance.sendNetMessage("system.DBApplication.\
setAttribute", "anySubject", [#application: "Basketball", \
#attribute: [#highScore: 1352, #lastUpdateTime: "2001/07/26 \
15:26:33:123456"]])
```

The server's response looks like this:

```
[#errorCode: 0, #recipients: ["userName"], #senderID: \
"system.DBApplication.setAttribute", #subject: "anySubject", \
#content: ["Basketball": [#lastUpdateTime: "2001/08/25 \
14:16:14.852673"]], #timeStamp: 187351603]
```

See also

`getAttribute`, `sendNetMessage()`

setBreakPoint()

Syntax

`scriptObject``.setBreakPoint(`*`#handlerName, whichLineNumberInHandler, enableTrueOrFalse`*`)`

Description

Multiuser Server server-side debugging command; enables or disables a breakpoint at the given line number in the given handler in the specified script object. Replace *`enableTrueOrFalse`* with `TRUE (1)` to turn on the breakpoint, or with `FALSE (0)` to turn it off.

In order for a breakpoint to be triggered, it must be it must be set in a handler that is running within a thread created with the `new (thread)` command. Breakpoints cannot be set in the server's default thread.

See also

`new (thread)`

setFields

This command is obsolete. Use `setAttribute` instead.

setNetBufferLimits

Syntax

```
gMultiuserInstance.setNetBufferLimits(tcpipReadSize, \
maxMessageSize, maxIncomingUnreadMessages)
```

Description

Multiuser Server Lingo command; sets the size of internal buffers and sets limits on the number of messages that the Xtra can queue in memory. The command is not normally needed, but it can be used to fine-tune memory management. If a movie sends or receives particularly large messages, such as picture data, these values should be set to accommodate them. Specify all the values in bytes.

In order to take effect, this command must be issued immediately after a Multiuser Xtra instance is created. It will not take effect once a server connection has been established.

tcpipReadSize controls the maximum amount of data read each time the Xtra takes data from the low-level TCP/IP data stream. This may be altered to tweak performance, particularly when receiving large messages. The default is 16K bytes.

maxMessageSize controls the buffers used to store parts of messages sent and received from another system. The default value is 16K bytes. This must be larger than the largest message sent or received.

maxIncomingUnreadMsgs sets a limit on the number of unread incoming messages that can be accumulated. The default is 100.

Example

This statement sets the maxMessageSize to 350K bytes to accommodate sending image data back and forth:

```
errCode = gMultiuserInstance.setNetBufferLimits(16 * 1024, \
350 * 1024, 100)
```

setNetMessageHandler

Syntax

```
gMultiuserInstance.setNetMessageHandler(#handlerSymbol, \
handlerObject, {subject, {sender}} {, integerPassMessage})
```

Description

Multiuser Server Lingo command; sets the callback handler that is invoked when messages arrive. This should be set before connecting with `connectToNetServer()` in order to capture the response message.

You can set as many callbacks as you need. It is much more efficient to have multiple callbacks that react to specific types of incoming messages than to have one callback that filters messages and then calls another handler.

This ability is key for creating flexible and effective experiences. You might have a scenario that uses avatar objects to represent other users, and set a callback to each specific object. In that way, any message related to that user is routed automatically when dealing with the object. The avatar object itself may even have multiple callbacks for messages with different subjects.

#handlerSymbol is a symbol that represents the handler.

handlerObject represents the object where the handler is located. This may be a behavior instance, a parent script, or any Lingo value. If it is a script or behavior instance, the handler associated with it is called. If it is a Lingo value, the handler must be in a global script and the Lingo value is passed as the first parameter to that script.

subject and **sender** are optional parameters that route messages with particular subjects, senders, or both to a handler. This lets avatar scripts receive all the messages from a sender. To route all messages from one sender to a handler, set the subject to 0 and specify the sender's ID.

IntegerPassMessage is an optional parameter that indicates whether Lingo should pass the contents of the incoming message directly to the specified handler as an argument. This allows your message handlers to be simpler because they can omit Lingo that serves the purpose of getting the message out of the message queue.

To disable message notification for a given type of message, call `setNetMessageHandler()` again and set the handler information to 0.

Examples

This statement sets the handler `on MyNetMessageHandler` located in the script cast member CallBackScript as the generic message callback handler (no specific subject or sender is specified):

```
errCode = gMultiuserInstance.setNetMessageHandler\
(#MyNetMessageHandler, script "CallBackScript")
```

The message handler might look like this:

```
on myNetMessageHandler
   global gMultiuserInstance
   newMessage = gMultiuserInstance.getNetMessage()
   member("messageOutput").text = newMessage
   if newMessage.errorCode <> 0 then
      alert "Incoming message contained an error."
   end if
end
```

The following statement sets the handler `on BobHandler` in the script cast member Callbacks as the message callback handler for messages with any subject received from user Bob. The last parameter of 1 tells Lingo to pass the messages to the handler as arguments.

```
errCode = gMultiuserInstance.setNetMessageHandler(#BobHandler, \
script "Callbacks", "", "Bob", 1)
```

The simplified message handler without Lingo for getting the messages out of the message queue might look like the following. Note the `message` argument that appears after the handler name.

```
on BobHandler me, message
   member("messageOutput").text = message
   if message.errorCode <> 0 then
      alert "Incoming message contained an error."
   end if
end
```

This statement sets the handler `on RedCarMsgHandler` in the script object `me` as the message callback handler for messages with a subject of setCarPosMsg and a sender of Fred:

```
errCode = gMultiuserInstance.setNetMessageHandler\
(#RedCarMsgHandler, me, "setCarPosMsg", "Fred")
```

This statement disables the handler declared above by specifying 0 in place of the handler symbol:

```
errCode = gMultiuserInstance.setNetMessageHandler(0, me, \
"setCarPosMsg", "Fred")
```

See also

`connectToNetServer()`

size

Syntax

```
file("whichFile").size
```

Description

Multiuser Server server-side function; returns the size of the file in bytes.

Example

This server-side statement assigns the size in bytes of the file Sunset.bmp to the variable `fileSize`:

```
fileSize = file("HardDrive:Images:Sunset.bmp").size
```

skip

This command is obsolete. See "Database commands" in the "Multiuser Lingo by Feature" chapter instead.

sleep()

Syntax

```
sleep(timeInMilliseconds)
```

Description

Multiuser Server server-side command; pauses the current thread (the thread that issues the `sleep()` command) to allow other threads to run. The thread is placed in sleep mode for the given number of milliseconds. If there is no *timeInMilliseconds* parameter, the thread sleeps indefinitely. To delete a permanently sleeping thread, use the `forget()` command on the thread and then set it to 0.

Example

This statement sleeps the thread that issues the statement for 1.5 seconds:

```
sleep(1500)
```

See also

`forget (thread)`

stackLevel

Syntax

whichThread.stackLevel

Description

Multiuser Server server-side debugging function; returns an integer indicating the number of levels of nested handler calls that are currently pending in the given thread.

Example

This statement sets the variable theLevel to the stackLevel number of the thread testThread:

```
theLevel = testThread.stackLevel
```

See also

stackSize, stepInto(), stepOver(), script (thread), setBreakPoint()

stackSize

Syntax

whichThread.stackSize

Description

Multiuser Server server-side debugging function; returns the current stack size of the thread in bytes. The stack size indicates how much memory is being used by the thread.

Example

This statement sets the variable sizeInBytes to the current stack size of the thread testThread:

```
sizeInBytes = testThread.stackSize
```

See also

stackLevel, stepInto(), stepOver(), script (thread), setBreakPoint()

status

Syntax

whichThread.status

Description

Multiuser Server server-side function; returns a symbol indicating the current status of the given thread. The possible `status` values are as follows:

Symbol	Definition
#awaitValue	Thread is awaiting a value from another thread. See produceValue().
#breakPoint	Thread is stopped at a breakpoint. See setBreakPoint().
#call	Thread has executed a call() command. See call().
#error	An error has occurred in the thread.
#lock	Thread is locked. See lock().
#produceValue	Thread has produced a value and no other thread has called awaitValue(). See awaitValue() and produceValue().
#resume	Thread has resumed from waiting. See resume().
#run	Thread is running.
#sleep	Thread is sleeping. See `sleep()`.
#stepInto	Thread is stepping through script instructions, including handler calls. See stepInto().
#stepOver	Thread is stepping through script instructions, excluding handler calls. See stepOver().

Example

This statement sets the variable `theStatus` to the current status of the thread `testThread`:

```
theStatus = testThread.status
```

See also

`sleep()`, `resume()`, `lock()`, `call()`, `produceValue()`, `setBreakPoint()`, `stepInto()`, `stepOver()`

stepInto()

Syntax

whichThread.stepInto()

Description

Multiuser Server server-side debugging command; executes the next Lingo instruction in the given thread. StepInto() branches into nested handler calls that occur in the current script.

Example

This statement steps into the script running in the thread testThread, including nested handler calls:

```
testThread.stepInto()
```

See also

stepOver(), setBreakPoint()

stepOver()

Syntax

whichThread.stepOver()

Description

Multiuser Server server-side debugging command; executes the next Lingo instruction in the given thread. StepOver() jumps over nested handler calls in the current script instead of branching into them.

Example

This statement steps into the script running in the thread testThread, excluding nested handler calls:

```
testThread.stepOver()
```

See also

stepInto(), setBreakPoint()

sweep()

Syntax

```
sweep().free()
sweep().status()
```

Description

Multiuser Server server-side command; deletes orphaned objects. Objects such as lists, script objects, and so on. can become orphaned if they contain mutual references to one another. If list A contains a reference to list B and list B contains a reference to list A, simply deleting list A will not work, since a reference to it still exists in list B.

`Sweep().status()` returns a list of all objects in the current thread that are currently orphans. It is a good idea during debugging to check whether your code is creating orphans unintentionally. Call `sweep().status()` during idle time only every few seconds, since visiting every object and checking its status is CPU intensive.

`Sweep().free()` deletes these orphaned objects and free the memory they were consuming.

Example

These statements test for the presence of orphaned objects in the current thread and deletes any that are found:

```
if sweep().status() <> [] then
   sweep().free()
end if
```

See also

`stackSize`

thread()

Syntax

`thread(`*`threadNameOrNumber`*`)`

Description

Multiuser Server server-side keyword; references a thread by its name or relative number in the thread list.

Example

These server-side statements display the name and status of each thread in the thread list in the server's console window:

```
repeat with i = 1 to n
   theThread = thread(i)
   put "Thread " & theThread.name & " status =" & theThread.status
end repeat
```

See also

`status`

type (variable)

Syntax

`whichVariable``.type`

Description

Multiuser Server server-side debugging function; returns a symbol indicating the variable's type. Variables can be of type `#param`, `#local`, `#global`, or `#property`.

Example

These statements set the variable `variableType` to the type symbol of the third variable in the variable list of frame 17 of thread `testThread`:

```
thread = thread(testThread)
frameNum = 17
frame = thread.frame(frameNum)
varNum = 3
variable = frame.variable(varNum)
variableType = variable.type
```

type (file)

Syntax

```
file("whichFile").type
```

Description

Multiuser Server server-side file property; on the Macintosh, gets or sets the 4-byte type code of the file.

Example

This server-side statement gets the `type` code of the Director file Testmovie.dir on the server and displays it in the server's console window:

```
put file("Hard Drive:Multiuser_Server:Testmovie.dir").type
-- "MV08"
```

See also

`creator`

unlock()

Syntax

```
unlock(whichObject)
```

Description

Multiuser Server server-side command; Unlocks the given data object so that other threads may have access to it with the `lock()` command. Lockable data objects include lists and property lists. Other threads that issue a `lock()` command are blocked until `unlock()` is called for the object by the thread that originally locked it.

Note: A list object will be automatically unlocked if a thread both locks the list and then resets the list to a new list value. The lock stays in effect only when individual values inside the list are edited.

Example

This statement unlocks the property list called `sharedData`:

```
unlock(sharedData)
```

See also

`lock()`, `wait()`

unlockRecord

This command is obsolete. See "Database commands" in the "Multiuser Lingo by Feature" chapter instead.

value (variable)

Syntax

whichVariable.value

Description

Multiuser Server server-side debugging property; indicates the current value of the given variable. This property can be tested and set.

Example

These statements set the variable `variableValue` to the value of the seventh variable in the variable list of frame 25 of thread `testThread`:

```
thread = thread(testThread)
frameNum = 25
frame = thread.frame(frameNum)
varNum = 7
variable = frame.variable(varNum)
variableValue = variable.value
```

variable()

Syntax

whichFrame.variable(*variableNumber*)

Description

Multiuser Server server-side debugging function; returns a reference to the variable at the given number in the variable list.

Example

This statement makes the variable `whichVariable` a reference to the seventh variable in the variable list of frame 23 of the thread `testThread`:

```
thread = thread(testThread)
frameNum = 23
frame = thread.frame(frameNum)
variableNum = 7
whichVariable = frame.variable(variableNum)
```

variableCount

Syntax

whichFrame.variableCount

Description

Multiuser Server server-side debugging function; returns the number of variables in the current stack frame. This number includes variables of all types: #local, #global, #property and #param.

Example

This statement sets the variable numVariables to the number of variables in the variable list of frame 23 of the thread testThread:

```
thread = thread(testThread)
frameNum = 23
frame = thread.frame(frameNum)
numVariables = frame.variableCount
```

volumeInfo

Syntax

file("*fileName*").volumeInfo

Description

Multiuser Server server-side function; returns information about the volume containing the specified file as a property list with the format [#blockSize: n, #freeBlocks: m]. The #blockSize property is the size of the minimum disk space allocation. The #freeBlocks property indicates the number of free blocks on the volume.

Example

This server-side statement displays the volumeInfo of the volume HardDrive in the server's console window:

```
put file("HardDrive:Images:Sunset.bmp").volumeInfo
```

wait()

Syntax

```
wait(whichObject)
```

Description

Multiuser Server server-side command; blocks the current thread from executing until another thread issues the `notify()` command for the given data object. `Wait()` and `notify()` are used to share data between threads while preventing more than one thread from reading or writing to the object at the same time.

Example

This statement causes the current thread to stop executing until another thread issues a notify command on the list variable `sharedList`:

```
wait(sharedList)
```

See also

`notify()`, `notifyAll()`, `lock()`, `unlock()`

waitForNetConnection()

Syntax

```
gMultiuserInstance.waitForNetConnection(userIDString, \
localTCPPortNumber {, maxNumberOfConnections, \
{encryptionKeyString}})

gMultiuserInstance.waitForNetConnection([#userID: userIDString \
{, #maxConnections: maxNumberOfConnections {, #localAddress: \
localIPAddress} {, #localTCPPort: TCPPortNumber} \
{, #encryptionKey: encryptionKeyString}]
```

Description

Multiuser Server Lingo function; listens for incoming peer-to-peer connections from other computers.

This function takes arguments in two formats. The first format contains the following parameters:

userIDString represents the log-on name of the user acting as the host and waiting for connections.

portNumber represents the Internet port the connecting system will contact. Multiuser servers should use port 1626 by default. Generic inbound TCP connection requests are not supported.

maxNumberOfConnections represents an optional parameter for the maximum number of possible connections to the host. Up to 16 peer connections are allowed.

encryptionKeyString is an optional parameter that supplies a key string to decode log-on information from other systems. If this parameter is used, the other systems must use an identical encryption key string when they connect using `connectToNetServer`.

The second format includes a property list containing the following optional parameters in addition to those listed above:

#localAddress indicates the local IP address of the host computer. Use this parameter on machines with multiple local IP addresses. You can obtain the IP address of the machine with the `getNetAddressCookie()` function.

The connecting computer uses `connectToNetServer` and appears to be connecting to a normal multiuser server, but it is actually connecting to a peer computer that has issued the `waitForNetConnection` function. In connections of this type, no server commands are available. The computer that calls `waitForNetConnection` must do so before a peer calls `connectToNetServer`.

Because waiting for an incoming connection can take a long time, this function returns an error code immediately. An error code of 0 indicates that the Xtra has begun listening successfully. A message is sent back from the Xtra (and the message handler called) when the incoming connection is actually established. For this to work, you must set a handler with `setNetMessageHandler` before calling `waitForNetConnection`. The message handler called should return `TRUE` if the host movie wants to accept the connection, or `FALSE` if it is to be rejected. The returned message is a list that contains the following items:

#errorCode	Resulting error code: 0 if there is no error
#senderID	System
#subject	WaitForNetConnection
#content	Remote user information in a property list containing `#userID`, `#password`, and `#movieID`

Up to 16 peer-to-peer connections can be established with each Xtra instance using `waitForNetConnection`. Multiple calls to `waitForNetConnection` cannot be made using the same port number, because the Xtra instances cannot all wait on the same port number.

After the function is called, you cannot turn off the wait for additional connections, except by deleting the Xtra instance. You can specify a limit to the number of connections when calling `waitForNetConnection`, or use the incoming message handler to return `TRUE` or `FALSE` to allow or reject the incoming connection attempts.

Do not make outgoing server connections using an Xtra instance that has called `waitForNetConnection`.

Examples

These statements show a range of typical calls to set the movie to receive connections:

```
errCode = gMultiuserInstance.waitForNetConnection("Fred", 1626)
errCode = gMultiuserInstance.waitForNetConnection("Mark", 1626, \
16, "Queen3ToRook2")
errCode = gMultiuserInstance.waitForNetConnection("Joe", 1626, \
2, "RogerWilco")
```

See also

`setNetMessageHandler`, `getNetAddressCookie()`, `connectToNetServer()`

write()

Syntax

`file("whichFile").write(stringValue)`

Description

Multiuser Server server-side function; Writes the specified string to the given file. The file is created if necessary. This function returns a nonzero error code if it fails.

Example

This server-side statement writes the string “This is some new text” to the file Shortspeech.txt:

```
file("HardDrive:Shortspeech.txt").write("This is some new text")
```

See also

`writeValue()`

writeValue()

Syntax

```
file("whichFile").writeValue(whatValue)
```

Description

Multiuser Server server-side function; writes a Lingo value to a file. All core lingo types are supported (void, integer, string, symbol, floating-point number, list, propertyList, point, rect, color, date, media, picture, 3D vector, and 3D transform). All other values are written as `VOID`. This function returns `0` if the operation succeeded, `1` if it failed. The file will be created if it does not yet exist.

Example

These server-side statements write an image value to the file Tempimage.tmp:

```
-- get the image data from the content of a message sent by
-- the movie
theImage = fullMsg.content

-- write value to the file
file("HardDrive:Images:Tempimage.tmp").writeValue(theImage)
```

See also

`write()`